Storage of Numeric Data in ADAMS

Russell F. Haddleton

IPC-TR-92-005
August 10, 1992

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22901

This research was supported in part by
DOE Grant #DE-FG05-88ER25063 and
JPL Contract #957721

Storage of Numeric Data in ADAMS

Abstract

This paper presents a scheme for the representation and storage of numeric data. The implemention of this scheme within the ADAMS Database System solves the major problem of maintaining numeric data in a bitwise ordered form, while providing the additional benefits of simplicity, expandability, portability, and compactness.

1. The Problem

There are many commonly employed formats for representing numeric data. The C language provides seperate types for the representation of integer and real data, and extends these float and int types to double and long in order to allow the language user access to a larger range of values and to a greater degree of precision. Another format involves simply storing numeric digits as characters (using, for example, ASCII or EBCDIC) . This allows for a flexible degree of precision and magnitude, but is not efficient in terms of storage. Some systems rely on one format for in-memory manipulation, which we will call *host format*, and another for disk or tape storage, which we will call *storage format*.

Criteria for successful in-memory representation and for persistent storage formats of numeric data depend on what is required of the system employing them. There are a number of goals and trade-offs.

A. Goals

1. Quick retrieval of stored data from the storage media is a primary goal for most systems. Of particular interest are not only schemes that aid in exact match retrievals, but those which efficiently support range searches.

2. Fast conversion between stored format and host format is an important component of fast data access. A design allowing quick conversion from the storage format(s) to the host format(s) of a particular host processor would be beneficial. No conversion whatsoever is needed if the storage format is identical to the host format. This is clearly the fastest possible implementation, but it limits the sharing of stored data between different host processors.

3. Type inheritance, especially as it relates to range searching within numeric data, is valuable. Many programming languages logically treat integer as a sub class of

type real, but this is implemented through coercion and the internal representations do not exhibit inheritance. Coercion among real and integer data at the database level would mean that in a database range search we would have to search among the integers, then coerce the search key and search among the reals, then combine the results of both searches, sorting so as to provide the expected order. This extra processing would clearly be undesirable, and a scheme supporting true inheritance would be both faster and simpler.

4. Storage space is a concern. A system maintaining very large quantities of data would benefit by using a scheme devoted to storing numeric data in as few bytes as possible. The space concern need not end with secondary storage, as some systems may cache large quantities of numeric data in memory.

5. Portability is also an important consideration. A system may frequently transfer numeric data to other systems either electronically or on secondary storage media; or a database may be distributed over a number of heterogeneous systems. A scheme supporting for easy conversion to and from a number of different host formats is then desirable.

6. Simplicity is an obvious design goal. While not as critical as the other five, it can contribute to their successful achievement.

Within an ADAMS database we seek to store scientific data (of possibly large precision or magnitude) in a non host-specific format (so as to allow data portability). As a great deal of numeric data may be stored, a compact format is beneficial. Fast access is important, with special consideration given to designs allowing both fast exact match and fast range search retrieval over very large data sets.

Many systems concern themselves with some form of storage to host conversion. The conversion task can be complicated. The desired numeric data formats can include character data, integer and real data of various lengths, as well as more obscure types such as PL/1’s picture. Further complications can include byte ordering, bit ordering, 1’s and 2’s complement integer representations, signed versus unsigned integers, and non-standard floating point representations in different host formats. While these issues are common to the portability goal, the goal of achieving high conversion speed on a given processor may directly conflict with the goals of portability, small storage, and simplicity.

The retrieval issue is generally the more important of the six goals. Reading N records to find an exact match for a requested key value on a system where N is extremely large and the storage medium is relatively slow will almost always be worse than the most convoluted conversion. ADAMS excels in this area, primarily because it employs sophisticated tree search.

B. Tree Search

B+-trees are frequently used to index large files [Sal88]. While they provide reasonable access time, there are undesirable characteristics. In particular, search requires in-memory comparison of the search key with individual item keys. This is due to the fact that the key values are not necessarily maintained in the tree in a bit-wise lexicographic order. Character valued keys may be maintained in an ASCII or EBCDIC (for example) lexicographic order, but numeric data keys are more likely to be found in a numeric order. Since traversal of the tree is governed by in-memory comparisons, individual item keys must be represented in the index. Long fixed-length keys may result in large indexes. Variable length keys lead to relatively slow search times.

The ADAMS system uses a related but more advanced index structure known as an O-tree [OrP88]. Rather than storing keys at each level in an O-tree, 8-bit surrogates known as bounding node depths are stored. The smaller entry size leads to larger fan-out, while fixing the size of internal node entries and in so doing making the node structure less chaotic. A further improvement is that keys are maintained in a bit-wise lexicographic order in the tree. This releases us from the need to use in-memory comparison operations and, with proper key design, speeds range searches on numeric (or other) data.

C. Lexicographic ordering of numeric representations

An examination of the bit orderings of one’s and two’s complement formats, real number formats, and ASCII formats provides some insight into the lexicographic ordering problem. The standard numerical data formats partially achieve the lexicographic ordering we seek. The usual representations of positive integers have the bitwise ordered property. If we consider only four bit integer keys, for example, with 0000 as our lowest possible key and 1111 as our greatest, this is easy to see (Figure 1.a). But the ordering fails when comparing positive and negative integers using either one’s or two’s complement notation (Figure 1.b) .

Bitwise lexicographic order	1’s complement	2’s complement
0000	0	0
0001	1	1
0010	2	2
:	:	:
0111	7	7

(a) correct orderings

Bitwise lexicographic order	1’s complement	2’s complement
0111	7	7
1000	-7	-8
1001	-6	-7
:	:	:
1111	0	-1

(b) incorrect orderings

Figure 1
4 bit integers using 1’s and 2’s complement representation

Similarly, standard representations of real numbers are not lexicographically ordered either. For illustrative purposes only we will assume that a standard real number host format consists of a leading sign bit, four exponent bits, and a four bit mantissa; that is *s eeee mmmm*. On a typical platform, such as a SUN workstation, 8 exponent bits and 23 mantissa bits are used to represent a C float [SUN90].

Bitwise lexicographic order	Numeric value
0 0000 1000	1
0 0001 1000	2
0 0001 1010	2.5

(a) Correct ordering

0 0000 0000	0
0 0000 1000	1
1 0000 1000	-1

(b) Incorrect ordering (due to sign bit)

1 0000 1100	-1.5
1 0000 1110	-1.75
1 0000 1111	-1.875

(c) Incorrect ordering (due to signed magnitude problem)

0 0001 1100	3
0 0011 1000	8
0 1001 1000	0.0078125

(d) Incorrect ordering (due to negative exponent)

Figure 2

9 bit real numbers (sign bit, 4 bit exponent, 4 bit mantissa)

The standard real representation not only has the obvious problem of the sign bit, but as illustrated in Figure 2 there are difficulties with negative exponents and with what we will call the signed magnitude problem. The problem with negative exponents is the same as that seen with negative integers. A negative exponent has a higher order than a positive one, and is therefore interpreted as a larger value in the bit-wise lexicographic order.

The idea of the signed magnitude problem is that after one has designated the value as negative with a sign flag the rest of the bits represent a magnitude. A negative value of large magnitude is smaller than a negative value of small magnitude, but the non-sign bits in the representation are ordered as if the number was positive and are in order of magnitude and not value. As a result, the lexicographic ordering of representations of negative real numbers is incorrect.

One could correct the negative exponent problem immediately by doing away with the exponent. A fixed point binary storage format, such as *iiii.ffff,* could be employed. This format uses neither exponent nor mantissa, instead representing the integral and fractional parts of the number seperated by a virtual decimal point. The negative exponent problem is vanquished, but with this simple format comes a severe storage space penalty.

In the case of the SUN workstation, we are required to store data in the range -2^{1023} to 2^{1023}, with magnitudes as small as 2^{-1023} (for a C double)[SUN90]. In the new format, some 2047 bits will be required for each data element in order to ensure no loss of precision and the appropriate ordering (although if a variable length format is used, trailing zeroes need not be stored). If we took a fixed point binary approach with our 9 bit representations from figure 2 our task would be to represent the range 2^7 to - 2^7 with magnitudes as small as 2^{-8} with bit-wise lexicographic ordering. This would require 16 bits as a maximum in order to insure no loss of precision. We would gain the ability to store numbers such as 64.0078125 ($2^6 + 2^{-8}$, not available under the 9-bit scheme), but to allow correct ordering all leading zeroes must be stored for each number (see Figure 3, in which the virtual decimal point has been replaced by a space).

00000001 00000001	(1.0078125)
00010000 00000001	(16.0078125)
00100000 00000001	(32.0078125)
01000000 00000001	(64.0078125)

Figure 3
Fixed point binary real examples

Other problems with this non-standard representation include negative real numbers. While there may be adequate solutions to the lexicographic ordering problem for this particular format, the storage issue, given the SUN target range, is quite enough to reaffirm the value of a mantissa and exponent representation.

“-101.90”	“0.”	“-1264.67”
“.23”	“13234.02”	“-4412.2”
“.6575”	“4753.6”	“-4562.97”
“468.78”	“54111.”	“-7712.07”
“5666.4”	“9”	“-9999.99”
(a) Correct ordering	(b) Incorrect ordering	(c) Incorrect ordering

Figure 4

Variable length ASCII examples

ASCII is a common storage format. Exchange of data between heterogeneous systems is commonly by ASCII files. ASCII representations of real numbers have the obvious drawback that every digit requires one byte of storage, but the representation is simple. Conversion to host format(s) for real numbers (using, for example, C’s scanf) is slow.

But ordering does pose problems for this format. A representation using variable length strings with no leading zeros will be ordered based on a character by character comparison. This can result in correct orderings if the first characters reflect the relative numeric order (Figure 4.a), but incorrect orderings (Figure 4.b) are more likely. The signed magnitude problem also causes incorrect orderings (Figure 4.c).

Employing a fixed length format with a fixed decimal point would correct the first ordering problem (Figure 5.b), but the signed magnitude problem remains (Figure 5.c). In

“-00101.90”	“000000.00”	“-01264.67”
“000000.23”	“000009.00”	“-04412.20”
“000000.65”	“004753.60”	“-04562.97”
“000468.78”	“013234.02”	“-07712.07”
“005666.40”	“054111.00”	“-09999.99”
(a) Correct ordering	(b) Correct ordering	(c) Incorrect ordering

Figure 5

Fixed length fixed decimal point ASCII examples addition, using a fixed format of form $sx_1x_2x_3...x_m.d_1d_2d_3...d_n$ results in a range of only $10^{-n} \le |value| \le 10^m$, at a cost of (m+n+1)*8 bits. If we wished to store the full range of C doubles on a SUN (-2^{1023} to 2^{1023}), not an unusual range for scientific databases, this would require approximately 500 bytes per number . As with the fixed point binary storage format, trailing zeroes need not be stored. But the storage overhead remains intolerable, and ASCII formats must be discarded, despite the obvious attractions of maintaining ASCII only data.

2. A Proposed Solution

Once the standard types are cast away, and the need to store both integer and numeric data in the same format is established, the task becomes clearer. Ignoring integers for the time being, safe in the knowledge that they can be represented as reals, the problem of storing real data in the required ordering remains.There were several problems with standard representations of real numbers, as illustrated in figure 2. The first, and easiest to solve, is the sign bit problem. Negative numbers must lexicographically precede positive numbers, so we define our sign bit as being 0 if the value is negative and 1 otherwise.

A second problem is that of negative exponents. Over our 4-bit example exponent range the lowest bit string lexicographically is 0000 while the highest is 1111. Our values, on the other hand, range from a low of 1000 (-8 in two's complement) to a high of 0111 (7). Adding a fixed bias to all exponents [HeP90] provides a simple order-preserving linear mapping from our value range into our lexicographic range. This is part of the IEEE 754 floating point standard. In our example case we add 1000 to elements in the value range, and this corrects the difficulties shown in figure 2 with positive real numbers having negative exponents.

The solution for the final problem, that of signed magnitude, revolves around the idea that the larger the exponent or mantissa of a negative number, the smaller the value of that number. After our sign bit and biasing operations above, 0 1111 1111 is -240, while 0 1111 1000 is -128 and 0 0001 1000 is -0.0078125. The lexicographic order is the reverse of the value order. The question is how to map large exponent/mantissa values into small lexicographic values while mapping small exponent/mantissa values into large lexicographic values.This mapping can be accomplished by complementing the exponent and mantissa bits for negative numbers.

	original		sign bit op		exponent bias op		man/exp comp op
-1.875	1 0000 1111	=>	0 0000 1111	=>	0 1000 1111	=>	0 0111 0000
-1.5	1 0000 1100	=>	0 0000 1100	=>	0 1000 1100	=>	0 0111 0011
0.0078125	0 1001 1000	=>	1 1001 1000	=>	1 0001 1000	=>	1 0001 1000
8.0	0 0011 1000	=>	1 0011 1000	=>	1 1011 1000	=>	1 1011 1000

Figure 6

Four examples of the conversion to ordered representation

We can quickly show that the complement operation is a linear mapping that takes range 0000 to 1111 to range 1111 to 0000. This is done by observing that if range element A was greater than range element B, due to bit A[i] being 1 while B[i] was 0, then $\bar{B}$[i] is now 1, $\bar{A}$[i] now 0 and $\bar{B}$ is greater than $\bar{A}$. So for all A and B, A>B implies that $\bar{B} > \bar{A}$(also $\bar{B} > \bar{A}$ implies that A>B) and our complementing both mantissa and exponent inverts the magnitudes as desired for our negative real representations.

It is also relatively easy to show that the intended real number ordering is acheived. Consider two real numbers C and D. If both C and D are positive (sign bits is 1) their lexicographic order will be governed by the exponents. If they are unequal, the possesor of the larger exponent is greater in order and in value ($2^k > \sum_{i=0}^{k-1} 2^i$). If the exponents are equal, their order is determined by the mantissas and we have our ordering. If C is positive and D is negative the sign bit (0 for negative, 1 for positive) immediately establishes their relative order. If both C and D are negative, we examine the exponents. If C has a larger exponent than D, then the complement operation has mapped the larger exponent of C from a value $\overline{exp\ (C)}$ smaller than the initial value $\overline{exp\ (D)}$ of D’s exponent. As $-2^{\overline{exp\ (C)}} > -2^{\overline{exp\ (D)}}$ we have our ordering. The same argument holds for negative C and D with equal exponents and unequal mantissas.

We have defined a representation of the real numbers which is bit-wise lexicographically ordered according to their numeric values. No constraints have been placed on the mantissa and exponent sizes, so any chosen degree of precision and magnitude is available. As desired, this representation will allow us to take full advantage of O-trees and their advanced retrieval capabilities.

3. Implementation

The ADAMS system currently employs SUN workstations as its primary hardware platform, and C as the primary host language platform. Unless otherwise stated we will assume a SUN processing environment. The numeric format discussed is well suited for conversion into the double format found in SUN’s C data representations. The system can be ported to different platforms, and any platform using the IEEE 754 floating-point standard [HeP90] will allow straightforward conversion.

The SUN C format for doubles[SUN90]is:

0 bit:	The sign bit. 1 for negative, 0 for positive
1-12 bits:	The exponent. Biased by 1023.
13-63 bits:	The mantissa. A 1 is assumed to be in the first bit, and is not stored.

There are some SUN numbers that do not fit properly into the above format. NaN’s (Not a Number) are stored with an exponent of all 1’s and one of the stored mantissa bits set, infinity is stored with an exponent of 2047 and zeroes in all the stored mantissa bits, zero is stored with all zeroes in both the exponent and mantissa. A denormal or subnormal number (zero exponent, but non-zero mantissa) is stored with 1022 as the exponent. This information is available in the SUN Programmer’s Languages Guides, but is reproduced here for completeness.

The ADAMS storage format (the real number representation discussed above) for numeric data is:

0 bit:	The sign bit. 0 for negative, 1 for positive.
1-12 bits:	The exponent. Biased by 1023. If the numeric data is negative, the bits are complemented before storing.
13-63 bits:	The mantissa. A 1 is assumed to be in the first bit, and not stored. If the numeric data is negative, the bits are complemented before storing.
64-95 bits:	Extra bits. These are available for future expansion, possibly to provide further precision, and are currently stored as 0s.

The other numbers, such as NaN, are stored as discussed above, except for the sign bit flip and exponent/mantissa bit flips (when negative). The C format for Adams numbers is:

```
struct
     {
        unsigned int sign_and_exponent;
        unsigned int fraction1;
        unsigned int fraction2;
     } _A_adams_nbr;
```

The actual code converting the SUN C double to the adams storage format is:

```
int     _A_double_to_internal (double *p_host_double,
                     _A_adams_number *p_result_struct)
        {
        int             positive;
        _A_adams_nbr our_struct;
        union   {
                int    int_rep[2];
                short  short_rep[4];
                double double_rep;
                } multi_rep;
        multi_rep.double_rep = *p_host_double;
        positive =
             (multi_rep.short_rep[0] & 0X8000) == 0;
        our_struct.sign_and_exponent =
              multi_rep.int_rep[0] & 0X7FFFFFFF;
        our_struct.fraction1 = multi_rep.int_rep[1];
        if (!positive)
              {
               our_struct.sign_and_exponent =
                    ~our_struct.sign_and_exponent;
               our_struct.fraction1 =
                    ~our_struct.fraction1;
               our_struct.sign_and_exponent &=
                    0X7FFFFFFF;  // clear sign bit
              }
        else
              {
               our_struct.sign_and_exponent |=
                      0X80000000;  // set sign bit
              }
        our_struct.fraction2 = 0;
        *p_result_struct = our_struct;
        return(1);
        }
```

There are several observations to be made about the above code (other than noting that several lines of comments have been removed). A first comment is that there is not much code; conversion is fast. It is, for example, must faster than C scanf(“%f”,float_value) conversions. One reason for its speed is that the numeric data format was designed to work well on this first SUN C platform. But conversion from any other real format consists of essentially same steps:

1. Check for special cases such as NaN. Convert appropriately.
2. Acquire the sign bit
3. Convert the exponent to a -1022 to 1023 range, biasing by 1023 (IEEE 754 standard for doubles), while maintaining the condition that a 1 is the first (hidden) bit in the mantissa.
4. Reverse the sign bit. Reverse the exponent and mantissa bits if the number was negative.

While ADAMS numeric data is maintained using much of the IEEE 754 format for convenience, it should be clear from the code that the algorithm would work properly for any real number format and any degree of precision or magnitude. All that needs to be done to maintain the ordering is some straightforward bit-level operations.

4. Numeric Data Examples

The following are several examples of pre and post converted numeric data. The pre-conversion data is in SUN C format. The bit strings are given in order, from bit 0 to bit 63 or 95.

```
1.000000
host double format is:
0011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
stored version is:
1011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000

-1.000000
host double format is:
1011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
stored version is:
0100 0000 0000 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000
```

To illustrate the remaining examples we will show just the first 32 bits of both the host and storage formats.

decimal value	host format	storage format
0.000000	0000 0000 0000 0000 0000 0000 0000 0000	1000 0000 0000 0000 0000 0000 0000 0000
1.500000	0011 1111 1111 1000 0000 0000 0000 0000	1011 1111 1111 1000 0000 0000 0000 0000
-1.500000	1011 1111 1111 1000 0000 0000 0000 0000	0100 0000 0000 0111 1111 1111 1111 1111
1.750000	0011 1111 1111 1100 0000 0000 0000 0000	1011 1111 1111 1100 0000 0000 0000 0000
-1.750000	1011 1111 1111 1100 0000 0000 0000 0000	0100 0000 0000 0011 1111 1111 1111 1111
0.000050	0011 1111 0000 1010 0011 0110 1110 0010	1011 1111 0000 1010 0011 0110 1110 0010
-0.000050	1011 1111 0000 1010 0011 0110 1110 0010	0100 0000 1111 0101 1100 1001 0001 1101
0.500000	0011 1111 1110 0000 0000 0000 0000 0000	1011 1111 1110 0000 0000 0000 0000 0000
5000.500000	0100 0000 1011 0011 1000 1000 1000 0000	1100 0000 1011 0011 1000 1000 1000 0000
-5000.500000	1100 0000 1011 0011 1000 1000 1000 0000	0011 1111 0100 1100 0111 0111 0111 1111
2.000000	0100 0000 0000 0000 0000 0000 0000 0000	1100 0000 0000 0000 0000 0000 0000 0000
-2.000000	1100 0000 0000 0000 0000 0000 0000 0000	0011 1111 1111 1111 1111 1111 1111 1111
670000.200000	0100 0001 0010 0100 0111 0010 0110 0000	1100 0001 0010 0100 0111 0010 0110 0000
-670000.200000	1100 0001 0010 0100 0111 0010 0110 0000	0011 1110 1101 1011 1000 1101 1001 1111
0.010000	0011 1111 1000 0100 0111 1010 1110 0001	1011 1111 1000 0100 0111 1010 1110 0001
-0.010000	1011 1111 1000 0100 0111 1010 1110 0001	0100 0000 0111 1011 1000 0101 0001 1110
0.300000	0011 1111 1101 0011 0011 0011 0011 0011	1011 1111 1101 0011 0011 0011 0011 0011
-0.300000	1011 1111 1101 0011 0011 0011 0011 0011	0100 0000 0010 1100 1100 1100 1100 1100
500000.000000	0100 0001 0001 1110 1000 0100 1000 0000	1100 0001 0001 1110 1000 0100 1000 0000
-500000.000000	1100 0001 0001 1110 1000 0100 1000 0000	0011 1110 1110 0001 0111 1011 0111 1111
500001.000000	0100 0001 0001 1110 1000 0100 1000 0100	1100 0001 0001 1110 1000 0100 1000 0100
-500001.000000	1100 0001 0001 1110 1000 0100 1000 0100	0011 1110 1110 0001 0111 1011 0111 1011
5.000000	0100 0000 0001 0100 0000 0000 0000 0000	1100 0000 0001 0100 0000 0000 0000 0000
-5.000000	1100 0000 0001 0100 0000 0000 0000 0000	0011 1111 1110 1011 1111 1111 1111 1111
46.287600	0100 0000 0100 0111 0010 0100 1101 0000	1100 0000 0100 0111 0010 0100 1101 0000
-46.287600	1100 0000 0100 0111 0010 0100 1101 0000	0011 1111 1011 1000 1101 1011 0010 1111
4.000000	0100 0000 0001 0000 0000 0000 0000 0000	1100 0000 0001 0000 0000 0000 0000 0000
-4.000000	1100 0000 0001 0000 0000 0000 0000 0000	0011 1111 1110 1111 1111 1111 1111 1111

5. Sorting Numeric Data under ADAMS

The above ADAMS numeric format bit strings were processed by a sorting program in order to verify the bit-wise lexicographic order. Note that for clarity the last 32 bits of each entry are not shown (each of those bits is currently defined to be 0). The resulting list was:

```
-670000.200000 0011 1110 1101 1011 1000 1101 1001 1111 1001 1001 1001 1001 1001 1001 1001 1001
-500001.000000 0011 1110 1110 0001 0111 1011 0111 1011 1111 1111 1111 1111 1111 1111 1111 1111
-500000.000000 0011 1110 1110 0001 0111 1011 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-5000.500000   0011 1111 0100 1100 0111 0111 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-46.287600     0011 1111 1011 1000 1101 1011 0010 1111 1110 1100 0101 0110 1101 0101 1100 1111
-5.000000      0011 1111 1110 1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-4.000000      0011 1111 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-2.000000      0011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-1.750000      0100 0000 0000 0011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-1.500000      0100 0000 0000 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-1.000000      0100 0000 0000 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
-0.300000      0100 0000 0010 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
-0.010000      0100 0000 0111 1011 1000 0101 0001 1110 1011 1000 0101 0001 1110 1011 1000 0100
-0.000050      0100 0000 1111 0101 1100 1001 0001 1101 0001 0100 1110 0011 1011 1100 1101 0010
0.000000      1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0.000050      1011 1111 0000 1010 0011 0110 1110 0010 1110 1011 0001 1100 0100 0011 0010 1101
0.010000      1011 1111 1000 0100 0111 1010 1110 0001 0100 0111 1010 1110 0001 0100 0111 1011
0.300000      1011 1111 1101 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011
0.500000      1011 1111 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1.000000      1011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1.500000      1011 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1.750000      1011 1111 1111 1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
2.000000      1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
4.000000      1100 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
5.000000      1100 0000 0001 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
46.287600      1100 0000 0100 0111 0010 0100 1101 0000 0001 0011 1010 1001 0010 1010 0011 0000
5000.500000   1100 0000 1011 0011 1000 1000 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000
500000.000000  1100 0001 0001 1110 1000 0100 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000
500001.000000  1100 0001 0001 1110 1000 0100 1000 0100 0000 0000 0000 0000 0000 0000 0000 0000
670000.200000  1100 0001 0010 0100 0111 0010 0110 0000 0110 0110 0110 0110 0110 0110 0110 0110
```

As expected, the list is in bit-wise lexicographic order.

6. Conclusions

Several objectives for a succesful storage scheme have been met. The proposed scheme provides for a compact representation of high precision and magnitude numeric data, while allowing for future expansion if needed.

There is a single numeric data format, which allows bitwise lexicographic ordering to be easily imposed on numeric data. This ordering facilitates the use of O-trees, and therefore reaps the rewards of faster range and exact match search while saving space. A drawback to the single representation is that some host data typing benefits may be lost. The ADAMS compiler will not be able to prevent, for example, an ADAMS programmer from trying to convert a large magnitude real value into a short integer. Many languages permit such type intermixing, so this particular issue is not critical. But it is worth noting that the programmer is responsible for knowing what he can do with his data in his host environment.

While there may be a future non-standard platform in which this is not the case, the conversion from host format to ADAMS storage format should be fast and simple on most platforms. Conformity with IEEE standards should help to ensure this benefit. ADAMS provides storage and retrieval functions, and relies on the host language for mathematical operations. Code to provide direct operations on the stored data could be written, but the fastest and most reliable approach would be to convert the data to the host format and rely on the host for this functionality.

7.References

[HeP90] D. Patterson and J. Hennessy, *Computer architecture: a quantitative approach* (Morgan Kaufmann Publishers, Inc. San Mateo, CA 1990).

[OrP88] R. Orlandic and John Pfaltz,”Compact 0-Complete Trees,” Proceedings of the 14th VLDB Conference, Los Angeles, CA.(1988).

[Sal88] B. Salzberg, *File structures: an analytic approach* (Prentice-Hall, Inc. Englewood Cliffs, NJ, 1988).

[SUN90] Programmer’s Language Guides, SUN Microsystems, Inc. (1990)