Atom's new concurrency-friendly buffer implementation

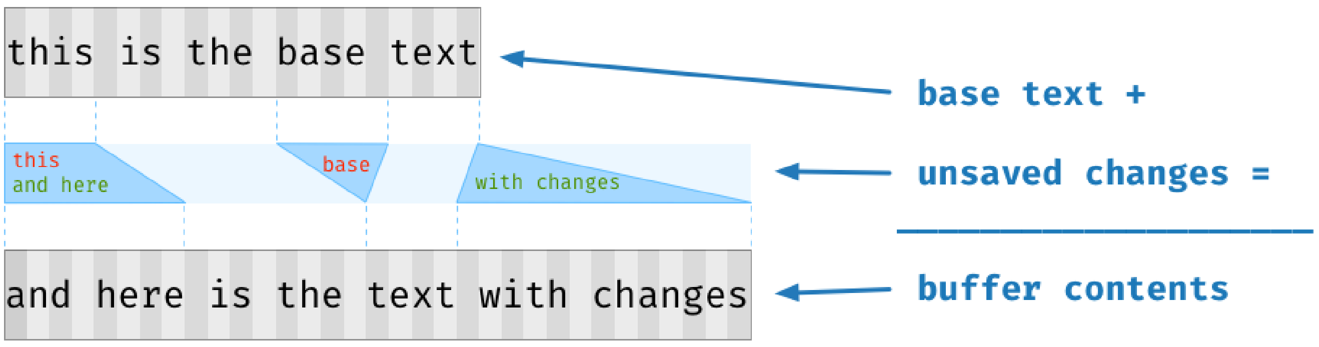
October 12, 2017 nathansobo Tweet

Several Atom features depend on potentially long-running computations based on the contents of open buffers, but until recently, it was only possible to access a buffer's text from JavaScript running on the main thread. This made it difficult to guarantee Atom's responsiveness in all scenarios, especially when editing larger files.

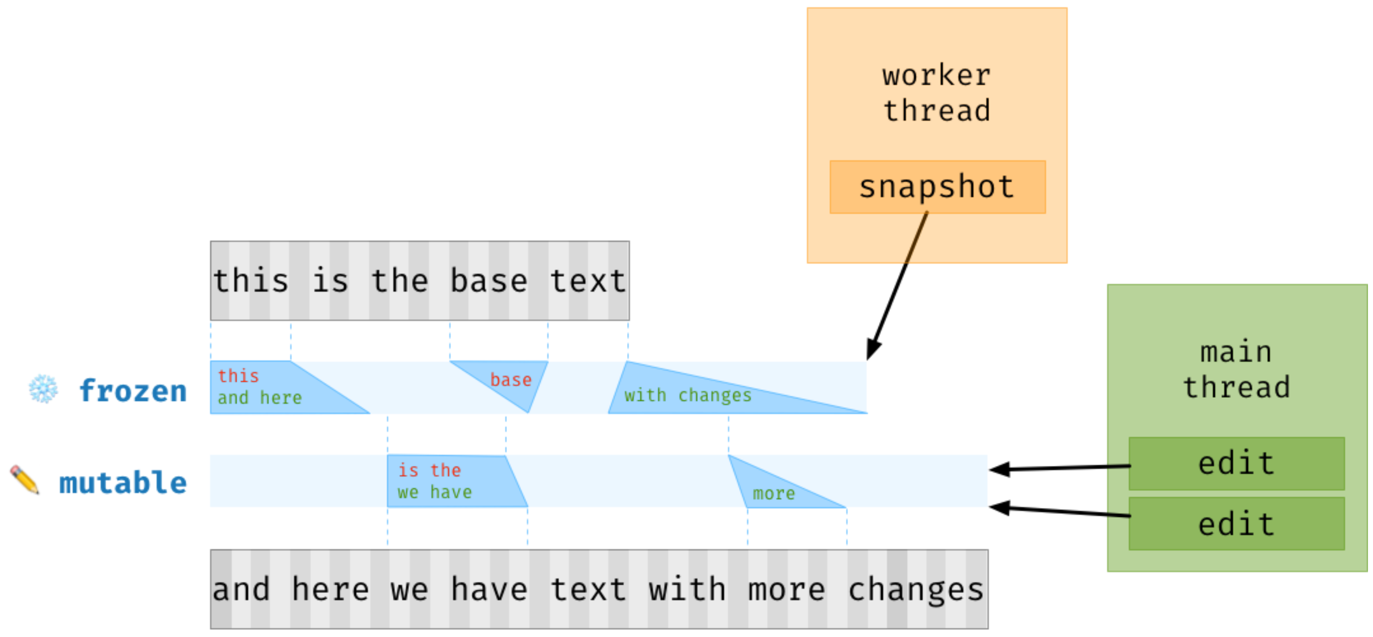
That situation changed with the release of Atom 1.19, which opened the door to greatly increased parallelism via a new text-storage data structure that is implemented in C++. This new design provides many benefits for performance and scalability, chief among them the ability for worker threads to read snapshots of previous buffer states without blocking writes on the main thread. In this post, we'll describe Atom's new approach to text storage in depth, then explore the first of many optimizations it makes possible.

Layering changes

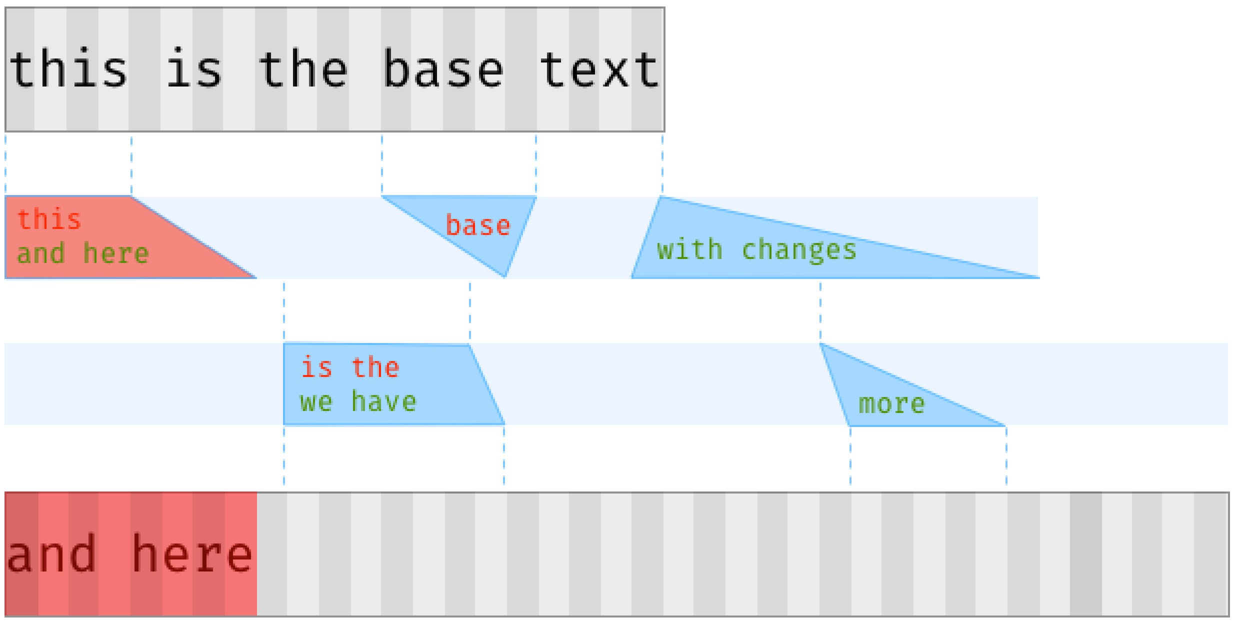
The key idea behind Atom's new buffer representation is to separate the contents of a buffer into two major pieces of state. First, there is the *base text*, which corresponds to the version of the document that was most recently read from or written to disk. The base text is immutable and stored in a single, contiguously allocated block of memory. Superimposed on top of the base text are the *unsaved changes*, which are stored in a separate sparse data structure called a *patch*. To record edits, rather than shifting the buffer's entire contents around in memory, we simply mutate the patch.



There can actually be multiple layers of patches in existence at any one time. The patch in the topmost layer is always mutable, but we can create a read-only *snapshot* of the current buffer contents by freezing the topmost patch and pushing a new patch to the top of the stack. Edits flow into this new patch until the snapshot is no longer needed, at which point the topmost patch can be merged down into the patch on the previous layer.



To read the buffer state, we iterate through successive “chunks”, where each chunk corresponds to a change from one of the layered patches or a slice from the array containing the base text.



The patch data structure

At the heart of this entire system is the `Patch` data structure, which describes how a sequence of textual changes can be applied on some input to produce a new output. It contains essentially the same information as you’d obtain by running a `diff` on the input and the output, but instead of being constructed by comparing the contents of two buffers, it is constructed incrementally by composing together a series of edits.

The problem

To better understand the problem solved by the `Patch`, consider the following example. We start with a buffer containing `xxxx`, then perform the following insertions:

- `insert B @ 2` -> `xxBxx`
- `insert C @ 4` -> `xxBxCx`
- `insert A @ 1` -> `xAxBxCx`

Afterward, we want a summary of our changes, expressed as a character-wise diff like this:

```
[
  {oldRange: [1, 1], oldText: '', newRange: [1, 2], newText: 'A'},
  {oldRange: [2, 2], oldText: '', newRange: [3, 4], newText: 'B'},
  {oldRange: [3, 3], oldText: '', newRange: [5, 6], newText: 'C'}
]
```

Each entry in this diff has an `oldRange` that does not account for any other changes present in the patch. For example, the entry describing the insertion of `C` has an `oldRange` of `[3, 3]`, which excludes the impact of inserting `A` and `B`. In contrast, each change's `newRange` reflects the spatial impact of all other edits in the patch. For example, the insertion of `C` has `newRange` of `[5, 6]`, which accounts for the insertion of `A` and `B` earlier in the buffer.

This kind of summary isn't available from the original stream of edits without additional processing. Consider the insertion of `C` at index `4`. This index already accounts for the prior insertion of `B`, but it doesn't account for `A`, which was inserted prior to `C` in *space* but subsequent to `C` in *time*. To produce the `oldRange` and `newRange` in the diff shown above, we need to understand the spatial relationship of each change to every other change, regardless of their temporal ordering.

A naive solution

A simple solution to this problem is to store each change in a list, with each change storing its `oldText`, `newText`, and `distanceFromPreviousChange`. We determine the insertion location for each new entry in this list by walking over existing changes. Here is how the change list would evolve given the insertions from the example above.

```
assert.deepEqual(patch.changes, [])

patch.splice(2, '', 'B')

assert.deepEqual(patch.changes, [
  {distanceFromPreviousChange: 2, oldText: '', newText: 'B'}
])

patch.splice(4, '', 'C')
```

```
assert.deepEqual(patch.changes, [
  {oldText: '', newText: 'B', distanceFromPreviousChange: 2},
  {oldText: '', newText: 'C', distanceFromPreviousChange: 1}
])

patch.splice(1, '', 'A')

assert.deepEqual(patch.changes, [
  {oldText: '', newText: 'A', distanceFromPreviousChange: 1},
  {oldText: '', newText: 'B', distanceFromPreviousChange: 1},
  {oldText: '', newText: 'C', distanceFromPreviousChange: 1}
])
```

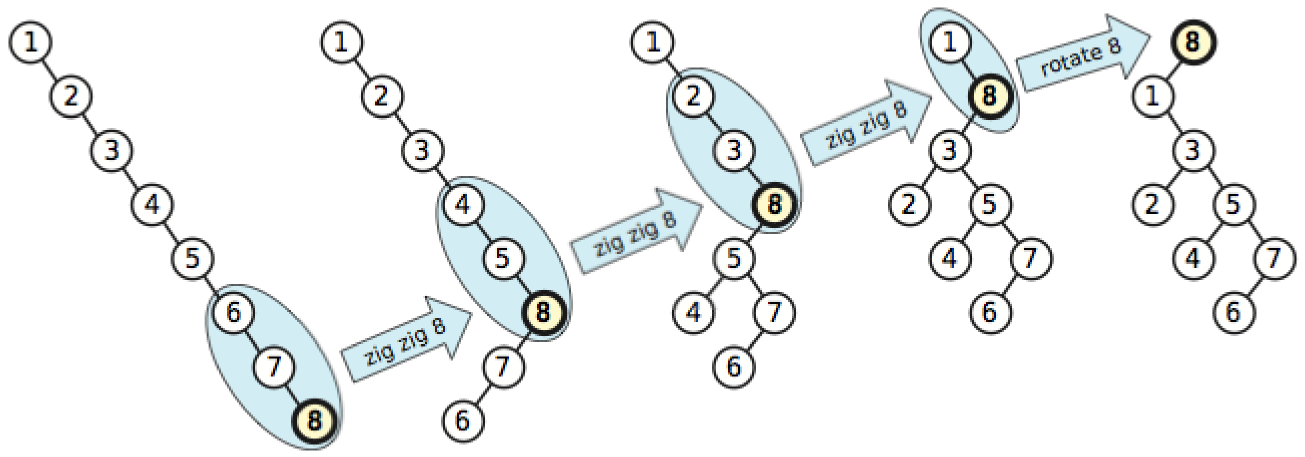
In this case, `oldText` is always empty because we're only performing insertions, but it's easy to express deletions or replacements by using a non-empty value for `oldText`. Once we have this list of changes, we can produce the desired summary by iterating through the list, basing the start of each change's `oldRange` and `newRange` on the end of these ranges in the previous change.

Splay trees

The problem with the above approach is that inserting a change in the list may require us to examine every other change, which produces a time complexity of $O(n^2)$.

To ensure good performance in the production implementation, we improve the time bound to $O(n \cdot \log_2 n)$ by using a *splay tree* instead of a simple list. Splay trees are a version of binary search trees that are fairly simple to implement and have a really cool property of being "self-optimizing". That means that as they are queried and modified, they automatically adjust their structure to make it cheaper to access nodes in close proximity to recently accessed nodes. For randomized workloads, this property isn't helpful, but for workloads exhibiting a high degree of locality such as those that occur in a text editor, this self-optimizing behavior is super useful.

Splay trees revolve around a remarkably simple principle. Each time a node is accessed, it is rotated to the root of the tree through a special series of pointer rotations known as a *splay*. Splaying not only moves the node to the root of the tree, but it also reduces the depth of the tree in the vicinity of the node, ensuring that next time we access a nearby node it will be closer to the root and therefore faster to find.

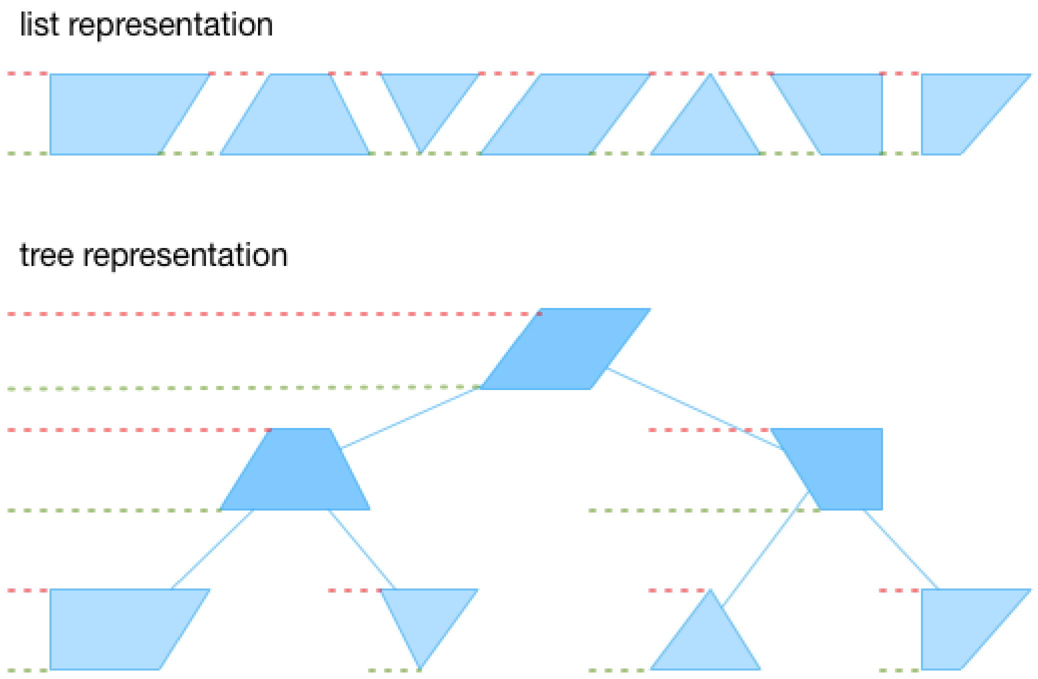


One caveat to this entire approach is that O(n•log$_2$n) is an *amortized* bound. The cost of any single operation could be as much as O(n), but we pay for any expensive operation by restructuring the tree to make subsequent operations cheaper. In practice, this is fine. Any single linear-time operation usually doesn't cause a performance issue. It's only when we start performing *multiple* linear-time operations in a batch that the time complexity becomes quadratic, and it's exactly this situation that splay trees help us avoid.

If you'd like to learn more about splay trees, this lecture by David Karger from MIT is a great introduction.

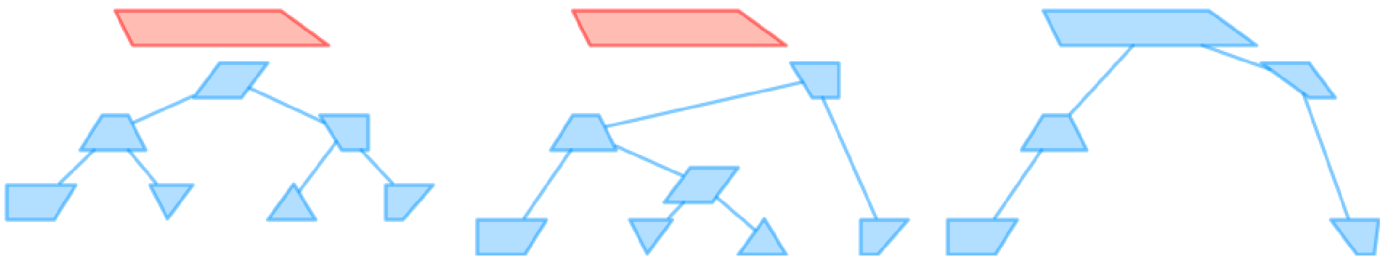
Augmenting splay trees for our use case

In the literature, splay trees are always presented as simple ordered mappings between keys and values. With our `Patch`, we're solving a somewhat more complicated problem: Our tree needs to maintain the position of each node in both the new and old coordinate spaces in such a way that we can efficiently update the positions of all subsequent nodes whenever a new change is inserted. To do this, instead of associating each node with a constant key, we'll associate each node with relatively-expressed values that represent the node's distance from its *left ancestor* in both old and new coordinate spaces.



In the diagram above, each change is shown as a trapezoid that graphically represents the spatial impact of replacing a run of characters. In the list representation we illustrated earlier, the distance from the previous change is always the same in both coordinate spaces, because the text between any two changes is left unmodified. In the tree version, each change stores the distance to its left ancestor, which summarizes the spatial impact of the entire subtree to the left of that change. Each of the darker-shaded nodes above have changes in their left subtrees and therefore differing values for each coordinate space for the distance to their left ancestor. To convert these relative distances into absolute positions, we accumulate a running total in both coordinate spaces as we descend the tree from root to leaf.

To insert a new change, we splay the existing changes that most closely bound the range we are replacing. As we rearrange pointers during the splay operations, we can update the distances to each node's left ancestor based on locally-available information. Once the lower and upper bound nodes have been rotated to the root of the tree, any nodes *between* them will be encompassed by the change we are inserting, meaning they can be deleted. We then insert our new change, merging it with one or both of the nodes at the root of the tree depending on whether it overlaps them.



For our patch structure, splaying is more than just a mechanism for keeping the tree more balanced. We actually depend on the ability to move nodes to the root of the tree in order to splice new changes into the structure. With a more rigidly balanced data structure such as a red-black tree, it would have been more difficult to rotate nodes to the root in this way without violating critical invariants.

It’s worth noting that in all of the above examples, we’ve used scalar values to represent positions and distances for clarity. In reality, these values are expressed as 2-dimensional vectors made up of rows and columns. This adds some complexity, but the basic ideas remain the same. It’s also worth noting that this structure has utility beyond the buffer representation described in this post. We originally created it to aggregate all changes occurring within each transaction so we can notify change listeners with a diff and store the most compact representation in the undo stack. We also use a patch to index the translation between buffer and screen coordinates in the presence of presentation-oriented transformations such as soft-wrapping and code folding. It’s a complex piece of code, but we get a lot out of it.

Some initial optimizations

Moving our buffer implementation to C++ was a win in itself in terms of Atom’s overall efficiency. JavaScript can be quite fast, but fundamentally it’s a scripting language with the unavoidable overhead that implies. By implementing buffers in C++, we eliminate the overhead of JS and gain the control we need to maximize efficiency. We also relieve pressure on the V8 garbage collector by simplifying the heap and allocating fewer short-lived objects on hot code paths. But these improvements are just the beginning. The true value of this new representation is in the optimizations enabled by its layered design.

Efficient backups of unsaved changes

Last January, we had just wrapped up another improvement that enabled Atom to handle much larger files when we discovered a frustrating bottleneck. One of the biggest annoyances when editing big files was the overhead associated with periodically writing the unsaved state of large buffers to disk. At sufficient sizes, even collecting the contents of the buffer up to write asynchronously introduced noticeable pauses, and though we could have muddled through with clever use of `requestIdleCallback` and an output stream, we were concerned about the energy impact of writing that much data several times a minute. We’d been thinking about this new buffer design for a while, and we decided that efficient background saves was a good initial motivation to build it.

For the purposes of crash recovery, we only care about the unsaved changes, which our new buffer representation conveniently provides. Now, instead of writing out the entire contents of the buffer, we simply compose all outstanding layers into a single patch and serialize it to disk along with a digest of the base text. The amount of data we’re writing scales with the number of changes rather than the file size, making it much more efficient in most circumstances. There’s still work to do to around files with tens of megabytes of unsaved changes, but these situations are quite rare.

Asynchronous saves

Prior to 1.19, saving buffers in Atom was a *synchronous* operation 😱. This is because the code path for writing files pre-dated the creation of Electron, and in those days performing asynchronous I/O from a browser-based desktop application wasn't as easy as it is today. Happily, this new buffer implementation gave us an opportunity to finally fix this problem in an elegant fashion. Converting the buffer's contents from UTF-16 to the user's desired encoding and writing them to disk is now performed entirely in C++ on a background thread. Before starting the save, we create a snapshot so that the user is free to make additional changes even when saving is slow, such as when using a network drive.

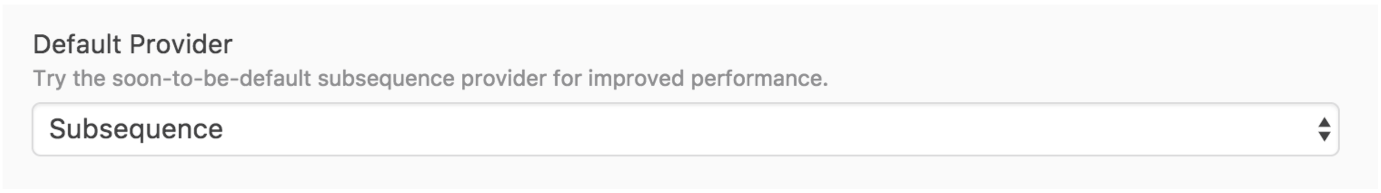
Subsequence-matching in the background during autocomplete

By default, Atom's autocomplete system suggests words from open buffers based on a subsequence match against the characters preceding the cursor. For example, `bna|` would produce `banana`, `bandaid`, and `bandana` as suggestions. We then sort suggestions based on a score indicating the quality of the match.

Prior to Atom 1.22, we implemented this feature by maintaining a unique word list for each buffer and running JavaScript on the main thread to match, score, and sort suggestions. While this worked okay for most files, as file sizes increased, the word lists started to consume serious memory and matching suggestions on the main thread could noticeably block the UI.

Thanks to the new buffer implementation, we're rolling out a new autocomplete provider behind a feature flag with Atom 1.22 that leverages snapshots to do the same job with no memory overhead and no threat to Atom's responsiveness. Most of the heavy lifting is now performed in a new `TextBuffer.findWordsWithSubsequence` method in core that performs matching, scoring, and sorting in a background thread. This means we can start searching for suggestions immediately after each keystroke while other work proceeds on the main thread. By the time the next frame paints, the suggestions are usually available, but we'll never delay a frame while we search for them. In the rare scenario that suggestions take longer than a frame to compute, we'll simply render them in a subsequent frame.

To give this new provider a try today, download 1.22 beta, navigate to `autocomplete-plus` in settings view, and switch the `Default Provider` option to `Subsequence`.


Default Provider
Try the soon-to-be-default subsequence provider for improved performance.
Subsequence

Please let us know if you experience any issues with the new matching logic. If all goes well, this will be the only default provider included in Atom 1.23.

Future returns

This new buffer representation lays the foundation for many improvements in the future. In the near term, the ability to do non-blocking reads in worker threads will help us improve responsiveness in a number of areas, many of which we have yet to explore.

In the long term, switching our buffer implementation to C++ opens the door to porting other subsystems as well. We're gradually building up a native library called `superstring` that implements multiple performance-critical components at the core of Atom, like the patch and text storage data structures described in this post. We interoperate with this library from JavaScript via the V8 embedding APIs, but it also has working Emscripten bindings for use outside of Electron. Now that the critical state associated with the buffer has made its way into `superstring`, we can incrementally port any performance-critical code path that needs access to the contents of buffers.

To be clear, the approachability and flexibility of JavaScript is a huge asset, so we will always think twice before trading those benefits away for the raw performance of C++, but hopefully this blog post has shown that the constraints of JavaScript don't represent a fundamental limitation in our quest to deliver excellent performance in Atom.

Have feedback on this post? Let **@AtomEditor** know on Twitter.

Need help or found a bug? Contact us.