

How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations

Vijayshankar Raman Garret Swart
IBM Almaden Research Center, San Jose, CA 95120

ABSTRACT

We present a method to compress relations close to their entropy while still allowing efficient queries. Column values are encoded into variable length codes to exploit skew in their frequencies. The codes in each tuple are concatenated and the resulting tuplecodes are sorted and delta-coded to exploit the lack of ordering in a relation. Correlation is exploited either by co-coding correlated columns, or by using a sort order that leverages the correlation. We prove that this method leads to near-optimal compression (within 4.3 bits/tuple of entropy), and in practice, we obtain up to a 40 fold compression ratio on vertical partitions tuned for TPC-H queries.

We also describe initial investigations into efficient querying over compressed data. We present a novel Huffman coding scheme, called segregated coding, that allows range and equality predicates on compressed data, *without* accessing the full dictionary. We also exploit the delta coding to speed up scans, by reusing computations performed on nearly identical records. Initial results from a prototype suggest that with these optimizations, we can efficiently scan, tokenize and apply predicates on compressed relations.

1. INTRODUCTION

Data movement is a major bottleneck in data processing. In a database management system (DBMS), data is generally moved from a disk, through an I/O network, and into a main memory buffer pool. After that it must be transferred up through two or three levels of processor caches until finally it is loaded into processor registers. Even taking advantage of multi-task parallelism, hardware threading, and fast memory protocols, processors are often stalled waiting for data: the price of a computer system is often determined by the quality of its I/O and memory system, not the speed of its processors. Parallel and distributed DBMSs are even more likely to have processors that stall waiting for data from another node. Many DBMS “utility” operations such as replication/backup, ETL (extract-transform and load), and internal and external sorting are also limited by the cost of data movement¹.

DBMSs have traditionally used compression to alleviate this data movement bottleneck. For example, in IBM’s DB2 DBMS, an administrator can mark a table as compressed, in which case individual records are compressed using a dictionary scheme [1]. While this approach reduces I/Os, the data still needs to be decompressed, typically a page or record at a time, before it can be queried. This decompression

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB ’06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

increases CPU cost, especially for large compression dictionaries that don’t fit in cache. Worse, since querying is done on uncompressed data, the in-memory query execution is not sped up at all. Furthermore, as we see later, a vanilla gzip or dictionary coder give suboptimal compression – we can do much better by exploiting semantics of relations.

Another popular way to compress data is a method that can be termed domain coding [6, 8]. In this approach values from a domain are coded into a tighter representation, and queries run directly against the coded representation. For example, values in a CHAR(20) column that takes on only 5 distinct values can be coded with 3 bits. Often such coding is combined with a layout where all values from a column are stored together [5, 6, 11, 12]. Operations like scan, select, project, etc. then become array operations that can be done with bit vectors.

Although it is very useful, domain coding alone is insufficient, because it poorly exploits three sources of redundancy in a relation:

- *Skew*: Real-world data sets tend to have highly skewed value distributions. Domain coding assigns fixed length (often byte aligned) codes to allow fast array access. But it is inefficient in space utilization because it codes infrequent values in the same number of bits as frequent values.
- *Correlation*: Correlation between columns within the same row is common. Consider an order ship date and an order receipt date; taken separately, both dates may have the same value distribution and may code to the same number of bits. However, the receipt date is most likely to be within a one week period of ship date. So, for a given ship date, the probability distribution of receipt date is highly skewed, and the receipt date can be coded in fewer bits.
- *Lack of Tuple Order*: Relations are multi-sets of tuples, not sequences of tuples. A physical representation of a relation is free to choose its own order – or no order at all. We shall see that this representation flexibility can be used for additional, often substantial compression.

Column and Row Coding

This paper presents a new compression method based on a mix of column and tuple coding.

Our method has three components: We encode column values with Huffman codes [16] in order to exploit the skew in the value frequencies – this results in variable length *field codes*. We then concatenate the field codes within each tuple to form *tuplecodes* and sort and then delta code these tuplecodes, taking advantage of the lack of order within a relation. We exploit correlation between columns within a tuple by using

¹ For typical distributions, radix sort runs in linear time and thus in-memory sort is dominated by the time to move data between memory and cache (e.g., see Jim Gray’s commentary on the 2005 Datamation benchmark [17]). External sorting is of course almost entirely movement-bound.

either value concatenation and co-coding, or careful column ordering within the tuplecode. We also allow domain-specific transformations to be applied to the column before the Huffman coding, e.g. text compression for strings.

We define a notion of entropy for relations, and prove that our method is near-optimal under this notion, in that it asymptotically compresses a relation to within 4.3 bits/tuple of its entropy.

We have prototyped this method in a system called *csvzip* for compression and querying of relational data. We report on experiments with *csvzip* over data sets from TPC-H, TPC-E and SAP/R3. We obtain compression factors from 7 to 40, substantially better than what is obtained with *gzip* or with domain coding, and also much better than what others have reported earlier. When we don't use co-coding, the level of compression obtained is sensitive to the position of correlated columns within the tuplecode. We discuss some heuristics for choosing this order, though this needs further study.

Efficient Operations over Compressed Data

In addition to compression, we also investigate scans and joins over compressed data. We currently do not support incremental updates to compressed tables.

Operating on the compressed data reduces our decompression cost; we only need to decode fields that need to be returned to the user or are used in aggregations. It also reduces our memory throughput and capacity requirements. The latter allows for larger effective buffer sizes, greatly increasing the speed of external data operations like sort.

Querying compressed data involves parsing the encoded bit stream into records and fields, evaluating predicates on the encoded fields, and computing joins and aggregations. Prior researchers have suggested *order-preserving codes* [2,15,18] that allow range queries on encoded fields. However this method does not extend efficiently to variable length codes. Just tokenizing a record into fields, if done naively, involves navigating through several Huffman dictionaries. Huffman dictionaries can be large and may not fit in the L2 cache, making the basic operation of scanning a tuple and applying selection or projection operations expensive. We introduce two optimizations to speed up querying:

- *Segregated Coding*: Our first optimization is a novel scheme for assigning codes to prefix trees. Unlike a true order preserving code, we preserve order only within codes of the same length. This allows us to evaluate many common predicates directly on the coded data, and also to find the length of each codeword *without accessing the Huffman dictionary*, thereby reducing the memory working set of tokenization and predicate evaluation.
- *Short Circuited Evaluation*: Our delta coding scheme sorts the tuplecodes, and represents each tuplecode by its delta from the previous tuplecode. A side-effect of this process is to cluster tuples with identical prefixes together, which means identical values for columns that are early in the concatenation order. This allows us to avoid decoding, selecting and projecting columns that are unchanged from the previous tuple.

Before diving into details of our method, we recap some information theory basics and discuss some related work.

1.1 Background and Related Work

1.1.1 Information Theory

The theoretical foundation for much of data compression is information theory [3]. In the simplest model, it studies the compression of sequences emitted by 0^{th} -order information sources – ones that generate values i.i.d (independent and identically distributed) from a probability distribution D . Shannon's celebrated source coding theorem [3] says that one cannot code a sequence of values in less than $H(D)$ bits per value on average, where $H(D) = \sum_{i \in D} p_i \lg(1/p_i)$ is the *entropy* of the distribution D with probabilities p_i .

Several well studied codes like the Huffman and Shannon-Fano codes achieve $1 + H(D)$ bits/tuple asymptotically, using a *dictionary* that maps values in D to *codewords*. A value with occurrence probability p_i is coded in roughly $\lg(1/p_i)$ bits, so that more frequent values are coded in fewer bits.

Most coding schemes are *prefix codes* – codes where no codeword is a prefix of another codeword. A prefix code dictionary is often implemented as a *prefix tree* where each edge is labelled 0 or 1 and each leaf maps to a codeword (the string of labels on its path from the root). By walking the tree one can tokenize a string of codewords without using delimiters – every time we hit a leaf we output a codeword and jump back to the root.

The primary distinction between relations and the sources considered in information theory is the lack of order: relations are multi-sets, and not sequences. Secondly, we want the compressed relation to be directly queryable, whereas it is more common in the information theory literature for the sequence to be decompressed and then pipelined to the application.

1.1.2 Related work on database compression

DBMSs have long used compression to help alleviate their data movement problems. The literature has proceeded along two strands: field wise compression, and row wise compression.

Field Wise Compression: Graefe and Shapiro [20] were among the first to propose field-wise compression, because only fields in the projection list need to be decoded. They also observed that operations like join that involve only equality comparisons can be done without decoding. Goldstein and Ramakrishnan [8] propose to store a separate reference for the values in each page, resulting in smaller codes. Many column-wise storage schemes (e.g., [5,6,12]) also compress values within a column. Some researchers have investigated order-preserving codes in order to allow predicate evaluation on compressed data [2,15,18]. [2] study order preserving compression of multi-column keys. An important practical issue is that field compression can make fixed-length fields into variable-length ones. Parsing and tokenizing variable length fields increases the CPU cost of query processing, and field delimiters, if used, undo some of the compression. So almost all of these systems use fixed length codes, mostly byte-aligned. This approximation can lead to substantial loss of compression as we argue in Section 2.1.1. Moreover, it is not clear how to exploit correlation with a column store.

Row Wise Compression: Commercial DBMS implementations have mostly followed the row or page level compression approach where data is read from disk,

Domain	Num. possible values	Num. Likely vals (in top 90 percentile)	Entropy (bits/value)	Comments
Ship Date	3650000	1547.5	9.92	We assume that the db must support all dates till 10000 A.D., but 99% of dates will be in 1995-2005, 99% of those are weekdays, 40% of those are in the 10 days each before New Year and Mother's day.
Last Names	2^{160} (char (20))	≈ 80000	26.81	We use exact frequencies for all U.S. names ranking in the top 90 percentile (from census.gov), and extrapolate, assuming that all $\approx 2^{160}$ names below 10 th percentile are equally likely. This over-estimates entropy.
Male first names	2^{160} (char (20))	1219	22.98	
Customer Nation	$215 \approx 2^{7.75}$	2	1.82	We use country distribution from import statistics for Canada (from www.wto.org) – the entropy is lesser if we factor in poor countries, which trade much less and mainly with their immediate neighbours.

Table 1: Skew and Entropy in some common domains

decompressed, and then queried. IBM DB2 [1] and IMS[10] use a non-adaptive dictionary scheme, with a dictionary mapping frequent symbols and phrases to short code words. Some experiments on DB2 [1] indicate a factor of 2 compression. Oracle uses a dictionary of frequently used symbols to do page-level compression and report a factor of 2 to 4 compression [21]. The main advantage of row or page compression is that it is simpler to implement in an existing DBMS, because the code changes are contained within the page access layer. But it has a huge disadvantage in that, while it reduces I/O, the memory and cache behaviour is worsened due to decompression costs. Some studies suggest that the CPU cost of decompression is also quite high [8].

Delta Coding: C-Store [6] is a recent system that does column-wise storage and compression. One of its techniques is to delta code the sort column of each table. This allows some exploitation of the relation's lack-of-order. [6] does not state how the deltas are encoded, so it is hard to gauge the extent to which this is exploited. In a different context, inverted lists in search engines are often compressed by computing deltas among the URLs, and using heuristics to assign short codes to common deltas (e.g., [19]). We are not aware of any rigorous work showing that delta coding can compress relations close to their entropy.

Lossy Compression: There is a vast literature on lossy compression for images, audio, etc, and some methods for relational data, e.g., see Spartan [7]. These methods are complementary to our work – any domain-specific compression scheme can be plugged in as we show in Section 2.1.4. We believe lossy compression can be useful for measure attributes that are used only for aggregation.

2. COMPRESSION METHOD

Three factors lead to redundancy in relation storage formats: skew, tuple ordering, and correlation. In Section 2.1 we discuss each of these in turn, before presenting a composite compression algorithm that exploits all three factors to achieve near-optimal compression.

Such extreme compression is ideal for pure data movement tasks like backup or replication. But it is at odds with efficient querying. Section 2.2 describes some relaxations that sacrifice some compression efficiency in return for simplified querying.

This then leads into a discussion of methods to query compressed relations, in Section 3.

2.1 Squeezing redundancy out of a relation

2.1.1 Exploiting Skew by Entropy Coding

Many domains have highly skewed data distributions. One form of skew is not inherent in the data itself but is part of the representation – a schema may model values from a domain with a data type that is much larger. E.g., in TPC-H, the C_MKTSEGMENT column has only 5 distinct values but is modelled as CHAR(10) – out of 256^{10} distinct 10-byte strings, only 5 have non-zero probability of occurring! Likewise, post Y2K, a date is often stored as eight 4-bit digits (mmddyyyy), but the vast majority of the 16^8 possible values map to illegal dates.

Prior work (e.g., [6, 12]) exploits this using a technique we'll call *domain coding*: legal values from a large domain are mapped to values from a more densely packed domain – e.g., C_MKTSEGMENT can be coded as a 3 bit number. To permit array based access to columns, each column is coded to a fixed number of bits.

While useful, this method does not address skew within the value distribution. Many domains have long-tailed frequency distributions where the number of possible values is much more than the number of likely values. Table 1 lists a few such domains. E.g., 90% of male first names fall within 1219 values, but to catch all corner cases we would need to code it as a CHAR(20), using 160 bits, when the entropy is only 22.98 bits. We can exploit such skew fully through *entropy coding*.

Probabilistic Model of a Relation: Consider a relation R with column domains COL_1, \dots, COL_k . For purposes of compression, we view the values in COL_i as being generated by an i.i.d. (independent and identically distributed) information source over a probability distribution D_i . Tuples of R are viewed as being generated by an i.i.d information source with joint probability distribution: $D = (D_1, D_2, \dots, D_k)$.² We can estimate

² By modeling the tuple sources as i.i.d., we lose the ability to exploit inter-tuple correlations. To our knowledge, no one has studied such correlations in databases – all the work on correlations has been among fields within a tuple. If inter-tuple correlations are significant, the information theory literature on compression of non zero-order sources might be applicable.

m	est. $H(\mathbf{delta}(R))$ in bits
10000	1.897577 m
100,000	1.897808 m
1,000,000	1.897952 m
10,000,000	1.89801 m
40,000,000	1.898038 m

Table 2: Entropy of $\mathbf{delta}(R)$ for a multi-set R of m values picked uniformly, i.i.d. from $[1, m]$ (100 trials)

each D_i from the actual value distribution in COL_i , optionally extended with some domain knowledge. For example, if COL_i has {Apple, Apple, Banana, Mango, Mango, Mango}, then D_i is the distribution $\{p_{Apple} = 0.333, p_{Banana} = 0.167, p_{Mango} = 0.5\}$.

Schemes like Huffman and Shannon-Fano code such a sequence of i.i.d values by assigning shorter codes to frequent values [3]. On average, they can code each value in COL_i with at most $1 + H(D_i)$ bits, where $H(X)$ is the entropy of distribution X – hence these codes are also called “entropy codes.” Using an entropy coding scheme, we can code the relation R with $\sum_{1 \leq i \leq k} |R| (1 + H(D_i) + \text{DictSize}(COL_i))$ bits, where $\text{DictSize}(COL_i)$ is the size of the dictionary mapping code words to values of COL_i .

If a relation were a *sequence* of tuples, assuming that the domains D_i are independent (we relax this in 2.1.3), this coding is optimal, by Shannon’s source coding theorem [3]. But relations are not sequences, they are multi-sets of tuples, and permit further compression.

2.1.2 Order: Delta Coding Multi-Sets

Consider a relation R with just one column, COL_1 , containing m numbers chosen uniformly and i.i.d from the integers in $[1, m]$. Traditional databases would store R in a way that encodes both the content of R and some incidental ordering of its tuples. Denote this order-significant view of the relation as \mathbf{R} (we use **bold** font to indicate a sequence).

The number of possible instances of \mathbf{R} is m^m , each of which has equal likelihood, so $H(\mathbf{R})$ is $m \lg m$. But R needs much less space because we don’t care about the ordering. Each distinct instance of R corresponds to a distinct outcome of throwing of m balls into m equal probability bins. So, by standard combinatorial arguments (see [14], [4]), there are $\binom{2m-1}{m} \approx$

$4^m / \sqrt{4\pi m}$ different choices for R , which is much less than m^m . A simple way to encode R is as a *coded delta sequence*:

- 1) Sort the entries of R , forming sequence $\mathbf{v} = v_1, \dots, v_m$
- 2) Form a sequence $\mathbf{delta}(R) = v_1, v_2 - v_1, v_3 - v_2, \dots, v_m - v_{m-1}$
- 3) Entropy code the differences in \mathbf{delta} to form a new sequence $\mathbf{code}(R) = v_1, \text{code}(v_2 - v_1), \dots, \text{code}(v_m - v_{m-1})$

Space savings by delta coding

Intuitively, sorting and delta coding compresses R because the distribution of deltas is tighter than that of the original integers – small deltas are much more likely than large ones. Formally:

Lemma 1: If R is a multi-set of m values picked uniformly with repetition from $[1, m]$, and $m > 100$, then each delta in $\mathbf{delta}(R)$ has entropy < 2.67 bits.

Proof Sketch: See Appendix 7. \square

Corollary 1.1: $\mathbf{code}(R)$ occupies $< 2.67 m$ bits on average.

This bound is far from tight. Table 2 shows results from a Monte-Carlo simulation where we pick m numbers i.i.d from $[1, m]$, calculate the distribution of deltas, and estimate their entropy. Notice that the entropy is always less than 2 bits. Thus, Delta coding compresses R from $m \lg m$ bits to $\lg m + 2(m-1)$ bits, saving $(m-1)(\lg m - 2)$ bits. For large databases, $\lg m$ can be about 30 (e.g., 100GB at 100B/tuple). As experiments in Section 4.1 show, when a relation has only a few columns, such delta coding alone can give up to a 10 fold compression.

This analysis applies to a relation with one column, chosen uniformly from $[1, m]$. We generalize this to a method that works on arbitrary relations in Section 2.1.4.

Optimality of Delta Coding

Such delta coding is also very close to optimal – the following lemma shows we cannot reduce the size of a sequence by more than $\lg m!$ just by viewing it as a multi-set.

Lemma 2: Given a vector \mathbf{R} of m tuples chosen i.i.d. from a distribution D and the multi-set R of values in \mathbf{R} , (\mathbf{R} and R are both random variables), $H(\mathbf{R}) \geq m H(D) - \lg m!$

Proof Sketch: Since the elements \mathbf{R} are chosen i.i.d., $H(\mathbf{R}) = m H(D)$. Now, augment the tuples t_1, t_2, \dots, t_m of \mathbf{R} with a “serial-number” column SNO, where $t_i.\text{SNO} = i$. Ignoring the ordering of tuples in this augmented vector, we get a set, call it R_1 . Clearly there is a bijection from R_1 to R , so $H(R_1) = m H(D)$. But R is just a projection of R_1 , on all columns except SNO. For each relation R , there are at most $m!$ relations R_1 whose projection is R . So $H(R_1) \leq H(R) + \lg m!$ \square

So, with delta coding we are off by at most $\lg m! - m(\lg m - 2) \approx m(\lg m - \lg e) - m(\lg m - 2) = m(2 - \lg e) \approx 0.6$ bits/tuple from the best possible compression. This loss occurs because the deltas are in fact mildly correlated (e.g., sum of deltas = m), but we do not exploit this correlation – we code each delta separately to allow pipelined decoding.

2.1.3 Correlation

Consider a pair of columns (partKey, price), where each partKey largely has a unique price. Storing both partKey and price separately is wasteful; once the partKey is known, the range of possible values for price is limited. Such inter-field correlation is quite common and is a valuable opportunity for relation compression.

In Section 2.1.1, we coded each tuple in $\sum_j H(D_j)$ bits. This is optimal only if the column domains are independent, that is, if the tuples are generated with an independent joint distribution (D_1, D_2, \dots, D_k) . For any joint distribution, $H(D_1, \dots, D_k) \leq \sum_j H(D_j)$, with equality if and only if the D_j ’s are independent [3]. Thus any correlation strictly reduces the entropy of relations over this set of domains.

We have three methods to exploit correlation: co-coding, dependent coding and column ordering.

Co-coding concatenates correlated columns, and encodes them using a single dictionary. If there is correlation, this combined code is more compact than the sum of the individual field codes.

A variant approach we call dependent coding builds a Markov model of the column probability distributions, and uses it to assign Huffman codes. E.g, consider columns partKey,

