

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 O^{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 $\{p_{\text{Apple}} = 0.333, p_{\text{Banana}} = 0.167, p_{\text{Mango}} = 0.5\}$.

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 $\binom{2m-1}{m} \simeq$

$4^m / \sqrt{4\pi m}$ 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 $\lg m! - m(\lg m - 2) \simeq m(\lg m - \lg e) - m(\lg m - 2) = m(2 - \lg e) \simeq 0.6$ 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

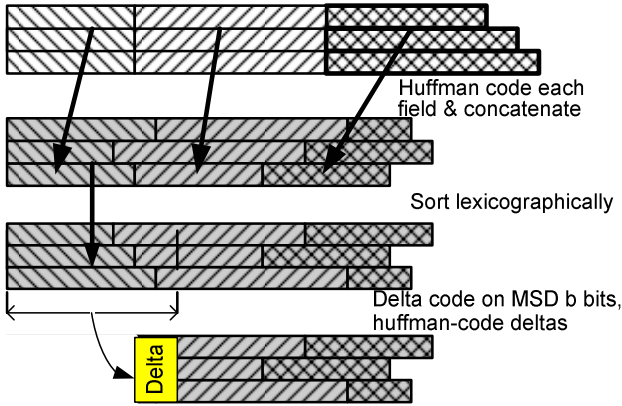
Consider a pair of columns (partKey, price), where each partKey largely has a unique price. Storing both partKey and price separately is wasteful; once the partKey is known, the range of possible values for price is limited. Such inter-field correlation is quite common and is a valuable opportunity for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



1. for each tuple $(t.c_1, t.c_2, \dots, t.c_k)$ of $|R|$ do
 - 1a. for each col c_i that needs type specific transform, do:

$$t.c_i \leftarrow \text{type_specific_transform}(t.c_i)$$
 - 1b. concatenate correlated columns together
 - 1c. for each col $i = 1$ to k do:

$$t.c_i \leftarrow \text{Huffman_Code}(t.c_i)$$
 - 1d. $\text{tupleCode} \leftarrow \text{concat}(t.c_1, t.c_2, \dots, t.c_k)$
 - 1e. If $\text{size}(\text{tupleCode}) < \lfloor \lg(|R|) \rfloor$ bits

$$\text{pad it with random bits to make it } \lfloor \lg(|R|) \rfloor \text{ bits long.}$$
2. sort the coded tuples lexicographically.
3. for each pair of adjacent tuples p, q in sorted order do:
 - 3a. let $p = p_1 p_2, q = q_1 q_2$,

$$\text{where } p_1, q_1 \text{ are } \lfloor \lg(|R|) \rfloor\text{-bit prefixes of } p, q$$
 - 3b. $\text{deltaCode} \leftarrow \text{Huffman_Code}(q_1 - p_1)$.
 - 3c. replace p with $\text{deltaCode}.p_2$

Algorithm 3: Pseudo-Code for compressing a relation

price, and brand, where (partKey,price) and (partKey,brand) are pair wise correlated, but price and brand are independent given the partKey. Instead of co-coding all three columns, we can assign a Huffman code to partKey and then choose the Huffman dictionary for coding price and brand based on the code for partKey. Both co-coding and dependent coding will code this relation to the same number of bits but when the correlation is only pair wise, dependent coding results in smaller Huffman dictionaries, which can mean faster decoding.

Both co-coding and dependent coding exploit correlation maximally, but cause problems when we want to run range queries on the dependent column. In Section 2.2.2, we present a different technique that keeps correlated columns separate (to allow fast queries), and instead exploits correlation by tuning the column ordering within a tuple.

Currently, csvzip implements co-coding and column ordering. The column pairs to be co-coded and the column order are specified manually as arguments to csvzip. An important future challenge is to automate this process.

2.1.4 Composite Compression Algorithm

Having seen the basic kinds of compression possible, we now proceed to design a composite compression algorithm that exploits all three forms of redundancy and allows users to plug in custom compressors for idiosyncratic data types (images, text, dates, etc). Algorithm 3 describes this in pseudo-code and

gives an example of how the data is transformed. Figure 4 shows a process flow chart. The algorithm has two main pieces:

Column Coding: For each tuple, we first perform any type specific transforms (supplied by the user) on columns that need special handling (1a). For example, we can apply a text compressor on a long VARCHAR column, or split a date into week of year and day of week (to more easily capture skew towards weekdays). Next we co-code correlated columns (1b), and then replace each column value with a Huffman code (1c). We use Huffman codes as a default because they are asymptotically optimal, and we have developed a method to efficiently run selections and projections on concatenated Huffman codes (Section 3). We currently compute the codes using a statically built dictionary rather than a Ziv-Lempel style adaptive dictionary because the data is typically compressed once and queried many times, so the work done to develop a better dictionary pays off.

Tuple Coding: We then concatenate all the field codes to form a bit-vector for each tuple, pad them on the right to a given length and sort the bit-vectors lexicographically. We call these bit vectors tuplecodes because each represents a tuple. After sorting, adjacent tuplecodes are subtracted to obtain a vector of deltas and each delta is further Huffman coded. By Lemma 2, we cannot save more than $\lg|R|$ bits/tuple by delta-coding, so our algorithm needs to pad tuples only to $\lg|R|$ bits (in Section 2.2.2 we describe a variation that pads tuples to more than $\lg|R|$ bits; this is needed when we don't co-code correlated columns).

The expensive step in this compression process is the sort. But it need not be perfect, as any imperfections only reduce the quality of compression. E.g., if the data is too large for an in-memory sort, we can create memory-sized sorted runs and not do a final merge; by an analysis similar to Theorem 3, we lose about $\lg x$ bits/tuple, if we have x similar sized runs.

Analysis of compression efficiency

Lemma 2 gives a lower bound on the compressibility of a general relation: $H(R) \geq m H(D) - \lg m!$, where $m = |R|$, and tuples of R are chosen i.i.d from a joint distribution D . The Huffman coding of the column values reduces the relation size to $m H(D)$ asymptotically. Lemma 1 shows that, for a multi-set of m numbers chosen uniformly from $[1, m]$, delta coding saves almost $\lg m!$ bits. But the analysis of Algorithm 3 is complicated because (a) our relation R is not such a multi-set, and (b) because of the padding we have to do in Step (1e). Still, we can show that we are within 4.3 bits/tuple of optimal compression:

Theorem 3: Our algorithm compresses a relation R of tuples chosen i.i.d from a distribution D to an expected bit size of no more than $H(R) + 4.3|R|$ bits, if $|R| > 100$.

Proof: See Appendix 8.

2.2 Relaxing Compression for Query Speed

csvzip implements the composite compression algorithm of Section 2.1.4 in order to maximally compress its input relations. But it also performs two relaxations that sacrifice some compression in return for faster querying.

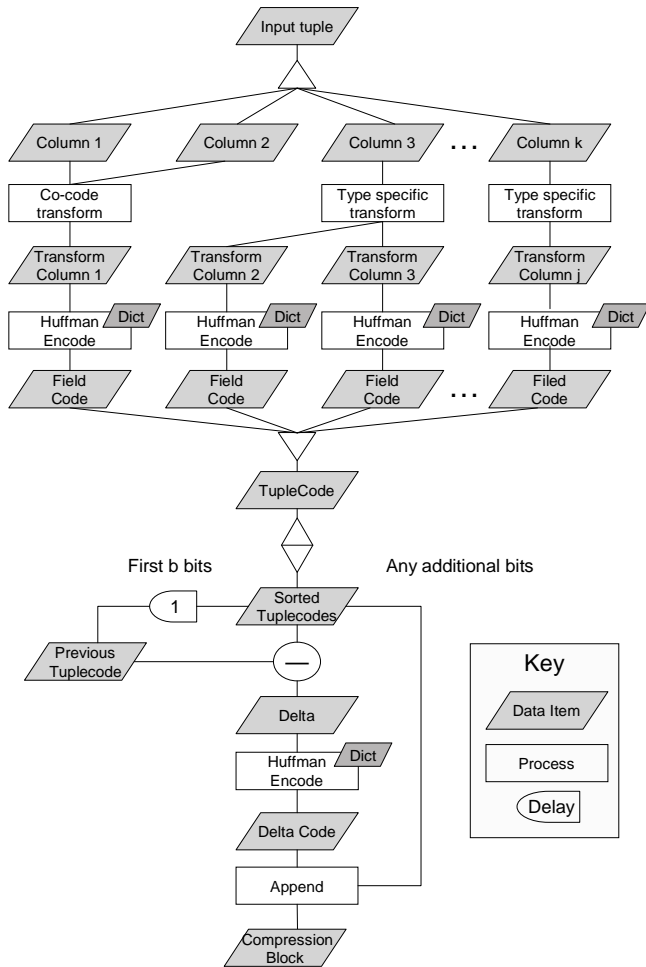


Figure 4: The Compression Process

2.2.1 Huffman coding vs. Domain coding

As we have discussed, Huffman coding can be substantially more space efficient than domain coding for skewed domains. But it does create variable length codes, which are harder to tokenize (Section 3). For some numerical domains, domain coding also allows for efficient decoding (see below). So domain coding is useful in cases where it does not lose much in space efficiency.

Consider a domain like “supplierKey integer” in a table with a few hundred suppliers. Using a 4 byte integer is obviously over-kill. But if the distribution of suppliers is roughly uniform, a 10-bit domain code may compress the column close to its entropy. Another example is “salary integer”. If salary ranges from 1000 to 500000, storing it as a 22 bit integer may be fine. The fixed length makes tokenization easy. Moreover, decoding is just a bit-shift (to go from 20 bits to a uint32). Decoding speed is crucial for aggregation columns. We use domain coding as default for key columns (like supplierKey) as well as for numerical columns on which the workload performs aggregations (salary, price, ...).

2.2.2 Tuning the sort order to obviate co-coding

In Section 2.1.3, we co-coded correlated columns to achieve greater compression. But co-coding can make querying harder.

Consider the example (partKey, price) again. We can evaluate a predicate `partKey=? AND price=?` on the co-coded values if the co-coding scheme preserves the ordering on (partKey, price). We can also evaluate standalone predicates on partKey. But we cannot evaluate a predicate on price without decoding. Co-coding also increases the dictionary sizes which can slow down decompression if the dictionaries no longer fit in cache.

We avoid co-coding such column pairs by tuning the order of the columns in the tuplecode. Notice that in Step 1d of Algorithm 3, there is no particular order in which the fields of the tuple t should be concatenated – we can choose any concatenation order, as long as we follow the same for all the tuples. Say we code partKey and price separately, but place partKey followed by price early in the concatenation order in Step 1d. After sorting, identical values of partKey will mostly appear together. Since partKey largely determines price, identical values of price will also appear close together. So the contribution of price to the delta (Step 3b) is a string of 0s most of the time. This 0-string compresses very well during the Huffman coding of the tuplecode deltas. We present experiments in Section 4 that quantify this trade-off.

3. QUERYING COMPRESSED DATA

We now turn our focus from compressing relations to running queries on the compressed relation. Our goals are to:

- Design query operators that work on compressed data,
- Determine as soon as possible that the selection criteria is not met, avoiding additional work for a tuple,
- Evaluate the operators using small working memory, by minimizing access to the full Huffman dictionaries.

3.1 Scan with Selection and Projection

Scans are the most basic operation over compressed relations, and the hardest to implement efficiently. In a regular DBMS, scan is a simple operator: it reads data pages, parses them into tuples and fields, and sends parsed tuples to other operators in the plan. Projections are usually done implicitly as part of parsing. Selections are applied just after parsing, to filter tuples early. But parsing a compressed table is more compute intensive because all tuples are concatenated together into a single bit stream. Tokenizing this stream involves: (a) undoing the delta coding to extract tuplecodes, and (b) identifying field boundaries within each tuplecode. We also want to apply predicates during the parsing itself.

Undoing the delta coding. The first tuplecode in the stream is always stored as-is. So we extract it directly, determining its end by knowing its schema and navigating the Huffman tree for each of its columns in order, as we read bits off the input stream. Subsequent tuples are delta-coded on their prefix bits (Algorithm 3). For each of these tuples, we first extract its delta-code by navigating the Huffman tree for the delta-codes. We then add the decoded delta to the running tuple prefix of the previous tuplecode to obtain the prefix of the current tuplecode. We then push this prefix back into the input stream, so that the head of the input bit stream contains the full tuplecode for the current tuple. We repeat this process till the stream is exhausted.

We make one optimization to speed decompression. Rather than coding each delta by a Huffman code based on its frequency, we Huffman code only the number of leading 0s in the delta, followed by the rest of the delta in plain-text. This “number-of-leading-0s” dictionary is often much smaller (and hence faster to lookup) than the full delta dictionary, while enabling almost the same compression, as we see experimentally (Section 4.1). Moreover, the addition of the decoded delta is faster when we code the number of leading 0s, because it can be done with a bit-shift and a 64-bit addition most of the time avoiding the use of multi-precision arithmetic.

A second optimization we are investigating is to compute deltas on the full tuplecode itself. Experimentally we have observed that this avoids the expensive push back of the decoded prefix, but it does increase the entropy (and thus Huffman code length) of the deltas, by about 1bit/tuple.

Identifying field boundaries. Once delta coding has been undone and we have reconstructed the tuplecode, we need to parse the tuplecode into *field codes*. This is challenging because there are no explicit delimiters between the field codes. The standard approach mentioned in Section 1.1 (walking the Huffman tree and exploiting the prefix code property) is too expensive because the Huffman trees are typically too large to fit in cache (number of leaf entries = number of distinct values in the column). Instead we use a new *segregated coding* scheme (Section 3.1.1).

Selecting without decompressing. We next want to evaluate selection predicates on the field codes without decoding. Equality predicates are easily applied, because the coding function is 1-to-1. But range predicates need order-preserving codes: e.g., to apply a predicate $c1 \leq c2$, we want: $code(c1) \leq code(c2)$ iff $c1 \leq c2$. However, it is well known [13] that prefix codes cannot be order-preserving without sacrificing compression efficiency. The Hu-Tucker scheme [15] is known to be the optimal order-preserving code, but even it loses about 1 bit (vs optimal) for each compressed value. Segregated coding solves this problem as well.

3.1.1 Segregated coding

For fast tokenization with order-preservation, we propose a new scheme for assigning code words in a Huffman tree (our scheme applies more generally, to any prefix code).

The standard method for constructing Huffman codes takes a list of values and their frequencies, and produces a binary tree [16]. Each value corresponds to a leaf, and codewords are assigned by labelling edges 0 or 1.

The compression efficiency is determined by the depth of each value – any tree that places values at the same depths has the same compression efficiency. Segregated coding exploits this

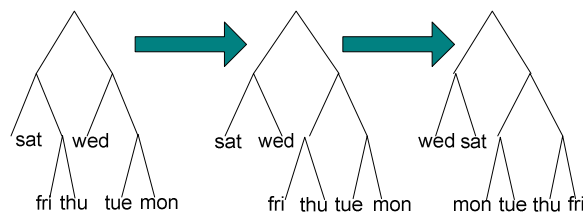


Figure 5: Segregated Huffman Coding Example

as follows. We first rearrange the tree so that leaves at smaller depth are to the left of leaves at greater depth. We then permute the values at each depth so that leaves at each depth are in increasing order of value, when viewed from left to right. Finally, we label each node’s left-edge as 0 and right-edge as 1. Figure 5 shows an example. It is easy to see that a segregated coding has two properties:

- within values of a given depth, greater values have greater codewords (e.g., $encode('tue') < encode('thu')$, in Figure 5)
- Longer codewords are numerically greater than shorter codewords (e.g., $encode('sat') < encode('mon')$, in Figure 5)

A Micro-Dictionary to tokenize Codewords

Using property 2), we can find the length of a codeword in time proportional to the log of the code length. We don’t need the full dictionary; we just search the value ranges used for each code length. We can represent this efficiently by storing the smallest codeword at each length in an array we’ll call *mincode*. Given a bit-vector b , the length of the only codeword that is contained in a prefix of b is given by $\max\{len : mincode[len] \leq b\}$, which can be evaluated efficiently using a binary or linear search, depending on the length of the array.

This array *mincode* is very small. Even if there are 15 distinct code lengths and a code can be up to 32 bits long, the *mincode* array consumes only 60 bytes, and easily fits in the L1 cache. We call it the *micro-dictionary*. We can tokenize and extract the field code using *mincode* alone.

Evaluating Range Queries using Literal Frontiers

Property 1) is weaker than full order-preservation, e.g., $encode(wed) < encode(mon)$ in Figure 5. So, to evaluate $\square < col$ on a literal \square , we cannot simply compare $encode(\square)$ with the field code. Instead we pre-compute for each literal a *list* of codewords, one at each length: $\phi(\lambda) [d] = \max\{c \text{ a code word of length } d \mid decode(c) \leq \lambda\}$

To evaluate a predicate $\lambda < col$, we first find the length l of the current field code using *mincode*. Then we check if $\phi(\lambda)[l] < field \text{ code}$. We call $\phi(\lambda)$ the *frontier* of λ . $\phi(\lambda)$ is calculated by binary search for $encode(\lambda)$ within the leaves at each depth of the Huffman tree. Although this is expensive, it is done only once per query. Notice that this only works for range predicates involving literals. Other predicates, such as $col1 < col2$ can only be evaluated on decoded values, but are less common.

3.1.2 Short circuited evaluation

Adjacent tuples processed in sorted order are very likely to have the same values for many of the initial columns. We take advantage of this during the scan by keeping track of the current value of sub-expressions used in the computations to evaluate selections, projections, and aggregations.

When processing a new tuple, we first analyze its delta code to determine the largest prefix of columns that is identical to the previous tuple. This is easy to do with our optimization of Huffman coding not the actual delta but the number of leading 0s in the delta. We do need to verify if carry-bits from the rest of the delta will propagate into the leading 0s, during the addition of the delta with the previous tuplecode. This verification is just a right-shift followed by a compare with the previous tuplecode. The shift does become expensive for large

tuplecodes; we are investigating an alternative XOR-based delta coding that doesn't generate any carries.

3.2 Layering other Query Plan operators

The previous section described how we can scan compressed tuples from a compressed table, while pushing down selections and projections. To integrate this scan into a query plan, we expose it using the typical iterator API, with one difference: getNext() returns not a tuple of values but a tuplecode – i.e., a tuple of coded column values. Most other operators, except aggregations, can be changed to operate directly on these tuplecodes. We discuss four such operators next: index-scan, hash join, sort-merge join, and group-by with aggregation.

3.2.1 Index Scan: Access Row by RID

Index scans take a bounding predicate, search through an index structure for matching row ids (RIDs), load the corresponding data pages, and extract matching records. The process of mapping predicates to RIDs occurs as usual. But extracting the matching records is trickier because it involves random access within the table. Since we delta-code tuples, the natural way to tokenize a table into tuples is to scan them sequentially, as in Section 3.1.

Our solution is to punctuate a compressed table with periodic non-delta-coded tuples (the fields in these tuples are still Huffman coded). This divides the table into separately decodable pieces, called *compression blocks (cblocks)*. We make each rid be a pair of cblock-id and index within cblock, so that index-based access involves sequential scan within the cblock only. Thus, short cblocks mean fast index access. However, since the first tuple is not delta-coded, short cblocks also mean less compression. In practice this is not a problem: A Huffman-coded tuple takes only 10-20 bytes for typical schemas (Section 4.1), so even with a cblock size of 1KB, the loss in compression is only about 1%. 1KB fits in L1-cache on many processors, so sequential scan in a cblock is fine.

3.2.2 Hash Join & Group By with Aggregation

Huffman coding assigns a distinct field code to each value. So we can compute hash values on the field codes themselves without decoding. If two tuples have matching join column values, they must hash to the same bucket. Within the hash bucket, the equi-join predicate can also be evaluated directly on the field codes.

One important optimization is to delta-code the input tuples

as they are entered into the hash buckets (a sort is not needed here because the input stream is sorted). The advantage is that hash buckets are now compressed more tightly so even larger relations can be joined using in-memory hash tables (the effect of delta coding will be reduced because of the smaller number of rows in each bucket).

Grouping tuples by a column value can be done directly using the code words, because checking whether a tuple falls into a group is simply an equality comparison. However aggregations are harder.

COUNT, COUNT DISTINCT, can be computed directly on code words: to check for distinctness of values we check distinctness of the corresponding codewords. MAX and MIN computation involves comparison between code words. Since our coding scheme preserves order only within code words of the same length, we need to maintain the current maximum or minimum separately on code words of each length. After scanning through the entire input, we evaluate the overall max or min by decoding the current code words of each codeword length and computing the maximum of those values.

SUM, AVG, STDEV, cannot be computed on the code words directly; we need to decode first. Decoding domain-coded integer columns is just a bit-shift. Decoding Huffman codes from small domains or ones with large skew is also cheap, because the Huffman tree is shallow. We also place columns that need to be decoded early in the column ordering, to improve the chance that the scanner will see runs of identical codes, and benefit from short circuited evaluation.

3.2.3 Sort Merge Join

The principal comparisons operations that a sort merge join performs on its inputs are < and =. Superficially, it would appear that we cannot do sort merge join without decoding the join column, because we do not preserve order across code words of different lengths. But in fact, sort merge join does not need to compare tuples on the traditional '<' operator – any total ordering will do. In particular, the ordering we have chosen for codewords – ordered by codeword length first and then within each length by the natural ordering of the values is a total order. So we can do sort merge join directly on the coded join columns, without decoding them first.

4. EXPERIMENTAL STUDY

We have implemented our compression method in a prototype system called csvzip. csvzip currently supports the compression

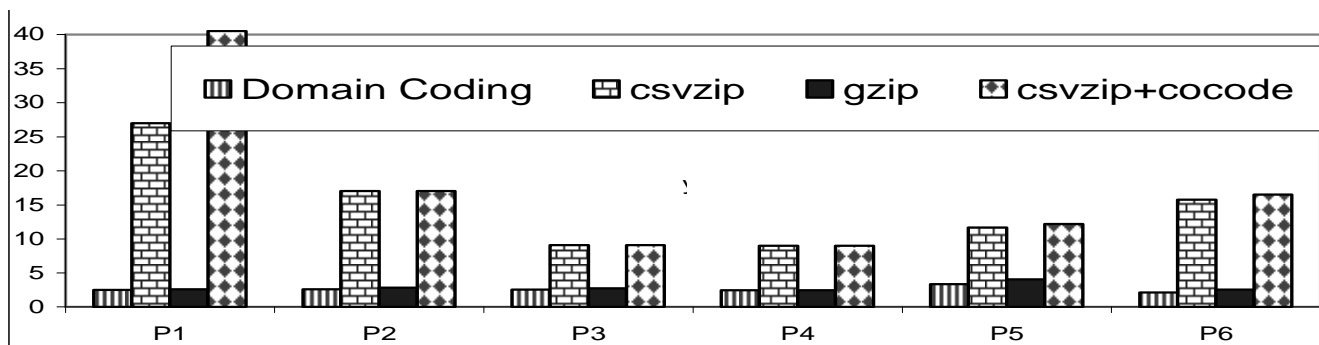


Figure 7: Comparing the compression ratios of four compression methods on 6 datasets

DATASET SCHEMA	Original size	DC-1	DC-8	Huffman (1)	csvzip (2)	Delta code savings (1)-(2)	Huffman + Cocode (3)	Correlation saving (1)-(3)	csvzip+cocode (5)	Loss by not Cocoding (2)-(5)	Gzip
P1. LPK LPR LSK LQTY	192	76	88	76	7.17	68.83	36	40	4.74	2.43	73.56
P2. LOK LQTY	96	37	40	37	5.64	31.36	37	0	5.64	0	33.92
P3. LOK LQTY <u>LODATE</u>	160	62	80	48.97	17.60	31.37	48.65	0.32	17.60	0	58.24
P4. LPK <u>SNAT</u> <u>LODATE</u> <u>CNAT</u>	160	65	80	49.54	17.77	31.77	49.15	0.39	17.77	0	65.53
P5. <u>LODATE</u> <u>LSDATE</u> <u>LRDATE</u> LQTY LOK	288	86	112	72.97	24.67	48.3	54.65	18.32	23.60	1.07	70.50
P6. OCK <u>CNAT</u> <u>LODATE</u>	128	59	72	44.69	8.13	36.56	39.65	5.04	7.76	0.37	49.66
P7. SAP SEOCOMPODF	548	165	392	79	47	32	58	21	33	14	52
P8. TPC-E CUSTOMER	198	54	96	47	30	17	44	3	23	7	69

Table 6: Overall compression results on various datasets (all sizes are in # bits/tuple). DC-1 and DC-8 are bit and byte aligned domain coding. Csvzip is Huffman followed by delta coding; csvzip+cocode is cocode followed by Huffman and then delta-coding.

Underscore indicates skew and *italics* denotes correlation. Column abbreviations are: LPK: partkey, LPR: extendedprice, LSK: supkey, LQTY: quantity, LOK: orderkey, LODATE: orderdate, SNAT: supNation, CNAT: custNation, LSDATE: shipDate, LRDATE: receiptDate. TPC-E schema is tier, country_1, country_2, country_3, area_1, first name, gender, middle initial, last name.

of relations loaded from comma separated value (csv) files, and table scan with selection, projection, and aggregation. Updates and joins are not implemented at this time. We do not currently have a SQL parser – we execute queries by writing C programs that compose select, project, and aggregate primitives.

We have used this prototype to perform an experimental study of both compression efficiency and scan efficiency. Our goal is to quantify the following:

- 1) How much can we compress, as compared to row coding or domain coding? What is the relative payoff of skew, order-freeness, and correlation (Section 4.1)?
- 2) How efficiently can we run scan queries directly on these compressed tables (Section 4.2)?

We use three datasets in our experiments:

TPC-H views: We choose a variety of projections of Lineitem x Orders x Part x Customer, that are appropriate for answering TPC-H queries (Table 6). Our physical design philosophy, like C-Store, is to have a number of highly compressed materialized views appropriate for the query workload.

TPC-E Customer: We tested using 648,721 records of randomly generated data produced per the TPC-E specification. This file contains many skewed data columns but little correlation other than gender being predicted by first name.

SAP/R3 SEOCOMPODF: We tested using projections of a table from SAP having 50 columns and 236,213 rows. There is a lot of correlation between the columns, causing the delta code savings to be much larger than usual.

One drawback with TPC-H is that uses uniform, independent value distributions, which is utterly unrealistic (and prevents us from showing off our segregated Huffman Coding ☺). So we altered the data generator to generate two skewed columns and 2 kinds of correlation:

- Dates: We chose 99% of dates to be in 1995-2005, with 99% of that on weekdays, 40% of that on two weeks each before New Year & Mothers' Day (20 days / yr).
- c_nationkey, s_nationkey: We chose nation distributions from WTO statistics on international trade (www.wto.org/english/res_e/statis_e/wt_overview_e.htm).
- Soft Functional Dependency: We made l_extendedprice be functionally dependent on l_partkey

- Arithmetic Correlation: We made l_receiptdate and l_shipdate be uniformly distributed in the 7 days after the corresponding o_orderdate.

There are two other correlations inherent in the schema:

- For a given l_partkey, l_supkey is restricted to be one of 4 possible values.
- De-Normalization: Dataset P6 in Table 6 is on lineitem x order x customer x nation, and contains a non-key dependency o_custkey → c_nationkey

We use a 1 TB scale TPC-H dataset (≈6B rows in each of our datasets). To make our experiments manageable, we did not actually generate, sort, and delta-code this full dataset – rather we tuned the data generator to only generate 1M row slices of it, and compressed these. For example, P2 of Table 6 is delta-coded by sorting on <l_orderkey, l_qty>. We generate 1M row slices of P2 by modifying the generator to filter rows where l_orderkey is not in the desired 1M row range.

4.1 Extent of Compression

Table 6 lists the detailed compression results on each dataset. Figure 7 plots the overall compression ratio obtained by csvzip (with and without co-coding) vs that obtained by:

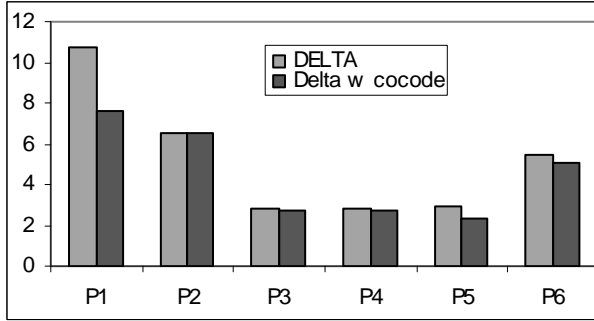
- a plain gzip (representing the ideal performance of row and page level coders),
- a fixed length domain coder aligned at bit boundaries (representing the performance of column coders). Table 6 also lists the numbers for domain coding at byte boundaries; it is significantly worse.

Note that all ratios are with respect to the size of the vertical partition, not size of the original tables – any savings obtained by not storing unused columns is orthogonal to compression.

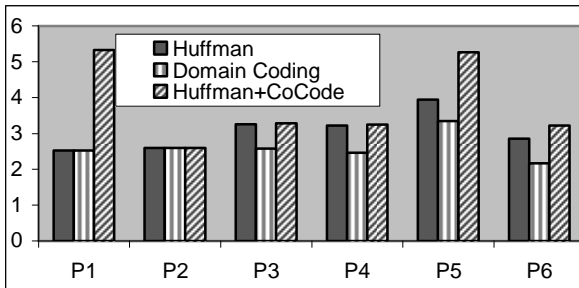
Even without co-coding, csvzip consistently gets 10x or more compression, in contrast to 2-3x obtained by gzip and domain coding. On some datasets such as LPK LPR LSK LQTY, our compression ratio without co-coding is as high as 192/7.17≈27; with co-coding it is 192/4.74≈41. From Table 6, notice that the absolute per tuple sizes after compression are typically 10-20bits.

We next analyze from where our substantial gains arise.

Exploitation of Skew: The first source of compression is skew. Observe the bars labelled Huffman and Domain Coding



in the chart below. They show the compression ratio achieved just by Huffman coding column values, in contrast to the savings obtained by domain coding. All columns except nationkeys and dates are uniform, so Huffman and domain coding are identical for P1 and P2. But for the skewed domains the savings is significant (e.g., 44.7 bits vs. 59 bits for P6). Table 6 lists the full results, including compression results on SAP and TPC-E datasets. These have significant skew (several char columns with few distinct values), so Huffman coding does very well. On SAP, Huffman coding compresses to 79 bits/tuple vs. 165 bits/tuple for domain coding.



Correlation: The best way to exploit correlation is by co-coding. Table 6 lists the extra benefit obtained by correlation, over that obtained by just Huffman coding individual columns. This is also illustrated in the bar labelled “Huffman+CoCode” in the above plot. For example, we go down from 72.97 bits/tuple to 54.65 bits per tuple for P5. In terms of compression ratios, as a function of the original table size, we compress 2.6x to 5.3x by exploiting correlation and skew.

Delta-Coding: The last source of redundancy is from lack-of-ordering. In Table 6, column(3) – column (5) gives the savings from delta coding when we co-code. Observe that it is almost always about 30 bits/tuple for all the TPC-H datasets. This observation is consistent with Theorem 3 and Lemma 2; $\lg m = \lg(6.5B) \approx 32.5$, and we save a little fewer bits per tuple by delta-coding.

	S1	S2	S3
Q1: select sum(lpr) from S1/2/3	8.4	10.1	15.4
Q2: Q1 where lsk>?	8.1-10.2	8.7-11.5	17.7-19.6
Q3: Q1 where oprio>?		10.2-18.3	17.8-20.2
Q4: Q1 where oprio=?		11.7-15.6	20.6-22.7

Again in Table 6, column (1) – column (2) gives the savings from delta coding when we don’t co-code correlated

columns. Notice that the savings is now higher – the delta coding is able to exploit correlations also, as we discussed in Section 2.2.2.

The plot on the opposite column illustrates the compression ratios obtained with the two forms of delta coding. The ratio is as high as 10 times for small schemas like P1. The highest overall compression ratios result when the length of a tuplecode and bits per tuple saved by delta coding are similar.

Exploiting Correlation via co-coding vs. delta-coding:

The extent to which it can exploit correlations is indicated by the column labelled (2)-(5) in Table 6 – notice that we are often close to the co-coding performance.

This strategy is powerful, because by not co-coding range queries on the dependent become much easier as we discussed in Section 2.2.2. But a drawback is that the correlated columns have to be placed early in the sort order – for example, LODATE, LRDATE, LSDATE in dataset P5. We have experimented with a pathological sort order – where the correlated columns are placed at the end. When we sort P5 by (LOK, LQTY, LODATE, ...), the average compressed tuple size increases by 16.9 bits. The total savings from correlation is only 18.32 bits, so we lose most of it. This suggests that we need to do further investigation on efficient range queries over co-coded or dependent coded columns.

4.2 Querying Compressed Relations

We now investigate efficient querying of compressed relations. We focus on scans, with selection, projections, and aggregations. Our queries test three aspects of operating over compressed data:

1. How efficiently can we undo the delta-coding to retrieve tuplecodes?
2. How efficiently can we tokenize tuples that contain Huffman-coded columns?
3. How well can we apply equality and range predicates on Huffman coded and domain coded columns?

The first is the basic penalty for dealing with compressed data, that every query has to pay. The second measures the effectiveness of our segregated coding. The third measures the ability to apply range predicates using literal frontiers.

We run scans against 3 TPC-H schemas: (S1: LPR LPK LSK LQTY) has only domain coded columns, (S2: LPR LPK LSK LQTY OSTATUS OCLK) has 1 Huffman coded column (OSTATUS), and (S3: LPR LPK LSK LQTY OSTATUS OPRIO OCLK) has 2 Huffman coded columns (OSTATUS,OPRIO). OSTATUS has a Huffman dictionary with 2 distinct codeword lengths, and OPRIO has a dictionary with 3 distinct codeword lengths. The table below plots the scan bandwidth (in nanoseconds/tuple) for various kinds of scans against these schemas. All experiments were run on a 1.2GHz Power 4, on data that fit entirely in memory.

Q1 is a straightforward scan plus aggregation on a domain coded column. On S1, which has no Huffman columns, Q1 just tests the speed of undoing the delta code – we get 8.4ns/tuple.

On S2 and S3, it tests the ability to tokenize Huffman coded columns using the micro-dictionary (in order to skip over them). Tokenizing the first Huffman coded column is relatively

cheap because its offset is fixed; subsequent ones incur an overhead of about 5ns/tuple to navigate the micro-dictionary. If a tuple has several Huffman coded columns, this suggests that it will pay to encode the tuple length in front.

Q2, Q3, and Q4 are queries with predicates plus aggregation. A range of numbers is reported for each schema because the run time depends on the predicate selectivity, due to short-circuiting. For small selectivities, the predicate adds at most a couple of ns/tuple beyond the time to tokenize. This is in line with our expectation, because once tokenized, the range predicate is evaluated directly on the codeword.

These numbers indicate we can process 50-100M tuples/s using a single processor (equivalent to 1-2GB/s), and are quite promising. Obviously, a full query processor needs much more than scan. Nevertheless, as we discussed in Section 3.2, hash joins and merge joins can be done on the compressed tuples using these basic equality and range predicate operations.

5. CONCLUSIONS AND FUTURE WORK

Compression is a promising approach to deal with the data movement bottleneck. We have developed a notion of entropy for relations and described a method to compress relations to within 4.3 bits/tuple of the entropy. This results in up to a compression factor of up to 40 on TPC-H data, much better than reported in prior literature. We have also developed techniques for efficient scans with selection and projection over relations compressed with this method.

Much work remains to exploit the promise of this idea. Although we have shown the basic feasibility, we need to further investigate the engineering issues involved in doing query operators both as a part of a commercial DBMS and as a utility, and figuring out how to utilize the 128 bit registers and hardware threading available on modern processors. A second direction is lossy compression, which we believe is vital for efficient aggregates over compressed data. Finally, we need to support incremental updates. We believe that many of the standard warehousing ideas like keeping change logs and periodic merging will work here as well.

6. REFERENCES

- [1] Bala Iyer., David Wilhite. Data Compression Support in Data Bases. In VLDB 1994.
- [2] G. Antoshenkov, D. Lomet, J. Murray. Order Preserving String Compression. In ICDE 1996.
- [3] T. Cover, J. Thomas. Elements of Information Theory. John Wiley, 1991
- [4] Eric W. Weisstein et al. "Catalan Number." In MathWorld. <http://mathworld.wolfram.com/CatalanNumber.html>.
- [5] A. Ailamaki, D.J. DeWitt, M. D. Hill, M. Skounakis Weaving relations for cache performance. In VLDB 2001.
- [6] M. Stonebraker et al. C-Store: A Column Oriented DBMS. In VLDB 2005.
- [7] S. Babu et al. SPARTAN: A model-based semantic compression system for massive tables. SIGMOD 2001.
- [8] J. Goldstein, R. Ramakrishnan, U. Shaft. Compressing Relations and Indexes. In ICDE 1998.
- [9] G.V. Cormack and R.N. Horspool. Data Compression using Dynamic Markov Modelling, Computer Journal 1987.
- [10] G. V. Cormack, Data Compression In a Database System, Comm. of the ACM 28(12), 1985
- [11] G. Copeland and S. Khoshafian. A decomposition storage model. In SIGMOD 1985.
- [12] Sybase IQ. www.sybase.com/products/informationmanagement
- [13] D. Knuth. The Art of Computer Programming. Addison Wesley, 1998.
- [14] S. Barnard and J. M. Child. Higher Algebra, Macmillan India Ltd., 1994.
- [15] T. C. Hu, A. C. Tucker, Optimal computer search trees and variable-length alphabetic cods, SIAM J. Appl. Math, 1971.
- [16] Huffman, D. A method for the construction of minimum redundancy codes. Proc. I.R.E. 40. 9. 1952
- [17] Jim Gray. Commentary on 2005 Datamation benchmark. <http://research.microsoft.com/barc/SortBenchmark>
- [18] A. Zandi et al. Sort Order Preserving Data Compression for Extended Alphabets. Data Compression Conference 1993.
- [19] K. Bharat et al. The Connectivity server: Fast access to Linkage information on the web. In WWW 1998.
- [20] G. Graefe and L. Shapiro. Data Compression and Database Performance. In Symp on Applied Computing, 1991.
- [21] M. Poess et al. Data compression in Oracle. In VLDB 2003.

7. ENTROPY OF MULTI-SET DELTAS

Lemma 1: For a multi-set R of m values picked uniformly and i.i.d from $[1..m]$, for $m > 100$, the value distribution of $\text{code}(R)$ has entropy $\leq 2.67 m$ bits.

Proof: Recall the pseudo-code for computing the coded delta:

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

Now consider this situation: we have thrown m balls numbered $1..m$ uniformly and i.i.d onto m numbered bins arranged in a circle. Define a random variable D_j for ball j that landed in bin i

$$D_j \equiv 0 \text{ if } j \text{ is not the highest numbered ball in bin } i \\ \equiv k \text{ if } j \text{ is the highest numbered ball bin } i \text{ and}$$

and k is the clockwise distance from bin i to the next occupied bin. In effect, D_j is the random variable for the bin gap between adjacent balls. The distribution of each code in $\text{code}(R)$ is that of D_j . By symmetry, this distribution is independent of j , so we shall call it D . Our goal is to compute $H(D)$.

Now, let $p_d \equiv \Pr(D = d)$, $0 \leq d < m$ and define

$$\lambda_d \equiv \Pr(D = d \mid D \neq 0), 1 \leq d < m$$

So, $p_d = (1 - p_0) \lambda_d$, and

$$H(D) = \sum_{0 \leq d < m} p_d \lg(1/p_d) = -p_0 \lg p_0 - \sum_{1 \leq d \leq m-1} (1-p_0) \lambda_d \lg(1-p_0) \lambda_d.$$

Since $\sum_{0 \leq d < m} p_d = 1 = p_0 + \sum_{1 \leq d < m} (1-p_0) \lambda_d$, we have,

$$H(D) = -p_0 \lg p_0 - (1-p_0) \lg(1-p_0) + (1-p_0) \sum_{1 \leq d < m} \lambda_d \lg(1/\lambda_d) \quad (1)$$

Consider $\alpha_d \equiv \Pr(D > d \mid D \neq 0)$, $1 \leq d < m$. This is the probability that, given that a ball is the highest numbered ball in a bin ($D \neq 0$), that there is a clockwise gap greater than d to the next occupied bin, i.e., that all $m - 1$ other balls fall into the other $m - d$ bins. So,

$$\alpha_d = (1 - d/m)^{m-1}, \text{ for } 1 \leq d < m,$$

$$\lambda_d = \Pr(D = d \mid D \neq 0) = \alpha_{d-1} - \alpha_d,$$

$$\alpha_d = (1 - d/m)^{m-1} = [m/(m-d)] [(1 - d/m)^m] < [m/(m-d)] e^{-d},$$

by standard exponential inequalities [14], and

$$\lambda_d = \alpha_{d-1} - \alpha_d < \alpha_{d-1} < [m/(m-d+1)] e^{1-d} \quad (2)$$

But, $\lambda_d = \alpha_{d-1} - \alpha_d = (1 - (d-1)/m)^{m-1} - (1-d/m)^{m-1}$

$$\lambda_d (m/(m-d))^{m-1} = [(m-d+1)/(m-d)]^{m-1} - 1 > 1 \text{ for } d \geq 1$$

$$\text{So } 1/\lambda_d < [m/(m-d)]^{m-1} \quad (3)$$

Now, each occurrence of a zero delta corresponds to an empty bin, so expected number of empty bins is mp_0 . But empty bins form a binomial distribution with probability $(1-1/m)^m$. So $p_0 = (1-1/m)^m$

$< 1/e$. Now, by simple calculus, $-x \lg x$ has its maximum value when $x=1/e$, and substituting into equation (1) we have

$$-p_0 \lg p_0 - (1-p_0) \lg(1-p_0) < (\lg e)/e + (1-1/e) \lg(1/(1-1/e)) \quad (4)$$

Combining (1), (2), (3), (4) we get,

$$H(D) < (\lg e)/e + (1-1/e) \lg(e/(e-1)) + (1-p_0) \lambda_1 \lg 1/\lambda_1 + (1-p_0) \sum_{2 \leq d \leq m-1} [m/(m-d+1)] e^{1-d} (m-1) \lg(1+d/(m-d))$$

As $\ln(1+x) < x$, $\lg(1+x) = \ln(1+x) / \ln 2 < x / \ln 2$ we have

$$H(D) < (\lg e)/e + (1-1/e) \lg(e/(e-1)) + (1-p_0) \lambda_1 \lg 1/\lambda_1 + ((1-p_0)/\ln 2) \sum_{2 \leq d \leq m-1} [dm(m-1)/((m-d)(m-d+1))] e^{1-d} \quad (5)$$

$$\text{Let } Y_m \equiv \sum_{d=2}^{m-1} [dm(m-1)/(m-d)(m-d+1)] e^{1-d}$$

$$Y_m - Y_{m+1} = \sum_{d=2}^{m-1} [2dm(d-1)/(m-d)(m-d+1)(m-d+2)] e^{1-d} - m^2(m+1)e^{-m+1}/2$$

$$= \sum_{d=2}^{m-2} [2dm(d-1)/(m-d)(m-d+1)(m-d+2)] e^{1-d} + me^{1-m} [m^2(e/3-0.5) + m(e+0.5) + 2e/3]$$

$$> 0 \text{ for } m > 7 \text{ (the last quadratic is } > 0 \text{ for } m > 7)$$

Hence $Y_m > Y_{m+1}$ for $m > 7$.

But $Y_{100} < 1.57$, by calculation. So, for $m > 100$, $Y_m < 1.57$. (6)

Also, $\lambda_1 = 1 - (1-1/m)^{m-1}$ is increasing in m . Since $x \lg(1/x)$ is maximum at $1/e$, $\lambda_1 \lg(1/\lambda_1) < 0.63 \lg 0.63 < 0.42$ for $m > 100$. (7)

Plugging (6), (7) into (5) and doing some algebra, $H(D) < \lg e - (1-1/e) \lg(e-1) + 0.42(1-p_0) + 1.57(1-p_0)/\ln 2$.

By calculus, $p_0 = (1-1/m)^m > 0.99^{100} > 0.36$ for $m > 100$.

Substituting, we get $H(D) < 2.67$ bits.

8. OPTIMALITY OF ALGORITHM 3

Lemma 3: Let X be a random variable over a distribution D , and let $\text{code}(X)$ be a random variable representing the codeword for X , according to some compression scheme. Let $\text{code}_\alpha(X)$ be the random variable for the α -bit prefix of $\text{code}(X)$. Then, if the compression scheme is optimal and given that $\text{code}(X)$ has $\geq \alpha$ bits, $\text{code}_\alpha(X)$ is uniformly distributed over all bit-vectors of length α .

Proof: If the distribution of bits were not uniform, we could further compress each α -bit prefix, contradicting the assumption that the compression scheme is optimal. \square

We now show the algorithm of Section 2.1.4 is near optimal.

Lemma 4: If tuples of R are generated i.i.d. over a distribution D , the algorithm of Figure 3 encodes R in at most $m(1+H(D)) - (m-1)(\lg m - 2.67) + \sum_{i \in D_s} m p_i (\lfloor \lg m \rfloor - b_i)$ bits on average, where $m = |R| > 100$, D_s is the subset of tuples that get coded in less than $\lfloor \lg m \rfloor$ bits at the end of step 1d, p_i is the occurrence probability of tuple $i \in D_s$, and b_i is the length to which tuple i is coded at end of step 1d.

Proof: By standard results on Huffman coding [3], the expected size of the tuples at the end of Step 1d is no more than $m(1+H(D))$ bits. The padding of Step 1e adds $(\lfloor \lg m \rfloor - b_i)$ random bits for each tuple that codes to $b_i < \lfloor \lg m \rfloor$ bits. So the overall padding has expected length of $\sum_{i \in D_s} m p_i (\lfloor \lg m \rfloor - b_i)$ bits. By

Lemma 3, after Step 1e, the $\lfloor \lg m \rfloor$ -bit prefix of the coded tuple is uniformly distributed over all bit vectors of length $\lfloor \lg m \rfloor$. So, by Lemma 1, average length of the coded delta is at most 2.67 bits. So the delta coding of Steps 2 and 3 saves on average $\lfloor \lg m \rfloor - 2.67$ bits for each pair of adjacent tuples: $(m-1)(\lfloor \lg m \rfloor - 2.67)$ bits overall. And by summing these quantities we get our result. \square

Theorem 3: Algorithm 3 compresses a relation R of tuples chosen i.i.d from a finite distribution D to at most $H(R) + 4.3|R|$ bits, assuming $|R| > 100$.

Proof: Consider m items chosen i.i.d. from D . Let \mathbf{R} be a random variable for these m items viewed as a sequence and R be a random variable for these m items viewed as a multi-set. Without loss of generality let the distribution D be given by a set of n values v_1, \dots, v_n and associated probabilities p_1, \dots, p_n , so that a sample drawn from D has value v_i with probability p_i . Let D_s be the subset of those values that code to less than $\lfloor \lg m \rfloor$ bits after step 1d. Let \mathbf{MUL} be a random variable for the vector of multiplicities of elements in D_s in the random multi-set R , $\text{Dom}(\mathbf{MUL}) = \mathcal{X}^{|D_s|}$. Use $\boldsymbol{\mu}$ to represent a particular multiplicity in $\text{Dom}(\mathbf{MUL})$, μ_i to denote the i 'th element of $\boldsymbol{\mu}$, the multiplicity of the associated v_i , and $p_{\boldsymbol{\mu}}$ to denote the probability that $\mathbf{MUL} = \boldsymbol{\mu}$.

$$H(\mathbf{R}) = H(\mathbf{MUL}) + H(\mathbf{R} | \mathbf{MUL})$$

$$= H(\mathbf{MUL}) + \sum_{\boldsymbol{\mu} \in \text{Dom}(\mathbf{MUL})} p_{\boldsymbol{\mu}} H(\mathbf{R} | \mathbf{MUL} = \boldsymbol{\mu})$$

We now employ an argument similar to Lemma 3. For any value of the multi-set R having multiplicity $\boldsymbol{\mu} = (\mu_1, \mu_2, \dots, \mu_{|D_s|})$, there are at most $m! / (\mu_1! \mu_2! \dots \mu_{|D_s|}!)$ instances of \mathbf{R} . So,

$$H(\mathbf{R} | \mathbf{MUL} = \boldsymbol{\mu}) \geq H(\mathbf{R} | \mathbf{MUL} = \boldsymbol{\mu}) - \lg(m! / (\mu_1! \dots \mu_{|D_s|}!))$$

$$H(\mathbf{R}) \geq H(\mathbf{MUL}) + H(\mathbf{R} | \mathbf{MUL} = \boldsymbol{\mu}) - \sum_{\boldsymbol{\mu}} p_{\boldsymbol{\mu}} \lg(m! / (\mu_1! \dots \mu_{|D_s|}!))$$

$$= H(\mathbf{R}) - \sum_{\boldsymbol{\mu} \in \text{Dom}(\mathbf{MUL})} p_{\boldsymbol{\mu}} \lg(m! / (\mu_1! \dots \mu_{|D_s|}!))$$

Let $k = \mu_1 + \dots + \mu_{|D_s|}$. Then, $k! \mu_1^{\mu_1} \dots \mu_{|D_s|}^{\mu_{|D_s|}} / (\mu_1! \dots \mu_{|D_s|}!)$ is just one term in the multinomial expansion of k^k . So $k! / (\mu_1! \dots \mu_{|D_s|}!) < k^k / \mu_1^{\mu_1} \dots \mu_{|D_s|}^{\mu_{|D_s|}}$. Clearly $k < m$, so

$$\lg(m! / (\mu_1! \dots \mu_{|D_s|}!)) < m \lg m - \sum_{1 \leq i \leq |D_s|} \mu_i \lg \mu_i$$

$$\text{So, } H(\mathbf{R}) \geq H(\mathbf{R}) - \sum_{\boldsymbol{\mu} \in \text{Dom}(\mathbf{MUL})} p_{\boldsymbol{\mu}} (m \lg m - \sum_i \mu_i \lg \mu_i)$$

But, the last term can be simplified and replaced with an expectation giving us

$$H(\mathbf{R}) \geq H(\mathbf{R}) - m \lg m + E(\sum_i \mu_i \lg \mu_i)$$

$$= H(\mathbf{R}) - m \lg m + \sum_i E(\mu_i \lg \mu_i)$$

Since $x \lg x$ is convex, we can apply Jensen's inequality: $E(f(X)) \geq f(E(X))$ for convex functions f , yielding

$$H(\mathbf{R}) \geq H(\mathbf{R}) - m \lg m + \sum_i E(\mu_i) \lg E(\mu_i) \quad (1)$$

By Lemma 4, Algorithm A encodes R in space

$$m(1+H(D)) - (m-1)(\lg m - 2.67) + \sum_{i \in D_s} m p_i (\lfloor \lg m \rfloor - b_i) \quad (2)$$

Since tuples are chosen i.i.d. $H(\mathbf{R}) = mH(D)$. Recall that p_i is the probability of the i 'th element of D_s and b_i is the length of the code for the i 'th element. So $E(\mu_i) = m p_i$. Thus, the space overhead of Algorithm A over entropy = (2) - (1)

$$\leq 3.67m + \lg m + m \sum_{i \in D_s} m p_i (\lfloor \lg m \rfloor - b_i - \lg m p_i)$$

$$\leq 3.67m + \lg m + m \sum_{i \in D_s} m p_i (\lg(1/p_i) - b_i)$$

Let $p = \sum_{i \in D_s} p_i$ and $q_i = p_i / p$. Then the overhead \leq

$$3.67m + \lg m + m p \sum_{i \in D_s} q_i (\lg(2^{-b_i} / q_i) - \lg p)$$

Since $\sum_{i \in D_s} q_i = 1$, and $\lg(x)$ is concave, by Jensen's inequality,

$$\sum_{i \in D_s} q_i \lg(2^{-b_i} / q_i) \geq \lg \sum_{i \in D_s} q_i (2^{-b_i} / q_i)$$

$$\text{So overhead } \leq 3.67m + \lg m + m \lg \sum_{i \in D_s} 2^{-b_i} + m p \lg(1/p)$$

Now, for any unambiguous code, Kraft's inequality [4] says that $\sum_{i \in D} 2^{-b_i} \leq 1$. So overhead $\leq 3.67m + \lg m + m p \lg(1/p)$.

By simple calculus, $x \lg(1/x) \leq (1/e) \lg e \approx 0.53$. So,

space taken by Algorithm A - $H(\mathbf{R})$ is at most

$$4.2m + \lg m \text{ bits } < 4.3m \text{ bits } (\lg x/x < 0.07 \text{ for } m > 100) \quad \square$$