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

Contents

List of Figures

1 Introduction

Frequently adding and removing data from hierarchical indexes causes them to repeatedly grow and shrink. A single insertion or deletion can trigger a sequence of structural index modifications (node insertions/deletions) in a hierarchical index. Skewed and update-heavy workloads trigger repeated structural index updates over a small subset of nodes to the index.

Informally, a frequently added or removed node is called *volatile*. Volatile nodes 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. [4] 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 quickly 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. [4] 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 [4] in Apache Jackrabbit Oak (Oak).

2 Background

2.1 Apache Jackrabbit Oak (Oak)

Oak is a hierarchical distributed database system which makes use of a hierarchical index. Multiple transactions can work concurrently by making use of Multiversion Concurrency Control (MVCC) [3], a commonly used optimistic concurrency control technique [2].

Figure 1 depicts Oak's multi-tier 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 Figure 1 under *Application Tier*. One such application is Adobe's enterprise content management system (CMS), the Adobe Experience Manager[2].

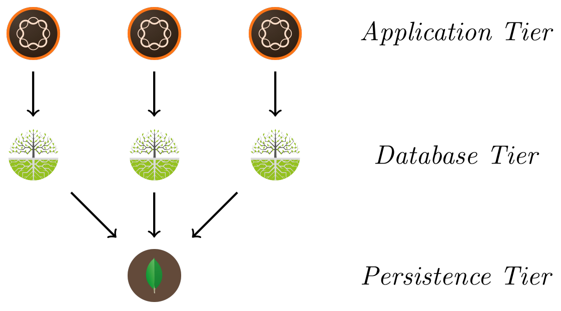


Figure 1: Oak's system architecture

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* [4]. 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, 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, it is sent to the CMS for processing. A background thread is periodically checking for pending jobs. A pending job is signalled using node properties in Oak. The CMS adds a property to the respective node in order to signal the background thread that the specific node is a pending job that needs processing.

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

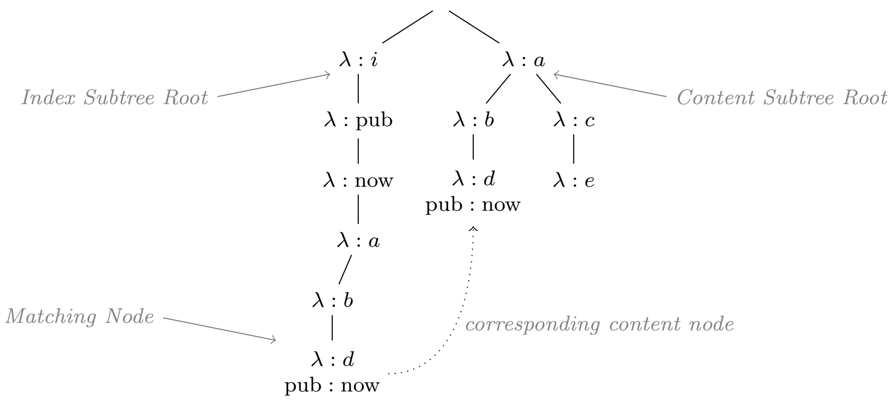


Figure 2: An instance of an hierarchical database

Once the background thread detects the node and finishes processing, it removes the previously set 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 [1]. 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 2. 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 2). The second index level consists of all properties k we want to index. The third index level contains any values v of property k. 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 $path(n) = \texttt{/i/k/v/m}$, then n's corresponding content node $*n$ must have 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 2. 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".

As mentioned in Section 2.2, the CMS workload has certain properties. From the workload we can infer that the index subtree has a small number of matching nodes relative to 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 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. [4] propose to look at the recent transactional workload to check whether a node n is volatile. The workload on Oak instance O_i is represented by a sequence $H_i = \langle \ldots, G^a, G^b, G^c \rangle$ of snapshots, called a history. A snapshot represents an immutable committed tree of the database. Let t_n be the current time and $t(G^b)$ be the point in time snapshot G^b was committed, $N(G^a)$ is the set of nodes which are members of snapshot G^a. We use a superscript a to emphasize that a node n^a belongs to tree G^a. $pre(G^b)$ is the predecessor of snapshot G^b in H_i.

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.

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.

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.

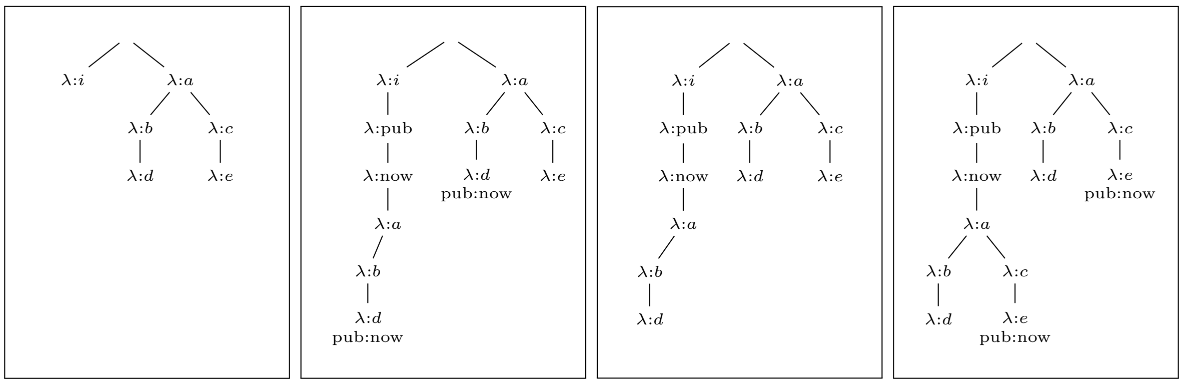
$$\begin{aligned} vol(n) = |\{G^b | G^b \in H_i \land t(G^b) \in [t_{n-L+1}, t_n] \land \exists G^a [\\ G^a = pre(G^b) \land ([n^a \notin N(G^a) \land n^b \in N(G^b)] \lor \\ [n^a \in N(G^a) \land n^b \notin N(G^b)])]\}| \end{aligned}$$

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 3. Assume volatility threshold $\tau = 1$, sliding window 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.

$$\begin{tikzcd} G^0 \arrow[r, "T_1"] & G^1 \arrow[r, "T_2"] & G^2 \arrow[r, "T_3"] & G^3 \end{tikzcd}$$



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

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

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

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

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 3: Volatile nodes becoming unproductive

3 Unproductive Nodes

When time passes and the database workload changes, volatile nodes cease to be volatile and they 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 quickly accumulate in the index and the query performance deteriorates over time [4].

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

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

Example 4. In our running example (cf. Figure 3), transaction T_3 adds property "pub" = "now" to `/a/c/e` given G^2 and commits G^3 at time $t(G^3) = t + 3$. `/i/pub/now/a/b/d` and `/i/pub/now/a/b` are the only unproductive index nodes. `/i/pub/now/a/c/e` and `/i/pub/now/a/c` are the only volatile nodes in G^3.

3.1 Impact on Query Execution Runtime

In this section we study and quantify the impact of unproductive nodes on query runtime. Informally, we call *Query Runtime* the time needed for a query to finish processing. 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.

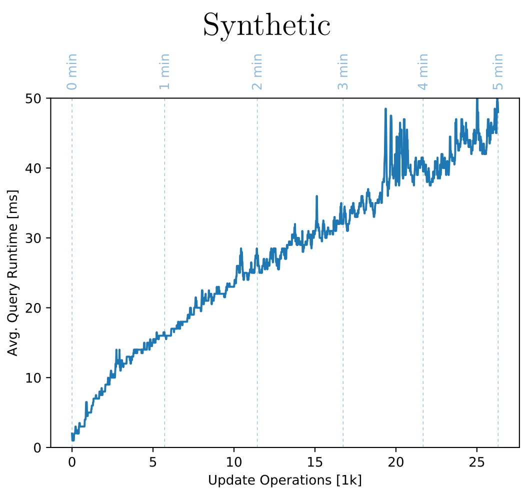
We formalize the statements above into the following hypotheses:

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

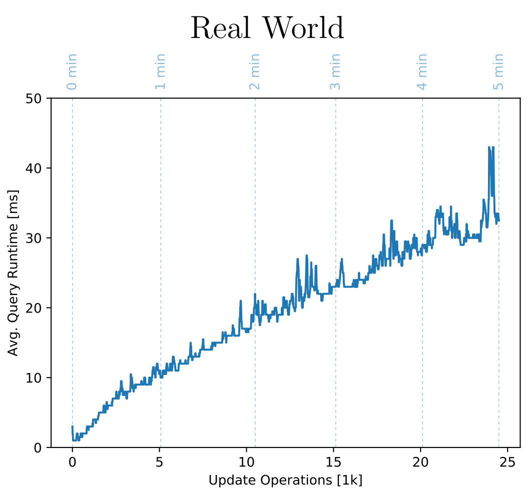
H_2: Unproductive nodes significantly affect WAPI's query runtime under a CMS workload.

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 a later section (**forward reference**). We record the query runtime throughout the experiment and present the data below.

Figures 4a and 4b show query runtime over time from the synthetic and real world dataset respectively. Each point corresponds to the moving median over 10 time points. We observe an 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. We also see the slope decrease over time. The identical phenomenon is observed in Figures 5a and 5b respectively because query runtime depends on the number of nodes traversed during query execution, i.e the increase in query runtime is explained by the increase in total traversed nodes per query.

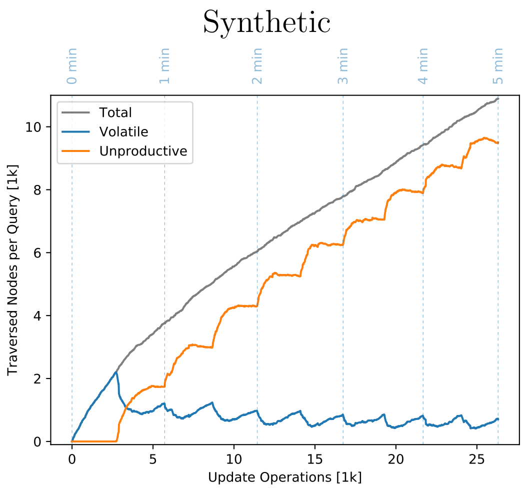


(a)

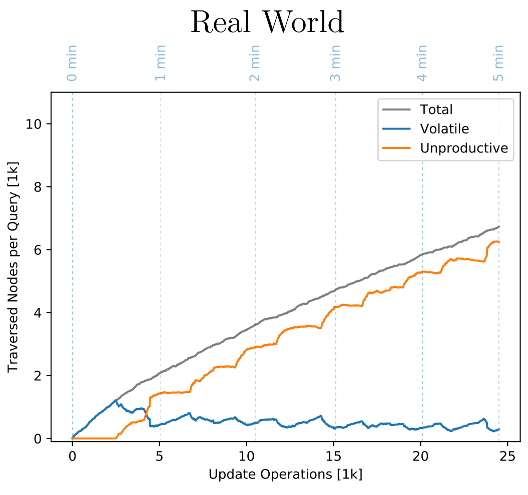


(b)

Figure 4: Query Runtime over time



(a)



(b)

Figure 5: Index Structure during Query Execution

Next, we present data regarding the type of index nodes traversed during query execution. Figures 5a and 5b depict the number of traversed volatile and unproductive index nodes during query execution for each dataset.

The total number of traversed nodes is increasing over time but the rate of growth is decreasing. As time passes, more and more content nodes are randomly selected by the CMS workload and become volatile and, most likely, become unproductive and are not pruned from the index anymore. Therefore, the probability of picking a non-indexed content node 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.

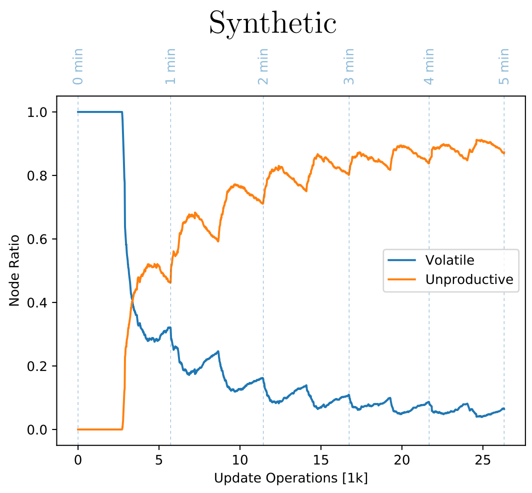
The sliding window of length L is set to 30 seconds, therefore we encounter no unproductive nodes during the first 30 seconds of the simulation. Once we reach the 30 second mark, our queries encounter unproductive nodes. From that point, we observe a steep increase in traversed unproductive nodes. After 3 minutes, we observe the traversed nodes being dominated by unproductive nodes. The rate of growth of traversed 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 a cycloid shaped curve. In Figure 5a, each cusp $(30, 60, \ldots, 300s)$ 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 relatively few do, since the skew in he 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.

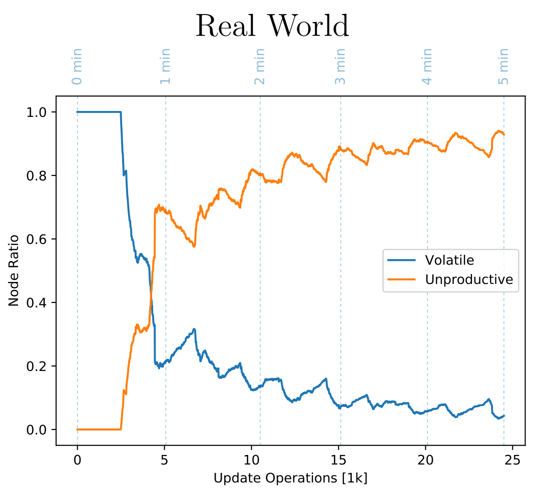
The cycloid pattern observed in Figure 5a is less obvious in Figure 5b. Our synthetic dataset resembles a complete binary tree and therefore has no skew. In comparison, the real world dataset is highly unbalanced.

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. Wellenzohn et al. and our experimental evaluation suggest 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.

Figures 6a to 6b show the ratio of volatile and unproductive nodes over time and update operations from our datasets. These figures quantify how strongly unproductive nodes dominate the 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, 90% and 10% on the real world dataset respectively.



(a)



(b)

Figure 6: Node Ratio during Query Execution

Concluding, the data strongly supports our hypotheses. We see the query runtime increase as the number of index nodes increases. We also see unproductive nodes being mostly accountable for the increase in index nodes. In the following sections we will study the effect of *Volatility Threshold* τ and *Sliding Window Length* L on query runtime and unproductive nodes.

3.2 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.

An increase in τ yields a decrease to WAPI's query runtime under a CMS workload. Having less unproductive nodes in the index decrease the number of nodes traversed during query execution and therefore decrease query runtime.

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 7a

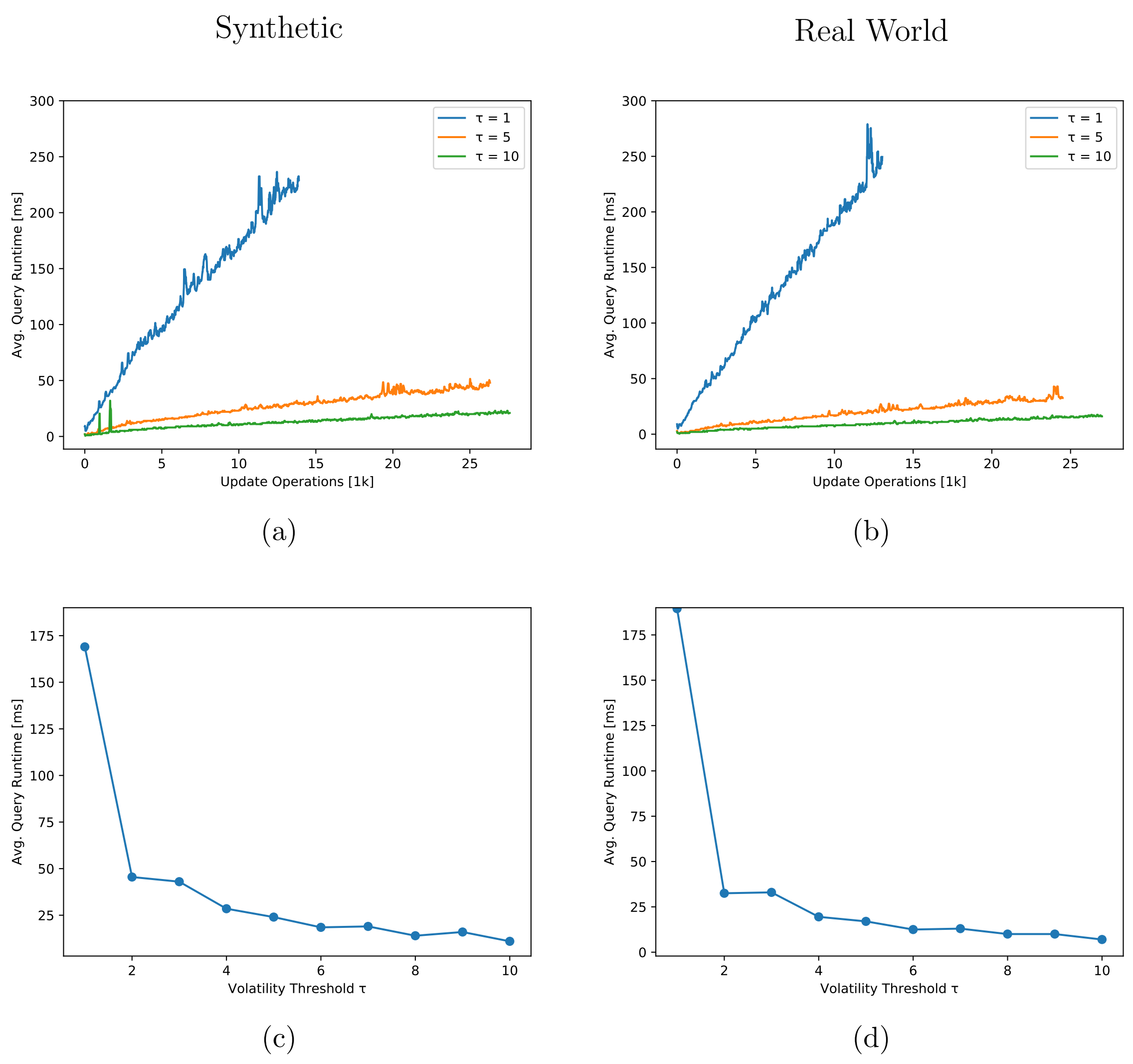


Figure 7: Impact of Volatility Threshold τ on Query Runtime

and 7b show thresholds $\tau \in \{1, 5, 10\}$ affecting query runtime over update operations. We observe that a lower threshold τ results in a faster rate of growth of query runtime. Figures 7c and 7d compare query runtime over a range of thresholds. The values picked correspond to the query runtime after 10^4 update operations. We observe a decrease in query runtime while increasing threshold τ. Having a lower volatility threshold should increase 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 8a and 8b 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 8c and 8d compare the number of

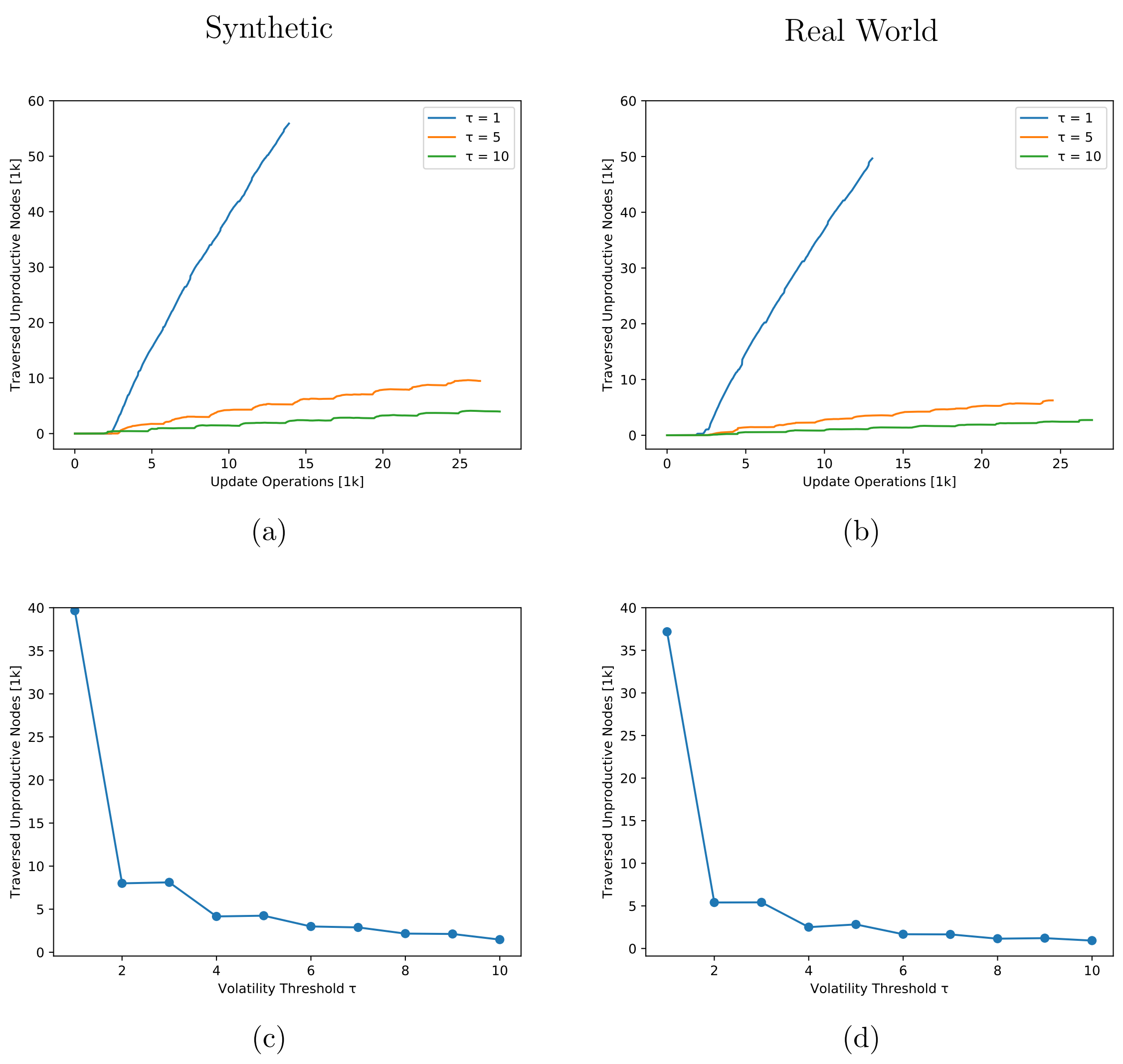


Figure 8: Impact of Volatility Threshold τ on Unproductive Nodes

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.

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.

3.3 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.

We also suggest that an increase in L yields an increase to the query runtime. More unproductive nodes should increase the number of unproductive nodes traversed during query execution and therefore increase query runtime.

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.

We conduct our experiment with a varying sliding window length and present our observations below.

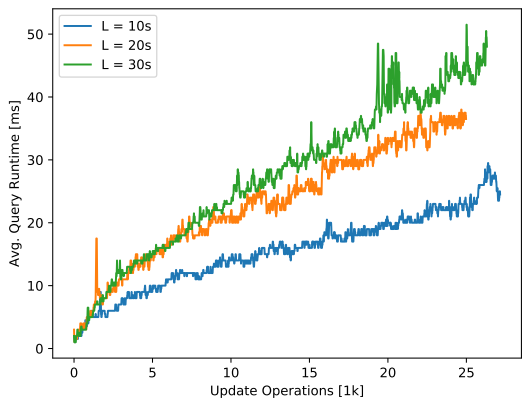
Figures 9a and 9b show WAPI's average query runtime over update operations with sliding window of length $L \in \{10, 20, 30\}$ seconds. Figures 9c and 9d depict query runtime with respect to the sliding window length. We see queries being executed by a WAPI with a greater sliding window length to have 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 10a and 10b 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.

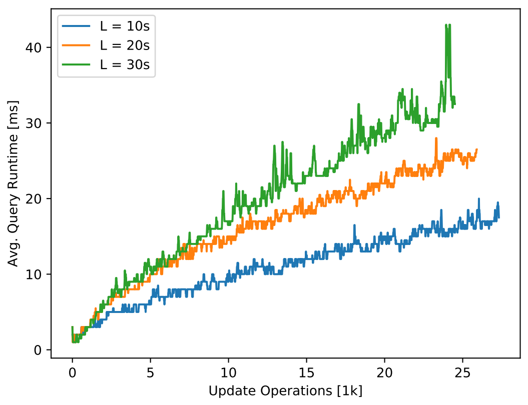
Concluding, we see sliding window of length L to be affecting query runtime. Increasing L does increase the likelihood of an index node become volatile. More volatile nodes cause an increase in unproductive index nodes. Since the WAPI has to traverse more index nodes during query execution, its query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

Synthetic

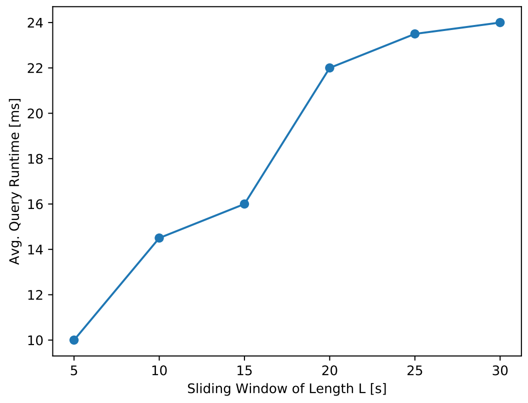
Real World



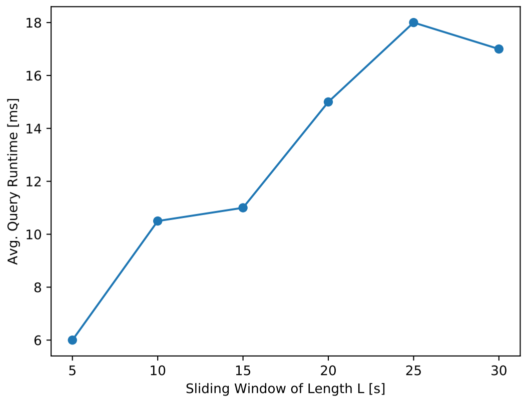
(a)



(b)



(c)

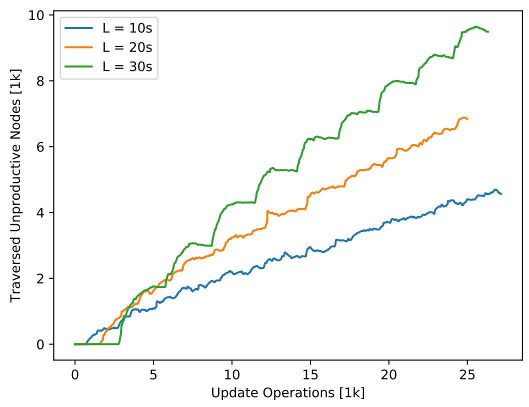


(d)

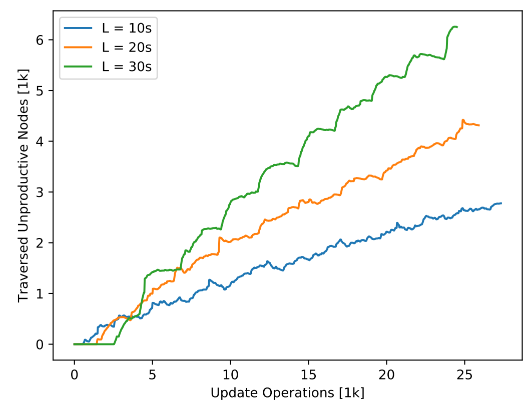
Figure 9: Impact of Sliding Window of length L on Query Runtime

Synthetic

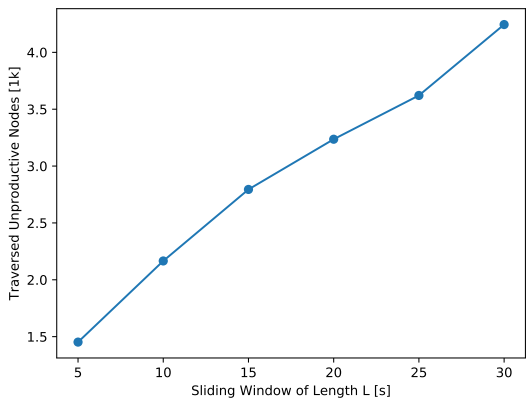
Real World



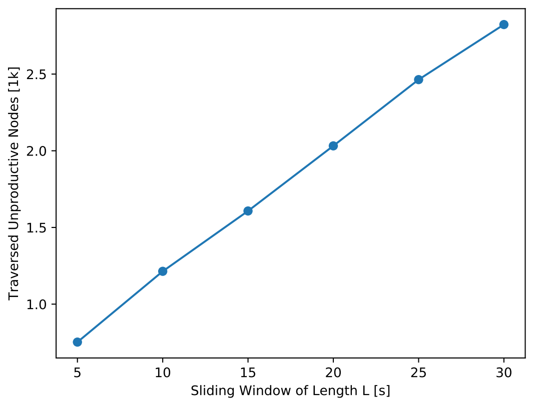
(a)



(b)



(c)



(d)

Figure 10: Impact of Sliding Window Length L on Unproductive Nodes

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

    /* filter nodes which have children, are matching or volatile */
    for (Node unproductiveNode : filter(
            (Node n) -> n.state.getChildNodeCount(1) == 0 &&
                        !n.state.getBoolean("match") &&
                        !n.state.isVolatile()
            ),

            /* bottom-up tree traversing iterable */
            bottomUp(n)
    ) {
        unproductiveNode.state.builder().remove();
    }
}
```

4 Periodic Garbage Collection (GC)

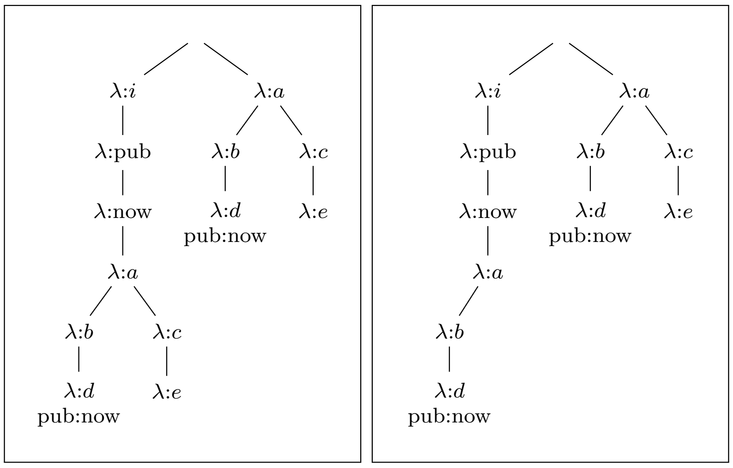
In the previous section, we saw how unproductive nodes slow down query execution. To prevent unproductive nodes from accumulating in the index, we clean the index periodically. Oak has a number of background processes that maintain the database. We add a background job that periodically executes garbage collection on Oak.

Algorithm 1 takes an index node n as parameter and prunes all its unproductive descendants bottom-up. The algorithm traverses the subtree rooted in n. If a descendant $d \in desc(n)$ has no children, is not matching and not volatile, it is pruned from the index. The bottom-up 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 bottom-up fashion 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.

Figure 11: Garbage collection applied on Oak

5 Query Time Pruning (QTP)

6 Experimental Evaluation

7 Appendix

```java
/**
 * 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 store: Oak's database interface
 * @returns an iterable with content paths satisfying the CAS query
 */
Iterable<String> QueryQTP(String k, String v, String m, NodeStore store) {

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

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

    NodeState targetNodeState = getNode(targetNodePath, store);

    /* map index nodes' path to corresponding content nodes' path */
    return map((Node node) -> relativize(indexRootPath, node.path),

        /* filter non-matching index nodes */
        filter((Node node) -> {
                NodeState s = node.state;
                boolean matching = s.getBoolean("match");

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

            /* bottom-up tree traversal of descendants of target node */
            bottomUp(new Node(targetNodeState, targetNodePath))
        )
    );
}
```

Figure 12: Java implementation of QTP

```
Algorithm 2: QueryQTP
Data: Query $Q(k, v, m)$, where $k$ is a property, $v$ a value and $m$ a node.
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 bottom-up fashion do
    if $matching(d)$ then
        $r \longleftarrow r \cup \{*d\}$
    else if $chd(d) = \emptyset \wedge \neg volatile(d)$ then
        delete node $d$
return $r$
```

```java
/**
 * Class containing immutable information about a node's
 * state and path.
 */
class Node {
    final NodeState state;
    final String path;
    Node(NodeState state, String path) {
        this.state = state;
        this.path = path;
    }
}
```

Figure 13: `Node` class implementation

```java
/**
 * Returns an iterable which lazily traverses a subtree rooted at root
 * in a bottomUp fashion.
 *
 * @param root: Root node of subtree we apply traversal on
 * @returns: iterable for bottom-up tree traversal
 */
Iterable<Node> bottomUp(Node root) {
    return () -> {
        /* Stacks */
        Deque<Node> s1 = new LinkedList();
        Deque<Node> s2 = new LinkedList();
        s1.push(root);
        return new Iterator<Node>() {
            @Override
            public boolean hasNext() {
                return s1.size() > 0 || s2.size() > 0;
            }
            @Override
            public Node next() {
                while (s1.size() > 0 && (s2.size() == 0 ||
                          isAncestor(s2.peek().path, s1.peek().path))
                ) {
                    Node n = s1.pop();
                    s2.push(n);
                    for (ChildNodeEntry child : n.state
                                                 .getChildNodeEntries()
                    ) {
                        s1.push(new Node(
                            child.getNodeState(),
                            concat(n.path, child.getName())
                        ));
                    }
                }
                return s2.pop();
            }
        }
    }
}
```

Figure 14: `bottomUp()` 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 15: `map()` implementation

```java
/**
 * Higher order function that removes all elements from an iterable
 * not satisfying the predicate.
 *
 * @param predicate: The predicate that tests elements
 * @param iterable: The iterable whose members are tested against
 * @returns: An iterable with members satisfying the predicate
 */
Iterable<T> filter(Predicate<T> predicate, Iterable<T> iterable) {
    return () -> {
        Iterator<T> iterator = iterable.iterator();
        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 16: `filter()` implementation

```java
/**
 * Fetches a node given its path
 *
 * @param path: the node's path
 * @param store: Oak's database interface
 * @returns the node with the given path
 */
NodeState getNode(String path, NodeStore store) {
    NodeState n = store.getRoot();
    for (String lambda : elements(path)) {
        n = n.getChildNode(lambda);
    }
}
```

Figure 17: `getNode()` implementation

References

[1] 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.

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

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

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