Persistent Data Structures (Version Control)

Ephemeral

Partial persistence

Full persistence

Confluently persistent

Purely functional

Retroactive

version list

version tree

version DAG

version

query only

updates at leaves
any version can be copied
query all versions

update/merge/query all versions

never modify
only create new pairs
only DAGs

update & query all versions

update & query all versions

updates in the past propagate

car cdr

create new pairs

create DAGs

Retroactive
Planar Point Location

Partial persistent search trees

$O(n \cdot \log n)$ preprocessing, $O(\log n)$ query
Path Copying (trees)
Partial Persistence

- **Version ID = time = 0, 1, 2,...**

- **Fat node (any data structure)**
  - record all updates in node (version, value) pairs
  - field updates $O(1)$
  - field queries $\equiv$ predecessor wrt version id (search tree/vEB)

- **Node copying ($O(1)$ degree data structures)**
  - Persistent node = collection of nodes, each valid for an interval of versions, with $\Delta$ extra updates, $\Delta = \max$ indegree
  - pointers must have subinterval of the node pointing to; otherwise copy and insert pointers (cascading copying)
  - NB: Needs to keep track of back-pointers

```
[0, 8[
field_1: (0, x) (3, y)
field_2: (0, a) (7, c)

[8, 13[
field_1: (8, z) (10, w)
field_2: (8, c) (9, d)

[13, \infty[
field_1: (13, w) (q5, y)
field_2: (13, e) (14, c)
```
Fat node
- Updates (1,x) (6,y) (7,z) to a field
- Queries = binary search among versions
- Update (7,z): Insert 7 as leftmost child of 4; insert pairs for 7 and 5 = succ(7)

Node splitting \( (\geq 2\Delta \text{ ekstra fields}) \)

Version tree
(numbers = version ids)

increasing version

preorder traversal

Version list
(order maintenance data structure)

field: \( (1,x) (7,z) (5,x) (6,y) (2,x) \)

\[ [0, \infty[ \]

\[ [4,3[ \]

\[ [0,5[ \]

\[ [4,5[ \]

\[ [5,\infty[ \]

field\(_1\): (1,a) (4,b) (3,a) (2,c)
field\(_2\): (1,f) (7,g) (5,f)

field\(_1\): (1,a) (4,b)
field\(_2\): (1,f) (7,g)
field\(_2\): (5,f)

field\(_1\): (5,b) (3,a) (2,c)

split

version 5

split
Persistence Techniques


- Partial persistence, trees, $O(1)$ access, amortized $O(1)$ update


- Partial & full persistence, $O(1)$ degree data structures, $O(1)$ access, amortized $O(1)$ update


- Partial persistence, $O(1)$ degree data structures, $O(1)$ access & updates update


- Full persistence, RAM structures, $O(\log\log n)$ access, $O(\log\log n)$ amortized expected updates
Comparison of Persistence Techniques

- Copy data structure for each version
  - no query overhead, slow updates & wastes a lot of space

- Record updates & keep current version
  - fast updates & queries to current version, space efficient, slow queries in the past

- Path copying
  - applies to trees, no query overhead, space overhead = depth of update

- Fat node
  - partial persistence: $O(1)$ updates and space optimal, loglog $n$ query overhead
  - full persistence: $O(\log \log n)$ expected amortized updates and space optimal, loglog $n$ query overhead

- Node copying/splitting
  - fast updates & queries (amortized updates for full persistence)
  - only works for pointer-based structures with $O(1)$ degree
Fractional Cascading

- Basic Idea : \(2 \times \text{BinSearch} \Rightarrow 1 \text{ BinSearch} + O(1)\)

- Build bridges (and pointers to nearest original element)
- Searches to next list : Traverse nearest bridge
- Construction : Repeatedly create bridges until all gaps \(O(1)\)

- Generalizes to catalog graphs of degree \(O(1)\)
Fractional Cascading Techniques


- Static fractional cascading, $O(1)$ worst-case access


- Dynamic fractional cascading, $O(\log \log N)$ worst-case access, amortized insert and delete
- Insertion or deletion only, $O(1)$ per worst-case access, amortized insert or delete