# How the SQL Server hierarchyid Data Type Works (kind of)

and how to design your own

by Adam Milazzo (<http://www.adammil.net>)

Yesterday, I stumbled across the SQL Server hierarchyid data type (<http://msdn.microsoft.com/en-us/library/bb677290.aspx>). It stores the position of an object within a tree. I was intrigued because I’ve developed similar mechanisms for efficiently querying hierarchical data, and I wanted to see how it compares. I’m always happy to offload complexity to other software when I can trust it to do the job right, and I trust SQL Server.

A hierarchyid is very compact, capable of storing a position within a tree containing 100,000 nodes with a branching factor of 6 in less than 39 bits on average. Interestingly, it supports arbitrary insertions and deletions without ever needing to update any rows (besides the one being inserted/deleted). This means you can always generate a new hierarchyid before or after any other ID, or between any two sibling IDs. Hierarchyids are designed so that sorting a set of hierarchyids puts them in the order that they would be visited on a depth-first traversal of the tree, which is the usual approach.

Hierarchyids are represented in text with the following scheme:

* / is the root.
* /1/, /2/, /-20/, etc. are children of the root. /-20/ comes before /1/, which comes before /2/.
* /1/2/ is a grandchild of the root.
* /1.3/ is a value between /1/ and /2/, and /1.3/100/ is a child of that value.
* /1.-5.3/ lies between /1.-5.2/ and /1.-5.4/. It’s also between /1.-5/ and /1.-4/, /1/ and /2/, etc.

I’ll assume that you’re familiar with bits and bytes. A hierarchyid is a bit string which is left-packed into bytes to allow them to be compared directly (i.e. by comparing byte values). Trailing zeros are not part of the ID. If you look at the binary values corresponding to hierarchyids, you’ll notice some patterns. (Note that the hex values often do not match the bit strings. This is because the hex values represent the actual bytes stored, while the bit strings have had trailing zeros removed.)

|  |  |
| --- | --- |
| / | (zero bytes – 0x) |
| /0/ | 01001 (0x48) |
| /1/ | 01011 (0x58) |
| /2/ | 01101 (0x68) |
| /3/ | 01111 (0x78) |
| /4/ | 100001 (0x84) |
| /5/ | 100011 (0x8C) |
| /6/ | 100101 (0x94) |
| /7/ | 100111 (0x9C) |
| /8/ | 1010001 (0xA2) |
| /9/ | 1010011 (0xA6) |
| /10/ | 1010101 (0xAA) |
| /11/ | 1010111 (0xAE) |
| /12/ | 1011001 (0xB2) |
| /13/ | 1011011 (0xB6) |
| /14/ | 1011101 (0xBA) |
| /15/ | 1011111 (0xBE) |
| /16/ | 110000010001 (0xC110) |
| /17/ | 110000010011 (0xC130) |
| /18/ | 110000010101 (0xC150) |
| /19/ | 110000010111 (0xC170) |
| /20/ | 110000011001 (0xC190) |
| /21/ | 110000011011 (0xC1B0) |
| /22/ | 110000011101 (0xC1D0) |
| /23/ | 110000011111 (0xC1F0) |
| /24/ | 110000110001 (0xC310) |
| /32/ | 110010010001 (0xC910) |
| /40/ | 110010110001 (0xCB10) |
| /48/ | 110100010001 (0xD110) |
| /56/ | 110100110001 (0xD310) |
| /64/ | 110110010001 (0xD910) |
| /72/ | 110110110001 (0xDB10) |
| /80/ | 111000000000010001 (0xE00440) |
| /88/ | 111000000000110001 (0xE00C40) |
| /96/ | 111000000010010001 (0xE02440) |
| /128/ | 111000000110010001 (0xE06440) |
| /136/ | 111000000110110001 (0xE06C40) |
| /192/ | 111000001110010001 (0xE0E440) |
| /320/ | 111000101110010001 (0xE2E440) |
| /576/ | 111001101110010001 (0xE6E440) |
| /1088/ | 111011101110010001 (0xEEE440) |
| /1104/ | 111100000000000010001 (0xF00088) |
| /2128/ | 111100100000000010001 (0xF20088) |
| /3152/ | 111101000000000010001 (0xF40088) |
| /4176/ | 111101100000000010001 (0xF60088) |
| /5200/ | 1111100000000000000000000000000000000010001 (0xF80000000220) |

In particular, within certain ranges there are bits that increment in the usual way, and there are bits that seem to be constant.

|  |  |
| --- | --- |
| /0/ through /3/ | 01xx1 |
| /4/ through /7/ | 100xx1 |
| /8/ through /15/ | 101xxx1 |
| /16/ through /79/ | 110zz0y1xxx1 |
| /80/ through /1103/ | 1110aaa0zzz0y1xxx1 |
| /1104/ through /5199/ | 11110aaaaa0zzz0y1xxx1 |

The values for /0/ through /15/ are straightforward enough, but when we get to /16/, something strange happens. The bits that increment appear to get broken up. When you look at further ranges, you see other apparent breakups, where bits are grouped and separated by zeros. But the bits increment normally, with carry between groups. What is going on here?

First of all, as you can see, a varying number of bits are used to encode the numbers. This is useful, as many trees are simply binary trees, and those that aren’t typically have small branching factors. But then it needs a way to know how many bits are in a particular number. It seems that hierarchyids use a prefix-free code at the beginning of each number to distinguish them: 01..., 100…, 101…, 110…, 1110…, 11110…, etc. So it knows that when it sees a 01 or 100, then it needs to read three more bits, and when it sees 101, it needs to read 4 more bits, etc.

Second, given that the endings in the broken-up patterns seem to be the same (always 0y1xxx1), I suspect that perhaps values are computed in multiple stages. First, the value of 1xxx1 is computed. This is added to 8y, and then the result is added to 16z (if it exists), which is added to 128a or 64a (if it exists), etc. Finally, the result is added to a constant which marks the beginning of its range (01xx1 starts at 0, 100xx1 at 4, etc). So perhaps the numbers can be broken down this way: prefix (n+0)\* y1xxx1. That is, a prefix followed a number (possibly zero) of groups, each of which contains some bits followed by a zero, and ending with y1xxx1.

What I don’t understand is why the numbers have such a strange pattern. Why not simply use 01xx1, 100xx1, 101xxx1, 1110xxxxxxxxxxxxx1, etc? Why separate the groups with zeros? It seems like a waste of space, but I assume there’s a good reason. There’s at least a good reason for the 1 bit at the end, which I’ll get to later.

Let’s consider negative numbers. Normally, numbers on computers are represented in two’s complement (<http://en.wikipedia.org/wiki/Two%27s_complement>). But in two’s complement, negative numbers are greater than positive numbers when you consider their bit strings. That is, the unsigned value of a negative integer is greater than the unsigned value of a positive integer. But hierarchyids need negative numbers to have bit string values less than those of positive numbers. One way to do this is to use a number representation where 0 is represented as a 1 followed by zeros. For instance, with four bits, 0 is represented as 1000. Addition and subtraction work as normal, so 1 through 7 are 1001 through 1111, and crucially, -1 is 0111, -2 is 0110, etc. Let’s look at some more bits.

|  |  |
| --- | --- |
| /-73/ | 0001101111101110111111 (0x1BEEFC) |
| /-72/ | 0010000010001 (0x2088) |
| /-64/ | 0010000110001 (0x2188) |
| /-56/ | 0010010010001 (0x2488) |
| /-48/ | 0010010110001 (0x2588) |
| /-40/ | 0010100010001 (0x2888) |
| /-32/ | 0010100110001 (0x2988) |
| /-24/ | 0010110010001 (0x2C88) |
| /-16/ | 0010110110001 (0x2D88) |
| /-10/ | 0010110111101 (0x2DE8) |
| /-9/ | 0010110111111 (0x2DF8) |
| /-8/ | 001110001 (0x3880) |
| /-7/ | 001110011 (0x3980) |
| /-6/ | 001110101 (0x3A80) |
| /-5/ | 001110111 (0x3B80) |
| /-4/ | 001111001 (0x3C80) |
| /-3/ | 001111011 (0x3D80) |
| /-2/ | 001111101 (0x3E80) |
| /-1/ | 001111111 (0x3F80) |
| /0/ | 01001 (0x48) |
| /-1/ through /-8/ | 00111xxx1 |
| /-9/ through /-72/ | 0010zz0y1xxx1 |

It doesn’t really use the scheme I expected. You can see that the prefixes continue: 01…, 0011…, 0010…, 00010…. I think negative bit strings are longer than positive ones because there are likely to be far fewer negative numbers, so they can make better use of prefix code bits this way. The patterns are quite similar to the positive case, with a prefix code, followed by some zero-separated groups of bits, followed by 1xxx1.

Let’s take a break from negative numbers for now, and look at that trailing 1 bit. First, let’s examine the way numbers are combined to form separate levels (e.g. /1/2/) and fractions (e.g. /1.1/). Thankfully, this is relatively easy to understand.

|  |  |  |  |
| --- | --- | --- | --- |
| /0/0/0/ | 01001 01001 01001 | /3.0/ | 100000 01001 |
| /0/1/2/ | 01001 01011 01101 | /3.1/ | 100000 01011 |
| /0.0.0/ | 01010 01010 01001 | /4.0/ | 100010 01001 |
| /0.1.2/ | 01010 01100 01101 | /14.0/ | 1011110 01001 |
| /0.0/0.0/ | 01010 01001 01010 01001 | /15.0/ | 110000010000 01001 |

The parent/child relationship is represented by simple concatenation. /0/ is 01001 and so /0/0/ is 01001 01001. This makes sense. Next, look at the dot patterns. At first, it seems that the last number in a dotted sequence is in its normal form, and the previous numbers are their normal forms plus one. This is quite clever. Since all the normal forms end in 1, adding one yields a string ending in zero. Given that /0/ is 01001, /0.X/ is 01010 X, and /1/ is 01011, you can see that all IDs /0/\*/ will be less than all IDs /0.X/\*/, which are themselves less than /1/.

Things get complicated when we get to /3.0/, however. /3/ is 01111, and adding one would normally produce 10000, but it instead produces 100000. I believe this is to handle the following scenario. Consider /3.4/. If the 3. did produce 10000, then the result would be 10000100001. This would be greater than /4/, which is 100001, but it’s supposed to be less. Let’s look at another case. /15/ is 101111, but /15.0/ produces 110000010000 01001. This time it isn’t simply a case of tacking on another zero. However, /16/ is 110000010001, so I think we can reformulate the /X.Y/ rule as “output /X+1/ *minus* one, and then output /Y/”. So for /3.0/, we take /3+1/ (i.e. /4/), which is 100001, and subtract one to get 100000. This works for all the cases we’ve seen so far.

Now we have enough information to parse and generate hierarchyids and combine them, using parent/child and sibling relationships, assuming we know the pattern for each prefix code, and those can be discovered empirically. Still unsolved is the mystery of why the patterns are so strange, and in particular, why there are seemingly constant zeros embedded within them. To see if we can figure it out, let’s try removing them and seeing if we run into trouble. We’ll design our own scheme, with the following patterns: 01xx1, 10xxxx1, and 110xxxxxxxx1. For negative numbers, we’ll use the following: 001xx1, 0001xxx1, and 00001xxxxxxxxxx1. (These were chosen more to help detect problems than for efficient coding.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| /0/ | 01001 | /0/ | 01001 | /0.0/ | 01010 01001 |
| /1/ | 01011 | /-1/ | 001111 | /1.0/ | 01100 01001 |
| /2/ | 01101 | /-2/ | 001101 | /2.0/ | 01110 01001 |
| /3/ | 01111 | /-3/ | 001011 | /3.0/ | 1000000 01001 |
| /4/ | 1000001 | /-4/ | 001001 | /4.0/ | 1000010 01001 |
| /5/ | 1000011 | /-5/ | 00011111 | /5.0/ | 1000100 01001 |
| /6/ | 1000101 | /-6/ | 00011101 | /6.0/ | 1000110 01001 |
| /7/ | 1000111 | /-7/ | 00011011 | /7.0/ | 1001000 01001 |
| /8/ | 1001001 | /-8/ | 00011001 | /18.0/ | 1011110 01001 |
| /9/ | 1001011 | /-9/ | 00010111 | /19.0/ | 110000000000 01001 |
| /10/ | 1001101 | /-10/ | 00010101 | /-1.0/ | 01000 01001 |
| /11/ | 1001111 | /-12/ | 00010011 | /-2.0/ | 001110 01001 |
| /12/ | 1010001 | /-13/ | 00010001 | /-3.0/ | 001100 01001 |
| /13/ | 1010011 | /-14/ | 0000111111111111 | /-4.0/ | 001010 01001 |
| /14/ | 1010101 | /-15/ | 0000111111111101 | /-5.0/ | 001000 01001 |
| /15/ | 1010111 | /-16/ | 0000111111111011 | /-6.0/ | 00011110 01001 |
| /16/ | 1011001 | /-17/ | 0000111111111001 | /-7.0/ | 00011100 01001 |
| /17/ | 1011011 | /-18/ | 0000111111110111 | /-8.0/ | 00011010 01001 |
| /18/ | 1011101 | /-19/ | 0000111111110101 | /-9.0/ | 00011000 01001 |
| /19/ | 1011111 | /-21/ | 0000111111110011 | /-14.0/ | 00010000 01001 |
| /20/ | 110000000001 | /-22/ | 0000111111110001 | /-15.0/ | 0000111111111110 01001 |
| /275/ | 110111111111 | /-23/ | 0000111111101111 | /-16.0/ | 0000111111111100 01001 |

Is this a valid scheme? It seems so. I reason that, given two bit strings, either the prefix codes of the first number match or they don’t. If they don’t match, then we can compare the strings by comparing the prefix codes, since we’ve chosen increasing codes for higher numbers. If the prefix codes do match, then the first two numbers are the same length in bits, and are exactly lined up, so they can be directly compared. If they are equal, then we can examine the next pair of numbers and repeat this logic. The scheme also shares the same logic for combining IDs. So why, then, does SQL Server use an excessively complex and seemingly wasteful formula? I honestly don’t know. If I had to venture a guess, it would be that the zero bits allow creating IDs between two other IDs without adding more bits, but I haven't observed this, and I’ve looked for it.

The above scheme can be optimized. The order of siblings is relevant when using hierarchyids, and tracking that uses substantial space. Most systems storing hierarchical data only need to know about ancestry relationships. We can redesign the above code so that it doesn’t keep track of order to yield one with substantially higher efficiency. We’ll use the following patterns: 0xx, 100xx, 101xxx, 110xxxxx, 1110xxxxxxxx, 11110xxxxxxxxxxx, and 11111xxxxxxxxxxxxxxxxxxx.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | /12/ | 101101 |
| /0/ | 001 | /13/ | 101110 |
| /1/ | 010 | /14/ | 101111 |
| /2/ | 011 | /15/ | 11000000 |
| /3/ | 10000 | /16/ | 11000001 |
| /4/ | 10001 | /17/ | 11000010 |
| /5/ | 10010 | /46/ | 11011111 |
| /6/ | 10011 | /47/ | 111000000000 |
| /7/ | 101000 | /302/ | 111011111111 |
| /8/ | 101001 | /303/ | 1111000000000000 |
| /9/ | 101010 | /2350/ | 1111011111111111 |
| /10/ | 101011 | /2351/ | 111110000000000000000000 |
| /11/ | 101100 | / 526638/ | 111111111111111111111111 |

Although I haven't put much work into optimizing it, this provides a good balance between overall capacity and efficiency with typical branching factors. The hypothetical 100,000 node tree with a branching factor of 6 that a hierarchyid can handle using about 39 bits on average takes only 24 bits on average with this scheme, a savings of almost 40%. Note that 000 is unused. This is partly due to SQL Server's disgusting behavior of ignoring trailing zeros when comparing varbinary values. (It considers it true that 0x10 = 0x1000 = 0x100000!) So you can't use codes without any 1 bits in them, or else SQL Server may fail to compare them correctly. But also, strings containing 6 or more contiguous zero bits would be ambiguous. (E.g. 0x00 could be /0/ or /0/0/.)

If you know that your data is a binary tree, you can use just one bit per level. (You might want to use an int or bigint column to store the ID, then.) You can do lots of things to adapt the scheme to your data, but I think the best idea would be to use hierarchyids for when you need to keep track of order among siblings (although the first scheme we developed can handle it better, if you adjust its coding), and create a second general scheme for the common case of trees where order among siblings doesn’t matter.

As a closing note, I’ll mention that I have probably made some mistakes, perhaps serious ones, while creating this document, and if anybody can find a flaw in the ideas behind the schemes I developed and/or explain the reason for the hierarchyid’s oddities, I would greatly appreciate it.