The Slicetable: a practical text editor buffer structure

Thomas Qu

August 11, 2021

Abstract

Both established and recent text editors tend to exhibit suboptimal performance with files of large size, long line length or whilst performing intensive bulk editing operations such as find/replace. However unresponsiveness in these conditions is far from a foregone conclusion given the capabilities of modern hardware. The bottleneck tends to be the underlying text buffer structure, typically implemented as naive chunked sequences which do not efficiently exploit the properties of the text editing domain.

In this paper we present a novel structure generalizing the rope and piece-table designs commonly employed in existing editors, and an efficient persistent implementation in the C language. We also consider the integration of our implementation into the environments of high-level languages perhaps more suited to editor development. Further, we present benchmarks of realistic editing scenarios which demonstrate the superior performance of slicetables compared to state-of-the-art structures used in existing editors as well as the recent functional Relaxed Radix Balanced Trees.

1 Introduction

The domain of editor structures has been analyzed fairly thoroughly by Finseth. [1] However recent advances in hardware including the exponential growth of persistent storage capacity, complex cache hierarchies and aggressive speculative execution mean new strategies must be developed to adapt editor structures to modern systems. Throughout this paper a *text buffer* will refer to a data structure suitable for maintaining a list of bytes representing a document being edited within a text editor, supporting at least the operations

- *load(file)* which loads a new text buffer from disk
- *save(file)* which writes the text buffer's contents to disk
- *range(i, n)* reports a range of n bytes starting at i
- *insert(i, c)* which inserts the character c at after the ith byte in the text buffer
- *delete(i)*[1] which the ith byte in the text buffer

We define an *edit* operation as being either an insert or a delete.

Note that indexing in codepoints is unnecessary in an text buffer, as in the vast majority of cases code operates sequentially on grapheme clusters (e.g. deleting a character, cursor movement) or bytes (e.g. searching). Further, the desired behavior is unclear in the case of files containing invalid multibyte sequences, which may often occur when dealing with binary files or corrupted data. Overall the additional complexity of tracking codepoints due to multibyte encodings does not seem to be generally worthwhile in the context of a text editor.

1.1 Efficiency of editing operations

The simplest representation of a text buffer is to lay out a file's bytes in a contiguous array. However

[1] While a generic replace(range, string) operation is conceptually elegant by encapsulating both insert and delete, it adds to implementation complexity unnecessarily. Caching likely mitigates the cost of performing a delete then insert when a replacement is actually needed.

then, the time taken for insertions near the start of the buffer grows $O(n)$, where n will be taken to mean the byte size of the file as all the text after the insertion point must be shifted to make room. Modern text editors typically employ one of three techniques:

- the gap buffer: use two arrays, with edits taking place at the end of the first array, which are $O(1)$. Text is copied to and from the end of the first array as the cursor moves.
- Divide the file into chunks of a maximum size f so that insertions and deletions move at most f bytes. Includes ropes and arrays of lines.
- the piece-table approach, where document is represented as a list of fat pointers (pointers to text and a length), which can be manipulated rather than large quantities of text.

1.2 File loading speed

In an editor we want to load large files quickly, whether the user opens a file for editing or by accident. Text buffer structures which manipulate an in-memory list of fixed-size chunks, such as the rope, do not scale well in this respect, as large file's full contents must be copied. Further, it's possible files may not even fit in memory. Given the relatively large capacities of modern persistent storage devices, it is reasonable to expect text editors to handle files (e.g. log files) of multiple hundred megabytes to gigabytes in size.

A solution is to implement a paging system, storing unused chunks of files on disk and loading them on demand. The well known Vim editor takes this approach.[2] However this involves considerable implementation complexity (better delegated to the operating system) as well as consequences for performance due to increased system calls for reading from disk.

The piece-table solves this problem by utilizing the operating system's virtual memory facilities to perform paging. Modern operating systems allow the usage of virtual address spaces larger than main memory by interfacing with the CPU's MMU. When a CPU attempts to load an unmapped chunk of memory, the MMU generates a page fault, signalling the operating system to load the required page (typically a few kB) of memory into RAM. The fetch is then restarted and proceeds normally.

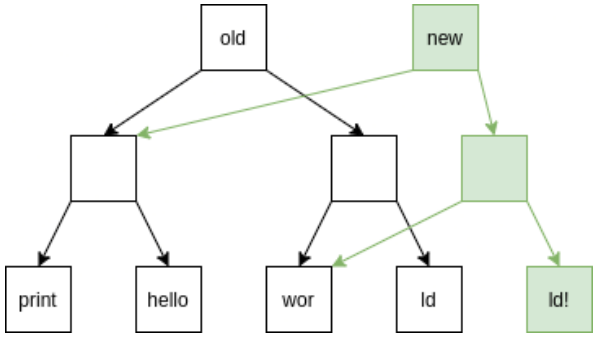

Figure 1: New represents the document returned after adding an exclamation mark, whose contents can be read via the 4 connected bottom boxes from left to right. Directional arrows represent pointers. Note how only $O(logn)$ nodes along the path to the modified chunk need to be updated.

1.3 Persistence

A modern editor is required to retain old versions of documents being edited for the purposes of undo/redo functionality. The typical approach is to keep a operation log of inserts/deletes and implement the ability to revert them.

Another approach is taken by Atom's superstring,[3] which essentially manipulates edits as an in-memory diff. Finseth [2] comments that the 'difference file' method eventually requires a separate text buffer to manage inserted text, which is visible when large contiguous portions of text are entered into Atom, due to it tracking inserted text via an array. However his argument that edit tracking may entail additional memory overhead is annulled by the need to support undo, as well as the large secondary memory capacities available today.

However we instead implemented a *persistent* structure, where edit operations return a new copy of the data structure sharing structure with the original. The extra overhead of this design is reasonable considering that the operation log method is still required to maintain a similar quantity of information - for example a copy must be made of deleted segments of text so that they may be undone later. There are other tradeoffs inherent to

[2] https://www.free-soft.org/FSM/english/issue01/vim.html

[3] https://github.com/atom/superstring

piece-tables structures, which will be detailed later. The maintenance of deleted and inserted text becomes implicit in our persistent design, which provides the further benefits of simple concurrent use (see figure 1).

1.4 Concurrency

Some editors can perform background saving and code analysis, which is useful when dealing with large files, allowing the user to proceed while the file is written to disk or parsed. This requires the data structure to support concurrent reads for safety.[4] We take the approach of functional updates via path copying so that old persistent versions are always safe for readers, though they may temporarily operate on an outdated copy.

1.5 Mark maintenance

A crucial consideration is the efficient maintenance of several *marks* (representing the user's editing cursor(s), compiler diagnostics, the first character of views into the buffer, etc.), which must be updated to correspond to a new index when edit operations occur (e.g. typing on the line above a compiler error should move the error forward in the document). Essentially, marks are attached to a segment of *text* rather than a fixed byte index.

While it is possible to manage marks within the structure itself, [5] the result is an increase the implementation complexity of edit operations, mixing with 'hot' code updating the structure itself. This was observed to impact cache locality, but it remains open whether some alternative approach works efficiently.

A more common approach is to employ a interval tree or similar structure which maintains the marks separately from the text buffer. However this still suffers from poor locality.

On the other hand, we observe that typically a small constant number of marks is necessary for maintaining views into the buffer and keep them in a simple array. A efficient mechanism for updating them during bulk replace operations is described later. It should be observed that typical multiple-cursor editing can be implemented by a macro system which only updates selections on screen, and commits the rest of the edits in bulk, with the benefit of far better locality.

2 Analysis of related work

2.1 Chunking

Chunking structures such as ropes divide the file into smaller chunks, so less text may be moved during edit operations. Some editors use an array of lines, causing edits to scale linearly in the length of lines. This method performs poorly with certain file types. [6]

Another variation commonly known as the rope[7], such the invariant that each text chunk must have contain between $C/2$ and C bytes, where C is a constant typically near the page size, the unit in which virtual memory is allocated.

```
node_offset(node, size_t *i)
  slot = 0
  while(*key > node->size[slot]) {
     // key stores an offset on exit
    *key -= node->size[slot++]
  }
  return i
```

Listing 1: Searching in an (a,b)-tree. Most implementations seem to have two branches per entry per level, which is suboptimal, as edge cases may be handled prior to recursion.

Arranging chunks in an (a,b)-tree, we may simply replace keys with the cumulative size of each corresponding subtree - the search code repeated at each level is shown in listing 1 and works for similar structures. Note that the loop terminates for all $0 \leq i \leq size(buf)$ - in particular for the size of the buffer, which causes the loop to terminate on the last chunk as i will equal the final chunk's size. A similar routine works if the chunk list is

[4] usually it does not make sense to support concurrent writes in a single user context, as edits will race with the user, leading to unintuitive results. The Xi project struggled with this problem.

[5] See github.com/Plisp/vico/blob/master/src/core/piece-table.lisp. Some appealing properties include O(1) local edits and finding the next/previous chunk

[6] e.g. minified javascript, script terminal dumps

[7] though the original rope structure is specifically a binary tree structure implementing edits with a split routine and confluently persistent concatenation

represented as a balanced BST, with an extra field at each node holding the cumulative size of its left subtree. This trick is well known and is used by Abiword [3] and VSCode. [4] The idea of storing cumulative sums can be viewed as analogous to the Fenwick tree. [5]

When inserting text, the size of a chunk may exceed C, requiring it to be split. If a chunk l would overflow, the combined size of inserted text and the original content exceeds C. Thus we can split the combined text into two chunks containing at least $C/2$, satisfying the invariant.

Also, deletions may cause a chunk to temporarily fall below $C/2$ bytes, becoming underfull. If a chunk l satisfying the invariant is adjacent to the underfull leaf u, if their combined text exceeds C we may restore the invariant by copying up to $C/2$ bytes from l into u. Otherwise merge it with l into a single chunk containing below C but more than l initially had, which is at least $C/2$.

It is easy to see that in both these cases at most C bytes are moved during each rebalancing operation and invariants are restored. Assuming the chunks are stored in a log-balanced tree with minimum branching factor $A > 1$, edit operations execute in $\Theta(log_A n)$ worst case.

786115235258909781

2.2 Piece-Table

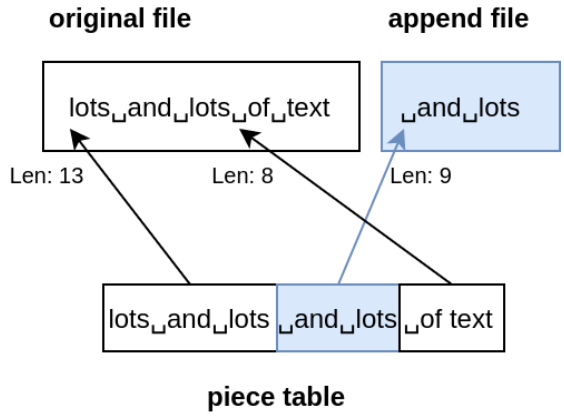

Figure 2: The pointers into the files are annotated with the length of the segment they refer to, similar to how they would be represented in memory.

The piece-table is a less common approach which avoids any mutation of text chunks by maintaining a list of *pieces* ((fat) pointers annotated with the positive length of the text segment they represent) into append-only chunks of text. As text never moves in memory once it's inserted, pieces may trivially point into these chunks. Further, implementing *load* is trivial, as we can simply request for the file to be mapped onto the virtual address space and setup a single piece pointing to the start of that chunk, with a length equal to the file's length in bytes.

In figure 2, a file initially contains "lots and lots of text", and the text " and lots" was just inserted. This is accomplished by splitting the first piece into two (note no text is moved in the process, so this is $O(1)$) and inserting a new piece in the list between the two halves, pointing to the inserted text. As more text is added, it is appended to the append file and new pieces pointing to it can be inserted in the list. Deletions involve similar manipulations of pieces, and are left as an exercise to the reader.

One downside of this approach is that accumulating a large number of edits, especially over a short period of time (perhaps during a global bulk search/replace) degrades performance significantly due to a proliferation of pieces in the list. The worst case is total fragmentation of the file into n 1-byte pieces in the piece list, C (4000) times more than a chunking structure. This behavior is especially apparent in editors such as Vis,[8] which employ a linked list to represent the piece table, leading to catastrophic $O(n)$ reads.

A common improvement on the classic linked-list is a red-black tree design, which is utilized by the Abiword word processor and (later heavily inspired) VSCode editor, achieving $O(log\ n)$ reads and edits in the worst case. However this is still greatly suboptimal, as potentially orders of magnitudes more pieces must be maintained in a list than necessary.

3 Slicetable

As we have seen, most chunking structures (e.g. the "rope" as the term is commonly used) necessitate either the loading of whole files into memory, or a custom implementation of a paging system.

[8] https://github.com/martanne/vis

The key idea of the Slicetable is to explicitly utilize the OS's paging facilities by manipulating large unmodified spans of text (viz. the freshly loaded file) through fat pointers in the manner of a piece-table whilst small chunks are mutated in-place and balanced as described earlier. In the following text we refer to these fat pointers as *slices*. This also allows for an efficient implementation of bulk inserting strings larger than C without splitting it into chunks and copying, which could be useful e.g. when inserting a large amount of output from a shell command.

In practice, we copy small portions of the original document as it fragments (less than some small constant C similar to the C for ropes) into *small* slices which may be modified inplace, whilst slices larger than C are fragmented during edits in the manner of a piece table . It should be observed that setting $C = 0$ is equivalent to a piece-table, and hence our structure constitutes a generalization of the piece-table.

However it is greatly wasteful in our case to utilize the obvious invariant of ensuring all small slices of text exceed $H/2$, as scattered edits will incur an allocation and copy of $C/2$ bytes from the file mapping for each operation to ensure a minimum size. To avoid this, we introduce the concept of *relaxed merging* (see listing 2), in contrast with the strict merging invariant of ropes Here we merge adjacent descriptors only if their combined content does not exceed C, maintaining an worst-case fill factor close to 1/2. TODO prove this below

```
merge_slices(spans[], data[], int fill)
  i = 1
  while(i < fill) {
    if(spans[i-1] + spans[i] <= C) {
      data[i-1] ++= data[i]
      spans[i-1] += spans[i]
      free(data[i])
      spans[i,fill-1] = spans[i+1,fill]
      data[i,fill-1] = data[i+1,fill]
      fill -= 1
    } else {
      i += 1
    }
  }
  return fill
```

Listing 2: Block merging algorithm

3.1 Height bound vs ropes

TODO

- similar implementation complexity to ropes
- discuss mark maintenance briefly. Note that lazy maintenance is not sufficient in an editor, and so it is efficient to do it elsewhere (record changes in array of index/cumulative offset, update in logm).
- mention truncation optimization for appends in delete leaf
- large blocks may only be created due to large insertions - usually not reverted unlike small changes, which should not be recorded due to transient operations
- slice merging - present independently of index structure - we want to shift minimal data upon inserts - but maintain a tunable bound on number of slices (better than pt)
- in both cases, large buffers are kept immutable and shared - on average, insertion into small pieces, which is equivalent in cost to typical balancing, or middle of very large ¿2f pieces, which is optimal. Especially space-efficient when most of file is untouched

  - inserted data is either converted to large or small slice
- standard buffer merging - lazily loaded rope - insertion into small always done directly, if overflow, split into two with half contents in each buffer - deletion may cause underflow - insertion into large may split off small, which must be merged using underflow handling - unclear what should occur when both adjacent slices are large. Splitting off is often unnecessary work - consider a disjoint set of small insertions (separated by over f bytes). Then stealing copies f/2 bytes for each insertion. Does not reduce number of descriptors significantly
- relaxed merging algorithm - ensure no adjacent

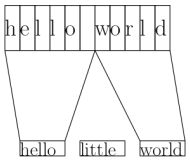

Figure 3: Example of a slicetable after an edit

slices ¡= f for

$$B \cdot \frac{f}{2} - f/2$$

fill factor. - adjacent small slices in the region are merged left where possible - if combined fill ¿f, we promote to a large slice

- no stealing cases - unnecessary
- relaxed splitting seems almost optimal for search/replace, resulting in simple appends
- however we may as well do rebalancing for inner nodes, as in reading the fill we have already loaded it into cache. Implementing stealing then merely takes a single conditional.

4 Implementation

4.1 Index structure

Our implementation is based on a b+-tree index structure, which improves cache performance by reducing block transfers.

We make the decision not to track line offsets, although it should be noted that they can simply be tracked with an extra field at each node and in text chunks, similar to the technique used for indexing bytes. However it should be noted that this necessitates eventually reading in the whole file, defeating the purpose of utilizing virtual paging. [9].

We concluded that in their typical use of assigning compiler diagnostics to locations in a file, the bottleneck will typically be the compiler rather than the editor structure (this should happen asynchronously in any case). The use of line numbers for identifying locations in a file is simply satisfied by referring to features of the document instead, such as function definitions, sections, etc. Therefore absolute line numbering is not optimized to $O(logn)$, but iterators allow traversal of newlines at multiple GB/s, which is certainly sufficient for basic display routines and cursor movements.

[9]it may be possible to do this asynchronously, but this involves further complexity

4.2 Language Integration

Our implementation requires low level communication with the OS, but editor development is likely more suited to a high level language with appropriate abstractions.

Concurrency is implemented by a *clone* primitive, which uses the standard path-copying method to create a snapshot of the text buffer. This primitive is unsafe, as the caller may neglect to call it prior to concurrent use, but allows for a transient API to be implemented by the type system of a high-level language. Edit operations on the persistent version always call *clone*, whilst the transient type only needs to clone once, and tracks its owning thread to ensure accesses do not occur from other threads.

Rather than implement error handling on the C side, which has rudimentary error reporting capabilities, do it in the high-level language.

5 Evaluation

5.1 Methodology

To test the performance of slicetables against other structures in the key areas listed in section 1, we tested the following:

- Load a 5.9mb XML file from disk
- Find up to 100000 occurences of 'thing' and replace them with 'thang'
- Find the 100000 occurences of 'thang' and replace them with 'thong', simulating a mistake

vis' linked list piece-tables (fast file loading, slow search/replace). rb tree slicetable (tuned as piece-table, slice table, talk about vscode, cache) red black tree is 25 percent slower than Btree piece-table searching/replacing '1000' - scan performance matters

We used Ropey as a primary competitor, as it was the most performant structure known, employing a balanced b+tree structure to track a list of fixed-size chunks. It was possible to remove line tracking, but not the codepoint tracking functionality which was built into the core of the structure (though the search implementation uses the Bytes iterator, which was found to increase performance by around 10%). Therefore these benchmarks are not fully indicative of the performance gap between the structures. However we still believe that the 2x increase in performance cannot be simply put down to locality.

6 Further work

Modern CPUs aggressively pipeline code execution, fetching memory and instructions long before they are executed. The Btree is not an ideal index structure for main memory [6] as branch mispredictions incurred by linear search at each level leads to pipeline stalls, while the CPU refreshes the instruction cache.

Converting the underlying structure to a trie is an interesting future direction for research. Same cache accesses, improvement in branches during search? (this probably isn't really worth it on reconsideration, as scan performance and insertion are more important)

The generalization of a focus is interesting but adds significant complexity to implementation. [7] It remains unclear whether the performance advantages in certain cases is significant, and there remains the problem of maintaining multiple cursors, which cannot be represented as a single focus.

References

[1] C. Crowley. Data structures for text sequences. 1998.

[2] Craig A. Finseth. *The Craft of Text Editing.* Springer-Verlag & Co, 1999. `https://www.finseth.com/craft/`.

[3] Joaquin Cuenca Abela. Improving the abiword's piece table.

[4] Peng Lyu. Text buffer reimplementation.

[5] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24:327–336, 1994.

[6] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[7] Juan Pedro Bolívar Puente. Persistence for the masses: Rrb-vectors in a systems language. *Proc. ACM Program. Lang.*, 1, August 2017.