Stratified B-trees and versioning dictionaries.

Andy Twigg*, Andrew Byde*, Grzegorz Miłoś*, Tim Moreton*, John Wilkes†* and Tom Wilkie*
*Acunu, †Google
firstname@acunu.com

Abstract

A classic versioned data structure in storage and computer science is the copy-on-write (CoW) B-tree – it underlies many of today's file systems and databases, including WAFL, ZFS, Btrfs and more. Unfortunately, it doesn't inherit the B-tree's optimality properties; it has poor space utilization, cannot offer fast updates, and relies on random IO to scale. Yet, nothing better has been developed since. We describe the 'stratified B-tree', which beats the CoW B-tree in every way. In particular, it is the first versioned dictionary to achieve optimal tradeoffs between space, query and update performance.

1 Introduction

The B-tree was presented in 1972 [1], and it survives because it has many desirable properties; in particular, it uses optimal space, and offers point queries in optimal $O(\log_B N)$ IOs[1]. In 1986, Driscoll et al. [7] presented the 'path-copying' technique to make pointer-based internal-memory data structures fully-versioned (fully-persistent). Applying this technique to the B-tree, the CoW B-tree was first deployed in the Episode File System in 1992. Since then, it has been the underlying data structure in many file systems and databases, for example WAFL [10], ZFS [4], Btrfs [8], and many more.

Unfortunately, the CoW B-tree does not share the same optimality properties as the B-tree; in particular, every update may require a walk down the tree (requiring random reads) and then writing out a new path, copying the previous blocks. Many file systems embed the CoW B-tree into an append-only log file system, in an attempt to make the writes sequential. In conjunction with the garbage cleaning needed for log file systems, this leads to large space blowups, inefficient caching, and poor performance. Until recently, no other data structure has been known that offers an optimal tradeoff between space, query and update performance.

This paper presents some recent results on new constructions for B-trees that go beyond copy-on-write, that we call 'stratified B-trees'. They solve two open problems: Firstly. they offer a fully-versioned B-tree with optimal space and the same lookup time as the CoW B-tree. Secondly, they are the first to offer other points on the Pareto optimal query/update tradeoff curve, and in particular, our structures offer fully-versioned updates in $o(1)$ IOs, while using linear space. Experimental results indicate 100,000s updates/s on a large SATA disk, two orders of magnitude faster than a CoW B-tree.

Since stratified B-trees subsume CoW B-trees (and indeed all other known versioned external-memory dictionaries), we believe there is no longer a good reason to use the latter for versioned data stores. Acunu is developing a commercial in-kernel implementation of stratified B-trees, which we hope to release soon.

[1]We use the standard notation B to denote the block size, and N the total number of elements inserted

2 Versioned dictionaries

A versioned dictionary stores keys and their values with an associated version tree, and supports the following operations:

- `update(key, value, version)`: associate a value to the key in the specified leaf version
- `range_query(start, end, version)`: return every key in the range [start,end] together with the value written in the closest ancestor to the specified version
- `clone(version)`: create a new version as a child of the specified version[2]
- `delete(version)`: delete a given version, and free the space used by all keys written there and no longer accessible by any version.

A versioned dictionary can be thought of as efficiently implementing the union of many dictionaries: the *live* keys at version v are the union of all the keys in ancestor versions, where if a key appears more than once, its closest ancestor takes precedence. If the structure supports arbitrary version trees, then we call it (fully-)versioned;

[2]Sometimes the literature refers to a 'snapshot' operation – here, a snapshot is exactly equal to cloning a leaf node.