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

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*. 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].

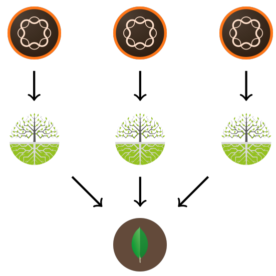


Figure 1: Apache Jackrabbit Oak's system architecture

2.2 Workload Aware Property Index (WAPI)

Oak mostly executes content-and-structure (CAS) queries [1], defined as follows.

Definition 1. (CAS-Query): Given node m, property k and value v, a CAS query $Q(k, v, m)$ returns all descendants of m which have k set to v, i.e

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

The WAPI is a hierarchical index and indexes the properties of nodes in order to answer CAS-queries efficiently. Additionally, it takes into account if an index node is volatile 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. [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. Let t_n be the current time

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

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. $pre(G^b)$ 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. Let n^i denote version i of node n that belongs to the node set $N(G^i)$ of snapshot G^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 2. (Volatility Count): The volatility count $vol(n)$ of node n is the number of times node n was added or removed from snapshots contained in a sliding window with length L over history H_i.

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

Definition 3. (Volatile Node): Node n is volatile iff n's volatility count (see Definition 2) is greater or equal than the volatility threshold τ, i.e

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

3 Unproductive Nodes

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

Definition 4. (Unproductive Node): Node n is unproductive iff n, and any descendant of n, is neither matching nor volatile, i.e

$$unproductive(n) \iff \nexists m(m \in (\{n\} \cup desc(n)) \wedge (volatile(m) \vee *m[k] \neq v))$$

Example 1. Consider the snapshots depicted in Figure 2. Assume $H_h = \langle G^0, G^1, G^2, G^3, G^4, G^5, G^6 \rangle$. O_h executes transactions $T_1, T_2, T_3, T_4, T_5, T_6$. Snapshot G^0 was committed at time $t(G^0) = t$. Given snapshot G^0, transaction T_1 adds property $x = 1$ `/a/b/d` and commits snapshot G^1 at time $t(G^1) = t + 1$. Next, transaction T_2 removes property x 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. Transaction T_3 adds property $x = 1$ to `/a/c/e` given G^2 and commits G^3 at time $t(G^3) = t + 3$. Notice how `/i/x/1/a/b/d` and `/i/x/1/a/b` are the only unproductive index nodes and `/i/x/1/a/c/e` as well as `/i/x/1/a/c/e` are the only volatile index nodes in G^3. Transaction T^4 again adds property $x = 1$ to `/a/b/d` given snapshot G^3 and commits snapshot

G^4 at time $t(G^4) = t + 4$. In G^4, nodes `/i/x/1/a/b/d` and `/i/x/1/a/b` are not unproductive anymore since `/i/x/1/a/b/d`'s content node is matching. Transaction T^5 again removes property x from `/a/b/d` given snapshot G^4 and commits snapshot G^5 at time $t(G^5) = t + 5$. Since nodes `/i/x/1/a/b/d` and `/i/x/1/a/b` are not volatile, they are pruned from the property index during T^5. Finally transaction T^6 removes property $x = 1$ from `/a/c/e` given snapshot G^5 and commits snapshot G^6 at time $t(G^6) = t + 6$. Index nodes `/i/x/1/a/c/e` and `/i/x/1/a/c` are pruned from the property index during T^6 because they are not volatile.

3.1 Impact on Query Execution Runtime

In this section we will study and quantify the impact of unproductive nodes on query runtime. Informally, we call *Query Execution Runtime* the time needed for a query to finish processing. We hypothesize that unproductive nodes slow down queries. During query execution, traversing an unproductive node is useless, because neither the node itself nor any of its descendants contains an indexed property and therefore cannot contribute a query match.

We formalize the statements above into the following hypotheses:

H_1: Query execution runtime increases over time.

H_2: Unproductive nodes account for the increase in query execution runtime.

In order to find supporting evindence for the hypotheses above, a series of experiments was conducted on Oak. We recorded the query execution runtime throughout the experiment and present the data below.

Figures 3a and 3b show the recorded query runtime over time as observed from the synthetic and AEM dataset respectively. Figures 3c and 3d show the recorded query runtime over update operations from the synthetic and AEM 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 4a to 4d depict the number of traversed volatile and unproductive index nodes encountered during query execution with respect to time and update operations from our datasets. Since the sliding window length L is set to 30 seconds, we record 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 4e to 4h show the density 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 density functions to converge towards a value above 0.8 and below 0.2 respectively.

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 [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.

An increase in τ yields a decrease to the unproductive node density during query execution.

Why?

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.

H_5: An increase in τ yields a decrease to the unproductive node density during query execution runtime.

The same experiment was conducted under a varying volatility threshold. The results seem to verify our hypotheses. Figures 5a and 5b show various thresholds $\tau \in \{1, 5, 10\}$ affecting query execution runtime over update operations. We observe that a lower threshold τ results in a steeper slope. Figures 5g and 5h 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 5c and 5d 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 5i and 5j 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. Figures 5e and 5f Figures 5k and 5l

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.

An increase in L yields an increase to the unproductive node density during query exection. **Why?**

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.

H_8: An increase in L yields an increase to the unproductive node density during query execution runtime.

4 Periodic Garbage Collection (GC)

5 Query Time Pruning (QTP)

6 Experimental Evaluation

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.

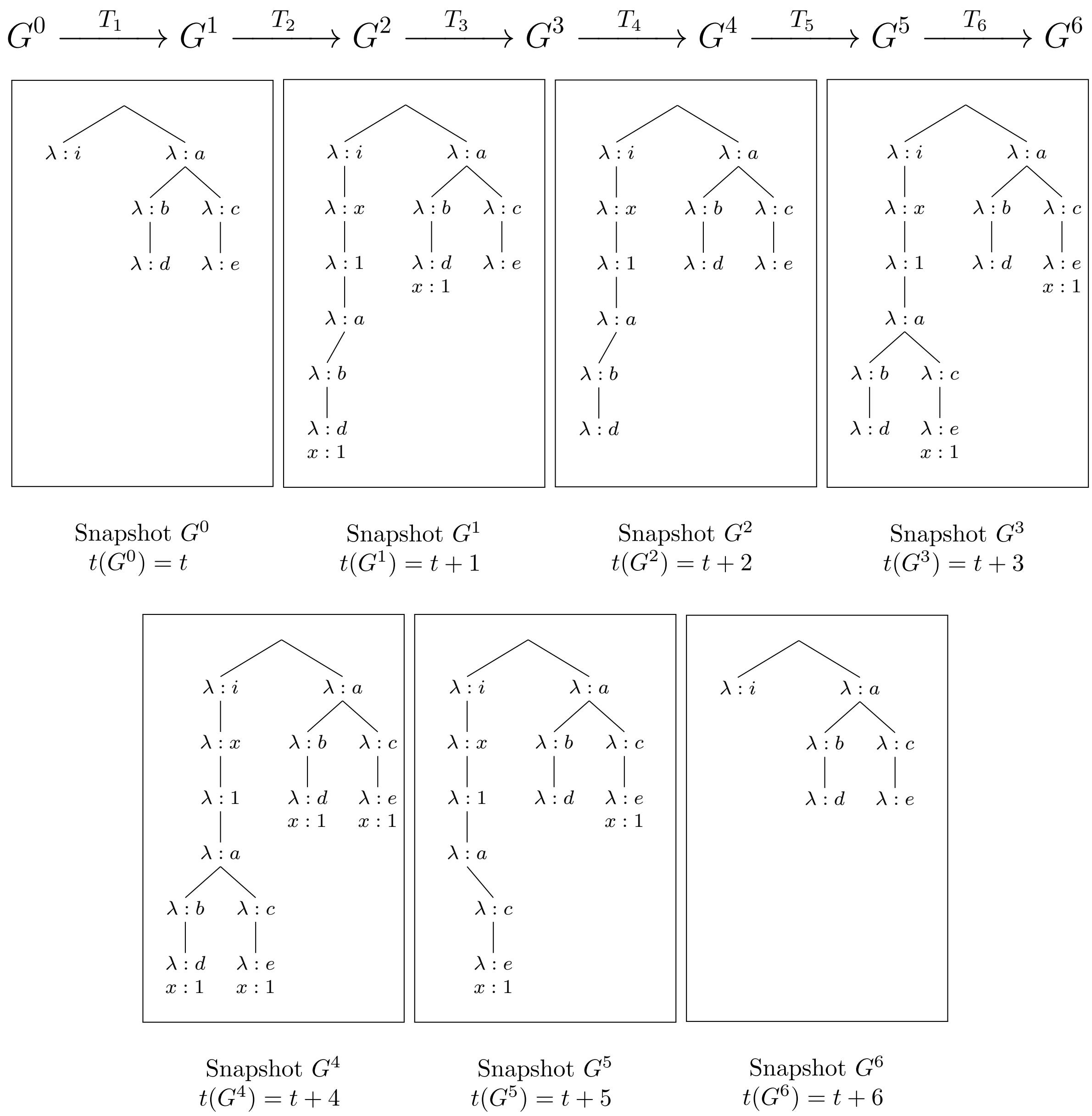


Figure 2: Unproductive Nodes Example

Nodes `/i/x/1/a/b/d` and `/i/x/1/a/b` are unproductive in snapshot G^3. They are not volatile and don't match either.

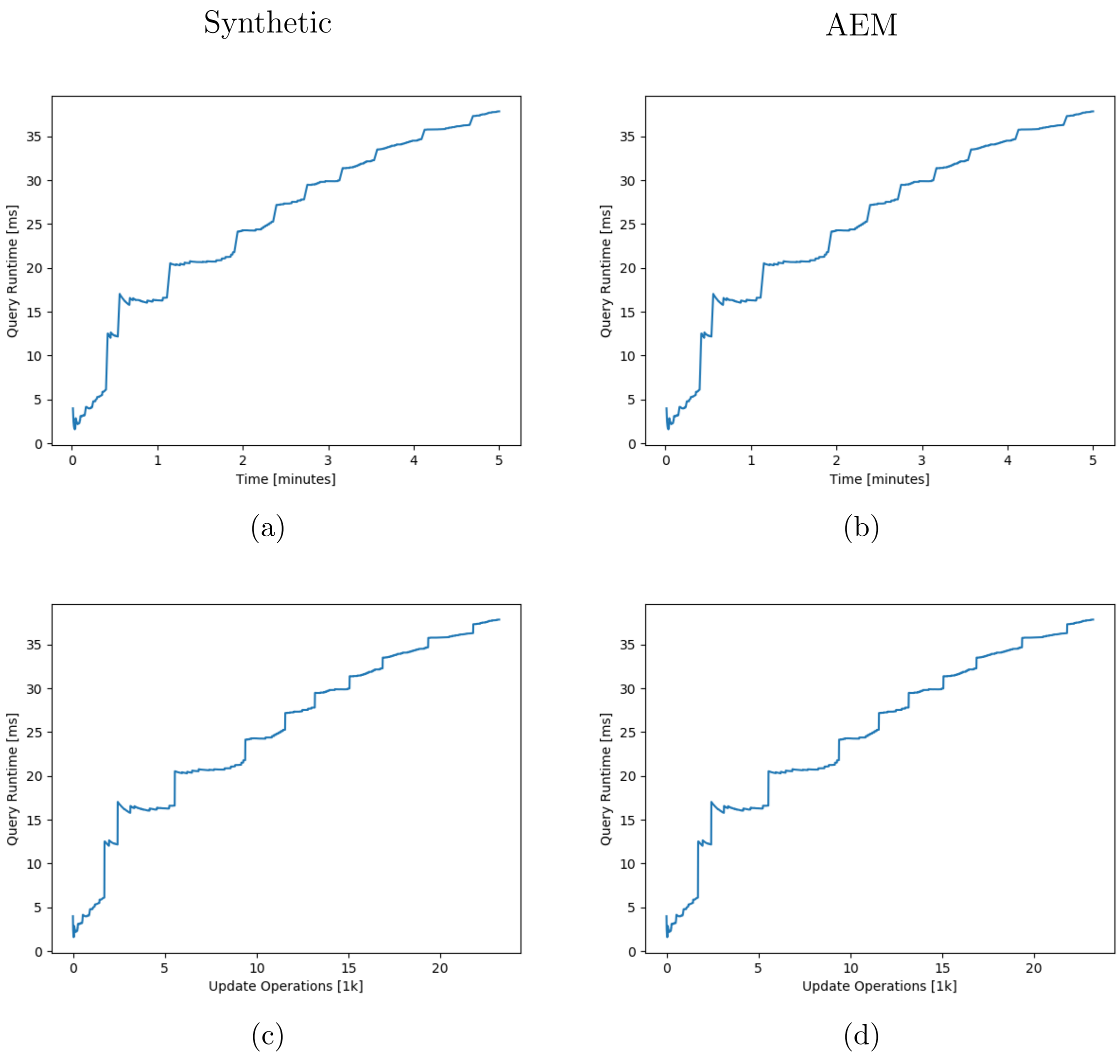


Figure 3: Query Execution Runtime

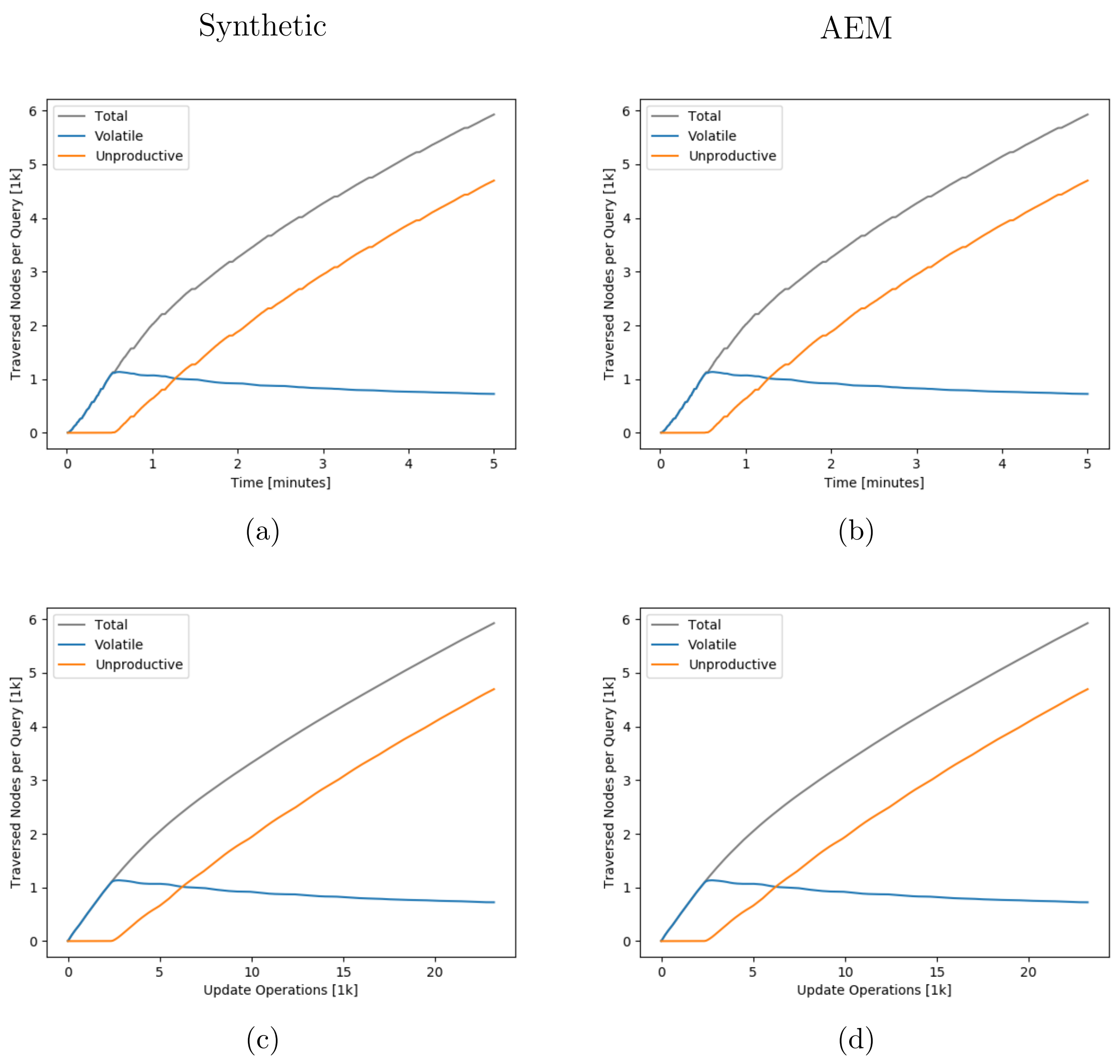
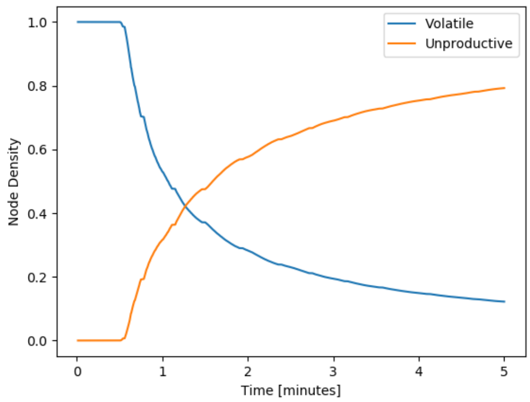
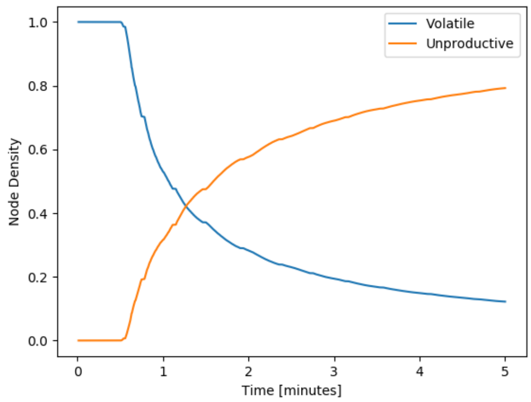


Figure 4: Index Structure during Query Execution

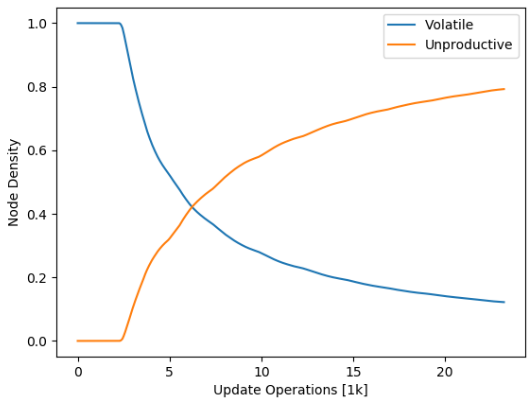
Synthetic AEM



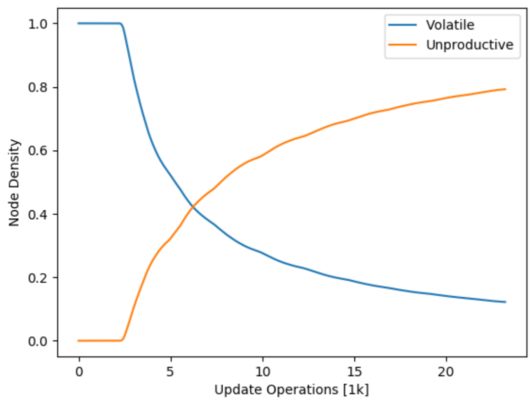
(e)



(f)



(g)



(h)

Figure 4: Index Structure during Query Execution (cont.)

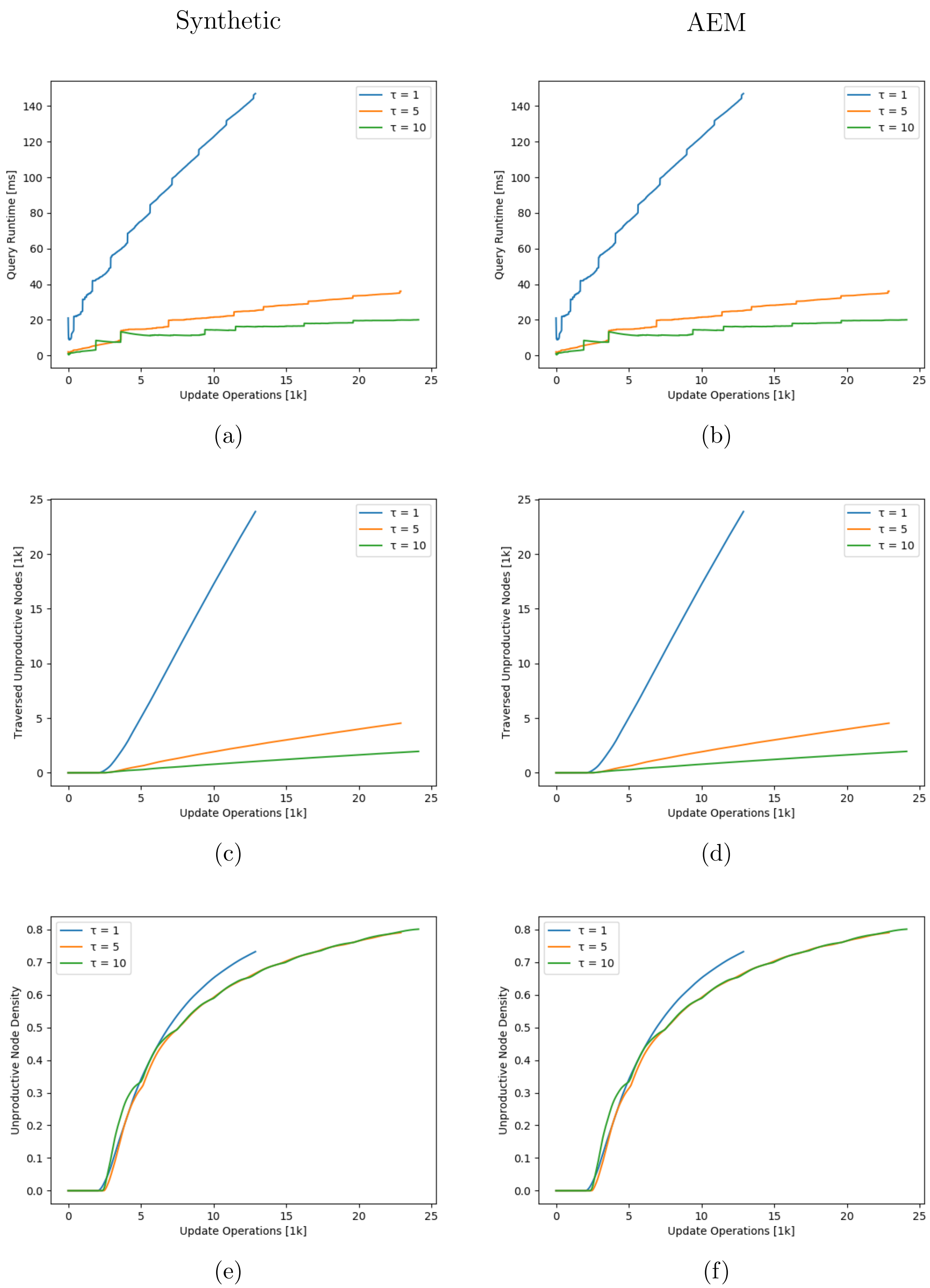


Figure 5: Impact of Volatility Threshold τ

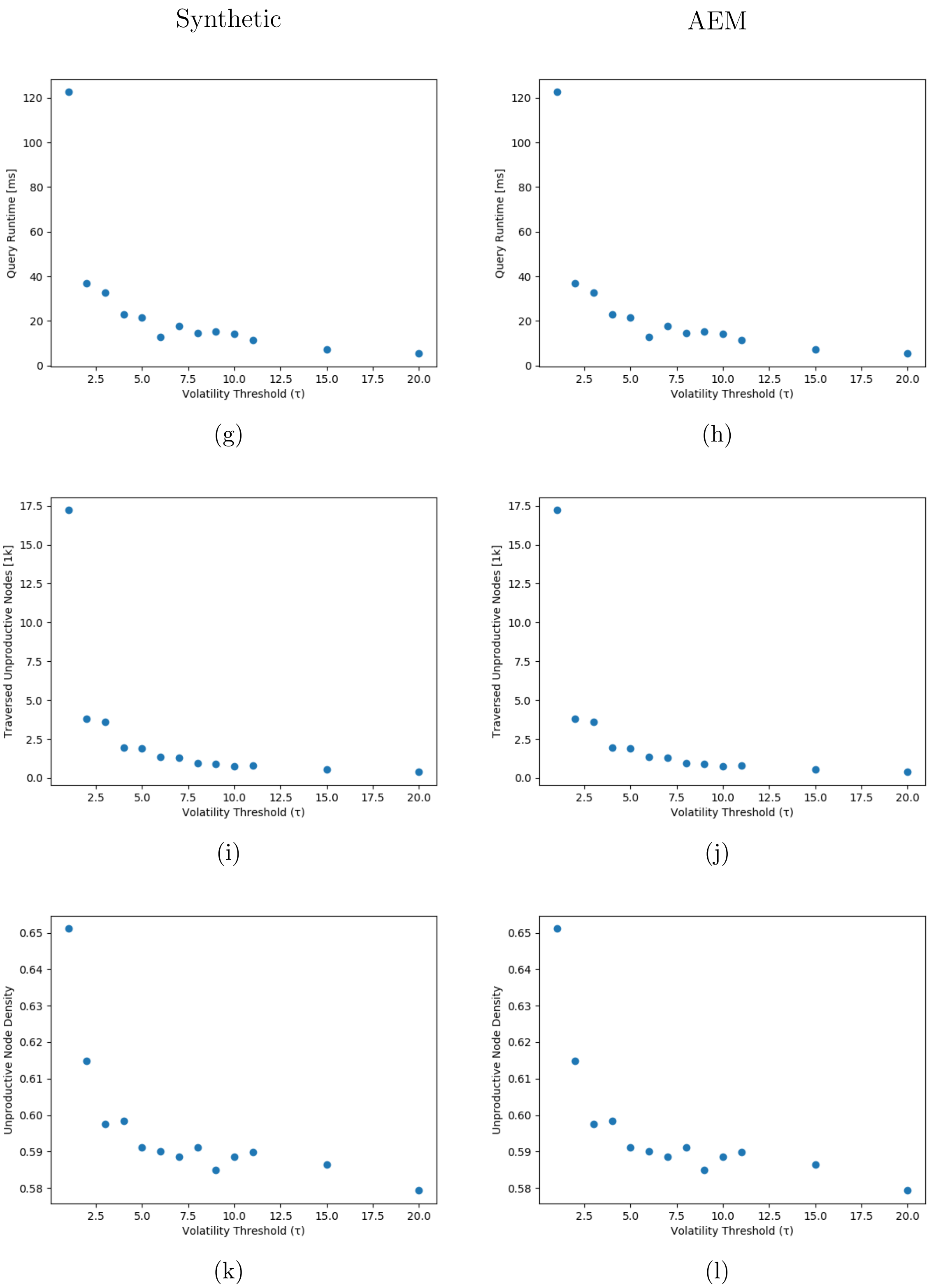


Figure 5: Impact of Volatility Threshold τ (cont.)

The values on the y axis are observations derived from the 10000th update operation.

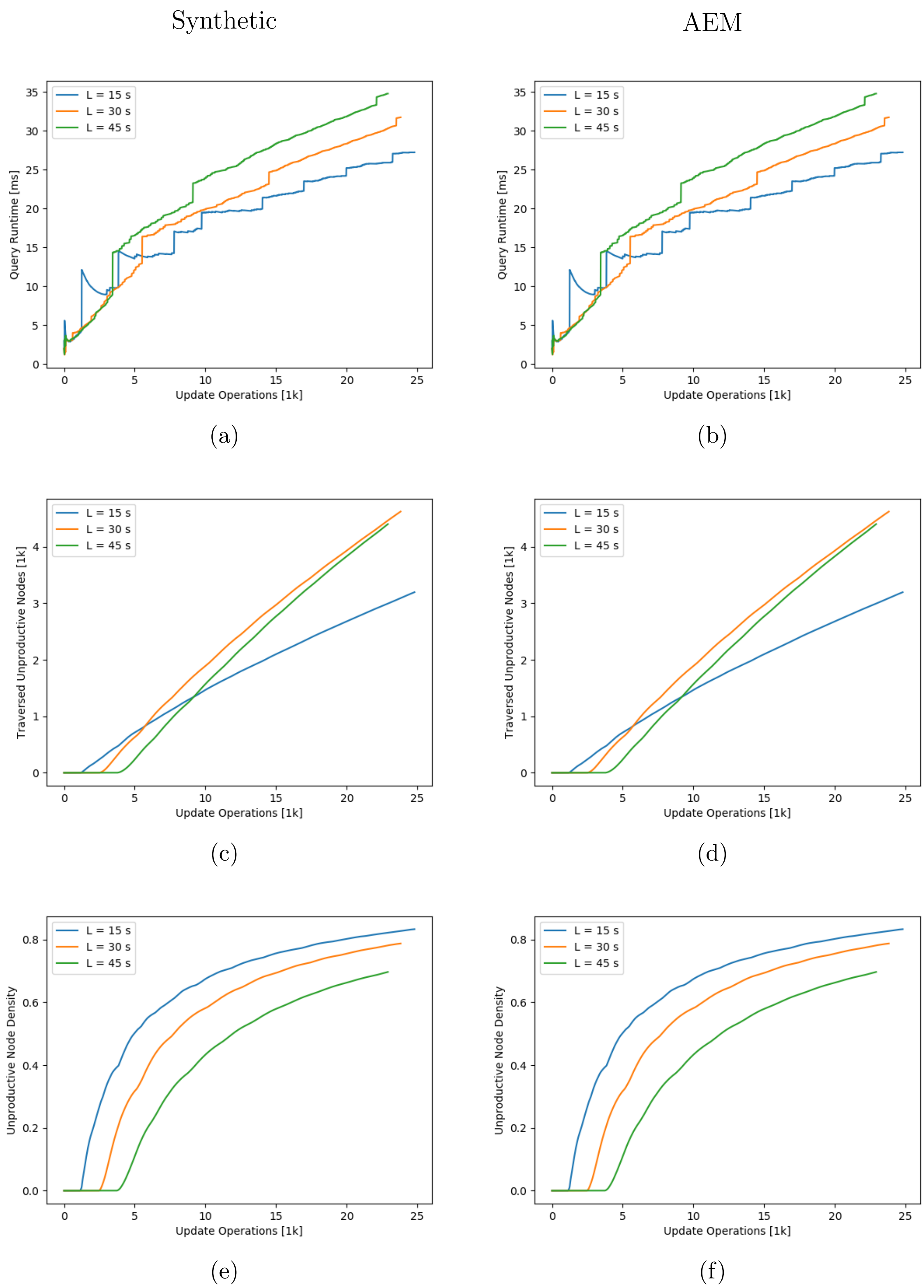


Figure 6: Impact of Sliding Window Length L

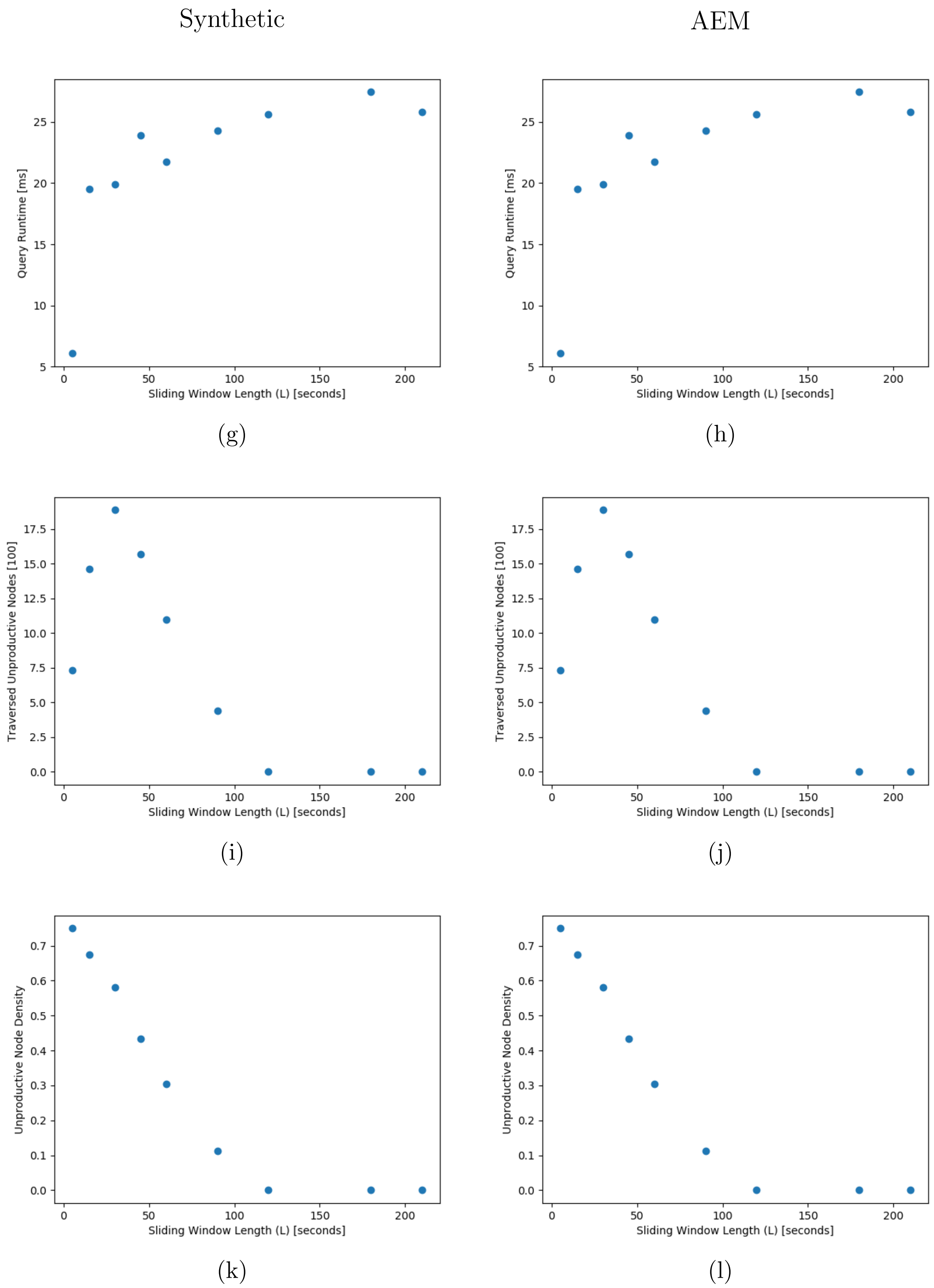


Figure 6: Impact of Sliding Window Length L (cont.)

The values on the y axis are observations derived from the 10000th update operation.