Full-Time Data Compression: An ADT for Database Performance

Goetz Graefe
Leonard D. Shapiro

CU-CS-503-90

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Full-Time Data Compression: An ADT for Database Performance

Goetz Graefe, Leonard D. Shapiro

CU-CS-503-90   December 1990

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430

(303) 492-7514
(303) 492-2844 Fax
graefe@boulder.colorado.edu

Full-Time Data Compression: An ADT for Database Performance

Goetz Graefe
University of Colorado
Boulder, CO 80309-0430

Leonard D. Shapiro
Portland State University and OACIS
Portland, OR 97207-0751

Abstract

Data compression is a classic technique for saving disk space and network bandwidth, and providing some level of security. It comes with a classic tradeoff: the CPU time required to compress and decompress the data whenever it is used, vs. the savings in disk space and network bandwidth, and perhaps security costs.

We propose a much wider use of compression in database systems, namely keeping data compressed almost all the time, until it is displayed to the end user. Database systems which support abstract data types (ADTs) can easily implement our approach. This approach will result in significantly better performance for two reasons: savings in compression and decompression time, and savings in workspace at several levels since compressed data will be used in place of decompressed data. It will also allow the database to assume the task of translating between symbols and their explanations, thereby freeing end users, applications programmers, or database administrators from this tedious burden. Finally, since our approach makes compression affordable in a database setting, it makes the traditional benefits of compression more attractive – added security and, in parallel and distributed systems, savings in network bandwidth.

1. Introduction

1.1. The Classic Use of Data Compression

For both data storage and data transmission, it is common to use data compression techniques to save disk storage and network bandwidth. This is done in three steps. First, data to be stored or transmitted is compressed, then it is stored or transmitted. Finally, when the data is to be used (i.e. brought into main memory for an application, or delivered to the network destination) it is decompressed back to its original form. We denote the first and third of these steps by the term *de/compression*. De/compression techniques range from the simplest (e.g. truncating trailing blanks), which may yield little benefit but are simple to implement, to the most complex, which can yield compression by a factor of 3 or 4 [2]. De/compression requires significant amounts of memory and CPU time, depending in part on the complexity of the compression algorithm used.

The three-step scheme traditionally used for compression requires significant memory and time to de/compress the data every time it is used. Since database systems may be CPU bound, compression is not a clear win in all situations and for all data.

1.2. Full-Time Data Compression

In order to avoid the costs of de/compression every time data is used, we suggest that in many situations database data, even a primary key, can be kept compressed "full-time". By "full-time compression" we mean that the data is uncompressed only when it is to be displayed to an end user, or when absolutely necessary for internal processing, e.g. for arithmetic operations on compressed fields. We shall see below that internal processing rarely requires decompression if data design is done correctly. We have outlined some of the ideas presented here in a preliminary report [11]; the contributions of this report are the use of abstract data types to integrate compression into existing relational systems, and a further analysis of when and how to exploit compression in physical database design.

We present some examples of query processing for full-time compressed data. Consider an attribute containing social security numbers (a prime candidate for full-time compression). Here are a few sample queries, assuming that social security number is stored in the database in compressed form:

(a) Retrieve the employee with social security number 123-45-6789: The given constant is compressed, and a search made for the compressed value in the employee relation. The employee relation might even be indexed and clustered on the compressed social security number.

(b) Join two relations on social security number: Because compressed values are unique, the usual join algorithms obtain the correct result if the same compression technique was used for social security number in both relations. Even sort-merge join will get the correct result, although it will use a different ordering (of the compressed data) to find matching tuples.

(c) Find the names of persons with social security numbers with middle entry -45-: This query requires internal decompression of the social security numbers before processing, although some special methods exist to avoid decompression even in this case [9].

Clearly not all attributes are candidates for compression, and even social security numbers should not be compressed if there is a predominance of queries of type (c). However, we claim that with the use of an abstract data type mechanism in data managers, full-time compression will be simple to implement. Full-time compression will also yield significant cost savings compared to the three-step compression described above because:

(a) CPU time is much less. Since data typically stays compressed until it reaches the end user, less decompression is needed.

(b) Storage space is saved at all levels of the storage hierarchy, from CPU caches space to I/O buffers in real memory to disk space. This is because the compressed data is much smaller. For example, testing two addresses for a match will require more time than checking compressed versions. After compression a relation may fit entirely in main memory, making joins and other operators much more efficient. This matter is discussed more fully in Section 7.

For similar reasons, full-time compression is also desirable compared to not using compression at all.

1.3. Overview

In the next section, we review related work. In Section 3 we discuss compression techniques, both those which are relevant to database compression, and more general techniques. In Section 4 we describe how to use ADTs to implement full-time compression. Section 5 discusses query processing of compressed data. In Section 6 we consider which attributes should be chosen to hold compressed data, and which compression techniques to use for which data. Section 7 shows how the functionality of database systems can be improved by using compressed data, and Section 8 discusses performance implications. Section 9 contains our summary and conclusions.

2. Related Work

The idea of using compression in database systems goes back many years. Alsberg [1] surveys dictionary-based techniques (see Section 3 for a definitions of these compression techniques), and Mulford and Ridall discuss techniques for general text compression [17].

Alsberg [1] suggests using order-preserving compression methods where feasible (e.g. for dates) and doing retrievals, including those on range predicates, by compressing the searched-for data instead of decompressing the searched data. We suggest even more use of compressed data, and we propose using ADTs to simplify implementation.

Severance [21] provides a more recent tutorial and literature survey of a wide variety of data compression methods. IMS has a compression option, whose performance is discussed in [5], which compresses and decompresses entire records at a time.

Other researchers have investigated compression and access schemes for scientific databases with very many constants (e.g., zeroes) [7, 15, 18] and considered a very special operation on compressed data (matrix transposition) [25].

For indices, many database systems use prefix- and postfix-truncation to save space and increase the fan-out of nodes, e.g. Starburst [12].

3. Compression Techniques

In this section, we discuss the specific techniques available for compression in database systems. We also discuss some modern and very effective compression techniques and show why they may not be applicable in the database setting.

We will use the following example:

3.1. Compression in Database Systems

The simplest and most common method for database compression is null suppression [20]. Null suppression replaces null field entries with some very short code, e.g. a bit which when on denotes a null field. Attribute 4 is a candidate for null suppression in Figure 1. Trailing blanks and leading zeroes are also commonly truncated in commercial systems.

We have referred to two surveys of the many other techniques used for database compression [1, 21]. Most of these are more sophisticated variants of two simple techniques, namely *dictionary coding* and

Attribute	Description	Size in Bytes
1. Empname	Employee Name	30
2. Socsecno	Social Security Number	10
3. Deptname	Department Name	25
4. Spousnam	Spouse Name	30

Figure 1. Example File Employee.

Huffman coding.

In dictionary coding, each entry in an attribute is replaced by some fixed-length symbol from a dictionary [2, 26, 27]. For example, attribute 3 in Figure 1 is a prime candidate for dictionary encoding, because there are likely to be few department names in the database. It is common in current database applications to use such codes (e.g. deptid) instead of full names. This can be done through views, for example. In Section 7 we discuss why full-time compression is a more desirable approach.

In Huffman Coding, each character is replaced by a string of bits [8, 13, 14]. Bit strings vary in length, and longer strings corresponding to less commonly observed characters. Attribute 1, Employee Name, in our example is a candidate for Huffman coding.

Dictionary compression and Huffman coding demonstrate some important properties of compression techniques:

(a) A compression technique in general requires both an *algorithm* (dictionary lookup, or walking a Huffman tree) and a *model* (the dictionary itself, or the Huffman tree). We refer to the combination of algorithm and model as a *compression structure*.

(b) The most appropriate compression structure for an attribute depends on the data in the attribute. For example, using dictionary coding for attribute 1, Employee Name, in our example, would require a prohibitively large dictionary, while Huffman Coding on attribute 3, Department Name, would be much less efficient than dictionary encoding.

Huffman coding yields, on the average, about a factor of 2 compression on typical English data, while dictionary compression can yield factors of 1 to 10 or more, depending on the data in the attribute. Variants of both these methods can achieve significantly greater compression, but require correspondingly greater resources.

3.2. Other Compression Techniques

In the past ten years, modern compression techniques have been developed which yield compression factors of 4 for general English text [2] as opposed to the factor of 2 more typical of previous algorithms. These improvements have been made through adaptive techniques which build large models of the data being compressed and use those models as a dictionary of text fragments to be used in further compression. Due to the size of such models, however, it is not clear that these techniques are applicable to database systems, since the time to retrieve such large models could significantly increase the time required to retrieve and decompress small amounts of data.

4. Implementing Full-Time Compression

In this section we discuss how to implement full-time data compression using ADTs. We remind the reader that any compression algorithms we use must be *lossless*, i.e. compressed versions of unequal data items must be unequal.

4.1. What is an ADT?

An ADT, or abstract data type, is a mechanism in database systems to allow a user or DBA to introduce new data types, e.g. "box". Some mechanism for defining ADTs is common in or even the basis of object-oriented systems and they are common in extended relational systems [4, 12, 24]. We will follow the model for ADTs given in [23]. In order to implement such an ADT the user must first define input and output routines for the ADT, to translate the external (character string) representation of the data type to the internal (in our case, compressed) version. For the ADT "box", the external representation might be the left and right corners, e.g. the unit square would be represented externally as:

((0,0) , (1,1)).

Along with input and output routines, the user must define all operators for this new data type. For example, a typical unary operator for box would be "nonempty", and a binary operator over two boxes would be "overlaps". We will consider input/output routines and operators, for full-time compressed data, in the next two sections.

We denote the new data type for full-time compressed data by CD.

4.2. Input and Output Routines for the ADT CD

Simply put, the input routine is compression and the output routine is decompression. The only complexity is that the compression structure to be used for de/compression depends on the attribute. For example, the input (compression) function for an attribute containing Part numbers is different for one containing manufacturers. If a given ADT facility does not allow the input and output function to depend on the attribute, it may be necessary to define a different ADT for each attribute, or at least for each domain. Henceforth we will assume that the input/output routines can depend on the attribute.

Input and output routines handle much of the work of implementing compression. For example, if we denote by XX, YY the compressed forms of the name "David Aucsmith" and the social security number 123-45-6789, the ADT mechanism will translate queries such as the first query in Figure 2 into the second form in Figure 2.

Similarly, queries using update, display or print will be modified to use the appropriate representation and translation.

4.3. Operators

We now turn to operators for data of type CD. We will redefine all the traditional operators on data types, namely the three categories in/equality, greater/less, and arithmetic (+, ×, etc.)

Suppose the attribute "data1" is of type CD. For a join precicate "data1 = data2", we define the following three cases:

(a) "data2" is also CD, using the same compression structure as "data1": In this case the equality operator is the usual one, since compression structures are lossless.

```
INSERT INTO EMP (Empname, Socsecno)
VALUES ('David Aucsmith', '123-45-6789')

INSERT INTO EMP (Empname, Socsecno)
VALUES (XX, YY)
```

Figure 2. Translation of Requests by ADT CD.

(b) "data2" is not of type CD: In this case, either the uncompressed attribute can be compressed, or vice versa. The choice might be made on the basis of whether it is easier to compress or decompress for the relevant compression structure.

(c) "data2" is CD using a different compression structures than "data1": Here there are three choices: (1) decompress "data1", then recompress it using the compression structure for "data2"; (2) use (1) but reverse the roles of "data1" and "data2"; (3) decompress both "data1" and "data2". Again, the choice can be made on the basis of costs, by the query optimizer or within the operator implementation.

For the remaining operators, greater/less or arithmetic, the only possibility in all cases is to decompress the compressed attribute(s). In fact more than decompression is involved. Decompressed data is ASCII, but we need a binary version to do arithmetic. For this purpose it may be appropriate to extend the ADT model to include a third, "working" representation.

A special case is pattern matching with strings. Formally, it falls into the equality category. However, most database managers will handle it as a special case, by interpreting the pattern as regular expression and generating a finite automaton to perform the work. In this case the resources required to decompress the compressed attribute may well be overshadowed by the work of the automaton.

Cases which require compression and decompression can be very expensive if done repeatedly. We will return to this issue in Sections 5 and 6.

5. Query Processing of Compressed Data

In this section, we discuss the effect of the CD type on query processing.

5.1. Unaffected Query Processing Operators

Most query processing algorithms are completely unaffected by CD attributes [11]. This is partly due to the fact that the compression structures are lossless, so equality of compressed data is identical to equality of uncompressed data.

Exact-match selection is not affected since the ADT input routine for the CD type compresses the value to be selected, similar to the transformation of Figure 2. Range selection, on the other hand, as well as selection predicates involving arithmetic, require that data be decompressed. Projection and duplicate elimination are similarly unaffected. Even sort- or hash-based methods for duplicate elimination [3] will still work without change to the data. For example, sort-based methods will sort the data in order of compressed values, but in any case identical values will be adjacent and can be removed. Most importantly for query processing, join operations can be performed on compressed data, including compressed join attributes, if the two join columns are compressed with the same model.

5.2. Dissimilar Join Attributes

Consider the standard join example

```
SELECT emp.name, dept.floor
FROM emp, dept
WHERE emp.deptname = dept.deptname
```

where emp.deptname is CD and dept.deptname is not. Assume that dept is much smaller than emp, so the join algorithm chosen (e.g. nested loops or hash join) requires repeatedly accessing the dept relation and comparing for equality, as shown in Figure 3. The query plan without the compression node will be prohibitively expensive. The preferable query plan is to compress all values in dept.deptname, then conduct the join. This can be done with a *compression operator*, e.g., the "filter" operator in Volcano [10]. A

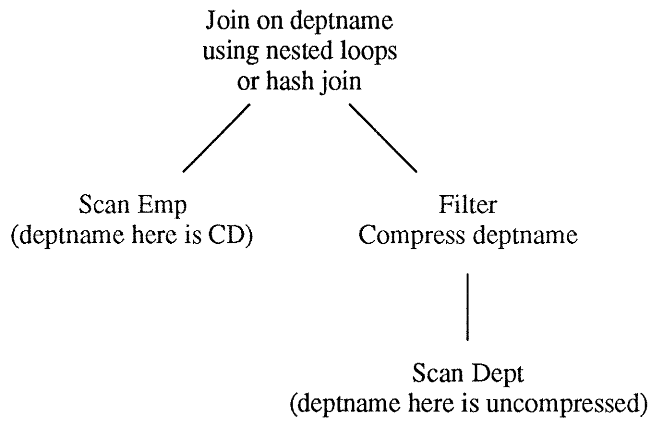

Figure 3. Query Plan for Dissimilar Join Attributes.

decompression operator will be needed for similar reasons.

Any join in which one of the joining attributes is of type CD and the other is either not CD or is CD but not using the same compression structure, is going to be a potential problem. We call these *dissimilar attribute joins*, and we discuss in Section 6.3 how to avoid them.

If we used sort-merge in the join of Figure 3, without the compression operator, we would get the wrong result, since emp and dept would have their deptname's in different order and none would match. Hence the query optimizer needs something like the "glue" concept of Starburst's optimizer [16] to avoid using algorithms which do not have properly structured input.

5.3. Indices

We close this section by commenting on the impact of compression on indices.

Hash indices are unaffected by compression except that their performance may improve since the compression operation may yield more uniformly distributed values for the hashed key and, because of the reduced size, an improved buffer hit rate.

Ordered indices support both single record and range retrieval. The retrieval of single records, for CD attributes, is unchanged, but range retrieval will no longer be available. On the other hand, it is unlikely that an attribute which will participate in a significant number of range retrievals will be made of type CD – see Section 6.1.

6. Choosing a Compression Structure

In this section, we discuss which attributes should be of type CD, what compression structure should be used, who should make those decisions, and when. We then point out a trap to avoid in such choices.

6.1. Which Attributes to Compress

The decision of which attributes to compress is a physical design decision. A bad decision will not cause any errors in the database. However, a bad decision will result in decreased performance either because a compressed attribute will need to be decompressed or because compression was not used and would have resulted in performance benefits. We give some guidelines for making this decision, then we consider some commercial data and demonstrate that a very large fraction of commercial data is a candidate for compression.

We recommend compressing all attributes except the two categories we discuss here, namely those for which compression will not save much space, and those which will need to be decompressed often.

For the first category, attributes for which compression will not save much space, the simplest example is numeric data, which is typically stored in binary format. To determine whether data falls into this first category, one need only look at the data. It is not necessary to have knowledge of the expected query plans. It is a fairly simple decision, since one can compress the data and then compare the compressed with decompressed sizes.

The second category, data that will often need to be decompressed, includes data which will participate in range and pattern matching queries (data participating in arithmetic queries is probably binary so will not be compressed). To determine which attributes fall into this category, one needs to know something about the expected queries and their frequency. It is much more difficult to decide whether range or arithmetic queries of this type will be frequent enough to make compression less attractive. The cost-benefit analysis is complicated by the fact that it is unclear in general how expensive it will be in a given database system to de/compress data.

6.2. Who Decides and When

Section 3, and its references, give us sufficient information to choose an optimal compression structure for a given attribute, IF we know the data in that attribute. This is of course a problem, since the data may change continually, while changing the compression structure for an attribute will mandate changes for each tuple in the relation.

One solution is for the DBMS to choose and update compression structures automatically. This has two problems. First, the DBMS cannot have knowledge of what data the attribute is likely to contain in the future. Second, changing compression structures will require the system to be unavailable for a substantial time, and it is unlikely that the DBMS can know the best time for such changes.

We feel it is more appropriate for the DBMS to merely give hints to the database administrator (DBA). These hints could be generated by the DBMS noticing that doing de/compression is taking substantial resources, or an encoding table has grown substantially in size, or that data in an attribute has changed significantly since the compression structure was last computed.

We feel the DBA should first choose compression structures at database design time, based on knowledge of the likely attribute contents and queries expected. The DBMS can assist this process by by storing some "default" structures, e.g. a table of Huffman codes for English data. The DBMS should also allow the user to specify the kind of data which is expected to be in the attribute, and should then chose a structure best suited to that data. The user should be able to specify the expected data as being:

(a) the actual data currently in the attribute, or

(b) the data in some file thought to be typical of the future contents of the relation, or

(c) that the attribute will contain English text.

The DBA will also need to change the compression structure as the database changes. The DBMS should give hints, when de/compression is taking substantial resources, or an encoding table has grown substantially in size, or when data in an attribute has changed significantly since the compression structure was last computed.

When a compression structure needs to be changed, alternative (a) above will normally be used, although it will require two scans of the relation, one to gather the data values and one to do the compression. Therefore it will be most appropriate to do this when the database is being backed up, or unloaded and reloaded for reorganization, so as to piggyback the data compression structure changes onto the un/reloading.

6.3. The Transitivity Trap

Suppose a database consists of the relations

Emp (empname, deptname)
Dept (deptname, floor)
Jobs (jobname, empname, percent)

Suppose further that joins are anticipated on empname and on deptname. Because of the dissimilar join problem discussed in Section 5.2, the compression structures for employee.empname and jobs.empname should be identical, as should the compression structures for dept.deptname and employee.deptname.

Now consider what happens when the DBA needs to update the compression structure on the attribute employee.empname. The DBA might decide to piggyback onto this an update to the compression structure for employee.deptname. A scan or two of the employee relation would then cascade to scans of the entire database. This trap can be avoided by planning ahead, and changing compression structures only when the total cost (including the cost of changes mandated by cascading) is worth the benefit.

7. Functionality

In this section, we show how the use of full-time compression will provide added functionality to database systems, by offloading responsibility for encoding.

Dictionary encoding is commonly used in applications. It can be used at three different levels.

Dictionary coding can be maintained throughout the system and displayed to the end user. For example, every car owner probably has tried to read a report like the following:

Insurance type: A4
Extra coverages: B2, C6
Premium discount: A

Here the end user must translate the codes, often using a dictionary on the back of the form into their "uncompressed form", which might show, for example, that an A discount is two percent.

A second level expects the application program to translate the code. Here, the application which writes the report above will translate each of the codes in the report into intelligible English, using a dictionary accessible to (or even hard coded in) the application program.

Finally, the work of translating codes into text could have been done by the DBA designer, who could declare a view. For example, if the relation

discount(code, text)

contains codes and texts for discounts, and

policy(id, discountcode, ...)

contains policy information such as discount code, the DBA could define a view:

```
define view as
select policy.id, discount.text
from policy, discount
where policy.discountcode = discount.code
```

Then the applications programmer could merely code against the view. This third method suffers from a fatal flaw, however: view updating. What will the database manager do if an application deletes the last policy with insurance type A4? The desirable action here is to keep the coding of that insurance type in the encoding table, so as to be prepared for future policies with type A4. But general purpose database managers cannot support such special case policies for views: most allow view updating only in special circumstances, typically not including join views such as our example.

Our full-time compression scheme appears to be more desirable than either of these three alternatives. Compared to all three, it provides tools to help choose a best compression structure and it gives hints regarding when that structure needs to be changed. Compared to the first two, it leaves the work up to the DBA, so the effort can be shared. Compared to the third method, it avoids the view update problem.

8. Performance

For the purpose of this performance comparison, consider the I/O costs for hybrid-hash Join [6, 22] using 400 pages of memory of two relations R and S that require uncompressed 1,000 pages for R and 5,000 pages for S, or half as much compressed. For simplicity, we ignore fragmentation and assume uniform hash value distributions.

First, consider the cost using uncompressed data. The cost of reading the stored data is 6,000 I/Os. When building the hash table on R, 398 pages can remain in memory and 602 pages must be written to two build overflow files. During probing with input S, records equivalent to 1,990 pages can be joined immediately with the resident hash table, and 3,010 pages must be written to two overflow files. Next, the two pairs of overflow files must be joined, requiring 602+3,010 I/Os to read them. The entire cost is 6,000 I/Os on permanent files and 7,224 I/Os on temporary files, for a total of 13,224 I/Os.

Now consider joining compressed input relations. Reading the initial compressed data requires 500+2,500 I/Os. In the build phase, 399 pages remain in memory, 101 pages are written to an overflow file. In the probe phase, records equivalent to 1,995 pages are joined immediately, and 505 pages are written to disk. Joining the two overflow files requires 101+505 I/Os. The entire cost is 3,000 I/Os to permanent files and 1,212 I/Os to temporary files, for a total of 4,212 I/Os.

The total I/O costs differ by a factor of more than three. While the I/O costs for the permanent files differ by a factor of two, as expected for a compression ratio of 50%, the I/O costs for temporary files differ by a factor of almost six. A factor of two could easily be expected; however, the improved utilization of memory (more records remain in the hash table during the build phase) significantly reduces the number of records that must be written to overflow files. Thus, compression reduces both the number and size of records written to temporary files, resulting in a reduction of I/O costs on temporary files by a factor of six.

If the compression scheme had been a little more effective, i.e., a factor of 2½ instead of 2 or a reduction to 40% instead of 50%, overflow files would have been avoided entirely for compressed data, leaving only the I/O on permanent data. The total I/O costs would have differed by a factor of 5½, 2,400 to 13,224 I/Os. Figure 4 shows the effect of the compression factor on hybrid-hash Join performance for relations R and S. The numbers above the curve indicate the exact I/O cost for the compression factors marked at the bottom axis.

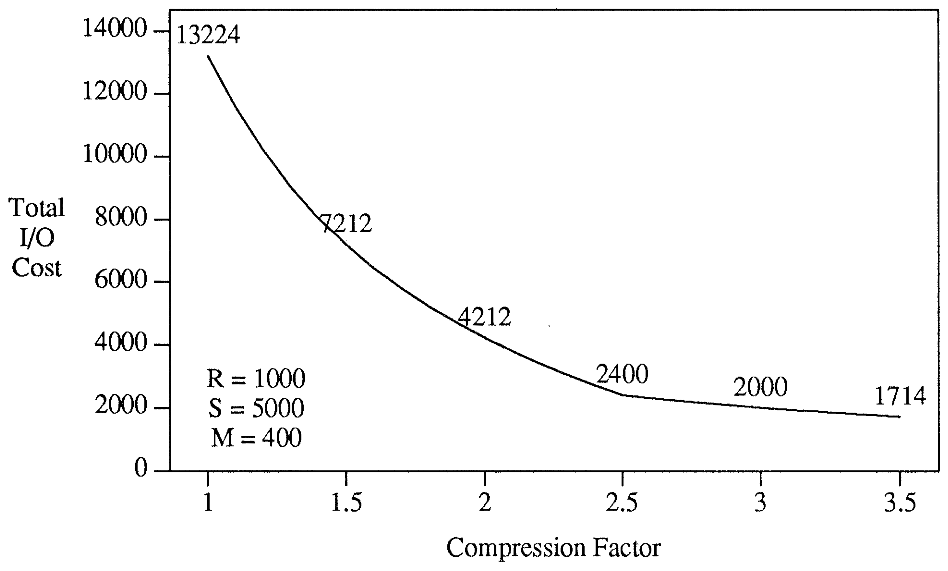

Figure 4. Effect of Compression on Hybrid-Hash Join Performance.

The graph can be divided into two regions. For compression factors below 2½, the build input is larger than memory, hash table overflow occurs, and I/O reduction by compression is more than the compression factor, similar to the example above. For compression factors above 2½, no overflow occurs and compression only reduces I/O on permanent files. It is very encouraging to observe in this graph that already very moderate compression factors, e.g., 1½, reduce the total I/O cost significantly. Even if some additional cost is incurred for decompressing output data, the performance gain through compressed permanent and temporary data on disk and in memory far outweights the costs of decompression.

Figure 5 shows the effect of compression on hybrid-hash Join performance of R and S for a variety of memory sizes. The bottom-most curve for a memory size of 1,000 pages reflects the situation without overflow. The curve for 500 pages of memory has a steep gradient up to compression factor 2. Beyond this point, the hash table fits into memory and the curves for 500 and 1,000 pages coincide. For 250 pages of memory, which is ¼ of R, the curve joins the other curves without overflow at a compression factor of 4. For all smaller memory sizes, the hash table does not fit into memory in the considered spectrum of compression factors. However, the performance gain is more than the compression factor for all memory sizes. For 50 or 100 pages of memory, the curves are very close to each other because almost all of R and S must be written to overflow files. If memory is very limited in a system, it might be more important to

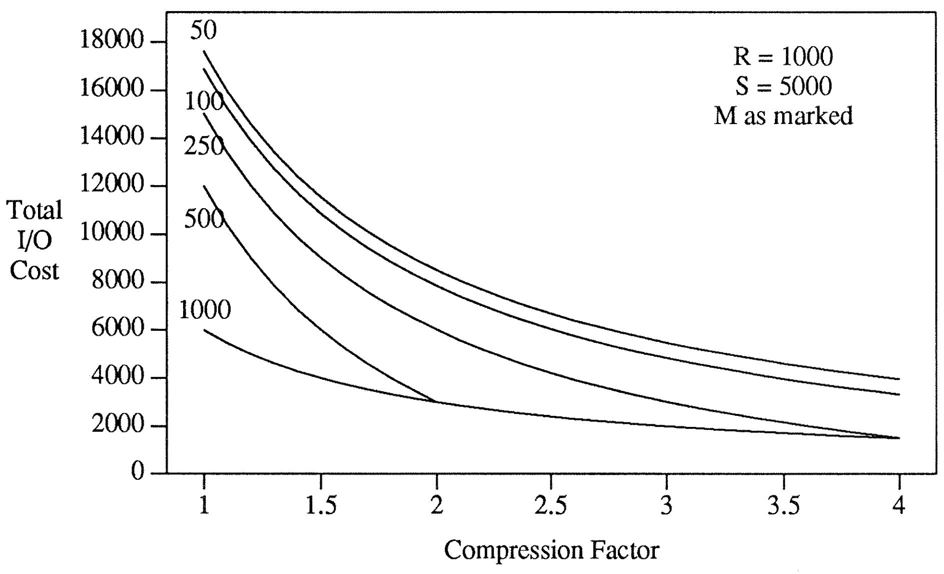

Figure 5. Effect of Compression and Memory Size on Hybrid-Hash Join Performance.

allocate it to operators that may be able to run without overflow, and to use memory there with maximal efficiency, i.e., the best compression scheme possible.

Figure 6 shows the speedup for the previous figure. The bottom-most curve, for 1,000 pages of memory, represents linear speedup. All other curves indicate super-linear speedup. The curve for 500 pages has an obvious "knee" at compression factor 2 which had been already visible in the previous figure. For 250 pages of memory, the knee would be located at compression factor 4, at the edge of the graph, where the curve indicates a speedup factor of slightly more than 10. Considering that a speedup of 10 could be achieved with a compression factor of only 4 makes it imperative to further investigate the effect of compression on database query processing algorithms and their performance.

Figure 7 shows the performance for larger input relations. As in the previous figures, even a small amount of compression improves the performance significantly. However, for small memory sizes compared to the build input, the effect of fitting more compressed records than uncompressed records in memory is not as strong as for medium-size relations. Nonetheless, the gain in I/O performance is at least equal to the compression factor, and has the steepest decline for the small compression factors. For larger memory sizes, 5,000 pages in Figure 7, for which the build input size is a small multiple of the memory, we see the same effect as before — a steep gradient for small compression factors with a "knee" where the

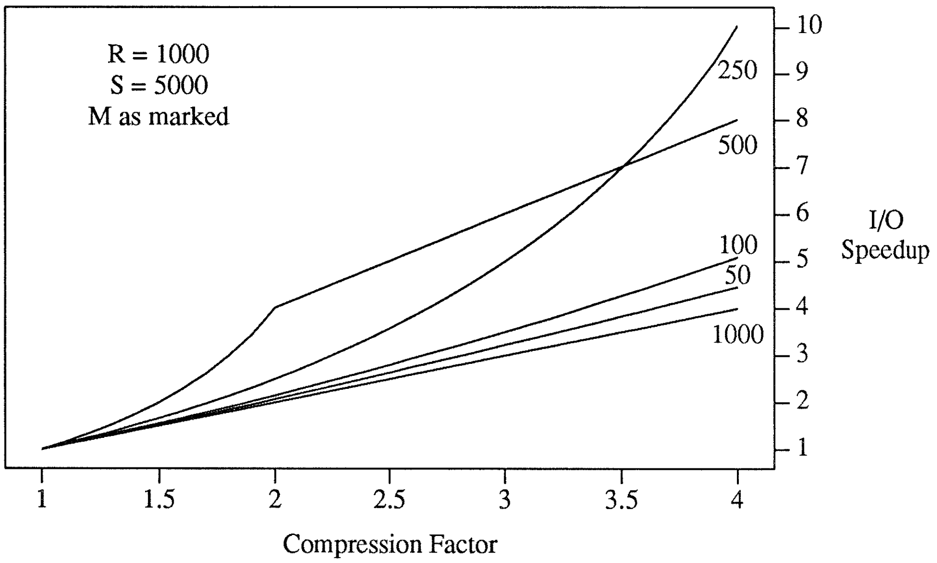

Figure 6. Speedup of Hybrid-Hash Join through Compression.

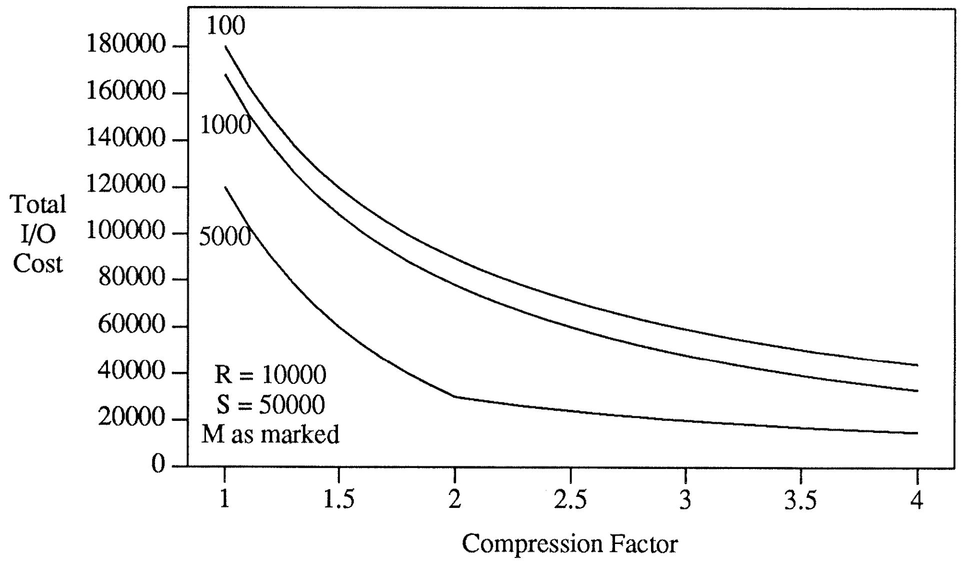

Figure 7. Effect of Compression on Large Hybrid-Hash Join Performance.

compression factor equals the quotient of build input size and memory size.

The same effects observed for a single hybrid-hash Join can also be observed for complex queries and for sort-merge join. We omit this analysis and instead refer to the comparisons of sort-merge join and hybrid-hash Join in [19, 22].

9. Summary and Conclusions

We have introduced the concept of full-time data compression, in which data is not only compressed during disk storage but remains compressed as much as possible during query processing.

We surveyed data compression techniques and pointed out that, though some of the most powerful techniques (giving compression factors of 4 and more) may not be appropriate, other techniques (giving compression by a factor of 2 or more) are well suited to full-time compression.

We demonstrated how full-time compression can be implemented easily using ADTs in systems supporting abstract data types. Full-time compression requires small changes to the DBMS's query processing and query optimization. We suggested that the DBA choose which attributes to compress, and the techniques to use in compressing them, with extensive help from the DBMS. Finally, we pointed out that full-time compression adds significantly to the functionality of a DBMS by taking over coding functions normally performed at a higher level.

Full-time compressed data saves space at many levels, from disk storage to memory cache. We have observed that it improves performance of typical query processing algorithms by a factor larger than the compression factor. This is true even for algorithms such as hash join, whose performance is inherently linear, since more data can remain in memory.

We conclude that full-time compression is attainable within the context of modern DBMSs (at least those which support ADTs) and can provide significant performance and functionality improvements.

Acknowledgements

Mike Stonebraker suggested using ADTs to support compressed data types, and Bruce Lindsay encouraged us to pursue this issue. David Aucsmith provided us with helpful background on compression algorithms.

References

1. P. A. Alsberg, ''Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring'', *Proc. of the IEEE 63*, 8 (August 1975), 1114-1122.
2. T. Bell, I. H. Witten and J. G. Cleary, ''Modelling for Text Compression'', *ACM Computing Surveys 21*, 4 (December 1989), 557.
3. D. Bitton and D. J. DeWitt, ''Duplicate Record Elimination in Large Data Files'', *ACM Transactions on Database Systems 8*, 2 (June 1983), 255-265.
4. M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita and S. Vandenberg, ''The EXODUS Extensible DBMS Project: An Overview'', in *Readings on Object-Oriented Database Systems*, D. M. S. Zdonik (editor), Morgan Kaufman, San Mateo, CA., 1989.
5. G. V. Cormack, ''Data Compression In a Database System'', *Communications of the ACM 28*, 12 (December 1985), 1336-1342.
6. D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, ''Implementation Techniques for Main Memory Database Systems'', *Proceedings of the ACM SIGMOD Conference*, Boston, MA., June 1984, 1-8.
7. S. J. Eggers, F. Olken and A. Shoshani, ''A Compression Technique for Large Statistical Data Bases'', *Proceeding of the Conference on Very Large Data Bases*, Cannes, France, September 1981, 424-434.
8. R. G. Gallager, ''Variations on a Theme by Huffman'', *IEEE Transactions on Information Theory IT-24*, 6 (1978), 668-674.
9. P. Goyal, ''Coding methods for text string search on compressed databases'', *Information Systems 8*, 3 (1983), 231.
10. G. Graefe, ''Volcano, An Extensible and Parallel Dataflow Query Processing System'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 481*, July 1990.
11. G. Graefe and L. D. Shapiro, ''Data Compression and Database Performance'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 496*, November 1990.
12. L. Haas, W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey and E. Shekita, ''Starburst Mid-Flight: As the Dust Clears'', *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 143-160.

13. D. A. Huffman, ''A method for the construction of minimum redundancy codes'', *Proc. IRE 40* (1952), 1098-1101.
14. D. A. Lelewer and D. S. Hirschberg, ''Data Compression'', *ACM Computing Surveys 19*, 3 (September 1987), 261-296.
15. J. Li, D. Rotem and H. Wong, ''A New Compression Method with Fast Searching on Large Data Bases'', *Proceeding of the Conference on Very Large Data Bases*, Brighton, England, August 1987, 311-318.
16. G. M. Lohman, ''Grammar-Like Functional Rules for Representing Query Optimization Alternatives'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 18-27.
17. J. E. Mulford and R. K. Ridall, ''Data Compression Techniques for Economic Processing of Large Commercial Files'', *ACM Proc. Symp. Information Storage and Retrieval*, 1971, 207-215.
18. F. Olken and D. Rotem, ''Rearranging Data to Maximize the Efficiency of Compression'', *Journal of Computer and System Sciences 38*, 2 (1989), 405.
19. D. Schneider and D. DeWitt, ''A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment'', *Proceedings of the ACM SIGMOD Conference*, Portland, OR, May-June 1989, 110.
20. D. G. Severance and W. L. Maxwell, ''Comparison of Alternatives for the Representation of Data Item Values in an Information System'', *Data Base 5* (1973).
21. D. G. Severance, ''A practitioner's guide to data base compression'', *Information Systems 8*, 1 (1983), 51.
22. L. D. Shapiro, ''Join Processing in Database Systems with Large Main Memories'', *ACM Transactions on Database Systems 11*, 3 (September 1986), 239-264.
23. M. Stonebraker, ''Inclusion of New Types in Relational Database Systems'', *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1986, 262.
24. M. Stonebraker, L. A. Rowe and M. Hirohama, ''The Implementation of Postgres'', *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (March 1990), 125-142.
25. H. K. T. Wong and J. C. Li, ''Transposition Algorithms on Very Large Compressed Data'', *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 304-311.
26. J. Ziv and A. Lempel, ''A Universal algorithm for sequential data compression'', *IEEE Transactions on Information Theory IT-23*, 3 (May 1977), 337-343.
27. J. Ziv and A. Lempel, ''Compression of individual sequences via variable-rate coding'', *IEEE Transactions on Information Theory IT-24*, 5 (September 1978), 530-536.