To nest or not to nest, that is the question. An argument that NCList is unnecessary. BH 2019.08.09 rev.2019.09.06

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.

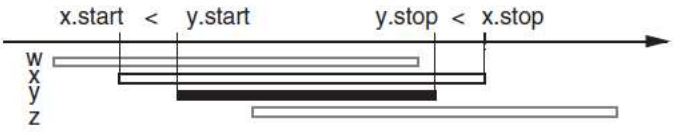

Fig. 2. Any contained interval breaks sortedness. If intervals (boxes) are sorted on their start coordinate, then any interval *y* (filled box) that breaks sortedness of the stop coordinate is properly contained in another interval *x*.

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 and theortically ideal. Create a list that is *monotonic* in both start and end 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*).

The problem addressed by NCList is that, if we just have a list ordered by start, 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 ordered by start but not monotonic in end. NCList creates a set of subsets that are internally monotonic in both start and end: 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.

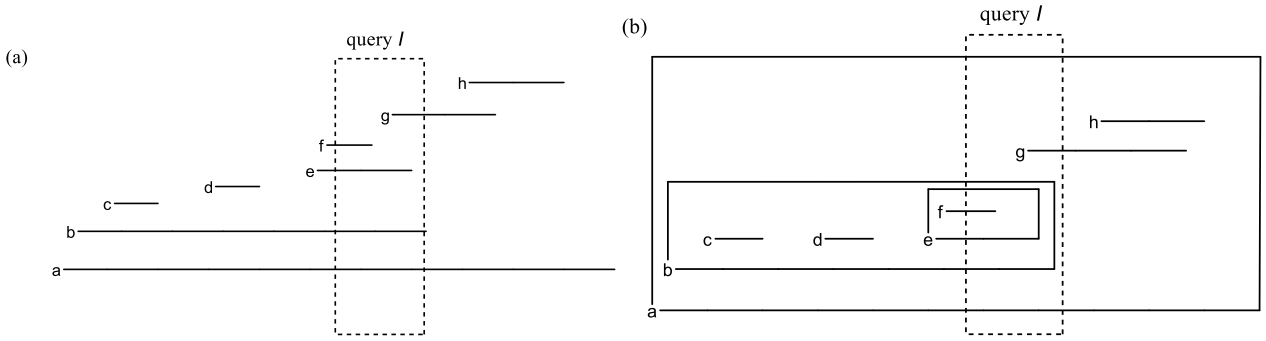

Figure 3. (a) Eight intervals a-h, in order of increasing start position is not monotonic in end position. The indicated query must return the set {a, b, e, f, g}. (b) NCList nests intervals that are not monotonically increasing in both start and end points, creating subset {b, g, h} within a, subset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

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 the bottom set of possibly overlapping but not nested intervals are pulled out and handled as a separate array. This does seem to add some efficiency. While this adds complications when adding or removing intervals from the set, and no longer is capable of efficiently returning an ordered query, 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?

Our initial finding was that creating this possibly huge set of objects might be avoided. Several experiments were done, and we have found two adaptations that work. One solution involves a linked list with similarly scalable performance, at least on a practical, if not theoretical, basis. 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*.

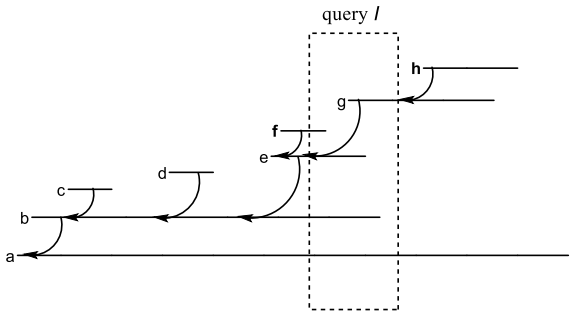

Figure 4. An alternative to NCList that utilizes a reference for each interval pointing to the nearest prior interval that contains its starting point. Note that intervals f and h flank this interval but are not included in it.

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 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.

This approach suffers from the possibility of a pathological nesting, where there are long chains of connections above One or more long intervals. Still, in practice (100+ levels), this method as implements beats anything by about a query rate factor of two.

The implementation "IntervalStore0" 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.

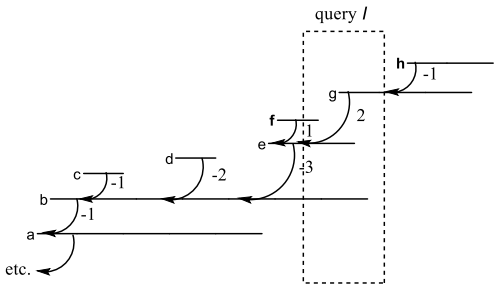

Figure 5: Offsets indicated with negative numbers to indicate a link to the interval with the highest end point of any previous intervals. The process (f -> e -> b -> a -> etc.) can be terminated after checking interval a, since its offset from b is -1, meaning that we know that no interval before Interval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

A second implementation of NCList

After much testing and revising, I ended up implementing NCList and NCList-with-unnesting as single ordered array buffer of intervals (nests[]), along with two relatively simple int[] arrays that give offset and length information about where each nest resides within the buffer. This array proves to be about 10-20% faster to search than a set of container objects with multiple ArrayLists. Its query is somewhat slower than the link-list method described above, but it is provably as scalable as NCList for query, with no *practical* limitation other than the allowed depth of nested re-entrant method calling that a nested search requires (same as NCList).

The basic idea is to create an IntervalI[] array buffer, *nests*, along with two int[] arrays, *nestOffsets*, and *nestLengths*. We reserve the last one or two elements of *nestOffsets* and *nestLengths* for pointer to the root nest and, if implemented, an unnested set, as in IntervalStoreJ.

For example, for the nesting of twelve intervals shown in Figure 6, we would have the array *nests* below, which partitions into five distinct binary-searchable sets. The first three elements, in blue, are the unnested Set 1, binary-searchable as the first three elements of *nests*. There is just one element in the root nest (Set 2, 10-80 in this case, shown in orange). Its subinterval Set 3, {4, 5, 6}, is indicated as starting at offset 4 and having three elements. The sixth interval, 50-80, is also a nest, pointing to Set 4, {7, 8, 9, 10}. Finally, interval 9 contains the single subinterval 11 as Set 5.

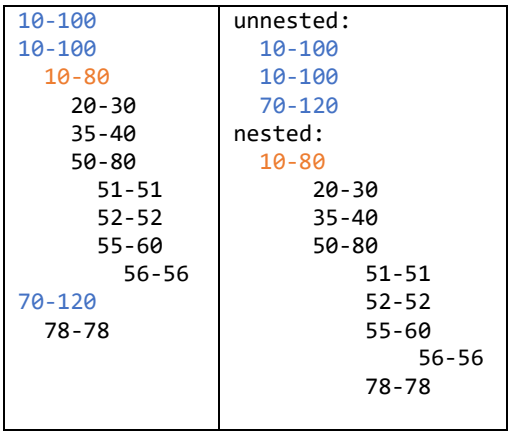

Figure 6. On the left, an NCList-ordered nesting, with three root intervals. On the right, an InteravalStoreI nesting, which pulls out four "unnested" intervals and leaves one element (in this case) as the sole element in the "root nest".

set		**1**		**2**		**3**			**4**			**5**	
index	{ 0	1	2	}{ 3	}{ 4	5	**6**	}{ 7	8	**9**	10 }{	11 }	
nests	[10-100,	10-100,	70-120,	10-80,	20-30,	35-40,	**50-80,**	51-51,	52-52,	**55-60,**	78-78,	56-56]	
nestOffsets	[0,	0,	0,	4,	0,	0,	**7,**	0,	0,	**11,**	0,	0,	3, 0]
nestLengths	[0,	0,	0,	3,	0,	0,	**4,**	0,	0,	**1,**	0,	0,	1, 3]

Note that this organization is *precisely* the organization of NCList, with the added option of pulling out an unnested set if desired. We retain all the advantages of a binary-searchable tree without any of the overhead of constructing and garbage collecting of ArrayList objects and NCList containers for those lists.

The code for quickly organizing these arrays is in IntervalStore#createArrays. It involves two phases. The first phase identifies the proper container for each interval, and the second uses that information to create the three critical arrays.

[Aside: An interesting aspect of this is construct is that it would be trivially serializable as a simple set of 2N + 4 integers. But I have not pursued that angle.]

Accelerating the loading of intervals

A major difficulty is trying to scale loading time, particularly if checking for duplicates. The problem is that every time a new interval is added, one has to check to see if it is a duplicate of any other in the set. As shown in Figure 7, this is a problem for both of the NCList Java implementations.

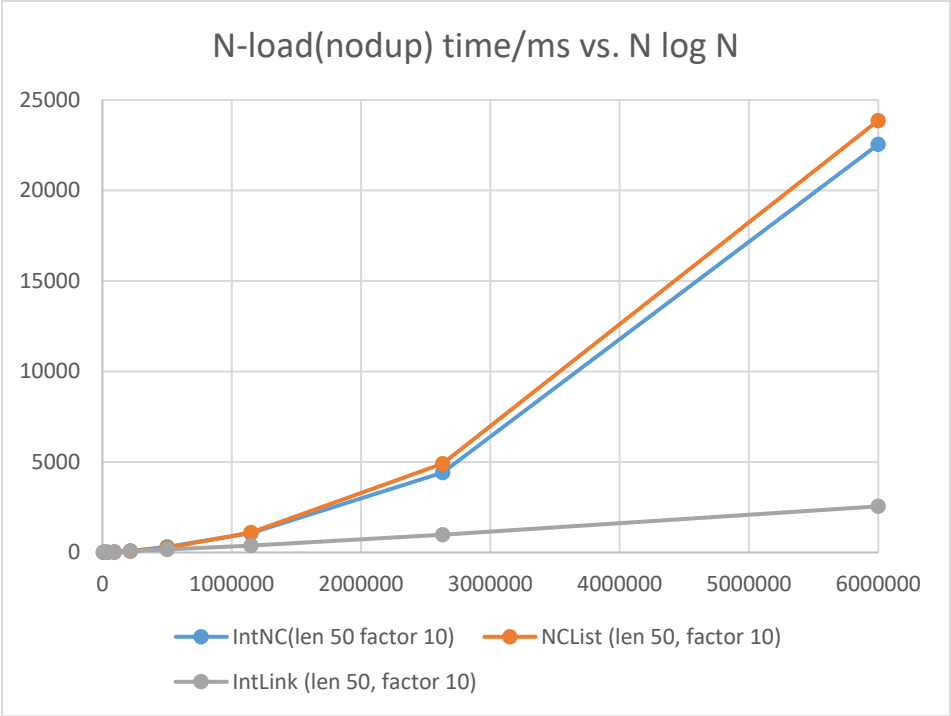

Figure 7. Comparison of loading time in ms vs. N log N for the loading of N intervals and not allowing duplicates. NCList does not scale, at least in these two implementations (top two data sets). Lower traces is for IntervalStore using a linked list. Load time includes finalization.

Two major improvements give a huge advantage to the linked list approach. First, in this version of intervalstore.nonc.IntervalStore, all private storage of array data is handled by a simple IntervalI[ ] array. ArrayList is not used at all. This allows substantially more control over array capacity and accelerates all array processing.

One might think this would lead to a waste of space, but actually not. We use the extra capacity to temporarily handle out-of-sequence additions, saving hugely in incremental sorting time. Thus, we grow the array *from both ends*, as shown in Figure 8, using the front end of the array to hold the growing trunk of ordered intervals and the tail end of the array to hold a binary linked list of ordered branches off this main trunk

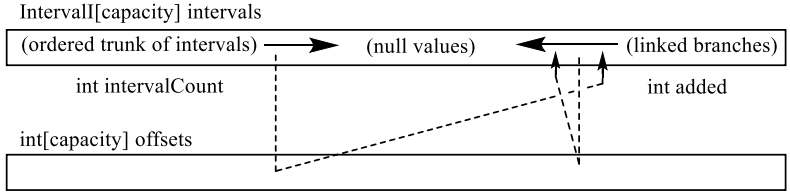

Figure 8. Double-ended array holding a growing binary tree of intervals with main trunk growing from the left that is searchable using a binary search. Dotted lines are links indicating branching points.

Periodically, when it is time to enlarge the array, we simply scan the main trunk from right to left, shifting blocks of intervals right, inserting the linked branches in a single pass, and discarding the branches. Array shifting uses highly efficient native System.arrayCopy calls. The array is fully trimmed after the incremental loading is completed.

Deletion of intervals

Figure 9 shows preliminary results for deletion timing. The test deletes 1000 intervals from a collection of N intervals. No effort has been made to optimize this for linked lists. Such is the cost of doing a rebuilding of the array after each deletion.

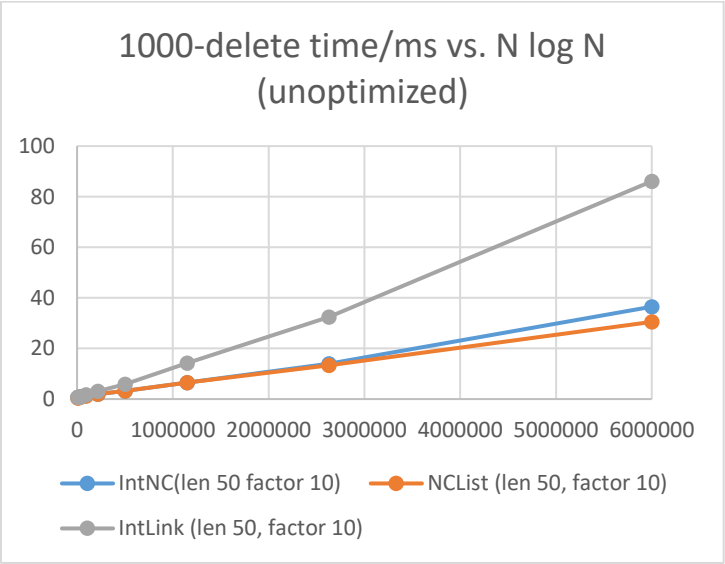

Figure 9. Timing for deletion of N intervals. No optimization has been done yet.

We can do better by simply logging each deletion to a BitSet at the time of deletion and then, lazy, only when needed, do a single run of array shifts to fill in the deleted intervals. With this optimization we see significant improvement (Figure 10).

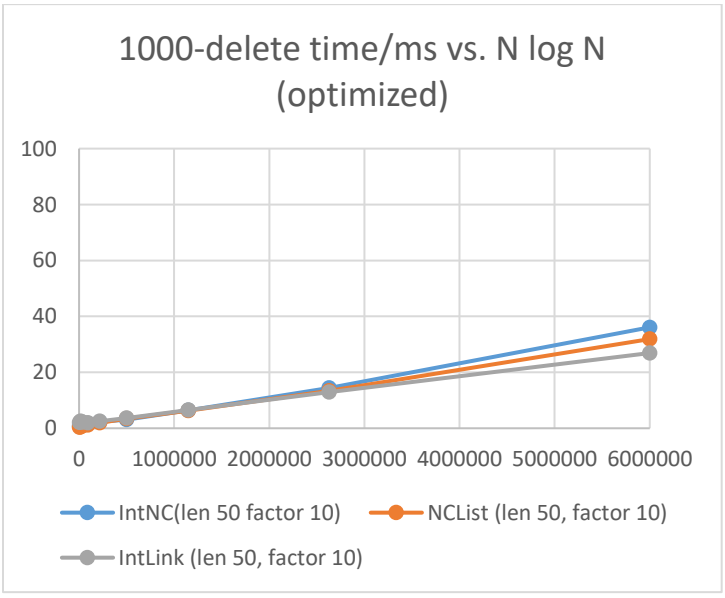

Figure 10. Timing for deletion of N intervals after optimization of the linked-list algorithm using a BitSet and lazy initialization.

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 queries very efficiently, with a single binary search, followed by a single (generally quick) 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, IntervalStoreJ’s implementation of NCList uses of separate nested and unnested lists, which are processed sequentially. It thus returns a list that might or might not be ordered. In some situations, this could be an advantage.

The advantage of using the array buffer approach to NCList is that it minimizes the number of objects created and later disposed of by the system (which can be a significant problem in JavaScript).

Performance test results (IntervalStoreListTest.java) are shown in Figure 11. The two approaches discussed here are the lower two lines, performing two to three times faster than NCList expresses as a nested set of objects.

Figure 11. Performance results for querying 100,000 times, each query an interval of 1000, on a data structure containing N intervals, each of length 1 to 50 (randomly chosen and positioned) in a “sequence” of length 10 * N. The bottom two lines are for linked (lowest), and NCList-array buffer. “0” indicates unoptimized.

Timing results: query (see testQuery.xlsx for linked-list comparison to NCList)

Timing results for querying (Table 1 and Figure 12) suggest that using a linked list is from two to three times faster than NCList alone or the IntervalStoreJ/NCList implementation. In fact, compared to NCList, the linked list alternative will return 100,000 queries from a set containing 464K intervals in the time it takes NCList to return 100,000 queries from a list that contains only 2K intervals. All three method times are linear in log N for large N and governed by other factors at low N.

logN	N	IntNC	NCList	IntLink
3.33	2154	160.4	222.9	88.2
3.67	4641	165.1	232.8	90.6
4.00	10000	173.5	241.8	95.4
4.33	21544	183.9	262.6	99.4
4.67	46415	201.2	306.9	105.7
5.00	100000	244.8	365.3	121.5
5.33	215443	289.7	406.5	144.3
5.67	464158	327.3	435.1	169.6
6.00	1000000	363.7	467.8	189.1

Table 1. Timing results (in ms) for returning 100,000 queries from a “sequence” of length N * 10 that contains N intervals of pseudorandom length 1 to 50 for IntervalStore/NCList (“IntNC”), Nclist alone, and IntervalStore/linked list (“IntLink”). The start positions of the intervals within the sequence are in the range 1 to N * 10 - 50. Each query has a length of 1000 and a pseudorandom position within the sequence.

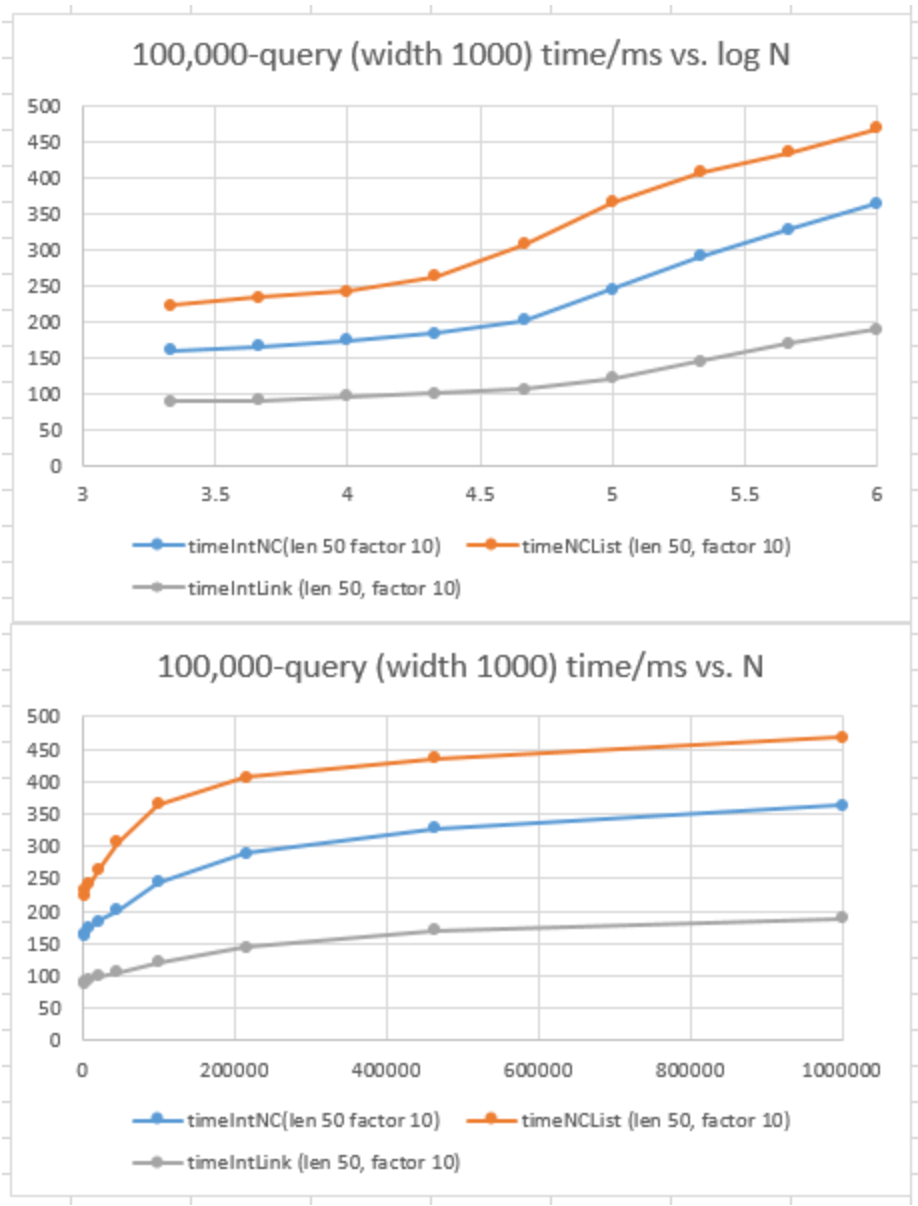

Figure 12. Query time vs. logN and N for the linked-list variation of IntervalStore (lowest set), along with the IntervalStore implementation of NCList (middle data set), and NCList without IntervalStore (highest set).

```
Full run:
```

[RemoteTestNG] detected TestNG version 6.14.2

Java version: 1.8.0_191

amd64 Windows 10 10.0 cores:4

14 Aug 2019 21:09:15 GMT

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	160.4	13.5	2.93	0.24
IntStoreNCList query	4641	10	165.1	28.1	0.54	0.09
IntStoreNCList query	10000	10	173.5	57.6	0.7	0.23
IntStoreNCList query	21544	10	183.9	117.2	0.48	0.3
IntStoreNCList query	46415	10	201.2	230.7	0.79	0.89
IntStoreNCList query	100000	10	244.8	408.4	0.54	0.9
IntStoreNCList query	215443	10	289.7	743.7	0.47	1.2
IntStoreNCList query	464158	10	327.3	1418.4	1.16	4.95
IntStoreNCList query	1000000	10	363.7	2749.4	0.57	4.33
# dimensions [7 1000000]						
# Query IntStoreNCList0 store interval size 50 store sequence factor 10 query width -1000 query count 100000						
IntStoreNCList0 query	2154	10	183.3	11.8	3.89	0.22
IntStoreNCList0 query	4641	10	190.5	24.4	1.6	0.19
IntStoreNCList0 query	10000	10	197.1	50.7	0.56	0.14
IntStoreNCList0 query	21544	10	207	104.1	0.46	0.23
IntStoreNCList0 query	46415	10	225.2	206.1	0.43	0.39
IntStoreNCList0 query	100000	10	286.5	351.2	8.01	8.63
IntStoreNCList0 query	215443	10	315.6	683	2.03	4.21
IntStoreNCList0 query	464158	10	354.2	1310.4	0.59	2.19
IntStoreNCList0 query	1000000	10	389.6	2567	0.99	6.6
# dimensions [7 0]						
# Query IntStoreLink store interval size 50 store sequence factor 10 query width -1000 query count 100000						
IntStoreLink query	2154	10	88.2	24.5	2.24	0.52
IntStoreLink query	4641	10	90.6	51.2	0.62	0.34
IntStoreLink query	10000	10	95.4	104.8	0.41	0.45
IntStoreLink query	21544	10	99.4	216.8	0.24	0.53
IntStoreLink query	46415	10	105.7	439.3	0.34	1.43
IntStoreLink query	100000	10	121.5	823.4	0.3	2
IntStoreLink query	215443	10	144.3	1493.6	0.58	5.89
IntStoreLink query	464158	10	169.6	2748.3	3.96	54.12
IntStoreLink query	1000000	10	189.1	5287.4	0.29	8.23
# dimensions [126 416364]						
# Query IntStoreLink0 store interval size 50 store sequence factor 10 query width -1000 query count 100000						
IntStoreLink0 query	2154	10	83.1	25.9	0.18	0.06
IntStoreLink0 query	4641	10	87	53.4	0.81	0.47
IntStoreLink0 query	10000	10	89.7	111.5	0.28	0.34
IntStoreLink0 query	21544	10	93.6	230.2	0.31	0.76
IntStoreLink0 query	46415	10	99.7	465.7	0.31	1.46
IntStoreLink0 query	100000	10	120	833	0.29	2.01

IntStoreLink0 query	215443	10	146	1476.1	0.14	1.45
IntStoreLink0 query	464158	10	170.2	2726.5	0.42	6.71
IntStoreLink0 query	1000000	10	193.3	5173.2	0.19	5.14

dimensions [126 416364]

Query NCList store interval size 50 store sequence factor 10 query width -1000 query count 100000

NCList query	2154	10	222.9	9.7	0.71	0.03
NCList query	4641	10	232.8	19.9	0.41	0.03
NCList query	10000	10	241.8	41.3	0.51	0.09
NCList query	21544	10	262.6	82.8	9.44	2.31
NCList query	46415	10	306.9	151.2	0.4	0.2
NCList query	100000	10	365.3	273.7	0.49	0.37
NCList query	215443	10	406.5	530	0.19	0.25
NCList query	464158	10	435.1	1066.9	0.21	0.52
NCList query	1000000	10	467.8	2142.7	8.04	32.38

dimensions [7 528974]

Query NCList0 store interval size 50 store sequence factor 10 query width -1000 query count 100000

NCList0 query	2154	10	230.3	9.4	2.1	0.08
NCList0 query	4641	10	239.7	19.4	0.86	0.07
NCList0 query	10000	10	246.3	40.6	0.27	0.04
NCList0 query	21544	10	260.3	82.8	0.85	0.27
NCList0 query	46415	10	310.2	149.6	0.37	0.18
NCList0 query	100000	10	365	274	0.64	0.48
NCList0 query	215443	10	404.9	532.1	0.33	0.44
NCList0 query	464158	10	434.4	1068.6	0.66	1.62
NCList0 query	1000000	10	462.4	2163.4	3.36	14.88

dimensions [7 ?]

resultcounts [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 100, 113, 96, 112, 100, 93, 98, 112, 88]

PASSED: testLoadTimeBulk

PASSED: testLoadTimeIncrementalAllowDulicates

PASSED: testLoadTimeIncrementalNoDuplicates

PASSED: testQueryTime

PASSED: testRemoveTime

==========

Default test

Tests run: 5, Failures: 0, Skips: 0

==========

==========

Default suite

Total tests run: 5, Failures: 0, Skips: 0

==========

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;
  }
```

Timing results – Loading (see testLoad.xlsx)

15 Aug 2019 21:11:19 GMT

Test	size N	tests	time/ms	rate/(N/ms)	time stderr	rate stderr
# incr allowDuplicates:true IntStoreNCList						
IntStoreNCList incr load dup	2154	10	1.5	1485.1	0.09	110.44
IntStoreNCList incr load dup	4641	10	3.4	1416.6	0.24	106.29
IntStoreNCList incr load dup	10000	10	7.4	1401.6	0.42	83.57
IntStoreNCList incr load dup	21544	10	16.5	1368.6	1.28	92.09
IntStoreNCList incr load dup	46415	10	43.3	1077.6	1.23	25.18
IntStoreNCList incr load dup	100000	10	172.2	582.9	3.74	11.57
IntStoreNCList incr load dup	215443	10	765.2	282.7	17.15	5.55
IntStoreNCList incr load dup	464158	10	3496.1	132.8	15.56	0.59
IntStoreNCList incr load dup	**1000000**	**10**	**18245.9**	**54.8**	**40.36**	**0.12**
# incr allowDuplicates:true IntStoreLink						
IntStoreLink incr load dup	2154	10	3.9	603.9	0.37	58.82
IntStoreLink incr load dup	4641	10	5.3	917.6	0.45	49.58
IntStoreLink incr load dup	10000	10	11.7	869.1	0.52	32.64
IntStoreLink incr load dup	21544	10	26.5	831.4	1.64	34.71
IntStoreLink incr load dup	46415	10	65.7	714.1	2.22	24.46
IntStoreLink incr load dup	100000	10	148.7	696.3	11.62	33.90
IntStoreLink incr load dup	215443	10	356.7	607.8	9.76	15.54
IntStoreLink incr load dup	464158	10	870.2	534.3	12.36	7.21
IntStoreLink incr load dup	**1000000**	**10**	**2200.7**	**454.7**	**19.94**	**4.09**
# incr allowDuplicates:true NCList						
NCList incr load dup	2154	10	1.5	1563.8	0.13	156.47
NCList incr load dup	4641	10	3.7	1353.7	0.32	114.84
NCList incr load dup	10000	10	7.4	1374.5	0.29	42.92
NCList incr load dup	21544	10	17.0	1320.1	1.19	79.90
NCList incr load dup	46415	10	50.1	927.0	0.48	8.80
NCList incr load dup	100000	10	206.5	487.1	5.72	11.64
NCList incr load dup	215443	10	906.3	237.9	9.40	2.37
NCList incr load dup	464158	10	4359.8	106.5	18.97	0.46
NCList incr load dup	**1000000**	**10**	**23051.1**	**43.6**	**495.51**	**0.91**
# incr allowDuplicates:false IntStoreNCList						
IntStoreNCList incr load nodup	2154	10	2.7	901.9	0.32	112.45
IntStoreNCList incr load nodup	4641	10	5.1	939.1	0.30	51.51
IntStoreNCList incr load nodup	10000	10	10.1	1001.2	0.30	26.86
IntStoreNCList incr load nodup	21544	10	27.6	780.3	0.30	8.37
IntStoreNCList incr load nodup	46415	10	84.7	550.4	1.96	11.30
IntStoreNCList incr load nodup	100000	10	308.5	329.2	13.18	13.27
IntStoreNCList incr load nodup	215443	10	1067.5	202.3	17.50	3.24
IntStoreNCList incr load nodup	464158	10	4411.9	105.3	50.45	1.16
IntStoreNCList incr load nodup	**1000000**	**10**	**22540.3**	**44.4**	**254.92**	**0.49**
# incr allowDuplicates:false IntStoreLink						
IntStoreLink incr load nodup	2154	10	2.8	902.8	0.62	77.97
IntStoreLink incr load nodup	4641	10	7.2	676.0	0.51	49.04
IntStoreLink incr load nodup	10000	10	11.6	868.3	0.24	17.19
IntStoreLink incr load nodup	21544	10	26.7	807.0	0.23	7.14
IntStoreLink incr load nodup	46415	10	69.3	676.8	2.46	21.94
IntStoreLink incr load nodup	100000	10	160.2	629.2	5.08	16.76
IntStoreLink incr load nodup	215443	10	371.5	585.2	12.54	17.21
IntStoreLink incr load nodup	464158	10	976.6	476.7	17.87	8.66
IntStoreLink incr load nodup	**1000000**	**10**	**2546.9**	**393.6**	**43.56**	**6.37**
# incr allowDuplicates:false NCList						
NCList incr load nodup	2154	10	1.8	1298.3	0.17	118.44
NCList incr load nodup	4641	10	3.5	1375.4	0.24	86.87
NCList incr load nodup	10000	10	7.8	1298.4	0.28	40.66
NCList incr load nodup	21544	10	21.0	1028.4	0.31	15.11
NCList incr load nodup	46415	10	69.5	669.3	0.93	8.52
NCList incr load nodup	100000	10	262.7	383.3	7.89	9.73
NCList incr load nodup	215443	10	1107.9	195.4	26.48	4.39
NCList incr load nodup	464158	10	4899.2	94.8	43.72	0.85
NCList incr load nodup	**1000000**	**10**	**23859.4**	**42.0**	**257.01**	**0.44**