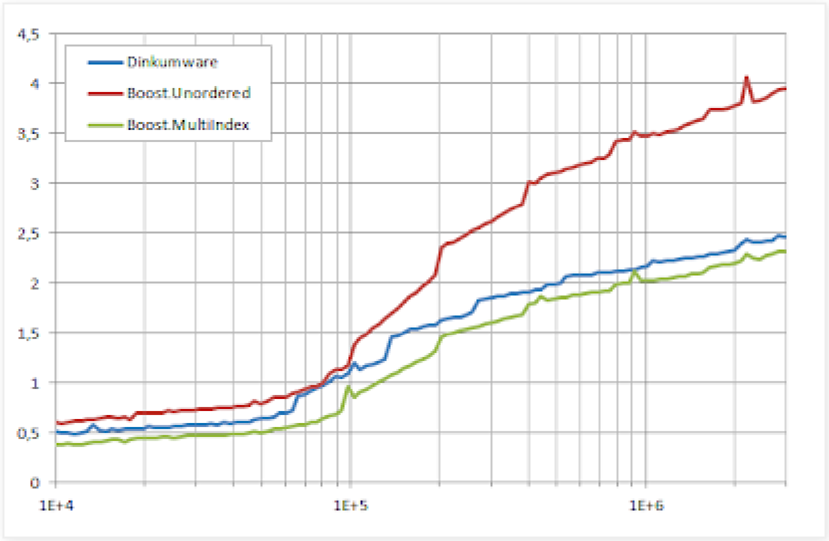
# Measuring erasure times for C++ unordered associative containers

After measuring insertion times for Dinkumware, Boost.Unordered and Boost.MultiIndex implementations of C++ unordered associative containers without and with duplicates, we now turn to profiling erasure. As a model for this operation we choose the following scenario:

```
void erase(container& c)
{
  for(auto it:rnd_it_range(c))c.erase(it);
}
```

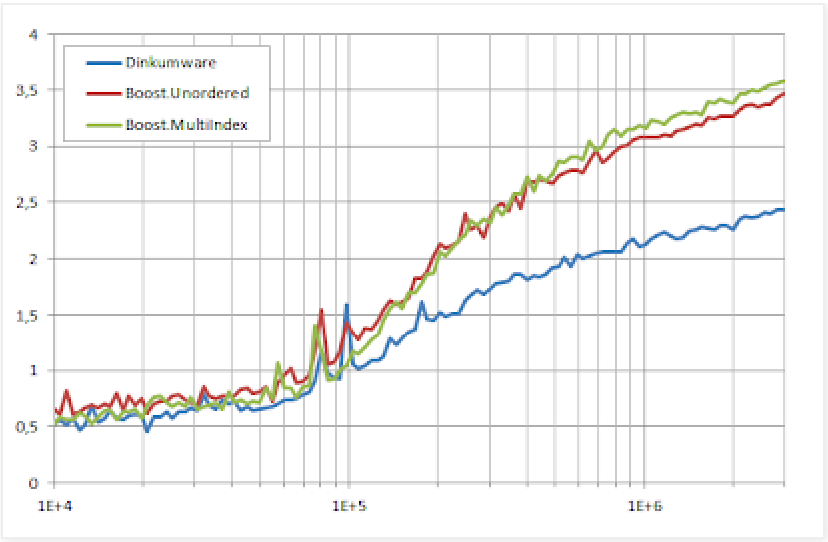
where `rnd_it_range(c)` is a random shuffling of [`c.begin()`, `++c.begin()`, ... , `c.end()`). I've written profiling programs for the non-duplicate and duplicate versions of the containers, respectively, which were built and run on the same environment as in previous entries. The graphics show resulting erasure times in microseconds/element for $n$ = 10,000 to 3 million elements.

These are the results for containers without duplicate elements:



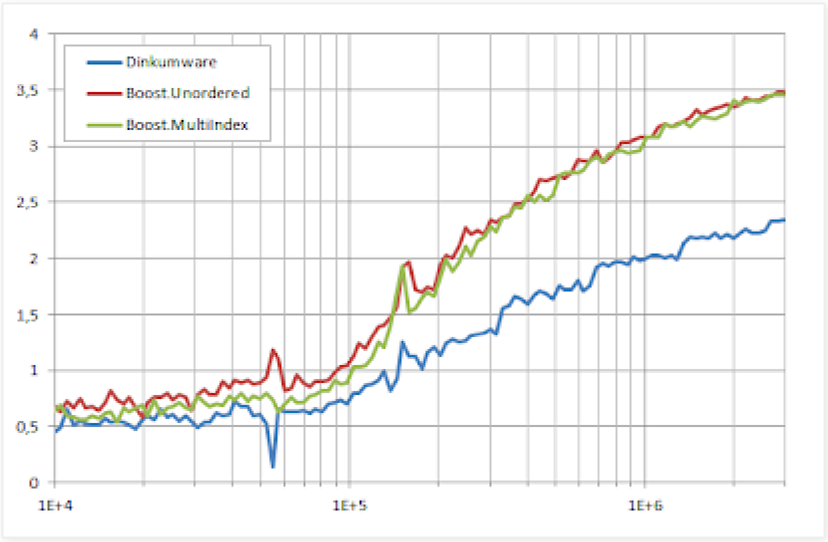
Dinkumware and Boost.MultiIndex have unconditional O(1) erasure since buckets need not be traversed to delete an element, and moreover the locality of their corresponding erasure procedures is very good: all of this shows as superior performance with respect to Boost.Unordered, which needs to do bucket traversal in order to locate the node preceding the one being deleted. The CPU memory cache, which favors algorithm locality, makes the differences in performance increase when $n$ is large.

Containers with duplicate elements are populated with a random sequence of elements with groups of equivalent values having an average size $G$ = 5. We first set the maximum load factor $F_{\text{max}}$ = 1:



Dinkumware performance does not change, as this library's containers treat equivalent elements in exactly the same manner than in the non-duplicate case, and bucket occupancy, greater in this case, does not affect the erasure procedure. Boost.Unordered performance increases somewhat: when deleting a element in the middle of a group of equivalent values, the library does not need to traverse the bucket because the preceding and following elements are linked in a circular list; the net result is that locality improves for these elements and stays the same for elements at the beginning or the end of a group, thus overall performance is better. Boost.MultiIndex behaves the worst here: even though erasure is still unconditionally constant-time, the complexity of the underlying data structure implies that as many as seven memory locations can be visited to erase an element, 75% more than Dinkumware and at about the same level as Boost.Unordered.

When we set $F_{\max} = 5$, the situation does not change dramatically:



Boost.Unordered is the only library that could theoretically be impacted by the higher bucket occupancy, but the effect is nonetheless not noticeable.

It should be noted that Dinkumware (in disagreement with the C++ standard) invokes the hash function of the element being erased, which would negatively affect its performance for elements harder to hash than simple `unsigned int`s. That said, the special provisions Boost.Unordered and Boost.MultiIndex implement to accelerate the processing of groups of equivalent elements have not yet achieved spectacular performance improvements, or even result in decidedly slower times. It is at lookup that these libraries excel, as we will see in a future entry.