Data & Knowledge Engineering 8 (1992) 339–351
North-Holland

Interval-based approach to lexicographic representation and compression of numeric data*

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

Abstract

Rishe, N., Interval-based approach to lexicographic representation and compression of numeric data, Data & Knowledge Engineering 8 (1992) 339–351.

This paper proposes a new method of encoding numbers by variable-length byte-strings. The primary property of the encoding is that the lexicographic comparison of the encoded numbers corresponds correctly to the order of the real numbers. The encoding is space-efficient. Further, unlike the fixed-length representations of numbers (fixed-point, floating-point, etc.,) the encoded numbers are not limited in their magnitude or the number of their significant digits. The paper also elaborates the application of the encoding method to the storage of numeric data in databases. The proposed application for databases is a uniform format for all the numbers, regardless of their types and attributes (fields). All the numbers are represented in a form of lexicographically-comparable byte-strings. This form simplifies the data management software (only one format to deal with at the physical database level) and hardware (when associative memory and storage devices etc. are used); makes the applications more flexible (by removing limitations on the sizes of numbers); and is space-efficient for all numbers while being especially concise for those numbers that are used more frequently in databases.

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 some advanced database management systems. These systems make the data formats transparent to the user. Furthermore, these systems should allow a logical field to hold numbers of unpredictably large or small magnitude and unpredictably 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. (For example, in any fixed-point format the number of bits, n, is the application's constant, and, therefore, when a datum exceeding 2^n is input, it cannot be represented.) A representation which satisfies the 'unboundness' requirement is the standard mathematical notation (on paper) using a variable-length string of arabic numerals and exponent (e.g. -3.57×10^{-101}), 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

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

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). This is true even in those data bases where the bulk of updates involve arithmetic adjustment of values (e.g. accounting databases) — most of the effort in such databases is spent on *finding* the data in the database, which involves comparisons, not arithmetics. 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 (e.g. a disk with a controller capable of string search and comparison). The efficiency of such applications would benefit if numbers could be handled as character strings. For example, in most cases the result of a *lexicographic* 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 (i.e. microcode or software resident in the storage device or its controller) 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 Section 2. Section 3 describes the proposed method of representing numbers. The method can be adapted to different types of applications by choosing a 'tree of intervals' which is more efficient for a particular type. Section 4 proposes such a 'tree of intervals' for database applications.

2. Specification of requirements

The encoding of numbers $E{:}\mathit{Numbers} \rightarrow \mathit{ByteStrings}$ proposed in this paper satisfies the following requirements:

1. Bitwise-lexicographic comparison of the encodings[1] 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. (This means that $E(n_1)^{strings} \leqslant E(n_2)$ iff $n_1{}^{reals} \leqslant n_2$. Thus, e.g., 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 $b_2^1 < b_2^2$ (i.e. the first byte in this example is identical in the two strings, while the second byte of the second string is greater than the second byte of the first string), then $n_1 < n_2$.) The conventional 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. (The *precision* of a number is the number of its significant digits in the standard mathematical notation on paper.) In a database or a file we wish to be able to compare and store integers, real numbers, numbers with very many significant digits, and numbers with just a few significant digits. We wish to have one uniform format to represent all these kinds of numbers. 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

[1] A bit-string (or byte-string) v is (bitwise-) lexicographically $\leqslant$ than a string w iff $w = vu$ for some string u (i.e. v is a prefix of w) or for some strings x, y, and z, $v = xy$ and $w = xz$ and the first bit of y is 0 and the first bit of z is '1' (x may be empty).

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 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 having only one significant digit each : 3,000,000; 5; 0.000,000,000,000,000,7. Each of these three numbers should require only a few bits each, while the number 12345678.90 should require many more bits. The number of bits in the representation of a number should be approximately equal to the amount of information[2] 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 facilitates the handling of records. It also results in the saving of space, at least for numbers whose encodings are short. For example, if we were to use a scheme with delimiters, the shortest numbers would be represented by two bytes: one for the contents and one for the delimiter; in a non-delimiter scheme one byte will suffice for the shortest numbers.
6. The representation of numbers is one-to-one. For example, there should not be several representations for the number zero, as there are in the mathematical notation on paper: 0, 0.00, −0.0, 0E23, 0E0. (This means that the encoding E is a *function*. Furthermore E is 1:1. Thus, $E(x) = E(y)$ iff $x = y$.)
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.

The conventional computer encodings of numbers do not satisfy the requirements of lexicographic comparability, unboundness, and others. The lexicographic comparability requirement is satisfied by a method proposed in [2]. Their encoding works nicely with small integers and with rational numbers p/q where p and q are of the same order of magnitude. It is not useful for numbers with large exponents. Nor can their encoding be tuned for databases as discussed in Section 4. On the other hand, their encoding has interesting properties useful for numeric processing, which is not a goal of the method that I propose below. [2] is based on the continued fraction theory. The encoding that I propose below is based on building an infinite tree of intervals of the real numbers, with the interval $(-\infty, \infty)$ at the root. The children of a node partition the parent interval into sub-intervals. (Although the tree is infinite, the algorithm by which the children of a node are generated is finite.)

Among applications of the encoding method proposed in this paper is the implementation

[2] The amount of information in an arbitrary number roughly corresponds to the size of the most compact representation of that number. The amount of information in a decimal number is roughly the number of its significant digits times $\log_2 10$. Thus, the amount of information in 0.00056 is roughly $2 \times \log_2 10 = 6.6$, because this number has only two significant digits. This rough estimate does not include the usually small information contained in non-redundant leading and trailing zero digits. If the number of such zero digits is z then the additional information is $\log_2 z$. The number 0.00056 has three such zeros; thus the additional information amounts to approximately $\log_2 3 = 1.6$. The number 123,456,789,000,000,000,000 has 9 significant digits and 12 non-redundant trailing zeros. Its amount of information is approximately $9 \times \log_2 10 + \log_2 12 = 33.5$. The above way to estimate the amount of information is appropriate to arbitrary numbers that do not come from a known context or application or distribution frequencies. A deeper discussion on the subject can be found in [10].

of the Semantic Binary Database Model [3, 6, 5], by an efficient data structure [7] and by a database machine [9].

3. The method of representing numbers

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

The least significant bit[3] 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, $i + 1 (i = 0, 1, \ldots, 127)$, of one of the 128 intervals $[a_i, a_{i+1})$. When the continuation bit is zero, the number v is a_i, which is the lower boundary of the interval. (Notice that there is no lower boundary in the first interval, $(-\infty, a_1)$, since it is open on the left.) Otherwise, when the continuation bit is '1', it is known that v is inside 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

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

[3] A byte is regarded as a bit strings of 8 bits. Its least-significant bit (for the purpose of comparison) is the rightmost bit of the string. Thus, the least significant bit of 10101010_2 is '0'.

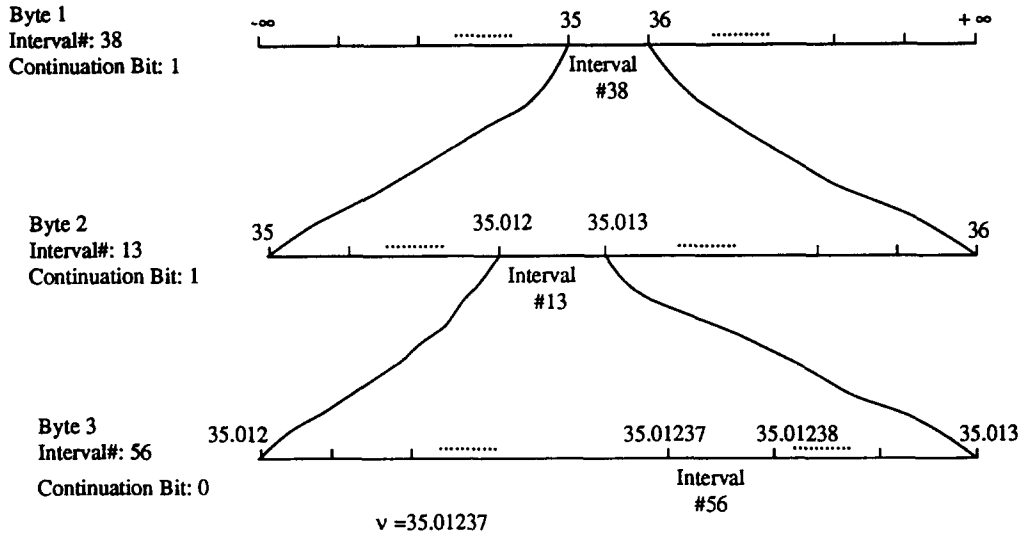


Fig. 1. Encoding of the number 35.01237 by 3 bytes.

(In the above, the first interval is open, and the remaining 127 left-closed intervals correspond to $i = 1, 2, \ldots, 127$.)

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 *Fig. 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.

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

A proof is given in Appendix 1.

4. A tree of intervals suggested for database systems

In many database applications, the most frequent numbers 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). Also, most numbers in a database normally have originated in a decimal form from a human user, or are the results of

simple arithmetic operations on those decimal numbers. (Notice that the precision of a number as measured by the number of its significant digits crucially depends on the digit base, e.g. the number 1/8 has only 1 significant octal digit: 0.1_8, but three significant decimal digits: 0.125_{10}; whereas the opposite holds for the number 1/10.) The tree of intervals for databases should provide an especially concise representation of the numbers occurring more frequently in databases, but also to satisfy all the requirements of Section 2 for *all* the rational numbers having a finite number of significant decimal digits.

The following is a recommendation for the tree of intervals for database management systems. Although the tree is infinite, it is fully defined by 7 small fixed tables. There are seven types of partitioning within the tree:

1. 'first-byte', for the initial interval $(-\infty, +\infty)$ (*Table 1*). The partitioning scheme proposed

Table 1
Partitioning of $(-\infty, \infty)$ in the first byte

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'
90	[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 $+\infty$'

for the first byte, i.e. the interval $(-\infty, +\infty)$, is designed in such a way that:

a. Many most frequently used integers of many typical applications are represented by just one byte, i.e. the first byte is terminal.
b. The above is also true for many single-significant-digit numbers, e.g. 2000, 300000, 1000000.
c. The sub-intervals grow larger and larger towards the edges of the interval.
d. The interval boundaries are established in such a way so as to allow a short representation of the remainder of the number being encoded by successive bytes.

2. 'successive-integers', normally partitioned into 128 equal sub-intervals (*Table 2*). This is useful for partitioning intervals like [1512, 1640), so that e.g. the number 1545 could be encoded by just two bytes.
3. '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*). This is useful for partitioning intervals like [5, 6); [800, 900); [21.5436456546, 21.5436456547). Thus, a byte with semi-arithmetic partitioning adds at least two significant decimal digits to the information about the number. In some cases it adds 3 significant digits. The author has chosen to cluster the latter cases at the edges of the interval because he believes that numbers like 500.1 and 599.9 are somewhat more frequent in databases than 535.7, due to the rounding errors. In the proposed scheme, the numbers 500.1 and 599.9 are

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'. ($R - L \neq 128$ 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) (i.e. $L = 7$, $R = 8$)

Sub-interval #	Sub-interval	Example
1	$\left(L, L + \frac{R-L}{1000}\right)$	(7, 7.001)
2–20	for $j = 2..20$: $\left[L + (j-1)\frac{R-L}{1000}, L + j\frac{R-L}{1000}\right)$	[7.001, 7.002)
		...
		[7.019, 7.02)
21–117	for $j = 3..99$: $\left[L + (j-1)\frac{R-L}{100}, L + j\frac{R-L}{100}\right)$	[7.02, 7.03)
		...
		[7.98, 7.99)
118–127	for $j = 991..1000$: $\left[L + (j-1)\frac{R-L}{1000}, L + j\frac{R-L}{1000}\right)$	[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 $j = 2 \ldots 99$: $[jL, L + jL)$	[2E6, 3E6)	'semi-arithmetic'
		...	
		[99E6, 100E6)	'semi-arithmetic'
100–108	for $j = 1 \ldots 9$: $[100jL, 100L + 100jL)$	[1E8, 2E8)	'semi-arithmetic'
		...	
		[9E8, 10E8)	'semi-arithmetic'
109–117	for $j = 1 \ldots 9$: $[1000jL, 1000L + 1000jL)$	[1E9, 2E9)	'semi-arithmetic'
		...	
		[9E9, 10E9)	'semi-arithmetic'
118–126	for $j = 1 \ldots 9$: $[10000jL, 10000L + 10000jL)$	[1E10, 2E10)	'semi-arithmetic'
		...	
		[9E10, 10E1)	'semi-arithmetic'
127	$[L \times 1E5, \min(R, L \times 1E10))$	[1E11, 1E16)	'semi-progressive to $+\infty$'
128	$[L \times 1E10, R)$	[1E16, ∞)	'semi-progressive to $+\infty$'

represented by two bytes each, whereas 535.7 requires 3 bytes. If it were not for rounding-error consideration, it would not matter where to cluster the 3-digit sub-intervals.

4. 'semi-progressive to $+\infty$' (*Table 4*), used for intervals of type $[L, \infty)$
5. 'semi-progressive to $-\infty$' (*Table 5*)
6. 'semi-progressive to $+0$' (analogous to $-\infty$.) This is used for intervals like $(0, H)$, where H is a number of very small absolute value.
7. 'semi-progressive to -0' (analogous to $+\infty$). This is used for intervals like $(L, 0)$, where L is a negative number of very small absolute value.

The above encoding satisfies the requirements of section 2 and also the following property of short representation of numbers frequently used in databases:

(a) 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.

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	$(L, R \times 1E10)$	($-\infty$, $-1E16$)	'semi-progressive to $-\infty$'
2	$[\max(L, R \times 1E10), R \times 1E5)$	[$-1E16$, $-1E11$)	'semi-progressive to $-\infty$'
3–11	for $j = 9 \ldots 1$: $[10000(j + 1)R, 10000jR)$	[$-10E10$, $-9E10$)	'semi-arithmetic'
		...	
		[$-2E10$, $-1E10$)	'semi-arithmetic'
12–20	for $j = 9 \ldots 1$: $[1000R + 1000jR, 1000jR)$	[$-10E9$, $-9E9$)	'semi-arithmetic'
		...	
		[$-2E9$, $-1E9$)	'semi-arithmetic'
21–29	for $j = 9 \ldots 1$: $[100R + 100jR, 100jR)$	[$-10E8$, $-9E8$)	'semi-arithmetic'
		...	
		[$-2E8$, $-1E8$)	'semi-arithmetic'
30–128	for $j = 99 \ldots 1$: $[R + jR, jR)$	[$-100E6$, $-99E6$)	'semi-arithmetic'
		...	
		[$-2E6$, $-1E6$)	'semi-arithmetic'

(b) 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.

(c) 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 runs efficiently under the UNIX and VMS operating systems. More discussion of this can be found in Appendix 2.

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

Acknowledgement

The Author thanks Michael Alexopoulos, Scott Graham, and Wei Sun for their comments and Vijaykumar Narayanan and Michael Alexopoulos for writing programs implementing the encoding. The Author is grateful for the suggestions of the anonymous referees and the Editor; these suggestions have resulted in a significant improvement of the revised version of the paper.

Appendix 1. Proof of the theorem

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: $E_1 \leqslant E_2$. This can be one of the following cases:

1. For some $k > 0$, $b_k^1 < b_k^2$ and $b_i^1 = b_i^2$ for $1 \leqslant i < k$. (This means that 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.)
   a. If b_k^1 and b_k^2 differ only in the least significant bit, then the kth 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, $v < v_2$, 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. □

Appendix 2. An implementation

We have utilized this scheme for number encoding in our semantic binary database management system [7]. The semantic data model used is a modification [3, 6] of the Binary Model of [1]. The implementation is based on an algebra-like low-level access language [6], such that an arbitrary query can be performed as one or several elementary queries of the language. Most elementary queries, including such non-trivial queries as range queries and others, can be performed in just one single access to the disk. Queries in higher-level languages, like the Semantic Predicate Calculus [8] and the fourth-generation semantic extension of Pascal [4] are translated into elementary queries of the low-level access language.

Logically, at any moment in time the database is a set of facts about *abstract objects*, which are entities of the real world represented by identifiers invisible to the user. The facts are: the unary facts xC stating that an object whose identifier is x belongs to a category whose identifier is C; and binary facts xRy stating that there is a relationship, R, between x and y, where x is an abstract object's identifier and y is an abstract object's identifier or a concrete object, i.e. a number, a character string, a date, etc. Non-binary relationships are decomposed into binary relationships.

The entire database is stored in a single file. This file contains all the facts of the database (xC and xRy) and additional information, called *inverted facts*, which are described below. The file is maintained in a format similar to a B-tree. The variation of the B-tree used here allows sequential access according to the lexicographic order of the items comprising the facts and the inverted facts, as well as random access by arbitrary prefixes of the facts and inverted facts. The facts which are close to each other in the lexicographic order reside close together in the file.

The file contains the original facts and the following 'inverted facts':

1. In addition to $\boldsymbol{xC}$, we store its inverse $\bar{\boldsymbol{C}}\boldsymbol{x}$. ($\bar{C}$ is the system-chosen identifier to represent the inverse information about the category C. For example, it can be defined as $\bar{C} = 0 - C$.) Thus, the elementary query to find all the objects of the category C, can be answered by examining the (inverted) facts whose prefix is $\bar{C}$. These inverted facts are clustered together in the lexicographic order of the physical database.
2. In addition to $\boldsymbol{xRv}$, where v is a concrete object (a number, a string, or a value of another type), we store $\bar{\boldsymbol{R}}\boldsymbol{vx}$. Thus, the elementary range query 'For given R, l and h find all the facts xRy such that $l \leqslant y \leqslant h$' is satisfied by all and only the inverted facts which are positioned in the file between $\bar{R}l$ and $\bar{R}h$HighSuffix. (HighSuffix is a suffix which is lexicographically greater than any other possible suffix.) Thus, the result will most probably appear in one physical block, if it can fit into one block.
3. In addition to $\boldsymbol{xRy}$, where both x and y are abstract objects, we store $\boldsymbol{y\bar{R}x}$. Thus, for any abstract object x, all its relationships xRy, xRv, zRx, and xC can be found in one place in the file: the regular and inverted facts which begin with the prefix x. (The infixes are:

categories for xC, relations for xRy and xRv, and inverse relations $x\bar{R}z$ from which we find z such that zRx.)

The 'records' of the B-tree are the regular and inverted facts. The records are of varying length. The B-tree-keys of the 'records' are normally the entire B-tree-records, i.e. facts, regular and inverted. (An exception to this is when the record happens to be very long. The only potentially long records represent facts xRv where v is a very long character string. We employ a special handling algorithm for very long character strings.) Access to this B-tree does not require knowledge of the entire key: any prefix will do. All the index blocks of the B-tree can normally be held in cache.

At the most physical level, the data in the facts is compressed to minimal space. Also, since many consecutive facts share a prefix (e.g. an abstract object identifier) the prefix need not be repeated for each fact. In this way the facts are compressed further. The duplication in the number of facts due to the inverses is 100%, since there is only one inverse per each original fact. The B-tree causes an additional 30% overhead. (This overhead occurs because in a B-tree the data blocks are only 75% full on the average, though this can be improved by periodic reorganization. The overhead due the index blocks of the B-tree is no more than 1–2% since they contain only one short fact for every data block.) The total space used for the database is therefore only about 160% more than the amount of information in the database, i.e. the space minimally required to store the database in the most compressed form with no regard to the efficiency of data retrieval or update. No separate index files are needed.

The scheme proposed in this paper is employed to encode the numbers in the facts. The following are the most important of the scheme's properties utilized:

1. Bitwise lexicographic comparison of the encodings coincides with the meaningful comparison of numbers. Our B-tree search algorithms operate on and compare variable strings like xRy, where y can be character string or a number encoding, transparently to the low-layer software of file-management.
2. There is no limit on arbitrarily large, arbitrarily small, or arbitrarily precise numbers. This should be a requirement of any fully-flexible semantic database management system. Additionally, in our implementation, this requirement is particularly important since all of the facts xRy for various attribute relations R are stored in the same file and treated in the same way. We do not wish to set a limit on the range of the data at the time of the design or creation of the file.
3. Every number bears its own precision, i.e. the precision is not uniform throughout the database. (This allows integers, reals, and values of different attributes with different precisions to be treated in a uniform way in one file in the database.)
4. The encodings are of varying length and are approximately maximally space efficient with respect to their informational content.
5. No additional byte(s) are required to store the length of the encoded representation or to delimit its end. The absence of delimiters gives some additional saving in space (at least for those numbers whose encodings are shorter than 7 bytes), and also facilitates the handling of records.

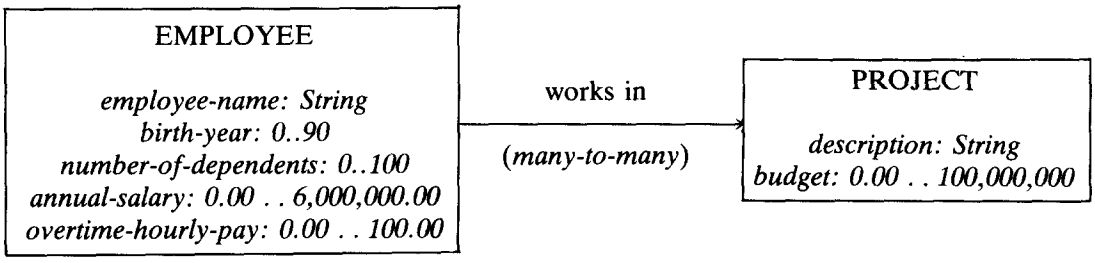
The file management software, which is the lowest layer of the DBMS, sees the numbers only in the encoded form and does not distinguish between numbers, strings, etc. The same applies to the intermediate layers of the DBMS, such as fact representation, integrity handling, concurrency control, query evaluation, etc. The encoding and decoding is done in the user interface software. This keeps the system simple and efficient.

The user interface layer of the system includes two functions: *encode*: *String* → *String* and its inverse *decode*. The *encode* function takes a printable number representation (a variable length string), e.g. '−2.34E107' or '102.3', and converts it into the encoded byte-string. The

encode/decode functions work when the data comes from/goes to a *human* user. These functions are not quite sufficient when the data originates from or is retrieved into a user's *program*. (Such a program may be written in a data manipulation language which is an extension of a regular programming language.) In this case, the program's numbers are first converted into their printable form and then encoded by the function *encode*.

The most important properties of the encodings are the lexicographic comparability, the variable length, and the uniformity between different data types. Another important property is the compactness (small sizes), as illustrated by the following example.

As an example, consider the following subschema of a database.



Consider the representation of attribute values for a hypothetical set of 10 employees.

The salaries are: 20500.25, 11700, 9E4, six times 25E3, 1E6. The average size in the printable format (plus delimiters): 5.3. The average size in the encoded format: 2.2. The size of each number in the fixed-size *decimal float* format: 8 (The latter size depends on the programming language and the environment. Alternatively, a fixed-point 9-digit decimal format may be used, which takes 5 bytes.)

The numbers of dependents are: seven times 0, 2, 2, 4. The average size in the printable format (plus delimiters): 2. The average size in the encoded format: 1. The size of each number in the fixed-size small-integer format: 2.

The overtime hourly pay numbers are: eight times 0 (meaning: no overtime pay allowed), 6.35, 12.20. The average size in the printable format (plus delimiters): 2.7. The average size in the encoded format: 1.2. The size of each number in the fixed-size decimal format: 3. (Assuming that the programming language allows a fixed-point decimal format with 5 decimal digits stored in two digits per byte.)

The birth years are: 40, 45, 60, 60, 61, 61, 61, 63, 64, 64. The average size in the printable format (plus delimiters): 3. The average size in the encoded format: 1. The size of each number in the fixed-size small-integer format: 2.

If we must have one fixed-size format for all the above numbers (as well as for other present and future attributes in the database), that would be the decimal float, with the size 8 (depending on the language and the environment and provided no number with more than 12 significant decimal digits will ever be stored in the database.)

The following are some of the facts in this database.

object1 **EMPLOYEE**
object1 ***#DEPENDENTS*** 0
object1 ***SALARY*** 9000
object1 ***WORKS-IN*** object2
object1 ***WORKS-IN*** object3

The objects are represented by integers. For example, object1 is represented by the number 36. The schema concepts, such as EMPLOYEE and WORKS-IN, are represented by small integers. Under the assumption that there are less than 80 schema concepts, the above 5 facts' sizes are: 2 (1 byte for the representation of object1 + 1 byte for the representation of the category — no delimiters needed); 3 (1 + 1 + 1); 3 (1 + 1 + 1); 3 (1 + 1 + 1); 3 (1 + 1 + 1). (In this estimate, object1 is encoded by just 1 byte, *encode*('36'). When the database grows, some other objects will be encoded by two or more bytes.)

References

[1] J.R. Abrial, Data semantics, in J.W. Klimbie and K.L. Koffeman, eds., *Data Base Management* (North-Holland, Amsterdam, 1974).

[2] D. Matula and P. Kornerup, An order preserving finite binary encoding of rationals, *Proc. 6th Symp. on Computer Arithmetics* (IEEE Computer Society Press, Silver Spring, MD, 1983).

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

[4] N. Rishe, Transaction-management system in a fourth-generation language for semantic databases, in: *Mini and Microcomputers: From Micros to Supercomputers (Proc. ISMM Internat. Conf. on Mini and Microcomputers*, Miami Beach, Dec. 14–16, 1988; M.H. Hamza, ed.) (Acta Press, 1988) 92–95.

[5] N. Rishe, Semantic database management: From microcomputers to massively parallel database machines, Keynote Paper, *Proc. Sixth Symp. on Microcomputer and Microprocessor Applications*, Budapest (Oct. 1989) 1–12.

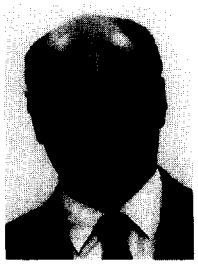
[6] N. Rishe, *Database Design: The Semantic Modeling Approach* (McGraw-Hill, New York, 1992).

[7] N. Rishe, A file structure for semantic databases., *Inform. Syst.* 16(4) (1991) 375–385.

[8] N. Rishe and W. Sun, A predicate calculus language for queries and transactions in semantic databases, in: N. Rishe, S. Navathe and D. Tal, eds. *Databases: Theory, Design and Applications* (IEEE Computer Society Press, Silver Spring, MD, 1991). 204–221.

[9] N. Rishe, D. Tal and Q. Li, Architecture for a massively parallel database machine, Microprocessing Microprogramming 25 (1989) 33–38.

[10] G. Salton, *Automatic Text Processing* (Addison-Wesley, Reading, MA, 1989).



Dr. Rishe is an Associate Professor of Computer Science at Florida International University. Dr. Rishe's publications on databases and related issues include two books (*Database Design Fundamentals: A Structured Introduction to Databases and a Structured Database Design Methodology*, Prentice-Hall, 1988; *Database Design: The Semantic Modelling Approach*, McGraw-Hill, 1992) and many papers. Dr. Rishe chaired the steering and program committees of the PARBASE-90 conference. Dr. Rishe also has extensive experience in database applications and database systems in the industry. This included eight years of employment as head of software and database projects (1976–84) and later consulting for companies such as Hewlett-Packard. Prof. Rishe has a Ph. D. in Computer Science from Tel Aviv University.