`pvector` — pure Scheme persistent vector implementation

Table of Contents

1 Motivation

1.1 Why?

The absence of an efficient Clojure-like purelly functional vector in the Guile Scheme standard library has always been a pain point for me. However, the idea of a bit-partitioned vector trie is very beautiful and relatively simple. So once I decided to implement it myself.

1.2 Sources of inspiration

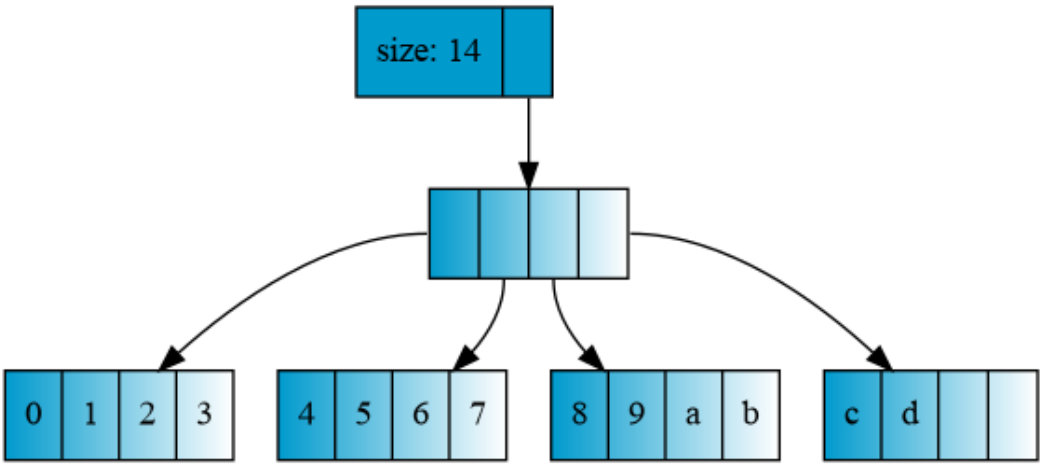
- Persistent vector implementation in Cloure by Rich Hickey: `https://github.com/clojure/clojure/blob/master/test/clojure/test_clojure/vectors.clj`
- Persistent vector implementation in Racket by Alexis King: `https://github.com/lexi-lambda/racket-pvector/tree/master`
- Blog post series «Understanding Clojure's Persistent Vectors» by Jean Niklas L'Orange:
  - `https://hypirion.com/musings/understanding-persistent-vector-pt-1`
  - `https://hypirion.com/musings/understanding-persistent-vector-pt-2`
  - `https://hypirion.com/musings/understanding-persistent-vector-pt-3`
  - `https://hypirion.com/musings/understanding-clojure-transients`
  - `https://hypirion.com/musings/persistent-vector-performance`
  - `https://hypirion.com/musings/persistent-vector-performance-summarised`
- A talk «Postmodern immutable data structures» by Juan Pedro Bolivar Puente at CppCon 2017: `https://www.youtube.com/watch?v=sPhpelUfu8Q`

2 Basic principles

2.1 Representing vector as a wide shallow tree

The simpliest way to store a vector is an array (reallocated quantum satis). But this representation is incompatible with the immutability: any write operation (change the element, push the new element, pop the element) will get $O(n)$ steps (where n is the number of vector elements). So, this naïve implementation will be unacceptably slow.

The most common approach to invent persistent data structures (popularized by Kris Okasaki in his PhD thesis[1]) is to store values in a tree and to copy (on write) only the target tree leaf and the path to the target leaf. We can apply this approach to the persistent vector too.



Vector of 14 elements as a shallow tree.
My reporoduction of a picture by Jean Niklas L'Orange
from `https://hypirion.com/musings/understanding-persistent-vector-pt-1`

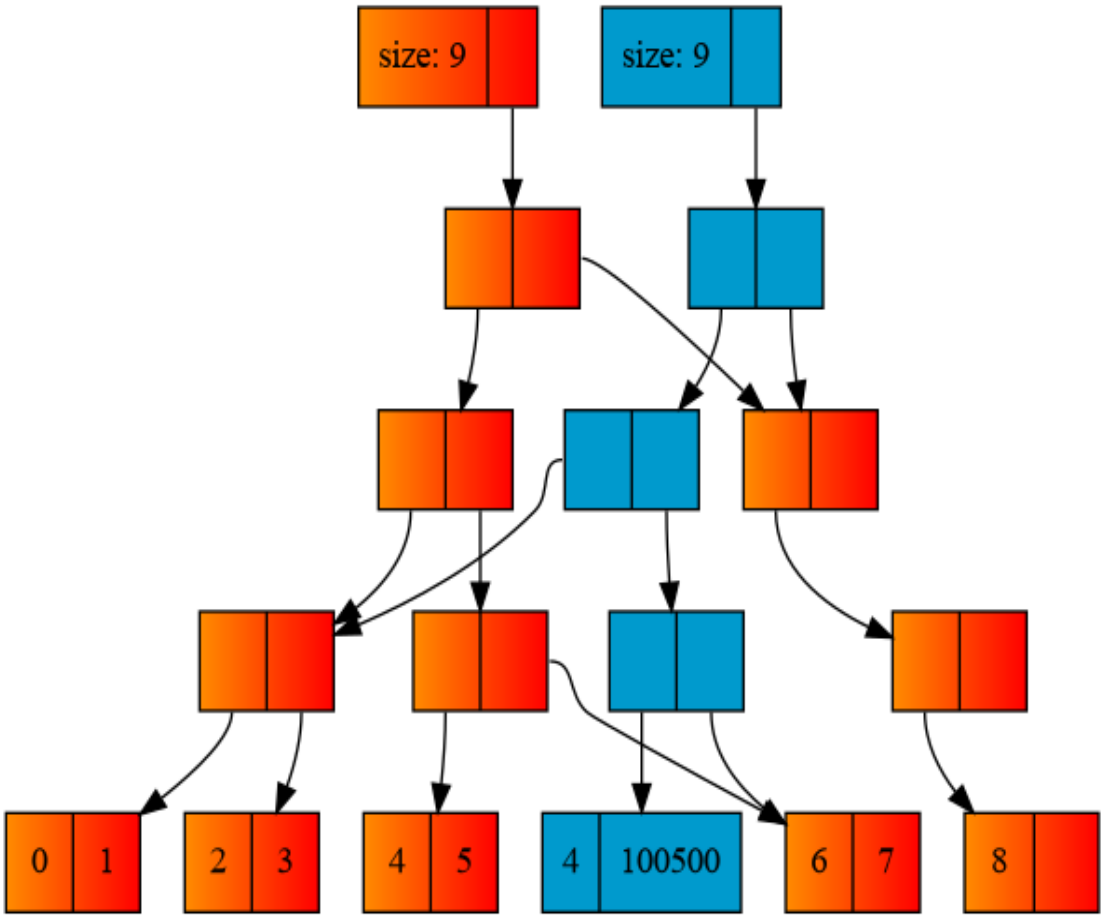
We can store the vector as a bunch of small arrays, stored as the leafs of the tree. So, any operation with the single element (access by index, replacing, pushing, popping) will have $O(\log_m n)$ steps, where n is the vector size, m — the branching factor of the tree. Note that those tree can be very broad and thus very shallow: while the binary tree representing a vector with a 1000000 of elements will have $\lceil \log_2(1000000) \rceil = 20$ levels, a 32-ary tree will have only $\lceil \log_3 2(1000000) \rceil = 4$ levels.

The `pvector` library uses 32-ary trees.

2.2 Implicit sharing

Purelly functional structure never changes. So in case of any changes unchanged parts of the structure can be safelly shared between a new copy and the old one.

[1] Purely Functional Data Structures by Chris Okasaki, Carnegie Mellon University, Pittsburgh, 1996; see also his book with the same name — Purely Functional Data Structures, Cambridge University Press, 1999



My reporoduction of a picture by Jean Niklas L'Orange
from `https://hypirion.com/musings/understanding-persistent-vector-pt-1`

At the illustrations orange nodes are shared by the old vector and the new one. This approach can give us a significant optimization of memory used (see `https://www.youtube.com/watch?v=sPhpelUfu8Q` for an example of editing a file with a size greater than the memory available).

2.3 Index bits as a route in a tree

We can consider our tree of arrays as a *trie* — a key/value data structure where the key is stored as a path to the given node. So, in a tree with a branching factor m the index in a positional numeral system with a radix m will describe a path to the element: the leftmost digit will describe the number (zero-based) of the root children, the next one — what child node to choose on the level 2 and so on, so the rightmost — the number of the target element in the leaf.

If the branching factor is the power of two, we can implement all the needed operations as a bit shifts and other bitwise operations. Those operations are very fast on modern computers.

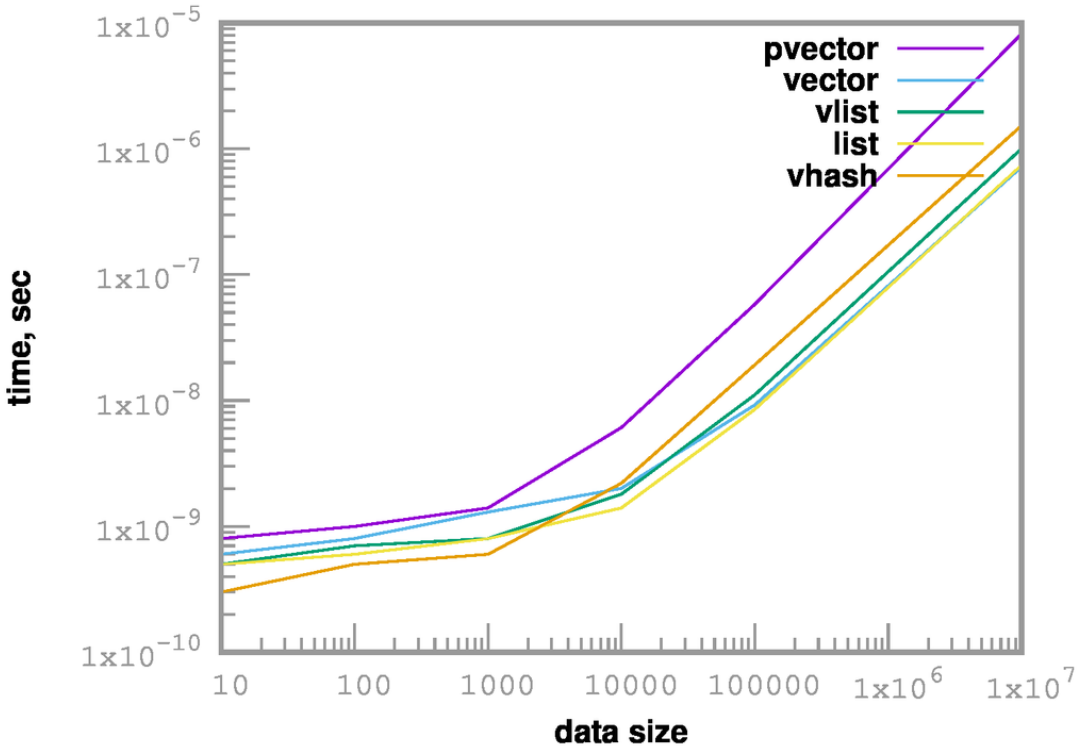
3 Optimizations

3.1 Elimination of n-times element lookup in full-vector operations (`fold`, `foldi`, `map`)

Every index lookup takes $O(\log_m n)$ steps, where n is the vector size and m is the branching factor.

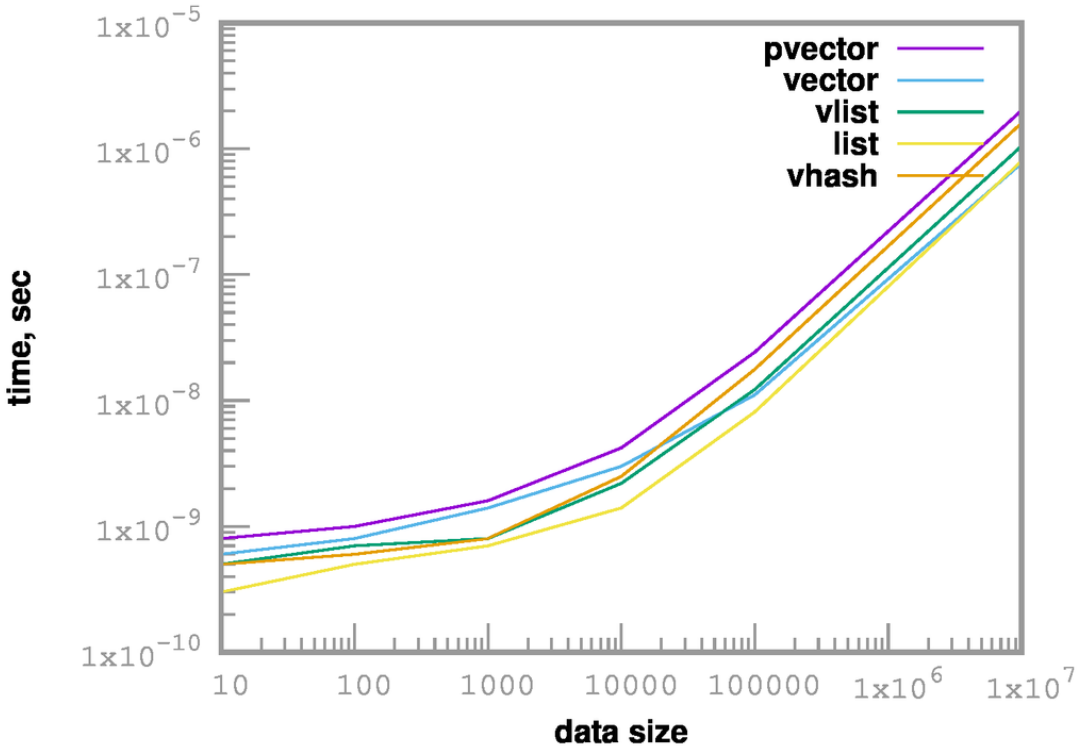
But in the operations with the whole vector (folds, mapping, search) we don't need to get every element by index. We can process the whole leafs instead.

According to Jean Niklas L'Orange's master thesis[1] the lookup of each element in the tree has the amortised $O(1)$ complexity, so the whole operation has $O(n)$ complexity (instead of $O(n \log n)$ for naïve solution).

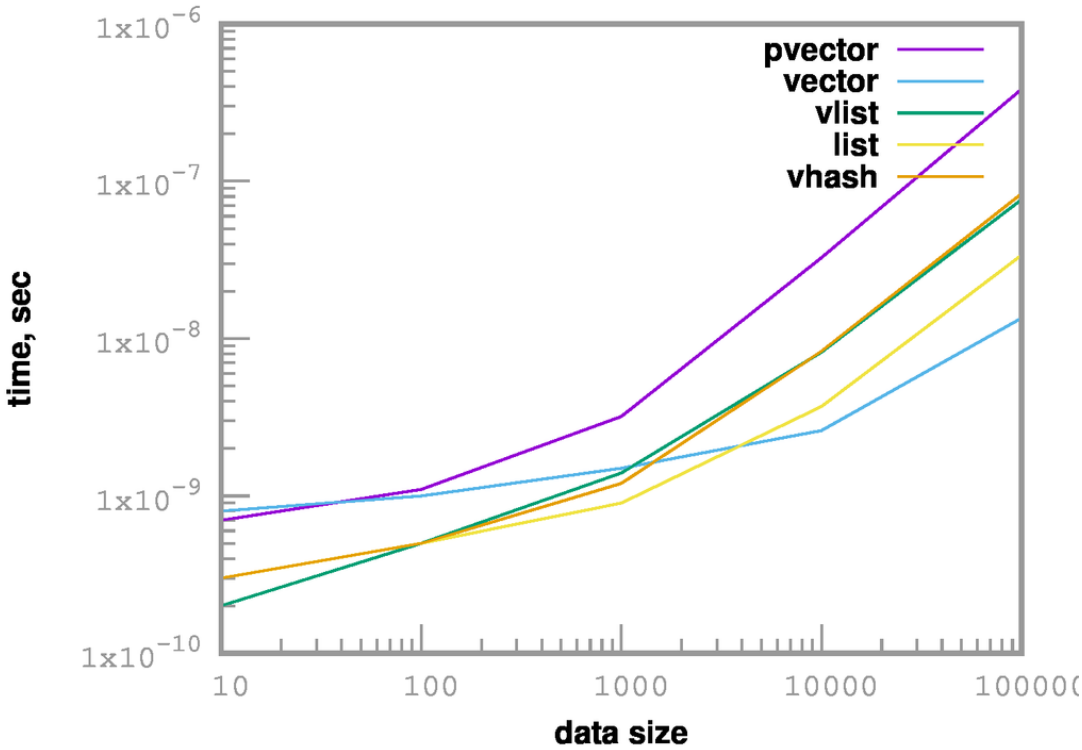


Performance of `fold` *function before optimization.*

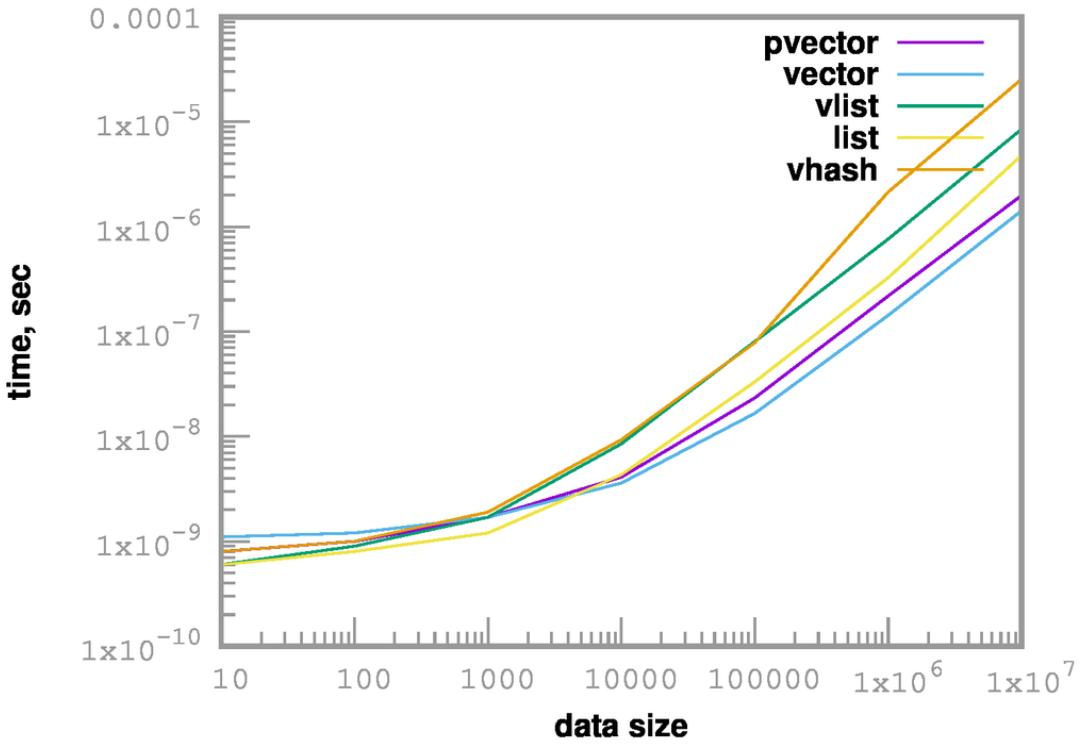
[1] Improving RRB-Tree Performance through Transience by Jean Niklas L'Orange, Norwegian University of Science and Technology, 2014, pp.21–22, Theorem 2.5 (`https://hypirion.com/thesis.pdf`)



Performance of `fold` *function after optimization.*



Performance of `map` *function before optimization.*



Performance of `map` *function after optimization.*

4 API

The following is the list of modules provided by this library.

4.1 (pvector)

pvector is a persistent vector library for Guile Scheme

This module provides the following interface.

4.1.1 Macros

`pvector-length` [Macro]

```
#<syntax-transformer pvector-length>
```

`pvector?` [Macro]

```
#<syntax-transformer pvector?>
```

4.1.2 Procedures

`list->pvector` *l* [Procedure]

(list->pvector l) -> pvector

Converts elements of list to pvector

`make-pvector` [Procedure]

(make-pvector) -> pvector

Creates empty persistent vector

`pvector` *. l* [Procedure]

(pvector l) -> pvector

Creates persistent vector from argument list `l`

`pvector->list` *pv* [Procedure]

(pvector->list pv) -> list

Converts values of pvector to list

`pvector-append` *pv other* [Procedure]

(pvector-append pv other) -> vector

Adds of element of persistent vector *other* to the end of the persistent vector *pv*

`pvector-cons` *v pv* [Procedure]

(pvector-cons v pv) -> pvector

Adds *v* to the end of persistent vector *pv*

`pvector-drop-last` *pv* [Procedure]

Returns a copy of persistent vector *pv*, but without a last element

`pvector-empty?` *pv* [Procedure]

(pvector-empty? pv) -> bool

Checks if `pv` is empty

`pvector-fold` *f acc pv* [Procedure]

(pvector-fold f acc pv) -> pvector

Accepts function `f`, accumulator `acc` and pvector `pv`. Function `f` accepts the element of `pvector` and an accumulator value and returns a new accumulator value. `pvector-fold` returns a result of a sequential application of `f` to all the values of `pvector`, with accumulating intermediate results in `acc`

`pvector-foldi` *f acc pv* [Procedure]

(pvector-fold f acc pv) -> pvector

Accepts function `f`, accumulator `acc` and pvector `pv`. Function `f` accepts an index of the current element, the current element of `pvector` and an accumulator value and returns a new accumulator value. `pvector-fold` returns a result of a sequential application of `f` to all the values of `pvector`, with accumulating intermediate results in `acc`

`pvector-map` *f pv* [Procedure]

(pvector-map f pv) -> pvector

Apply `f` to all values of `pv`. Accepts function `f` and pvector `pv`. Function `f` accepts the element of `pv` and returns the respective value for the new `pvector`.

`pvector-push` *pv v* [Procedure]

(pvector-push pv v) -> pvector

Adds *v* to the end of persistent vector *pv*

`pvector-ref` *pv index* [Procedure]

(pvector-ref pv index) -> value

Returns element of persistent vector *pv* with an index *index*

`pvector-set` *pv index v* [Procedure]

(pvector-set pv index v) -> pvector

Returns a new pvector, like *pv*, but with a *v* at index *index*