Lexicographic Encoding of Numeric Data Fields

Naphtali Rishe

School of Computer Science
Florida International University —
The State University of Florida at Miami
University Park, Miami, FL 33199

ABSTRACT

This paper proposes a method of variable-radix representation of numeric data. The method allows compact representation of arbitrary numbers. Among its properties is that bitwise lexicographic comparison (">", "<") is consistent with correct numeric comparison of numbers.

Keywords: numeric data fields, number encoding, comparison operations, databases, file structures, compactness of data, data independency, formats, floating point, variable-length data fields, real numbers, data compression.

1. Introduction

Many applications require compact variable-length representations of arbitrary numbers. Among such applications are advanced database management systems which make the data formats transparent to the user and should allow a logical field to hold numbers of unpredictably large or small magnitude and unpredictable varying precision, if the user's logic warrants this. I shall call this requirement "unboundness". The typical representations of numbers, such as floating point or fixed point formats, fail the "unboundness" requirement. A representation which satisfies this requirement is the standard mathematical notation (on paper) using arabic numerals and exponent or its imitation in computer printouts (e.g., -3.57E-101).

A further requirement on the number representation is the efficiency of the application's typical operations. The predominant operations performed on stored numbers in databases and many other applications are comparisons (=, >, *etc.*), rather than the arithmetic operations (*i.e.*, +, ×, *etc.*, which are more typical for applications involving engineering calculations). Consider, for example, a search for a record with a given key value in an index-sequential or B-tree file, or in associative memory or storage device. The efficiency of such applications would benefit if numbers could be handled as character strings. For example, in most cases the result of a comparison of two long character strings can be found by comparing their short prefixes, while the whole strings need not be scanned or even retrieved from the storage devices. This would also simplify the hardware and lower-level software since they would not have to distinguish between numbers and character strings, i.e. they would store and compare numbers in the same way they work for character strings. I shall call this requirement "lexicographic comparability".

This paper proposes an encoding of numbers which is unbound, lexicographically comparable, and compact. The properties of the encoding are fully defined in the following section.

2. Specification of Requirements

The encoding of numbers proposed in this paper satisfies the following requirements:

1. Bitwise lexicographic comparison of the encodings will coincide with the meaningful comparison of numbers, *i.e.* the order of the real numbers. This is essential for fast search of sorted and indexed files containing character strings and numeric data. Thus, if n_1 is encoded by a byte string $b_1^1 b_2^1 b_3^1$ and n_2 is encoded by a byte string $b_1^1 b_2^2 b_3^2 b_4^2$, where $b_2^1 > b_2^2$, then $n_1 > n_2$. The standard representations of numbers do not allow bitwise comparison. (Consider, for example, the representation of floating point numbers by mantissa and exponent.)
2. There is no limit on arbitrarily large, arbitrarily small, or arbitrarily precise numbers. In a database or a file we wish to be able to compare and store in a uniform format integers, real numbers, numbers with very many significant digits, and numbers with just a few significant digits. We do not wish to set a limit on the range of the data at the time of the design of the file formats. For example, the number π truncated after the first 1000 digits is a very precise number of 1000 significant digits. The number 10^{100} is large, but not precise — it has only one significant digit. We need one common format convention to represent both numbers.
3. Every number bears its own precision, *i.e.* the preci-

This research has been supported in part by a grant from Florida High Technology and Industry Council

sion is not uniform. (In a database, the varying precision will allow us to treat integers, reals, and values of different attributes with different precisions, in a uniform way in one file in the database.)

4. The encodings are of varying length and are about maximally space efficient with respect to their informational content. For example, consider the following three numbers: 3,000,000 with precision 500,000; integer 5 with precision 0.5; 0.000,000,000,000,000,7 with precision 0.000,000,000,000,000,05. Each of those three numbers should require only a few bits each, while the number 12345678.90 should require many more bits. The number of bits in a number's representation should be approximately equal to the amount of information in that number.
5. No additional byte(s) are required to store the length of the encoded representation or to delimit its end: the representation should contain enough information within itself so that the decoder would know where the representation of one number ends and where that of the next number begins (within the same record in the file.) The absence of delimiters gives an additional saving in space, and also facilitates handling of records.
6. The representation of numbers is one-to-one. For example, there should not be several representations for 0, like 0.00, -0.0, 0E23, 0E0.
7. The encoding and decoding should be relatively efficient (linear in the length of the data string), but they need not be as efficient as comparisons. The database system can handle encoded numbers in all the internal operations, and translate them only on input/output to the external user. The translation can be done in user interfaces.

Among applications of the encoding method proposed in this paper is the implementation of the Semantic Binary Database Model ([Rishe-88-DDF], [Rishe-89-DDS]) by an efficient data structure [Rishe-89-EO] and by aX database machine [Rishe *et al*-89-AM].

3. The Method of Representing Numbers

The input number v is translated into a sequence (string) of bytes.

The least significant bit of each byte is the continuation bit: "1" means "more bytes to follow", "0" means "the current byte ends the number's encoding". The other 7 bits of the byte give partial information about the number v by specifying which one of 128 intervals the number falls into. The intervals are not necessarily of equal length, and some may be infinite. Thus, the first byte specifies a partitioning of $(-\infty, +\infty)$ into 128 intervals

$$(-\infty,a_1), [a_1,a_2), [a_2,a_3), \cdots, [a_{127},\infty)$$

All the intervals except the first one are closed on the left and open on the right. The interval boundaries $a_1,\ldots,a_{127}$ are constants (they may depend on the application: one partitioning is better for database management systems, while another may be preferable for manufacturing control.)

The first seven bits of the byte give the interval number, i, of one of the 128 intervals $[a_i,a_{i+1})$. When the continuation bit is zero, the number v is the lower boundary a_i of the interval. (There is no lower boundary in the first interval, since it is open on the left.) Otherwise, when the continuation bit is "1", it is known that v is in the interval (a_i,a_{i+1}) and further information is provided by the bytes that follow. The boundaries of the intervals are selected in such a way as to minimize the average length of encoding of numbers in the application. Particularly, numbers which appear most frequently in the application should be encoded by just one byte. That is, those numbers must be the lower boundaries of the intervals in the partitioning specified by the first byte.

The second byte partitions the interval (a_i,a_{i+1}) into 128 sub-intervals:

$$(a_i,b_1), [b_1,b_2), \cdots, [b_{127},a_{i+1})$$

and so forth in the bytes that follow.

The interval boundaries can and must be chosen in such a manner so as to satisfy all the requirements from the encodings as listed above.

The tree of all intervals is infinite, but the interval boundaries must be constants hard-coded in the application's encoding algorithms. Thus, the algorithm must be able to generate those constants by a finite number of interval-partitioning methods known to the algorithm. The simplest interval-partitioning method is the "arithmetic sequence": an interval (x,y), where both x and y are finite, is partitioned into

$$(x, x+\frac{y-x}{128}), \cdots,$$
$$[x+\frac{y-x}{128}\times i, x+\frac{y-x}{128}\times(i+1)), \cdots,$$
$$[x+\frac{y-x}{128}\times 127, y)$$

However, for most intervals the use of the "arithmetic sequence" is either impossible (e.g., one cannot partition an infinite interval into equal subintervals) or would violate some of the requirements of the encoding. In some incorrect partitionings it would happen that the decimal precision of v is less than the size of an interval, but v is not the lower boundary of the interval and many additional bytes would be needed to zero down on the number v. That would not be a compact representation. A correct partitioning must avoid such situations.

Consider, as an example, a possible encoding of the number 35.01237 as shown in Figure 1. Assume that one of the intervals in the first byte is [35, 36). Say, e.g., it is the interval #38. It may be, that the algorithm further partitions the interval (35, 36) so that there is a sub-interval #13 which is [35.012, 35.013). The second byte would indicate interval #13 and continuation bit '1'. It may be that the algorithm further partitions the interval [35.012, 35.013) so that there is a sub-interval #56 which is [35.01237, 35.01238). Since the original number is the lower boundary of this sub-interval, the third and last byte would indicate interval #56 and continuation bit '0'.

An example of a correct tree of intervals particularly suitable for database applications is given in the last section.

4. Lexicographic Comparability

Theorem. Bitwise lexicographic comparison of the encodings coincides with the meaningful comparison of numbers, *i.e.* the order of the real numbers.

Proof.

Consider two input numbers $v_1 > v_2$. Assume v_1 is encoded by a byte string $E_1 = b_1^1 b_2^1 b_3^1 \cdots b_n^1$ and v_2 is encoded by a byte string $E_2 = b_1^2 b_2^2 b_3^2 \cdots b_m^2$. We have to show that lexicographically $E_1 > E_2$.

Assume the contrary: $E_1 \le E_2$. This can be one of the following cases:

1. The two encodings have an identical, possibly empty prefix, after which the byte in E_2 is greater than the corresponding byte in E_1. This means: for some $k > 0$, $b_k^1 < b_k^2$ and $b_i^1 = b_i^2$ for $1 \le i < k$.
   a. If b_k^1 and b_k^2 differ only in the least significant bit, then the k-th byte puts both numbers in the same interval I. The least significant bit of b_k^2 is thus '1', meaning that v_2 is inside I, while the least significant bit of b_k^1 is '0', meaning that v_1 is the lower boundary of I. Thus, $v_1 < v_2$, a contradiction.
   b. The first seven bits of b_k^1 are lexicographically less than those of b_k^2. Therefore, v_1 falls into an interval I_1 and v_2 into I_2 where I_1 precedes I_2. Thus $v_1 < v_2$ in contradiction.
2. E_1 is a prefix of E_2, i.e.: $E_2 = E_1 s$, where s is any string. The last byte of every encoding has continuation bit '0' (meaning it is the end of the string). Thus, the last byte of E_1 has continuation bit '0'. But the last byte of E_1 is also the byte before s in E_2 (E_2 is E_1 followed by s). Thus, the byte before s has continuation bit '0', meaning that it is the last byte in E_2. Thus s must be empty. Thus $E_2 = E_1$. Thus v_1 and v_2 are each the lower boundary of the same interval defined by the byte-string E_1. Thus, $v_1 = v_2$, a contradiction.

5. A Tree of Intervals Suggested for Database Systems

Typical frequent numbers in databases include zero, small positive integers, the number -1 (which is often abused to represent null values), numbers with two decimal digits after the period (representing dollars and cents).

The following is a recommendation for the tree of intervals for database management systems. There are seven types of partitioning within the tree:

- "first-byte", for the initial interval $(-\infty, +\infty)$ (Table 1)
- "successive-integers", normally partitioned into 128 equal sub-intervals (Table 2)
- "semi-arithmetic", in which an interval is partitioned into 97 sub-intervals of size 1% and 30 sub-intervals of size 0.1% of the original interval (Table 3)
- "semi-progressive to $+\infty$" (Table 4), used for intervals of type $[L, \infty)$
- "semi-progressive to $-\infty$" (Table 5)
- "semi-progressive to $+0$" (analogously to $-\infty$)
- "semi-progressive to -0" (analogously to $+\infty$)

The above encoding satisfies the requirements and also the following property of short representation of numbers frequently used in databases:

— 127 numbers are represented in a single byte (including the delimiter). These numbers include:
  - all integers from -1 to 80;
  - all positive numbers having only one significant digit from 90 through the number 1,000,000.

— 16383 numbers are represented by at most two bytes, including the delimiter. These numbers include:
  - all integers from -100 to +2000
  - all dollars-and-cents between $-1.00 and $80.00
  - all positive numbers having only three or less significant digits from the number 1 through the number 1,000,000.

— Numbers with many significant digits require on the average less than 0.5 bytes per significant digit.

Table 6 gives an example of encoding the number 35.01237 by the 3-byte self-delimiting string 010010110001100101101110, i.e. hexadecimal 4B196E.

The algorithm of encoding has been implemented and efficiently runs under UNIX and VMS operating systems.

References

[Rishe-88-DDF] N. Rishe. *Database Design Fundamentals: A Structured Introduction to Databases and a Structured Database Design Methodology.* Prentice-Hall, Englewood Cliffs, NJ, 1988. 436 pages. ISBN 0-13-196791-6.

[Rishe-89-DDS] N. Rishe. *Database Design: The Semantic Modeling Approach.* Prentice-Hall, Englewood Cliffs, NJ, accepted to appear in 1990, approx. 550 pages.

[Rishe-89-EO] N. Rishe. "Efficient Organization of Semantic Databases" Proceedings of the Third International Conference on Foundations of Data Organization, Paris, June 21-23, 1989. Springer-Verlag. In press.

[Rishe *et al*-89-AM] N. Rishe, D. Tal, and Q. Li. "Architecture for a Massively Parallel Database Machine" *Microprocessing and Microprogramming.* The Euromicro Journal. 1989, in press.

Table 1. Partitioning of (-∞,∞) in the first byte.

sub-interval #	sub-interval	partitioning of sub-interval
1.	(−∞, -1)	"semi-progressive to −∞"
2.	[-1, 0)	"semi-progressive to −0"
3.	[0, 1)	"semi-progressive to +0"
4.	[1, 2)	"semi-arithmetic"
5.	[2, 3)	"semi-arithmetic"
	...	
82.	[79, 80)	"semi-arithmetic"
83.	[80, 90)	"semi-arithmetic"
84.	[90, 100)	"semi-arithmetic"
85.	[100, 200)	"semi-arithmetic"
86.	[200, 300)	"semi-arithmetic"
87.	[300, 400)	"semi-arithmetic"
88.	[400, 500)	"semi-arithmetic"
89.	[500, 600)	"semi-arithmetic"
80.	[600, 700)	"semi-arithmetic"
91.	[700, 800)	"semi-arithmetic"
92.	[800, 900)	"semi-arithmetic"
93.	[900, 1000)	"semi-arithmetic"
94.	[1000, 1128)	"successive-integers"
95.	[1128, 1256)	"successive-integers"
96.	[1256, 1384)	"successive-integers"
97.	[1384, 1512)	"successive-integers"
98.	[1512, 1640)	"successive-integers"
99.	[1640, 1768)	"successive-integers"
100.	[1768, 1896)	"successive-integers"
101.	[1896, 2000)	"successive-integers"
102.	[2000, 3000)	"semi-arithmetic"
103.	[3000, 4000)	"semi-arithmetic"
	...	
109.	[9000, 10000)	"semi-arithmetic"
110.	[10000, 20000)	"semi-arithmetic"
111.	[20000, 30000)	"semi-arithmetic"
	...	
117.	[80000, 90000)	"semi-arithmetic"
118.	[90000, 1E5)	"semi-arithmetic"
119.	[1E5, 2E5)	"semi-arithmetic"
120.	[2E5, 3E5)	"semi-arithmetic"
	...	
127.	[9E5, 1E6)	"semi-arithmetic"
128.	[1E6, +∞)	"semi-progressive to +∞"

Table 2. Successive-integers partitioning of interval (L,R).

All the sub-intervals of (L,R) have the "semi-arithmetic" partitioning (see Table 3).
Examples are given for interval (1000, 1128).
When R-L=128, the successive-integers partitioning becomes "arithmetic sequencing".
(R-L≠128 only for the interval (1896, 2000)).

sub-interval #	sub-interval	example
1.	$(L, L+1)$	(1000, 1001)
2-128	for j=2..128 $[L+j-1, L+j)$	[1001, 1002)
		...
		[1127, 1128)

Table 3. Semi-arithmetic partitioning of interval (L,R).
All the sub-intervals have the "semi-arithmetic" partitioning as well.
Examples are given for interval (7,8). (That is, L=7, R=8.)

sub-interval #	sub-interval	example
1.	$(L, L+\frac{R-L}{1000})$	(7, 7.001)
2-20	for j=2..20 $[L+(j-1)\frac{R-L}{1000}, L+j\frac{R-L}{1000})$	[7.001, 7.002) . . . [7.019, 7.02)
21-117	for j=3..99 $[L+(j-1)\frac{R-L}{100}, L+j\frac{R-L}{100})$	[7.02, 7.03) . . . [7.98, 7.99)
118-127	for j=991..1000 $[L+(j-1)\frac{R-L}{1000}, L+j\frac{R-L}{1000})$	[7.990, 7.991) . . . [7.999, 8)

Table 4. Semi-progressive to +∞ partitioning of interval (L,R).
Examples are given for interval (1E6, ∞).

sub-interval #	sub-interval	example	sub-interval partitioning
1.	(L, 2L)	(1E6, 2E6)	"semi-arithmetic"
2-99	for j=2..99 [jL, L+jL)	[2E6, 3E6)	"semi-arithmetic"
		. . .	
		[99E6, 100E6)	"semi-arithmetic"
100-108	for j=1..9 [100jL, 100L+100jL)	[1E8, 2E8)	"semi-arithmetic"
		. . .	
		[9E8, 10E8)	"semi-arithmetic"
109-117	for j=1..9 [1000jL, 1000L+1000jL)	[1E9, 2E9)	"semi-arithmetic"
		. . .	
		[9E9, 10E9)	"semi-arithmetic"
118-126	for j=1..9 [10000jL, 10000L+10000jL)	[1E10, 2E10)	"semi-arithmetic"
		. . .	
		[9E10, 10E10)	"semi-arithmetic"
127.	[L*1E5, min(R,L*1E10))	[1E11, 1E16)	"semi-progressive to +∞"
128.	[L*1E10, R)	[1E16, ∞)	"semi-progressive to +∞"

Table 5. Semi-progressive to −∞ partitioning of interval (L,R).
Examples are given for interval (−∞, -1E6).

sub-interval #	sub-interval	example	sub-interval partitioning
1.	(L, R*1E10)	(−∞, -1E16)	"semi-progressive to −∞"
2.	[max(L,R*1E10), R*1E5)	[-1E16, -1E11)	"semi-progressive to −∞"
3-11	for j=9..1 [10000(j+1)R, 10000jR)	[-10E10, -9E10)	"semi-arithmetic"
		. . .	
		[-2E10, -1E10)	"semi-arithmetic"
12-20	for j=9..1 [1000R+1000jR, 1000jR)	[-10E9, -9E9)	"semi-arithmetic"
		. . .	
		[-2E9, -1E9)	"semi-arithmetic"
21-29	for j=9..1 [100R+100jR, 100jR)	[-10E8, -9E8)	"semi-arithmetic"
		. . .	
		[-2E8, -1E8)	"semi-arithmetic"
30-128	for j=99..1 [R+jR, jR)	[-100E6, -99E6)	"semi-arithmetic"
		. . .	
		[-2E6, -1E6)	"semi-arithmetic"

Table 6. An example of encoding the number 35.01237

byte nbr.	interval	interval nbr.	binary for (intrv# – 1)	continuation bit	byte code
1	[35, 36)	38	0100101	1	01001011
2	[35.012, 35.013)	13	0001100	1	00011001
3	[35.01237, 35.01338)	56	0110111	0	01101110

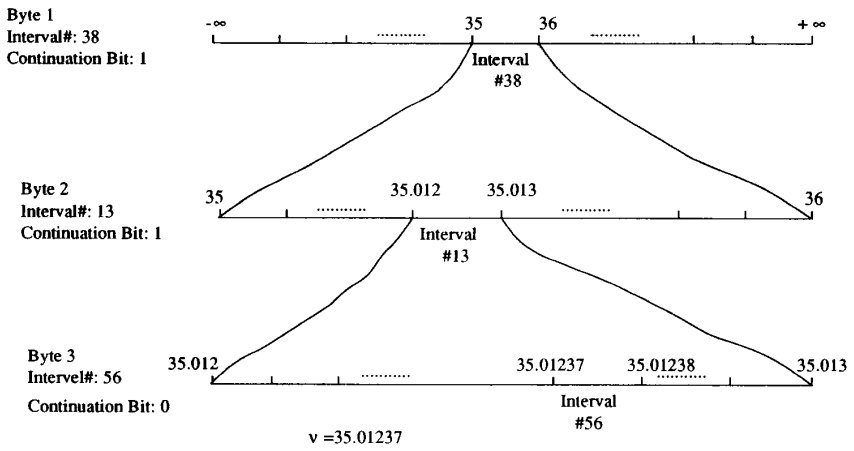

Figure 1. Encoding of the number 35.01237 by 3 bytes.