Commentary: Persistent Data Structures

by Alexei Finski, July 3, 2019

A motivational quote by James R.Driscoll, Neil Sarnak, Daniel D.Sleator, and Robert E.Tarjan:
"One method is to store every version explicitly, copying the entire ephemeral structure after each update operation. This costs O(n) time and space per update. An alternative method is to store no versions but instead to store the entire sequence of update operations, rebuilding the current version from scratch each time an access is performed. If storing an update operation takes O(1) space, this method uses only O (m) space, but accessing version i takes O(i) time even if a single update operation takes O(1) time."

Partial Persistence

read

Start with the access pointer corresponding to version *v*. The access pointer points to the root node containing mod with ≤ *v* version closest to *v*. Read the fields of the root node by applying all its mods with version ≤ *v*, sequentially in increasing order. Follow the pointers in the fields to the next nodes and repeat the reading procedure.

=> Given a version *v*, the partially persistent structure is read with O(1) overhead per node in non-amortized worst case.

write

A mod is added to a node. If the node has free mod space, add the mod. Else, make a new node:

1) *New node and current mod*. Make an empty node. Read the old full node by applying all mods in the old full node to its fields and enter the resulting values in the fields of the new node. Apply the current mod to a field in the new node. The mod space of the new node remains empty. The new node gets a version stamp (if in formulation). **The old node remains full and unchanged.**

2) *Backpointers of new node.* Copy backpointers from the old full node to the new node. Follow copied backpointers. Add a mod to each reached node. The mod includes the new version, the index of the field that previously had the pointer to the old full node, and a pointer to the new node. If a reached node is full, it is "copied". The procedure is recursive upto the root node that may need to be "copied".

=> All pointers leading to the old full node from access pointers corresponding to versions in the old full node, are preserved.

3) *Field pointers of new node*. Follow each pointer of the new node and change the backpointer of each reached node. The backpointer now points to the new node instead of the old full node. No recursion necessary.

=> Old full node is locked from recursive updates in the future.

If indegree (recursive updates through backpointers) is bounded by a constant in the ephemeral structure, then the ephemeral structure is made partially persistent by the above construction. The partially persistent structure only requires amortized O(1) factor overhead per update operation and O(1) space per update step (both amortized). Read operation is O(1) in non-amortized worst case.

Amortized analysis. During node copying, the added mod becomes the new node version (1), keeping its mod space empty. Adding a mod does not contribute to $\Delta\Phi$ in the node copying case in amortized analysis, as reflected in the improved potential function that I provide in the lecture notes.

Full Persistence

read

In partial persistence, versions are linearly ordered. A node stores mods corresponding to a subset of the global set of versions. To read the fields of a node, we apply its mods in increasing version order (version numbers may not be consecutive).

In full persistence, versions form a tree. To read the fields of a node at version *v*, we determine which mods in the mod space of the node have versions that are ancestors of *v* in the global tree of versions. Because the timestamp of ancestor versions monotonically increases along the path to *v* in the global version tree, mods with ancestor versions are applied to the fields of the node in increasing version order.

Order maintenance data structure, provides O(1) insertion for maintaining the global version tree and O(1) relative order query, used for determining version ancestors of *v* among versions in the mod space of a node. The number of mods in the mod space of a node is bounded by a constant.

=> Given a version *v*, the fully persistent structure is read with O(1) overhead per node in non-amortized worst case.

write

1) *Split old node*.
a) Split mods into two trees 1/2 : 1/2, by finding a subtree of sufficient size with the ancestor procedure of the order maintenance data structure. This procedure takes O(1) time, because the number of mods in a node is bounded by a constant.
b) Copy the subtree of mods into the new node.
c) Read the old node by applying mods upto the root mod of the copied subtree, and copy the resulting values of fields to the fields of the new node. These values now include backpointer modifications, because backpointers are versioned. The root mod can now be applied and deleted from the mod space of the new node.

2) *Pointers to new node*. Any version in nodes pointing to the old node that is descendent in the global version tree of the versions in the copied subtree, now must be “routed” to the new node instead of the old node. Reading any such version in the old node previously led to the nearest ancestor version, which is in the subtree copied to the new node.

The “routing” of each such version from the pointing node to the old node was provided by its nearest ancestor version in the pointing node with a pointer to the old node. This nearest ancestor version with a pointer to the old node in the pointing node may also “route” versions to the old node that must continue to be “routed” to the old node after copying the version subtree to the new node.

A set of operations that can lead to recursive node splits are described by Driscoll et al. to restore the correct “routing” of versions after a node split.

=> Correct version “routing” through pointers and backpointers is restored after a node split. All versions of the ephemeral data structure remain embedded and can be updated in the fully persistent data structure.

3) *Current mod*. The current modification is now added to a version of interest resulting in new version. The version of interest can be in the old or new node. None of them are full.

If indegree is bounded by a constant in the ephemeral structure, then the ephemeral structure is made fully persistent by the above construction. The fully persistent structure only requires amortized O(1) factor overhead per update operation and O(1) space per update step (both amortized). Read operation is O(1) in non-amortized worst case.

Amortized analysis. The root version of the copied subtree becomes the new node version, reducing the number of stored mods by one. This subtraction is later compensated by the addition of the current mod to one of the nodes. Thus only 2(d + p +1) emptied slots are considered in $\Delta\Phi$ recursively in the node-splitting case in amortized analysis.

References

J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. 1986. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing* (STOC '86). ACM, New York, NY, USA, 109-121. DOI: http://dx.doi.org/10.1145/12130.12142

Lecture materials at https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/lecture-videos/session-1-persistent-data-structures/