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

D
B
T G

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. [**?**] 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 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. [**?**] 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 [**?**] 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) [?], a commonly used optimistic concurrency control technique [?].

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].
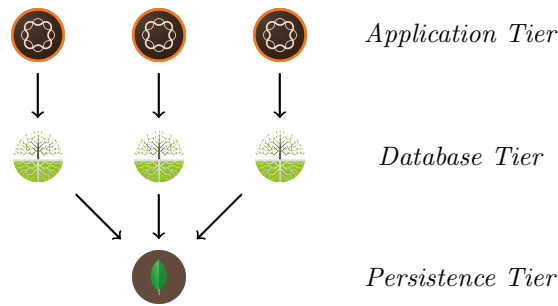

Figure 1: Apache Jackrabbit Oak's system architecture

2.2 Application Scenario

Content management systems that make use of Oak have specific workloads. These workloads have distinct properties: they are *skewed*, *update-heavy*, *changing* [?] and *job-queue* like.

Consider a social media feed as a running example. When browsing the feed, more recent posts are displayed first, therefore creating a skewed workload. Users submit new posts or interact with existing posts by writting 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, making the workload change 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 "pub" property with value "now" to the respective node in order to signal the background thread that the specific node is a pending job that needs processing. Once the background thread detects the node and finishes processing,

---

[1]https://www.mongodb.com/what-is-mongodb

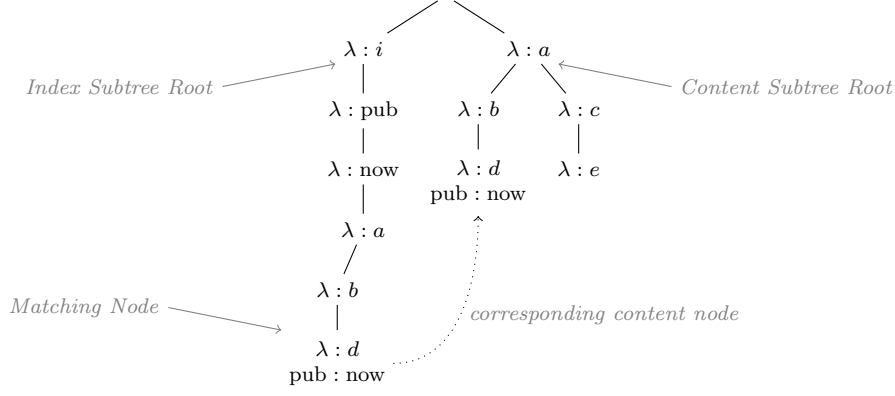[2]http://www.adobe.com/marketing-cloud/experience-manager.html

4

Figure 2: An instance of an hierarchical database

it removes the "pub" property and successfully publishes the post. Informally, we call such a workload job-queue like.

From now on, we 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 [**?**]. We denote node n's property k as $n[k]$ and node n's desendants 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

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

The WAPI is an hierarchical index and 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.

Example 1 (CAS Query). Consider Figure 2. CAS-Query $Q(\text{pub}, \text{now}, /\text{a})$, which queries for every descendant of /a with "pub" set to "now", would evaluate to $Q(\text{pub}, \text{now}, /\text{a}) = \{/\text{a}/\text{b}/\text{d}\}$, since /a/b/d is the only descendant of /a with "pub" set to "now".

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, then n's corresponding content node $*n$ must have path $path(*n) = \text{m} = $ /λ_1/ . . ./λ_d.

5

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 **(mainly ?)** used to signal pending jobs to the background thread using the "pub" property. Assuming jobs are processed by the background thead faster and removed from the index 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 when to remove a node or not from the index. Wellenzohn et al. [**?**] 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 commited tree of the datbase. 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.

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

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 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.

3 Unproductive Nodes

When time passes and the database workload changes, volatile nodes cease to be volatile and they become unproductive.

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

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

Example 4. In our running example (cf. Figure 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.
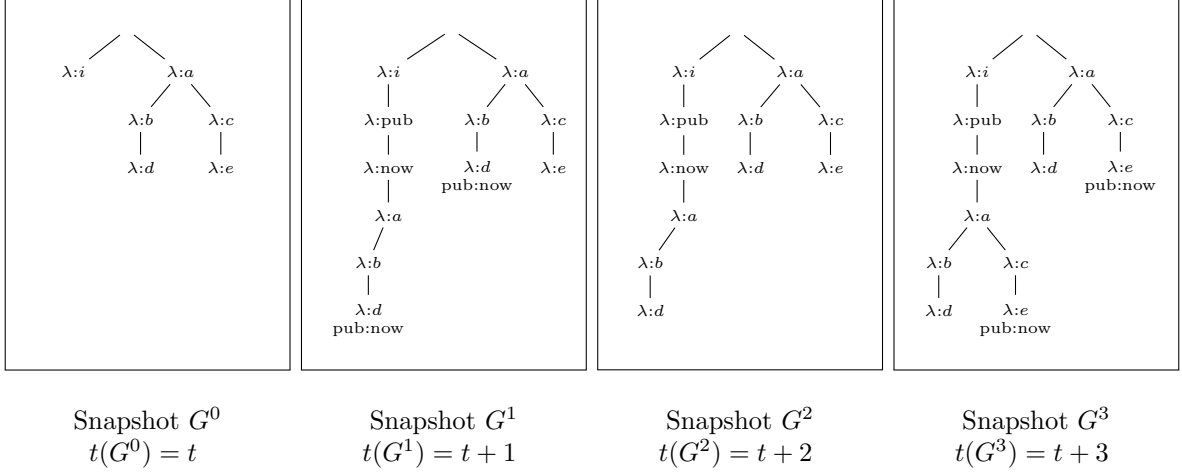
3.1 Impact on Query Execution Runtime

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 average query runtime under a CMS workload.

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

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

Given $\tau = 1$, $L = 1$, nodes `/i/pub/now/a/b/d` and `/i/pub/now/a/b` are unproductive in snapshot G^3. They are not volatile and don't match either.

Figure 3: Volatile nodes becoming unproductive

In order to find supporting evidence for the hypotheses above, a series of experiments was conducted on Oak. The experimental evaluation and the used datasets will be described in a later section (**forward reference**). We recorded the query execution runtime throughout the experiment and present the data below.

Figures 4a and 4b show query runtime from the synthetic and real world dataset respectively. We observe an increase of the runtime from $2ms$ to $38ms$ after running the simulation for 5 minutes ($2 \cdot 10^4$ update operations) on the synthetic dataset.

Next, we present data regarding the type of nodes encountered during index traversal while executing a query. Figures 5a and 5b depict the number of traversed volatile and unproductive index nodes encountered during query execution from our datasets. Since the sliding window length L is set to 30 seconds, 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. We observe two findings. First, we see a descend in volatile nodes. Second, we see a significant increase in unproductive nodes. Towards the end of the simulation, we observe the traversed nodes being dominated by unproductive 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. Interestingly, we see the ratios converging towards a value above 0.8 and below 0.2 respectively.
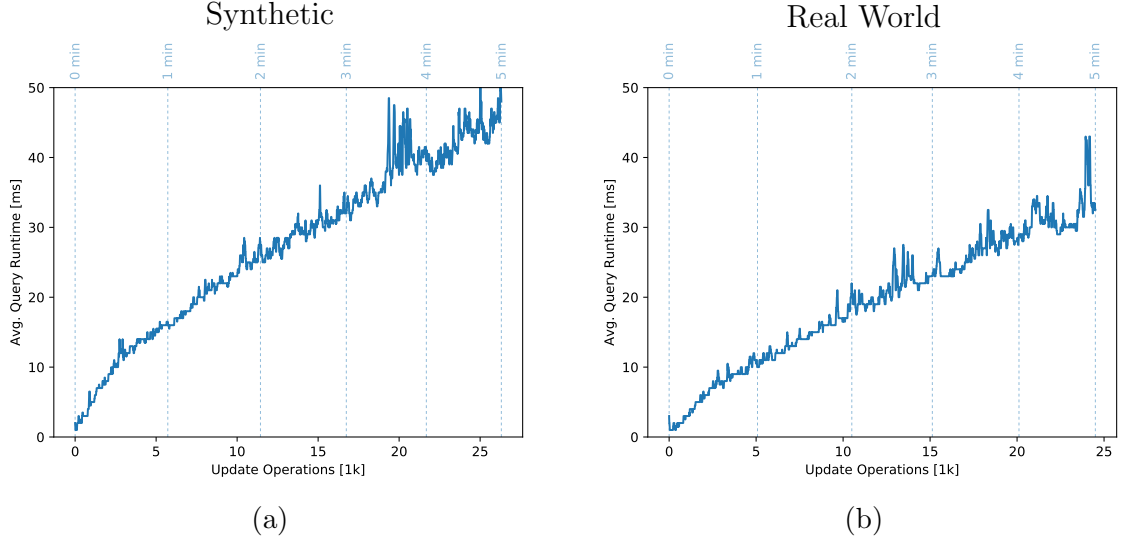
Figure 4: Query Runtime over time

In the following sections we will study the effect of *Volatility Threshold* τ and *Sliding Window Length L* on query execution 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 execution runtime.

We hypothesize that an increase in τ yields a decrease to the number of traversed unproductive nodes during query execution. If τ increases, it is less likely for a node to become volatile. Having less volatile nodes should imply a decrease in unproductive nodes.

An increase in τ yields a decrease to the query execution runtime. Less unproductive nodes should decrease the number of nodes traversed during query execution and therefore decrease query execution runtime.

H_3: An increase in τ yields a decrease to the number of traversed unproductive nodes during query execution.

H_4: An increase in τ yields a decrease to the query execution runtime.

The same experiment was conducted under a varying volatility threshold. The results seem to verify our hypotheses. Figures 7a and 7b show various thresholds $\tau \in \{1, 5, 10\}$ affecting query execution runtime over update operations. We observe that a lower
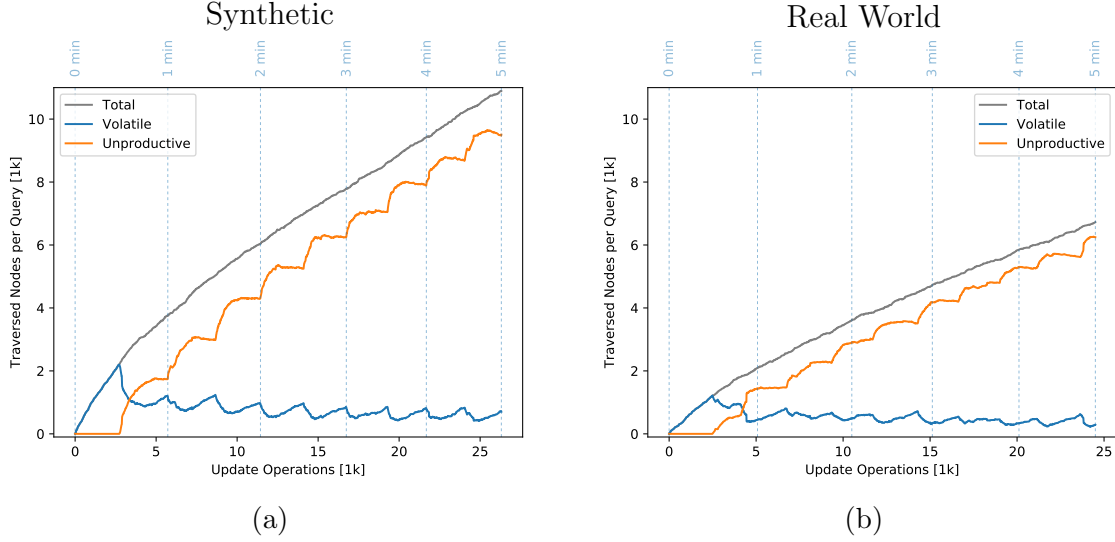
9

Figure 5: Index Structure during Query Execution

threshold τ results in a steeper slope. Figures 7c and 7d compare query execution runtime over a range of thresholds. The values picked correspond to the query execution runtime after 10^4 update operations. We observe a decrease in query execution runtime while increasing threshold τ. 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 traversed unproductive nodes during query execution over a range of thresholds. As mentioned earlier, the values were recorded after 10^4 update operations. We observe a decrease in unproductive nodes while increaseing threshold τ.

In conclusion, all observations verify hypotheses H_4, H_5 and H_6. Increasing volatility threshold τ decreases the number of unproductive nodes traversed, query runtime and unproductive node density. In the following section, we will take a look at sliding window length L, another factor affecting unproductive nodes.

3.3 Sliding Window Length L

Sliding window length L determines the length of the recent workload that WAPI considers to compute an index node's volatility count. In this section, we study the effect of sliding window length L on unproductive nodes and query execution runtime.

We hyptothesize 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. Having more volatile nodes should imply an increase in unproductive nodes.

An increase in L yields an increase to the query execution runtime. More unproductive nodes should increase the number of nodes traversed during query execution and therefore increase query execution time.
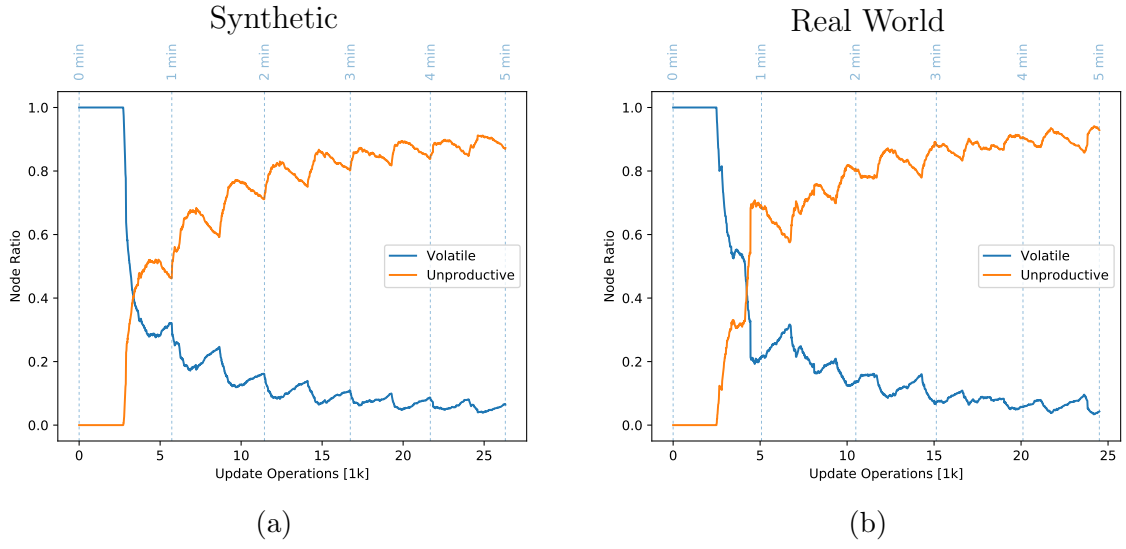
Figure 6: Node Ratio during Query Execution

H_6: An increase in L yields an inrease to the query execution runtime.

H_7: An increase in L yields an increase to the number of traversed unproductive nodes during query execution.
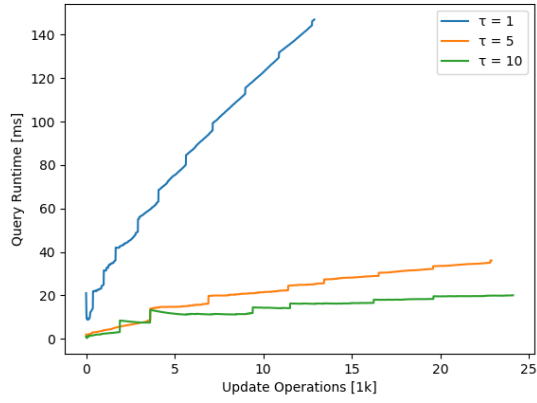
4 Periodic Garbage Collection (GC)

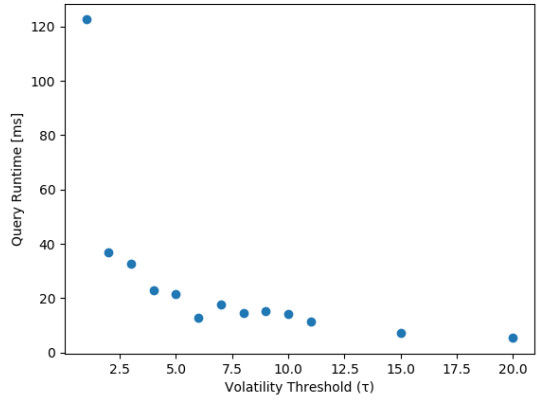# 5 Query Time Pruning (QTP)

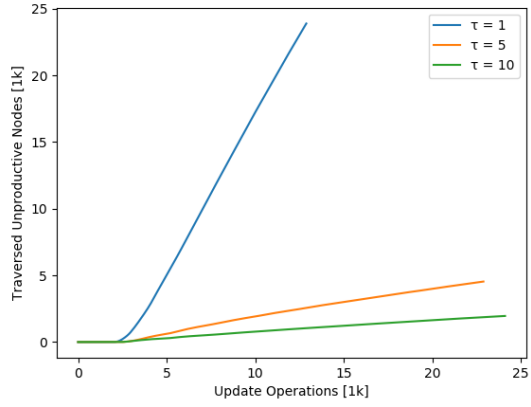# 6 Experimental Evaluation

11

Synthetic

Real World

(a)

(b)

(c)

(d)

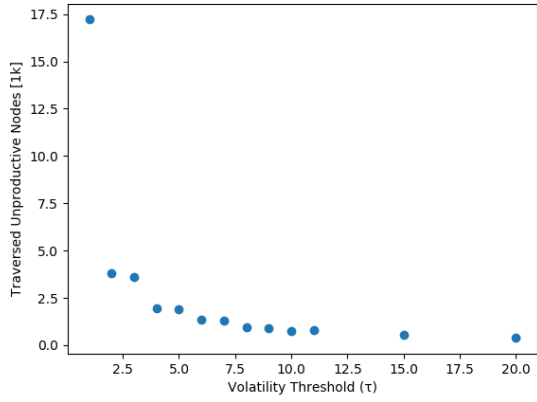Figure 7: Impact of Volatility Threshold τ on Query Execution Runtime

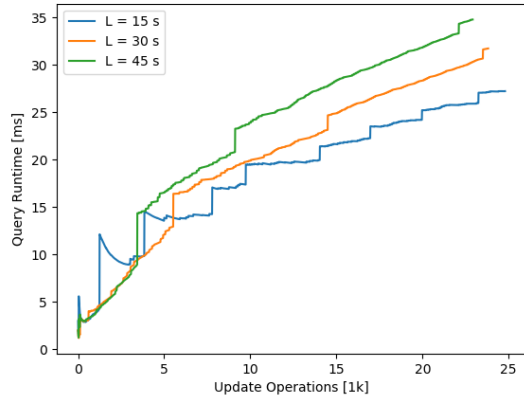Synthetic   Real World

(a)

(b)

(c)

(d)

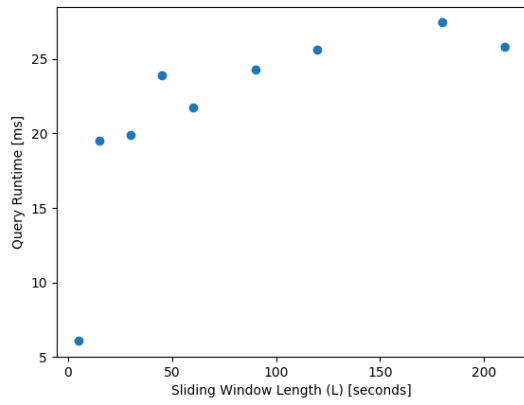Figure 8: Impact of Volatility Threshold τ on Unproductive Nodes

13

Synthetic

Real World

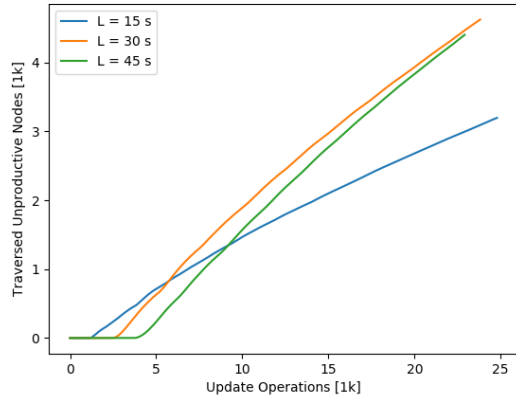(b)

(a)

(d)

(c)

Figure 9: Impact of Sliding Window Length L on Query Execution Runtime

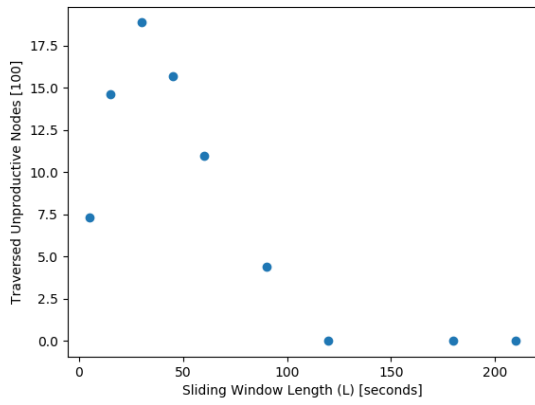Synthetic   Real World

(a)

(b)

(c)

(d)

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