An Adaptive Index for Hierarchical Database Systems

Rafael Kallis

BSc Thesis

February 4, 2018

University of Zurich[UZH]

Department of Informatics

The Workload-Aware Property Index (WAPI):

- Detects frequently updated nodes
- Stops pruning such volatile nodes
- Significantly improves update throughput

Unproductive Nodes are an unwanted byproduct:

- When the workload changes, volatile nodes cease to be volatile
- They waste space and slow down queries
- They do not contribute to a query match and contain no data

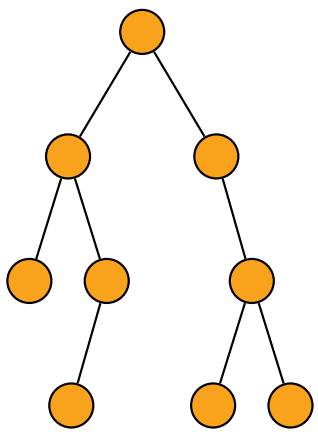
In this thesis we:

- Design and implement two solutions in order to mitigate unproductive nodes
- Analyze the factors impacting the production of unproductive nodes
- Empirically evaluate and compare our two solutions

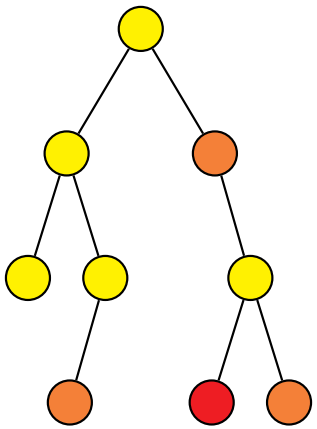
A Content Management System's (CMS) workload is:

- skewed
- update-heavy
- changing

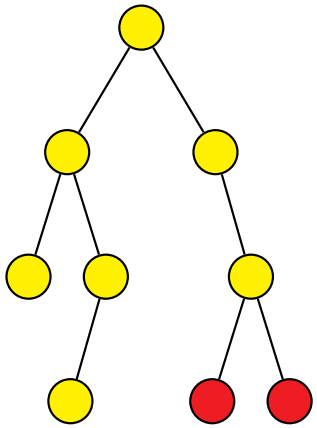
Skewed workload: small subset of nodes gets frequently updated



No skew

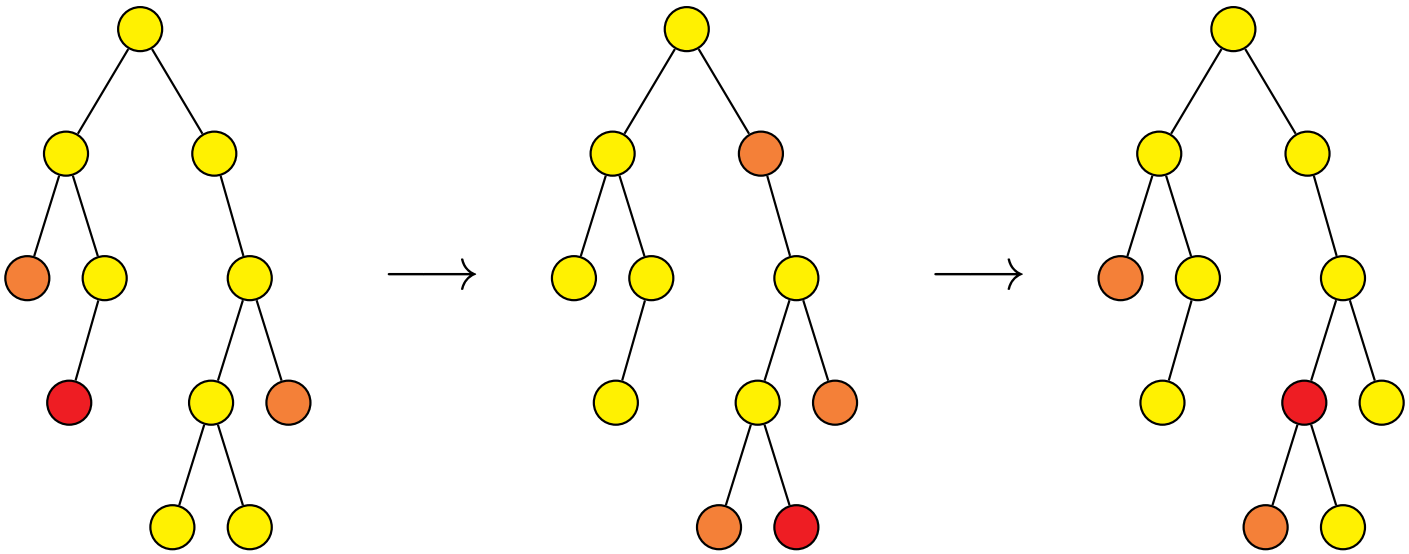


Normal skew



High skew

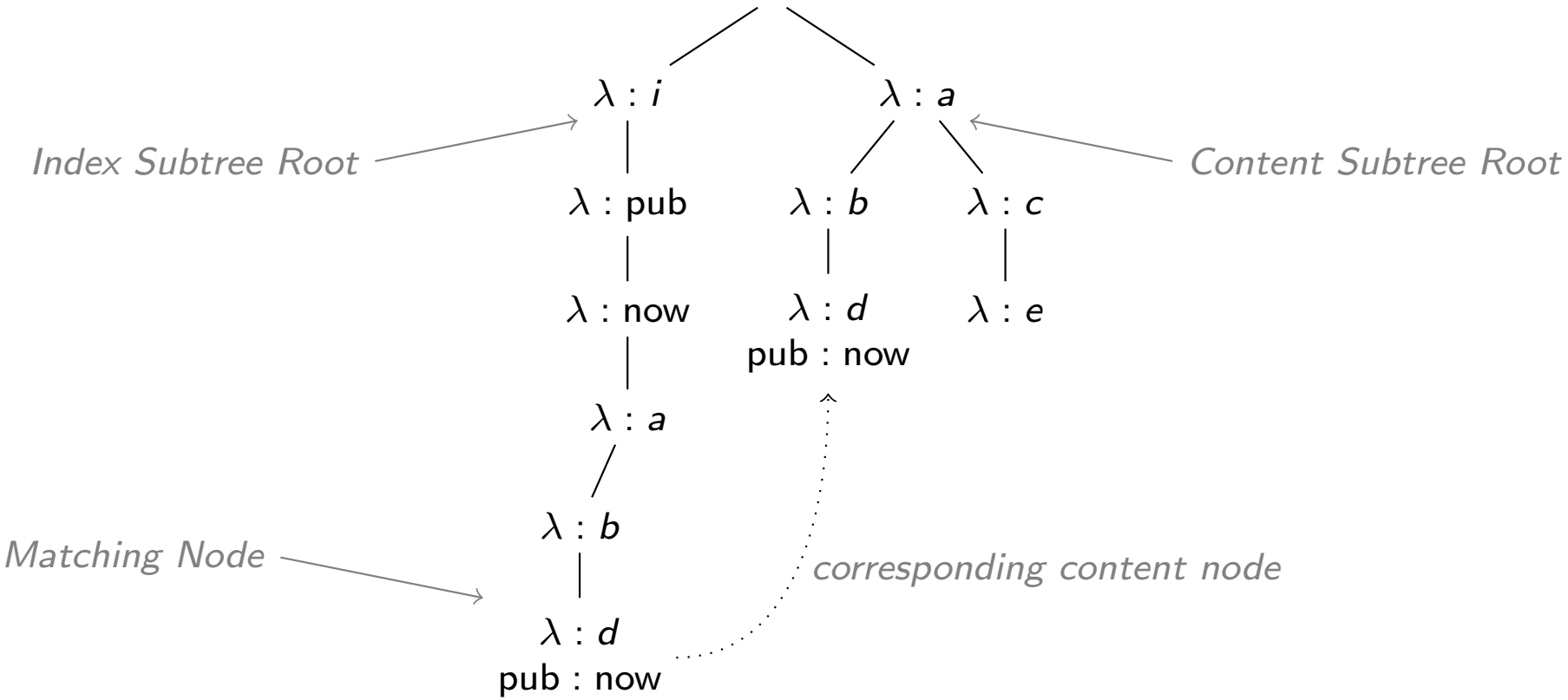
Changing workload: as time passes, hotspots change



CMSs usually use a job-queuing system that has the noted characteristics

Workload-Aware Property Index

Hierarchical Database with WAPI



We mostly executes content-and-structure (CAS) queries [1]. We denote node n's property k as $n[k]$ and node n's descendants as $desc(n)$.

Definition (CAS Query)

Given node m, property k and value v, a CAS query $Q(k, v, m)$ returns all descendants of m which have k set to v, i.e.,

$$Q(k, v, m) = \{n | n \in desc(m) \wedge n[k] = v\}$$

Volatility is the measure which is used by the WAPI in order to distinguish whether to remove a node or not from the index. Wellenzohn et al. [2] propose to look at the recent transactional workload to check whether a node n is volatile.

Definition (Volatility Count)

The volatility count $vol(n)$ of index node n on database instance O_i, is the number of times node n was added or removed from snapshots contained in a Sliding Window of Length L over history H_i, i.e.,

$$vol(n) = |\{G^b | G^b \in H_i \wedge t(G^b) \in [t_n - L + 1, t_n] \wedge \exists G^a [G^a = pre(G^b) \wedge ([n^a \notin N(G^a) \wedge n^b \in N(G^b)] \vee [n^a \in N(G^a) \wedge n^b \notin N(G^b)])]\}|$$

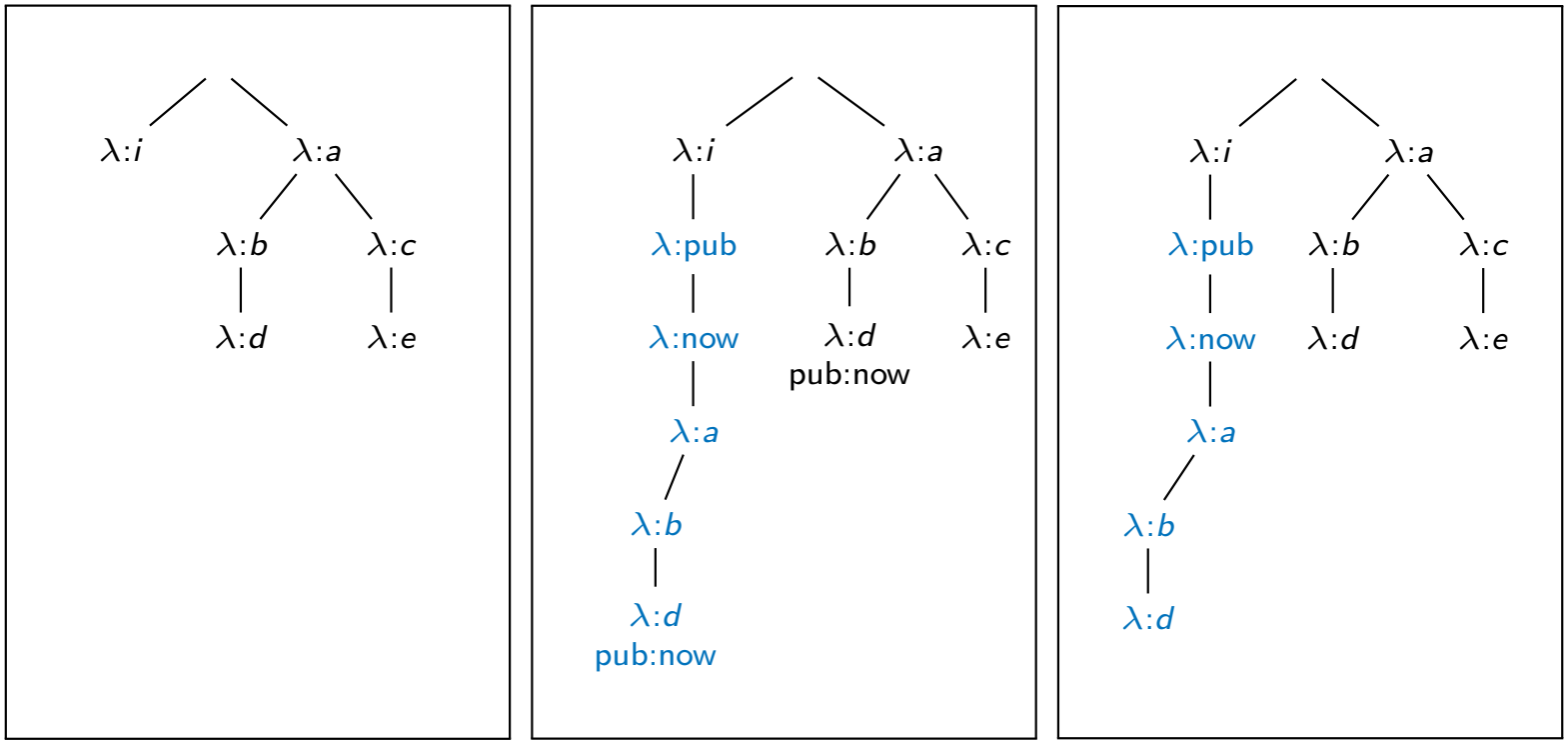
Definition (Volatile Node)

Index node n is volatile iff n's volatility count is greater or equal than the volatility threshold τ, i.e.,

$$volatile(n) \iff vol(n) \geq \tau$$

Index nodes becoming volatile

$$G^0 \xrightarrow{T_1} G^1 \xrightarrow{T_2} G^2$$



Snapshot G^0
$t(G^0) = t$

Snapshot G^1
$t(G^1) = t + 1$

Snapshot G^2
$t(G^2) = t + 2$

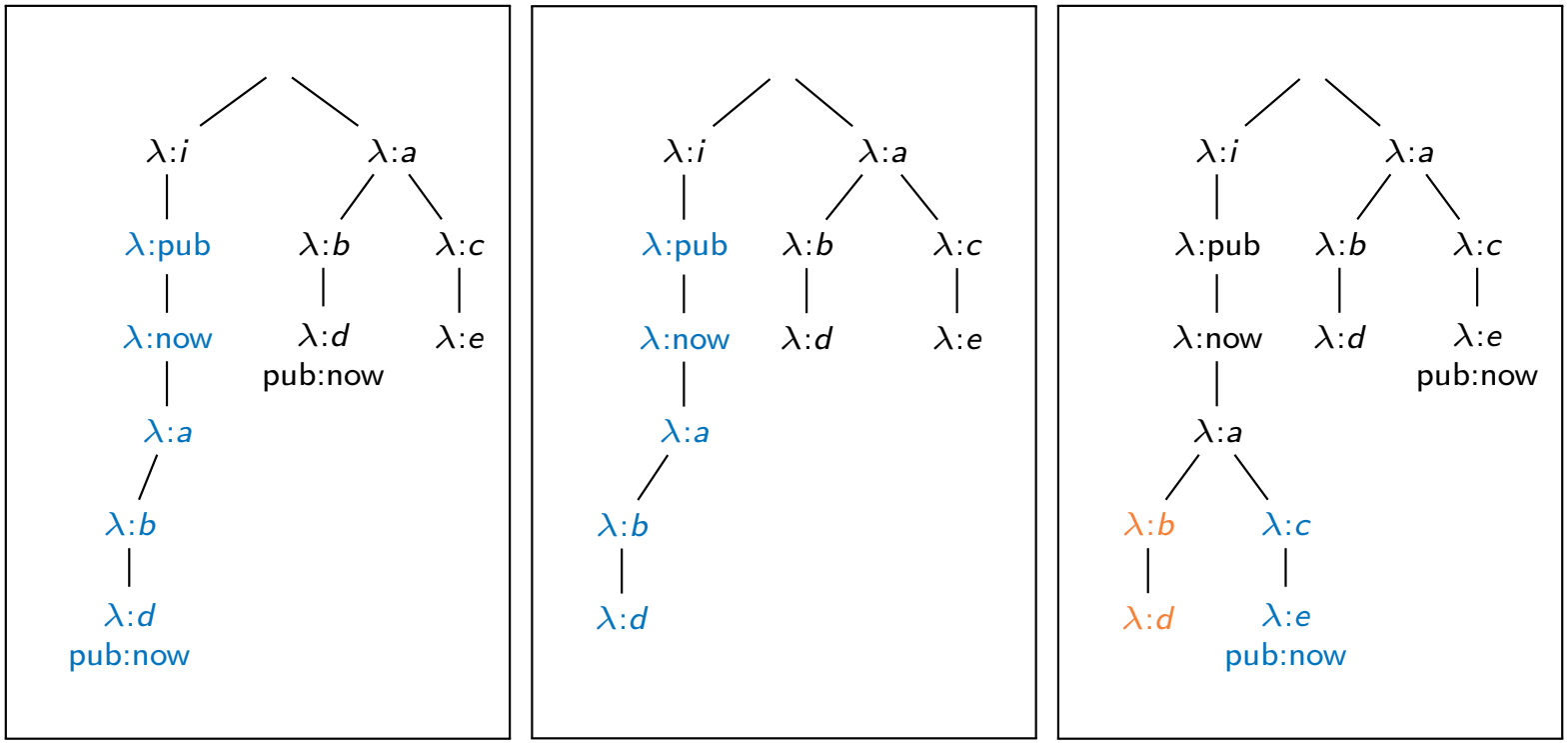
$\tau = 1, L = 2$

Unproductive Nodes

- When nodes are volatile, their volatility count has to be at least τ
- Time passes and the database workload changes
- Insertions and deletions that increased the volatility count drop out of the sliding window
- If the volatility count drops below threshold τ, the node ceases to be volatile
- If the same holds for the node's descendants, we call the node and its descendants unproductive

Index nodes becoming unproductive

$$G^1 \xrightarrow{T_2} G^2 \xrightarrow{T_3} G^3$$



Snapshot G^1
$t(G^1) = t + 1$

Snapshot G^2
$t(G^2) = t + 2$

Snapshot G^3
$t(G^3) = t + 3$

$\tau = 1, L = 2$

Variables impacting unproductive node production rate:

- Volatility threshold τ
- Sliding window length L
- Workload skew s
- Update tx per second

Unproductive index node cleaning, we propose:

- Periodic Garbage Collection (GC)
- Query-Time Pruning (QTP)

Periodic Garbage Collection (GC)

Periodic GC

Main idea:

- Background process
- Periodically traverse index subtree
- Prune any visited unproductive node

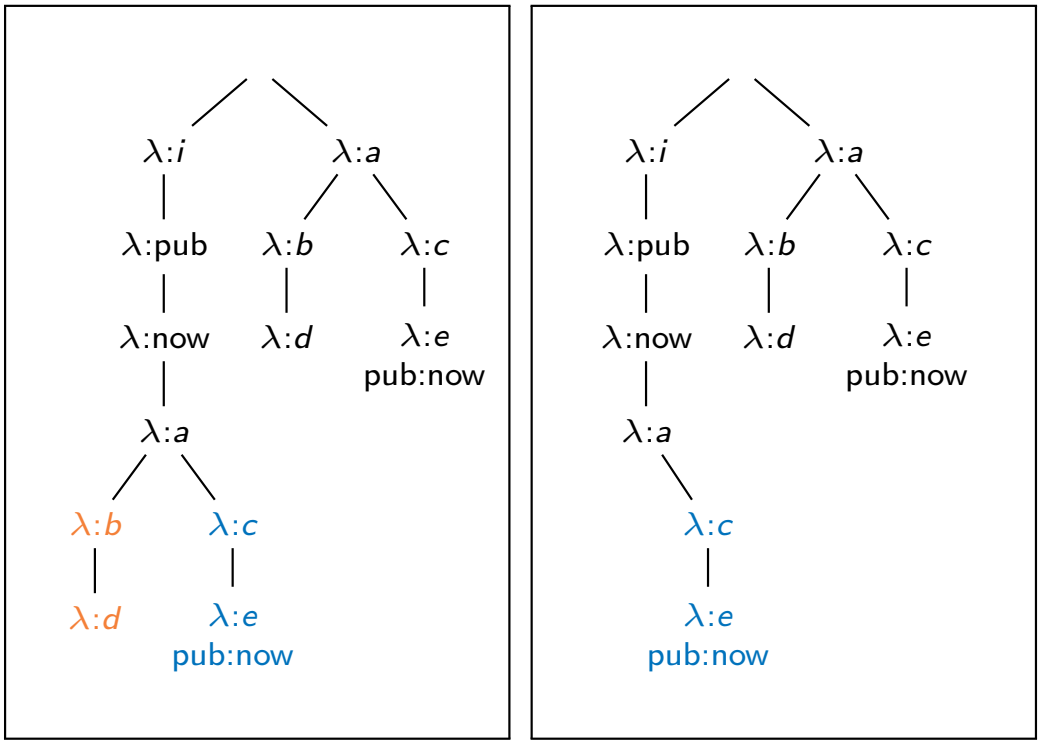
Periodic GC

Algorithm: GarbageCollect

for *node* $n \in desc(/i)$ *in postorder tree walk* **do**
    if $chd(n) = \emptyset \wedge \neg matching(n) \wedge \neg volatile(n)$ **then**
        delete node n

Periodic GC

$$G^3 \xrightarrow{T_4} G^4$$



Snapshot G^3
$t(G^3) = t + 3$

Snapshot G^4
$t(G^4) = t + 4$

$\tau = 1, L = 2$

Query-Time Pruning (QTP)

Query-Time Pruning

Main idea:

- Prune unproductive nodes during query execution
- Piggypacking on query execution
- Adds overhead on query runtime

Query-Time Pruning

Algorithm: QueryQTP

Data: Query $Q(k, v, m)$, where k is a property, v a value and m $(= /\lambda_1/\ldots/\lambda_d)$ a content node's path.

Result: A set of nodes satisfying $Q(k, v, m)$

$r \longleftarrow \emptyset$

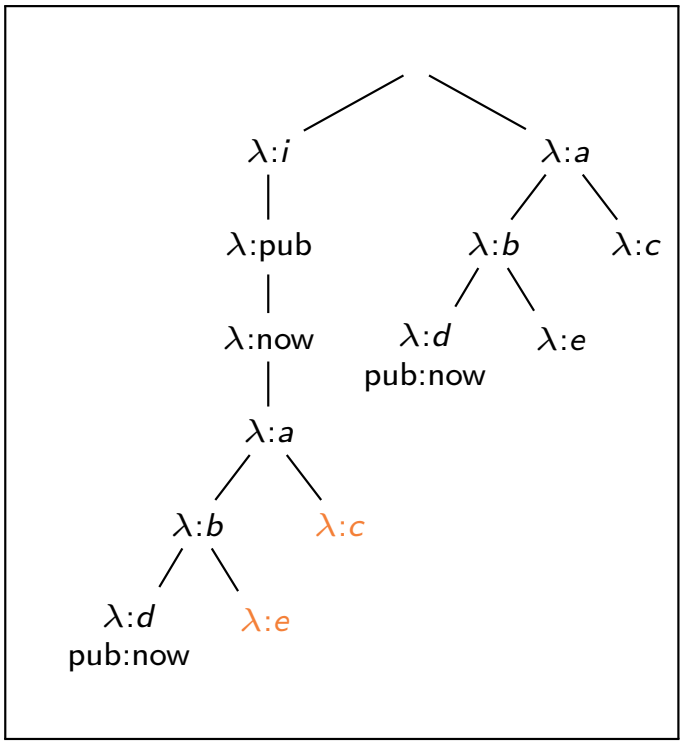
for *node* $n \in desc(/i/k/v/\lambda_1/\ldots/\lambda_d)$ *in postorder tree walk* **do**

- **if** $matching(n)$ **then**
  - $r \longleftarrow r \cup \{*n\}$
- **else if** $chd(n) = \emptyset \wedge \neg volatile(n)$ **then**
  - delete node n

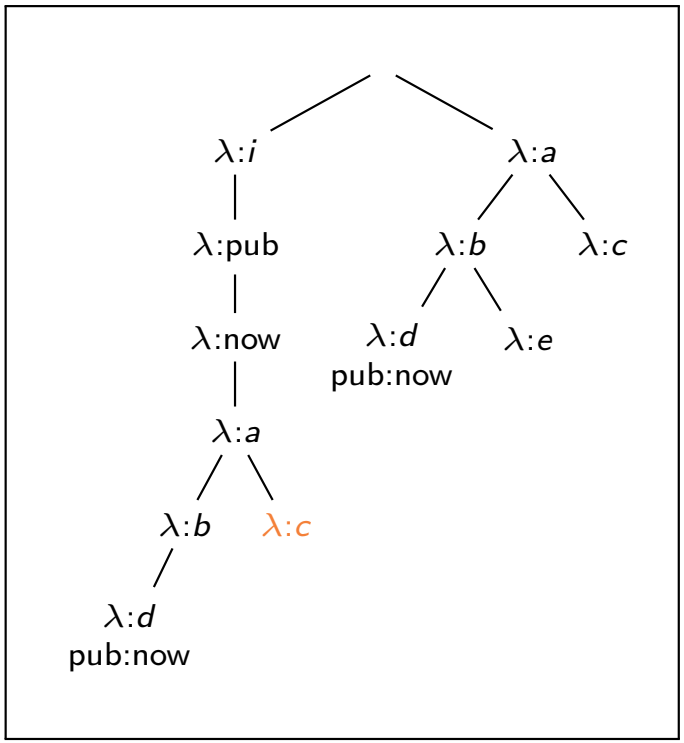
return r

Query-Time Pruning

$$G^5 \xrightarrow{T_6} G^6$$



Snapshot G^5



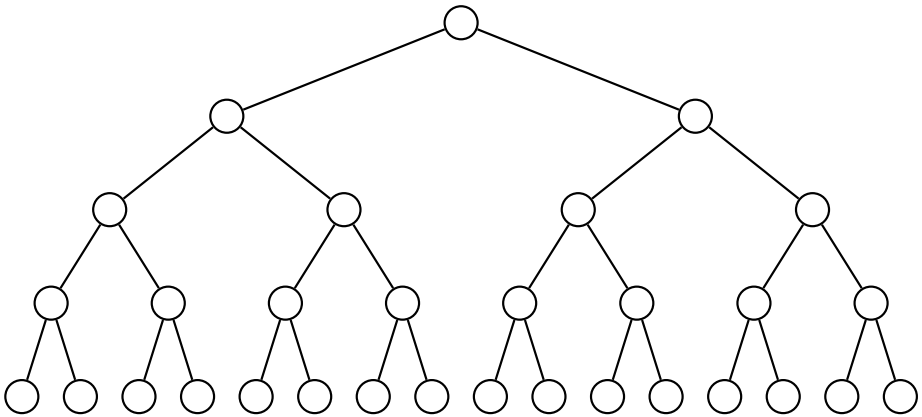
Snapshot G^6

$\tau = 1, L = 2, Q(\texttt{pub}, \texttt{now}, \texttt{/a/b})$

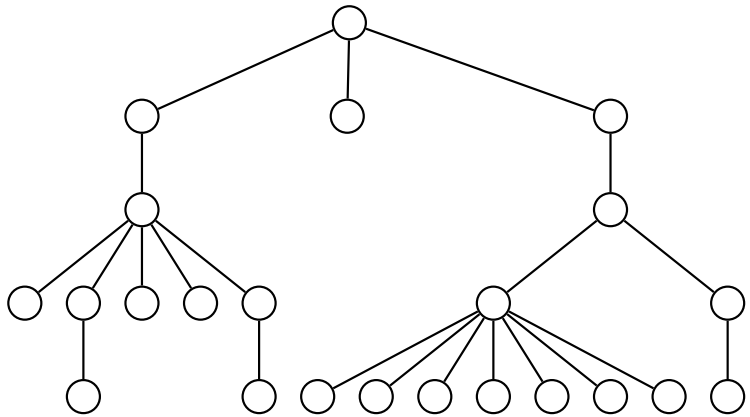
Experimental Evaluation

Datasets

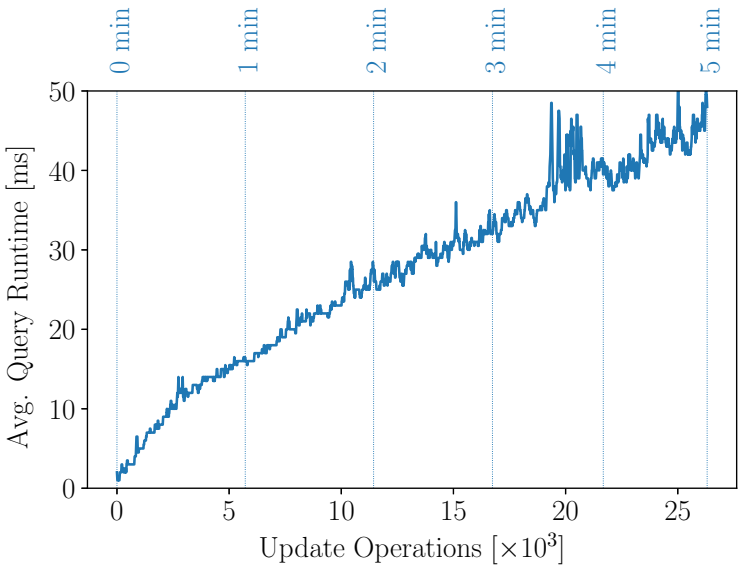
(a) Synthetic

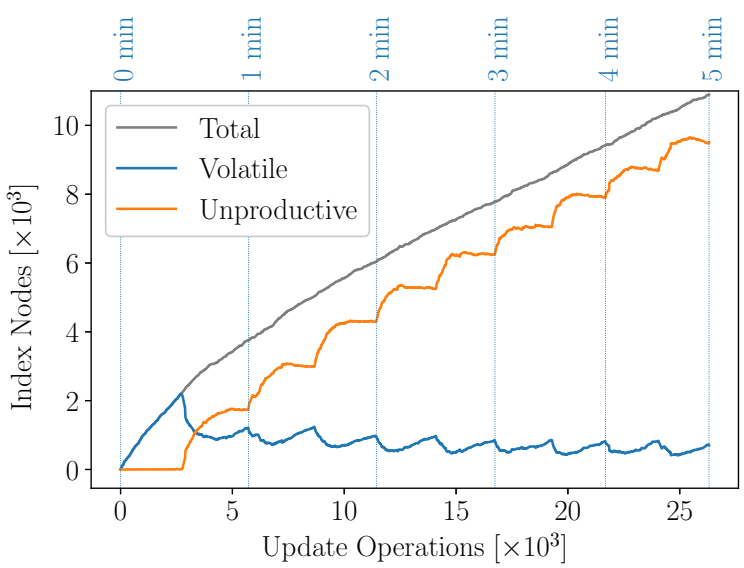


(b) Real-World

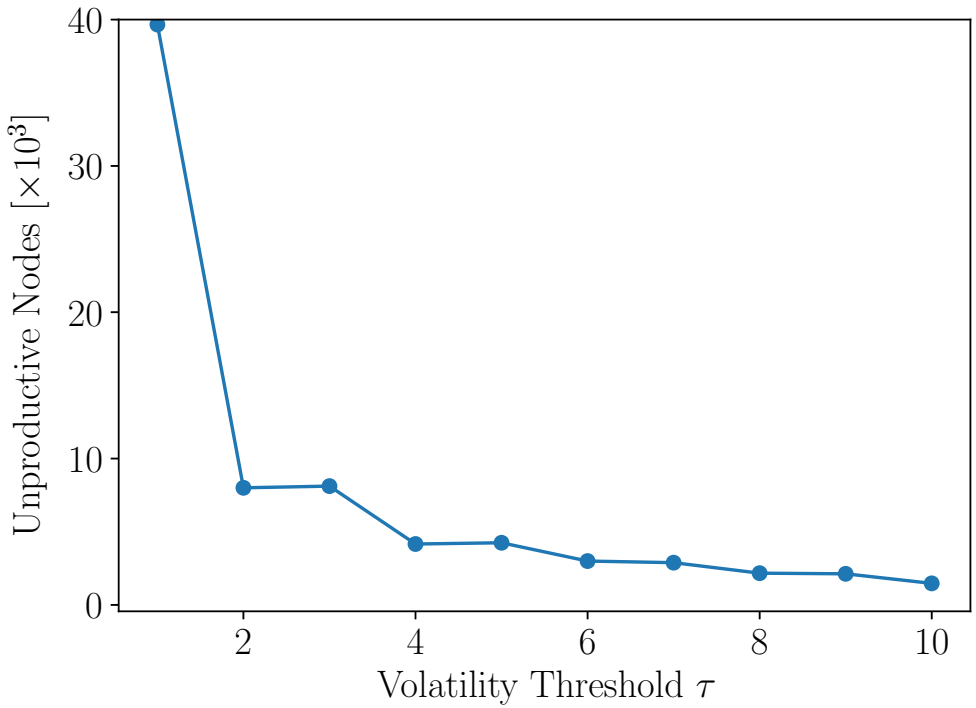


Impact on Query Runtime





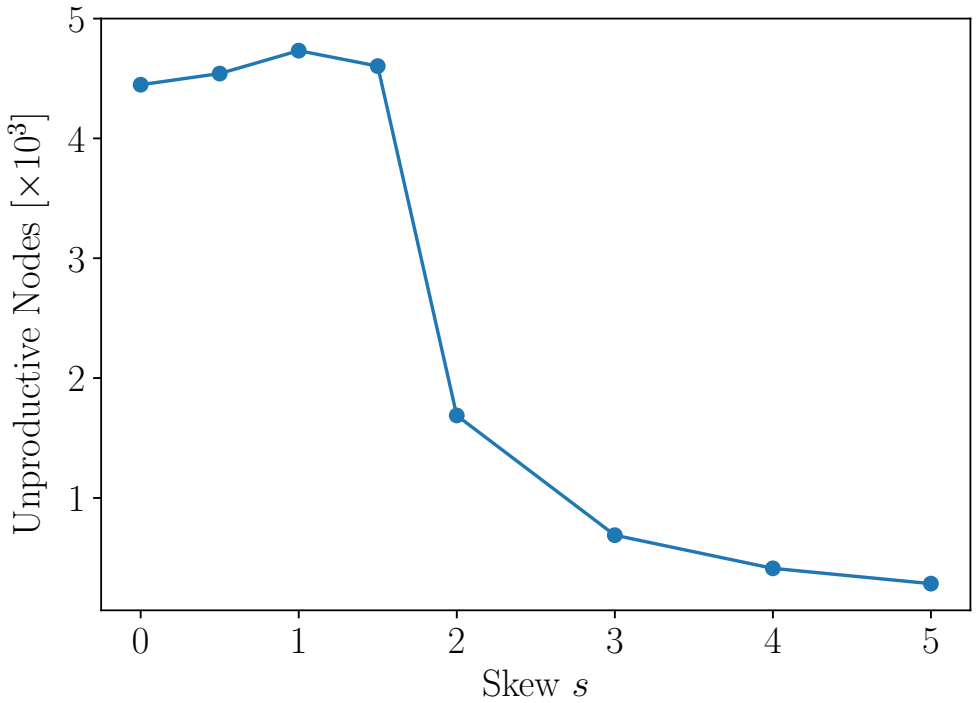
Volatility threshold τ



Volatility threshold τ

- τ increases $\Longrightarrow$ nodes are less likely to become volatile
- Fewer volatile nodes $\Longrightarrow$ fewer unproductive nodes
- Power law relationship between #unproductive nodes and τ

Workload skew s



Workload skew s

- very high $s \implies$ very small hotspot
- very small s (uniform) $\implies$ no hotspot
- very small/no hotspot $\implies$ nodes are less likely to become volatile and later on unproductive

References


Mathis, C., Härder, T., Schmidt, K., and Bächle, S.
XML indexing and storage: fulfilling the wish list.
Computer Science - R&D 30, 1 (2015), 51–68.


Wellenzohn, K., Böhlen, M., Helmer, S., Reutegger, M., and Sakr, S.
A Workload-Aware Index for Tree-Structured Data.
To be published.