88-CM

PROCEEDINGS
26th ANNUAL
SOUTHEAST REGIONAL
CONFERENCE

James H. Cross II and Edwin Ellis, Editors

Mobile, Alabama
April 20-22, 1988

A Compact Monotonic Universal Encoding of Numbers

Naphtali Rishe

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

Many applications, and many database management systems in particular, require compact variable-length representations of arbitrary numbers. The primary operations performed on stored numbers in those applications are comparisons (=, >, *etc.*), rather than the arithmetic operations. This paper proposes an encoding of numbers which is particularly convenient for such applications.

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. 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 $b_1^2 b_2^2 b_3^2 b_4^2$, 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 precision is not uniform. (In a database, the varying precisision will allow 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 is 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 of the next 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 may 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] by a database machine [Tal-88].

The method of encoding

The input number v is translated into a sequence 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 one of 128 intervals into which the number falls. 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 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 bound a_i of the interval. (There is no lower bound in the first interval, since it is open on the left.) Otherwise, when the continuation bit is "1", it is known that the number is in the interval (a_i,a_{i+1}) and further information is provided by the bytes that follow.

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

$$\left(x,\ x+\frac{y-x}{128}\right),\ \cdots,$$

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

$$\left[x+\frac{y-x}{128}\times 127,\ y\right)$$

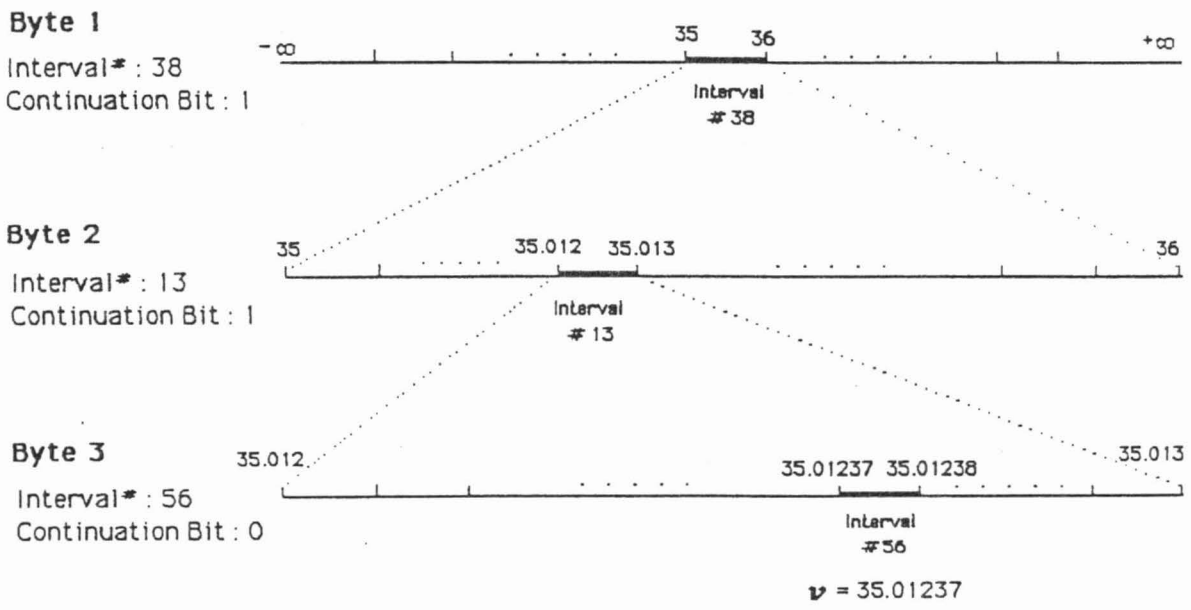

Figure 1. An Example of a Possible Encoding of The Number 35.01237

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 compactness or monotonicity of the encoding. In many situations it would happen that the decimal precision of ν is less than the size of an interval, but ν is not the lower boundary of the interval and many additional bytes would be needed to zero down on the number ν. 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.

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)

Table 1. First-byte partitioning.

sub-interval #	sub-interval	partitioning of sub-interval
1.	$(-\infty, -1)$	"semi-progressive to $-\infty$"
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, $+\infty$)	"semi-progressive to $+\infty$"

- "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 2)
- "successive-integers", normally partitioned into 128 equal sub-intervals (Table 3)
- "semi-progressive to $+\infty$" (Table 4)
- "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:

— 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.

— 16386 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 digit 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.

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).

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 $+\infty$ 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 $+\infty$"
128.	[L*1E10, R)	[1E16, ∞)	"semi-progressive to $+\infty$"

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

sub-interval #	sub-interval	example	sub-interval partitioning
1.	(L, R*1E10)	($-\infty$, -1E16)	"semi-progressive to $-\infty$"
2.	[min(L,R*1E10), R*1E5)	[-1E16, -1E11)	"semi-progressive to $-\infty$"
3-11	for j=9..1 [10000(j+1)R, 10000jR)	[-10E10, -9E10)	"semi-arithmetic"
		...	
		[-2E10, -1E10)	"semi-arithmetic"
12-20	for j=1..9 [1000R+1000jR, 1000jR)	[-10E9, -9E9)	"semi-arithmetic"
		...	
		[-2E9, -1E9)	"semi-arithmetic"
21-29	for j=1..9 [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

References

[Rishe-88] N. Rishe. *Database Design Fundamentals: A Structured Introduction to Databases and a Structured Database Design Methodology.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Tal-88] D. Tal, N. Rishe, D. Barton and N. Prabhakaran. "High-throughput Highly-parallel Database System." Proceedings of the 26th Annual Conference of Southeast Region of the Association for Computing Machinery, 1988.