Some Basic Concepts And Terminology

PERSISTENT DATA STRUCTURE

PROJECT MEMBERS: ANANNYO DEY, SOUMYAJIT RUDRA SARMA , DEBASMIT ROY, KANKO GHOSH AND KUSHAL DAS

Persistent Data Structures

Introduction and motivation

In computing, a **persistent data structure** or **not ephemeral data structure** is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. The term was introduced in Driscoll, Sarnak, Sleator, and Tarjans' 1986 article.

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. The data structure is **fully persistent** if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called **confluently persistent**. Structures that are not persistent are called *ephemeral*.

These types of data structures are particularly common in logical and functional programming,[2] as languages in those paradigms discourage (or fully forbid) the use of mutable data.

The obvious way to provide persistence is to make a copy of the data structure each time it is changed.

This has the drawback of requiring space and time proportional to the space occupied by the original data structure.

In turns out that we can achieve persistence with O(1) additional space and O(1) slowdown per operation for a broad class of data structures.

Partial versus full persistence

In the partial persistence model, a programmer may query any previous version of a data structure, but may only update the latest version. This implies a linear ordering among each version of the data structure.[3] In the fully persistent model, both updates and queries are allowed on any version of the data structure. In some cases the performance characteristics of querying or updating older versions of a data structure may be allowed to degrade, as is true with the Rope data structure.[4] In addition, a data structure can be referred to as confluently persistent if, in addition to being fully persistent, two versions of the same data structure can be combined to form a new version which is still fully persistent.[5]

Making pointer-based data structures persistent

Now let's talk about how to make arbitrary pointer-based data structures persistent. Eventually, we'll reveal a general way to do this with O(1) additional space and O(1) slowdown, first published by Sleator and Tarjan et al. We're mainly going to discuss partial persistence to make the explanation simpler, but their paper achieves full persistence as well.

First try: fat nodes

One natural way to make a data structure persistent is to add a modification history to every node. Thus, each node knows what its value was at any previous point in time. (For a fully persistent structure, each node would hold a version tree, not just a version history.)

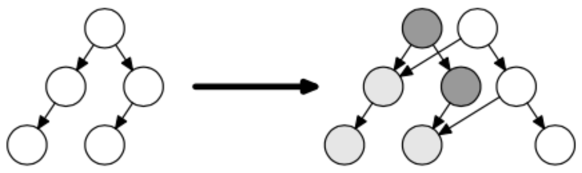
This simple technique requires O(1) space for every modification: we just need to store the new data. Likewise, each modification takes O(1) additional time to store the modification at the end of the modification history. (This is an amortized time bound, assuming we store the modification history in a growable array. A fully persistent data structure would add O(log m) time to every modification, since the version history would have to be kept in a tree of some kind.)

Unfortunately, accesses have bad time behavior. We must find the right version at each node as we traverse the structure, and this takes time. If we've made m modifications, then each access operation has O(log m) slowdown. (In a partially persistent structure, a version is uniquely identified by a timestamp. Since we've arranged the modifications by increasing time, you can find the right version by binary search on the modification history, using the timestamp as key. This takes O(log m) time to find the last modification before an arbitrary timestamp. The time bound is the same for a fully persistent structure, but a tree lookup is required instead of a binary search.)

Second try: path copying

Another simple idea is to make a copy of any node before changing it. Then you have to cascade the change back through the data structure: all nodes that pointed to the old node must be modified to point to the new node instead. These modifications cause more cascading changes, and so on, until you reach a node that nobody else points to—namely, the root. (The cascading changes will always reach the root.) Maintain an array of roots indexed by timestamp; the data structure pointed to by time t's root is exactly time t's data structure. (Some care is required if the structure can contain cycles, but it doesn't change any time bounds.)

Figure shows an example of path copying on a binary search tree. Making a modification creates a new root, but we keep the old root around for later use; it's shown in dark grey. Note that the old and new trees share some structure (light grey nodes).



Path copying on binary search trees

Access time does better on this data structure. Accesses are free, except that you must find the correct root. With m modifications, this costs O(log m) additive lookup time—much better than fat nodes' multiplicative O(log m) slowdown.

Unfortunately, modification time and space is much worse. In fact, it's bounded by the size of the structure, since a single modification may cause the entire structure to be copied. That's O(n).

Path copying applies just as well to fully persistent data structures.

Sleator, Tarjan et al.

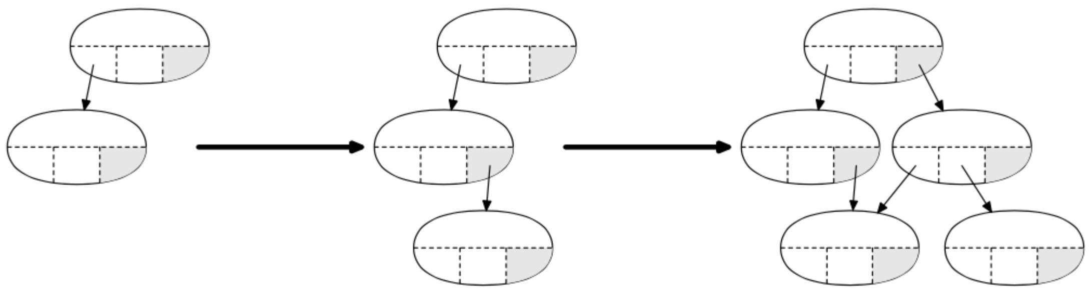
Sleator, Tarjan et al. came up with a way to combine the advantages of fat nodes and path copying, getting O(1) access slowdown and O(1) modification space and time. Here's how they did it, in the special case of trees.

In each node, we store one modification box. This box can hold one modification to the node—either a modification to one of the pointers, or to the node's key, or to some other piece of node-specific data—and a timestamp for when that modification was applied. Initially, every node's modification box is empty.

Whenever we access a node, we check the modification box, and compare its timestamp against the access time. (The access time specifies the version of the data structure that we care about.) If the modification box is empty, or the access time is before the modification time, then we ignore the modification box and just deal with the normal part of the node. On the other hand, if the access time is after the modification time, then we use the value in the modification box, overriding that value in the node. (Say the modification box has a new left pointer. Then we'll use it instead of the normal left pointer, but we'll still use the normal right pointer.)

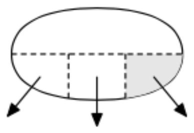
Modifying a node works like this. (We assume that each modification touches one pointer or similar field.) If the node's modification box is empty, then we fill it with the modification. Otherwise, the modification box is full. We make a copy of the node, but using only the latest values. (That is, we overwrite one of the node's fields with the value that was stored in the modification box.) Then we perform the modification directly on the new node, without using the modification box. (We overwrite one of the new node's fields, and its modification box stays empty.) Finally, we cascade this change to the node's parent, just like path copying. (This may involve filling the parent's modification box, or making a copy of the parent recursively. If the node has no parent—it's the root—we add the new root to a sorted array of roots.)

Following figure shows how this works on a persistent search tree. The modification boxes are shown in grey



Modifying a persistent search tree.

With this algorithm, given any time t, at most one modification box exists in the data structure with time t. Thus, a modification at time t splits the tree into three parts: one part contains the data from before time t, one part contains the data from after time t, and one part was unaffected by the modification.



How modifications split the tree on time.

How about time bounds? Well, access time gets an O(1) slowdown (plus an additive O(log m) cost for finding the correct root), just as we'd hoped! (We must check the modification box on each node we access, but that's it.)

Time and space for modifications require amortized analysis. A modification takes O(1) amortized space, and O(1) amortized time. To see why, use a potential function ϕ, where $\phi(T)$ is the number of full live nodes in T. The live nodes of T are just the nodes that are reachable from the current root at the current time (that is, after the last modification). The full live nodes are the live nodes whose modification boxes are full.

So, how much does a modification cost? Each modification involves some number of copies, say k, followed by 1 change to a modification box. (Well, not quite—you could add a new root—but that doesn't change the argument.) Consider each of the k copies. Each costs O(1) space and time, but decreases the potential function by one! (Why? First, the node we copy must be full and live, so it contributes to the potential function. The potential function will only drop, however, if the old node isn't reachable in the new tree. But we know it isn't reachable in the new tree—the next step in the algorithm will be to modify the node's parent to point at the copy! Finally, we know the copy's modification box is empty. Thus, we've replaced a full live node with an empty live node, and ϕ goes down by one.) The final step fills a modification box, which costs O(1) time and increases ϕ by one.

Putting it all together, the change in ϕ is $\Delta\phi = 1 - k$. Thus, we've paid $O(k + \Delta\phi) = O(1)$ space and $O(k + \Delta\phi + 1) = O(1)$ time!

What about non-tree data structures? Well, they may require more than one modification box. The limiting factor is the in-degree of a node: how many other nodes can point at it. If the in-degree of a node is k, then we must use k extra modification boxes to get O(1) space and time cost.

Persistent hash array mapped trie

A persistent hash array mapped trie is a specialized variant of a hash array mapped trie that will preserve previous versions of itself on any updates. It is often used to implement a general purpose persistent map data structure.[10]

Hash array mapped tries were originally described in a 2001 paper by Phil Bagwell entitled "Ideal Hash Trees". This paper presented a mutable Hash table where "Insert, search and delete times are small and constant, independent of key set size, operations are O(1). Small worst-case times for insert, search and removal operations can be guaranteed and misses cost less than successful searches".[11] This data structure was then modified by Rich Hickey to be fully persistent for use in the Clojure programming language.[12]

Conceptually, hash array mapped tries work similar to any generic tree in that they store nodes hierarchically and retrieve them by following a path down to a particular element. The key difference is that Hash Array Mapped Tries first use a hash function to transform their lookup key into a (usually 32 or 64 bit) integer. The path down the tree is then determined by using slices of the binary representation of that integer to index into a sparse array at each level of the tree. The leaf nodes of the tree behave similar to the buckets used to construct hash tables and may or may not contain multiple candidates depending on hash collisions.[10]

Most implementations of persistent hash array mapped tries use a branching factor of 32 in their implementation. This means that in practice while insertions, deletions, and lookups into a persistent hash array mapped trie have a computational complexity of $O(\log n)$, for most applications they are effectively constant time, as it would require an extremely large number of entries to make any operation take more than a dozen steps.[13]

Making ADTs (e.g. Stack, Queue …) persistent:

We may use the Persistent Pointer Based Models to implement the ADT Operations. *But, out team has developed some efficient methods specific to the ADTs.*

Applications

1. In addition to the obvious 'look-back'applications, we can use persistent data structures to solve problems by representing one of their dimensions as time.
2. To achieve Immutability we may use Persistent Data structure. In several languages, in immutability perspective several versions versions of that imm. data structure is generated rather hampering the same version.
3. Multithreading is much easier!! Because if different threads try to modify a same unlocked data-structure, the threads will independently modify the data-structure and will create different versions. Race Problem won't occur!!

4. Optimistic UI:

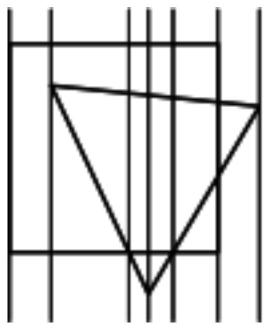
this point is discussed in the slide named "Application and Other Issues"

5. Undo and Browsing Web History: We may use persistent data-structure to go back to the past history, and, proceed from the past.

3. Once example is the computational geometry problem of planar point location. Given a plane with various polygons or lines which break the area up into a number of regions, in which region is a query point is located?

In one dimension, the linear point location problem can be solved with a splay tree or a binary tree that simply searches for the two objects on either side of the query point.

To solve the problem in two dimensions, break the plane into vertical slices at each vertex or point where lines cross. These slices are interesting because crossovers don't happen inside slices: inside each slice, the dividing lines between regions appear in a fixed order, so the problem reduces to the


Breaking the plane into slices for planar point location

linear case and requires a binary search (plus a bit of linear algebra). Figure 2.1 shows an example of how these slices look. To locate a point, first find the vertical slice it is in with a search on the point's x coordinate, and then, within that slice, find the region it is in with a search on the point's y coordinate (plus algebra). To do two binary searches takes only O(log n) time, so we can locate a point in O(log n) time. However, setting up the trees for searching a figure with n vertices will require n different trees, taking $O(n^2 \log n)$ time and $O(n^2)$ space to do the preprocessing.

Notice that between two adjacent slices of the picture there will only be one change. If we treat the horizontal direction as a timeline and use a persistent data structure, we can find the horizontal location of the point as a 'version' of the vertical point location data structure. In this way, we can preserve the O(log n) query time and use only O(n) space and O(n log n) preprocessing time.

Usage in programming languages

Haskell

Haskell is a pure functional language and therefore does not allow for mutation. Therefore, all data structures in the language are persistent, as it is impossible to not preserve the previous state of a data structure with functional semantics.[14] This is because any change to a data structure that would render previous versions of a data structure invalid would violate referential transparency.

In its standard library Haskell has efficient persistent implementations for linked lists,[15] Maps (implemented as size balanced trees),[16] and Sets[17] among others.[18]

Clojure

Like many programming languages in the Lisp family, Clojure contains an implementation of a linked list, but unlike other dialects its implementation of a Linked List has enforced persistence instead of being persistent by convention. [19] Clojure also has efficient implementations of persistent vectors, maps, and sets based on persistent hash array mapped tries. These data structures implement the mandatory read-only parts of the Java collections framework. [20]

The designers of the Clojure language advocate the use of persistent data structures over mutable data structures because they have value semantics which gives the benefit of making them freely shareable between threads with cheap aliases, easy to fabricate, and language independent.[21]

These data structures form the basis of Clojure's support for parallel computing since they allow for easy retries of operations to sidestep data races and atomic compare and swap semantics.[22]

Elm

The Elm programming language is purely functional like Haskell, which makes all of its data structures persistent by necessity. It contains persistent implementations of linked lists as well as persistent arrays, dictionaries, and sets.[23]

Elm uses a custom virtual DOM implementation that takes advantage of the persistent nature of Elm data. As of 2016 it was reported by the developers of Elm that this virtual DOM allows the Elm language to render HTML faster than the popular JavaScript frameworks React, Ember, and Angular.[24]

Java

The Java programming language is not particularly functional. Despite this, the core JDK package java.util.concurrent includes CopyOnWriteArrayList and CopyOnWriteArraySet which are persistent structures, implemented using copy-on-write techniques. The usual concurrent map implementation in Java, ConcurrentHashMap, is not persistent, however. Fully persistent collections are available in third-party libraries, or other JVM languages.

Prolog

Prolog terms are naturally immutable and therefore data structures are typically persistent data structures. Their performance depends on sharing and garbage collection offered by the Prolog system.[32] Extensions to non-ground Prolog terms are not always feasible because of search space explosion. Delayed goals might mitigate the problem.

Some Prolog systems nevertheless do provide destructive operations like setarg/3, which might come in different flavors, with/without copying and with/without backtracking of the state change. There are cases where setarg/3 is used to the good of providing a new declarative layer, like a constraint solver.[33]

Scala

The Scala programming language promotes the use of persistent data structures for implementing programs using "Object-Functional Style".[34] Scala contains implementations of many Persistent data structures including Linked Lists, Red–black trees, as well as persistent hash array mapped tries as introduced in Clojure.[35]

Major Problem In Persistent Data Structure: Garbage collection

Because persistent data structures are often implemented in such a way that successive versions of a data structure share underlying memory[36] ergonomic use of such data structures generally requires some form of automatic garbage collection system such as reference counting or mark and sweep.[37] In some platforms where persistent data structures are used it is an option to not use garbage collection which, while doing so can lead to memory leaks, can in some cases have a positive impact on the overall performance of an application. [38] .

Our team has discussed a possible way to manage this problem in the slide named "Application and Other Issues" using LRU Cache.