Key insights:

1. *Fingers allow constant-time access to oldest and youngest.*
2. *Specially defined aggregates shield sections of tree from each other.*
3. *Choice of min and max arity plus lazy splitting and merging avoids unnecessary tree rebalancing.*

Optimal and General Out-of-Order Sliding Window Aggregation

Kanat Tangwongsan[#], Martin Hirzel[+], Scott Schneider[+]

[#]*Mahidol University International College*
[+]*IBM T. J. Watson Research Center*

Problem and Motivation

Sliding window aggregation is a key component of streaming applications. Prior work established that efficient processing maintains *incremental* aggregations on *partial results*. Abstracting aggregations as *monoids* enables genericity.

Assuming *ordered* data, window operations take $O(1)$ time [1]. But *unordered* data is more common. What is the best we can achieve when the data is unordered?

[1] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. DEBS 2017.

Consider:

$\langle {}^{2.0}_{4} \rangle, \langle {}^{3.0}_{3} \rangle, \langle {}^{4.0}_{0} \rangle, \langle {}^{6.0}_{4} \rangle$
max: 4, maxcount: 2

$\langle {}^{2.0}_{4} \rangle, \langle {}^{3.0}_{3} \rangle, \langle {}^{4.0}_{0} \rangle, \langle {}^{6.0}_{4} \rangle, \langle {}^{\mathbf{6.5}}_{\mathbf{4}} \rangle$
max: 4, maxcount: **3**

$\langle {}^{2.0}_{4} \rangle, \langle {}^{\mathbf{2.3}}_{\mathbf{5}} \rangle, \langle {}^{3.0}_{3} \rangle, \langle {}^{4.0}_{0} \rangle, \langle {}^{6.0}_{4} \rangle, \langle {}^{6.5}_{4} \rangle$
max: **5**, maxcount: **1**

~~$\langle {}^{2.0}_{4} \rangle$~~, $\langle {}^{2.3}_{5} \rangle, \langle {}^{3.0}_{3} \rangle, \langle {}^{4.0}_{0} \rangle, \langle {}^{6.0}_{4} \rangle, \langle {}^{6.5}_{4} \rangle$
max: 5, maxcount: 1

~~$\langle {}^{2.3}_{5} \rangle$~~, $\langle {}^{3.0}_{3} \rangle, \langle {}^{4.0}_{0} \rangle, \langle {}^{6.0}_{4} \rangle, \langle {}^{6.5}_{4} \rangle$
max: **4**, maxcount: **2**

We define a data structure for sliding window aggregations with binary operator on a time-ordered window of $\langle {}^{t_1}_{v_1} \rangle, \ldots, \langle {}^{t_n}_{v_n} \rangle$, $t_i < t_{i+1}$ as:

`insert(t: Timestamp, v: Agg)` inserts $\langle {}^{t}_{v} \rangle$ into the window ordered by t; if t already exists, $\langle {}^{t}_{v \otimes v'} \rangle$

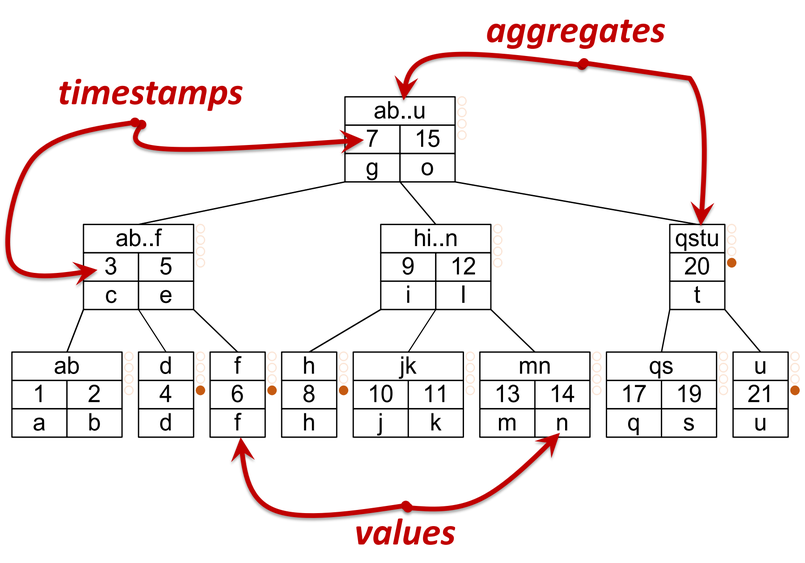
`evict(t: Timestamp)` removes $\langle {}^{t}_{v} \rangle$ from window

`query(): Agg` returns $v_1 \otimes \ldots \otimes v_n$

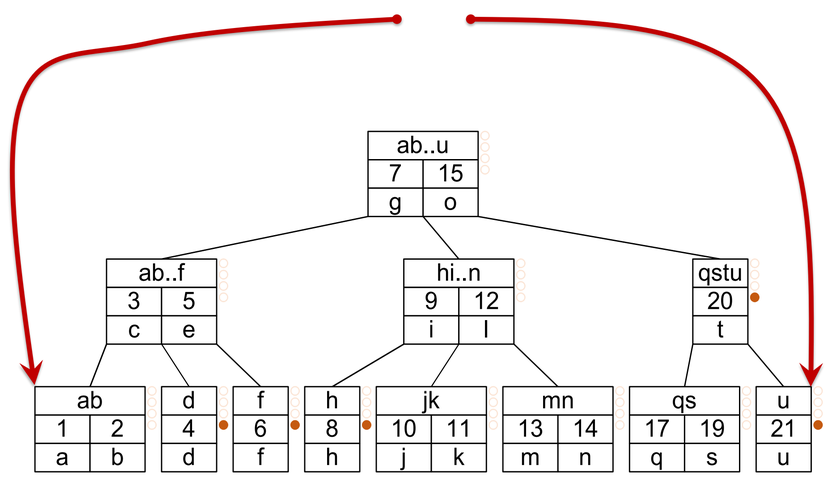
We implement the above in amortized $O(\log d)$ time, where d is the out-of-order distance. With ordered data, it reduces to amortized $O(1)$.

FiBA: Finger B-Tree Aggregator

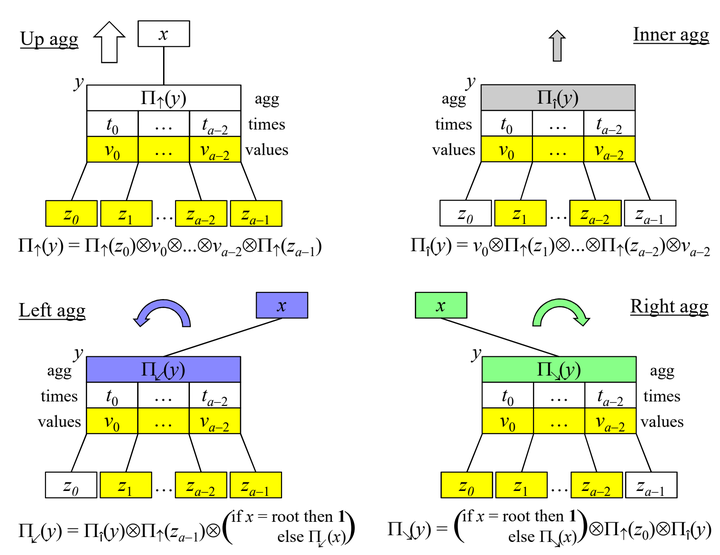
Start with a B-Tree modified with aggregates:



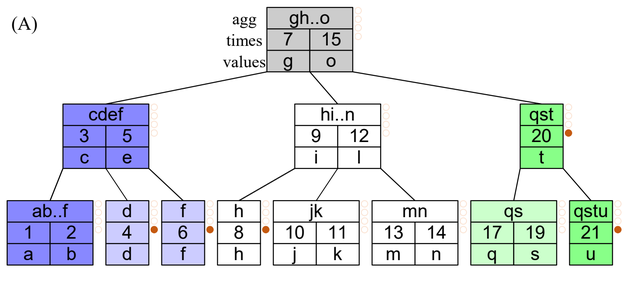
Add *fingers*:



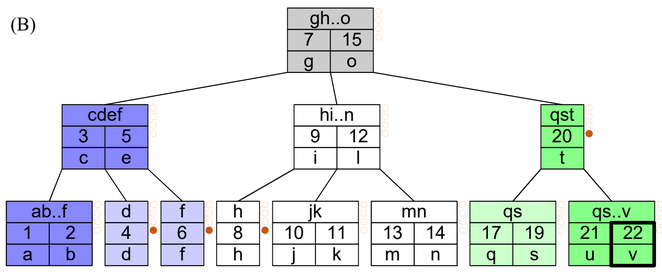
Define position-aware aggregates:



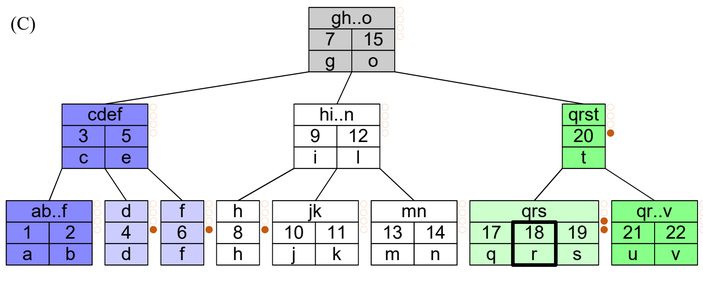
Examples:



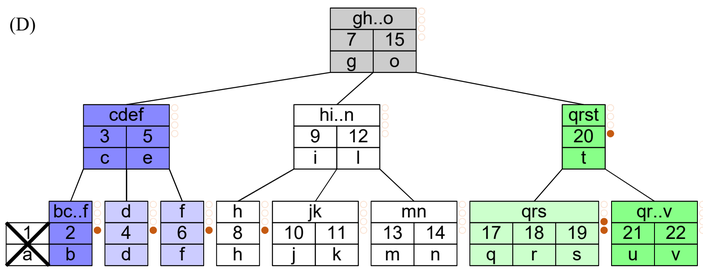
In-order insert:



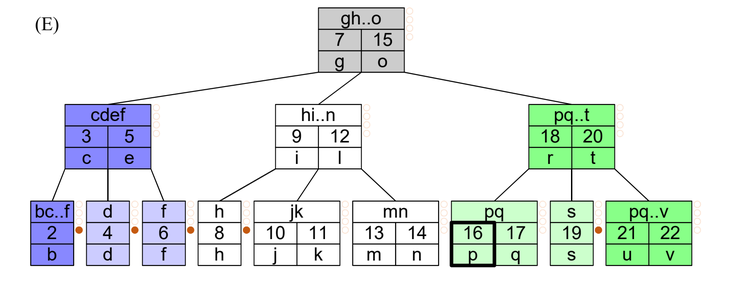
Out-of-order insert:



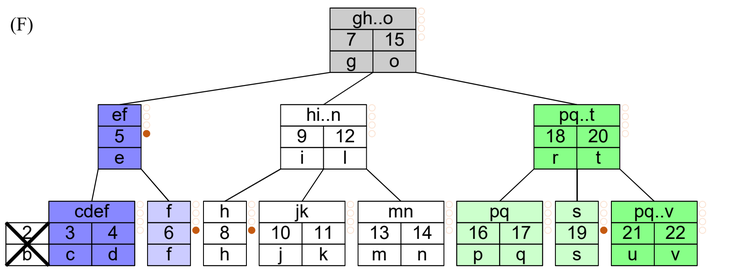
In-order evict:



Out-of-order insert (with node split):



In-order evict (with node merge):



Experimental Analysis