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 transaction to abort and restart in a later point in time.

Wellenzohn et al. [**?**] propose a workload aware property index (WAPI). The WAPI exploits the workloads' skewness by not removing volatile nodes from the index. By not removing these volatile nodes, we increase performance since we:

- Reduce the number of structural index modifications, which are expensive because they implicitly cause a sequence of nodes to be added or removed from the index.
- By reducing index modifications, we also decrease the number of index conflicts, which limit throughput since they cause a transaction to abort and restart.

The goal of this project is to implement a WAPI, as proposed by (ref kevin paper) in Apache Jackrabbit Oak in order to improve the transactional throughput of Jackrabbit Oak.

1.1 System Architecture

Apache Jackrabbit Oak (reference) (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) (reference), a commonly used optimistic technique (reference Principals of Distributed databases).

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.

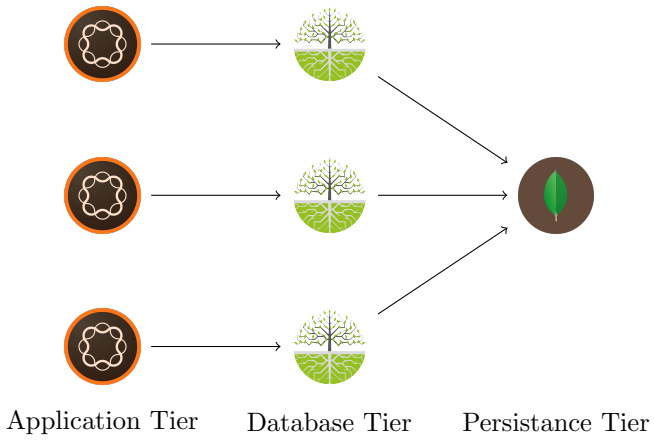


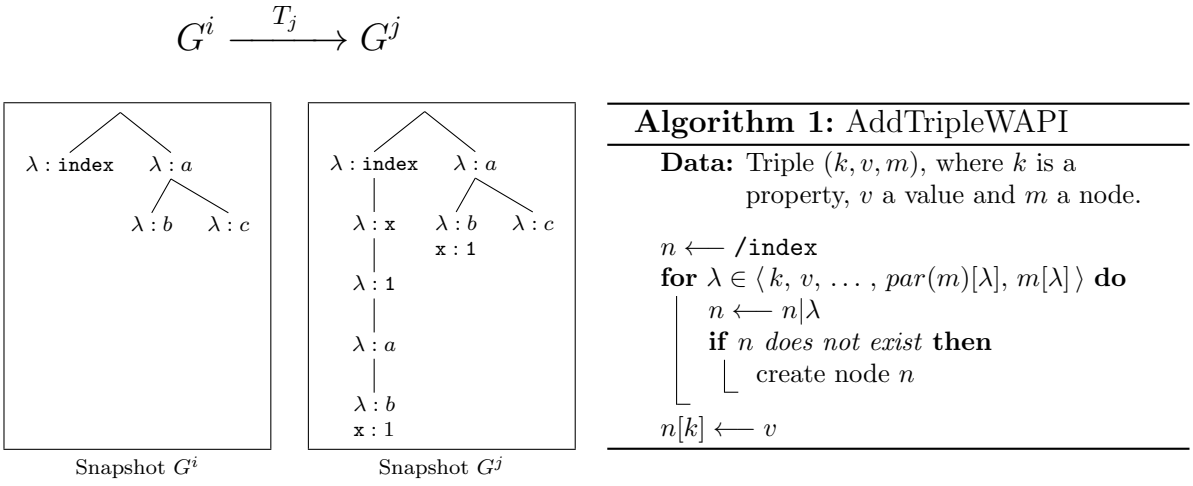
Figure 1.1: Apache Jackrabbit Oak's system architecture.

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.



Algorithm 1: AddTripleWAPI

Data: Triple (k, v, m), where k is a property, v a value and m a node.

$n \longleftarrow$ `/index`
for $\lambda \in \langle\, k, v, \ldots, par(m)[\lambda], m[\lambda] \,\rangle$ **do**
  $n \longleftarrow n|\lambda$
  if n *does not exist* **then**
    create node n
$n[k] \longleftarrow v$

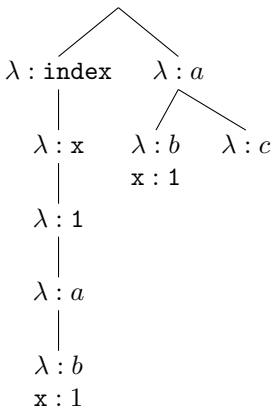
Figure 2.1: Adding a node in a workload aware property index.

2.2 Querying

Oak mostly executes content-and-structure (CAS) queries **(ref)**. An example of such a query can be found in Fig. 2.2.

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

Algorithm 2 describes how we execute the query. Given a property k, a value v and node m, we start descending from the index root node `/index`. If k is a child of `/index`, we descent down to k, else we return the empty set. If v is a child of k, we descent down to v, else we return the empty set. Next, we descent down a path which is an exact replica of m's content node path. While descending, if we encounter a node which does not exist in the index, we return the empty set. Otherwise, we iterate through all descendants of m and filter each node which does not have property k set to v. We then return the content node of every descendant of m with k set to v.



Algorithm 2: QueryWAPI

Data: Triple (k, v, m), where k is a property, v a value and m a node.
Result: A set of paths satisfying CAS query $Q_{k,v,m}$

$n \longleftarrow$ `/index`
for $\lambda \in \langle\, k,\, v,\, \ldots,\, par(m)[\lambda],\, m[\lambda] \,\rangle$ **do**
  $n \longleftarrow n|\lambda$
  if n *does not exist* **then**
    return $\emptyset$
$r \longleftarrow \emptyset$
for $d \in desc(n)$ **do**
  if $d[k] = v$ **then**
    $r \longleftarrow r \cup \{trunc(d)\}$
return r

Where $par(m)$ is the parent of node m, $desc(n)$ is the set of descendants of node n, $d[k]$ is property k of node d and $trunc(d)$ returns the path of the content node of d e.g $trunc($`/index/x/1/a/b`$) =$ `/a/b`.

Having the following query, that is every descendant of `/a` with `x` set to 1, we receive a set including node `/a/b`.

$$Q_{\texttt{x},1,\texttt{/a}} = \{\, \texttt{/a/b} \,\}$$

Figure 2.2: CAS Query example.

2.3 Deletion

A workload aware property index differentiates itself mostly when nodes are removed. It detects which nodes are volatile and avoids removing them. The process of classifying a node as volatile, will be explained in more details in Chapter 3. 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 or have a node with children or a volatile node.

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` from `/a/b` and commits snapshot G^j. Since `/index/x/1/a/b` is volatile, it is 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 is removed from the WAPI.

Algorithm 3 describes the process of removing a node from the workload aware property index.

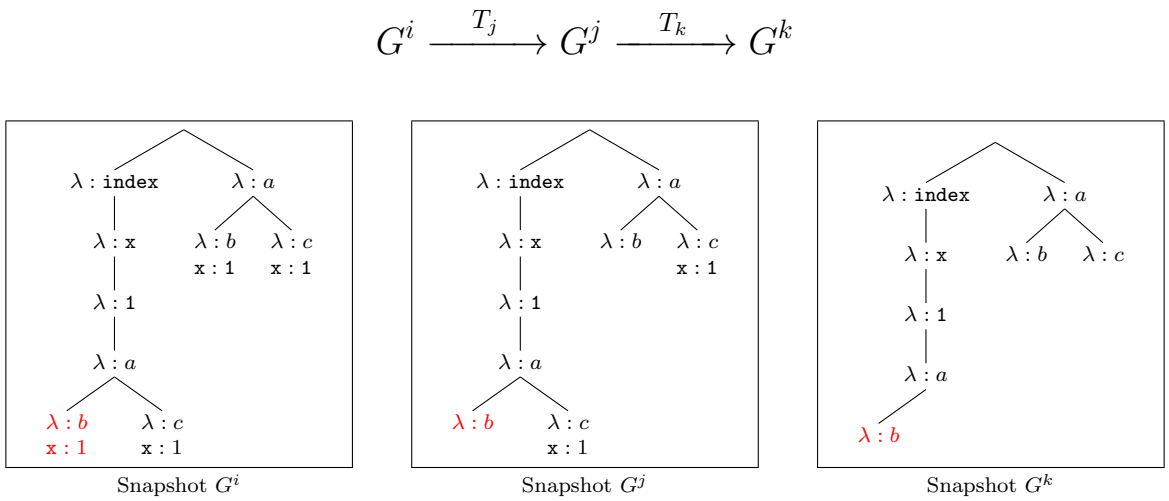


Figure 2.3: Removing a node from the WAPI. Assume `/index/x/1/a/b` (colored red) is volatile in all three snapshots G^i, G^j, G^k.

Algorithm 3: RemoveTripleWAPI

Data: Triple (k, v, m), where k is a property, v a value and m a node.
begin
    $n \longleftarrow$ `/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

Definition 2

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

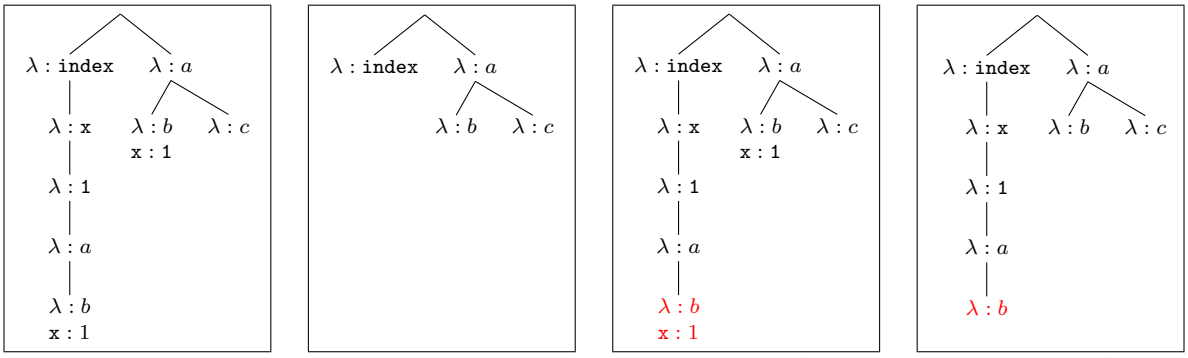
(Volatility Count): The number of times node n was added or removed from snapshots contained in a sliding window with length L over history H_i. t_n is the current time. $t(G^b)$ is the point in time snapshot G^b was committed, $N(G^a)$ is the set of nodes which are members of snapshot G^a. $pre(G^b)$ is the predecessor of snapshot G^b.

Definition 3

$$isVolatile(n) \iff vol(n) \geq \tau \tag{3.2}$$

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

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



$\langle G^i, G^j, G^k, G^l \rangle$ is a partition of history H_h.
Given G^i, transaction T_j removes property `x` from `/a/b` and commits G^j.
Given G^j, transaction T_k adds the property-value pair `x:1` to `/a/b` and commits G^k.
Given G^k, transaction T_l removes property `x` from `/a/b` and commits G^l.
Assume $t(G^i) = t, \quad t(G^j) = t+1, \quad t(G^k) = t+2, \quad t(G^l) = t+3$. 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: $vol(n) = 0 \geq \tau \iff isVolatile(n) = \bot$
- at time $t_n = t+1$ we have that: $vol(n) = 1 \geq \tau \iff isVolatile(n) = \bot$
- at time $t_n = t+2$ we have that: $vol(n) = 2 \geq \tau \iff isVolatile(n) = \top$
- at time $t_n = t+3$ we have that: $vol(n) = 3 \geq \tau \iff isVolatile(n) = \top$

Figure 3.1: Volatility count changes with each snapshot.

4 IMPLEMENTATION

4.1 Checking Node Volatility

```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()) {
            return true;
        }
    }
    return false;
}
```

Figure 4.1: 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 4.2: 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 4.3: 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	$\bot$	$\top$	0	$\bot$
15 13:44	2	$\top$	$\top$	1	$\bot$
15 04:10	1	$\top$	$\top$	2	$\bot$
14 16:29	1	$\top$	$\top$	3	$\bot$
14 15:32	1	$\top$	$\top$	4	$\top$
14 09:23	2	$\top$	$\bot$	4	$\bot$
14 08:12	1	$\top$	$\bot$	4	$\top$
14 08:10	1	$\top$	$\bot$	4	$\top$
14 07:43	1	$\top$	$\bot$	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.