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

Department of Informatics

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].

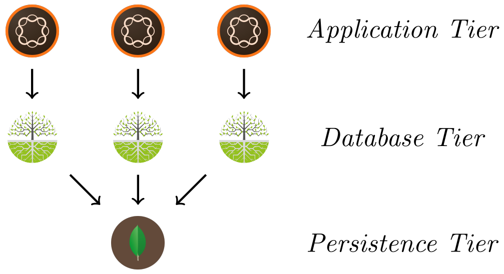


Figure 1: Apache Jackrabbit Oak's system architecture.

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.

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

Definition 3. (Volatile Node): Node n is volatile iff n's volatility count (see Definition 5) is greater or equal than the volatility threshold τ, i.e

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

3 Simulating a Changing Workload

In order to empirically evaluate and compare GC and QTP under a changing workload, a experiment harness has to be setup.

datasets

zipf distribution

changing Workload

parameters

4 Unproductive Nodes

time / index nodes

time / volatile nodes

time / unproductive nodes

time / query execution runtime

time / update execution runtime

unproductive nodes / query execution runtime

unproductive nodes / update execution runtime

$\tau / unproductivenodes$

L / unproductive nodes

5 Periodic Garbage Collection (GC)

6 Query Time Pruning (QTP)

7 Benchmarks

8 OLD STUFF

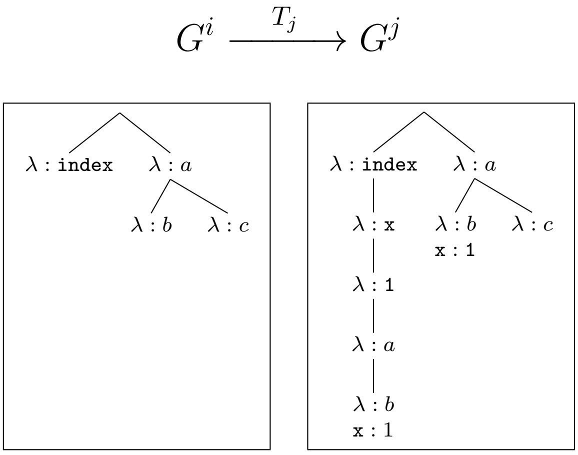
9 Workload Aware Property Index (WAPI)

The WAPI is a hierarchical index and indexes the properties of nodes. 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. In the following section, we will see how to add, query and remove nodes from the index.

9.1 Insertion

The WAPI is hierarchically organized under `/index` node. 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. Some node m is added to the WAPI iff m has a property k set to v, as shown in Algorithm 1. Starting from `/index`, we descend down to `/index/k`, followed by `/index/k/v`. Next, we descend down from `/index/k/v` along the index nodes on the absolute path from the root to node m. While we descend the WAPI, we create any node n that does not exist and assign it to variable *tail*. At the end of the tree traversal, *tail* corresponds to index node `/index/k/v/m`. *tail*'s property k is finally set to v.

Example 1. Consider Figure 2. Given snapshot G^i, transaction T_j adds the property-value pair $x = 1$ to `/a/b` and commits snapshot G^j. Starting from `/index` and descending down to `/index/x/1/a/b`, we create each node on the way since they do not exist yet. Finally, we set property $x = 1$ on `/index/x/1/a/b`.



Snapshot G^i

Snapshot G^j

```
Algorithm 1: AddTripleWAPI
Data: Triple $(k, v, m)$, where $k$ is a
      property, $v$ a value and $m$ a content
      node.
$tail \longleftarrow$ /index
for $n \in \langle$/i/k, /i/k/v, ..., /i/k/v/m$\rangle$ do
    if $n$ does not exist then
        create node $n$
    $tail \longleftarrow n$
$tail[k] \longleftarrow v$
```

Figure 2: Adding a node in WAPI. `/i` is an abbreviation for `/index`. Property λ denotes the label of a node.

9.2 Querying

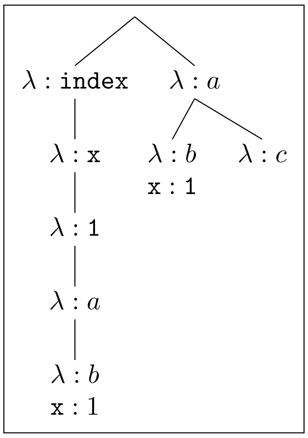
Oak mostly executes content-and-structure (CAS) queries [1], defined as follows.

Definition 4. (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

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

Algorithm 2 describes how we answer a CAS query using the WAPI. Given property k, value v and node m, we start descending down to node `/index/k/v/m`. Next, we iterate through all its descendants n. We return a set consisting of *content* nodes $*n$ corresponding to every *index* node n with property k set to v. The path of content node $*n$ is obtained by removing the first three nodes on the path of index node n. For example, if $n =$ `/index/x/1/a/b` is an index node, the corresponding content node is $*n =$ `/a/b`. If `/index/k/v/m` does not exist, then $desc($`/index/k/v/m`$) = \emptyset$.

Example 2. Consider $Q(x, 1, $`/a`$)$, which queries for every descendant of `/a` with `x` set to 1. Assuming we execute the query on the tree depicted in Figure 3, WAPI descends to node `/index/x/1/a` and traverses all descendants. Its only descendant is $n =$ `/index/x/1/a/b` and since $n[x] = 1$ the content node $*n =$ `/a/b` is returned. That is, $Q(x, 1, $`/a`$) = \{$ `/a/b` $\}$



Algorithm 2: QueryWAPI

```
Data: Query $Q(k, v, m)$, where $k$ is a property, $v$
      a value and $m$ a node.
Result: A set of nodes satisfying $Q(k, v, m)$
$r \longleftarrow \emptyset$
for $n \in desc($/index/k/v/m$)$ do
    if $n[k] = v$ then
        $r \longleftarrow r \cup \{*n\}$
return $r$
```

Where *desc*(`/index/k/v/m`) is the set of descendants of node `/index/k/v/m`, $n[k]$ is property k of node n and $*n$ is the content node corresponding to n.

Figure 3: CAS Query example.

9.3 Deletion

If a property-value pair is deleted from a content node, the corresponding index entry is deleted from WAPI. Volatile nodes influence the logic of the deletion process. A workload aware property index detects which nodes are volatile and does not remove them. The process of classifying a node as volatile, will be explained in more details in Section 10. For the moment we assume that a function $isVolatile(n)$ is given that classifies n either as volatile or as non-volatile.

Algorithm 3 describes the process of removing node $n =$ `/index/k/v/m` from WAPI after property k was changed or removed from content node m. We first descend down to node $n =$ `/index/k/v/m`, which we intend to remove. We remove property k from n by setting k's value to NIL. If n is (a) a leaf node, and (b) does not have property k set to v and (c) is not volatile, we remove it. If n was removed, we repeat the process on its parent node $par(n)$. The process repeats on all ancestors and ends if we propagate up to `/index` or reach a node that violates at least one of the above three conditions.

Example 3. Figure 4 depicts the following scenario. Assume `/index/x/1/a/b` (colored red) is volatile in all three snapshots G^i, G^j, G^k. Given snapshot G^i, transaction T_j removes property $x = 1$ from `/a/b` and commits snapshot G^j. Since `/index/x/1/a/b` is volatile, it was not removed from the WAPI, only its property $x = 1$ is removed. Given snapshot G^j, transaction T_k removes property `x` from `/a/c` and commits snapshot G^k. Since `/index/x/1/a/c` was not volatile, was a leaf-node, and property $x = 1$ was just removed, the index node was removed from the WAPI. Since its parent node has another child node, the parent is not removed and the deletion process stops.

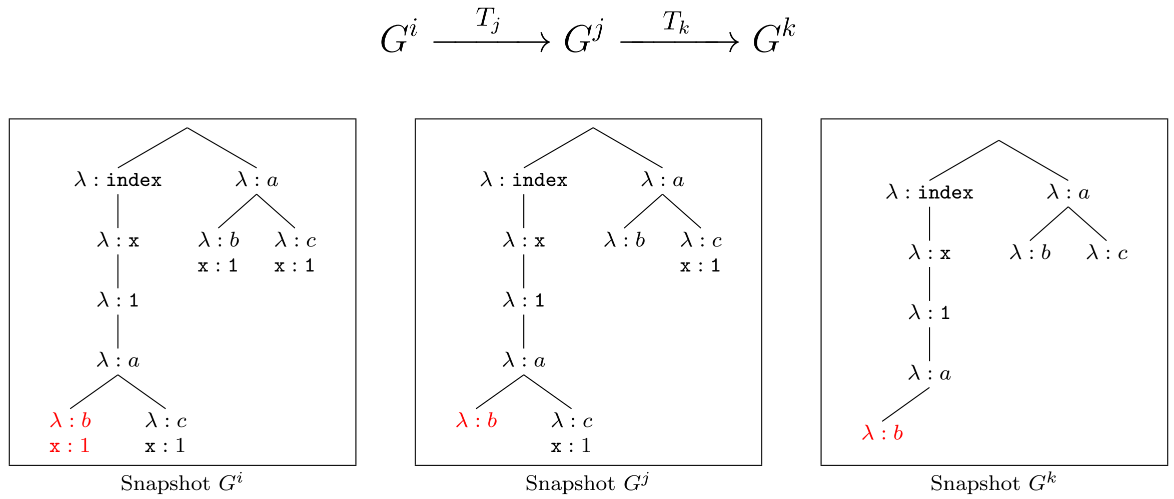


Figure 4: Removing a node from the WAPI. Assume `/index/x/1/a/b` (colored red) is volatile in all three snapshots G^i, G^j, G^k.

Algorithm 3: RemoveTripleWAPI

```
Data: Triple $(k, v, m)$, where $k$ is a property, $v$ a value and $m$ a node.
$n \longleftarrow$ /index/k/v/m
$n[k] \longleftarrow$ NIL
while $n \neq$ /index $\wedge\ chd(n) = \emptyset \wedge n[k] \neq v \wedge \neg\ isVolatile(n)$ do
    $u \longleftarrow n$
    $n \longleftarrow par(n)$
    remove node $u$
```

Where $chd(n)$ is the set of children of node n and $par(n)$ is the parent of n.

10 Volatility

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 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 5. (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.

$$\begin{aligned} vol(n) = |\{G^b | G^b \in H_i \wedge t(G^b) \in [t_{n-L+1}, t_n] \wedge \exists G^a [\\ G^a = pre(G^b) \wedge ([n^a \notin N(G^a) \wedge n^b \in N(G^b)] \vee \\ [n^a \in N(G^a) \wedge n^b \notin N(G^b)])]\}| \end{aligned} \tag{2}$$

Definition 6. (Volatile Node): Node n is volatile iff n's volatility count (see Definition 5) is greater or equal than the volatility threshold τ, i.e

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

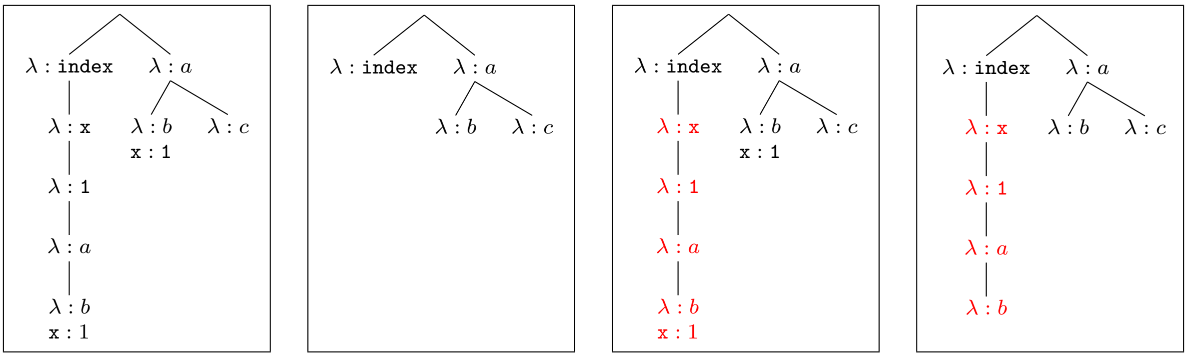
Example 4. Consider the snapshots depicted in Figure 5. Assume $H_h = \langle G^i, G^j, G^k, G^l \rangle$. O_h executes transactions T_j, T_k, T_l. Snapshot G^i was committed at time $t(G^i) = t$. Given snapshot G^i, transaction T_j removes property x from `/a/b` and commits snapshot G^j at time $t(G^j) = t + 1$. Next, transaction T_k adds the property $x = 1$ to `/a/b` given snapshot G^j and commits snapshot G^k at time $t(G^k) = t + 2$. Finally transaction T_l removes property `x` from `/a/b` given G^k and commits G^l at time $t(G^l) = t + 3$.

If $\tau = 2$ (volatility threshold), $L = 4$ (sliding window length) and $n =$ `/index/x/1/a/b`, then:

- at time $t_n = t$ we have that: $vol(n) = 0 \implies isVolatile(n) = \perp$
- at time $t_n = t + 1$ we have that: $vol(n) = 1 \implies isVolatile(n) = \perp$
- at time $t_n = t + 2$ we have that: $vol(n) = 2 \implies isVolatile(n) = \top$
- at time $t_n = t + 3$ we have that: $vol(n) = 2 \implies isVolatile(n) = \top$

Since index node n is not volatile at $t_n = t$, transaction T_j removes it from the index. But at $t_n = t + 2$, n is volatile (colored red) and transaction T_l does not remove it, instead it only removes property $x = 1$ from n.

$$\begin{tikzcd} G^i \arrow[r, "T_j"] & G^j \arrow[r, "T_k"] & G^k \arrow[r, "T_l"] & G^l \end{tikzcd}$$



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

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

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

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

Figure 5: Node `/index/x/1/a/b` becomes volatile after a deletion by T_j and insertion by T_k. Therefore the nodes cannot be deleted by transaction T_l.

11 Implementation

11.1 Checking Node Volatility

In order to classify a node n as volatile, we have to compute n's volatility count using the corresponding JSON document on MongoDB. The document contains all *revisions* (i.e, versions) of n throughout history H_i of an Oak instance O_i. Figure 6 depicts such a document. We omit non relevant properties. Property `"_deleted"` contains key value pairs which encode when the node was added or removed from snapshots. A key is a revision that is composed of three parts connected by a dash (-): (1) a timestamp, (2) a counter that is used to differentiate between value changes at the same instance of time, and (3) the identifier i of the Oak instance O_i committing the change. A value is a boolean variable which is `true` ($\top$) iff n was added at the point of time indicated by the revision key, and `false` ($\perp$) otherwise.

Example 5. Consider revision (`r15cac0dbb00-0-2`, `false`) in Figure 6. Character `r` is a standard prefix and can be neglected. The `15cac0dbb00` following `r`, is a timestamp (number of milliseconds since the Epoch) in hexadecimal encoding which represents the time at which the change was committed, Thursday June 15 2017 2:00:00 PM in this example. The `0` following the timestamp, is a counter which is used for tie-breaking between transactions committed at the same instance of time. Since all revisions in this example are committed at different time points, all counters are 0. The `2` following the counter, tells that the change was committed by Oak instance O_2 with an ID of 2. Value `false` indicates `/index/x/1/a/b` was added at that point of time.

Having seen what a node document looks like, we can now describe how we classify

```
{
   "_id": "5:/index/x/1/a/b",
   "_deleted": {
      "r15cac0dbb00-0-2": false,
      "r15cabff1500-0-2": true,
      "r15ca9f191c0-0-1": false,
      /* ... */
   },
   /* ... */
}
```

Figure 6: JSON document of an index node.

```
         \begin{minted}{java}
/**
   * Determines if node is volatile.
   * @param nodeDocument: document of node.
   * @returns true iff node is volatile.
   */
   boolean isVolatile(NodeDocument nodeDocument) {

   int vol = 0;

   for (Revision r : nodeDocument.getLocalDeleted().keySet()) {
      if (!isInSlidingWindow(r)){
         break;
      }
      if (!isVisible(r)){
         continue;
      }
      if (vol++ >= getVolatilityThreshold()) {
         break;
      }
   }
   return vol >= getVolatilityThreshold();
}
\end{minted}
```

Figure 7: Java implementation for detecting volatile index nodes.

a node as volatile. Figure 7 shows the native Java-implementation of $isVolatile(n)$ in Oak. $isVolatile(n)$ is given a node's corresponding JSON document. We iterate through the revisions of property `"_deleted"` in most-recent first fashion. Notice that the `keySet` referred to in the Java code is based on Java's ordered sets and is maintained in descending order according to the revision. If a revision is outside the sliding window we stop iterating because remaining revisions cannot be more recent. We increment the volatility count for every visible revision. A revision is visible if it is contained in the Oak instance's history. If the volatility count reaches at least τ we break the loop. When exiting the loop, we finally check if the volatility count is at least τ and return the result. The Java-implementation of the helper functions `isVisible` and `isInSlidingWindow` is provided in the Appendix.

11.2 Document Splitting

Jackrabbit Oak periodically checks a node's corresponding document for its size and if necessary splits it up and moves old data to a new split document. Performance suffers if looking at split documents is required while computing the volatility count, since MongoDB has to be accessed in order to lookup the split document. We modify Oak's document splitting implementation as mentioned in [4] in order to prevent split document lookups. Essentially, all necessary information is kept in a document to compute the volatility count. A race condition could occur if multiple Oak instances split the same document at the same time. In order to avoid such a race condition, each Oak instance only moves changes committed by itself.

Figure 8 depicts the Java implementation of the document splitting process. We iterate through the revisions of the `"_deleted"` property of the given document in most-recent first fashion. A revision gets moved to the split document iff: (1) it is not the most recent revision committed by the local Oak instance, (2) it is not among the τ first visible revisions contained in the sliding window, and (3) the revision was committed by the local Oak instance.

Example 6. Consider Figure 9. We see how a node's corresponding document is split on Oak instance O_1, assuming $\tau = 3$, $t_{\texttt{last\_sync}} = \text{2017.06.15 13:59}$, $L = 24$ hours, $t_n = \text{2017.06.15 14:01}$. Figure 10 shows a table with intermediate values during computation. "$t(r)$" is the point in time revision r was committed. Only the day, hours and minutes are shown for brevity. "$c(r)$" is the ID of the cluster node that committed revision r. "Vis." is true iff the revision is visible to the local cluster node. "$\in$Win." is true iff the revision is in the sliding window. "Vol." represents the volatility count at that step of the iteration. "Split" is true iff the revision is moved to the split document.

We will briefly walk through the iterations during the document split depicted in Figure 9. Revision r^1 is not visible to the local Oak instance O_1 because it was committed on O_2 after the last synchronization, i.e. $t_{\texttt{last\_sync}} < t(r^1)$. Therefore, r^1 does not increment the volatility count and is not moved to the split document. The three next revisions, r^2, r^3, r^4, increment the volatility count because they are in the sliding window but are not moved to the split document because $vol \leq \tau$. r^5 is still in the sliding window and therefore increments the volatility count. Since there are already τ revisions in the document and r^5 was committed on O_1, we move r^5 to the split document. Finally, any following revisions committed by the local Oak instance (i.e, r^7, r^8, r^9) are moved to the split document since there are already τ revisions in the document, enough to decide the node's volatility.

```
            \begin{minted}{java}
/**
* Splits the "_deleted" property on the given document.
* @param NodeDocument the node document.
*/
void splitDeleted(NodeDocument nodeDocument) {

    int vol = 0;
    boolean first = true;

    for (Revision r : nodeDocument.getLocalDeleted().keySet()) {
        if (first && r.getClusterId() == getClusterId()) {
            first = false;
            if (isInSlidingWindow(r)) {
                ++vol;
            }
            continue;
        }
        if (isInSlidingWindow(r) && isVisible(r) && vol++ < getVolatilityThreshold()) {
            continue;
        }
        if (r.getClusterId() != getClusterId()) {
            continue;
        }
        moveToSplitDocument(r);

    }
}
            \end{minted}
```

Figure 8: Java implementation for splitting the node document.

```
{
    "_id": "5:/index/x/1/a/b",
    "_deleted": {                 /* DD HH:MM */
      "r15cac0dbb00-0-2": false,  /* 15 14:00 */
      "r15cabff1500-0-2": true,   /* 15 13:44 */
      "r15ca9f191c0-0-1": false,  /* 15 04:10 */
      "r15ca76fc8e0-0-1": true,   /* 14 16:29 */
      "r15ca73b9980-0-1": false,  /* 14 15:32 */
      "r15ca5e9c520-0-2": true,   /* 14 09:23 */
      "r15ca5a8c480-0-1": false,  /* 14 08:12 */
      "r15ca5a6efc0-0-1": true,   /* 14 08:10 */
      "r15ca58e37a0-0-1": false   /* 14 07:43 */
    },
    /* ... */
}
```

(a) Before splitting.

```
{
    "_id": "5:/index/x/1/a/b",
    "_deleted": {                 /* DD HH:MM */
      "r15cac0dbb00-0-2": false,  /* 15 14:00 */
      "r15cabff1500-0-2": true,   /* 15 13:44 */
      "r15ca9f191c0-0-1": false,  /* 15 04:10 */
      "r15ca76fc8e0-0-1": true,   /* 14 16:29 */
      "r15ca5e9c520-0-2": true    /* 14 09:23 */
    },
    /* ... */
},
{
    "_id": "6:p/index/x/1/a/b/r15ca58e37a0-0-1",
    "_deleted": {                 /* DD HH:MM */
      "r15ca73b9980-0-1": false,  /* 14 15:32 */
      "r15ca5a8c480-0-1": false,  /* 14 08:12 */
      "r15ca5a6efc0-0-1": true,   /* 14 08:10 */
      "r15ca58e37a0-0-1": false   /* 14 07:43 */
    },
    /* ... */
}
```

(b) After splitting.

Figure 9: Document splitting. We use the same parameters as in Example 6.

r	$t(r)$	$c(r)$	Vis.	$\in$Win.	Vol.	Split
r^1	15 14:00	2	$\bot$	$\top$	0	$\bot$
r^2	15 13:44	2	$\top$	$\top$	1	$\bot$
r^3	15 04:10	1	$\top$	$\top$	2	$\bot$
r^4	14 16:29	1	$\top$	$\top$	3	$\bot$
r^5	14 15:32	1	$\top$	$\top$	4	$\top$
r^6	14 09:23	2	$\top$	$\bot$	4	$\bot$
r^7	14 08:12	1	$\top$	$\bot$	4	$\top$
r^8	14 08:10	1	$\top$	$\bot$	4	$\top$
r^9	14 07:43	1	$\top$	$\bot$	4	$\top$

Figure 10: Intermediate values of computation while splitting document `/index/x/1/a/b` as shown in Figure 9. We use the same parameters as in Example 6.

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.

12 Appendix

12.1 Helper Functions

Figure 11 presents the Java implementation of two helper functions. `isVisible(r)` determines if revision r is visible to the local Oak instance O_i. `isInSlidingWindow(r)` determines if revision r is in the sliding window.

```
    \begin{minted}{java}
/**
 * Checks if r is visible to the local cluster node
 * @param r the revision
 * @returns true iff r is visible to the local cluster node
 */
boolean isVisible(Revision r) {
    return r.getClusterId() == getClusterId()
    || (r.compareRevisionTime(documentNodeStore
    .getHeadRevision()
    .getRevision(getClusterId())) < 0);
}

/**
 * Checks if r is in the sliding window
 * @param r the revision
 * @returns true iff r is in the sliding window
 */
boolean isInSlidingWindow(Revision r) {
    return System.currentTimeMillis() - getSlidingWindowLength() < r.getTimestamp();
}
    \end{minted}
```

Figure 11: Java implementation for helper functions.