Department of Informatics, University of Zürich

BSc Thesis

An Adaptive Index for Hierarchical Distributed 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

Abstract

Unproductive nodes in hierarchical indexes waste space and slow down queries. We propose periodic garbage collection and query time pruning in order to clean the index. We implement our techniques in Apache Jackrabbit Oak and provide extensive experimental evaluation to stress test the algorithms and show that the database throughput increases considerably if we apply garbage collection to the index.

Zusammenfassung

Unproduktive Knoten in hierarchischen Indizes verschwenden Speicherplatz und verlangsamen die Abfragen. Wir schlagen periodische Indexreinigung und Abfragezeitbereinigung vor, um den Index zu bereinigen. Wir implementieren unsere Techniken in Apache Jackrabbit Oak und bieten eine umfangreiche experimentelle Auswertung, um die Algorithmen zu Stress-testen und zu zeigen, dass der Datenbankdurchsatz erheblich zunimmt, wenn wir den Index reinigen.

Contents

1 Introduction **6**

2 Background **7**
2.1 Apache Jackrabbit Oak (Oak) 7
2.2 Application Scenario 7
2.3 Workload Aware Property Index (WAPI) 8

3 Unproductive Nodes **12**
3.1 Introduction 12
3.2 Impact on Query Runtime 13

4 Cleaning Unproductive Nodes **16**
4.1 Periodic Garbage Collection (GC) 16
4.2 Query Time Pruning (QTP) 20

5 Experimental Evaluation **25**
5.1 Goals 25
5.2 Setup 25
5.3 Datasets 25
5.4 Workload 25
5.5 Experiments 26
5.5.1 Volatility Threshold τ 26
5.5.2 Sliding Window of length L 29
5.5.3 GC Periodicity 29
5.5.4 QTP Queried Nodes 32
5.5.5 Workload Skew 32
5.5.6 Update to Query ratio 32

6 Appendix **32**

List of Figures

1 An instance of an hierarchical database 9
2 Volatile nodes becoming unproductive 11
3 Query Runtime over time 13
4 Index composition during Query Execution 14
5 Node Ratio during Query Execution 15
6 Garbage collection applied on Oak 17
7 Java implementation of periodic garbage collection 18
8 Query Runtime with periodic GC 19
9 Index composition with periodic GC 19
10 Query Time Pruning applied on Oak 21
11 Java implementation of QTP 22
12 Query Runtime with QTP 23
13 Index composition with QTP 23
14 Impact of Volatility Threshold τ on Query Runtime 27
15 Impact of Volatility Threshold τ on Unproductive Nodes 28
16 Impact of Sliding Window of length L on Query Runtime 30
17 Impact of Sliding Window of length L on Unproductive Nodes 31
18 `DFS()` implementation 33
19 `map()` implementation 34
20 `filter()` implementation 35

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 deteriorate 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 deteriorate update performance since concurrency control protocols need to resolve the conflict.

Wellenzohn et al. [5] propose the Workload-Aware Property Index (WAPI). The WAPI exploits the workloads' 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.

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 neither the node itself nor any of its descendants contain an indexed property 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.

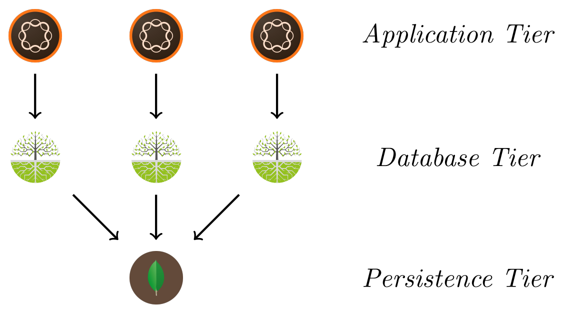
Wellenzohn et al. [5] propose periodic Garbage Collection (GC), which traverses the entire index subtree and prunes all unproductive index nodes at once. Additionally we propose Query-Time Pruning (QTP), an incremental approach to cleaning up unproductive nodes in the index. The idea is to turn queries 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 [5] in the open-source hierarchical distributed database Apache Jackrabbit Oak (Oak).

2 Background

2.1 Apache Jackrabbit Oak (Oak)

Oak is a hierarchical distributed database system which uses a hierarchical index for efficient query processing. Multiple transactions can work concurrently by making use of Multiversion Concurrency Control (MVCC) [4], a commonly used optimistic concurrency control technique [3].

Section 2.1 depicts Oak's data-sharing architecture. Oak embodies the *Database Tier*. Multiple Oak instances can operate concurrently. Whilst Oak is responsible for handling the database logic, it stores the actual data on MongoDB[1], labeled as *Persistence Tier*. On the other end, applications can make use of Oak as shown in Section 2.1 under *Application Tier*. One such application is Adobe's enterprise content management system (CMS), the Adobe Experience Manager.[2]



2.2 Application Scenario

Content management systems (CMSs) that make use of Oak have specific workloads. These workloads have distinct properties: they are *skewed*, *update-heavy* and *changing* [5]. CMSs frequently use a job-queuing system that has the noted characteristics.

Consider a social media feed as a running example. Some posts are more popular and have more interactions than others. Many people write comments or leave a like on these popular posts, therefore implying a skewed workload. Users submit new posts or interact with existing posts by writing comments for example, thus 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. For example, if the post had an image, the CMS needs to compress the image and create a thumbnail. A job is created for processing the uploaded image. 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

[1]https://www.mongodb.com/what-is-mongodb
[2]http://www.adobe.com/marketing-cloud/experience-manager.html

that the specific node is a pending job that needs processing. A node has the "pub" property signals when the job has to be published. We set the value to "now" to signal that the job can be published immediately. Once the background thread detects the node and finishes processing, it removes the "pub" property and successfully publishes the post.

From now on, we shall refer to a workload with the properties mentioned above as a *CMS workload*.

2.3 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 content 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".

The WAPI is an hierarchical index which indexes the properties of nodes in order to answer CAS queries efficiently. The WAPI is hierarchically organized under node `/i` (denoted *Index Subtree Root* in Figure 1). The second index level consists of all properties k we want to index, such as "pub" in Figure 1. The third index level contains any values v of property k, for example "now" in Figure 1. The remaining index 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 $path(*n) = \texttt{m} = \texttt{/}\lambda_1\texttt{/} \ldots \texttt{/}\lambda_d$.

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

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

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

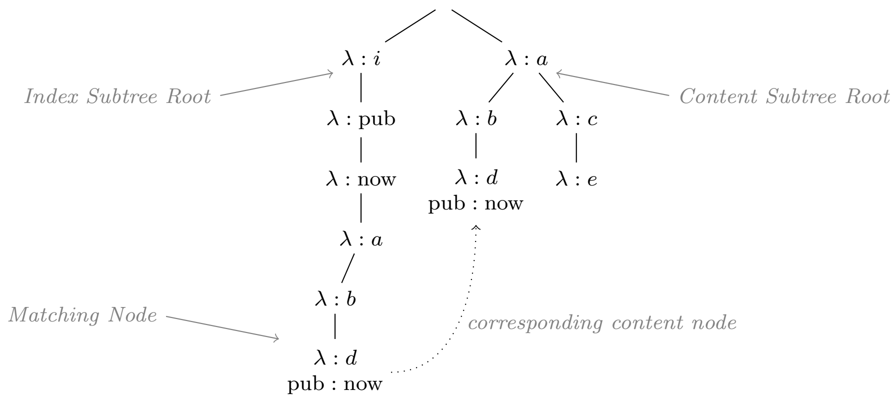


Figure 1: An instance of an hierarchical database

From the CMS workload described in Section 2.2 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 0.

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. [5] 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 $H_i = \langle \dots, G^a, G^b, G^c \rangle$ 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.

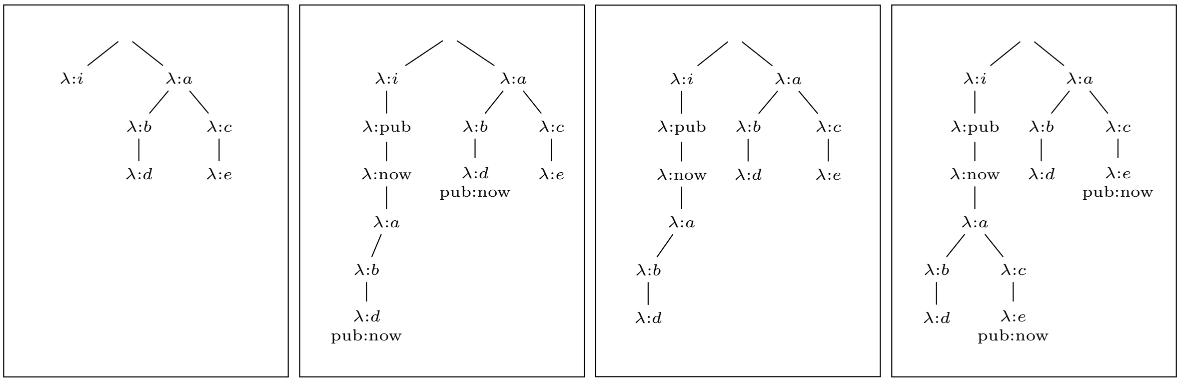
$$\begin{aligned} vol(n) = |\{G^b | G^b \in H_i \wedge t(G^b) \in [t_{n-L+1}, t_n] \wedge \exists G^a [\\ G^a = pre(G^b) \wedge ([n^a \notin N(G^a) \wedge n^b \in N(G^b)] \vee \\ [n^a \in N(G^a) \wedge n^b \notin N(G^b)])]\}| \end{aligned}$$

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 $L = 1$ and history $H_h = \langle G^0, G^1, G^2, G^3 \rangle$. Oak instance O_h executes transactions $T_1, \ldots, T_3$. 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 $t(G^2) = t + 2$. The index nodes are not pruned during T_2 since they are volatile. The index nodes were added during the last snapshot and therefore have a volatility count of 1. Since the threshold is $\tau = 1$, the nodes are considered volatile.

$$G^0 \xrightarrow{T_1} G^1 \xrightarrow{T_2} G^2 \xrightarrow{T_3} G^3$$



Snapshot G^0	Snapshot G^1	Snapshot G^2	Snapshot G^3
$t(G^0) = t$	$t(G^1) = t + 1$	$t(G^2) = t + 2$	$t(G^3) = t + 3$

Given $\tau = 1$, $L = 1$, 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.

Figure 2: Volatile nodes becoming unproductive

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 τ. When time passes, insertions and deletions are not in the sliding window anymore, causing the volatility count to drop. If the volatility count drops below threshold τ, the node ceases to be volatile and becomes unproductive. Unproductive index nodes slow down queries as traversing an unproductive node is useless, because neither the node itself nor any of its descendants contain an indexed property 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 quickly accumulate in the index and the query performance deteriorates over time [5].

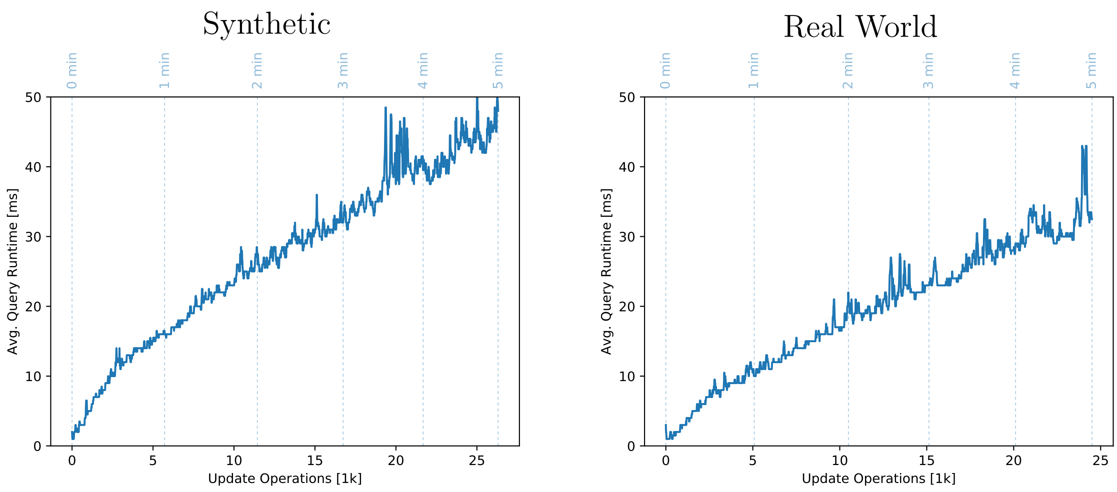
Definition 5 (Unproductive Node)**.** Index node n is unproductive iff n, and any descendant of n, is neither matching (see Definition 2) nor volatile (see Definition 4), i.e

$$unproductive(n) \iff \nexists m(m \in (\{n\} \cup desc(n)) \wedge (matching(m) \vee volatile(m)))$$

Example 4. 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 ($\tau = 1, L = 1$). 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 are below the threshold. The sliding window has length $L = 1$, 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 $vol($`/i/pub/now/a/b/d`$) = vol($`/i/pub/now/a/b`$) = 0$. Since the threshold is $\tau = 1$, the nodes are not volatile. In addition to not being volatile, they are not matching either, therefore they are unproductive according to Definition 5. Index node `/i/pub/now/a` is not unproductive in snapshot G^3 since it has two volatile descendants (`/i/pub/now/a/c/e`, `/i/pub/now/a/c`) and one of them is additionally matching.

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 an non volatile index node has a volatile descendant and the descendant ceases to be volatile, the index node also becomes unproductive, although it never was 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`. The query has to traverse two index nodes `/i/pub/now/a/b/d` and `/i/pub/now/a/b` which are useless and slow down the query because they 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. We hypothesize that unproductive nodes significantly slow down queries under a CMS workload. 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. An index under a CMS workload (see Section 2.2) is dominated by unproductive nodes, that is unproductive nodes constitute a large percentage of all index nodes.

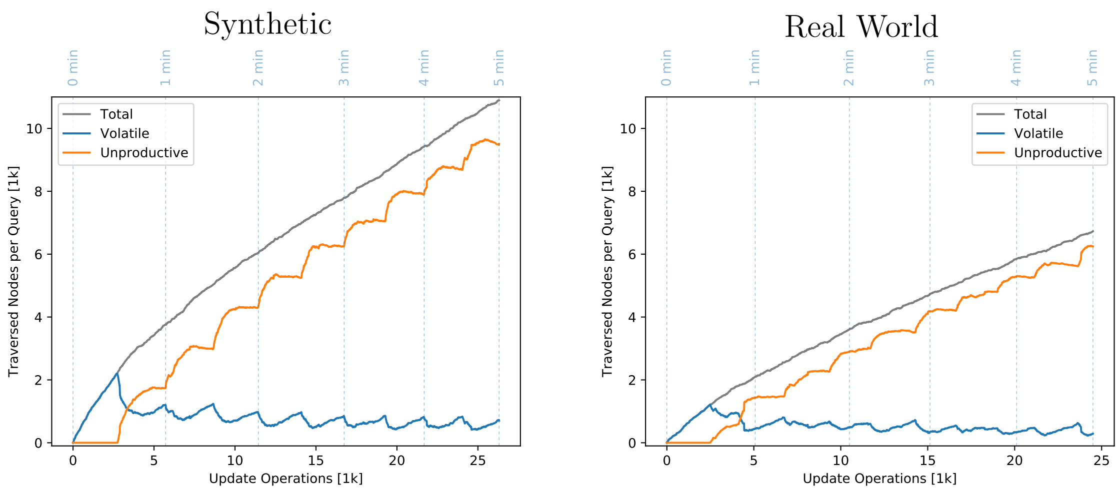
We summarize the statements above into the following hypotheses:

H_1: WAPI's average query runtime increases over time under a CMS workload.

H_2: Unproductive nodes significantly affect WAPI's query runtime under a CMS workload.

In order to find supporting evidence for the hypotheses above, we conduct a series of experiments on Oak. The experimental evaluation and datasets will be described 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 ($2.6 \cdot 10^4$ update operations) on the synthetic dataset and an increase from $2ms$ to $35ms$ ($2.5 \cdot 10^4$ update operations) on the real-world dataset.



(a) (b)

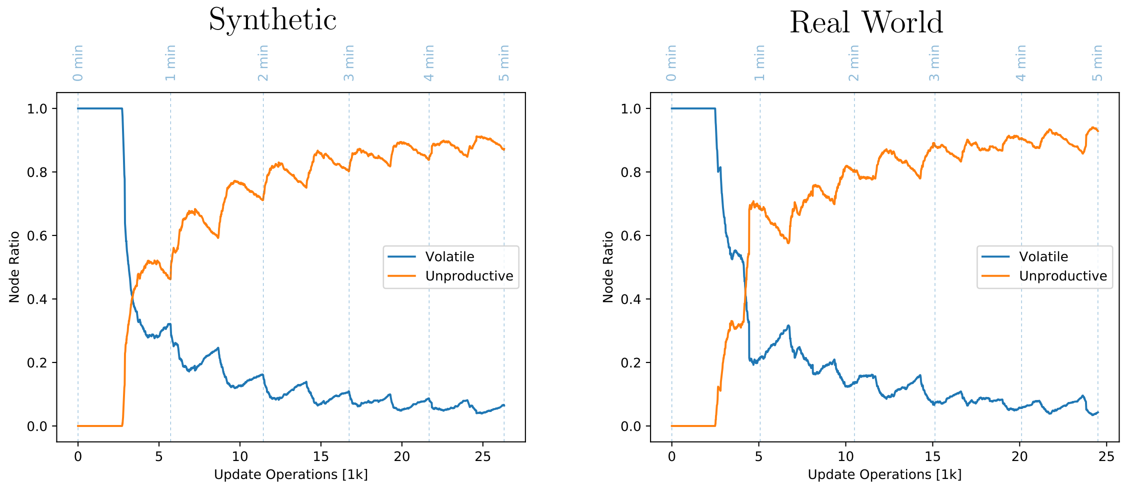
Figure 4: Index composition during Query Execution

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 and, most likely, become unproductive after some time and are not pruned from the index anymore. Therefore, the probability of picking a 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.

Furthermore, we see the number of volatile nodes descend after the 30 second mark. We believe it becomes less likely for nodes to become volatile, as time passes. When a workload randomly picks a node whose index node is unproductive, the index node becomes matching but was not added, thus the volatility count of the node does not increment. In comparison, if the index node would not exist, the created index node's volatility count would have been incremented. We infer that it is less likely for an unproductive node to become volatile again 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.

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, our queries encounter unproductive nodes. From that point, we observe a steep increase in traversed unproductive nodes. After 1 minute, we observe the traversed nodes being dominated by unproductive nodes. The rate of growth of traversed



(a) (b)

Figure 5: Node Ratio during Query Execution

unproductive nodes seems to decrease over time, which can be explained by the same rationale that cause the decrease of the rate of growth of total index nodes.

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 increase in unproductive 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 fewer times to become volatile in comparison to the time shortly after the workload kicked in.

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 skewed workload. Having a smaller chance to be picked by the workload implies having less volatile and unproductive nodes in the index.

Figures 5a to 5b show the ratio of volatile over total and 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. The data shows that unproductive nodes account for over 80% of the traversed nodes whilst less than 20% are volatile on the synthetic dataset after 5 minutes. Similarly, on the real-world dataset, we observe 90% traversed unproductive and 10% traversed volatile index nodes.

Concluding, the data strongly supports our hypotheses. We see the query runtime increase by an order of magnitude after 5 minutes. We also see unproductive nodes, which dominate the index, being accountable for the increase in query runtime. In the following sections we present two ways dealing with unproductive nodes.

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 clean the index periodically. In the following two subsections, we suggest two different approaching for dealing with unproductive nodes. We will empirically investigate their performance in Section 5.

4.1 Periodic Garbage Collection (GC)

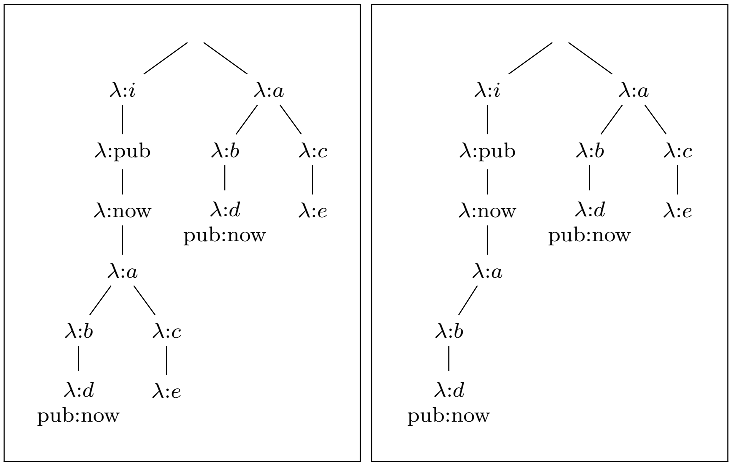
First, we propose to clean the index periodically. Using GC, we can guarantee that all unproductive nodes are removed from the index. Since GC has to be explicitly called, GC takes resources from the system. Executing GC too often may decrease system performance. Oak has a number of background processes that maintain the database. We add a background job that periodically executes garbage collection on Oak.

Algorithm 1 takes an index node n as parameter and prunes all its unproductive descendants. The algorithm traverses the subtree rooted at n depth-first. If a descendant $d \in desc(n)$ has no children, is not matching and not volatile, it is pruned from the index. The depth-first tree traversal ensures that the algorithm prunes a child before its parent node. This guarantees that all unproductive nodes are pruned.

Algorithm 1: CleanIndexWAPI

Data: Index node n
for *node* $d \in desc(n)$ *in DFS* **do**
  if $chd(d) = \emptyset \land \neg matching(d) \land \neg volatile(d)$ **then**
    delete node d

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



Snapshot G^0 Snapshot G^1

Assume nodes `/i/pub/now/a/c/e` and `i/pub/now/a/c` are unproductive in snapshot G^0. Transaction T_1 is executed by the periodic garbage collector and committs the resulting snapshot G^1.

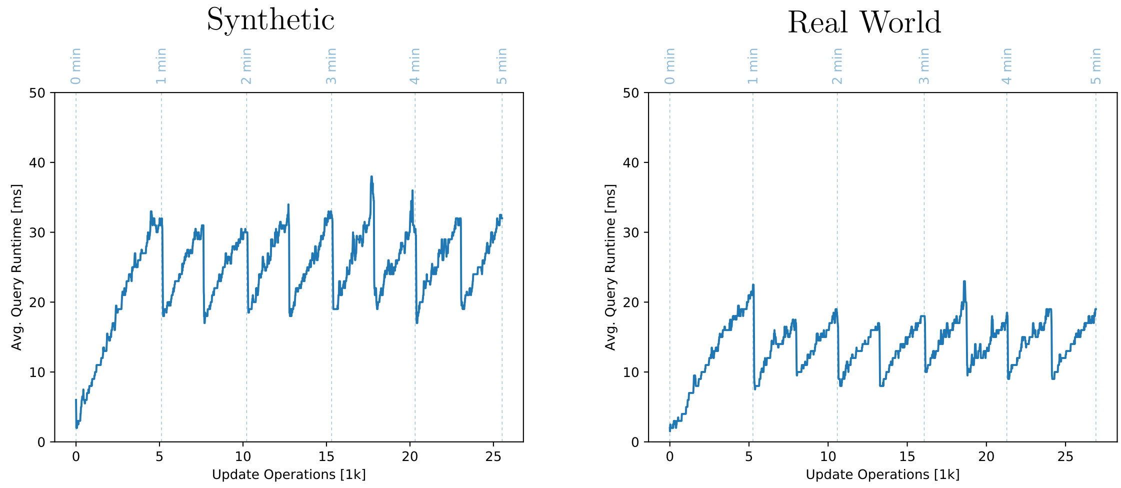
Figure 6: Garbage collection applied on Oak

Example 6 (Periodic GC)**.** Consider Figure 6. Assume index nodes `/i/pub/now/a/c/e` and `/i/pub/now/a/c` are unproductive in snapshot G^0. Oak's background thread executes a periodic garbage collection inside transaction T_1. During T_1, the index is traversed in depth-first order. The first unproductive node we encounter and prune is `/i/pub/now/a/c/e`. Next, the garbage collector encounters and prunes `/i/pub/now/a/c`. No further unproductive node is left for pruning. We see the index after garbage collection in snapshot G^1.

```java
/**
 * Removes any unproductive descendant of index node n.
 *
 * @param n: index node whose descendants are being cleaned
 */
void cleanIndexWAPI(Tree n) {

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

            /* DFS tree traversing iterable */
            DFS(n)
    ) {
        unproductiveNode.remove();
    }
}
```

Figure 7: Java implementation of periodic garbage collection

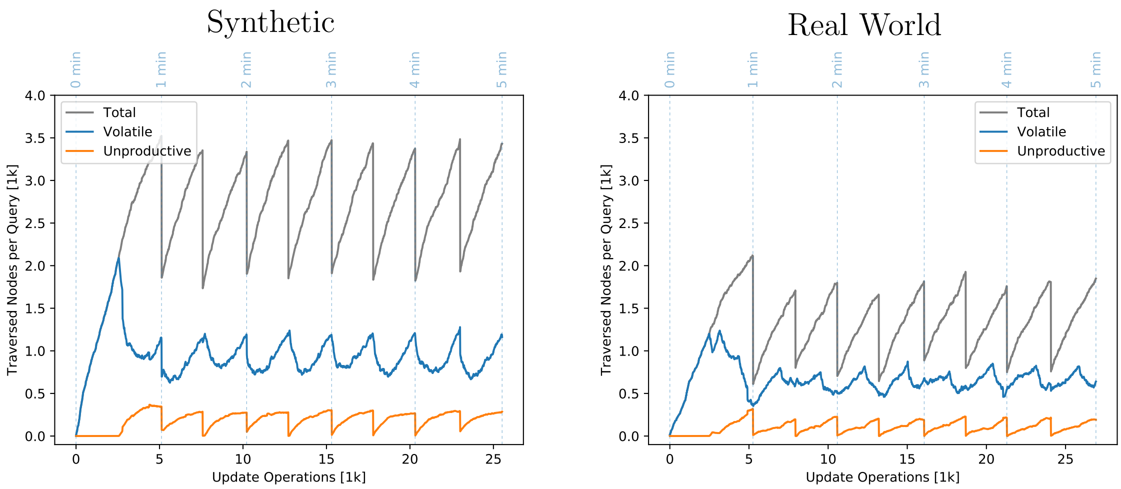
Figure 7 shows the implementation of the periodic garbage collection task in Java. Calling `DFS(n)` returns a lazy sequence of nodes which correspond to a depth-first tree traversal of the subtree rooted at node `n`. Next, any node that has children, is matching or volatile is filtered from the sequence since they are productive. All other nodes are pruned from the index.



(a) (b)

Figure 8: Query Runtime with periodic GC

Figures 8a and 8b 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 increasing during the first minute of the simulation. Afterwards, we observe query runtime having a sawtooth wave due to periodic garbage collection.



(a) (b)

Figure 9: Index composition with periodic GC

Figures 9a and 9b show the index structure during query execution with periodic GC. We observe the unproductive nodes increase until GC is run, after which they are removed again. The traversed volatile nodes have a cycloid pattern, as seen in Figures 4a and 4b, but they are not decreasing over time. Since the number of unproductive nodes is near a constant throughout the experiment, the likelihood of becoming volatile does not change. 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. We also observe the total number of traversed nodes to be significantly different between the two datasets although the difference in volatile nodes is not so big.

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 clears unproductive nodes whilst Oak answers queries. By doing so, we benefit by avoiding the cost of explicitly traversing the index in order to clean it.

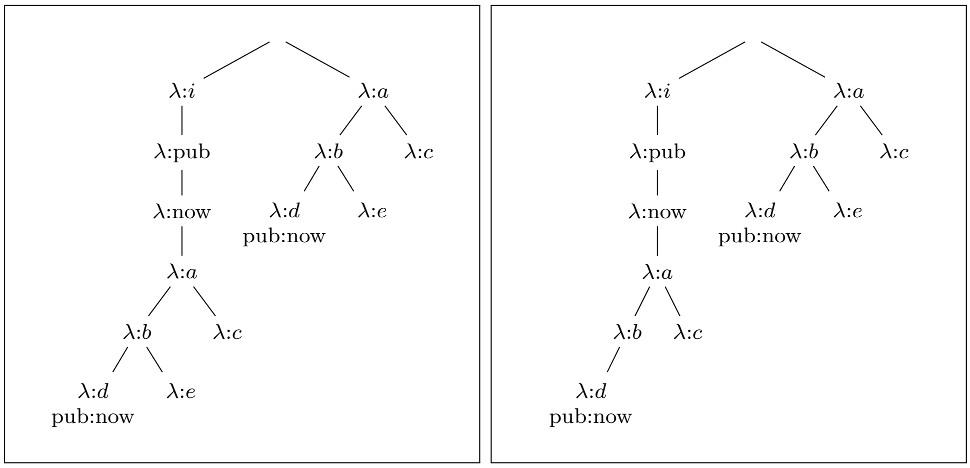
Algorithm 2: QueryQTP

Data: Query $Q(k, v, m)$, where k is a property, v a value and m a content node's path.
Result: A set of nodes satisfying $Q(k, v, m)$

$r \longleftarrow \emptyset$
$n \longleftarrow$ `/i/k/v/m`
for *node* $d \in desc(n)$ *in DFS* **do**
  if $matching(d)$ **then**
    $r \longleftarrow r \cup \{*d\}$
  else if $chd(d) = \emptyset \wedge \neg volatile(d)$ **then**
    delete node d
return r

Algorithm 2 takes a CAS query $Q(k, v, m)$ as an argument, where k is a property, v a value and m a content node's path. We declare set r and assign it the empty set. r will hold all paths satisfying the CAS query. We declare node n and assign it the index node corresponding to the content node with path m. Next, we traverse any descendant d of n in depth-first order. If node d is matching, we add its corresponding content node's path to r and proceed to the next descendant. If node d is not matching, has no children and is not volatile, it is deleted from the index. After finishing traversing the subtree rooted at n, we return the result set r.

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



Snapshot G^0 Snapshot G^1

Assume nodes `/i/pub/now/a/b/e` and `i/pub/now/a/c` are unproductive in snapshot G^0. Transaction T_1 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 committs the resulting snapshot G^1. QTP is used during query execution.

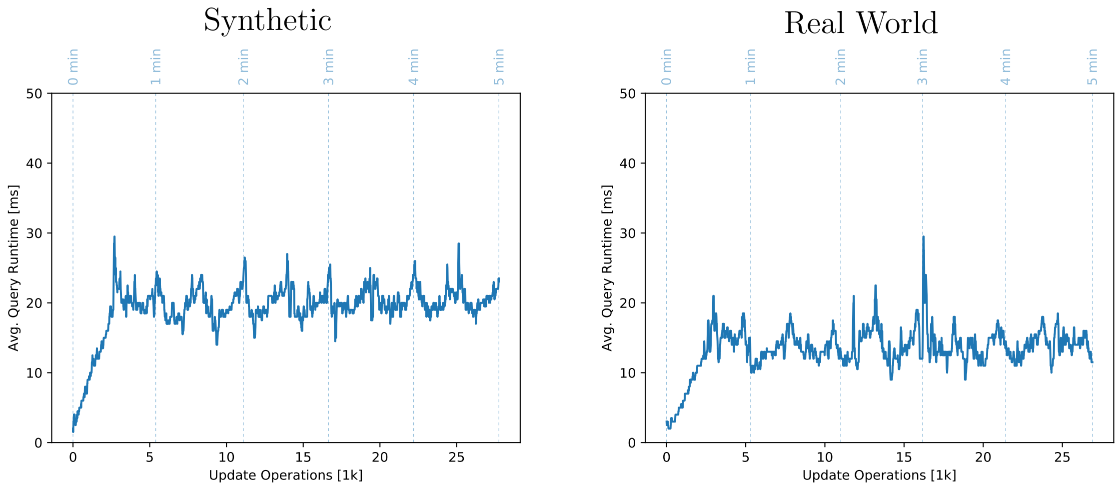
Figure 10: Query Time Pruning applied on Oak

Example 7 (QTP)**.** Consider Figure 10. Transaction T_1 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^0. 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 nodes are not traversed during query execution, they are not pruned and remain in the index unproductive. The resulting snapshot G^1 is committed by T_1 after finishing query execution.

```
/**
 * Answers a CAS Query and prunes traversed unproductive index nodes.
 *
 * @param k: Property
 * @param v: Value
 * @param m: path of content node in which we execute the query
 * @param root: Oak root
 * @returns an iterable with content paths satisfying the CAS query
 */
Iterable<String> QueryQTP(String k, String v, String m, Root root) {

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

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

    Tree n = root.getTree(targetNodePath);

    /* map index nodes' path to corresponding content nodes' path */
    return map((Tree d) -> relativize(indexRootPath, d.getPath()),

        /* filter non-matching index nodes */
        filter((Tree d) -> {
                /* prune if no children, not matching and not volatile */
                if (d.getChildrenCount(1) == 0 &&
                    !isMatching(d) &&
                    !isVolatile(d)
                ) {
                    d.remove();
                }
                return isMatching(d);
            },

            /* DFS tree traversal of descendants of n */
            DFS(n)
    );
}
```

Figure 11: Java implementation of QTP

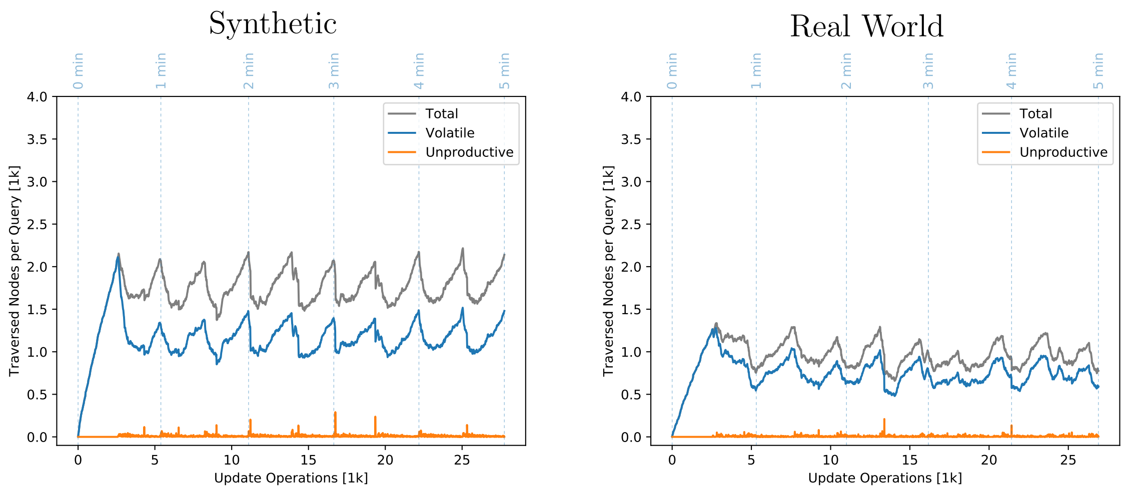
Figure 11 shows the java implementation of QTP in Oak. The algorithm takes a property `k`, value `v`, path `m`, a `Root` and returns a lazy sequence of content node paths satisfying the CAS query $Q(\mathtt{k}, \mathtt{v}, \mathtt{m})$. We first find index node n. By Calling `DFS(n)` we get a sequence of nodes representing a depth-first search in the subtree rooted at n. We filter any node that is not matching from the sequence. If a node has no children, is not matching and is not volatile, we prune it from the index. next, we take the filtered sequence of matching index nodes and map them to their corresponding content nodes' path.



(a) (b)

Figure 12: Query Runtime with QTP

Figures 12a and 12b 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.



(a) (b)

Figure 13: Index composition with QTP

Figures 13a and 13b 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 throughout the simulation because we clean the index during query execution. Since query execution is continuous, so is our garbage collection. Since we always query the root content node in our experiment, we apply garbage collect to the whole tree. We will see in Section 5.5.4 how the algorithm's performance changes when querying different ranges of nodes. We also see the gap between total and volatile nodes to be greater in the synthetic dataset.

5 Experimental Evaluation

5.1 Goals

Using the experiment, we wish to learn more about how different parameters affect the performance of periodic garbage collection and query time pruning. We initially 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 growth of volatile nodes and therefore can affect the production of unproductive nodes in the index. Next, we investigate the effects periodicity of garbage collection has on its algorithm's performance. Executing GC more often might increase system performance since more unproductive nodes are pruned. On the other hand side, executing GC too often might decrease performance. We will investigate how different query workloads affect the overall performance of QTP. We also conduct an experiment 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.

5.2 Setup

All experiments are conducted on a virtual machine on a 15" Macbook Pro 2015. 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 create noise during the simulations.

5.3 Datasets

For our experiments we make use of 2 datasets. Each dataset resembles the content subtree. The *synthetic* dataset is a complete binary tree with max height 20 and $2^20 \approx 10^6$ nodes. The *real-world* dataset contains $1.3 \cdot 10^6$ and is unbalanced. It is taken from an online shop built on top of Adobe's AEM.

5.4 Workload

We paid a lot of attention to the way we simulate a changing workload during the experiments. The goal was to design a function f, which given two integer arguments, one randomly selected integer k and another representing the current workload. f returns a new integer representing the picked content node.

We calculate the current workload integer by dividing the time passed since the experiment started (in milliseconds) by the workload interval, 30 seconds in our instance. Doing so, we can change the workload every 30 seconds.

It is crucial that f has an an avalanche effect [1]. When changing the workload, we want f to pick completely different content nodes.

We selected a function that takes the two inputs, concatenates them, hashes the concatenated value and returns a single integer.

5.5 Experiments

5.5.1 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 and 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.

H_3: An increase in τ yields a decrease to WAPI's query runtime under a CMS workload.

H_4: An increase in τ yields a decrease to the number of traversed unproductive nodes during WAPI's query execution under a CMS workload.

We conduct the same experiment under a varying volatility threshold. Figures 14a and 14b show thresholds $\tau \in \{1, 5, 10\}$ affecting query runtime over update operations. We observe a decrease in query runtime while increasing threshold τ. A lower volatility threshold increases the likelihood of a node becoming volatile. The amount of volatile nodes also affect the number of unproductive nodes, since volatile nodes eventually stop being frequently updated and become unproductive. The increase in unproductive nodes in the index directly affects query runtime because Oak has to traverse these nodes during query execution. Figures 14c and 14d compare query runtime over a range of thresholds. The values picked correspond to the query runtime after 10^4 update operations. We observe a power law relationship between threshold τ and average query runtime. As explained above, a lower threshold increases the number of volatile and consequently also the number of unproductive nodes. More unproductive nodes do increase the query runtime. We believe the power law relationship is explained by the zipf distribution our workload uses.

Figures 15a and 15b 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. Figures 15c and 15d 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. Volatile nodes eventually cease to be volatile and become unproductive.

We also see in this figure the two variables sharing a power law relationship as well. We believe the workload's zipf skew to be responsible for the power law relationship. We see the query executor having to traverse $5k$ unproductive index nodes with a threshold

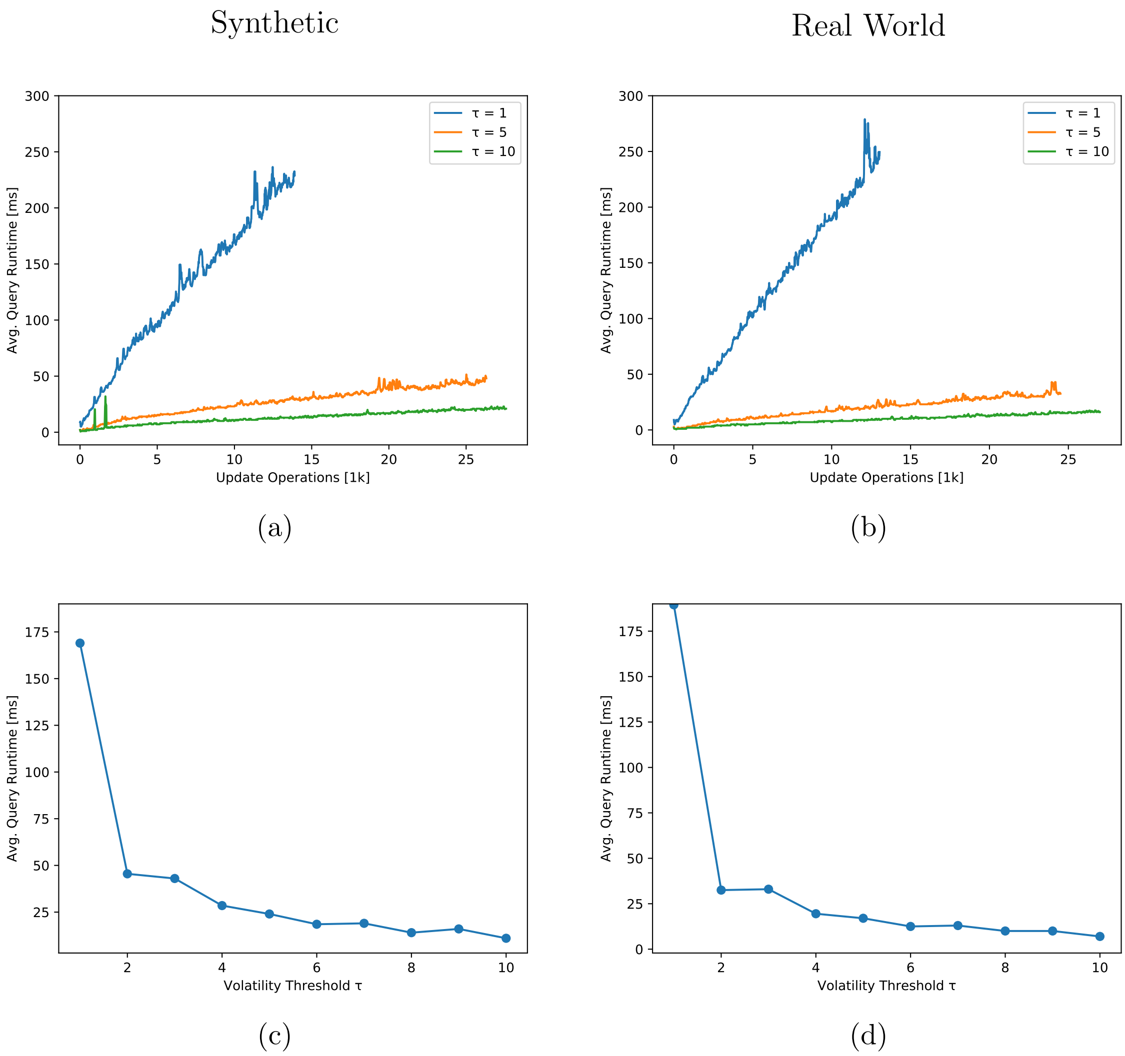
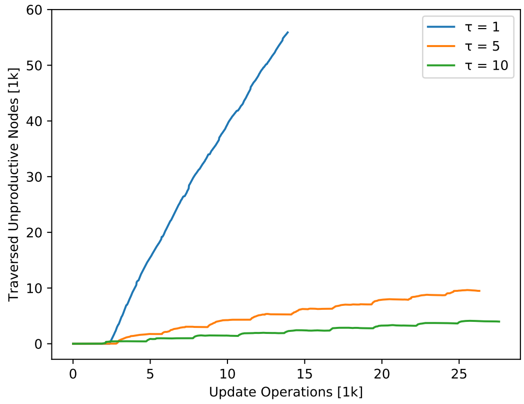


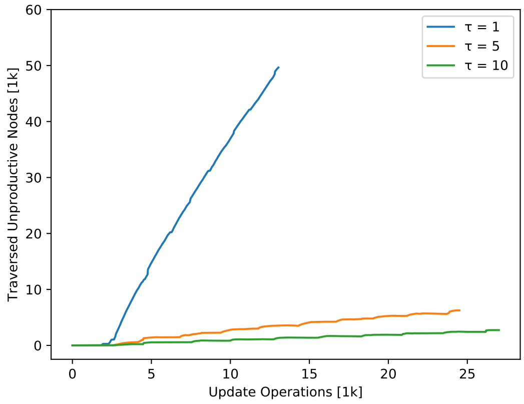
Figure 14: Impact of Volatility Threshold τ on Query Runtime

Synthetic

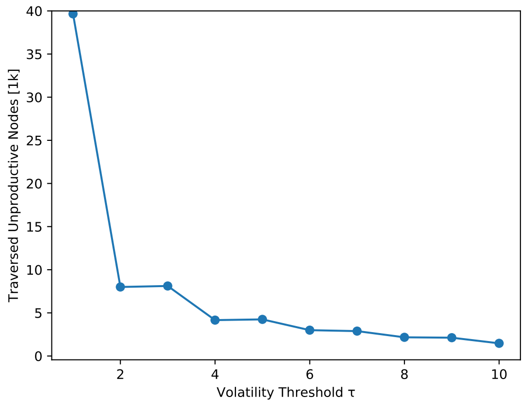
Real World



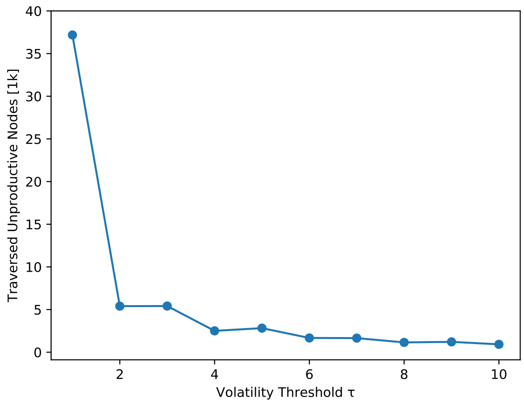
(a)



(b)



(c)



(d)

Figure 15: Impact of Volatility Threshold τ on Unproductive Nodes

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 to be bigger if the distribution were less skewed. Having less skew implies more nodes being picked by the workload.

Summarizing, all observations verify hypotheses H_3 and H_4. Increasing volatility threshold τ decreases the number of unproductive nodes traversed which decreases query runtime. Increasing the volatility threshold causes less nodes become volatile. Since we create less volatile nodes, we also reduce the number of unproductive nodes. Less unproductive nodes yield lower WAPI query runtimes. Another factor affecting the rate of growth of unproductive nodes is sliding window of length L. The following section addresses the effects of the sliding window.

5.5.2 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 consider 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 and query runtime.

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.

H_5: An increase in L yields an increase to WAPI's query runtime under a CMS workload.

H_6: An increase in L should increase the number of unproductive nodes WAPI traverses during query execution under a CMS workload.

Figures 16a and 16b show WAPI's average query runtime over update operations with sliding window of length $L \in \{10, 20, 30\}$ seconds. Figures 16c and 16d depict query runtime with respect to the sliding window. We see queries being executed by a WAPI with a greater sliding window length to have greater runtimes on average. By increasing the sliding window, we increase the likelihood of a node becoming volatile. More volatile nodes result in an increase in unproductive nodes. Since the WAPI has to traverse more unproductive nodes during query execution, the query runtime increases.

Figures 17a and 17b show the number of unproductive nodes the WAPI has traversed during query execution. As expected, we observe greater sliding window lengths to cause an increase to the rate of growth 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 query runtime. 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.

5.5.3 GC Periodicity

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

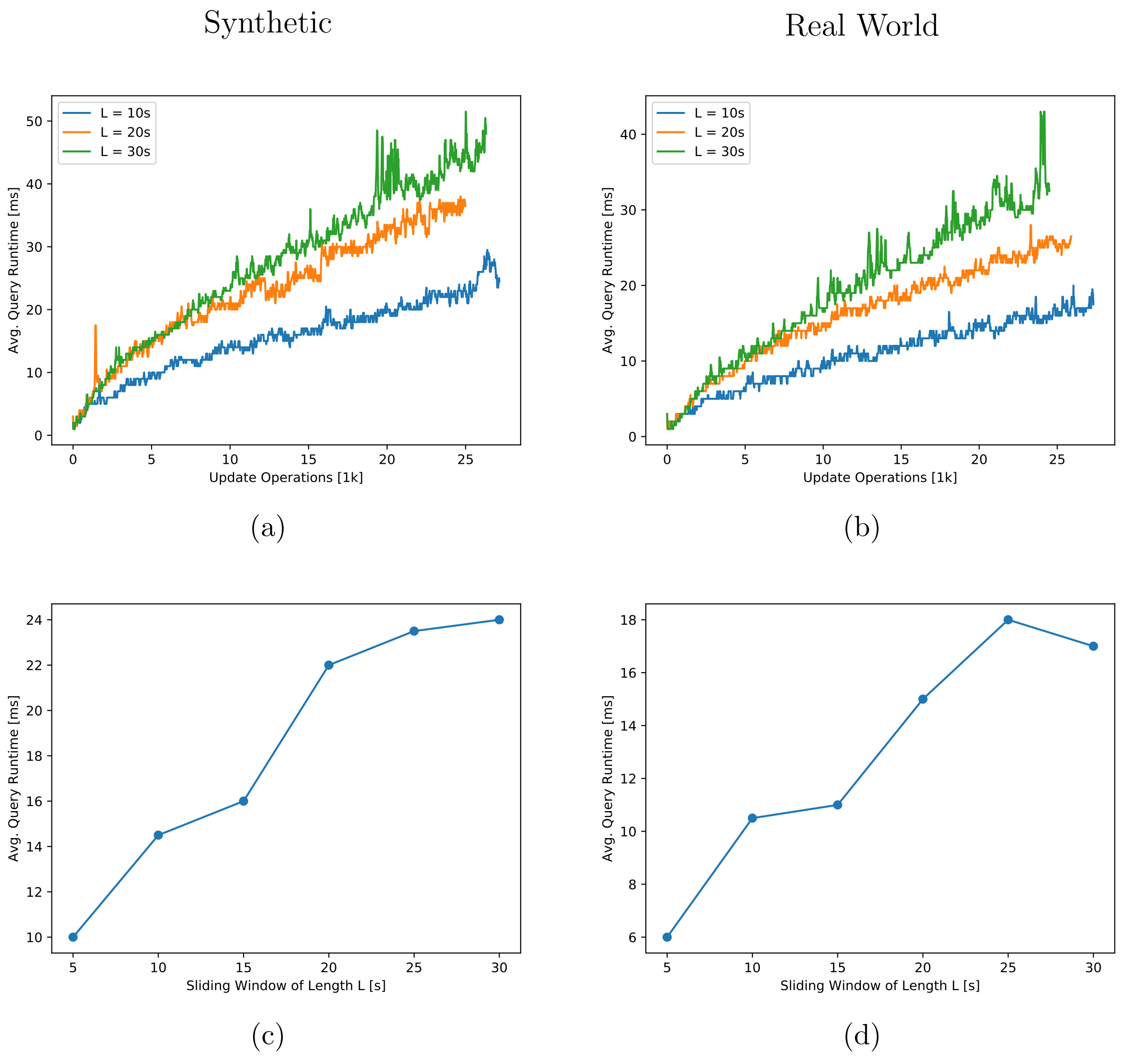


Figure 16: Impact of Sliding Window of length L on Query Runtime

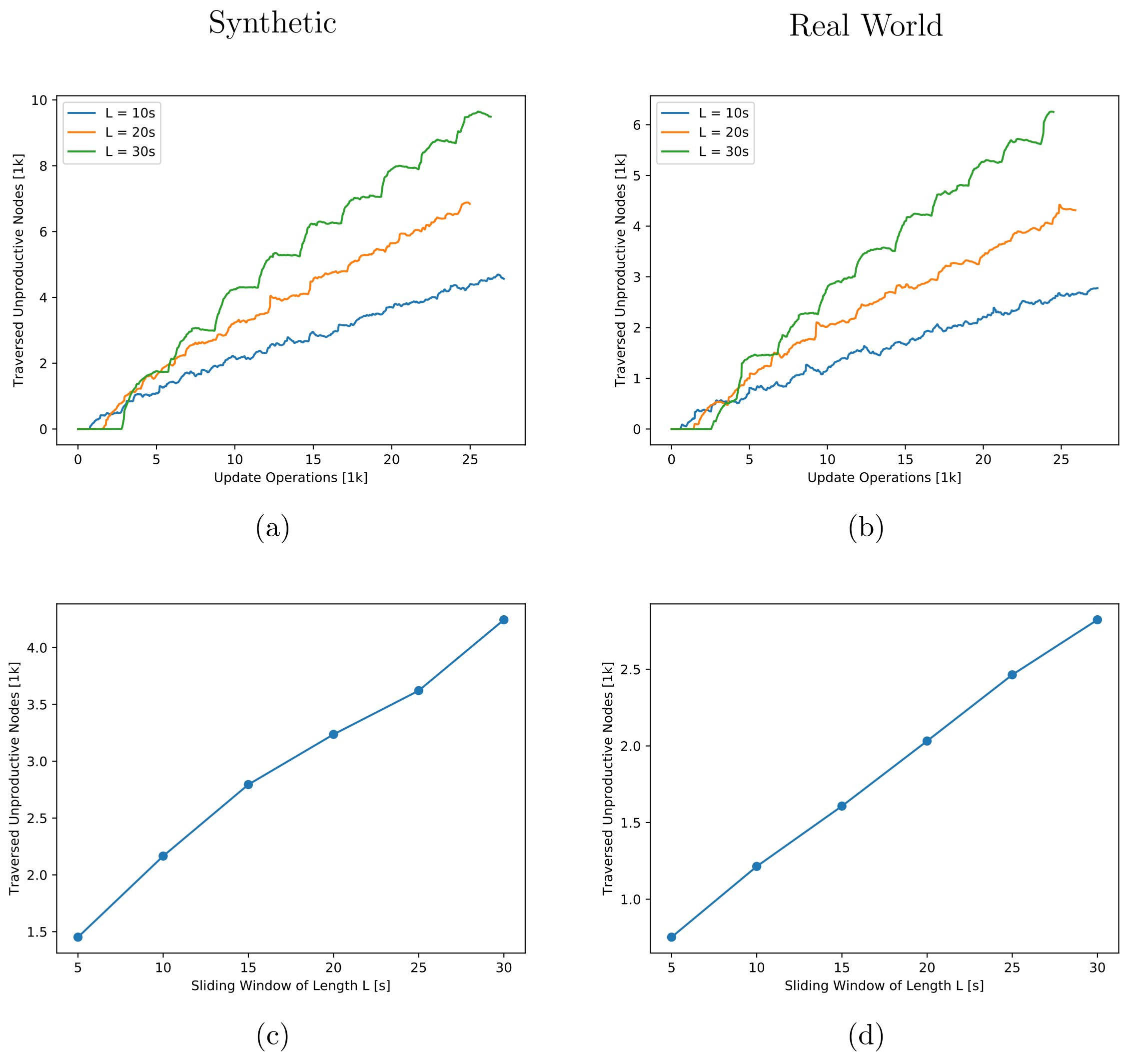


Figure 17: Impact of Sliding Window of length L on Unproductive Nodes

5.5.4 QTP Queried Nodes

5.5.5 Workload Skew

5.5.6 Update to Query ratio

6 Appendix

```java
/**
 * Returns an iterable which lazily traverses a subtree rooted at root
 * in a depth-first order.
 *
 * @param root: Root node of subtree we apply traversal on
 * @returns: iterable for DFS tree traversal
 */
Iterable<Tree> DFS(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 18: `DFS()` implementation

```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 19: `map()` implementation

```
/**
 * 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();
        return new Iterator<T>() {
            private T cur = null;
            @Override
            public boolean hasNext() {
                nextIfNeeded();
                return cur != null;
            }
            @Override
            public T next() {
                nextIfNeeded();
                T tmp = cur;
                cur = null;
                return tmp;
            }
            @Override
            private void nextIfNeeded() {
                while (cur == null && iterator.hasNext()) {
                    T candidate = iterator.next();
                    if (predicate.test(candidate)) {
                        cur = candidate;
                    }
                }
            }
        };
    };
}
```

Figure 20: `filter()` implementation

References

[1] H. Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[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] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[4] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

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