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

Contents

List of Figures

List of Tables

1 Introduction

A content store is a database system which supports hierarchical (tree-structured) data. It behaves similarly to a traditional file system, that is, storing files/ data with metadata. Like common relational DBMSs, content stores also can provide transactions and querying.

Apache Jackrabbit Oak (reference) is such a tree-structured content store, 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 MVCC (reference), a commonly used optimistic technique (reference). Whilst Oak is responsible for handling the database logic (?), Oak does not persist the data itself. It relies on MongoDB (reference), a popular document store, to persist the data. Although Oak is an Open-source project, it is being actively maintained by Adobe (reference). Adobe makes use of Oak in one of their products, specifically the Adobe Experience Manager (AEM).

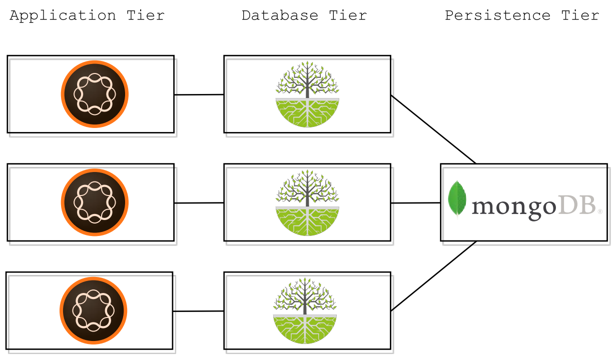


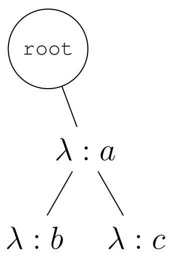
Figure 1.1: Apache Jackrabbit Oak's system architecture. The application, Adobe Experience Manage in this figure, connects to Oak.

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 I will introduce you to an instance of a problem Oak is having and we will briefly introduce a solution which results in higher throughput under certain circumstances. We will revisit Oak's reference implementations and modify them in order to satisfy our solution. Lastly, we will formalize the solution.

2 Oak's Mechanics

2.1 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. 2.1 shows a tree alongside its persisted state. Each tree node is persisted in the shape of a JSON document in MongoDB. Each node is identifiable its tree-depth concatenated with its absolute path from the root node, denoted as `"_id"`. A node's properties, are represented by key-value pairs inside the node's JSON document. `"_id"` can be considered as a property.



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

Figure 2.1: A tree and its JSON representation.

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. In **??**, we take a look at node `/a/c` and see how `x`'s value changes over time. 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 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`.

2.2 Oak's basic Operations

In this chapter we will try to understand how Oak handles basic operations such as queries and writes.

```
[
    {
        "_id": "2:/a/c",
        "x": {
            "r15ca9f191c0-0-1": "1",
            "r15cabff1500-0-2": "2",
            "r15cac0dbb00-0-2": "3"
        },
        /* ... */
    },
    /* ... */
]
```

Figure 2.2: A node's property in detail.

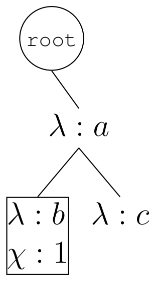
2.2.1 Querying

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.

Definition 1 *(CAS-Query):*

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

Where k denotes the property name (key), v the value, m a node, $n[k]$ property k of node n, $desc(m)$ all descendants of node m. It is worth mentioning that $m \notin desc(m)$.



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

$$Q_{\chi,1,a} = \{\, b \,\}$$

Figure 2.3: CAS Query example.

In order to answer CAS Queries efficiently, a property index (PI) can be implemented. A property index can be constructed as follows:

1. We create an `/index` node.
2. `/index` is a child of the root node.
3. k is a child of `/index` iff $\exists n(n[k] \neq \texttt{NIL})$.
4. v_k is a child of k iff $\exists n(n[k] = v_k)$.

5. A sequence of nodes s starting with the top level node t and ending with node n, i.e $s = \langle t, \ldots, par(n), n \rangle$, are descendants of v_k iff $n[k] = v_k$

By structuring the PI as shown in 2.2.1, a CAS query can be executed as stated in 1.

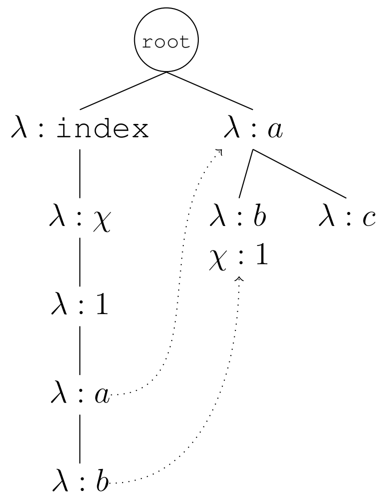


Figure 2.4: Tree with Property Index.

Algorithm 1: CAS Query with Property Index.

```
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 $d[k] = k$ then
            $r \longleftarrow r \cup \{trunc(d)\}$
    return $r$
```

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, $d[k]$ returns the latest value of property k of node d and $trunc($`/k/v/a/b`$) =$ `/a/b` truncates the property name and value from the node's path.

We see that Algorithm 1'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 2.2.1. 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.

2.2.2 Updates

In this chapter we will see how Oak handles writes. 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). An update is composed as follows:

$$Read \longrightarrow Validate \longrightarrow Write$$

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 the tree. As you remember, each value of a node's property is accompanied by a timestamp. A transaction only sees the most recent value of a property up to the time the transaction started. Let T_i denote a recently started transaction. Let $t_{start}(T_i)$ denote the instance of time T_i started. Let v_i denote a version of a value and $t(v_i)$ the instance of time the value was committed. Value v_i is visible to T_i iff **a)** v_i was successfully committed before $t_{start}(T_i)$, i.e., $t(v_i) \leq t_{start}(T_i)$ and **b)** there does not exist any other value v_j such that $t(v_i) < t(v_j) \leq t_{start}(T_i)$.

This ensures that concurrent transactions can not mutate T_i's read values. Figure 2.2.2 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 2.5: Assume transaction T_i starts at $t_{start}(T_i) =$ `01:02`. This implies that node's `/a/b` property `x` has value 1 during transaction T_i, i.e., $t_{start}(T_i) =$ `01:02` $\wedge\, n_i =$ `/a/b` $\implies n_i[k] = 1$.

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:

$$
\begin{aligned}
\Delta T_i = \{ & \\
& wp(\texttt{/a/b}, \texttt{x}), && \triangleright \texttt{ remove x from /a/b} \\
& wn(\texttt{/index/x/1/a/b}), && \triangleright \texttt{ remove node /index/x/1/a/b} \\
& wn(\texttt{/index/x/1/a}), && \ldots \\
& wn(\texttt{/index/x/1}), && \\
& wn(\texttt{/index/x} && \\
\} &
\end{aligned}
\tag{2.1}
$$

Validate

During the validation phase, Oak determines if there is any interference with concurrent transactions. Since Oak uses MVCC, an Optimistic Technique, in order to handle Concurrency Control, the validation phase is executed after the read phase.

WLOG, the validation phase of a transaction T_j when using Optimistic Techniques is passed iff one of the following mutually exclusive conditions is true (reference principles of distributed database systems, tamer özsu):

- Iff all transactions T_i, where $t_{start}(T_i) < t_{start}(T_j)$, have finished writing before T_j started reading.
- Iff 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.
- Iff 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.

Bibliography