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

Oktober 3, 2017

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]
Department of Informatics

1 INTRODUCTION

Hierarchical indexes with skewed and update-heavy workloads grow and shrink often. Since index modifications propagate up and down, adding and removing nodes cause a sequence of nodes to be added or removed in addition, thus deteriorating the index update performance. Index update performance also suffers from conflicting index updates. Highly skewed workloads create hotspots where nodes are frequently added or removed, increasing conflicting index updates, as shown in (ref kevin paper).

A content management system's workloads share common properties. They are a) skewed and the same data item is repeatably added or removed from the index and b) update-heavy.

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

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

An property index which can detect which nodes are volatile, is a **workload aware** property index (WAPI).

Apache Jackrabbit Oak (reference) (Oak) a hierarchical database system which makes use of a hierarchical index. Oak has two design goals in mind. It needs to be able to operate in a distributed environment and guarantee write throughput. Multiple Oak instances can work concurrently by making use of Multiversion Concurrency Control (MVCC) (reference), a commonly used optimistic technique (reference Principals of Distributed databases). Whilst Oak is responsible for handling the database logic, it stores the actual data on MongoDB. Although Oak is an open-source project, it is being actively maintained by Adobe (reference). Adobe's content management system (CMS) makes use of Oak in one of their products, specifically Adobe Experience Manager.

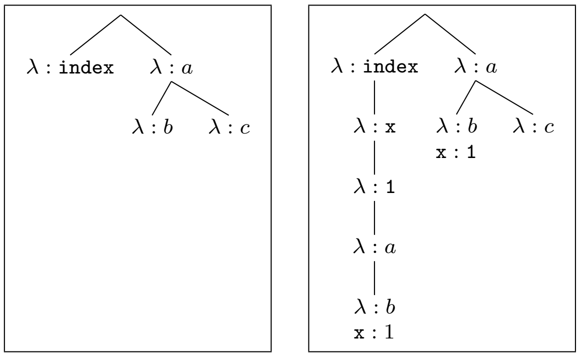
The goal of this project is to implement a WAPI, as proposed by (ref kevin paper) in Apache Jackrabbit Oak in order to improve throughput.

2 WAPI

2.1 Adding nodes to the WAPI

Node n is added to the WAPI iff n has a property k set to v. Let's consider fig. 2.1. Given snapshot G^i, transaction T_j adds property `x` with value 1 to `/a/b` and commits snapshot G^j. The WAPI is updated as described in alg. 1.

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



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 node.

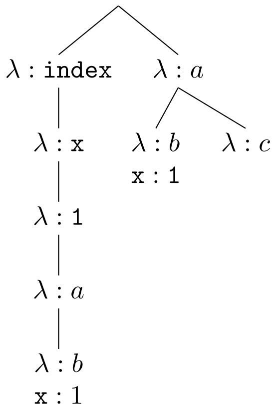
begin
- $n \longleftarrow$ `/index`
- **for** $\lambda \in \langle\, k,\ v,\ \ldots,\ par(m)[\lambda],\ m[\lambda]\,\rangle$ **do**
  - $n \longleftarrow n|/\lambda$
  - **if** $\nexists n$ **then**
    - create node n
- $n[k] \longleftarrow v$

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

2.2 Querying

Oak mostly executes content-and-structure (CAS) queries **(ref)**. Given node m, property k and value v, a CAS query returns all descendants of m which have k set to the v. An example of such a query can be found in fig. 6.1.

Definition 1 *(CAS-Query):* $Q_{k,v,m} = \{\, n \mid n[k] = v \wedge n \in desc(m) \,\}$



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

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

Algorithm 2: QueryPropertyIndex

Data: Triple (k, v, m), where k is a property, v a value and m a node.
Result: $\{\, n \mid n[k] = v \wedge n \in desc(m) \,\}$
begin
- $n \longleftarrow$ `/index`
- **for** $\lambda \in \langle\, k,\ v,\ \ldots\ ,\ par(m)[\lambda],\ m[\lambda] \,\rangle$ **do**
  - $n \longleftarrow n|/\lambda$
  - **if** $\nexists n$ **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

Figure 2.2: CAS Query example.

2.3 Removing nodes from the WAPI

A workload aware property index differentiates itself from a property index mostly during node removal. 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.

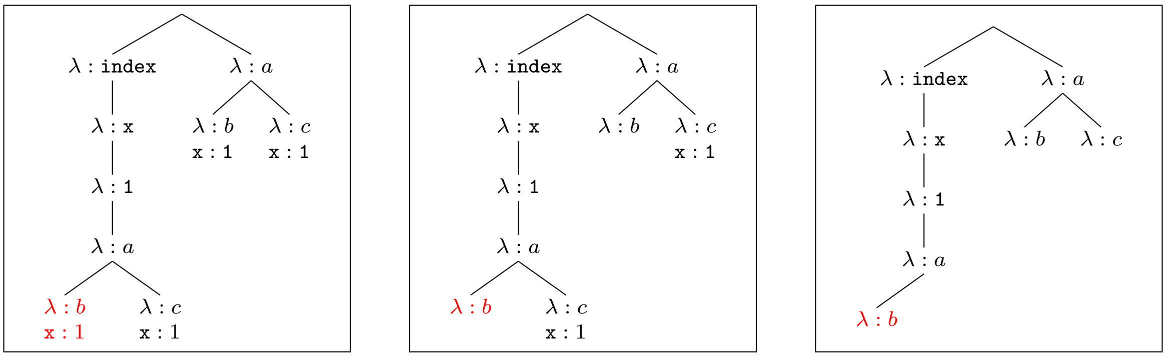
Figure 2.3 depicts the following scenario. Assume `/index/x/1/a/b` (colored red) is volatile in all three snapshots G^i, G^j, G^k. Given snapshot G^i, transaction T_j removes property `x` from `/a/b` and commits snapshot G^j. Since `/index/x/1/a/b` is volatile, it is not removed from the WAPI. Given 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.

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** $n[\lambda] \neq \textbf{\textit{index}} \wedge chd(n) = \emptyset \wedge n[k] \neq v \wedge \neg\ isVolatile(n)$ **do**
  - $u \longleftarrow n$
  - $n \longleftarrow par(n)$
  - remove node u

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



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.

3 VOLATILITY

Definition 2

$$isVolatile(n^j) \iff vol(n^j) \geq \tau \tag{3.1}$$

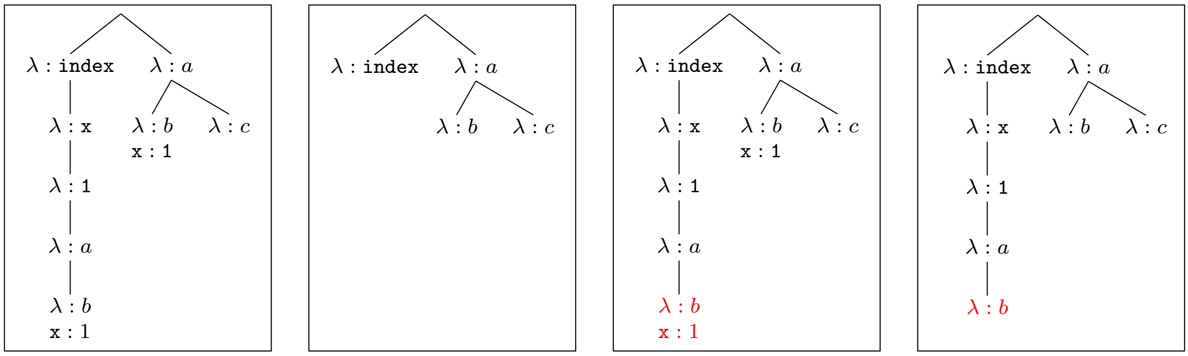
(Volatile Node): Let n^j denote a node. Node n^j is a member of the node-set of snapshot G^j, i.e $n^j \in N(G^j)$. Snapshot G^j is a member of history H_i, i.e $G^j \in H_i$. Node n^j is volatile iff n^j's volatility count (definition 3) is greater or equal than the volatility threshold τ.

Definition 3

$$\begin{aligned} vol(n^j) = |\{G^b | G^b \in H_i \wedge t(G^b) \in [t_{n-L+1}, t_n] \wedge \exists G^a [\\ G^a = pre(G^b) \wedge ([n^a \notin N(G^a) \wedge n^b \in N(G^b)] \vee \\ [n^a \in N(G^a) \wedge n^b \notin N(G^b)])]\}| \end{aligned} \tag{3.2}$$

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

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



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

$\langle G^i, G^j, G^k, G^l \rangle$ is a partition of history H_h.
Assume $\tau = 2$ (volatility threshold) and $L = 4$ (sliding window length).
Given G^i, transaction T_j removes property `x` from `/a/b` and commits G^j. $G^i = pre(G^j)$.
Given G^j, transaction T_k adds the property-value pair `x:1` to `/a/b` and commits G^k. $G^j = pre(G^k)$.
Given G^k, transaction T_l removes property `x` from `/a/b` and commits G^l. $G^k = pre(G^l)$.
$t(G^i) = t, \quad t(G^j) = t + 1, \quad t(G^k) = t + 2, \quad t(G^l) = t + 3, \quad t(pre(G^i)) = t - 4.$
n^i, n^j, n^k, n^l represent versions of node $n =$ `/index/x/1/a/b` in snapshots G^i, G^j, G^k, G^l respectively.
Then:

- $vol(n^i) = 0 \iff isVolatile(n^i) = F$
- $vol(n^j) = 1 \iff isVolatile(n^j) = F$
- $vol(n^k) = 2 \iff isVolatile(n^k) = T$
- $vol(n^l) = 3 \iff isVolatile(n^l) = T$

Figure 3.1: Volatility count changes with each snapshot.

4 IMPLEMENTATION

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

    int count = 0;

    for (Revision r : nodeDocument.getLocalDeleted().keySet()) {
        if (!isInSlidingWindow(r)){
            break;
        }
        if (!isVisible(r)){
            continue;
        }
        if (++count >= getVolatilityThreshold()) {
            return true;
        }
    }
    return false;
}
```

Figure 4.1: Java implementation for detecting volatile index nodes.

```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) {

    // 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);

        int count = 0;

        // collect committed changes of this cluster node
        for (Map.Entry<Revision, String> entry : valueMap.entrySet()) {
            Revision rev = entry.getKey();

            if (property.equals("_deleted")) {
                // not visible
                if (!isVisible(rev)){
                    continue;
                }

                // not tau more recent revisions and is in sliding window
                if (++count <= getVolatilityThreshold() && isInSlidingWindow(rev)){
                    continue;
                }

            }

            if (rev.getClusterId() == context.getClusterId()) {
                changes.add(rev);
                if (isCommitted(context.getCommitValue(rev, doc))) {
                    splitMap.put(rev, entry.getValue());
                } else if (isGarbage(rev)) {
                    addGarbage(rev, property);
                }
            }
        }
    }
}
```

Figure 4.2: Java implementation for splitting the node document.

```java
boolean isVisible(Revision r) {
    return r.getClusterId() == getClusterId()
            || (r.compareRevisionTime(documentNodeStore
                    .getHeadRevision()
                    .getRevision(getClusterId())) < 0);
}

boolean isInSlidingWindow(Revision r){
    return System.currentTimeMillis() - getSlidingWindowLength() < r.getTimestamp();
}
```

Figure 4.3: Java implementations for helper functions.

5 Introduction

Apache Jackrabbit Oak (reference) (Oak) a tree-structured database system, with two design goals in mind. It needs to be able to **a)** *operate in a distributed environment* and **b)** *guarantee write throughput*. Multiple Oak instances can work concurrently by making use of Multiversion Concurrency Control (MVCC) (reference), a commonly used optimistic technique (reference Principals of Distributed databases). Whilst Oak is responsible for handling the database logic, it stores the actual data on MongoDB. Although Oak is an Open-source project, it is being actively maintained by Adobe (reference). Adobe's Content Management System (CMS) makes use of Oak in one of their products, specifically Adobe Experience Manager (AEM).

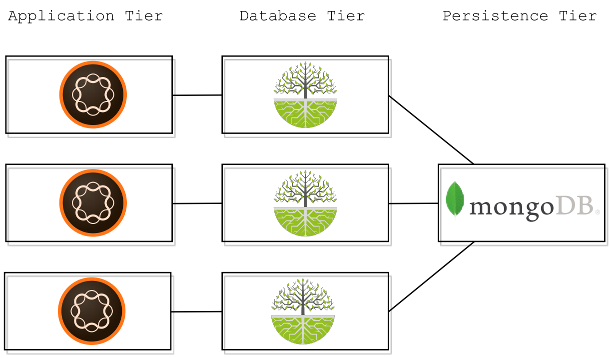


Figure 5.1: Apache Jackrabbit Oak's system architecture. The application, Adobe Experience Manage in this figure, connects to Oak.

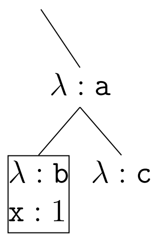
In the following pages, we will take a closer look at Oak. Specifically, we will see how Oak handles querying, writing and concurrency control. After that we will describe an instance of a problem Oak is having and we will briefly introduce a solution which results in higher throughput under certain circumstances. Lastly, we will modify Oak's reference implementation in order to satisfy our solution.

6 Problem definition

6.1 Property Index

Oak mostly executes content-and-structure (CAS) queries **(ref)**. Given node m, property k and value v, a CAS query returns all descendants of m which have k set to the v. An example of such a query can be found in fig. 6.1.

Definition 4 *(CAS-Query):* $Q_{k,v,m} = \{\, n \mid n[k] = v \land n \in desc(m) \,\}$



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

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

Figure 6.1: CAS Query example.

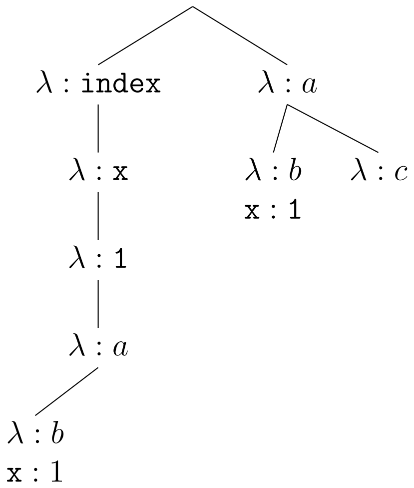
In order to answer such queries efficiently, Oak implements a Property Index (PI) (ref kevin's paper). A property index is a hierarchical index. Figure 6.2 depicts a PI and shows how a CAS query can be answered by using it. In addition to that you can find how a node is added or removed from the index.

6.2 Workload implications

A Content management system's workloads share common properties. They are **a)** *skewed and the same data item is repeatably added or removed from the index* and **b)** *update-heavy.*

Hierarchical indexes with skewed and update-heavy workloads grow and shrink often. Since index modifications propagate up and down, adding and removing nodes cause a sequence of nodes to be added or removed in addition, thus deteriorating the index update performance, as shown in fig. 7.1. Index update performance also suffers from conflicting index updates. Highly skewed workloads create hotspots where nodes are

(a) Tree with PI.



(b) Querying a tree with a PI.

Algorithm 4: QueryPropertyIndex

Data: Triple (k, v, m), where k is a property, v a value and m a node.
Result: $\{\, n \mid n[k] = v \wedge n \in desc(m) \,\}$
begin
  $n \longleftarrow$ `/index`
  for $\lambda \in \langle\, k,\ v,\ \ldots\ ,\ par(m)[\lambda],\ m[\lambda]\,\rangle$ **do**
    $n \longleftarrow n|/\lambda$
    if $\nexists n$ **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

(c) Adding a node to the PI.

Algorithm 5: AddTriple

Data: Triple (k, v, m), where k is a property, v a value and m a node.
begin
  $n \longleftarrow$ `/index`
  for $\lambda \in \langle\, k,\ v,\ \ldots\ ,\ par(m)[\lambda],\ m[\lambda]\,\rangle$ **do**
    $n \longleftarrow n|/\lambda$
    if $\nexists n$ **then**
      create node n
  $n[k] \longleftarrow v$

(d) Removing a node from the PI.

Algorithm 6: RemoveTriple

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 $n[\lambda] \neq \mathit{index} \wedge chd(n) = \emptyset \wedge n[k] \neq v$
  do
    $u \longleftarrow n$
    $n \longleftarrow par(n)$
    remove node u

Where λ is a node's label, i.e., node `/a/c` has `c` as a label, $|$ is the concatenation operator, $par(m)$ returns the parent node of m, $chd(n)$ returns the set of children nodes of n, $desc(n)$ returns the set of descendant nodes of n, $d[k]$ is the value of property k of node d and $trunc($`/k/v/a/b`$) =$ `/a/b` truncates the property name and value from the node.

Figure 6.2: Answering CAS queries efficiently using a Property Index.

frequently added or removed, increasing conflicting index updates, as shown in (ref kevin paper).

Since Oak's PI is hierarchical, its performance deteriorates if used with CMS' workloads. You can find application scenarios which demonstrate how the PI's performance suffers in chapter 7.

6.3 Proposal

In order to improve performance, the following is being proposed. We exploit the workloads' skewness by not removing frequently accessed nodes from the PI. By not removing these **volatile** nodes, we:

- Trade query performance for update performance.
- Decrease the number of index conflicts thus increasing update performance.

A property index which can detect which nodes are volatile, is a **workload aware** property index (WAPI). A WAPI removes nodes as shown in alg. 3. Besides checking if a node is volatile, alg. 3 shares no other differences in comparison with alg. 6.

7 Application Scenarios

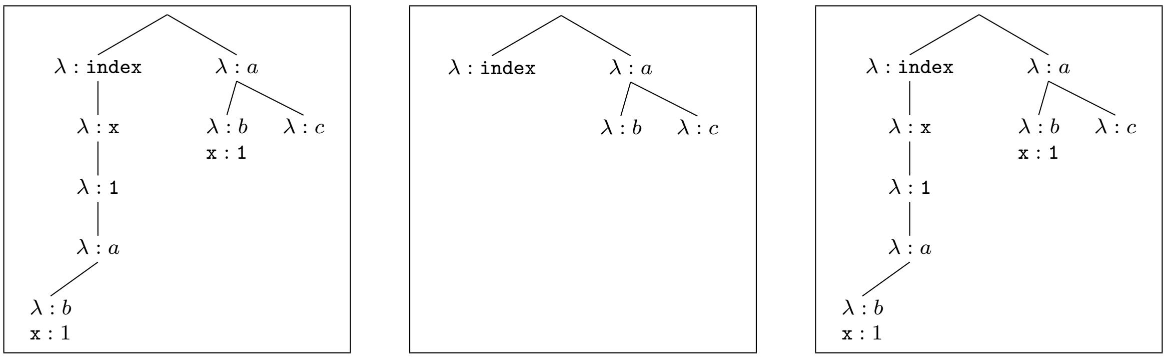
7.1 Tree growing and shrinking.

Let's consider fig. 7.1. The tree in fig. 7.1 has a property index which is not workload aware.

Transaction T_1 removes property `x` from `/a/b`. `x` is removed from `/index/x/1/a/b`. In addition to that, all ancestor nodes, $anc($`/index/x/1/a/b`$) = \{$`/index/x`, `/index/x/1`, `/index/x/1/a`$\}$ G^0, are removed as well. (c.f alg. 6)

Transaction T_2 adds property `x` to `/a/b`. We first have to add all ancestors in order to add `/index/x/1/a/b` to the property index (c.f alg. 5).

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

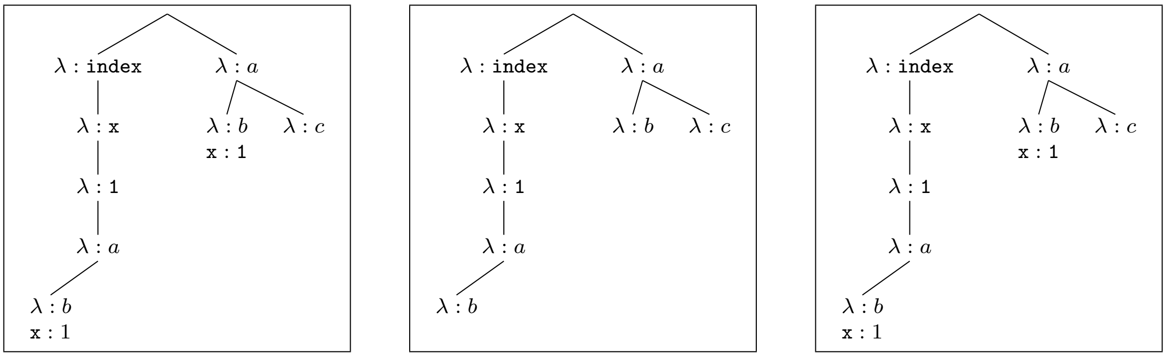


Tree G^0 Tree G^1 Tree G^2

Figure 7.1: Updating a node in a property index.

Figure 7.2 depicts the same transactions but the property index is workload aware. Assume `/index/x/1/a/b` is volatile in all . Transaction T_1 removes property `x` from `/a/b`. `x` is removed from `/index/x/1/a/b` but since it is volatile, the node is not removed from the WAPI (c.f alg. 3). This avoids removal of its ancestors. Transaction T_2 adds property `x` to `/a/b`. Since `/index/x/1/a/b` is volatile, it already exists in the WAPI along with its ancestors. This avoided adding its ancestors.

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



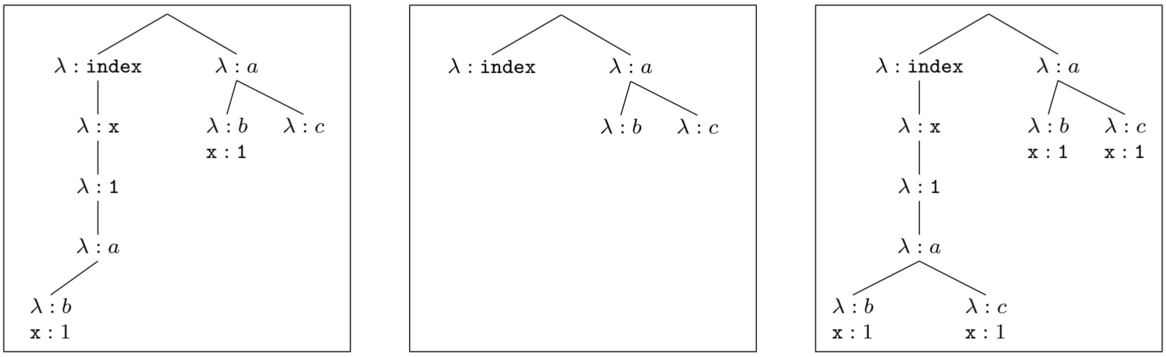
Tree G^0 Tree G^1 Tree G^2

Figure 7.2: Updating a volatile node in a workload aware property index. Assume `/index/x/1/a/b` is volatile.

7.2 Index conflicts

Although two concurrent transactions may have conflicts, some conflicts are **avoidable**. Transactions T_1 and T_2 have an avoidable conflict iff T_1 and T_2 have a conflict and all conflicting nodes are nodes in the property index. (kevin's paper) describes extensively when two transactions have a conflict. Figure 7.3 depicts an avoidable index conflict in a non workload aware PI. Assume T_1 committed before T_2. Since Oak uses a first committer wins strategy, T_2 is forced to abort and restart

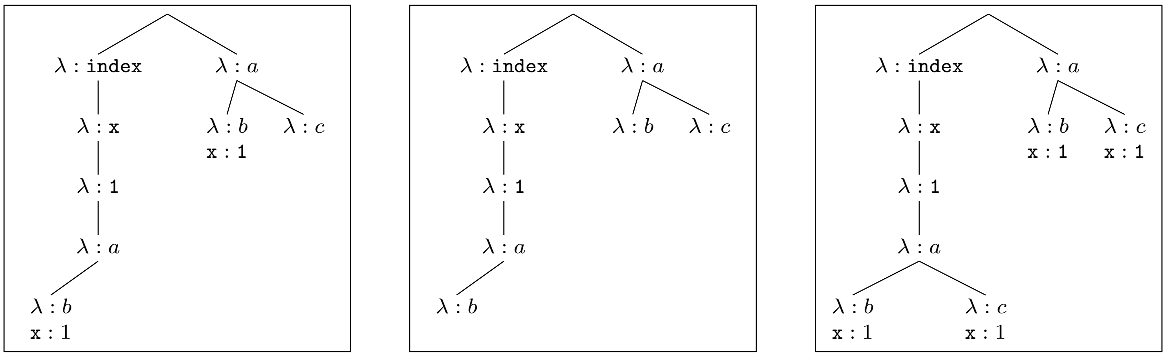
$$G^0 \xrightarrow{T_1} G^1$$
$$G^0 \xrightarrow{T_2} G^2$$



Tree G^0 Tree G^1 Tree G^2

Figure 7.3: Index conflict in the property index.

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



Tree G^0 Tree G^1 Tree G^2

Figure 7.4: Avoided index conflict in the workload aware property index. Assume `/index/x/1/a/b` is volatile.

8 Oak's Mechanics

8.1 Persistence Tier

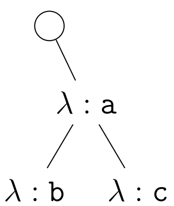
Before we get into the inner workings of Oak, we need to understand how Oak chose to persist data. As mentioned earlier, Oak's data is tree-structured and is stored in a MongoDB instance. Each tree node is persisted in the shape of a JSON document. Each tree node contains properties, which are represented by key-value pairs inside the JSON document. Each node is identifiable by its tree-depth concatenated with its absolute path from the root node, denoted as `"_id"`, which is also a unique identifier. Figure 8.1 illustrates a tree alongside its persisted state.

In order to support MVCC, Oak keeps a history of values each property had in the past such that we are able to tell when each value was persisted and which Oak instance committed the change. Properties which have such a history are called versioned properties. In fig. 8.2, we take a look at node `/a/c` and see how property `x`'s value changes over time.

Oak also keeps a versioned property in order to determine if a node was added or deleted. The property is called `"_deleted"` and has a boolean value.

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. Let's consider `r15cac0dbb00-0-2`. `r` is a standard prefix and can be neglected. The `15cac0dbb00` following `r`, is an 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, tells that the change was the 1st change of the specific property during that instance of time. The `2` following the counter, tells that the change was committed by the Oak instance with an id of `2`.

It is worth mentioning that certain properties exist which are not meant for public usage. Such values are prefixed with an underscore (_).




```
[
    { "_id": "0:/",    /* ... */ },
    { "_id": "1:/a",   /* ... */ },
    { "_id": "2:/a/b", /* ... */ },
    { "_id": "2:/a/c", /* ... */ }
]
```

Figure 8.1: A tree and its JSON representation.

```
[
    {
        "_id": "2:/a/c",
        "x": {
            "r15ca9f191c0-0-1": "1",
            "r15cabff1500-0-2": "2",
            "r15cac0dbb00-0-2": "3"
        },
        "_deleted": {
            "r15ca9eb2920": false,      /* added */
            "r15ca9ec1380": true,       /* removed */
            "r15ca9ecfde0-0-1": false   /* added */
        },
        /* ... */
    },
    /* ... */
]
```

Figure 8.2: A node's property in detail.

8.2 Periodic Synchronization

Even though every cluster node O_i accesses the same MongoDB instance, each O_i sees a different slice of MongoDB. Every O_i performs periodic synchronization with the MongoDB instance. During the synchronization, the cluster node:

1. Makes locally committed changes visible to other cluster nodes.
2. Gains access to changes other cluster nodes have made visible.

The synchronization is executed on a background process and is independent from the transactions. In the following chapter, we will see how synchronization works in more detail. Specifically, making locally committed changes visible (1) is covered in

cref{transaction-write}

and gaining access to new changes (2) is covered in section 8.3.1.

8.3 Transactions

In this chapter we will see how Oak handles transactions. Since Oak has to operate in a distributed environment, the underlying data structure is immutable in order to prevent side-effect and keep Oak thread-safe (reference oak). Specifically, Oak implements a Persistent Tree (reference cormen). A transaction is composed as follows:

$$Read \longrightarrow Validate \longrightarrow Write$$

8.3.1 Read

The read phase extends from the start of the transaction until just before it commits. During this phase, a transaction is able to read and write changes on a (local) copy of the tree. As you might remember, every property value is accompanied by a timestamp. A transaction only sees the most recent value of a property up to the time the transaction started. Let O_i denote a cluster node. Let T_i denote a recently started transaction on O_i. Let $t_{sync}(T_i)$ denote the most recent instance of time O_i synchronized just before T_i started. Let n^i denote a version of node (or property value) n. Let $t_{sync}(n^i)$ denote the instance of time the node (or property value) was made visible.

Definition 5 *(Visibility): The version n^i of node (or property value) n is visible to T_i iff one of the following mutuaally exclusive conditions is true:*

1. *a) n^i was committed by transaction T_{i-1} on the same cluster node **and***

   b) there does not exist any more recent version n^j which was committed by a O_i synchronized, i.e.,

2. *a) n^i was made visible before O_i synchronized, and*

   b) there does not exist any more recent version n^j which was made visible before O_i synchronized, i.e.,

$$ts(n^i) \leq ts(T_i) \wedge \nexists n^j (ts(n^i) < ts(n^j) \leq ts(T_i))$$

This ensures that concurrent transactions can not mutate T_i's read values. Figure 8.3 shows how Oak reads values from the tree in a more illustrative manner.

```
[
    { "_id": "0:/", /* ... */ },
    { "_id": "2:/a/b", "x": {
            "r15e830cae80-0-1": 0, /* 01:00 */
            "r15e830d98e0-0-1": 1, /* 01:01 */
            "r15e830f6da0-0-2": 2, /* 01:03 */
        },
        /* ... */
    },
    /* ... */
]
```

Figure 8.3: Assume cluster node O_1 synchronized at `01:02`. Assume O_1 starts transaction T_1 just after synchronization finished. This implies that `/a/b`'s property `x` has value 1 during transaction T_1, i.e., $t_{sync}(T_1) = \texttt{01:02} \wedge n_1 = \texttt{/a/b} \implies n_1[\texttt{x}] = 1$.

Assume transaction T_i wants to remove property `x` from node `/a/b`. We define any change to a property, a property level change. A property level change occurs if a property was added, removed or its value changed. `wp(/a/b, x)` denotes such a property level change. In the particular example, property `x` of node `/a/b` had a change.

Analogously, any change to a node is defined as node level change. A node level change occurs if a node was added or deleted. `wn(/a/b)` denotes such a node level change. In the particular example, node `/a/b` had a change.

Note that if a PI exists, a property level change can cause an update in the PI, such that nodes are added or removed from the PI. Node level changes in the PI caused by property level changes are also called implicit node level changes.

During T_i, any node- or property- level change is added to the write set (reference). The write set of T_i is defined as:

$$\begin{aligned}
\Delta T_i = \{ & \\
& wp(\texttt{/a/b}, \texttt{x}), && \triangleright \texttt{ remove x from /a/b} \\
& wn(\texttt{/index/x/1/a/b}), && \triangleright \texttt{ remove node /index/x/1/a/b} \\
& wn(\texttt{/index/x/1/a}), && \ldots \\
& wn(\texttt{/index/x/1}), && \\
& wn(\texttt{/index/x} && \\
& \} &&
\end{aligned} \tag{8.1}$$

8.3.2 Validate

During the validation phase, Oak determines if there is any interference with concurrent transactions.

Definition 6 *(Conflict-zone): Transaction T_i is a member of transaction T_j's conflict zone iff*

Since Oak uses Optimistic Techniques (MVCC) in order to handle Concurrency Control, the validation phase is executed after the read phase.

WLOG, when using Optimistic Techniques, a transaction T_j passes iff all of the following conditions are true (reference principles of distributed database systems, tamer özsu):

- All transactions T_i, where $t_{start}(T_i) < t_{start}(T_j)$, finished writing before T_j started reading.
- All transactions T_i, where $t_{start}(T_i) < t_{start}(T_j)$ and T_i are writing while T_j is reading, are writing items not read by T_j.
- All transactions T_i, where $t_{start}(T_i) < t_{start}(T_j)$ and T_i are validating while T_i is reading, are writing items not read by T_j and T_i is not writing items written by T_j.

8.4 Querying

Oak is commonly queried using content-and-structure (CAS) queries. Given a node, a property and its value, a CAS query returns all descendants of the node which have the property set to the value.

We see that Algorithm 4's performance is dependent on node m's tree depth (1st loop) and on the number of descendants of n (2nd loop).

Descendants of a value node (v_k) in the PI are not guaranteed to satisfy a CAS Query. Let's consider the example depicted in Figure **??**. Obviously $Q_{\chi,1,/} = \{$ `/a`, `/a/b` $\}$. Assume now that `/a` does not have a property χ anymore. $Q_{\chi,1,/} = \{$ `/a/b` $\}$ but the PI remains the same. $/a$ still is a member of the PI (under `/index/`χ`/1`) but does not satisfy the CAS Query anymore.