Department of Informatics, University of Zürich

BSc Thesis

An Adaptive Index for Hierarchical Database Systems

Rafael Kallis

Matrikelnummer: 14-708-887

Email: rk@rafaelkallis.com

February 1, 2018

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]

Department of Informatics

D B T G

Abstract

The workload aware property index is a hierarchical index which adapts to the database's recent transactional workload by not pruning volatile index nodes, that is nodes which are frequently inserted or deleted, in order to increase update performance. When the workload changes, these nodes cease to be volatile and become unproductive if they and their descendants neither contribute to a query match nor are volatile.

Unproductive nodes in hierarchical indexes waste space and slow down queries. We propose periodic Garbage Collection and Query-Time pruning in order to clean unproductive nodes in the index. We implement the techniques in Apache Jackrabbit Oak and provide an extensive experimental evaluation to stress test the algorithms and show that the database throughput increases considerably when periodic Garbage Collection or Query-Time pruning is applied.

Zusammenfassung

Der “workload aware property index” ist ein hierarchischer Index, der sich an die jüngste Transaktionslast der Datenbank anpasst, indem er volatile Indexknoten, dh Knoten, die häufig eingefügt oder gelöscht werden, nicht löscht, um die Schreibleistung zu erhöhen. Wenn sich die Arbeitslast ändert, sind diese Knoten nicht mehr volatil und werden unproduktiv, wenn sie und ihre Nachkommen weder zu einer Abfrage beitragen noch volatil sind.

Unproduktive Knoten in hierarchischen Indizes verschwenden Speicherplatz und verlangsamen die Abfragen. Wir schlagen periodische Indexreinigung und Abfragezeitbereinigung vor, um den Index von unproduktiven Knoten zu bereinigen. Wir implementieren unsere Techniken in Apache Jackrabbit Oak und bieten eine umfangreiche experimentelle Auswertung um die Algorithmen unter hoher Last zu testen und zeigen, dass der Datenbankdurchsatz erheblich zunimmt, wenn periodische Indexreinigung oder Abfragezeitbereinigung benutzt wird.

Contents

List of Figures

1 Introduction

Frequently adding and removing data from hierarchical indexes causes them to repeatedly grow and shrink. A single insertion or deletion can trigger a sequence of structural index modifications (node insertions/deletions) in a hierarchical index. Skewed and update-heavy workloads trigger repeated structural index updates over a small subset of nodes to the index. Informally, a frequently added or removed node is called *volatile*. Volatile nodes decrease index update performance due to two reasons. First, frequent structural index modifications are expensive since they cause many disk accesses. Second, frequent structural index modifications also increase the likelihood of conflicting index updates by concurrent transactions. Conflicting index updates further decrease update performance since concurrency control protocols need to resolve the conflict.

Wellenzohn et al. [3] propose the Workload-Aware Property Index (WAPI). The WAPI exploits the workload's skewness by identifying and not removing volatile nodes from the index, thus significantly reducing the number of expensive structural index modifications. Since fewer nodes are inserted/deleted, the likelihood of conflicting index updates by concurrent transactions is reduced.

Some Content Management Systems (CMS) make use of hierarchical databases. The Adobe Experience Manager,[1] Adobe's enterprise CMS works with the hierarchical database system Apache Jackrabbit Oak (Oak). CMSs yield skewed, update-heavy and changing workloads. They frequently update a small changing subset of index nodes. Such workloads decrease WAPI's query performance.

When the workload characteristics change, new index nodes can become volatile while others cease to be volatile and become *unproductive*. Unproductive index nodes slow down queries as traversing an unproductive node is useless, because unproductive nodes do not contains any data and thus cannot yield a query match. Additionally, unproductive nodes occupy storage space that could otherwise be reclaimed. lf the workload changes frequently, unproductive nodes accumulate in the index and the query performance deteriorates over time. Therefore, unproductive nodes must be cleaned up to keep query performance stable over time and reclaim disk space as the workload changes.

Wellenzohn et al. [3] propose periodic Garbage Collection (GC), which traverses the entire index subtree and prunes all unproductive index nodes at once. Additionally we propose Query-Time Pruning (QTP), an incremental approach to cleaning up unproductive nodes in the index. The idea is to turn queries into updates. Since Oak already traverses unproductive nodes as part of query processing, these nodes could be pruned at the same time. In comparison to GC, with QTP only one query has to traverse an unproductive node, while subsequent queries can skip this overhead and thus perform better. The goal of this BSc thesis is to study, implement, and empirically compare GC and QTP as proposed by [3] in the open-source hierarchical distributed database Apache Jackrabbit Oak.

[1] http://www.adobe.com/marketing-cloud/experience-manager.html

2 Background

2.1 CMS Workload

Depending on the data model, some applications might use a hierarchical database system such as Apache Jackrabbit Oak, called Oak. A content management system (CMS) might choose Oak because the database reflects the hierarchical structure of a webpage. Content management systems have specific workloads. These workloads have distinct properties: they are *skewed*, *update-heavy* and *changing* [3]. CMSs frequently use a job-queuing system that has the noted characteristics.

Consider a social media feed as a running example. Only a few posts are popular. These posts have many comments or likes. Since most of the interactions (comments, likes) are on a small subset of the posts, we have a skewed workload. Users submit new posts or interact with existing posts by writing comments for example, creating an update-heavy workload. As time passes, new posts are created. Users are more likely to interact with recent posts than older ones, hence the workload changes over time.

When a user submits a new post, a job is sent to the CMS for processing. A background thread is periodically checking for pending jobs. A pending job is signaled using node properties in Oak. The CMS adds a property to the respective node in order to signal the background thread that the specific node has a pending job requiring processing. A node's "pub" property signals when the job has to be processed. Setting the value of "pub" to "now" indicates that the job must be processed immediately. Once the background thread detects the node, the job is processed and the "pub" property is removed.

From now on, we shall refer to a workload with the properties mentioned above as a *CMS workload*. When a hierarchical database operates under a CMS workload, we can increase its update performance using the workload aware property index.

2.2 Workload Aware Property Index (WAPI)

Oak mostly executes content-and-structure (CAS) queries [2]. We denote node n's property k as $n[k]$ and node n's descendants as $desc(n)$.

Definition 1 (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.,

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

Example 1 (CAS Query)**.** Consider Figure 1. CAS-Query $Q(\text{pub}, \text{now}, \texttt{/a})$, which queries for every descendant of `/a` with "pub" set to "now", would evaluate to $Q(\text{pub}, \text{now}, \texttt{/a}) = \{\texttt{/a/b/d}\}$, since `/a/b/d` is the only descendant of `/a` with "pub" set to "now".

Figure 1 depicts a database instance with the workload aware property index (WAPI). The WAPI is a hierarchical index which indexes the properties of nodes in order to answer

CAS queries efficiently. The WAPI is hierarchically organized under node `/i`, the *Index Subtree Root*. The second level consists of all properties k we want to index, such as "pub". The third level contains any values v of property k, for example "now". The remaining levels replicate all nodes from the root node to any content node with k set to v.

When processing a CAS query, Oak traverses the WAPI in order to answer the query efficiently. Any index node has a *corresponding* content node. Given index node n, we denote n's corresponding content node as $*n$. If index node n's path is $path(n) = \texttt{/i/k/v/m} = \texttt{/i/k/v/}\lambda_1\texttt{/}\ldots\texttt{/}\lambda_d$, then n's corresponding content node $*n$ has the path $path(*n) = \texttt{m} = \texttt{/}\lambda_1\texttt{/}\ldots\texttt{/}\lambda_d$.

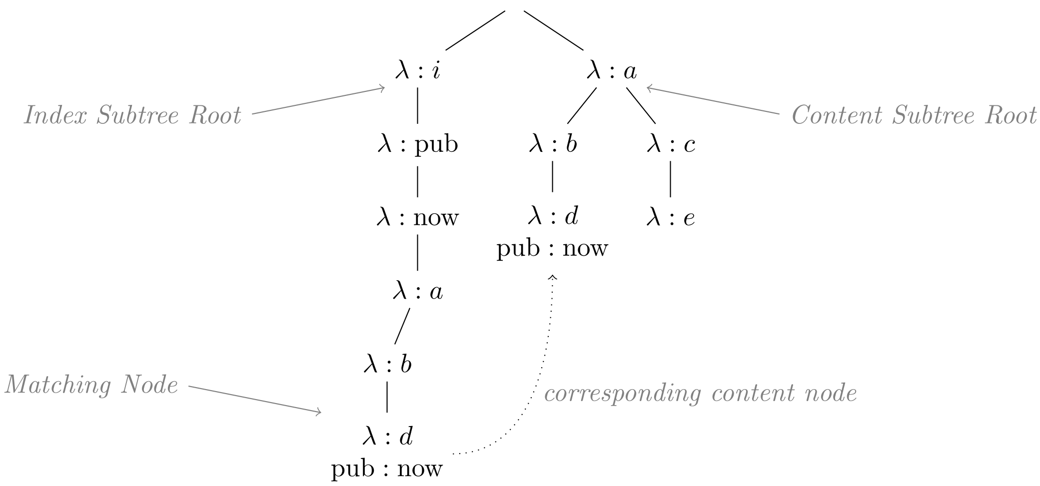


Figure 1: An instance of a hierarchical database. The index subtree is rooted at `/i`. The content subtree is rooted at `/a`. Index node `/i/pub/now/a/b/d` is matching, since its corresponding content node `/a/b/d` has property "pub" set to "now".

Definition 2 (Matching Node)**.** Index node n, with path `/i/k/v/m`, is matching iff n and n's corresponding content node $*n$, with path `m`, have property `k` set to `v`, i.e.,

$$matching(n) \iff *n[k] = v \wedge n[k] = v$$

Example 2 (Matching Node)**.** Consider Figure 1. The subtree rooted at `/a` is the content subtree. The subtree rooted at `/i` is the index subtree. Node `/i/pub/now/a/b/d` is matching, since its corresponding content node, `/a/b/d`, has property "pub" set to "now" as well as `i/pub/now/a/b/d` itself, too.

From the CMS workload described in Section 2.1 we can infer that the index subtree has a small number of matching nodes relative to the number of nodes in the content subtree. The index is used to signal pending jobs to the background thread using the "pub" property. Assuming jobs are processed by the background thread and removed from the index faster than they are created, the number of matching nodes should be close to zero. We infer that an index of a database which resembles a job-queuing system under a CMS workload should have almost no matching nodes.

The CMS workload is also skewed and update-heavy, therefore causing repeated structural index updates (insertions/deletions) over a small subset of nodes to the index. The WAPI takes into account if an index node is frequently added and removed, i.e. *volatile* (see Definition 4), before performing structural index modifications. If a node is considered volatile, we do not remove it from the index.

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. [3] propose to look at the recent transactional workload to check whether a node n is volatile. The workload on Oak instance O_i is represented by a sequence $H_i = \langle \ldots, G^a, G^b, G^c \rangle$ of snapshots, called a *history*. A snapshot represents an immutable committed tree of the database. Let t_n be the current time and $t(G^b)$ be the point in time snapshot G^b was committed, $N(G^a)$ is the set of nodes which are members of snapshot G^a. We use a superscript a to emphasize that a node n^a belongs to tree G^a. $pre(G^b)$ is the predecessor of snapshot G^b in H_i.

Given two snapshots G^a and G^b we write n^a and n^b to emphasize that nodes n^a and n^b are two versions of the same node n, i.e., they have the same absolute path from the root node.

Node n is volatile iff n's volatility count is at least τ, called volatility threshold. The volatility count of n is defined as the number of times n was added or removed from snapshots in a sliding window of length L over history H_i.

Definition 3 (Volatility Count)**.** The volatility count $vol(n)$ of index node n on Oak 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.

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

Definition 4 (Volatile Node)**.** Index node n is volatile iff n's volatility count (see Definition 3) is greater or equal than the volatility threshold τ, i.e.,

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

Example 3 (Volatile Node)**.** Consider the snapshots depicted in Figure 2. Assume volatility threshold $\tau = 1$, sliding window of length $L = 2$ and history $H_h = \langle G^0, G^1, G^2, G^3 \rangle$. Oak instance O_h executes transactions $T_1, \ldots, T_3$. Note that volatile index nodes are color-coded blue in Figure 2. Snapshot G^0 was committed at time $t(G^0) = t$. Given snapshot G^0, transaction T_1 adds property "pub" = "now" to `/a/b/d` and commits snapshot G^1 at time $t(G^1) = t{+}1$. Next, transaction T_2 removes property "pub" from `/a/b/d` given snapshot G^1 and commits snapshot G^2 at time $t(G^2) = t+2$. During T_2 we have $t_n = t+2$ and the sliding window is $[t_n - L + 1, t_n] = [t+2-2+1, t+2] = [t+1, t+2]$. Snapshot G^1 is in the sliding window since $t(G^1) = t+1$ and $t+1 \in [t+1, t+2]$. Snapshot G^0 is not in the sliding window because $t(G^0) = t$ and $t \notin [t+1, t+2]$. Index node $n =$ `/i/pub/now/a/b/d` exists in the set of nodes which are members of snapshot G^1, $n^1 \in N(G^1)$ but not in its predecessor $pre(G^1) = G^0$, $n^0 \notin N(G^0)$ and therefore has a volatility count of $vol(n) = 1$. Since the threshold is $\tau = 1$, the node is volatile. The same holds for $n's$ ancestors within the index subtree.

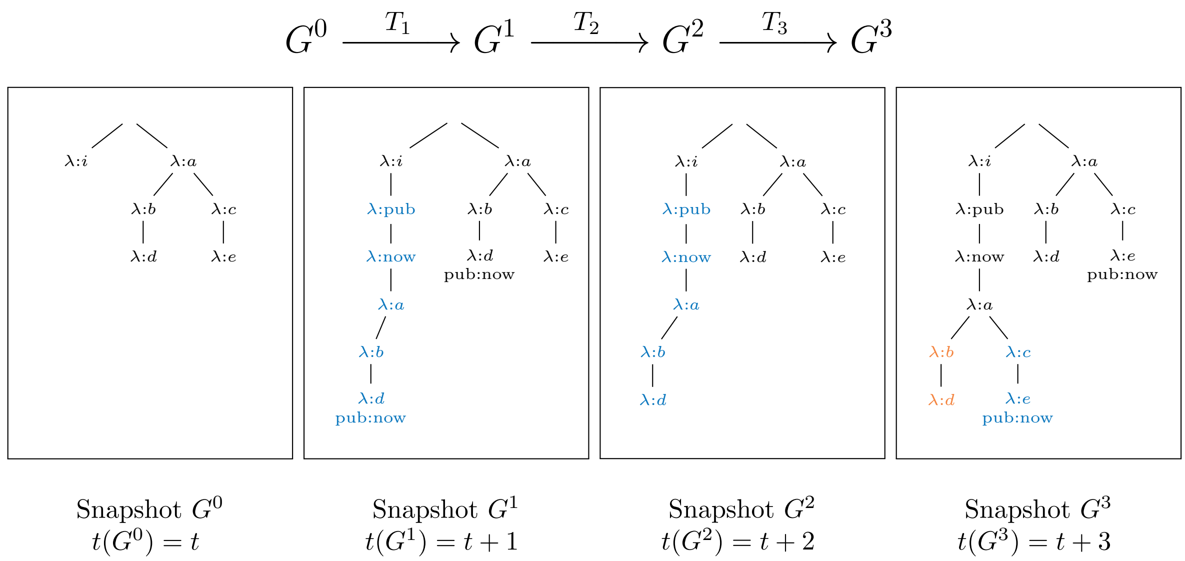


Figure 2: Volatile nodes becoming unproductive. Given $\tau = 1$, $L = 2$, nodes `/i/pub/now/a/b/d` and `/i/pub/now/a/b` are unproductive in snapshot G^3. They are not volatile and don't match either. Note that volatile and unproductive index nodes are color-coded blue and orange, respectively.

3 Unproductive Nodes

3.1 Introduction

When time passes and the database workload changes, volatile nodes cease to be volatile and they become unproductive. When nodes are volatile, their volatility count has to be at least τ. When time passes, insertions and deletions that increased the volatility count drop out of the sliding window, causing the volatility count to decrease. If the volatility count drops below threshold τ, the node ceases to be volatile. If the now non-volatile node is also non-matching, and the same holds for its descendants, we call the node and its descendants unproductive.

Unproductive index nodes slow down queries as traversing an unproductive node is useless, because neither the node itself nor any of its descendants are matching and thus cannot yield a query match. Additionally, unproductive nodes occupy storage space that could otherwise be reclaimed. If the workload changes frequently, unproductive nodes accumulate in the index and the query performance deteriorates over time [3].

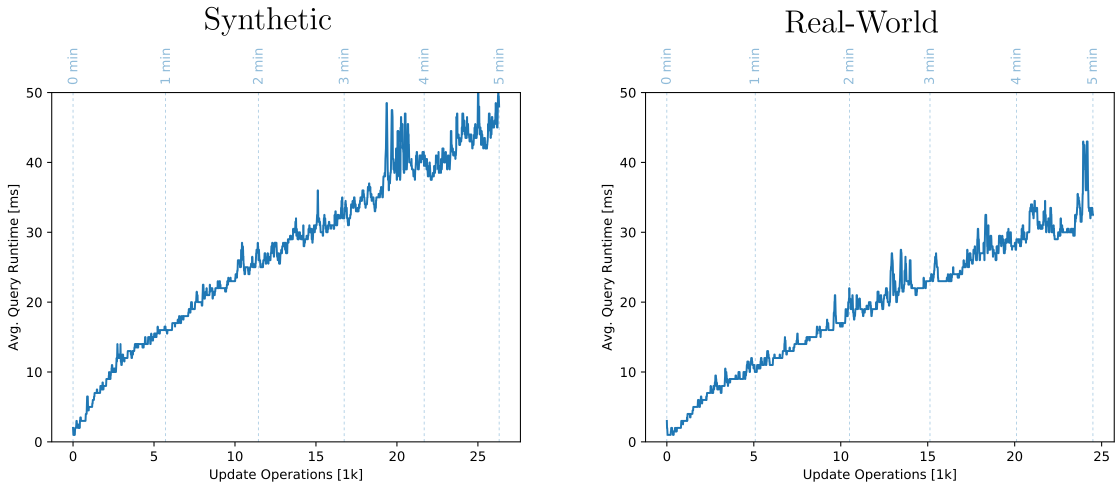
Definition 5 (Unproductive Node)**.** Index node n is unproductive iff n, and any descendant of n, is neither matching (see Definition 2) nor volatile (see Definition 4), i.e.,

$$\mathit{unproductive}(n) \iff \forall m(m \in (\{n\} \cup \mathit{desc}(n)) \wedge \neg \mathit{matching}(m) \wedge \neg \mathit{volatile}(m))$$

Example 4 (Unproductive Node)**.** In our running example (cf. Figure 2), transaction T_3 adds property "pub" = "now" to `/a/c/e` to G^2 and commits G^3 at time $t(G^3) = t+3$. We assume the same parameterization as in the last example ($\tau = 1, L = 2$). Volatile and unproductive index nodes are color-coded blue and orange, respectively. In snapshot G^3, index nodes `/i/pub/now/a/b/d` and `/i/pub/now/a/b` cease to be volatile because their volatility counts drop below the threshold. The sliding window has length $L = 2$, so we only consider snapshots G^2, G^3 towards the volatility count. The two nodes were not inserted or deleted in any of the mentioned snapshots and therefore the volatility count is $\mathit{vol}($`/i/pub/now/a/b/d`$) = \mathit{vol}($`/i/pub/now/a/b`$) = 0$. Since the threshold is $\tau = 1$, the nodes are not volatile. In addition, they are not matching either, therefore they are unproductive (cf. Definition 5). Index node `/i/pub/now/a` is productive in snapshot G^3 since it has two volatile descendants (`/i/pub/now/a/c/e`, `/i/pub/now/a/c`) one of which is matching, too.

In the example above, we saw how index nodes become unproductive after being volatile. Index nodes do not necessarily have to be volatile before they become unproductive. If a non-volatile and non-matching index node with no matching descendants has a volatile descendant, and the descendant ceases to be volatile, the index node becomes unproductive, even though it was not volatile.

Example 5 (CAS Query with Unproductive Nodes)**.** Consider CAS-Query $Q(\text{pub}, \text{now},$ `/a`$)$ from Example 1 again. We apply the query to snapshot G^3 in Figure 2. The query executor has to traverse the four descendants of `/i/pub/now/a`. Traversing nodes `/i/pub/now/a/b/d` and `/i/pub/now/a/b` is useless and slows down the query because these nodes are unproductive. The query evaluates to $Q(\text{pub}, \text{now},$ `/a`$) = \{$`/a/c/e`$\}$.



(a) (b)

Figure 3: Query Runtime over time

3.2 Impact on Query Runtime

In this section we study and quantify the impact of unproductive nodes on query runtime. During query execution, traversing an unproductive node is useless, because neither the node itself nor any of its descendants are matching and therefore cannot contribute a query match. We hypothesize that unproductive nodes significantly slow down queries under a CMS workload. An index under a CMS workload (see Section 2.1) is dominated by unproductive nodes, that is unproductive nodes constitute a large percentage of all index nodes.

In order to find supporting evidence for the hypotheses above, we conduct a series of experiments on Oak. The setup of the experimental evaluation and datasets will be described in detail in Section 5. We record the query runtime throughout the experiment and present the data below.

Figures 3a and 3b show the query runtime of the same query as time passes by for the synthetic and real-world dataset respectively. Each point corresponds to the moving median over 10 time points. We observe a sublinear increase of the runtime from $2ms$ to $50ms$ after running the simulation for 5 minutes on the synthetic dataset and an increase from $2ms$ to $35ms$ on the real-world dataset.

Next, we present data regarding the type of index nodes traversed during query execution. Figures 4a and 4b depict the total number of traversed nodes in addition to the number of traversed volatile and unproductive nodes during query execution for each dataset.

The total number of traversed nodes is increasing sublinearly over time. This explains the increase in query runtime in Figures 3a and 3b. As time passes, more and more content nodes are randomly selected by the CMS workload and their corresponding index nodes become volatile. After some time these nodes, most likely become unproductive and are not being pruned from the index anymore. Therefore, the probability of picking a

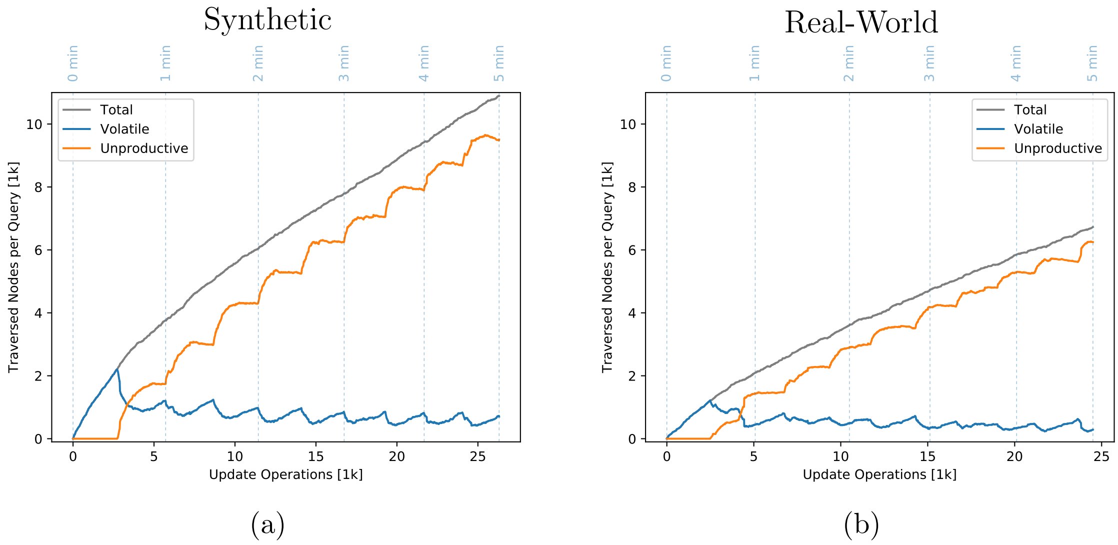


Figure 4: Index composition during Query Execution

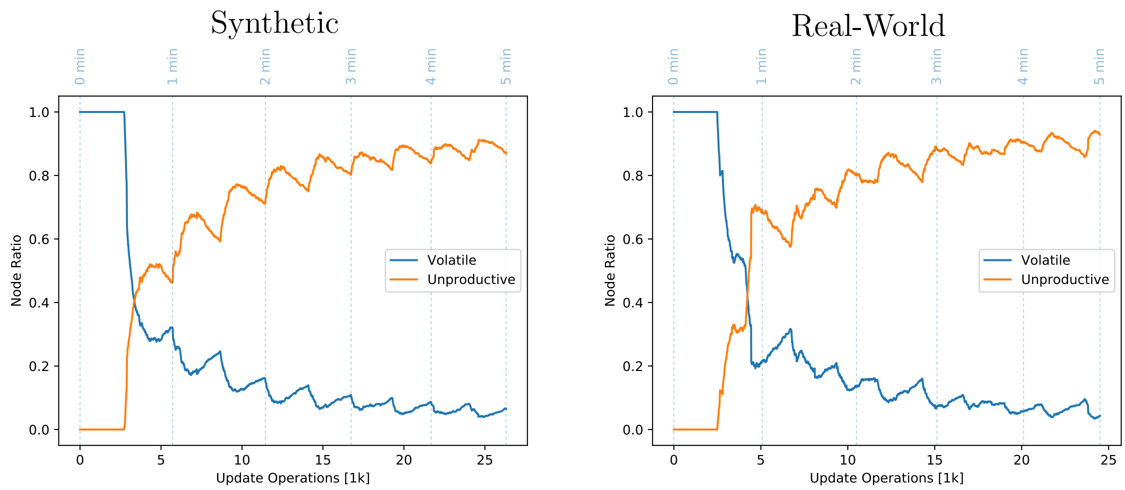
content node with no corresponding index node (non-indexed) decreases over time. Since it becomes less and less likely for a non-indexed content node to be randomly picked by the CMS workload, the rate of growth of total traversed index nodes decreases over time. We expect that if we change the workload sufficiently often, the number of total index nodes converges to the number of content nodes.

Furthermore, we see the number of volatile nodes have a downward trend from the 30 second mark till the end of the experiment. We believe it becomes less likely for nodes to become volatile if unproductive nodes are not pruned, as time passes. When a workload randomly picks a content node whose index node is unproductive, the index node becomes matching but was not physically added, thus the volatility count of the index node does not increment. In comparison, if the workload randomly picks a non-indexed content node, the corresponding index node's volatility count increments. We infer that it is less likely for an unproductive node to become volatile than a non-indexed one. Our experimental evaluation suggests that the number of unproductive nodes increases over time. Therefore it becomes less likely for any node to become volatile over time. This explains the decreasing number of volatile nodes over time.

The sliding window of length L is set to 30 seconds, therefore we encounter no unproductive nodes during the first 30 seconds of the simulation. Once we reach the 30 second mark, the query executor encounters unproductive nodes. From that point, we observe a steep increase in traversed unproductive nodes. After 1 minute, we observe that the total traversed nodes are dominated by unproductive nodes. The rate of growth of traversed unproductive nodes seems to decrease over time. When content nodes are picked by the workload, their corresponding index nodes are inserted into the WAPI, where some become volatile. When the workload changes, these volatile index nodes most likely become unproductive and accumulate in the index, hence the number of non-indexed content nodes decrease. Since less and less non-indexed content nodes are available to become volatile and thereafter unproductive, the rate of growth of traversed unproductive nodes decreases.

Additionally, we observe the functions of the unproductive and volatile index nodes having the shape of a cycloid. In Figure 4a, each cusp $(30s, 60s, \ldots, 300s)$ represents a point in which the workload changes during the experiment. When the workload changes, we initially see a steep decrease in volatile nodes. During that phase, more nodes cease to be volatile than become volatile. Nodes need to reach the threshold in order to become volatile and few do, since the skew in the workload only picks a subset of nodes frequently. Before a new workload kicks in, we observe the opposite phenomenon. More nodes become volatile than cease to be volatile, because many nodes are on the verge of becoming volatile and therefore need to be picked only a few times more to become volatile.

Lastly, we also observe the real-world dataset having a more gentle slope over the synthetic dataset. Since the real-world dataset has more content nodes, it is less likely for each content node to be picked by the workload. Having a smaller chance to be picked by the workload implies less volatile and consequently less unproductive nodes in the index.



(a) (b)

Figure 5: Node Ratio during Query Execution

Figures 5a to 5b show the ratio of 1) volatile over total and 2) unproductive over total index nodes over time and update operations from our datasets. These Figures quantify how strongly unproductive nodes dominate the total traversed nodes. After 5 minutes, unproductive nodes account for over 80% of the traversed nodes whilst less than 20% are volatile on the synthetic dataset. Similarly, on the real-world dataset, we observe 90% traversed unproductive and 10% traversed volatile index nodes.

Concluding, the experiments strongly supports our hypothesis. The query runtime increase by an order of magnitude after 5 minutes. Also, unproductive nodes, which dominate the index, are accountable for the increase in query runtime.

4 Cleaning Unproductive Nodes

In the previous section, we saw how unproductive nodes slow down query execution. To prevent unproductive nodes from accumulating in the index, we need strategies to clean up the index. In the following two subsections, we suggest two different approaches for dealing with unproductive nodes. We will empirically investigate their performance in Section 5.

4.1 Periodic Garbage Collection (GC)

First, we propose to clean the index periodically with a garbage collector. We add a background process that periodically executes garbage collection of unproductive nodes inside transactions in Oak.

The naïve approach for garbage collection is to traverse the entire index subtree, apply Definition 5 to each visited node n, and delete n if it is unproductive. Deciding if n is unproductive, requires us to check that no descendant $desc(n)$ of n is matching or volatile. Checking the descendants of each index node n causes GC to have a quadratic time complexity.

Example 6 (Naïve GC)**.** Assume we apply naïve GC to the index subtree depicted in Figure 6. The subtree resembles the worst-case scenario for GC. It contains n nodes, each node is productive and each internal node has a single child. In order to determine if node a_1 is unproductive, we have to traverse each descendant of a_1 and check if it is matching or volatile. The same holds for determining if any of a_1's descendants is unproductive. Since a_1 has $(n-1)$ descendants, we have to traverse $(n-1)$ nodes to determine if a_1 is unproductive. For node a_2, we have to traverse $(n-2)$ nodes since a_2 has $(n-2)$ descendants.

To determine if the nodes in the subtree rooted at a_1 are unproductive, we have:

$$\begin{aligned}
(n-1) + (n-2) + \cdots + (n-n-1) + (n-n) \\
&= \sum_{k=1}^{n} n-k \\
&= \sum_{k=1}^{n} n - \sum_{k=1}^{n} k \\
&= n^2 - \frac{n(n+1)}{2} \\
&= n^2 - \frac{n^2+n}{2} \\
&= \frac{n^2-n}{2} \\
&= \Theta(n^2)
\end{aligned}$$

We showed that applying naïve GC on an index subtree of n nodes is bound by $\Theta(n^2)$ in the worst case.

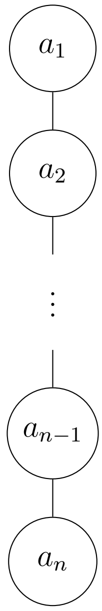


Figure 6: Worst case scenario during naïve Garbage Collection. No node is unproductive and all internal nodes have a single child.

By applying a *postorder tree walk* [1], we can garbage collect the index subtree in linear time complexity. The postorder tree walk allows us to process n's descendants before n. When visiting node n during a postorder tree walk, any unproductive descendant of n was pruned, hence any child of n cannot be unproductive. Thus, if n has at least one child, n cannot be unproductive and therefore checking if n has at least one child, in addition to checking if n is matching or volatile, is sufficient to determine if n is unproductive, when applying a postorder tree walk. Figure 7 depicts a postorder tree walk on snapshot G^3 from Figure 2. The numbers represent the order in which the nodes are checked and pruned if unproductive. A node's descendants are always evaluated before the node itself.

Algorithm 1 prunes all unproductive descendants of the index subtree rooted at `/i`. The algorithm traverses the subtree rooted at `/i` using a postorder tree walk. If a descendant $n \in desc(\texttt{/i})$ has no children and is neither matching, nor volatile, it is unproductive and pruned from the index. If n has at least one child, we infer that n has at least one matching or volatile descendant and thus n cannot be unproductive. The postorder tree walk ensures that the algorithm prunes a child before its parent node. This guarantees that all unproductive nodes are pruned.

Algorithm 1: CleanIndexWAPI

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

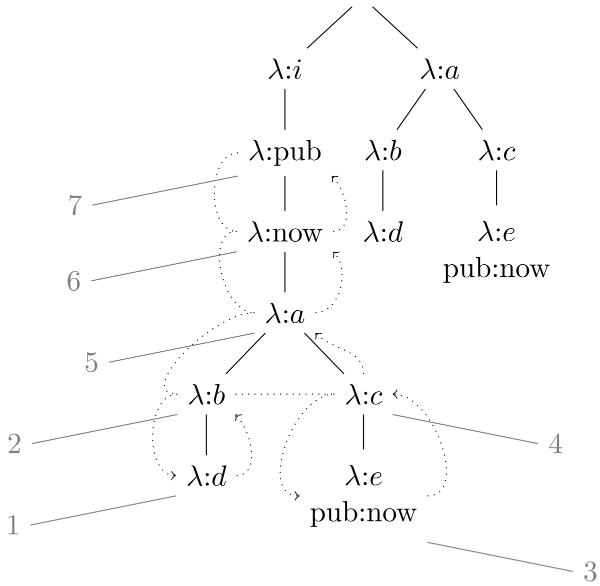


Figure 7: Postorder tree walk on the index subtree rooted at `/i` of G^3. The numbers represent the order the corresponding nodes where visited, e.g. `/i/pub/now/a/b/d` was visited first, `/i/pub/now/a/b` second, etc.

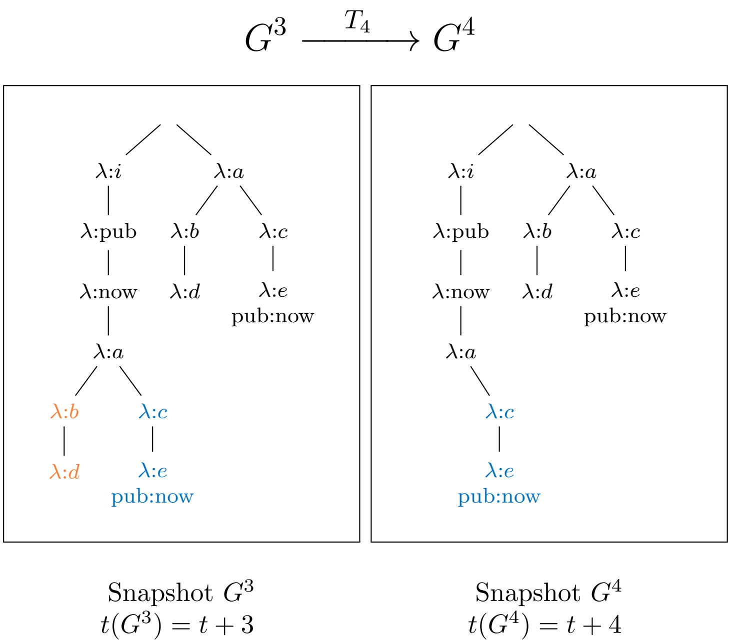


Figure 8: Garbage collection applied on Oak. Assume nodes `/i/pub/now/a/b/d` and `i/pub/now/a/b` are unproductive (color-coded orange) in snapshot G^3. Garbage collection is executed during transaction T_4. G^4 is the committed snapshot.

Example 7 (Periodic GC)**.** Figure 8 shows snapshot G^3 from Figure 2. Index nodes `/i/pub/now/a/b/d` and `/i/pub/now/a/b` are unproductive in snapshot G^3, as explained in Example 4. Oak's background process executes garbage collection during T_4. While executing GC, the index subtree of G^3 is traversed in postorder. The first unproductive node we encounter and prune is `/i/pub/now/a/b/d`. Next, the garbage collector encounters and prunes `/i/pub/now/a/b`. No further unproductive node is left for pruning. We see the cleaned index after garbage collection in snapshot G^4.

```java
/**
 * Removes any unproductive descendant from the index subtree.
 *
 * @param root: latest Oak snapshot
 */
void cleanIndexWAPI(Root root) {

    /* index subtree root */
    Tree indexRoot = root.getTree(OAK_INDEX_PATH);

    /* filter nodes which have children, are matching or volatile */
    for (Tree unproductiveNode : filter(
            (Tree n) -> n.getChildrenCount(1) == 0 &&
                        !isMatching(n) &&
                        !isVolatile(n),

            /* postorder tree walk iterable */
            postOrder(indexRoot)
    ) {
        unproductiveNode.remove();
    }
}
```

Figure 9: Periodic Garbage Collection implemented in Java

Figure 9 shows the implementation of the periodic GC in Java inside Apache Jackrabbit Oak. Calling `postOrder()` returns a lazy sequence of nodes which correspond to a postorder tree walk of the index subtree (cf. Figure 28 in the Appendix). Next, using `filter()` we remove any node that has children, is matching or volatile from the sequence. All other nodes are unproductive and therefore pruned from the index.

By applying periodic GC on Oak, we introduce a new parameter. *GC period* T defines how often garbage collection is run by the background process on Oak. If we pick a smaller period T, garbage collection is run more often and this reduces the average number of unproductive nodes in the index. As a result, we increase the query performance of Oak, since the query executor has to traverse less unproductive nodes on average.

It is also worth mentioning that GC uses system resources in order to clean the index. If run too often, GC might degrade query performance, because the system is busy cleaning the index, instead of processing queries.

We suggest running GC when the system is not busy executing queries, e.g, every day during the early morning hours. Like so, garbage collection minimally interferes with other transactions. We will revisit GC period in Section 5.8, where we conduct an experiment in order to determine the impact of period T on query performance.

4.2 Query-Time Pruning (QTP)

While periodic garbage collection is explicitly executed by Oak in order to clean unproductive nodes, Query-Time pruning is an approach which cleans unproductive nodes whilst Oak answers queries. Doing so, we benefit by avoiding the cost of explicitly traversing the index.

Frequent queries also benefit from QTP. If a query is executed for the first time, QTP cleans all unproductive nodes traversed during query execution. If the query is executed a second time shortly thereafter and no new nodes become unproductive in the meantime, the query executor encounters no unproductive nodes anymore and therefore the query is answered faster.

If the path filter of the queries changes often, the query executor starts traversing new subtrees in the index and consequently, some subtrees are not queried anymore. If an unproductive node is a member of a subtree that is not queried anymore, it is not pruned and remains in the index. Such unproductive nodes waste space because they contain no data but do not impact query performance since they are not traversed during query execution. At the present time, it remains unknown what the performance impact of a changing path filter is. We will investigate the very question in Section 5.9.

When using QTP to query for all descendants of node `m`, we apply a postorder tree walk. With the postorder tree walk, we traverse and clean all unproductive nodes in the subtree of $\texttt{/i/k/v/m} = \texttt{/i/k/v/}\lambda_1\texttt{/}\dots\texttt{/}\lambda_d$ in linear time complexity, as explained in Section 4.1. In the same Section we mention, that during a postorder tree walk, checking if node n has at least on child, in addition to checking if n is matching or volatile, is sufficient to determine if n is unproductive. The postorder tree walk ensures that the algorithm prunes a child before its parent node. This guarantees that all unproductive nodes are pruned in the subtree rooted at `/i/k/v/m`.

Algorithm 2 takes a CAS query $Q(\texttt{k}, \texttt{v}, \texttt{m})$ as an argument, where `k` is a property, `v` a value and `m` a content node's path. We initialize set r as the empty set. r will hold all content nodes satisfying the CAS query (cf. Definition 1). We traverse any descendant n of $\texttt{/i/k/v/m} = \texttt{/i/k/v/}\lambda_1\texttt{/}\dots\texttt{/}\lambda_d$ in a postorder tree walk. If node n is matching, we add its corresponding content node to r and proceed to the next descendant. If node n has no children and is neither matching, nor volatile, it is unproductive and pruned from the index. After we finish traversing all descendants n, we return the result set r.

Algorithm 2: 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* $n \in desc(\texttt{/i/k/v/}\lambda_1\texttt{/}\ldots\texttt{/}\lambda_d)$ *in postorder tree walk* **do**

  if $matching(n)$ **then**

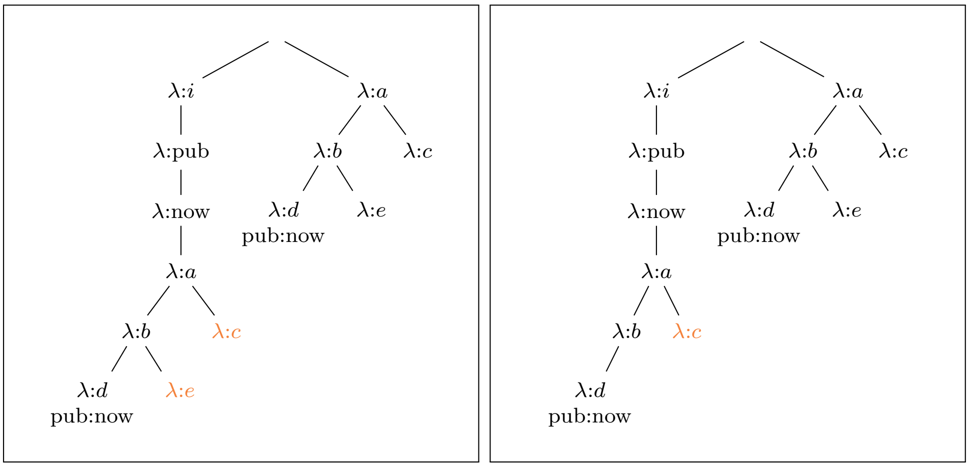
    $r \longleftarrow r \cup \{*n\}$

  else if $chd(n) = \emptyset \wedge \neg volatile(n)$ **then**

    delete node n

return r

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



Snapshot G^5 Snapshot G^6

Figure 10: Query-Time Pruning applied on Oak. Assume nodes `/i/pub/now/a/b/e` and `/i/pub/now/a/c` are unproductive in snapshot G^5. Transaction T_6 executes CAS query $Q(\texttt{pub}, \texttt{now}, \texttt{/a/b})$ which queries for all descendants of `/a/b` with "pub" set to "now" and commits the resulting snapshot G^6. QTP is used during query execution.

Example 8 (QTP)**.** Consider Figure 10. Transaction T_6 executes CAS query $Q(\texttt{pub}, \texttt{now}, \texttt{/a/b})$ which queries for all descendants of `/a/b` with "pub" set to "now" in snapshot G^5. Assume the query executor uses QTP and nodes `/i/pub/now/a/b/e` and `/i/pub/now/a/c` are unproductive. The query executor traverses all descendants of `/i/pub/now/a/b` and therefore prunes the unproductive index node `/i/pub/now/a/b/e`. Since the other unproductive index node (`/i/pub/now/a/c`) is not traversed during query execution, it is not pruned and remains in the index unproductive. The resulting snapshot G^6 is committed by T_6 after finishing query execution.

Figure 11 shows the Java implementation of QTP in Oak. The algorithm takes a property `k`, value `v`, path `m`, a `Root` and returns a lazy sequence of content nodes satisfying the CAS query $Q(\mathtt{k}, \mathtt{v}, \mathtt{m})$. By Calling `postOrder()` we get a sequence of nodes representing a postorder tree walk in the subtree rooted at `/i/k/v/m`. We remove any node that is not matching from the sequence using `filter()`. If a node has no children and is neither matching, nor volatile, it is unproductive and we prune it from the index.

```java
/**
 * Answers a CAS Query and prunes traversed unproductive index nodes.
 *
 * @param k: Property we query for
 * @param v: Value we query for
 * @param m: Path of content node which the descendants we query for
 * @param root: Latest Oak snapshot
 * @returns An iterable with content nodes satisfying the CAS query
 */
Iterable<Tree> QueryQTP(String k, String v, String m, Root root) {

    /* e.g.: /oak:index/pub/:index/now */
    String indexRootPath = concat(OAK_INDEX_PATH, k, v);

    /* e.g.: /oak:index/pub/:index/now/a */
    String queryNodePath = concat(indexRootPath, m);

    /* map index nodes to corresponding content nodes */
    return map((Tree n) -> {
            return root.getTree(relativize(indexRootPath, n.getPath()));
        },

        /* filter non-matching index nodes */
        filter((Tree n) -> {

                boolean isMatchingMemo = isMatching(n);

                /* prune if no children, not matching and not volatile */
                if (n.getChildrenCount(1) == 0 &&
                    !isMatchingMemo &&
                    !isVolatile(n)
                ) {
                    n.remove();
                }
                return isMatchingMemo;
            },

            /* postorder tree walk of descendants of /i/k/v/m */
            postOrder(root.getTree(queryNodePath))
        )
    );
}
```

Figure 11: Query-Time Pruning implemented in Java

5 Experimental Evaluation

5.1 Goals

The goal of our experimental evaluation is to compare GC and QTP under different parameterizations and workloads. In Sections 5.5 and 5.6, we see how different values of the volatility threshold τ and sliding window of length L impact unproductive nodes in the index. These parameters directly impact the number of volatile nodes and therefore can affect the production of unproductive nodes in the index. We compare GC and QTP side by side and list their differences in Section 5.7. Next, we investigate the effects GC period has on garbage collection in Section 5.8. Executing GC more often might increase system performance since more unproductive nodes are pruned. We investigate how the query filter affects QTP's overall performance in Section 5.9. Also, we conduct an experiment in Section 5.10 in order to examine how workload skew impacts the performance of GC and QTP. Finally, we run simulations in order to see how the update to query ration affects the performance of GC and QTP in Section 5.11.

5.2 Setup

All experiments are conducted on a 15" Macbook Pro 2015 inside a virtual machine running Linux Arch[2]. We allocate 4 out of 8 virtual cores (Intel i7-4980HQ 2.7 - 4.0 GHz) to the virtual machine and 12 GB of RAM. We allocated half the available cores of the machine because we wanted to ensure the CPU cooling performance will not affect the simulations.

5.3 Datasets

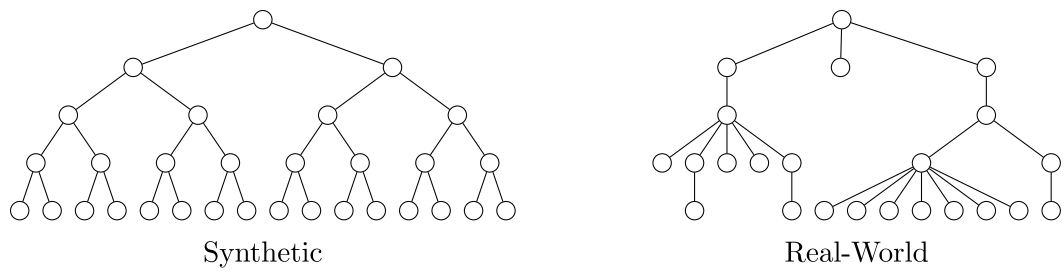


Figure 12: Excerpts of both datasets visualized

We use two datasets in our experiments. Each dataset resembles the content subtree of an Oak instance. The *synthetic* dataset is a complete binary tree of height 19, that is a binary tree in which all leaf nodes have depth 19 [1]. It contains $2^{20} \approx 10^6$ nodes, of which 50% are leaf nodes. Each node has a mean depth and degree of 18 and 2, respectively. The *real-world* dataset is based on DELL's website (http://dell.com) which is built on

[2]https://www.archlinux.org/about/

top of Adobe's AEM that uses Oak. [3] It contains $13 \cdot 10^6$ nodes, 65% of which are leaf nodes. Each node has a mean depth and fanout of 13.68 and 1, respectively. The tree's nodes have a max depth and degree of 24 and 1729, respectively. There are many nodes with a single child and few with many children. Figure 12 shows a sample subtree for each dataset.

5.4 Workload

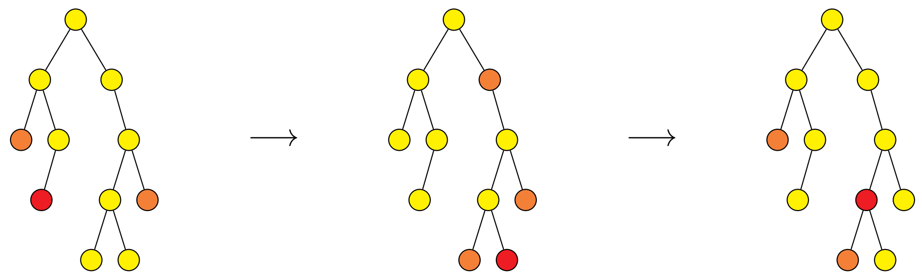


Figure 13: CMS Workloads visualized

In Section 2.1, we introduced the characteristics of a CMS workload. Using a running example we explained that a CMS workload is skewed, update-heavy, changing and CMSs frequently make use of a job-queuing system. For our experiments, we designed a workload that has the noted characteristics.

In order to have skew, we incorporate the Zipf distribution. The Zipf distribution picks a small subset of nodes more frequent than other nodes. Let N be the number of nodes the workload can draw from the dataset. We randomly and uniquely assign an integer $k \in [1, N]$ to each node, creating a $k \to$ node mapping. The probability of randomly drawing the k-th node from the dataset according to the Zipf distribution is

$$Zipf(k, N, s) = \frac{\frac{1}{k^s}}{\sum_{i=1}^{N} \frac{1}{i^s}}$$

The k-th node corresponds to the node which is assigned integer k in the mapping. Skewness s of the zipf distribution is parameterizable and we use $s = 1$ by default in our experiments. By having a greater skew, the subset of nodes which is frequently selected becomes smaller and the member nodes are selected more frequent. In Section 5.10, we make an experiment to compare the performance of GC and QTP under workloads with different skew.

A CMS workload is also update-heavy. We create many update operations before creating a query operation. We fix the defaul tupdate to query ratio to 10:1, 10 update operations are created for each query operations. In Section 5.11, we evaluate and compare GC and QTP under various update to query ratios.

Additionally, the workload we design has to be changing. We periodically permute the node mappings by reassigning a random and unique integer $k \in [1, N]$ to each

[3] https://wwwimages2.adobe.com/content/dam/acom/en/customer-success/pdfs/dell-case-study.pdf

node from the dataset in orderto change the hotspots of the simulation. We change the hotspot by default every 30 seconds. During a 5 minute experiment we expect 10 different workloads.

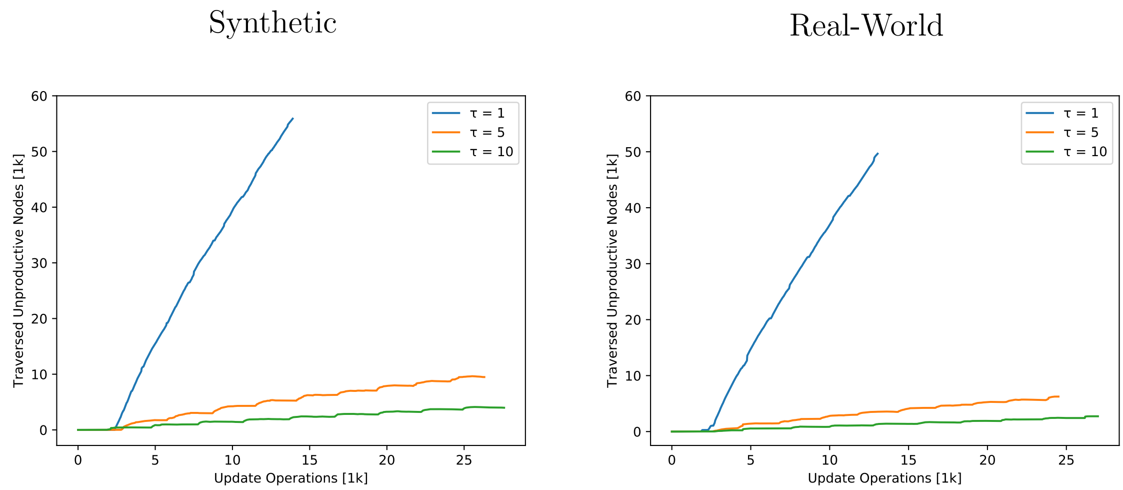
Lastly, the workload has to simulate a job-queuing system. An update operation executed during the experiment is composed from two actions. We first set the "pub" = "now" property-value pair to the node and then consecutively remove it. The actions simulate a node being inserted into the queue and then being processed and removed.

Figure 13 depicts heatmaps of a content subtree of Oak over time. These heatmaps show how often the designed workload selects specific nodes inside the content subtree. Red shaded nodes are the most frequently selected nodes.

5.5 Volatility Threshold τ

Volatility threshold τ determines after how many insertions/deletions of an index node it becomes volatile (see Definition 4). In this section, we study the impact of volatility threshold τ on unproductive nodes, which directly affect query runtime.

We hypothesize that an increase in τ yields a decrease to the number of traversed unproductive nodes during query execution under a CMS workload. If τ increases, it is less likely for a node to become volatile. Having less volatile nodes should cause a decrease to the number of unproductive nodes and consequently also query runtime in the CMS workload.



(a) (b)

Figure 14: Traversed Unproductive Nodes over update operations with threshold $\tau \in \{1, 5, 10\}$

Figures 14a and 14b depict thresholds $\tau \in \{1, 5, 10\}$ affecting the number of unproductive nodes traversed during query execution over update operations. We observe lower thresholds τ yielding in a steeper slope. A lower volatility threshold increases the likelihood of a node becoming volatile. The amount of volatile nodes also affect the

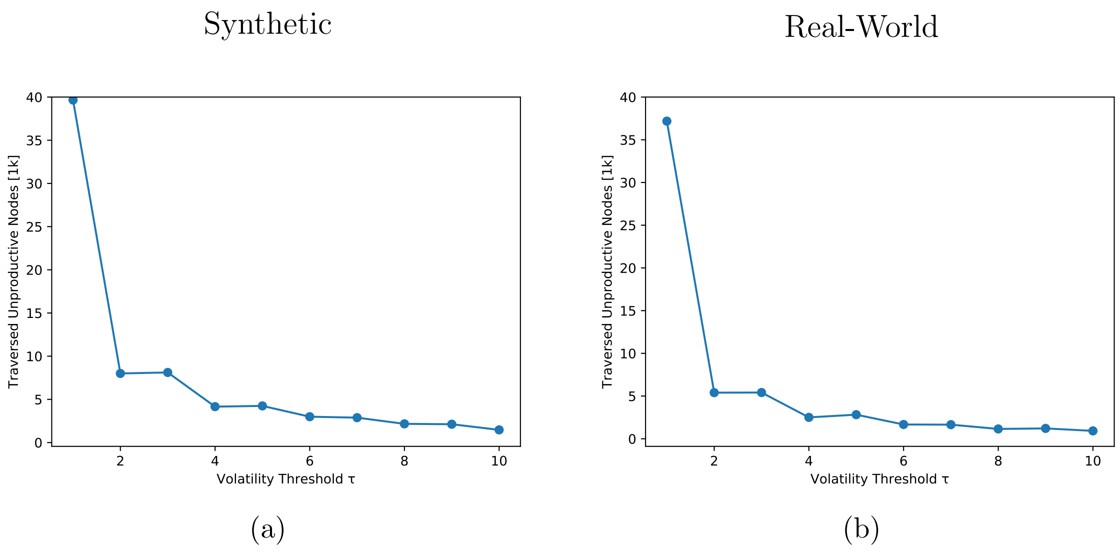


Figure 15: Traversed Unproductive Nodes over volatility threshold τ

number of unproductive nodes, since volatile nodes eventually stop being frequently updated and become unproductive. The increase in unproductive nodes in the index also affects query runtime because Oak has to traverse these nodes during query execution.

Figures 15a and 15b compare the number of traversed unproductive nodes during query execution over a range of thresholds. We observe a decrease in unproductive nodes while increasing threshold τ. As suggested earlier, a lower volatility threshold increases the amount of volatile nodes in the index and consequently also increases the number of unproductive nodes.

We see the two variables sharing a power law relationship. We believe the workload's skew to be responsible for the power law relationship. We see the query executor having to traverse $5k$ unproductive index nodes with a threshold of $\tau = 5$. By increasing the threshold to $\tau = 6$, the average number of traversed unproductive nodes does not decrease significantly because of the zipf-distribution's skewness. We believe the decrease in traversed unproductive nodes is bigger is bigger if the distribution is less skewed. Having less skew implies nodes being picked by the workload more uniformly.

Summarizing, all observations verify our hypotheses. Increasing volatility threshold τ decreases the number of unproductive nodes traversed which decreases query runtime. Increasing the volatility threshold causes less nodes to become volatile. Since we create less volatile nodes, we also reduce the number of unproductive nodes. Less unproductive nodes yield lower WAPI query runtimes.

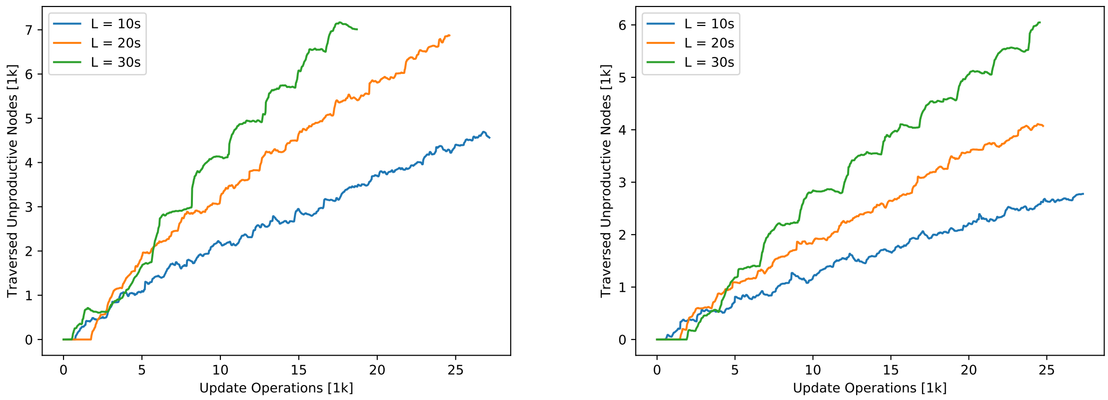
5.6 Sliding Window of Length L

Sliding Window of Length L determines the length of the recent workload that WAPI considers to compute an index node's volatility count. Greater values of L allow WAPI to count more updates and therefore increase the chances of a node becoming volatile. In this section, we study the effect of the sliding window on unproductive nodes. Less unproductive nodes in the index imply faster query execution.

We hypothesize that an increase in L yields an increase to the number of traversed unproductive nodes during query execution. If L increases, it is more likely for a node to become volatile, since more updates are considered towards the volatility count. Having more volatile nodes should imply an increase in unproductive nodes and consequently also query runtime in the CMS workload.

Synthetic

Real-World



(a)

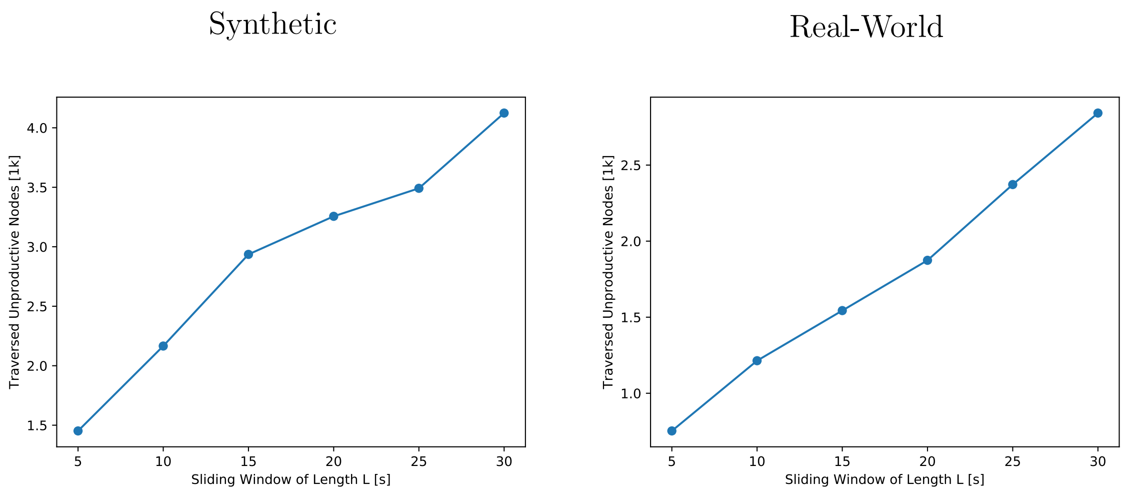
(b)

Figure 16: Traversed Unproductive Nodes over update operations with sliding window of length $L \in \{10s, 20s, 30s\}$

Figures 16a and 16b visualize the number of unprpductive nodes WAPI traverses during query execution for three distinct lengths $L \in \{10s, 20s, 30s\}$ of the sliding window with respect to update operations. As expected, the simulation with the longest sliding window ($L = 30s$) has the highest rate of growth of unproducive nodes. When L increases, more updates are considered towards a node's volatility count, hence more nodes become volatile and consequently unproductive, later on.

Figures 17a and 17b show the number of unproductive nodes the query executor has to traverse with respect to the sliding window length L. The two variables share a linear relationship. We observe greater sliding window lengths to cause an increase to the number of unproductive nodes traversed by WAPI during query execution. More volatile nodes imply an increase in unproductive nodes in the index.

Concluding, we see sliding window of length L to be affecting the rate of growth of unproductive nodes. Increasing L does increase the likelihood of an index node become volatile. More volatile nodes cause an increase in unproductive index nodes. Since the WAPI has to traverse more index nodes during query execution, its query runtime increases.

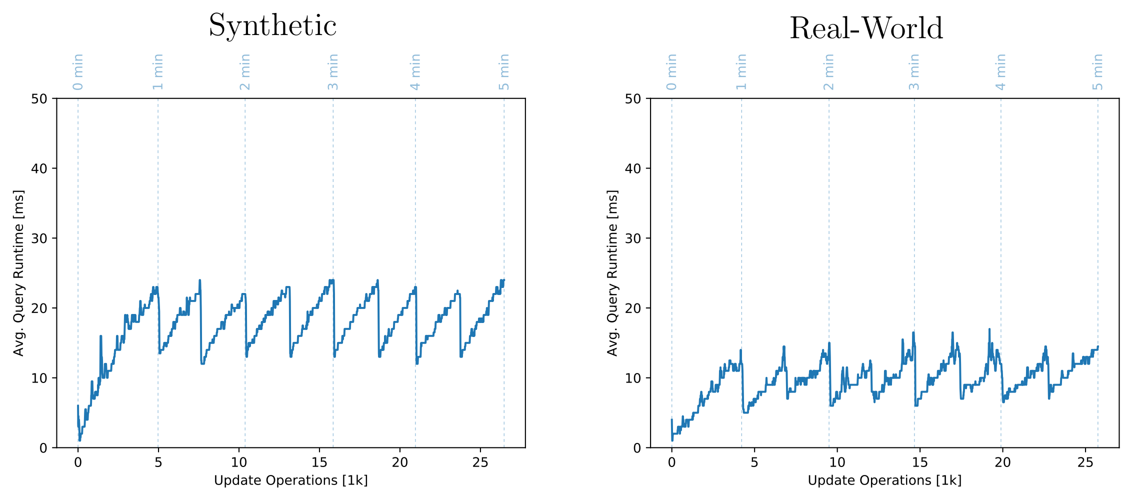


(a) (b)

Figure 17: Traversed Unproductive Nodes over sliding window length L

5.7 GC and QTP Query Performance

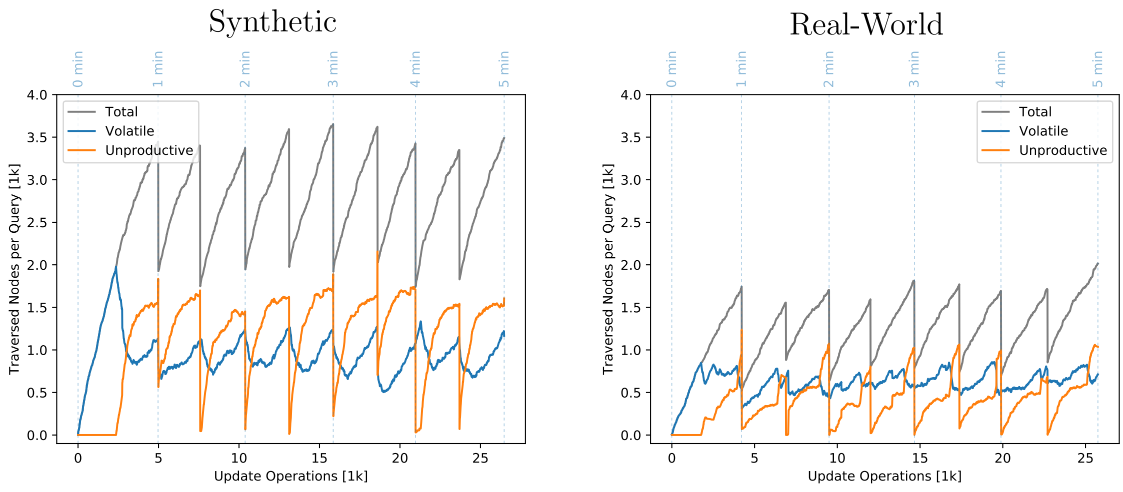
Our next experiment compares the query performance of periodic garbage collection and query-time pruning, when applied on Oak. We record the average query runtime and the index structure during query execution first while having GC exclusively enabled and then with QTP exclusively enabled.



(a) (b)

Figure 18: Query Runtime over update operations, periodic GC enabled

Figures 18a and 18b show the resulting decrease in query runtime when applying periodic GC on Oak. The garbage collector is run every 30 seconds. We observe the query runtime increase during the first minute of the simulation. When GC is run for the first time (at 30 seconds), it does not encounter any unproducitve nodes yet because the sliding window has a length of 30 seconds during the experiment. Afterwards, every succeding GC encounters and prunes unproductive nodes. The pruned nodes are made visible by the sawtooth wave depicted in the Figure. The query runtime oscillates between 20 and 30 milliseconds with a mean of 25 milliseconds on the synthetic dataset. The real world's query runtime oscillates between 8 and 18 milliseconds with a mean value of 13 milliseconds.



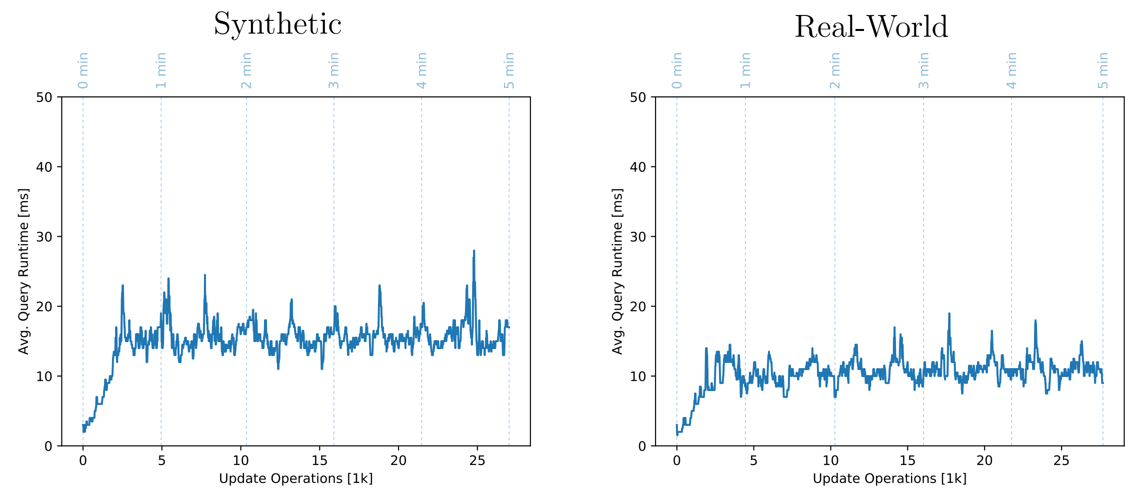
(a) (b)

Figure 19: Index composition over update operations, periodic GC enabled

Figures 19a and 19b visualize the index structure during query execution with periodic GC. We observe the unproductive nodes increase until GC is run, after which they are completely removed again. The traversed volatile nodes have a cycloid pattern, as seen and explained in Figures 4a and 4b. In contrast to the mentioned Figures, we don't see the number of volatile nodes having a downward trend from the 30 second mark till the end of the experiment. The number of unproductive nodes is near a constant throughout the experiment, hence the likelihood of becoming volatile does not change. Since the likelihood of node becoming volatile remains constant, there is no increase or decrease in volatile nodes, on average.

The cycloid pattern is created from the workload change and skewness. When the workload changes, we see a decrease in volatile nodes. Nodes need to reach the threshold in order to become volatile and few do, since the skew in the workload only picks a subset of nodes frequently. Before a new workload kicks in, we observe an increase in volatile nodes because many nodes are on the verge of becoming volatile and therefore need to picked fewer times by the workload.

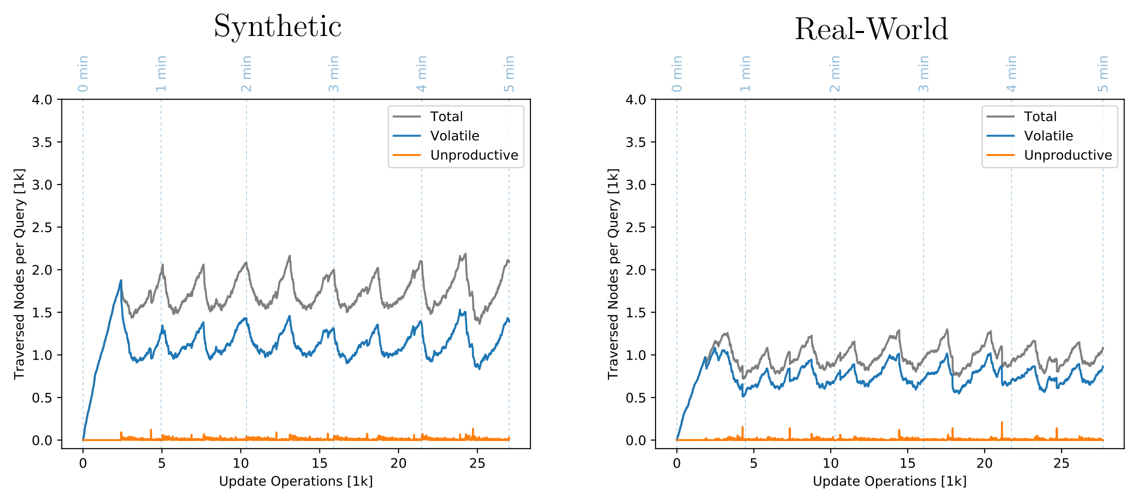
Comparing the synthetic dataset to the real-world one, we observe another phenomenon. The gap between traversed volatile and total traversed nodes is significantly bigger in the synthetic dataset. In other words, the ratio of trversed volatile over total traversed index nodes is higher in the real-world dataset. We suggest that non-volatile ancestors of volatile nodes account for the gap. As mentioned in Section 5.3, the real-world dataset has many nodes with a single child and few with many children, forming long chains of nodes. If the leaf node in such a chain is updated, each ancestor in the chain is also added or removed, given they are not volatile. In that scendario, when the leaf node becomes volatile, most likely its ancestors inside the chain also become volatile, since they were added or removed every time the leaf node did. It is more likely for an ancestor to become volatile in the real-world dataset due to the long chains, hence the higher ratio of traversed volatile over total traversed index nodes.



(a) (b)

Figure 20: Query Runtime over update operations, QTP enabled

Figures 20a and 20b depict the resulting decrease in query runtime when applying QTP on Oak. We see the runtime rise during the first 30 seconds and then see the runtime remain stable. We suggest that volatile nodes account for the query runtime.



(a) (b)

Figure 21: Index composition over update operations, QTP enabled

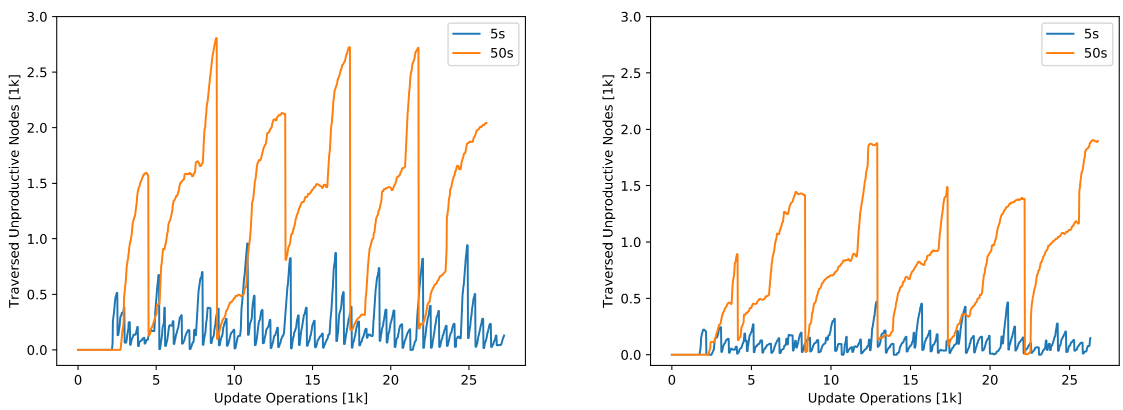
Figures 21a and 21b show the index structure with QTP enabled during query execution. We observe the number of unproductive nodes to be close to 0 throughout the entire simulation. We suggest that unproductive nodes should be near a constant at any time during the simulation because the index is continuously cleaned from the queries. As long as query execution is continuous, so is our index cleaning. Since we always query the root content node in our experiment, we clean the whole index subtree from unproductive nodes. As seen earlier, we also observe a bigger gap between traversed volatile and total traversed index nodes in the synthetic dataset compared to the real-world one due to more non-volatile ancestors of volatile nodes.

The workload during the experiment always query the subtree root node. It still remains open how QTP affects performance when the query filter changes. The question will be handled in Section 5.9, where we will see how the algorithm's performance changes when querying different ranges of nodes.

5.8 GC Period T

In this section we discuss the performance impact of GC period T. Oak can run GC arbitrary many times. Given our setup, we would like to find out what the optimal period of GC is. We run GC under varying period and compare the results.

Synthetic | Real-World



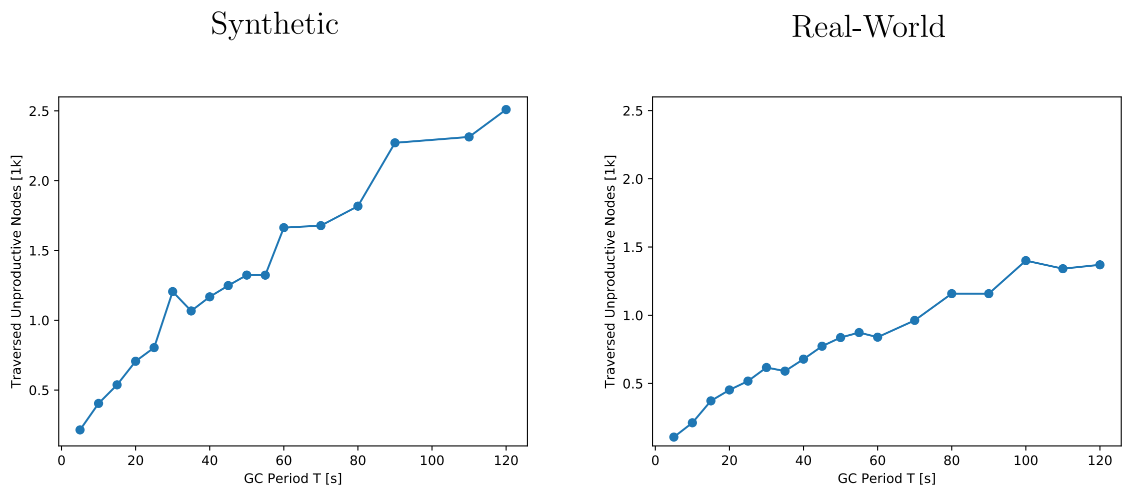
(a) | (b)

Figure 22: Traversed Unproductive Nodes over update operations with GC period $T \in \{5s, 50s\}$

5.9 QTP Queried Nodes

5.10 Workload Skew s

One of our initial assumptions was that Oak's workload is update heavy, amongst others. To emphasize the implications of such an update heavy workload, we conduct the follow-

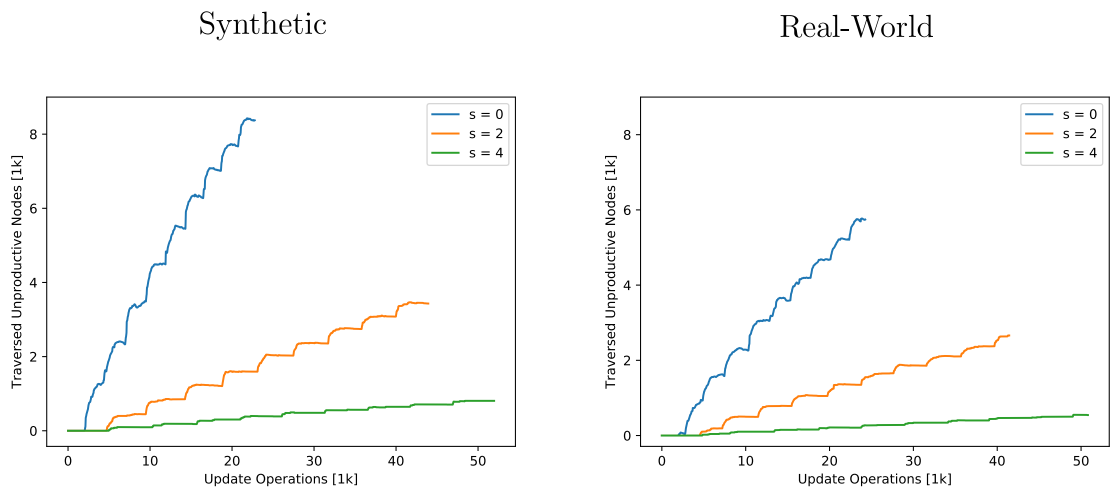


(a) (b)

Figure 23: Traversed Unproductive Nodes over GC period T

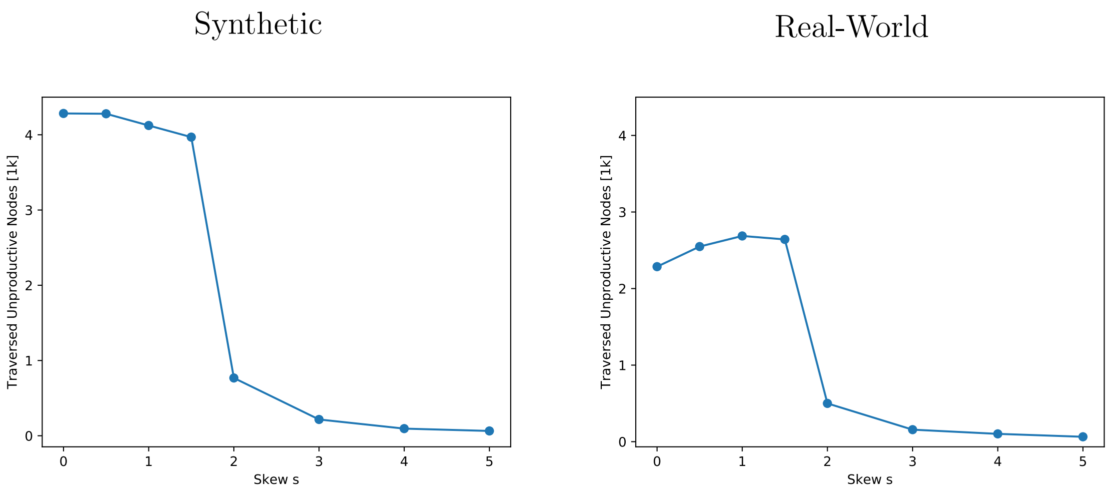
ing experiment. Oak is benchmarked with different update to query ratios, that is the number of updates executed between two consequtive queries, in order to show the effect of such a workload on unproductive nodes. As mentioned in Section 5.4, we use the Zipf distribution to model a skewed workload. If skew s increases, the group of frequently drawn nodes becomes smaller but the member nodes are drawn more frequent.

We expect skew s to affect the number of unproductive nodes WAPI traverses during query execution. Since some nodes are updated more often, more nodes become volatile and eventually also unproductive. On the other hand, increasing skew s also reduces the number of members of the group of frequently updated nodes and therefore reduce the number of volatile nodes, hence unproductive nodes also reduce.



(a) (b)

Figure 24: Traversed Unproductive Nodes over update operations with skew $s \in \{0, 2, 4\}$



(a) (b)

Figure 25: Traversed Unproductive Nodes over skew s

Figures 24a and 24b show the number of unproductive nodes traversed by WAPI during query execution with workloads having skew $s \in \{0, 2, 4\}$. The workload with skew $s = 0$ is uniformy distributed. We see a smaller skew s increase the number of unproductive nodes. We believe that even though nodes are more frequently updated, many more stop being updated and therefore the total number of nodes becoming volatile, and eventually unproductive, decreases.

Figures 25a and 25b offer a more detailed perspective by showing the traversed unproductive nodes with respect to skew s. We see a steep decrease in unproductive nodes when the skew is $s = 1.5$. We believe the hotspot reaches a point where it loses significantly more nodes than them becoming volatile, and eventually unproductive. From there on we see a sublinear rate of convergence. When $s \to \infty$, the hotspot only has a single member which gets exclusively updated, as proven below:

$$\begin{aligned}
\lim_{s\to\infty} Zipf(k, N, s) &= \lim_{s\to\infty} \frac{\frac{1}{k^s}}{\sum_{i=1}^{N} \frac{1}{i^s}} = \lim_{s\to\infty} \frac{\frac{1}{k^s}}{\frac{1}{1^s} + \sum_{i=2}^{N} \frac{1}{i^s}} \\
&= \frac{\frac{1}{k^\infty}}{\frac{1}{1^\infty} + \sum_{i=2}^{N} \frac{1}{i^\infty}} = \frac{\frac{1}{k^\infty}}{\frac{1}{1^\infty} + \sum_{i=2}^{N} \frac{1}{\infty}} \\
&= \frac{\frac{1}{k^\infty}}{\frac{1}{1} + \sum_{i=2}^{N} 0} = \frac{\frac{1}{k^\infty}}{1 + 0} \\
&= \frac{1}{k^\infty}
\end{aligned}$$

$$\text{for } k = 1: \quad \frac{1}{k^\infty} = \frac{1}{1^\infty} = 1$$

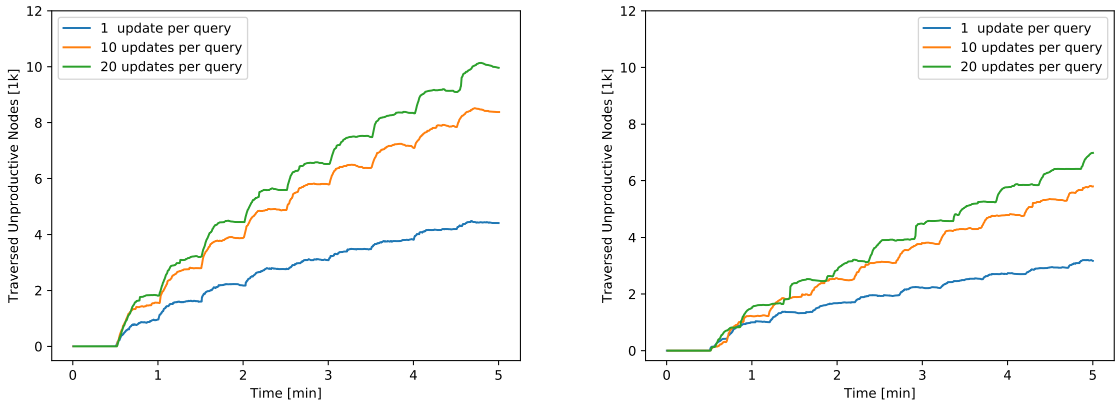
$$\text{for } k > 1: \quad \frac{1}{k^\infty} = \frac{1}{\infty} = 0$$

No other nodes besides the node with $k = 1$ gets drawn from the Zipf distribution when $s \to \infty$.

We also see another interesting phenomenon. Comparing the synthetic to the real-world dataset, we see the number of traversed unproductive nodes decrease in the interval $s \in [0,1]$, whilst it increases during the same interval in the real-world dataset. The synthetic dataset produces the most unproductive nodes with no skew ($s = 0$). We conjecture if the absence of long chains of nodes and presence of many non-volatile ancestors of volatile nodes in the synthetic dataset are responsible for the observed phenomenon.

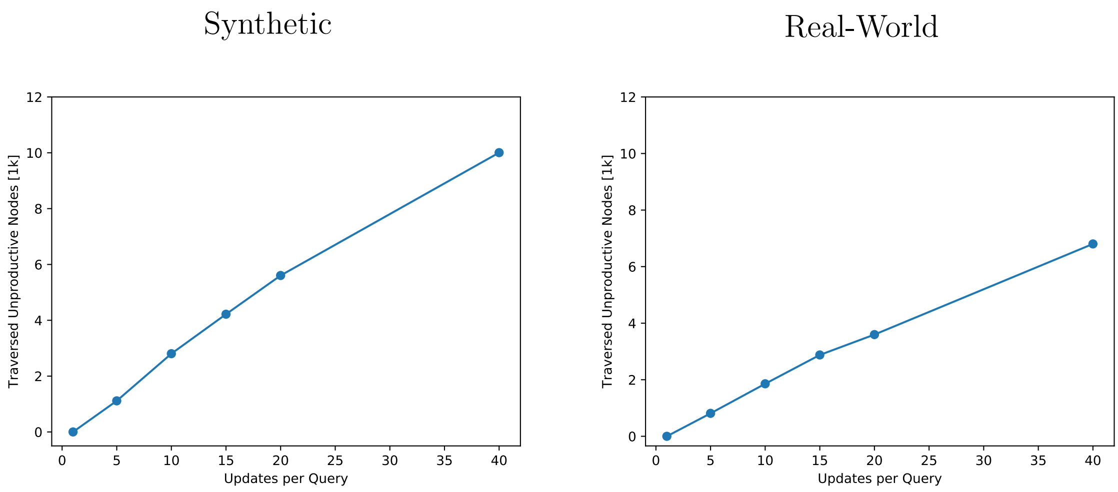
5.11 Update to Query ratio

Synthetic Real-World



(a) (b)

Figure 26: Traversed Unproductive Nodes over time for 1,10 and 20 updates per query



(a) (b)

Figure 27: Traversed Unproductive Nodes over updates per query

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

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

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

6 Appendix

```java
/**
 * Returns an iterable which lazily traverses a subtree rooted at root
 * in postorder.
 *
 * @param root: Root node of subtree we apply traversal on
 * @returns: Iterable for postorder tree walk
 */
Iterable<Tree> postOrder(Tree root) {
    return () -> {
        /* Stacks */
        Deque<Tree> s1 = new LinkedList();
        Deque<Tree> s2 = new LinkedList();
        s1.push(root);
        return new Iterator<Tree>() {
            @Override
            public boolean hasNext() {
                return s1.size() > 0 || s2.size() > 0;
            }
            @Override
            public Tree next() {
                while (s1.size() > 0 && (
                    s2.size() == 0 ||
                        isAncestor(
                            s2.peek().getPath(),
                            s1.peek().getPath()
                        )
                )) {
                    Tree n = s1.pop();
                    s2.push(n);
                    for (Tree child : n.getChildren()) {
                        s1.push(child);
                    }
                }
                return s2.pop();
            }
        };
    };
}
```

Figure 28: `postOrder()` implementation in Java

```java
/**
 * Higher order function that applies func to each element of iterable.
 *
 * @param func: The function to apply on each element of iterable
 * @param iterable: The iterable func is applied on
 * @returns: An iterable with the resulting elements of the application
 */
Iterable<R> map(Function<T,R> func, Iterable<T> iterable) {
    return () -> {
        Iterator<T> iterator = iterable.iterator();
        return new Iterator<R>() {
            @Override
            public boolean hasNext() {
                return iterator.hasNext();
            }
            @Override
            public R next() {
                return func.apply(iterator.next());
            }
        };
    };
}
```

Figure 29: `map()` implementation in Java

```java
/**
 * Higher order function that removes all elements from an iterable
 * not satisfying the predicate.
 *
 * @param predicate: The predicate that tests elements
 * @param iterable: The iterable whose members are tested against
 * @returns: An iterable with members satisfying the predicate
 */
Iterable<T> filter(Predicate<T> predicate, Iterable<T> iterable) {
    return () -> {
        Iterator<T> iterator = iterable.iterator();
        T n = null;
        return new Iterator<T>() {
            @Override
            public boolean hasNext() {
                nextIfNeeded();
                return n != null;
            }
            @Override
            public T next() {
                nextIfNeeded();
                T tmp = n;
                n = null;
                return tmp;
            }
            @Override
            private void nextIfNeeded() {
                while (n == null && iterator.hasNext()) {
                    T candidate = iterator.next();
                    if (predicate.test(candidate)) {
                        n = candidate;
                    }
                }
            }
        };
    };
}
```

Figure 30: `filter()` implementation in Java