To nest or not to nest, that is the question. An argument that NCList is unnecessary. BH 2019.08.09 rev.

Just for discussion; based on *Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases*, Alexander V. Alekseyenko and Christopher J. Lee, Vol. 23 no. 11 **2007**, pages 1386–1393. doi:10.1093/bioinformatics/btl647

These authors present a nested approach to finding overlapping intervals within a potentially very large set of intervals. They argue that in a situation such as given in their Fig. 2 (Figure 1), it is better to nest interval *y* into *x* than to leave it in the initially searched list.

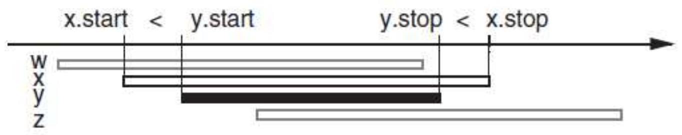


Fig. 2. Any contained interval breaks sortedness. If intervals (boxes) are sorted on their start coordinate, then any interval *y* (filled box) that breaks sortedness of the stop coordinate is properly contained in another interval *x*.

Figure 1. The original issue leading to the development of NCList.

The authors state the problem in this way (bold mine):

> *Interval query can be slow because the overlapping intervals for any given query may not be contiguous in standard indexing.* ***Therefore, the database query cannot stop at the first non-overlapping interval, but must scan the rest of the database.***

And their solution:

> *We can easily solve this problem by realizing that it is caused solely by the intervals that are contained within other intervals, i.e. x. start < y. start y.stop < x.stop. (see Fig. 2). If a sorted list of intervals has both start and stop coordinates in ascending order, then the overlapping intervals for any query are guaranteed to be contiguous in the list.*

The solution is clever. By partitioning the set into sets of fully contained intervals, NCList's findOverlap(start,end) method involves *n* binary searches that are of sets potentially smaller than *N*, each followed by a linear contiguous scan. For example, consider using NCList, for all overlaps of interval *I,* as shown in (my) Figure 2. A binary search followed by a linear scan finds *w*, *x,* and *z*. A second binary search and linear scan through the subintervals of *x* finds *y*.

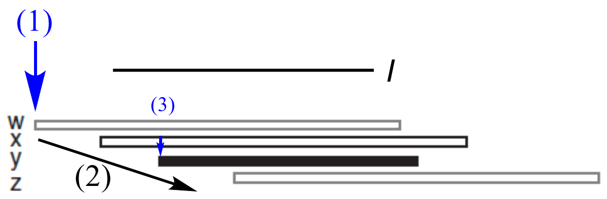


Figure 2: NCList looking for overlaps with *I* finds *w* first using a binary search for the closest end point to the start of *I* (1), then scans forward for intervals that have a starting point before the end of *I* (2), finding *x* and *z*. Interval *y* is found within *x* using a second binary search (3), followed again by a second forward search (not shown, because in this case *y* is the only subinterval of *x*).

ASIDE: I note that a minor efficiency has been missed here. The binary search is for an interval, not a number. Any one of *w*, *x*, or *z* would be a suitable end point for the first binary search, since all intervals that overlap must be contiguous. However, the difference in performance appears to be insignificant.

Jalview's NCList implementation – IntervalStore

The implementation of NCList in Jalview, IntervalStore<SequenceFeature> is identical to the original specification, with the added efficiency that during initialization all intervals that have no subintervals are set aside in a separate list. While this adds complications when adding or removing intervals from the set, it is more efficient in terms of storage (just a simple ArrayList<SequenceFeature>). The Jalview implementation is further optimized by separating different *types* of features (plaf, variant, etc.) into separate FeatureStore objects, each with their own IntervalStore. This substantially reduces the number of levels of nesting as well as the number of intervals in any single binary search.

But do we have to nest?

There is another, considerably simpler, solution involving a linked list with similarly scalable performance. The problem addressed by NCList is that once the first binary search is over, we still don't know which intervals that start ahead of our found interval overlap with it. But this is just a problem of initialization. If we allow for one object to point to another, all we have to do is to add a single "starting point contained by" pointer to the intervals such that

$$z \text{ -> } y \text{ -> } x \text{ -> } w$$

where "z -> y" indicates that the starting point of z is contained by interval y. Now one binary search for the closest interval C such that *C.start >= I.start* will give y. From here, we simply scan backward through our pointers and forward through our list until we are out of range. Both scans are *contiguous*.

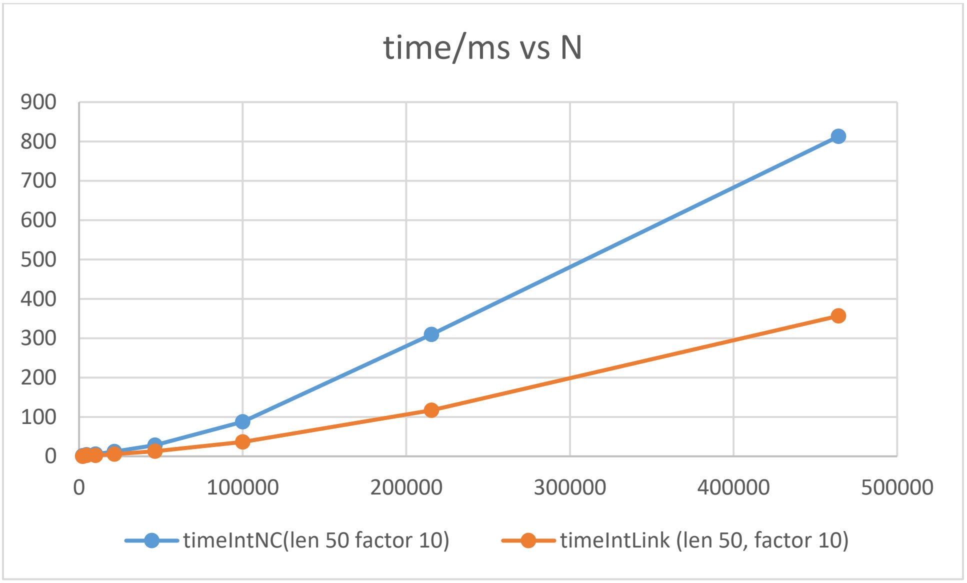
The simple linked list avoids the necessity for all the nesting structure that comes with NCNodes and NCList, and also provides an opportunity for "lazy initialization" of the list. Performance tests show it to be about twice as fast as NCList.

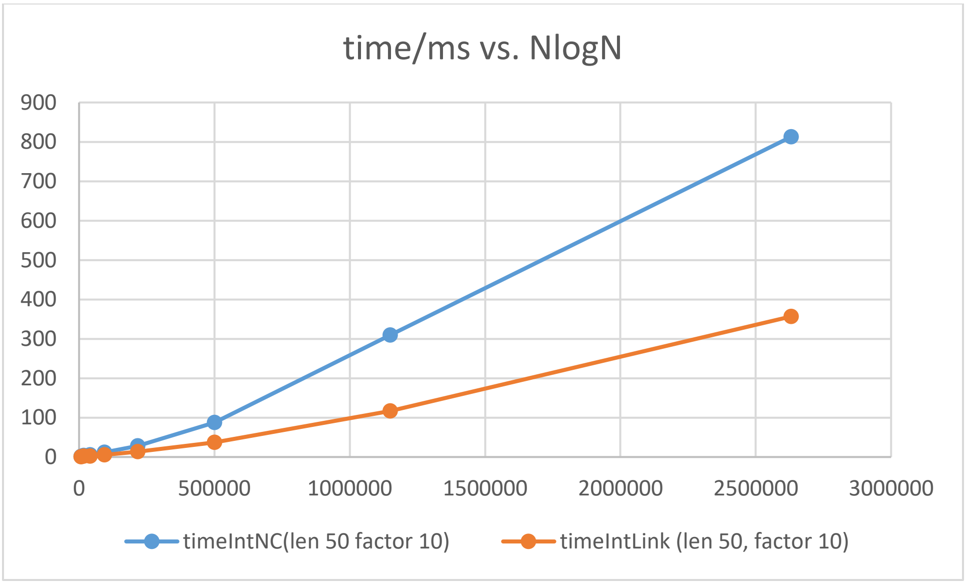
The advantage of NCList is that it pre-partitions the binary search of N objects into a set of n binary searches of $m_i \leq N$ objects, where SUM(i){m_i} = N. Depending upon the extent of nesting, this could be significant. With minimally nested sets, however, it is unlikely that this advantage would be noticeable.

The disadvantage of NCList is that it requires substantial initialization and query processing involving at least 3N objects (N NCNodes, each with a reference to its associated interval as well as a reference to its containing NCList). Thus, it is trivially shown that a single pointer and an integer index added to each of N intervals requires far less initialization and storage than a set of N Node/List objects. The storage is a single object reference and, in my implementation, an integer index. Querying involves a single initial binary search, followed by a similar relatively short *contiguous* scan of linked values in the backward direction only.

There are two additional advantages of the linked list approach. The first is that it allows for lazy initialization. We can do all the loading of the list, including minor addition/removal with the option to not sort the actual list or build the links until it is absolutely necessary (the first findOverlap() call, generally). In addition, the return list is in the same order (albeit reversed, for performance reasons) as the original sorted list. NCList returns a list that is not in any predictable order. In some situations, this could be an advantage.

NlogN	logN	N	Time/ms IntervalStore/NClist (len 50 factor 10)	Time/ms IntervalStore/Link (len 50, factor 10)
7179.811	3.333246	2154	1.5	0.3
17016.74	3.666612	4641	4.1	2.1
40000	4	10000	5.3	2.3
93357.18	4.333326	21544	11.9	5.7
216602.9	4.666658	46415	28.1	13.3
500000	5	100000	87.4	36.9
1149029	5.333332	215443	309.8	116.8
2630228	5.666666	464158	812.7	356.7





Settings in test/intervalstore/TimingTests2.java (IntervalStore+NCList) and

test/intervalstore/NoNCListTimingTests.java (IntervalStore+linked list)

```
  private static final int QUERY_STORE_INTERVAL_SIZE = 50;

  private static final int QUERY_STORE_SIZE_FACTOR = 10;
```

These constants determine the maximum size of the ranges in the test feature set (50) and the factor that is used to determine the overall effective width of the test sequence that the features are applied to (10):

```
{
  int maxPos = 4 * count * QUERY_STORE_SIZE_FACTOR; // 10
  List<Range> ranges = new ArrayList<>();
  for (int j = 0; j < count; j++)
  {
    int from = 1 + rand.nextInt(maxPos);
    int to = from + rand.nextInt(QUERY_STORE_INTERVAL_SIZE); // 50
    ranges.add(new Range(from, to));
  }
  return ranges;
}
```