

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](mailto:rk@rafaelkallis.com)

February 1, 2018

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn



University of  
Zurich<sup>UZH</sup>

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

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Apache Jackrabbit Oak (Oak)	7
2.2	Application Scenario	7
2.3	Workload Aware Property Index (WAPI)	8
<b>3</b>	<b>Unproductive Nodes</b>	<b>11</b>
3.1	Introduction	11
3.2	Impact on Query Runtime	12
<b>4</b>	<b>Cleaning Unproductive Nodes</b>	<b>15</b>
4.1	Periodic Garbage Collection (GC)	15
4.2	Query Time Pruning (QTP)	18
<b>5</b>	<b>Experimental Evaluation</b>	<b>21</b>
5.1	Goals	21
5.2	Setup	21
5.3	Datasets	21
5.4	Workload	21
5.5	Experiments	22
5.5.1	Volatility Threshold $\tau$	22
5.5.2	Sliding Window of length $L$	23
5.5.3	GC and QTP Query Performance	26
5.5.4	GC Periodicity	31
5.5.5	QTP Queried Nodes	31
5.5.6	Workload Skew	31
5.5.7	Update to Query ratio	31
<b>6</b>	<b>Appendix</b>	<b>31</b>

## List of Figures

1	An instance of an hierarchical database . . . . .	9
2	Volatile nodes becoming unproductive . . . . .	10
3	Query Runtime over time . . . . .	12
4	Index composition during Query Execution . . . . .	13
5	Node Ratio during Query Execution . . . . .	14
6	Postorder tree walk . . . . .	15
7	GC applied on Oak . . . . .	16
8	Java implementation of periodic garbage collection . . . . .	17
9	QTP applied on Oak . . . . .	19
10	Java implementation of QTP . . . . .	20
11	Visualizing CMS Workloads . . . . .	22
12	Impact of Volatility Threshold $\tau$ on Query Runtime . . . . .	24
13	Impact of Volatility Threshold $\tau$ on Unproductive Nodes . . . . .	25
14	Impact of Sliding Window of length $L$ on Query Runtime . . . . .	27
15	Impact of Sliding Window of length $L$ on Unproductive Nodes . . . . .	28
16	Query Runtime with periodic GC . . . . .	29
17	Index composition with periodic GC . . . . .	29
18	Query Runtime with QTP . . . . .	30
19	Index composition with QTP . . . . .	31
20	DFS() implementation . . . . .	32
21	map() implementation . . . . .	33
22	filter() implementation . . . . .	34

# 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. If 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.

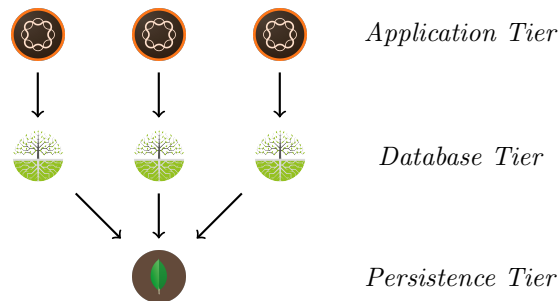
Wellenzohn et al. [5] propose periodic Garbage Collection (GC), which traverses the entire index subtree and prunes all unproductive index nodes at once. Additionally we propose Query-Time Pruning (QTP), an incremental approach to cleaning up unproductive nodes in the index. The idea is to turn queries 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,<sup>1</sup> 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.<sup>2</sup>



### 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. 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. 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

---

<sup>1</sup><https://www.mongodb.com/what-is-mongodb>

<sup>2</sup><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  $Q(\text{pub}, \text{now}, /a)$ , which queries for every descendant of  $/a$  with “pub” set to “now”, would evaluate to  $Q(\text{pub}, \text{now}, /a) = \{/a/b/d\}$ , since  $/a/b/d$  is the only descendant of  $/a$  with “pub” set to “now”.

Figure 1 depicts an Oak instance with the WAPI. 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*). The second index level consists of all properties  $k$  we want to index, such as “pub”. The third index level contains any values  $v$  of property  $k$ , for example “now”. 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  $path(n) = /i/k/v/m = /i/k/v/\lambda_1/\dots/\lambda_d$ , then  $n$ ’s corresponding content node  $*n$  has the path  $path(*n) = m = /\lambda_1/\dots/\lambda_d$ .

**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 subtree. Node  $/i/pub/now/a/b/d$  is matching, since its corresponding content node,  $/a/b/d$ , has property “pub” set to “now”.



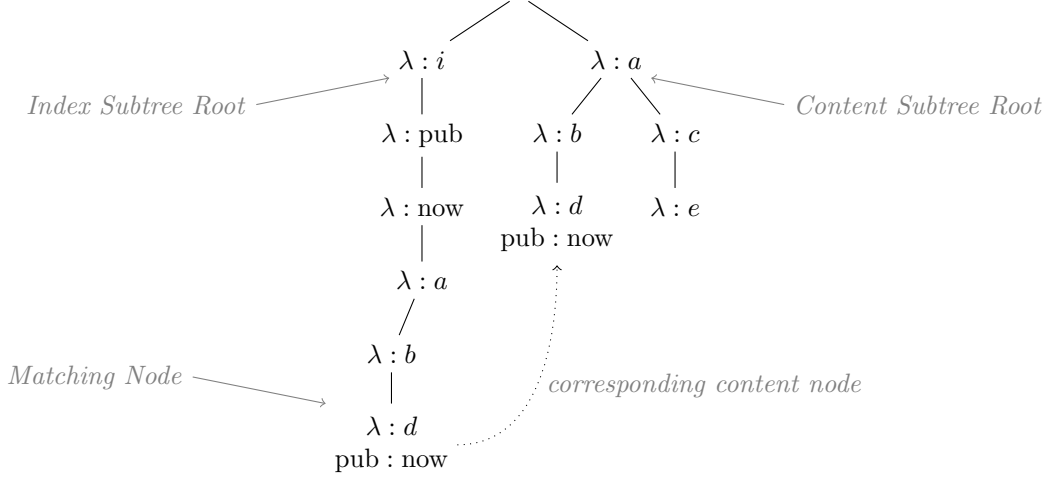


Figure 1: An instance of an hierarchical database. The index subtree is rooted at  $/i$ . The content subtree is rooted at  $/a$ . Index node  $/i/pub/now/a/b/d$  is matching, since its corresponding content node  $/a/b/d$  has property “pub” set to “now”.

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  $\tau$ , 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  $H_i$ .

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

**Definition 4** (Volatile Node). Index node  $n$  is volatile iff  $n$ 's volatility count (see Definition 3) is greater or equal than the volatility threshold  $\tau$ , 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  $T_1, \dots, T_3$ . 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  $t(G^2) = t + 2$ . 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.

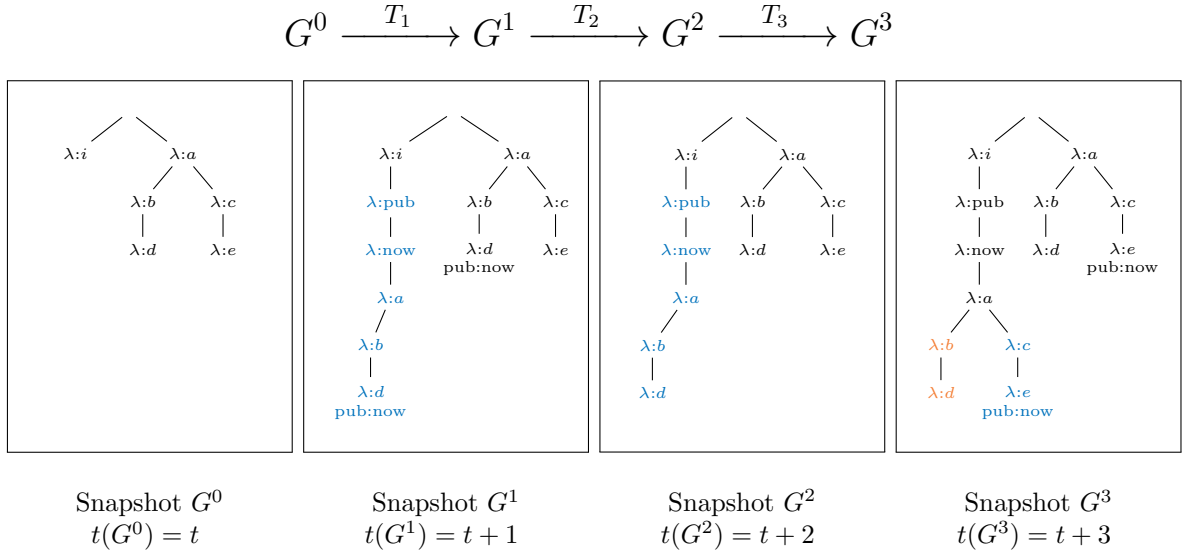


Figure 2: Volatile nodes becoming unproductive. Given  $\tau = 1$ ,  $L = 1$ , nodes /i/pub/now/a/b/d and /i/pub/now/a/b are unproductive in snapshot  $G^3$ . They are not volatile and don't match either. Note that volatile and unproductive index nodes are color-coded blue and orange, respectively.

## 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  $\tau$ . 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  $\tau$ , 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

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

**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 ( $\tau = 1, L = 1$ ). 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 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 (cf. 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 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 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  $Q(pub, now, /a)$  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  $Q(pub, now, /a) = \{/a/c/e\}$ .

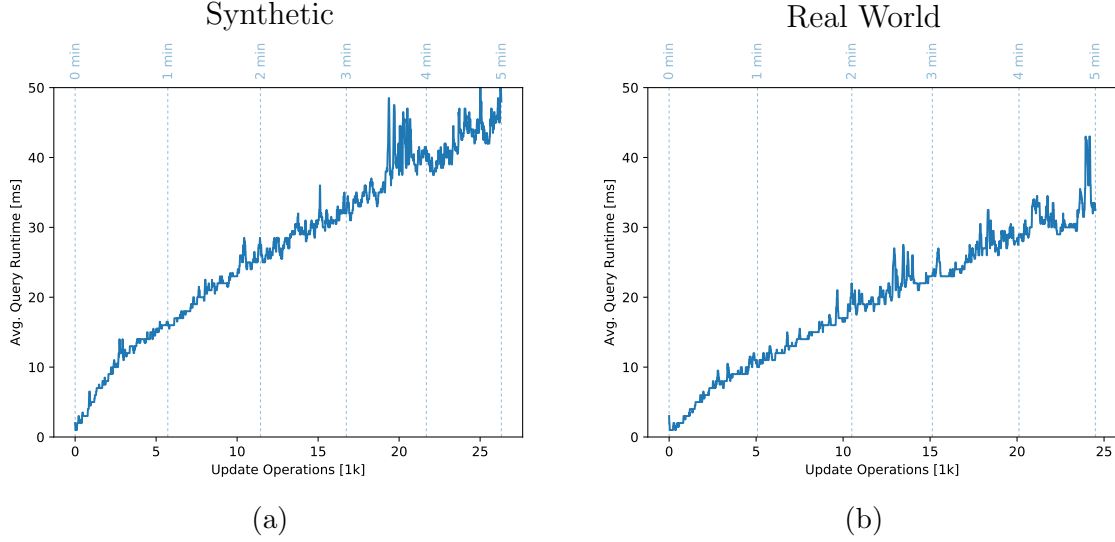


Figure 3: Query Runtime over time

### 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.

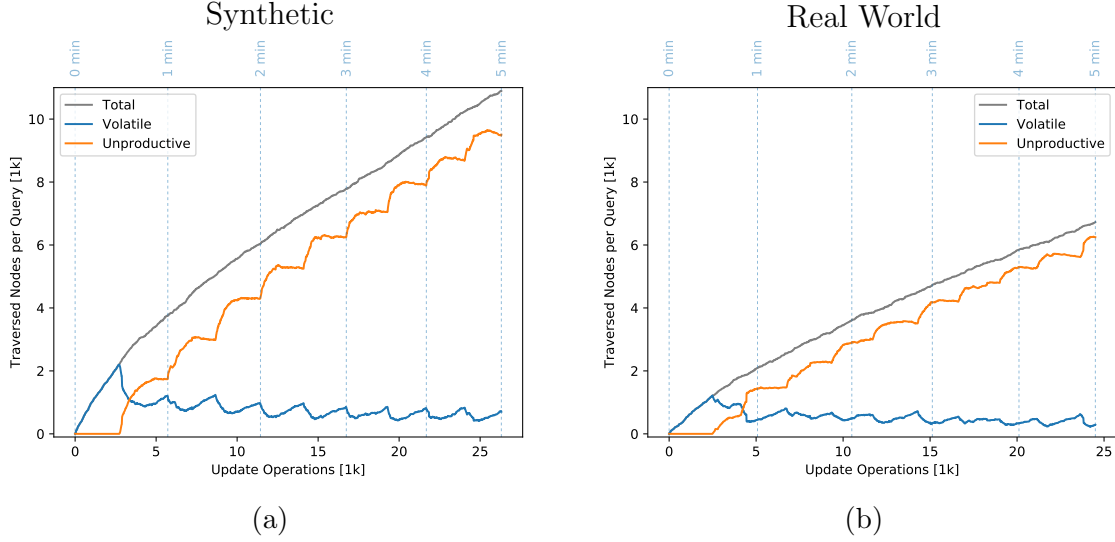


Figure 4: Index composition during Query Execution

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

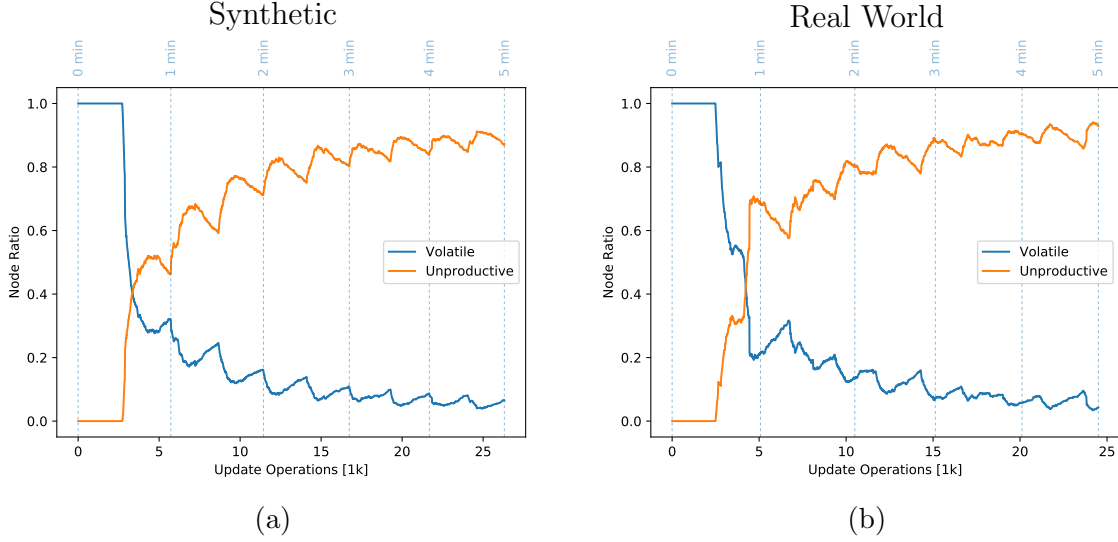


Figure 5: Node Ratio during Query Execution

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 ( $30s, 60s, \dots, 300s$ ) 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.

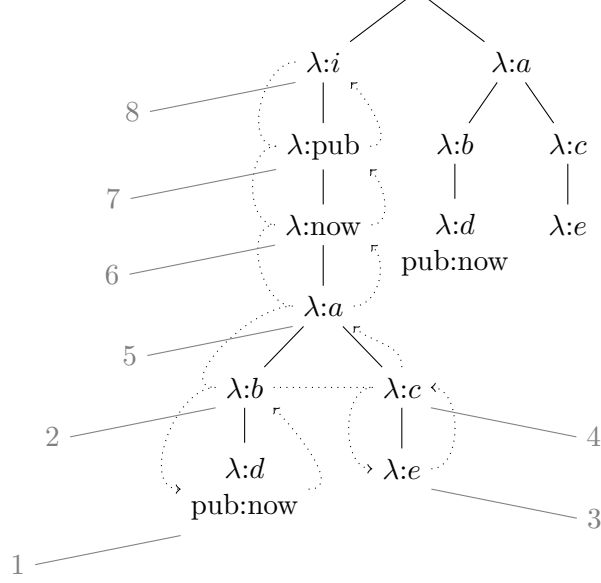


Figure 6: Postorder tree walk on the index subtree rooted at `/i`. The numbers represent the order the corresponding nodes were visited, e.g `/i/pub/now/a/b/d` was visited first, `/i/pub/now/a/b` second, etc.

## 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. 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  $n$ . Checking if  $n$  has children is sufficient to determine if  $n$  has matching or volatile descendants. Figure 6 depicts a postorder tree walk on an instance of Oak. 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  $/i$  postorder. If a descendant  $n \in \text{desc}(/i)$  has no children, is not matching and not volatile, it is pruned from the index. If  $n$  has at least one child, we infer that it has at least one matching or volatile descendant and thus  $n$  is not 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  $n \in \text{desc}(/i)$  in postorder tree walk do
  if  $\text{chd}(n) = \emptyset \wedge \neg \text{matching}(n) \wedge \neg \text{volatile}(n)$  then
    delete node  $n$ 

```

---

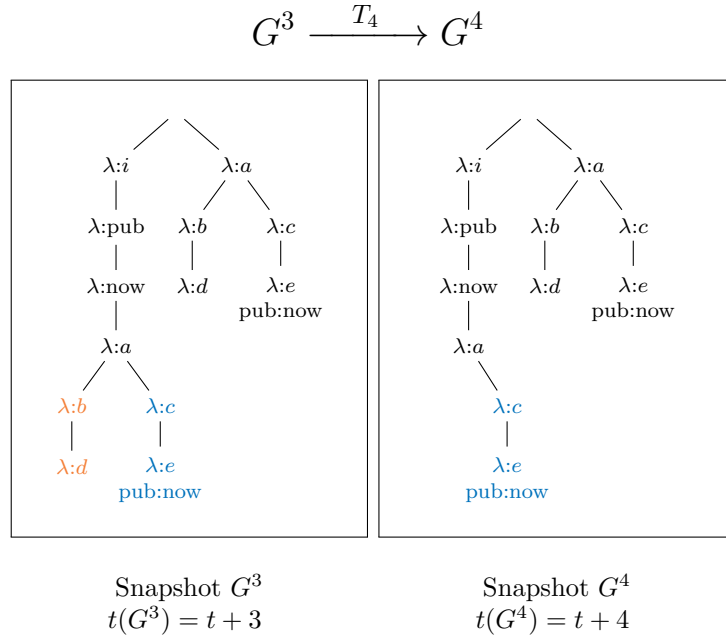


Figure 7: Garbage collection applied on Oak. Assume nodes  $/i/\text{pub}/\text{now}/a/b/d$  and  $i/\text{pub}/\text{now}/a/b$  are unproductive (color-coded orange) in snapshot  $G^3$ . Garbage collection is executed during transaction  $T_4$ .  $G^4$  is the committed snapshot.

**Example 6** (Periodic GC). Figure 7 shows snapshot  $G^3$  from Figure 2. Index nodes  $/i/\text{pub}/\text{now}/a/b/d$  and  $/i/\text{pub}/\text{now}/a/b$  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 in  $G^3$  is traversed in postorder. The first unproductive node we encounter and prune is  $/i/\text{pub}/\text{now}/a/b/d$ . Next, the garbage collector encounters and prunes  $/i/\text{pub}/\text{now}/a/b$ . No further unproductive node is left for pruning. We see the cleaned index after garbage collection in snapshot  $G^4$ .



```

/**
 * 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_);

    /* 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 8: Java implementation of periodic garbage collection

Figure 8 shows the implementation of the periodic GC in Java. Calling `postOrder(n)` returns a lazy sequence of nodes which correspond to a postorder tree walk of the index subtree. Next, any node that has children, is matching or volatile is filtered from the sequence since they are not unproductive. All other nodes are unproductive and therefore pruned from the index.

By applying periodic GC on Oak, we are introducing a new parameter. *Periodicity* defines how often garbage collection is run by the background process on Oak. When picking smaller values for the periodicity, garbage collection is run more often. If we run GC more often, we reduce the average number of unproductive nodes in the index. By decreasing the average number of unproductive nodes, 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 during early morning hours. Like so, garbage collection is run without degrading Oak's performance. We will revisit GC's periodicity in Section 5.5.4, where we conduct an experiment in order to determine the impact of periodicity 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 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.

QTP also benefits from frequent queries. If a query is executed for the first time, QTP cleans all unproductive nodes from the relevant subtree of the query. If the query is executed a second time, the subtree contains no unproductive nodes and therefore is executed fast.

The drawback of QTP are *leftover* unproductive nodes we get from subtrees which are not queried anymore. Leftover unproductive nodes use space but do not impact query performance since they are not traversed during query execution.

When using QTP to answer query  $Q(k, v, m)$ , we apply a postorder tree walk. With postorder tree walk, we traverse the subtree of  $/i/k/v/m = /i/k/v/\lambda_1/\dots/\lambda_d$  in linear time complexity. If any descendant  $n \in \text{desc}(/i/k/v/m)$  has at least one child, we infer that it has at least one matching or volatile descendant, therefore making  $n$  not 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: QueryQTP

---

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

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

$r \leftarrow \emptyset$

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

**if**  $\text{matching}(n)$  **then**

$r \leftarrow r \cup \{n\}$

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

        delete node  $n$

**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 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  $/i/k/v/m = /i/k/v/\lambda_1/\dots/\lambda_d$  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  $n$  is not matching, has no children and is not volatile, it is unproductive and deleted from the index. After we finish traversing all descendants  $n$ , we return the result set  $r$ .

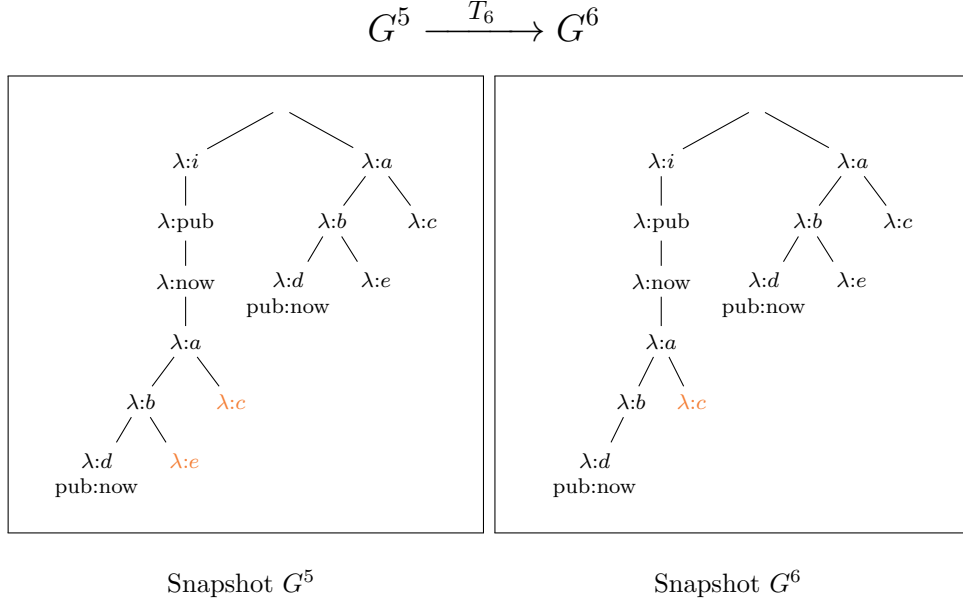


Figure 9: Query Time Pruning applied on Oak. Assume nodes  $/i/pub/now/a/b/e$  and  $i/pub/now/a/c$  are unproductive in snapshot  $G^5$ . Transaction  $T_6$  executes CAS query  $Q(pub, now, /a/b)$  which queries for all descendants of  $/a/b$  with “pub” set to “now” and commits the resulting snapshot  $G^6$ . QTP is used during query execution.

**Example 7 (QTP).** Consider Figure 9. Transaction  $T_6$  executes CAS query  $Q(pub, now, /a/b)$  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 nodes are not traversed during query execution, they are not pruned and remain in the index unproductive. The resulting snapshot  $G^6$  is committed by  $T_6$  after finishing query execution.

Figure 10 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  $Q(k, v, m)$ . 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.

```

/**
 * 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_, k, _INDEX_, 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 10: Java implementation of QTP

## 5 Experimental Evaluation

### 5.1 Goals

The goal of the experiments is to evaluate and compare GC and QTP under different parameterizations and workloads. We initially see how different values of the volatility threshold  $\tau$  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. We compare GC and QTP side by side and list their differences. Next, we investigate the effects periodicity has on garbage collection. 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 ratio 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

In Section 2.2, 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. The 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 5.5.6, we make an experiment to compare the performance of GC and QTP under workloads with different skew.

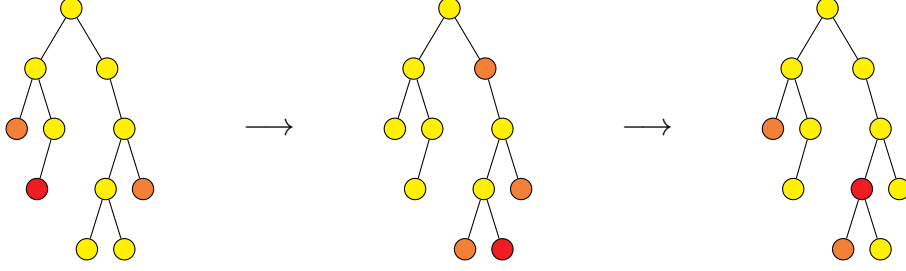


Figure 11: Visualizing CMS Workloads

A CMS workload is also update-heavy. We create many update operations before creating a query operation. We fix the update to query ratio to 10:1, 10 update operations are created for each query operations. In Section 5.5.7, we evaluate and compare GC and QTP under various update to query ratios.

Additionally, the workload we design has to be changing. We permute the node mappings periodically in order to change the hotspots of the simulation. We chose to 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 operations 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 11 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 Experiments

### 5.5.1 Volatility Threshold $\tau$

Volatility threshold  $\tau$  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  $\tau$  on unproductive nodes and query runtime.

We hypothesize that an increase in  $\tau$  yields a decrease to the number of traversed unproductive nodes during query execution under a CMS workload. If  $\tau$  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  $\tau$  yields a decrease to WAPI’s query runtime under a CMS workload.

$H_4$ : An increase in  $\tau$  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 12a and 12b show thresholds  $\tau \in \{1, 5, 10\}$  affecting query runtime over update operations. We observe a decrease in query runtime while increasing threshold  $\tau$ . 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 12c and 12d 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  $\tau$  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 13a and 13b depict thresholds  $\tau \in \{1, 5, 10\}$  affecting the number of unproductive nodes traversed during query execution over update operations. We observe lower thresholds  $\tau$  yielding in a steeper slope. Figures 13c and 13d 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  $\tau$ . 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 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  $\tau$  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

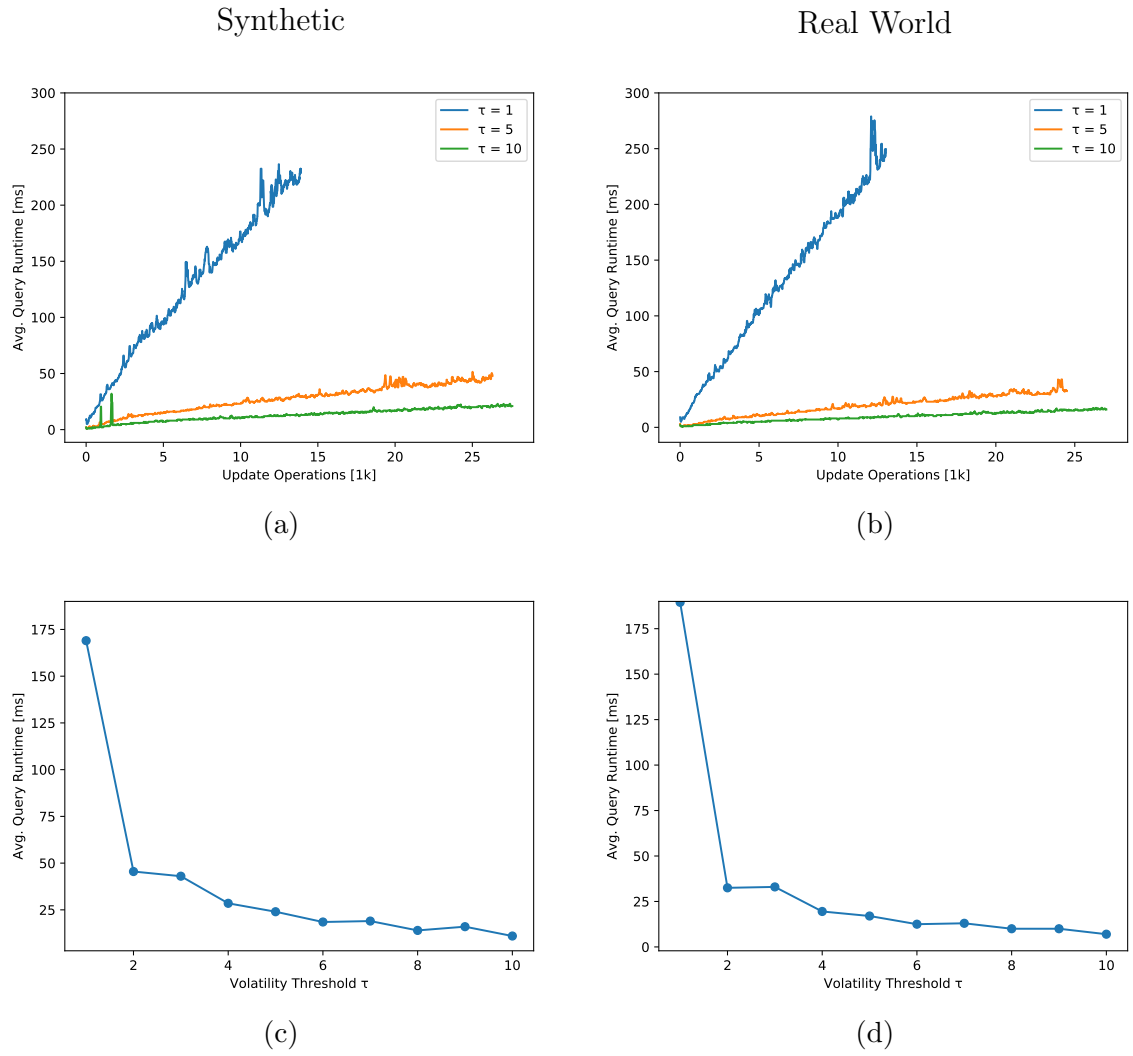


Figure 12: Impact of Volatility Threshold  $\tau$  on Query Runtime



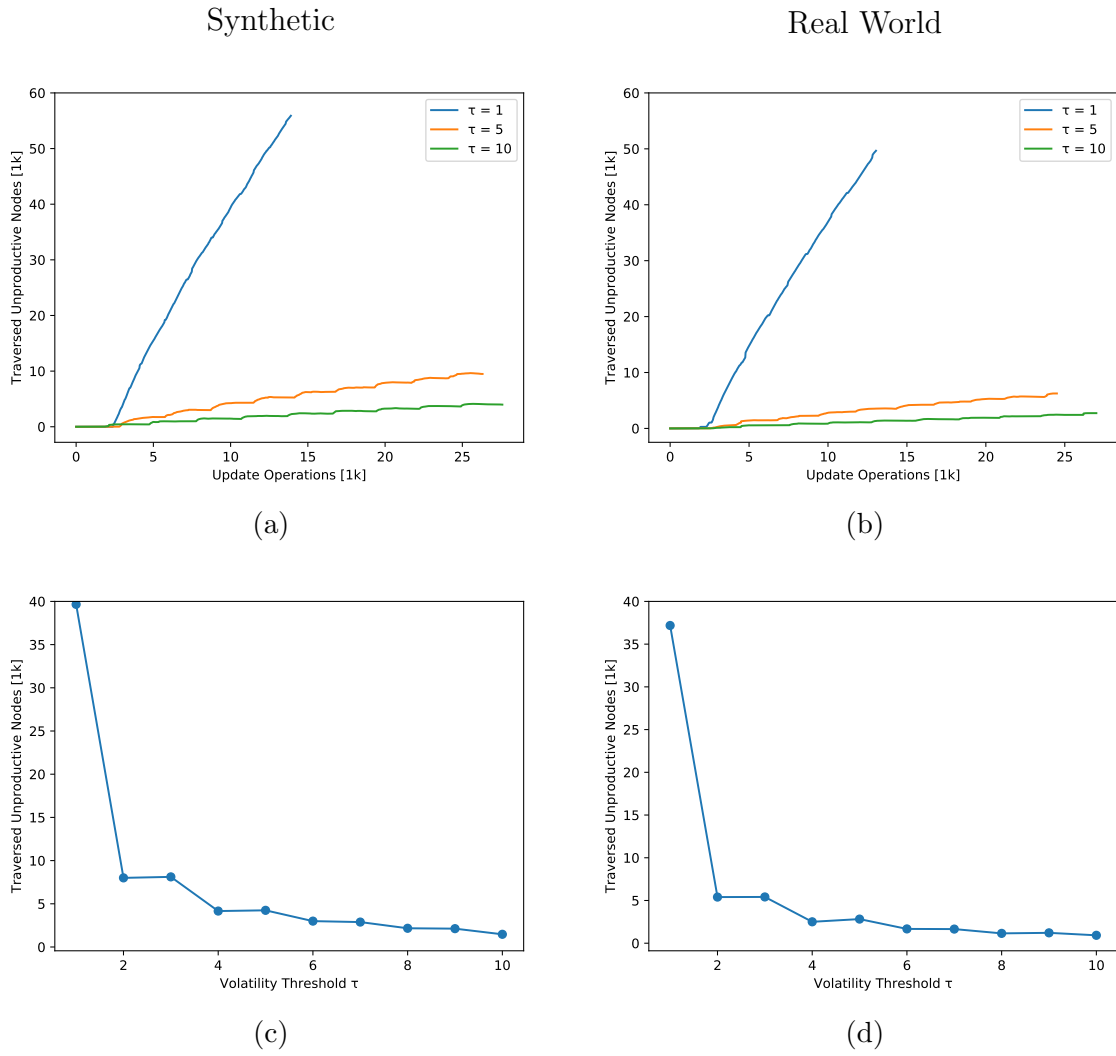


Figure 13: Impact of Volatility Threshold  $\tau$  on Unproductive Nodes

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 14a and 14b show WAPI's average query runtime over update operations with sliding window of length  $L \in \{10, 20, 30\}$  seconds. Figures 14c and 14d 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.

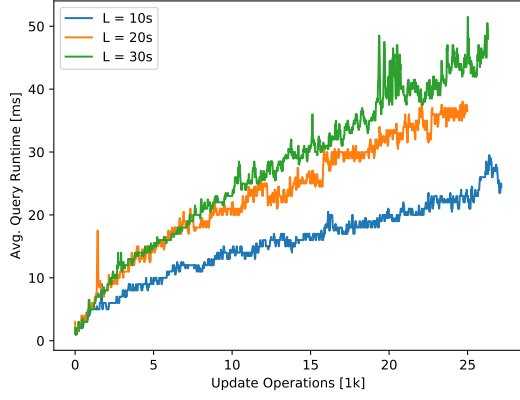
Figures 15a and 15b show the number of unproductive nodes the WAPI has traversed during query execution. As expected, we observe greater sliding window lengths to cause an increase to the rate of growth of unproductive nodes traversed by WAPI during query execution. More volatile nodes imply an increase 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 and QTP Query Performance

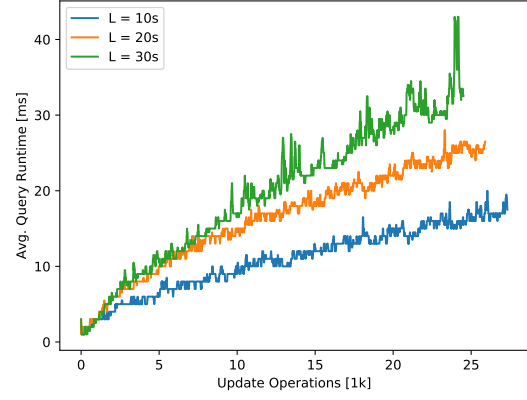
Our next experiment compares the query performance of periodic garbage collection and query time pruning, when applied on Oak. We record the average query runtime and the index structure during query execution first while having GC exclusively enabled and then with QTP exclusively enabled.

Synthetic

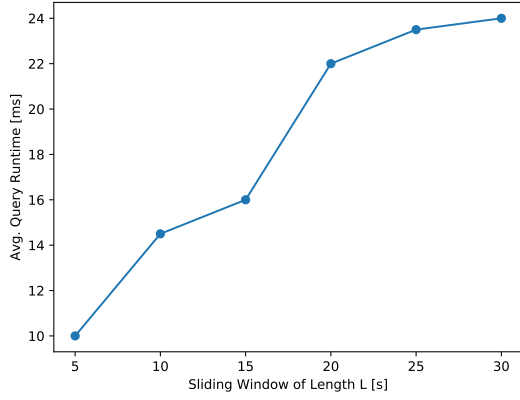


(a)

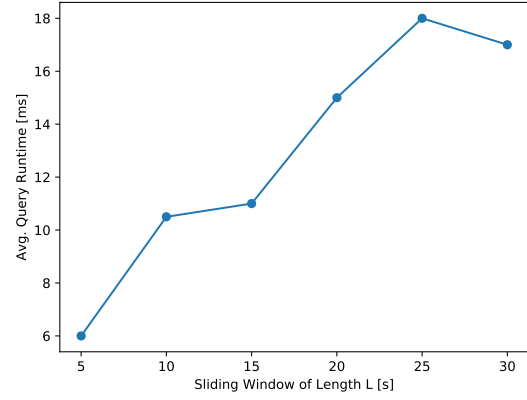
Real World



(b)



(c)



(d)

Figure 14: Impact of Sliding Window of length  $L$  on Query Runtime

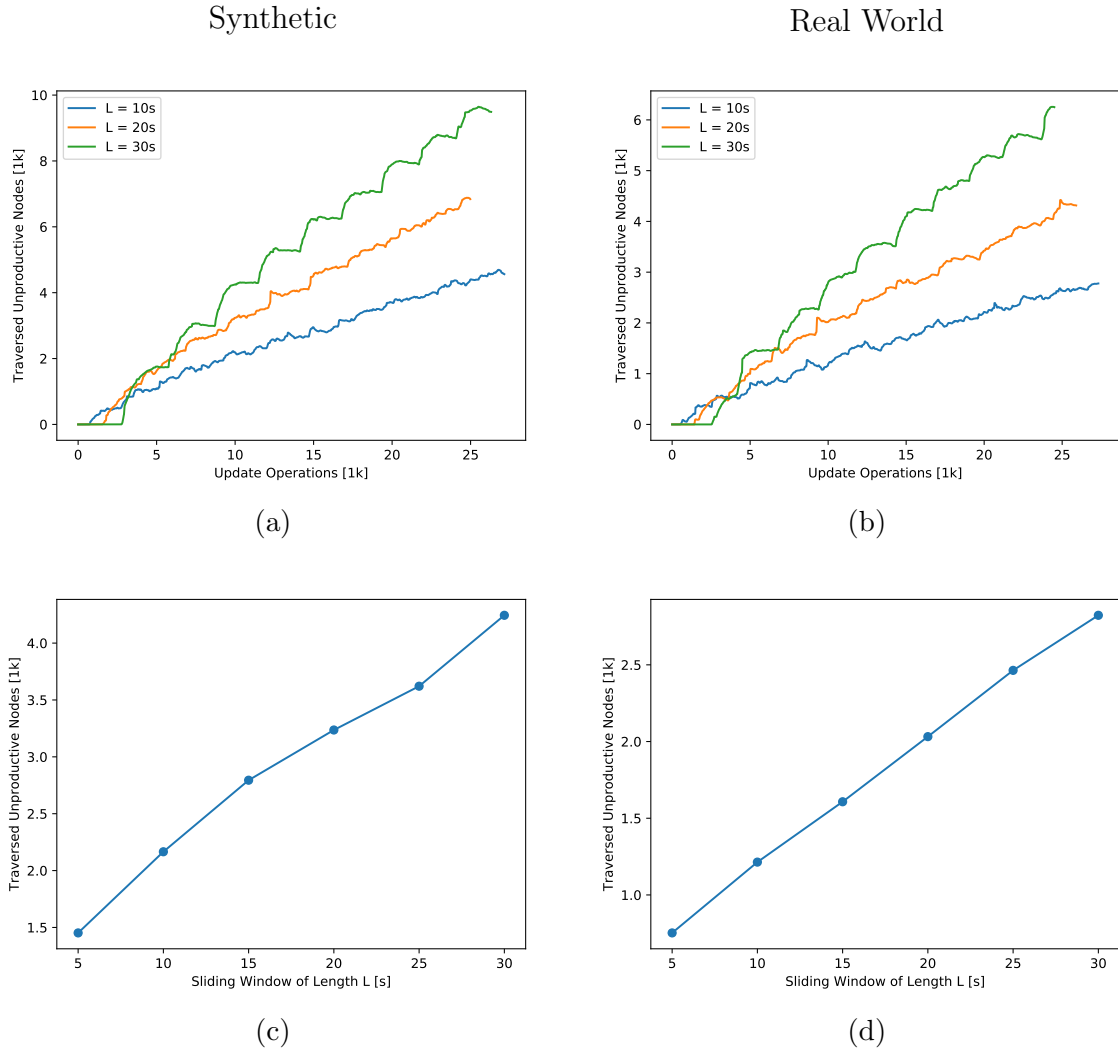


Figure 15: Impact of Sliding Window of length  $L$  on Unproductive Nodes

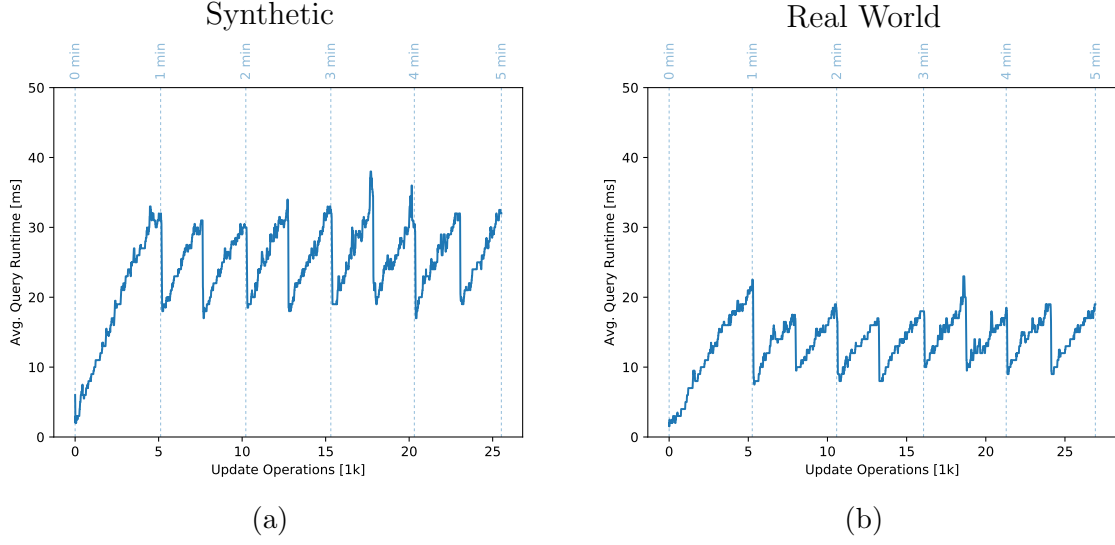


Figure 16: Query Runtime with periodic GC

Figures 16a and 16b 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.

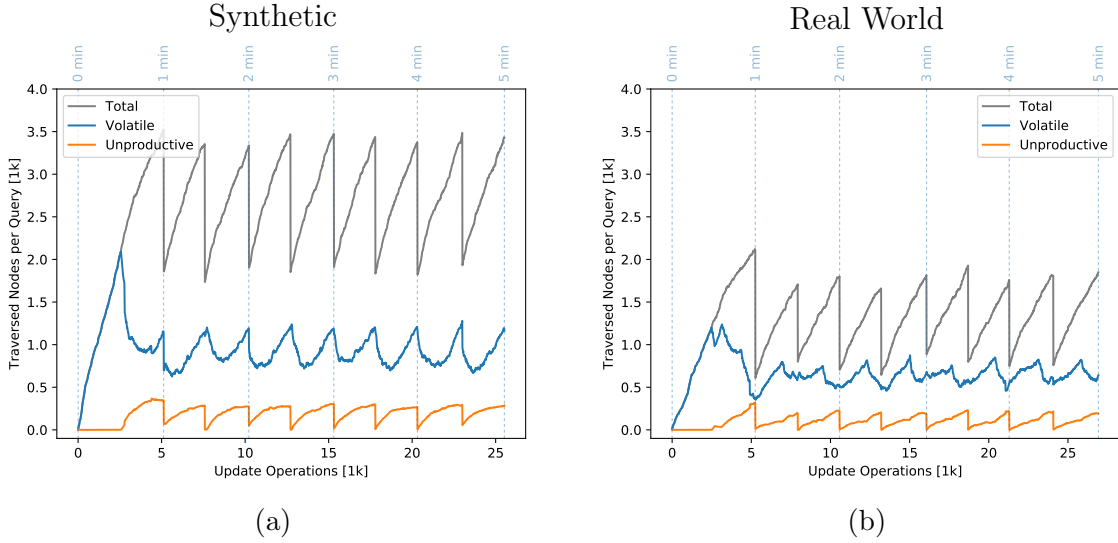


Figure 17: Index composition with periodic GC

Figures 17a and 17b 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 be 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.

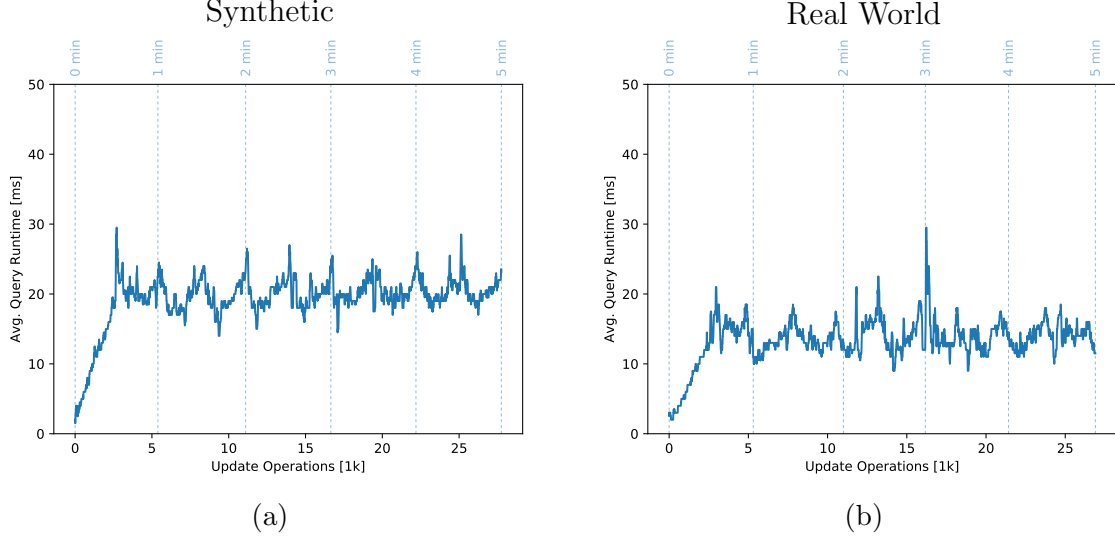


Figure 18: Query Runtime with QTP

Figures 18a and 18b 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.

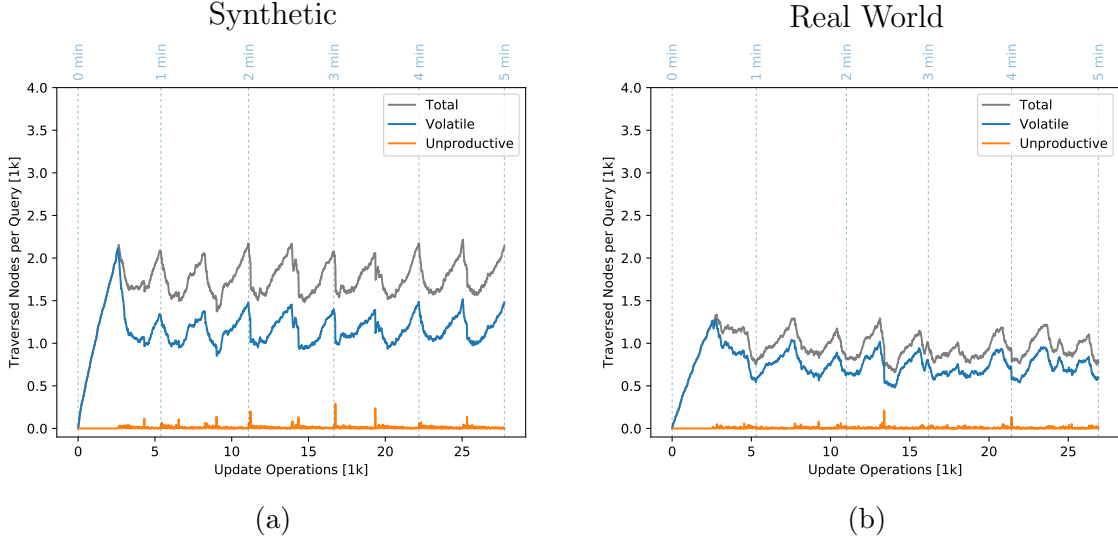


Figure 19: Index composition with QTP

Figures 19a and 19b 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 5.5.5 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.

#### 5.5.4 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.

#### 5.5.5 QTP Queried Nodes

#### 5.5.6 Workload Skew

#### 5.5.7 Update to Query ratio

## 6 Appendix

```

/**
 * 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 20: DFS() implementation



```

/**
 * 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 21: 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 22: filter() implementation

## 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] 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.