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.

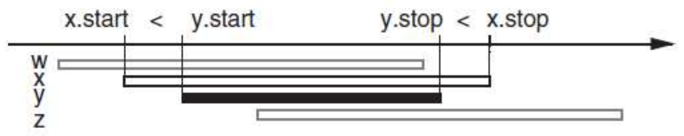


Fig. 2. Any contained interval breaks sortedness. If intervals (boxes) are sorted on their start coordinate, then any interval *y* (filled box) that breaks sortedness of the stop coordinate is properly contained in another interval *x*.

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. Create a list that is *monotonic* in both start and end points for all intervals by packaging any nonconforming intervals (*y* in this case) into one of those conforming intervals (i.e., *x*). NCList's findOverlap(from,to) method involves a series of 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. Intervals *w*, *x*, and *z* are monotonically increasing in both start and end positions, but *y* is not. We package *y* into *x*. A binary search for "the first end point after the start of interval *I*" finds *w*. A linear forward scan finds *x,* and *z*. A second binary search and linear scan through the subintervals of *x* finds *y*.

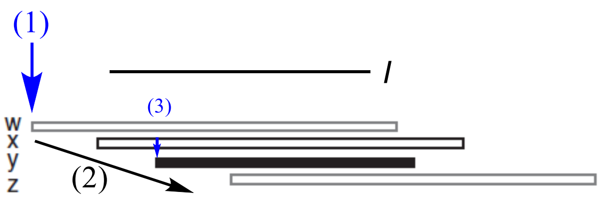


Figure 2: NCList looking for overlaps with *I* finds *w* first using a binary search for the closest end point after the start of *I* (1), then scans forward for intervals that have a starting point before the end of *I* (2), finding *x* and *z*. Interval *y* is found within *x* using a second binary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

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 specified interval overlap with it. Consider the situation in Figure 3. A set of eight intervals, a-h, are not monotonic. NCList creates a set of monotonic subsets: Set a contains b, g, and h; Set b contains c, d, and e; and Set e contains f. The key point here is that we are removing c, d, e, and f from the initial binary search.

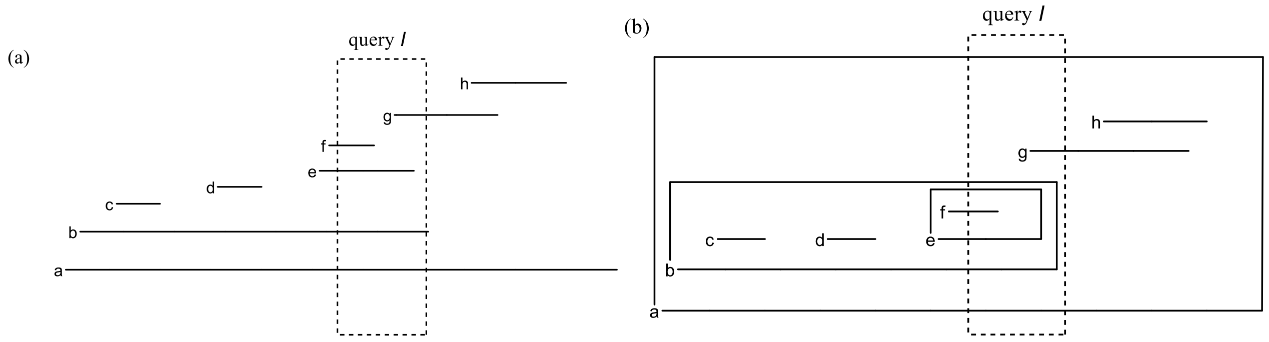


Figure 3. (a) Eight intervals a-h, in order of increasing start position is not monotonic in end position. The indicated query must return the set {a, b, e, f, g}. (b) NCList nests intervals that are not monotonically increasing in both start and end points, creating subset {b, g, h} within a, subset {c, d, e} within b, and subset {f} within e. Within each subset the intervals are monotonic in end point.

NCList processing finds the "first interval having an end point not before the query interval." In this case, it finds a. Then, within a's subsets it finds b, within b it finds e, and within e it finds f. Interval g is found by scanning the set {b, g, h} in sequential order until the start of an interval (h in this case) is past the end of the query interval. This works, but might it be overly complicated?

Now consider the simple alternative shown in Figure 4, where we allow for a linked list of pointers from an interval to the *nearest prior interval that contains its starting point*.

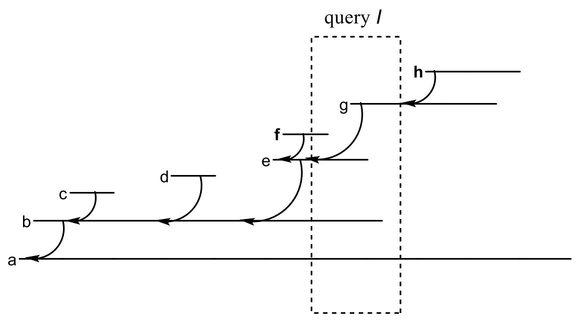


Figure 4. An alternative to NCList that utilizes a reference for each interval pointing to the nearest prior interval that contains its starting point. Note that intervals f and h flank this interval but are not included in it.

A *dual binary search* for the *two nearest starting points flanking this query* gives us f and h. The set of all intervals starting between these ({g} in this case) is added to the results. Interval f along with all intervals that are linked from it (f -> e -> b -> a), are added only if their end point is not before the start of the query interval. Note that intervals c and d are never checked, since they are not in the linked list starting from f.

The implementation at https://github.com/BobHanson/IntervalStoreJ uses a simple int[] array to manage the linked list, where the elements of the array are relative index offsets rather than actual object references. Thus, for the set {a b c d e f g h} the offsets would be {*, 1, 1, 2, 3, 1, 2, 1}, where * is a reserved number (Integer.MIN_VALUE, as implemented) meaning "not contained", which stops all link processing when it is found.

As described, the algorithm is not scalable for sets where there is substantial overlap. The problem is that this sort of set produces long strings of pointers. An improvement involves indicating an offset with a negative number when the pointer is to an interval that has a higher end point than any that comes before it (Figure 5). If that interval is checked and found to have an end point prior to the start of the query interval, it is guaranteed that no further checking along the chain will lead to a result. We can stop scanning.

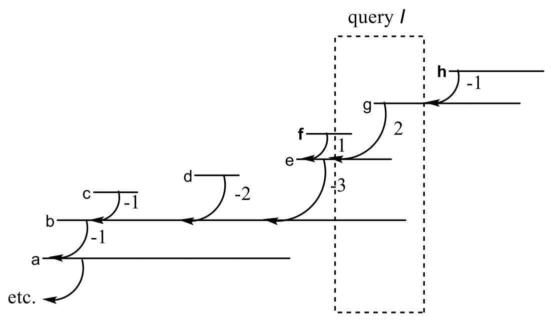


Figure 5: Offsets indicated with negative numbers to indicate a link to the interval with the highest end point of any previous intervals. The process (f -> e -> b -> a -> etc.) can be terminated after checking interval a, since its offset from b is -1, and so we know that no interval before Interval a extends beyond Interval a (into query *I*).

Advantages and Disadvantages

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 primary advantages of the linked list approach include:

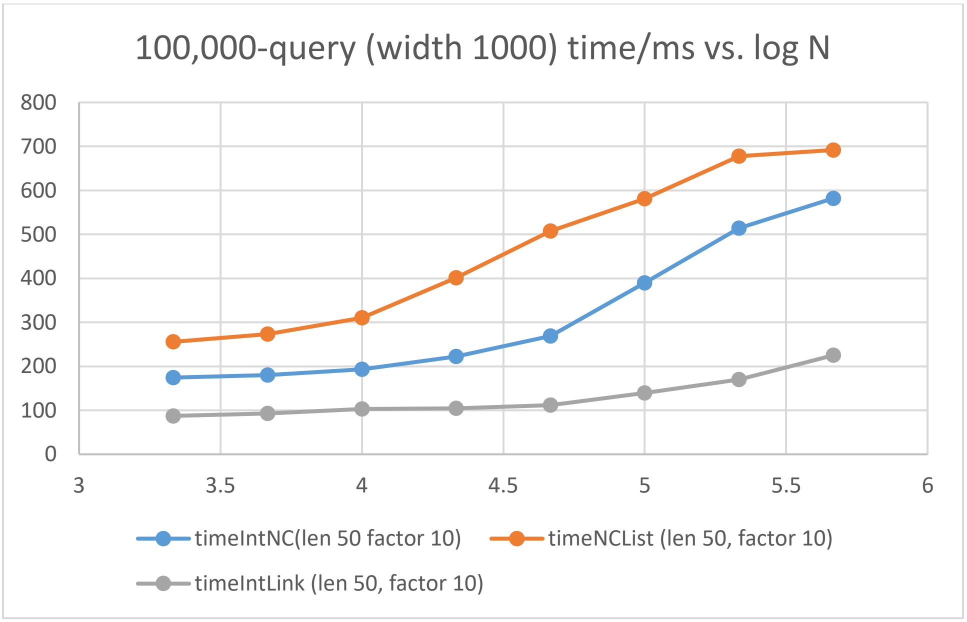
(1) It processes very efficiently, with a single binary search, followed by a single link-based check.

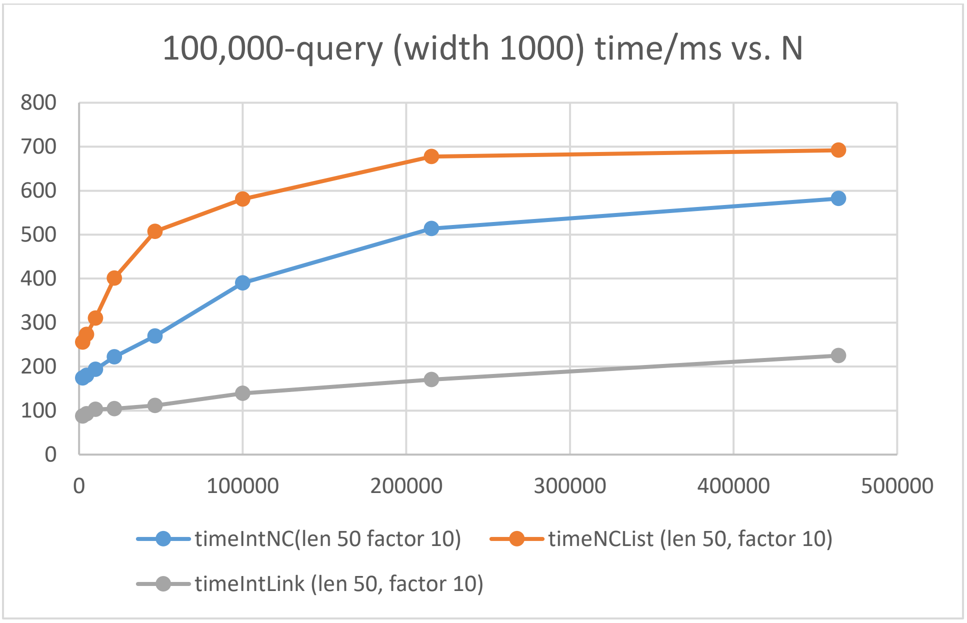
(2) It requires minimal initialization, with very little allocated memory (just simple arrays IntervalI[N] and int[N]). The simple linked list avoids the necessity for all the nesting structure that comes with NCNodes and NCList, as well as all initialization that goes with those objects.

(3) It allows for "lazy" initialization. That is, 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). Rebuilding after addition or removal is simply a recalculation of the offsets array.

(4) The return list is in the same order (albeit reversed, for performance reasons) as the original sorted list. In contrast, NCList returns a list that might or might not be ordered. In some situations, this could be an advantage.

logN	N	timeIntNC(len 50 factor 10)	timeNCList (len 50, factor 10)	timeIntLink (len 50, factor 10)
3.333246	2154	174.4	255.7	87.3
3.666612	4641	179.8	273	92.7
4	10000	193.5	310	102.9
4.333326	21544	222.4	401.2	104.5
4.666658	46415	269.1	507.4	111.8
5	100000	389.9	581	139
5.333332	215443	513.9	677.8	170.2
5.666666	464158	582	691.8	225





test/intervalstore/nonc/ISListTimingTests.java settings:

```
  /**
   * factor to multiply first parameter of generateIntervals(sequenceWidth,
   * count, length) by to set store sequence width; higher number reduces number
   * of overlaps
   */
  private static final int QUERY_STORE_SEQUENCE_SIZE_FACTOR = 10;// 10;

  /**
   * interval size for the store; absolute(negative) or maximum(positive);
   */
  private static final int QUERY_STORE_INTERVAL_SIZE = 50;// -1 for SNPs;

  /**
   * width of query intervals; negative for absolute, positive for max value
   *
   */
  // private static final int QUERY_WINDOW = -1;// overview single-pixel overlap
  private static final int QUERY_WINDOW = -1000;// -1000 standard view

  /**
   * number of queries to generate (independently of the size of the sequence
   *
   */
  private static final int QUERY_COUNT = 100000;


     int sequenceWidth = count * QUERY_STORE_SEQUENCE_SIZE_FACTOR;
                …
        List<Range> ranges = generateIntervals(sequenceWidth, count,
                QUERY_STORE_INTERVAL_SIZE);
        List<Range> queries = generateIntervals(sequenceWidth, QUERY_COUNT,
                QUERY_WINDOW);
…
  /**
   * Generates a list of <code>count</code> intervals of length [1,length] in
   * the range [1, sequenceWidth]
   *
   * @param sequenceWidth
   *          scale of the sequence, based on the number of intervals present,
   *          not the number of queries
   * @param count
   *          the number of intervals to generate
   * @param length
   *          maximum (positive) or absolute(negative) number of intervals to
   *          generate
   *
   * @return list of intervals
   */

  private synchronized List<Range> generateIntervals(int sequenceWidth,
          int count, int length)
  {
    int maxPos = sequenceWidth - Math.abs(length);
    List<Range> ranges = new ArrayList<>();
    for (int j = 0; j < count; j++)
    {
      int from = 1 + rand.nextInt(maxPos);
      int to = from + (length < 0 ? -length - 1 : rand.nextInt(length));
      ranges.add(new Range(from, to));
    }
    return ranges;
  }
```

Full run:

Test	size N	tests	time/ms	rate/(N/ms)	time stderr	rate stderr
# Query IntStoreNCList store interval size 50 store sequence factor 10 query width -1000 query count 100000						
IntStoreNCList query	2154	10	174.4	12.5	6.75	0.46
IntStoreNCList query	4641	10	179.8	25.9	3.75	0.51
IntStoreNCList query	10000	10	193.5	51.8	2.88	0.83
IntStoreNCList query	21544	10	222.4	98	8.13	3.35
IntStoreNCList query	46415	10	269.1	174.4	8.68	6.84
IntStoreNCList query	100000	10	389.9	268.6	25.99	20.62
IntStoreNCList query	215443	10	513.9	429.1	24.17	23.78
IntStoreNCList query	464158	10	582	832	34.03	66.58
# dimensions [7 464158]						
# Query IntStoreLink store interval size 50 store sequence factor 10 query width -1000 query count 100000						
IntStoreLink query	2154	10	85.8	25.1	0.99	0.28
IntStoreLink query	4641	10	98.8	47.7	4.35	1.96
IntStoreLink query	10000	10	99	101.1	1.24	1.2
IntStoreLink query	21544	10	111.8	193.9	3.2	4.66
IntStoreLink query	46415	10	117.4	395.3	0.68	2.29
IntStoreLink query	100000	10	170	589.2	2.36	8.34
IntStoreLink query	215443	10	267.2	823.8	12.57	42.87
IntStoreLink query	464158	10	335.8	1389.2	7.46	34.54
# dimensions [132 192882]						
# Query IntStoreLink2 store interval size 50 store sequence factor 10 query width -1000 query count 100000						
IntStoreLink2 query	2154	10	87.3	24.7	0.53	0.15
IntStoreLink2 query	4641	10	92.7	50.1	0.7	0.38
IntStoreLink2 query	10000	10	102.9	97.6	2.27	2.06
IntStoreLink2 query	21544	10	104.5	206.1	0.29	0.57
IntStoreLink2 query	46415	10	111.8	415.3	0.69	2.51
IntStoreLink2 query	100000	10	139	728.7	5.72	25.42
IntStoreLink2 query	215443	10	170.2	1266.9	1.53	10.78
IntStoreLink2 query	464158	10	225	2083.9	8.09	64.87
# dimensions [132 192882]						
# Query NCList store interval size 50 store sequence factor 10 query width -1000 query count 100000						
NCList query	2154	10	255.7	8.4	4.1	0.13
NCList query	4641	10	273	17	4.57	0.3
NCList query	10000	10	310	32.3	3.64	0.41
NCList query	21544	10	401.2	54.7	19.46	2.26
NCList query	46415	10	507.4	94	26.35	5.45
NCList query	100000	10	581	176.7	28.32	10.62
NCList query	215443	10	677.8	321.8	23.03	12.89
NCList query	464158	10	691.8	708.7	48.26	60.47
# dimensions [6 245774]						