(Partially) Persistent Streaming Indexes

Andrew Twigg

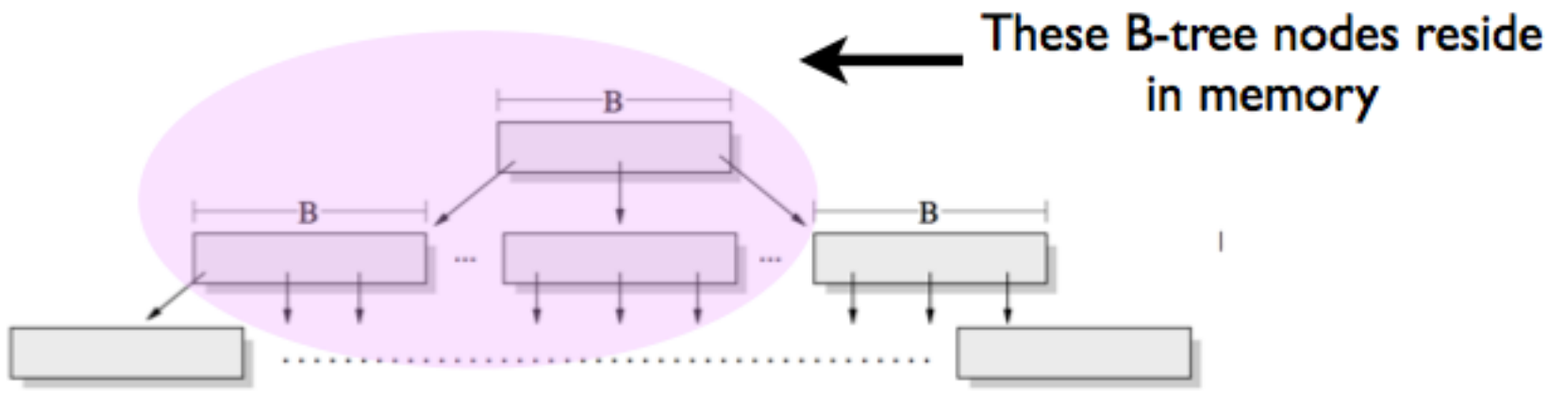
Oxford University
Computing Laboratory
UK

outline

- motivation
- related work: COLA
- persistent operations
- our data structure: partial persistence
- operations: update, query
- analysis

Motivation

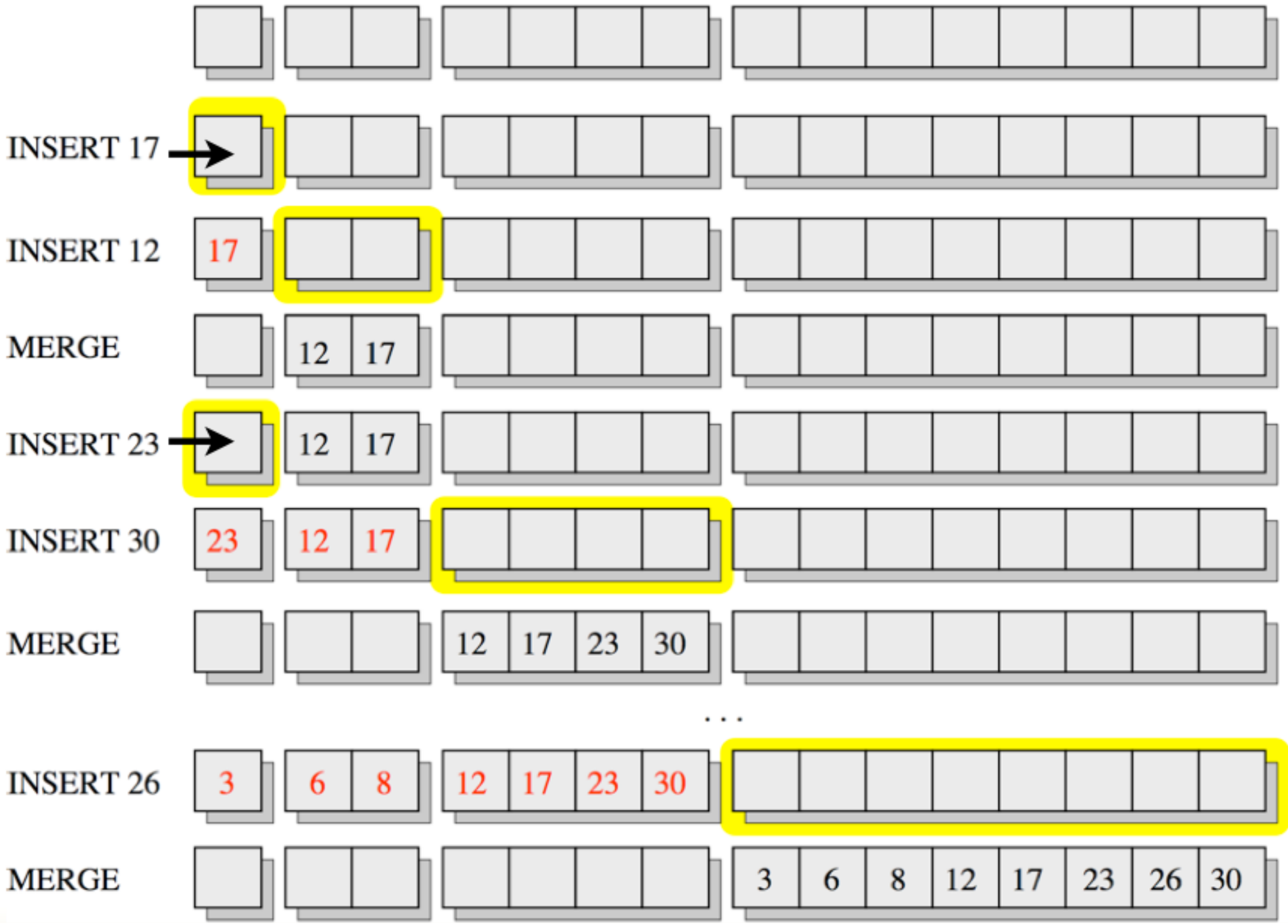
- [Bender et al]: B-trees are slow at random inserts
  - at least 1 random IO per insert
  - DAM model implies disks need big blocks to amortize seek costs (~1MB)
  - want a structure that does large batched writes



[Picture credit: Michael Bender]

COLA

- Cache-Oblivious Lookahead Array [Bender et al, SPAA 2007]
- Space O(N)
- Update: O(log N / B) amortized IO
- Query: O(log N + K/B) IO
  - Easy to get O(log^2 N) without fractional cascading
  - Logarithmic method; exponentially growing arrays



[Picture credit: Michael Bender]

Motivation

- [this work]: persistence is a core feature of file systems/ databases
- all known persistent B-trees have the above problem (CoW B-tree, MVBT, ..)
- We'd like a persistent analogue of the COLA

outline

- motivation
- related work: COLA
- persistent operations
- our data structure: partial persistence
- operations: update, query
- analysis

Persistent operations

- A (global) version tree V
- Each version v has a dictionary D_v: {k : x}
- `update`$_v$`(k,x):` create a new child v' of v with $D_{v'} = D_v + \{k : x\}$
- `query`$_v$`(`k_1`,`k_2`):` return {(k,x) in D_v : k in $[k_1,k_2]$}
- update only works on leaves: partially-persistent

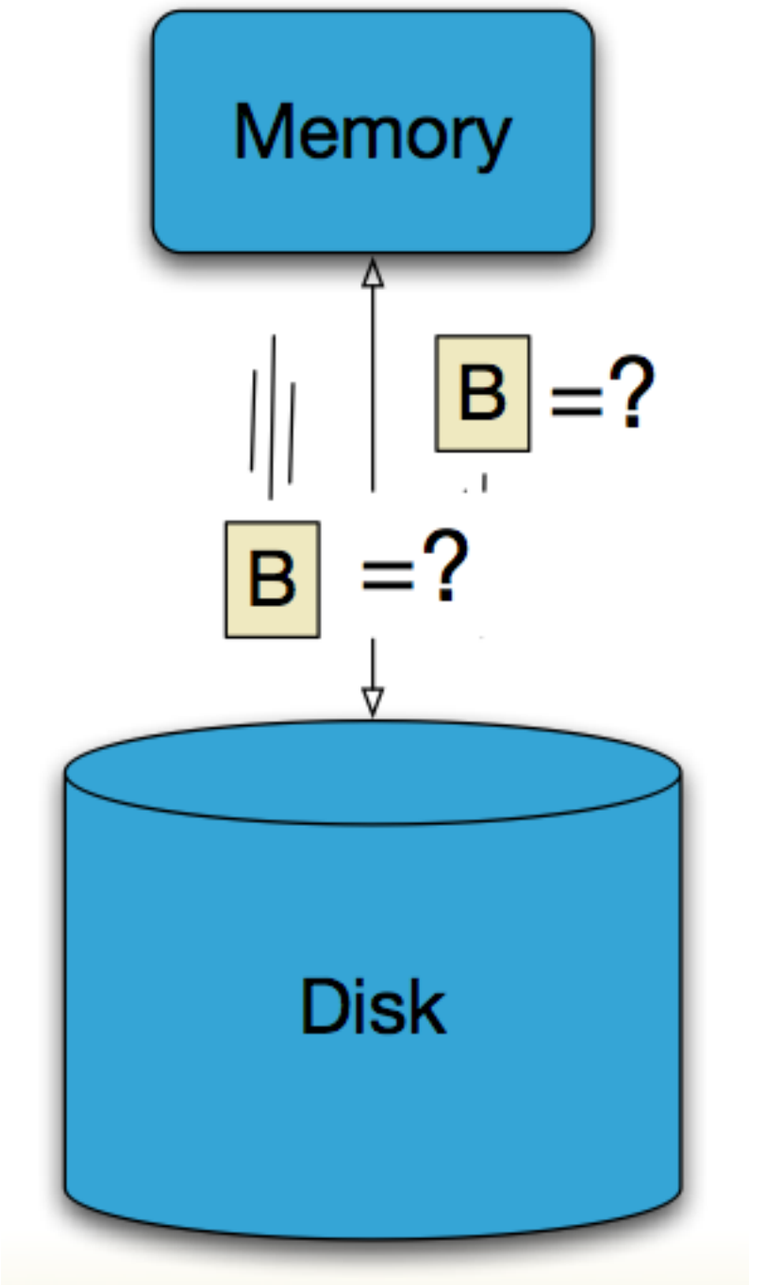
Our data structure

- similar to COLA except arrays have version sets: (A, W)
- more complex promotion, merging, etc.
- achieve amortized bounds in CO model
  - update O(log N_v / B) amortized IOs
  - query O(log N_v + K/B) IOs in *average-case*
  - space O(N)

$N_v = |D_v|$
i.e. number of keys 'live' at v

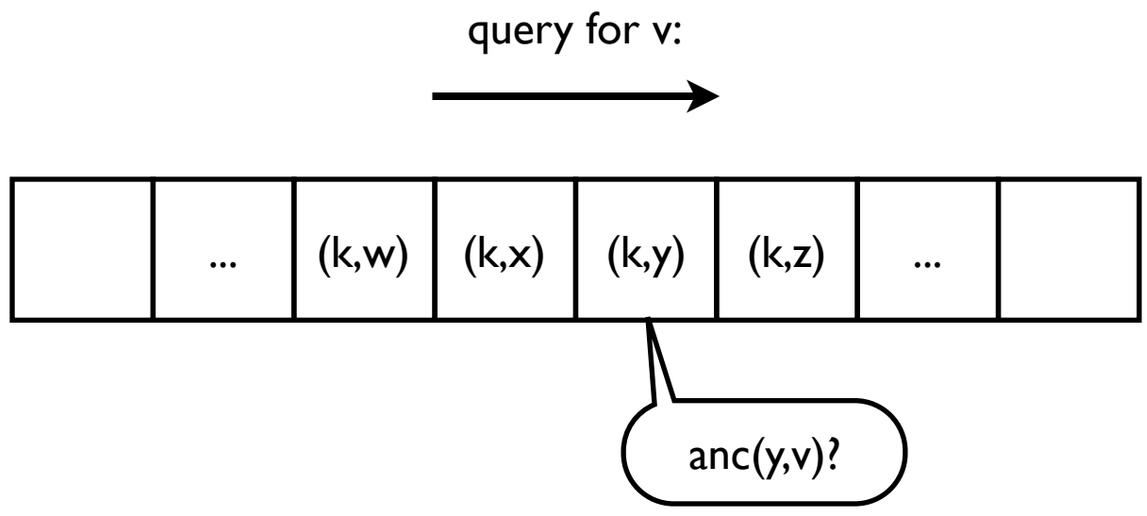
Model

- Disk access model (DAM) [Aggrawal,Vitter 88]
  - two levels of memory
  - block size B, memory size M - needed by the algorithm
- Cache-oblivious model (CO) [Frigo et al 99]
  - parameters B,M are unknown to the algorithm
  - => algorithm works well at *every* level of the memory hierarchy



versioned arrays

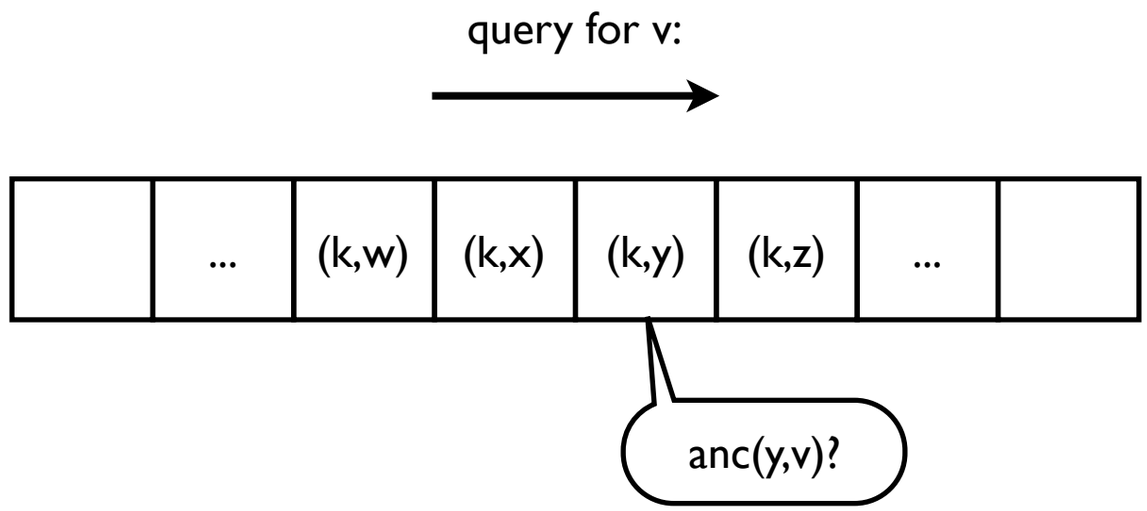
- Array (A,W) stores entries *live* at some w in W
- Elements ordered by (key, version)
- Versions ordered by decreasing DFS number in W
- `query`$_v$`(k1,k2)` at (A,W):
  - binary search for k1, start scanning to the right
  - for each k in [k1,k2], output the first element (k,y) where anc(y,v)



- might be inefficient if lots of ‘not-relevant’ elements

testing ancestorship

- We need to be able to test for ancestorship while scanning an array



- Store the interval I(w)=[DFS(w), $\max_{\{x:anc(w,x)\}}$ DFS(x)] in the array
- All we need to know is DFS(v) - easy for partially-persistent case

Live, lead, density

- (k,v) is *lead* at version w if v==w ("written at w")
  - lead(A,w)=# elements in A lead at w
  - *lead fraction* of (A,W) = $\Sigma_{w \text{ in } W}$ lead(A,w) / |A|
  - There are exactly n lead elements; each element is lead in exactly one array

Live, lead, density

- (k,v) is *lead* at version w if v==w ("written at w")
  - lead(A,w)=# elements in A lead at w
  - *lead fraction* of (A,W) = $\Sigma_{w \text{ in } W}$ lead(A,w) / |A|
  - There are exactly n lead elements; each element is lead in exactly one array
- (k,v) is *live* at version w if (k,x) in D_w ("accessible from some query at w")
  - live(A,w) = # elements in A live at w
  - note that live(.) increases down the version tree
  - *density* of (A,W) = $\min_{w \text{ in } W}$ live(A,w) / |A| for all w in W

Live, lead, density

- (k,v) is *lead* at version w if v==w ("written at w")
  - lead(A,w)=# elements in A lead at w
  - *lead fraction* of (A,W) = $\Sigma_{w \text{ in } W}$ lead(A,w) / |A|
  - There are exactly n lead elements; each element is lead in exactly one array
- (k,v) is *live* at version w if (k,x) in D_w ("accessible from some query at w")
  - live(A,w) = # elements in A live at w
  - note that live(.) increases down the version tree
  - *density* of (A,W) = $\min_{w \text{ in } W}$ live(A,w) / |A| for all w in W
- Why density?
  - Density "too low" ⇒ scanning [(k1,*), (k2,*)] in an array examines mostly useless (non-live) stuff
  - Density "too high" ⇒ elements are replicated too many times => high space

Live, lead, density

- (k,v) is *lead* at version w if v==w ("written at w")
  - lead(A,w)=# elements in A lead at w
  - *lead fraction* of (A,W) = $\Sigma_{w \text{ in } W}$ lead(A,w) / |A|
  - There are exactly n lead elements; each element is lead in exactly one array
- (k,v) is *live* at version w if (k,x) in D_w ("accessible from some query at w")
  - live(A,w) = # elements in A live at w
  - note that live(.) increases down the version tree
  - *density* of (A,W) = $\min_{w \text{ in } W}$ live(A,w) / |A| for all w in W
- Why density?
  - Density "too low" ⇒ scanning [(k1,*), (k2,*)] in an array examines mostly useless (non-live) stuff
  - Density "too high" ⇒ elements are replicated too many times => high space
  - Want, for all arrays: density Θ(1) and lead fraction Ω(1) (goldilocks)
  - We'll show density ≥ 1/6 and lead fraction ≥ 1/3 for almost all arrays

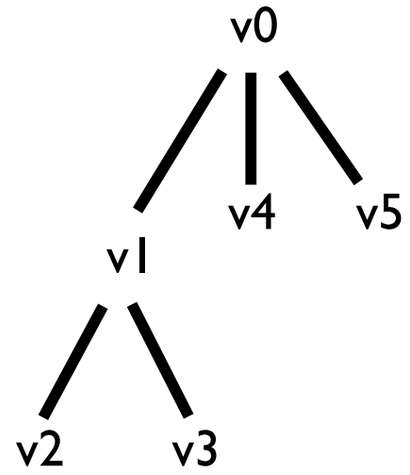
live but not lead

keys

versions

	k0	k1	k2	k3
v0				
v4				
v5				
v1				
v2				
v3				

lead



W={v1,v2,v3}	k0, v0, x	k1, v0, x	k1, v2, x	k2, v1, x	k2, v2, x	k2, v3, x	k3, v1, x	k3, v2, x

Arrays in levels

- Arrays divided into levels; level l array satisifes

1. $|A| \leq 2^{l+1}$
2. live(A,w) $\geq 2^{l}/3$ for all w in W (=> dense)
3. W is a connected subtree of V

- in each level, all sets W are disjoint ⇒ queries need to examine one array

outline

- motivation
- related work: COLA
- persistent operations
- our data structure: partial persistence
- operations: update, query
- analysis

Query

```
query_v(k1,k2):

     for each level l:

        let (A_l, W_l) be the array with v in W_l

        binary search for k1 in A_l

        let S_l = scan to the right, for each key k in
        [k1,k2] output the first element (k,y) where anc(y,v)

return S = merge(S_1...S_l), but for each key k only
output the closest ancestor to v
```

Each array (A,W) identified by [min(W),max(W)] so use a separate COLA per level

versions from different arrays easy to compare if we use global numbering, i.e. DFS(V)

Update

```
update(k,v,x):
        promote({(k,v+1,x)}, {v+1}, 0)

promote(A,W,l):
        let (B,Y) be the array where Y contains the closest ancestor to min(W)
        merge to get (A',W')=(A+B, W+Y) = (A+B, [min(Y),max(W)])
        register (A',W') at level l

                    l+1
if |A'| > 2     then
    let (A'',W'') = extract_promotable(A',W',l)
    promote(A'',W'',l+1)
    update (A',W') := (A',W') \ (A',W'')

if density(A',W') < 1/6 then
    let {(A_1,W_1)...(A_k,W_k)} = subdivide(A',W',l)
    for each i: promote(A_i,W_i,l) // only to register them
    deregister (A',W') at level l
```

Update

```
update(k,v,x):
        promote({(k,v+1,x)}, {v+1}, 0)

promote(A,W,l):
        let (B,Y) be the array where Y contains the closest ancestor to min(W)
        merge to get (A',W')=(A+B, W+Y) = (A+B, [min(Y),max(W)])
        register (A',W') at level l

                                  l+1
        if |A'| > 2                  then
            let (A'',W'') = extract_promotable(A',W',l)
            promote(A'',W'',l+1)
            update (A',W') := (A',W') \ (A',W'')

        if density(A',W') < 1/6 then
            let {(A_1,W_1)...(A_k,W_k)} = subdivide(A',W',l)
            for each i: promote(A_i,W_i,l) // only to register them
            deregister (A',W') at level l
```

Let v be the highest version v where `live(A,v)` > $2^{l+1}/3$ and `S(v)` > 2^{l+1}.
Return all live elements at any of v's descendants

Greedily subdivide (A',W') into output arrays, each with density ≥ 1/6 and all but one with lead fraction ≥ 1/3
May create copies of elements!

Greedy subdivide

Starting from v_i = root(W)

Walk down the tree W until we find the first version v_j with child v_{j+1} having S(v_{j+1}) < 2^{l+1}

Output the array (S(v_{j+1}),W[v_{j+1}]), and start again on the remainder W \ W [v_{j+1}]

`S(v)` = elements live at any of v's descendants

`W[v]` = subtree of W rooted at v

Greedy subdivide

`S(v)` = elements live at any of v's descendants

Starting from v_i = root(W)

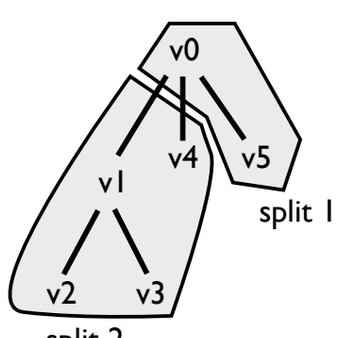
Walk down the tree W until we find the first version v_j with child v_{j+1} having S(v_{j+1}) < 2^{l+1}

Output the array (S(v_{j+1}),W[v_{j+1}]), and start again on the remainder W \ W [v_{j+1}]

`W[v]` = subtree of W rooted at v

live(v0) = 2
density = 2/13

	k0	k1	k2	k3
v0	■	■		
v4	■	■		
v5			■	■
v1	■	■	■	
v2	■			
v3		■		



split 1

	k0	k1	k2	k3
v0	■	■		
v4				
v5	■	■	■	■
v1				
v2				
v3				

live(v0) = 2
live(v5) = 4

density = 2/6

split 2

	k0	k1	k2	k3
v0				
v4	■	■		
v5				
v1	■	■	■	
v2	■			
v3		■		

live(v4) = 2
live(v1) = 3
live(v2) = 3
live(v3) = 3

density = 2/7

outline

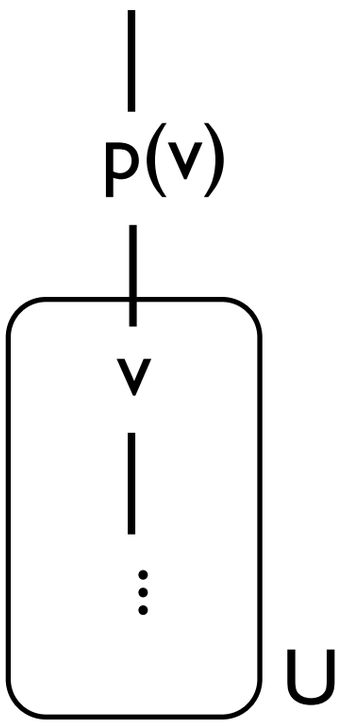
- motivation
- related work: COLA
- persistent operations
- our data structure: partial persistence
- operations: update, query
- analysis

Analysis

LEMMA 1 (PROMOTION). *Consider an array (A, W) promoted from level l to $l+1$. It satisfies (1) $\text{live}(A, v) \geq 2^{l+1}/3$ for all $v \in W$; (2) at least $1/3$ of the elements are lead elements; (3) the array is dense.*

Analysis

LEMMA 1 (PROMOTION). *Consider an array (A, W) promoted from level l to $l+1$. It satisfies (1) $\text{live}(A, v) \geq 2^{l+1}/3$ for all $v \in W$; (2) at least $1/3$ of the elements are lead elements; (3) the array is dense.*



Let v be the highest version v where
`live(A,v)` > $2^{l+1}/3$ and `S(v)` > 2^{l+1}.

lead(U) = S(p(v)) - live(p(v))
≥ S(v) - live(p(v))
≥ (2/3) 2^{l+1}

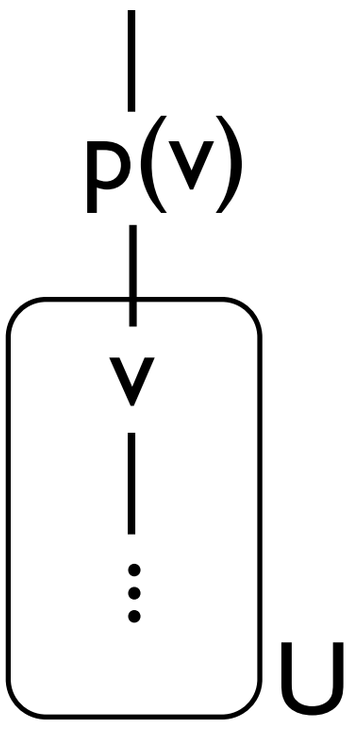
and the array has size ≤ 2^{l+2}

Analysis

LEMMA 2 (SUBDIVISION). *Consider the remainder* (A, W) *after extracting all subarrays* (A', W') *having* $|A'| \geq 2^{l+1}$ *and* $\text{live}(A', v) \geq 2^{l+1}/3$ *for all* $v \in W'$. *Algorithm 1 outputs arrays* (A_i, W_i) *where (1)* $|A_i| < 2^{L+1}$; *(2)* $lead(A_i) \geq \frac{|A_i|}{3}$ *for all but at most one array; (3) all arrays are dense, i.e.* $\delta(A_i, W_i) \geq 1/6$;

Analysis

LEMMA 2 (SUBDIVISION). *Consider the remainder* (A, W) *after extracting all subarrays* (A', W') *having* $|A'| \geq 2^{l+1}$ *and* $\text{live}(A', v) \geq 2^{l+1}/3$ *for all* $v \in W'$. *Algorithm 1 outputs arrays* (A_i, W_i) *where (1)* $|A_i| < 2^{L+1}$; *(2)* $lead(A_i) \geq \frac{|A_i|}{3}$ *for all but at most one array; (3) all arrays are dense, i.e.* $\delta(A_i, W_i) \geq 1/6$;



For every v, either $|S(v)|<2^{L+1}$ or $live(v)<2^{(L+1)}/3$
Let v be the first point where $|S(v)|<2^{L+1}$.
Then:
(1) $|S(p(v))| \geq 2^{L+1}$ and $live(p(v)) < 2^{(L+1)}/3$
(2) $lead(U) = |S(p(v))| - live(p(v))$
$> (2/3)2^{L+1}$
(3) $|S(v)| \leq 2^{L+1}$ so lead fraction $\geq 2/3$

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- COLA argument:
  - each array involved in merge has size > B
  - so every merge costs O(1/B) per element involved
  - each entry exists in one array and merged O(log n) times

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- COLA argument:
  - each array involved in merge has size > B
  - so every merge costs O(1/B) per element involved
  - each entry exists in one array and merged O(log n) times

Here: entries may be in many arrays (live), and may be merged many times at the same level (subdivide and leave at level l)

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- COLA argument:
  - each array involved in merge has size > B
  - so every merge costs O(1/B) per element involved
  - each entry exists in one array and merged O(log n) times

Here: entries may be in many arrays (live), and may be merged many times at the same level (subdivide and leave at level l)

- idea: charge the cost of merges to *promoted lead* elements only
  - every element (k,v) is lead in exactly one array (N lead elements)
  - *promoted lead* means the element is charged at most once per level (when promoted)

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- consider a merge triggered by promotion of (A,W)
- charge entire cost to lead(A,W); when (A,W) promoted, give $c/B to each lead element

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- consider a merge triggered by promotion of (A,W)
- charge entire cost to lead(A,W); when (A,W) promoted, give \$c/B to each lead element
- **subdivision lemma** ⇒ every subdivided output array (except one) has lead(A_i,W_i)>1/3
  - so total output size ≤ sum_i 3*lead(A_i,W_i) + 2^{l+1} ≤ 3*lead(A,W) + 2^{l+1} ≤ 4*2^{l+1}

single array with no lead guarantee

each lead in the input is lead in exactly one output array

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- consider a merge triggered by promotion of (A,W)
- charge entire cost to lead(A,W); when (A,W) promoted, give \$c/B to each lead element
- **subdivision lemma** ⇒ every subdivided output array (except one) has lead(A_i,W_i)>1/3
  - so total output size ≤ sum_i 3*lead(A_i,W_i) + 2^{l+1} ≤ 3*lead(A,W) + 2^{l+1} ≤ 4*2^{l+1}
- **promotion lemma** ⇒ at least (1/3)*2^{l+1} promoted lead elements
  - so c > 24 (say) works (ignoring IOs to read/merge etc.)

Update bound

THEOREM 1. *The operation* $\texttt{update}_v(\cdot)$ *costs amortized* $O((\log N \log N_v)/B)$ *cache-oblivious IOs.*

- consider a merge triggered by promotion of (A,W) into level l
- charge entire cost to lead(A,W); when (A,W) promoted, give \$c/B to each lead element
- **subdivision lemma** ⇒ every subdivided output array (except one) has lead(A_i,W_i)>1/3
  - so total output size ≤ sum_i 3*lead(A_i,W_i) + 2^{l+1} ≤ 6*lead(A,W) + 2^{l+1} ≤ 2^{l+4}
- **promotion lemma** ⇒ at least (2/3)*2^l promoted lead elements
  - so c > 24 (say) works (ignoring IOs to read/merge etc.)
- **density** ⇒ any (A,W) with v in W exists in level O(log N_v)
  - so each input element charged O(c (log N_v) / B) total

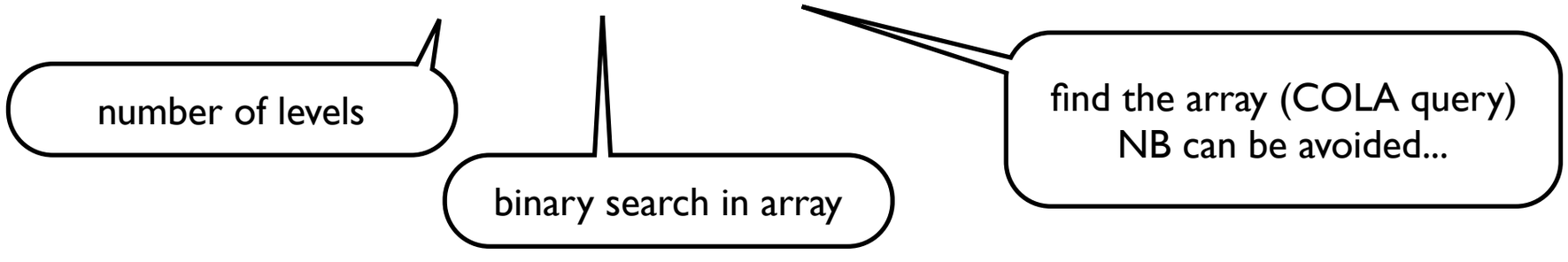
Query bound

LEMMA 3. *For an array A, and a query at version v, the average number of IOs per element returned from A is $O(1/B)$, where the average is taken over all queries that return disjoint sets of keys from A that are live at v.*

Query bound

LEMMA 3. *For an array A, and a query at version v, the average number of IOs per element returned from A is $O(1/B)$, where the average is taken over all queries that return disjoint sets of keys from A that are live at v.*

- density implies voluminous queries (return some fraction of the input keys) are efficient
- for smaller queries: two key-disjoint queries (for same version v) touch disjoint parts of the array. But density only applies to the whole array, so a particular part may be 'sparse'
- summing over all such disjoint ranges, we scan the array once in total, so density means the average cost attributable to each returned key is O(1/B)
- Gives total query bound $O(\log N_v \log N_v \log N + K/B)$



Space bound

- Previously showed: whenever an array with k lead elements is promoted, we use space O(k) after merge/subdivide
- By density, we have O(log N) levels
- Gives space O(N)

Future

- possible to get better query/update bounds
  - avoid searching in each level for the right array to examine
    - use lookahead pointers as in COLA
- *open*: maintain DFS numbers of a tree under O(log N / B) IOs/insert and O(log N) IOs/query ⇒ fully-persistent case...
- *open*: query/update tradeoffs (as in B^eps tree/X-dict)?
- *open*: amortized/worst-case range queries?