Department of Informatics, University of Zürich

BSc Vertiefungsarbeit

Detecting Volatile Index Nodes in a Hierarchical Database System

Rafael Kallis

Matrikelnummer: 14-708-887

Email: rk@rafaelkallis.com

October 3, 2017

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]

Department of Informatics

1 Introduction

Adding and removing data from hierarchical indexes causes them to grow and shrink. They grow and shrink, since adding or removing nodes cause a sequence of nodes to be added or removed in addition. 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 the frequent structural index modifications. Frequent structural index modifications also increase the likelihood of conflicting index updates by concurrent transactions. Conflicting index updates further deteriorate update performance since they cause the transactions to synchronize in order to resolve the conflict.

Wellenzohn et al. [4] propose a workload aware property index (WAPI). The WAPI exploits the workloads' skewness by not removing volatile nodes from the index, thus significantly reducing the number of structural index modifications. By reducing the number of structural index updates, we also decrease the likelihood of conflicting index updates by concurrent transactions.

The goal of this project is to implement a WAPI, as proposed by [4] in Apache Jackrabbit Oak in order to improve the transactional throughput of Jackrabbit Oak.

1.1 System Architecture

Apache Jackrabbit Oak[1] (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 technique [2].

Figure 1.1 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, labeled as *Persistance Tier*. On the other end, applications can make use of Oak as shown in Fig. 1.1 under *Application Tier*. One such application is Adobe's enterprise content management system (CMS), the Adobe Experience Manager.

[1]https://jackrabbit.apache.org/oak/

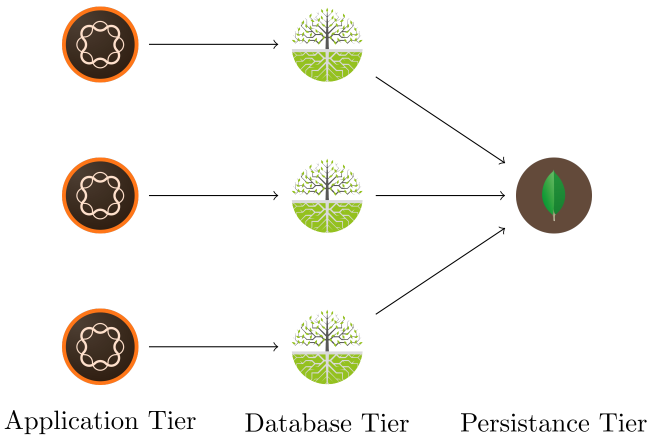


Figure 1.1: Apache Jackrabbit Oak's system architecture.

2 WAPI

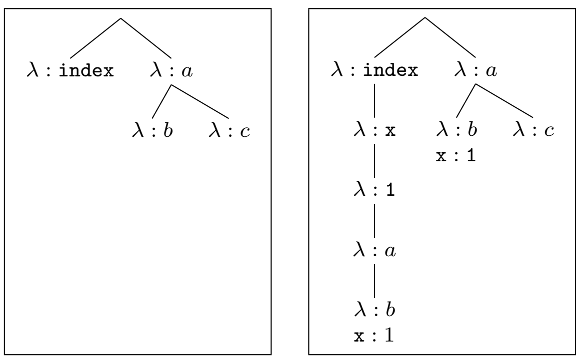
The general idea behind the WAPI is to take into account if an index node is volatile before performing structural index modifications. If a node is considered volatile, we prevent removing it from the index.

In the following chapter, we will see how to add, query and remove nodes from the index.

2.1 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 k. The remaining index levels replicate all nodes from the root node to any content node with k set to v. Node m is added to the WAPI iff m has a property k set to v. Let's consider Fig. 2.1. Given snapshot G^i, transaction T_j adds the property-value pair `x:1` to `/a/b` and commits snapshot G^j. The WAPI is updated as described in algorithm 1. Starting from /index, we descent down to /index/k, /index/k/v. Next, we descent down from /index/k/v with a replica from content node m's absolute path from root. While we descent the WAPI, we create any node n that does not exist. Finally, we set n's property k to v.

$$G^i \xrightarrow{T_j} G^j$$



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.

for $n \in \langle /i/k,\ /i/k/v,\ \ldots,\ /i/k/v/m \rangle$ **do**
  if n *does not exist* **then**
    create node n

$n[k] \longleftarrow v$

Where $/i$ is an abbreviation for /index.

Figure 2.1: Adding a node in a workload aware property index.

2.2 Querying

Oak mostly executes content-and-structure (CAS) queries [1].

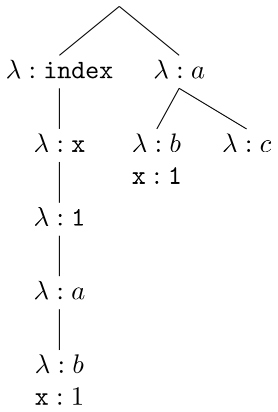
Definition 1 *(CAS-Query): Given node m, property k and value v, a CAS query returns all descendants of m which have k set to the v, i.e*

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

.

Algorithm 2 describes how we answer the 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 descendants n of node m. We return a set consisting of content nodes $*n$ corresponding to every n with property k set to v. If `/index/k/v/m` does not exist, then $desc($`/index/k/v/m`$) = \emptyset$.

Example 1 *Let's consider query $Q(x, 1, /a)$, that is every descendant of* `/a` *with* `x` *set to 1. Assuming we execute the query on the tree depicted in Fig. 2.2, we receive a set including node $/a/b$, i.e $Q(x, 1, /a) = \{\ /a/b\ \}$*



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 2.2: CAS Query example.

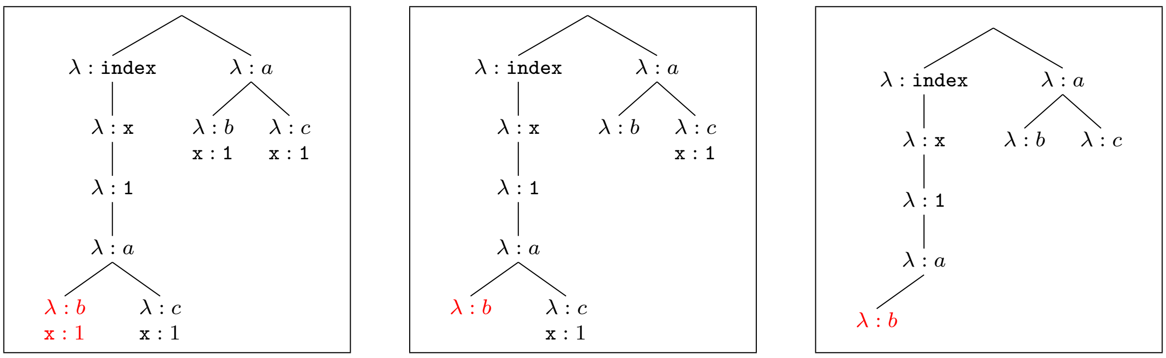
2.3 Deletion

During deletion, we intend to remove a node from the 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 Chapter 3. 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 a node from the workload aware property index. We first propagate down to node m, which we intend to remove. We remove property k from m. If m is a leaf node and does not have property k and is not volatile, we remove it. If m was removed, we repeat the process on its parent node. The process ends if we propagate up to `/index` or have a node with children or a volatile node or a node that has property k.

Example 2 *Figure 2.3 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. Given snapshot* G^j*, transaction* T_k *removes property* `x` *from* `/a/c` *and commits snapshot* G^k*. Since* `/index/x/1/a/b` *is not volatile, it was removed from the WAPI.*

$$G^i \xrightarrow{T_j} G^j \xrightarrow{T_k} G^k$$



Snapshot G^i Snapshot G^j Snapshot G^k

Figure 2.3: 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

---

3 Volatility

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. t_n is the current time. $t(G^b)$ is 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.

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 history H_i over a sliding window of length L.

Definition 2 *(Volatility Count): The number of times node n was added or removed from snapshots contained in a sliding window with length L over history H_i.*

$$\begin{aligned} vol(n) = |\{G^b | G^b \in H_i \wedge t(G^b) \in [t_{n-L+1}, t_n] \wedge \exists G^a [\\ G^a = pre(G^b) \wedge ([n^a \notin N(G^a) \wedge n^b \in N(G^b)] \vee \\ [n^a \in N(G^a) \wedge n^b \notin N(G^b)])]\}| \end{aligned} \tag{3.1}$$

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

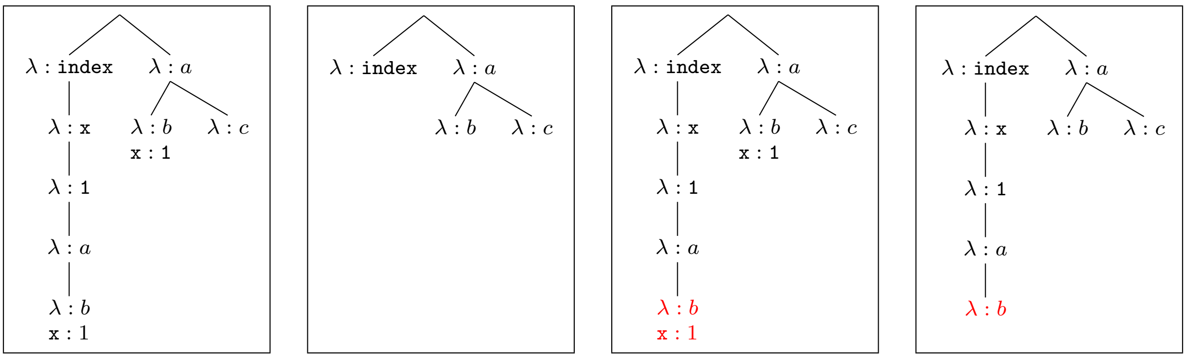
$$isVolatile(n) \iff vol(n) \geq \tau$$

Example 3 *Let's consider the snapshots depicted in Fig. 3.1. Assume $\langle G^i, G^j, G^k, G^l \rangle$ is a partition of history H_h on Oak instance O_h. O_h executes transactions T_j, T_k, T_l. Snapshot G^i was committed during $t(G^i) = t$. Given snapshot G^i, transaction T_j removes property x from `/a/b` and commits snapshot G^j during $t(G^j) = t+1$. Next, transaction T_k adds the property $x = 1$ to `/a/b` given snapshot G^j and commits snapshot G^k during $t(G^k) = t+2$. Finally transaction T_l removes property `x` from `/a/b` given G^k and commits G^l during $t(G^l) = t+3$.*

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

$$G^i \xrightarrow{T_j} G^j \xrightarrow{T_k} G^k \xrightarrow{T_l} G^l$$



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 3.1: Volatility count changes with each snapshot.

4 Implementation

4.1 Checking Node Volatility

In order to classify node n as volatile, we have to compute n's volatility count using the corresponding JSON[1] document on MongoDB. The document contains all revisions of n throughout history H_i of an Oak instance O_i. Figure 4.1 depicts such a document. We omitted non relevant properties. Property `"_deleted"` contains key value pairs which encode when the node was added or removed from snapshots, called revisions. Each value's key is composed with a timestamp, a counter that is used to differentiate between value changes during the same instance of time and the identifier of the oak instance committing the change.

Example 4 *Let's consider* `r15cac0dbb00-0-2` *in Fig. 4.1.* `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 during which the change was committed. The* `0` *following the timestamp, is a counter which is used for tie-breaking between transactions committed during the same instance of time. The* `2` *following the counter, tells that the change was committed by the Oak instance with an id of* `2`*.*

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

Figure 4.1: JSON document of an index node.

Having seen what a node document looks like, we can now describe how we classify a node as volatile. Figure 4.2 shows the native implementation of $isVolatile(n)$ in Java.

Example 5 *As shown in Fig. 4.2, $isVolatile(n)$ is given a node's corresponding document. We iterate through the revisions of property* `"_deleted"` *in most-recent first fashion. If a revision is outside the sliding window we stop iterating because remaining*

[1]http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

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.

```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();
}
```

Figure 4.2: Java implementation for detecting volatile index nodes.

4.2 Document Splitting

```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();
}
```

Figure 4.3: Java implementation for helper functions.

```java
/**
* Collects all local property changes committed by the current
* cluster node.
* @param committedLocally local changes committed by the current cluster node.
* @param changes all revisions of local changes (committed and uncommitted).
*/
void collectLocalChanges(
        Map<String, NavigableMap<Revision, String>> committedLocally,
        Set<Revision> changes) {

    int vol = 0;

    // for each public property or "_deleted"
    for (String property : filter(doc.keySet(), PROPERTY_OR_DELETED)) {
        NavigableMap<Revision, String> splitMap =
                new TreeMap<Revision, String>(StableRevisionComparator.INSTANCE);
        committedLocally.put(property, splitMap);

        // local property revisions
        Map<Revision, String> valueMap = doc.getLocalMap(property);

        // for each Revision & Value tuple in
        for (Map.Entry<Revision, String> entry : valueMap.entrySet()) {
            Revision r = entry.getKey();

            if (property.equals("_deleted")) {
                if (!isVisible(r)){
                    continue;
                }
                if (isInSlidingWindow(r) && vol++ < getVolatilityThreshold()){
                    continue;
                }
            }
            if (r.getClusterId() != context.getClusterId()) {
                continue;
            }

            // move to split document
            changes.add(r);
            if (isCommitted(context.getCommitValue(r, doc))) {
                splitMap.put(r, entry.getValue());
            } else if (isGarbage(r)) {
                addGarbage(r, property);
            }
        }
    }
}
```

Figure 4.4: Java implementation for splitting the node document.

Algorithm 4: SplitDocumentWAPI

```
Data: Document d.
vol ⟵ 0
foreach versioned property k ∈ d do
    foreach revision r ∈ d[k] do
        if k = _deleted then
            if c(r) ≠ O_i ∧ t_last_sync < t(r) then
                continue
            if t(r) ∈ [t_{n−L+1}, t_n] then
                vol ⟵ vol + 1
                if vol ≤ τ then
                    continue
        if c(r) ≠ O_i then
            continue
        moveToSplitDocument(d, k, r)
```

Where τ is the volatility threshold, L the sliding window length, O_i the local cluster node, t_n the current time, $c(r)$ the cluster node which committed revision r and $t(r)$ the point of time revision r was committed.

```
{
  "_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 */
  },
  /* ... */
}
```

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

Intermediate values of computation while splitting document a.
Assume $\tau = 3$, $O_i = 1$, $t_{\texttt{last\_sync}} = 2017.06.15\ 13{:}59$, $L = 24$ hours, $t_n = 2017.06.15\ 14{:}01$.

- "$t(r)$" is the point of time revision r was committed. Only the day, hours and minutes are showed for brevity.
- "$c(r)$" is the cluster node which 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 during that step of the iteration.
- "Split" is true iff the revision is moved to the split document.

Bibliography

[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.