Application Note: Value Sums

6/22/2001

Problem

How can the sum of a set of array values over large, arbitrary, spans of data quickly be determined? That is, how can the sum be determined faster than summing each individual value?

Explanation

`JLC()` (the `JudyLCount()` macro) allows you to count the number of data elements (indexes) in a JudyL array between any two indexes. However, there is no function to sum the element values within this span of data. The summing of values between any two indexes allows you to compute the data intensity (or density) of that value over the given index range.

Suppose that each JudyL array value records the number of times a corresponding index was inserted. Then a sum of the values between any pair of indexes reveals the density of indexes seen in that range.

This might be useful, say, to look for "massive clusters" of user data points (indexes). Suppose each index is the IP address of a client system, and each value is the number of server requests initiated by that IP address. Counting by values can be used to find "highly active" subnets.

Or, suppose that each index represents a code address in a software program, and each value is the number of "hits" from a program counter sampling mechanism. Counting by values can be used to isolate "hot" code sequences for performance tuning.

Solution

Consider an array with the following indexes and values (missing indexes have an implied value of 0):

Index	Value
5	1
6	2
8	2
20	1
100	2
102	10
103	14
105	7
106	1
200	10

The data show a small "blip" between indexes 5 and 8, an insignificant isolated hit at index 20, an interesting cluster between indexes 100 and 106, and an isolated spike at index 200.

Using current techniques, one could visit each index and sum the values, yielding a grand total of 50. Successive subsections could be traversed to obtain subsection sums and eventually locate the areas of high density. To make this brute force method of successive summing work well over large numbers of values you need a very fast value counting technique.

Now consider the same array using the base-2 (binary) representation of the values.

Index	Value (base-10)	Value (base-2)			
5	1				1
6	2			1	0
8	2			1	0
20	1				1
100	2			1	0
102	10	1	0	1	0
103	14	1	1	1	0
105	7		1	1	1
106	1				1
200	10	1	0	1	0
Column sum		3	2	7	4

The sum of the values can be calculated by summing each base-2 column, multiplying that column sum by the appropriate power of 2 (1, 2, 4, 8, ...) and summing the result. (Note that multiplying by powers of 2 is implemented very efficiently by bit shifting.)

Starting with the rightmost column, the grand total is represented as:

$$(4 * 2^0) + (7 * 2^1) + (2 * 2^2) + (3 * 2^3) = 50$$

To sum the values between indexes 100 and 110:

$$(2 * 2^0) + (4 * 2^1) + (2 * 2^2) + (2 * 2^3) = 34$$

Summing a column of the base-2 representation is the same as counting the number of indexes of that column that contain non-zero values. If, instead of storing the values in a JudyL array, they are stored as columns in separate Judy1 bit arrays, then each column sum can be achieved by using the fast index counting already present in the current Judy1 implementation. Using this method, a value count between two indexes in this "striped" array of 32-bit values can be achieved with at most 32 calls to the `J1C()` (`Judy1Count()`)

function, with subsequent shifting and adding of the results. You can see that there is nothing magic about 32 bits in this context. You could have 51 bit counters or 3 bit counters if you wish.

The advantage of these "striped" array counters becomes apparent when you consider very highly populated arrays where the time to visit each index and sum the value becomes prohibitively expensive. The cost of visiting each index increases linearly with the population of the array. The time required by the "striped" method grows at a much slower logarithmic rate because it uses the fast index counting already present in the Judy design.

Caveat

The cost for this fast counting is the time for incrementing new values in a "striped" array as well as the space cost of those arrays. Also, the "fast" count may be slower than simply counting each value in an array if the number of values is very small. Clearly, the benefits of this technique are very application dependent. Profile, wisely.

Example Code

This example code shows two functions, `increment()` and `count()`.

The `increment()` function takes an index and increments the value associated with that index. It implements a "striped" binary counter across 32 Judy1 arrays. The average number of Judy1 arrays that are modified is 2.

```
#define CBITS 32            /* counter bits     */
Pvoid_t parray_exp[CBITS];  /* initialized to 0 */
long
increment(Word_t index)
{
    int bit, rv;
    for (bit = 0; bit < CBITS; ++bit) {
        J1S(rv, parray_exp[bit], index);

        if (rv == 1)break;  /* was unset; now set */

        J1U(rv, parray_exp[bit], index); /* was set; now unset */
    }
    if (bit >= CBITS) {
        printf("overflow\n");
        exit(2);
    }
    return 0;
}
```

The increment algorithm can be generalized as: Starting at the LSB, toggle the bit; if the new value is 1 then end, else move left to the next bit and repeat.

The `count()` function takes two indexes and returns the sum of the values contained between those two indexes including any values associated with the indexes themselves.

```
Word_t
count(Word_t Index1, Word_t Index2){

    Word_t rv, total = 0;
    int bit;

    for (bit = 0; bit < CBITS; ++bit) {

        if (parray_exp[bit] != (Pvoid_t)NULL){
            J1C(rv, parray_exp[bit], Index1, Index2);
            total += (rv << bit);
            }
        }
        return (Word_t)total;
    }
```