Two-Stack Sliding-Window Aggregation

2026-10-02

An aggregation is some kind of summary of a set of data. This can be the sum, length, minimum, etc. It is quite common to want to calculate such a summary repeatedly, e.g. “the maximum noise level in dB for the past 30 seconds” for a nuisance detector. In such a case we say there is a *sliding window* over our data, and we want to aggregate over our window.

If our aggregation is a binary operator with an inverse, like integer sums, there is a very easy solution using a double-ended queue:

```
from collections import deque

class SlidingWindowSum:
    def __init__(self):
        self.sum = 0
        self.elems = deque()

    def push(self, x):
        self.sum += x
        self.elems.append(x)

    def pop(self):
        self.sum -= self.elems.popleft()

    def eval(self):
        return self.sum
```

But what if our operator has no inverse? This is actually the case for most interesting summaries such as minimum, quantile, approximate unique count (for example using HyperLogLog), etc. In fact, even something as simple as a *floating-point* sum suffers from the fact that floating-point addition is not invertible. For example, if you ever have a `NaN` in your input data with the above naive algorithm your sum will forever remain `NaN`, even long after the bad value has left your window.

Six years ago I came up with an algorithm for maintaining just the minimum/maximum in a sliding window and posted it to cs.stackexchange. I now consider this algorithm pointless, because it turns out there is a simple and efficient algorithm that solves this problem for a very wide class of aggregations. I’m writing this blog post to spread the word, because I feel it should be more widely known.

Folklore

I came across this algorithm while reading a far more advanced paper, *Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time* by Tangwongsan et al. Why is this paper titled low-latency? Because it does the same as what I'm about to describe, but in O(1) time for each step. However, in it they also described a "two-stack" algorithm, which does it in *amortized* O(1), and is far, far simpler.

Amortized O(1) means that across many operations the total amount of work per element is constant, but an individual operation can take much longer. This is almost always fine, unless you absolutely need a low upper bound on latency.

Funnily enough that paper attributes this algorithm to "adamax" from a 2011 Stack Overflow post. They in turn credit a 2001 lecture note by D. Sleator for the inspiration. However, this lecture note does not describe a sliding window aggregate, it describes the classical two-stack algorithm for implementing a FIFO queue and does amortized analysis on it. Ultimately I would not be surprised to find that this algorithm was already described in an obscure paper from the 1970s, seeing how simple and brilliant it is.

Two stacks

Like the authors of the paper, I will generalize the two-stack algorithm to arbitrary associative aggregation functions. By abstracting the aggregation as a set of functions, `empty()`, `unit(x)`, `combine(x, y)` and `finalize(x)`, you can describe many possible aggregations, for example a mean:

```
empty = lambda: (0, 0)
unit = lambda x: (x, 1)
combine = lambda x, y: (x[0] + y[0], x[1] + y[1])
finalize = lambda x: x[0] / x[1] if x[1] else None
```

I'd like to note here that these functions have the following signatures:

```
fn empty() -> Agg;
fn unit(x: Value) -> Agg;
fn combine(x: Agg, y: Agg) -> Agg;
fn finalize(x: Agg) -> Out;
```

I'm making a distinction here between `Value`, `Agg` and `Out` because while they seem superficially similar for something like an integer sum, for an approximate unique count on strings you would have `(Value, Agg, Out) = (String, HyperLogLogSketch, u64)`, three wildly different types.

Without further ado, the algorithm:

```
class TwoStackAgg:
    def __init__(self):
        self.values = []
        self.values_agg = empty()
        self.cum_aggs = []

    def push(self, x):
        self.values.append(x)
        self.values_agg = combine(self.values_agg, unit(x))

    def pop(self):
        if not self.cum_aggs:
            cum_agg = empty()
            while self.values:
                cum_agg = combine(unit(self.values.pop()), cum_agg)
                self.cum_aggs.append(cum_agg)
            self.values_agg = empty()
        self.cum_aggs.pop()

    def eval(self):
        return finalize(
            combine(self.cum_aggs[-1], self.values_agg)
            if self.cum_aggs else self.values_agg
        )
```

That's it, the entire algorithm. There's two stacks (`values` and `cum_aggs`) and one more aggregate, `values_agg`. At any point in time `values_agg` holds the aggregate of `values`, and `cum_aggs` contains the cumulative aggregates of all values in our window that *aren't* in `values`, in *reverse* order. From this we can get the aggregate over our entire window in constant time by by combining the last value of `cum_aggs` with `values_agg`.

The neat part is that (assuming `w` is our window size) every `w`th operation we drain all of `values` and maintain a running aggregate while pushing the partial cumulative aggregates onto `cum_aggs`. This is what makes it *amortized* O(1), doing `O(w)` internal operations every `w`th pop bounds the total amount of work per element to `O(1)`, even though a singular operation might not be constant time.

I think this is best visualized. Suppose we sum `[1, 2, ..., 10]` with a fixed-size sliding window of four elements, then the state on each `eval()` call would look like this (`values_agg` not shown as it is simply the aggregate of the values):

```
             cum_aggs values        out
                   [] []            = 0
                   [] [1]           = 1
                   [] [1, 2]        = 1 + 2
                   [] [1, 2, 3]     = 1 + 2 + 3
                   [] [1, 2, 3, 4]  = 1 + 2 + 3 + 4
[4, 3 + 4, 2 + 3 + 4] [5]           = 2 + 3 + 4 + 5
           [4, 3 + 4] [5, 6]        = 3 + 4 + 5 + 6
                  [4] [5, 6, 7]     = 4 + 5 + 6 + 7
                   [] [5, 6, 7, 8]  = 5 + 6 + 7 + 8
[8, 7 + 8, 6 + 7 + 8] [9]           = 6 + 7 + 8 + 9
           [8, 7 + 8] [9, 10]       = 7 + 8 + 9 + 10
                  [8] [9, 10]       = 8 + 9 + 10
                   [] [9, 10]       = 9 + 10
                 [10] []            = 10
                   [] []            = 0
```

In total the memory usage is `O(w)`, where `w` is your maximum window size. Note that for simplicity of analysis and the example I assumed a fixed-size window `w`, but there is nothing about the two-stack algorithm that requires this. You can call `push(x)` and `pop()` as many times as you'd like between each `eval()`, growing and shrinking the window size as needed.

Floating-point non-associativity

Note that we required above that our aggregate `combine` is associative, meaning:

```
combine(combine(x, y), z) = combine(x, combine(y, z))
```

Technically speaking, floating-point addition doesn't respect this. Nevertheless, the above algorithm is still very useful because the results closely match the expected outcome, even more so if you use a compensated summation algorithm like Kahan summation.

Another neat thing about the two-stack algorithm is that it doesn't require commutativity, if you follow the above implementation precisely. The order of operands is maintained, which can matter for things like string concatenation.

However, there is a second very useful property of the above algorithm. Each aggregate is strictly a combination of the elements in the window, and none outside the window. This means if your window contains a `NaN` or infinity (or some other outlier), that value only poisons the windows that contain it rather than the rest of your computation.

But even without `NaN` or infinity it is useful, due to not propagating errors endlessly. E.g. if your sliding window starts with `[1e20, 1]`, this is what would happen with a naive rolling sum:

```
>>> 1e20 + 1 - 1e20 - 1
-1.0
```

Compensated summation will reduce these effects, but not making your result depend on values outside of the window will eliminate long-term error accumulation entirely.

← Prev post Archive Next post →