# Contention-Management in Indexes for Hierarchical Data[1]

Dmytro Polyanskyy[2]

June 2019

Department of Informatics
Master Basic Module
Kevin Wellenzohn & Michael H. Böhlen

[1]I am grateful for the helpful input of Kevin Wellenzohn and Professor. Böhlen
[2]Matriculation Number : 16-909-160, dmytro.polyanskyy@uzh.ch

## Abstract

Today, hierarchical index structures exist for many various indexing applications. Particularly, in this Master Basic Module, we look at the contention management of transactions from the perspective of a Content Management System (CMS) that has a hierarchical setup. In such a system, we can envision information being constantly published, removed and/or updated. Such frequent operations oftentimes lead to conflicts  particularly when insert and delete operations in the index ultimately propagate to a common ancestor. This is particularly a problem in a Property-and-Path Index (PP-Index). The purpose of this basic module is to experimentally show that is it possible to improve overall performance of such indexing operations by locating frequently updated regions in the data structure where additions/deletions occur regularly. Once these regions are identified, the proposed structure - a Robust Property and Path Index (RPP-Index) identifies such volatile nodes (which would otherwise be constantly inserted and/or deleted by a regular PP Index) and keeps them in the index. Although this sacrifices query performance (since we have a bigger data structure to index), we reduce conflicts which ultimately improves the throughput of the system.

# Contents

## 1 Intuition and Motivation

We consider a Property and Path Index (PP-Index) that supports insertion, deletion and querying. A PP-Index does not assign a special status nor does it distinguish nodes which are updated more frequently than others in its structure. In such an index, path conflicts are prone to occurring because nodes are constantly being added and deleted and in this process, concurrent additions and deletions are likely to occur. This in turn causes conflicts between transactions as some nodes are bound to share common ancestors. In fact, when such an event occurs, that is, when both an insert and delete happen concurrently on the same node, a path conflict occurs and a transaction is subsequently aborted  thus decreasing performance. In Figure 1, we can see an example of such a conflict: Suppose we want to insert a node called Apple under an existing node dubbed as Phones. However, concurrently, we also want to delete the node Blackberry and its upward ancestor node, Phones. Thus, we have two operations that conflict as an operation would like to add a node (in green) to a deleted segment (in red).

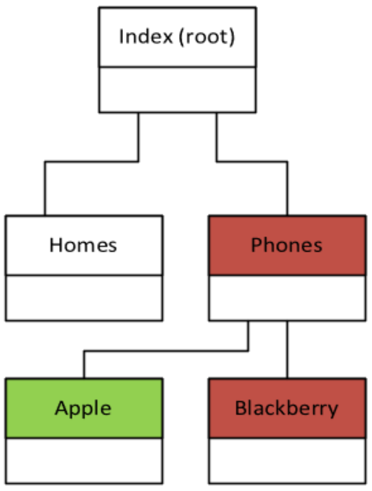


Figure 1: Transaction Conflicts

The goal of this Basic Module is to implement and experimentally evaluate a Robust Property and Path Index (RPP-Index). The RPP-Index improves contention management by isolating highly volatile nodes in the index - that is, nodes which have operations performed on them often. Thus after classifying a node as volatile, an RPP-Index elects to leave that particular node in place rather than delete it (and likely having to reinsert it later). The RPP-Index detects and suspends the deletion of these so called volatile nodes to decrease the number of conflicts which lead to an abort. However it is important to note that the RPP-Index does come with a compromise in the sense that the index to traverse will be larger in size. This is because nodes that would otherwise have been deleted in a PP-Index may be classified as volatile and thus not be removed in an RPP-Index. The goal for this experiment is to investigate parameters for an RPP-Index (called a threshold) using a real world data set to determine how sensitive the index should be towards classifying nodes as volatile and also how threshold parameters impact different types of workloads.

## 2 RPP Index

### 2.1 Volatile Nodes and Sliding Windows

Here we define two terms, volatility as well as the sliding window. Simply put, the volatility of a node is defined by the number of times it was inserted or deleted. The sliding window parameter helps classify a node as volatile or not; it is the measure of how much of a recent workload is used to determine if a node is volatile or not. As the sliding window parameter is increased, the likelihood of a node being classified as volatile in also increased - this is because the sliding window forces the RPP-Index to consider a larger transaction history for a node.

#### 2.1.1 Path Conflicts

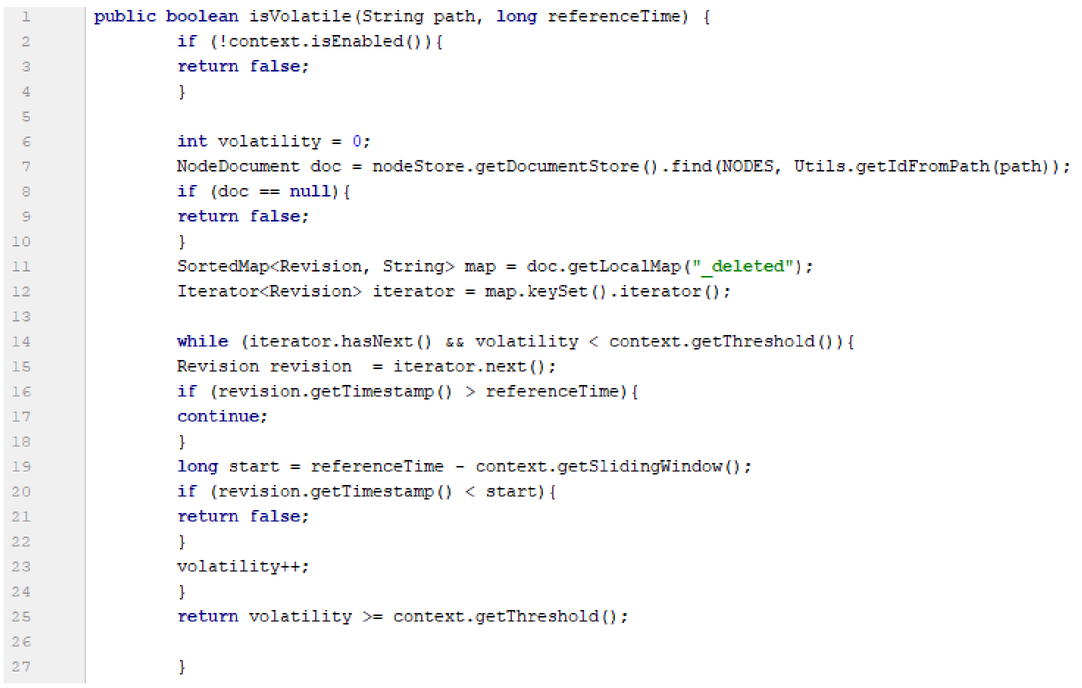
Volatile nodes in an index are the key contributor for path conflicts and transaction aborts. These nodes often exist where transactions modify their children nodes regularly (i.e. often insert/delete its leaf nodes). What an RPP-Index does is holds off on removing these volatile node since it is reasonably likely to have operations done on it again in the near future.

### 2.2 Threshold

The threshold parameter, $t$ is the heart of an RPP-Index. It sets the value for determining if a node in the index is to be classified as volatile or not. A threshold value of infinity is equivalent to a regular PP-Index as this indicates nodes will never become volatile. At lower threshold values, nodes are more likely to be classified as volatile. In this subsequent sections we will experimentally examine threshold values in more detail.

### 2.3 Computing Volatility

The RPP-Index attempts to prevent aborts by doing a Volatility Computation. In the Figure 2 below, we can see that the function isVolatile whose purpose is to determine whether or not a node is to be classified as volatile has two main components - the threshold and the sliding window. The functions main purpose is to determine whether a node in the path is not a child node and also has a volatility that falls within the specified threshold (Line 14). Thus, within Jackrabbit Oak, each of our nodes has a field called _deleted (Line 11) which contains information on whether a node was inserted or deleted and also a timestamp on when the operation occurred. Subsequently, by using the sliding window and comparing the reference time with the timestamp from the _deleted field, we are able to determine whether a node should be classified as volatile or not for a given threshold.



```
public boolean isVolatile(String path, long referenceTime) {
        if (!context.isEnabled()){
        return false;
        }

        int volatility = 0;
        NodeDocument doc = nodeStore.getDocumentStore().find(NODES, Utils.getIdFromPath(path));
        if (doc == null){
        return false;
        }
        SortedMap<Revision, String> map = doc.getLocalMap("_deleted");
        Iterator<Revision> iterator = map.keySet().iterator();

        while (iterator.hasNext() && volatility < context.getThreshold()){
        Revision revision  = iterator.next();
        if (revision.getTimestamp() > referenceTime){
        continue;
        }
        long start = referenceTime - context.getSlidingWindow();
        if (revision.getTimestamp() < start){
        return false;
        }
        volatility++;
        }
        return volatility >= context.getThreshold();

        }
```

Figure 2: Volatility Code

## 3 Experimental Setup

The implementation of the RPP-Index and its subsequent volatility node determination is done in conjunction with Java and Apache Jackrabbit Oak.[3] Each experiment was run for 5 minutes in a balanced workload manner at a prepopulation factor of 0.1 (10%) - meaning that the index already contained 10% of the nodes from the data-set before the experiment actually began. The data used is a real-life dataset from the Dell website and contains 12.244.893 nodes.[4]

### 3.1 Skew

Skew is important for our experiment because it allows us to concentrate a certain amount of operations on a specific region of the index. For the purposes of our experiment we use a zipf distribution whose skew will range from $[0 - 2]$, where a 0 skew represents no contortion of transactions on a specific region, while a higher skew (up to 2) represents operations on certain nodes to have a greater likelihood of reoccurring. The skew allows us to study the performance of an RPP-Index on different types of transactions. For a regular PP-Index, a high skew is unfavorable for performance because then you always focus on the same nodes which causes many conflicts. In an RPP-Index however, a higher skew causes the abort ratio to decrease since more active nodes in the index become volatile which consequently

[3]Credits to Kevin Wellenzohn's paper - we will follow a similar approach/setup in this experiment

[4]https://dell.com; Dell uses AEM as CMS and Oak as HDDBS for its website. The Dell dataset has been extracted from a dump of Oak.

leads to less transaction aborts.

## 4 Experiments and Discussion

When assigning a lower volatility threshold, the RPP classifies more nodes as volatile. While this increases the time it takes for every transactions to execute, it has the favourable effect of decreasing the number of transaction aborts - this is confirmed in our experiment. Furthermore, even though there are more nodes that need to be traversed, throughput is still increased due to the significantly improved abort-ratio. The lower the threshold, the slower the query  however a lower number of transaction aborts occur as well. Small values for $t$ reduce the number of aborts but slow down queries while large values for $t$ improve query time but increase the number of aborts that one would expect to happen.

### 4.1 Abort Ratio

Looking at Figure 3, we can see our experiments confirm the above intuition. For example, we see that a higher skew (where skew is $> 1$) negatively affects a PP-Index (as the threshold parameter tends towards infinity). More interestingly, we see that for lower value thresholds (e.g. ones less than $< 100$), a skew higher than 0 actually decreases the abort ratio. This is because under lower threshold values, small regions become volatile rather quickly and do not conflict with each other as much. For lower skew values, low thresholds still benefit, but because this means that transactions are not as concentrated in one region, aborts are more prone to occurring. Lastly, for the PP-Index (when the threshold tends toward infinity), a high skew is bad as transactions focus on the same nodes and you endure many aborts since there is no threshold parameter to make a node volatile and prevent it from constant inserting/deleting.

For thresholds 0 and 10, we can clearly observe low abort ratios that tend to 0 as the skew increases. For an RPP-Index with a higher threshold, 100, the improvements in abort ratio's are more pronounced as the skew increases (since node operations are more likely to happen in clustered volatile areas). Finally, we see that for a threshold value that approaches infinity (essentially a PP-Index), the abort ratio's are worse then when compared to all the RPP-Index's. However, we can observe that when skew is 0, the RPP index with the highest non-infinite threshold (i.e. threshold of 100) has similar abort ratio's to that of the regular PP-Index where nodes never become volatile and conclude that the improvements are less dramatic under such conditions.

### 4.2 Throughput

Throughput is defined as the number of successful transaction commits per unit time. In Figure 4, we can see that the number of successful commits for an RPP-Index clearly outperforms the number of successful commits in a normal PP-Index for our Data-set. Furthermore, we can see that at higher threshold levels, where nodes rarely become volatile (and is the index is then basically a PP-Index), skew does not affect successful commits too much. On the other hand, in an RPP Index with a low threshold, we can see that we improve throughput massively with a higher level of skew due to the lack of transaction aborts that would occur otherwise. We furthermore see that throughput deteriorates quite rapidly as we increase threshold values past 100 for all levels of skew - this is again due to the fact that the index resembles a PP-Index where the inserts and deletes are uncontrolled. Hence, for our experiment, we see that the optimal threshold to choose should be between 0 and 100 depending on the skewness of the workload.

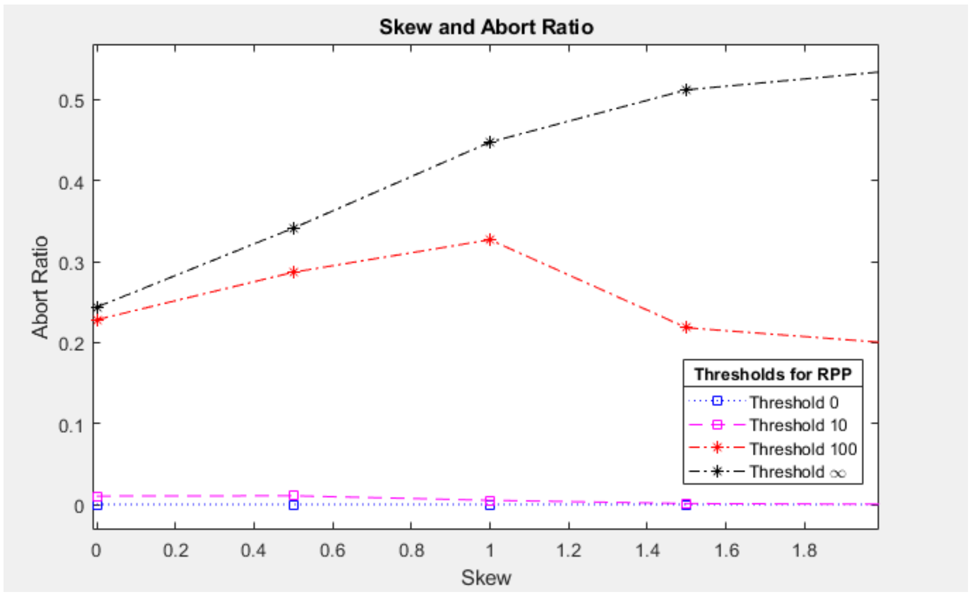


Figure 3: Abort Ratio

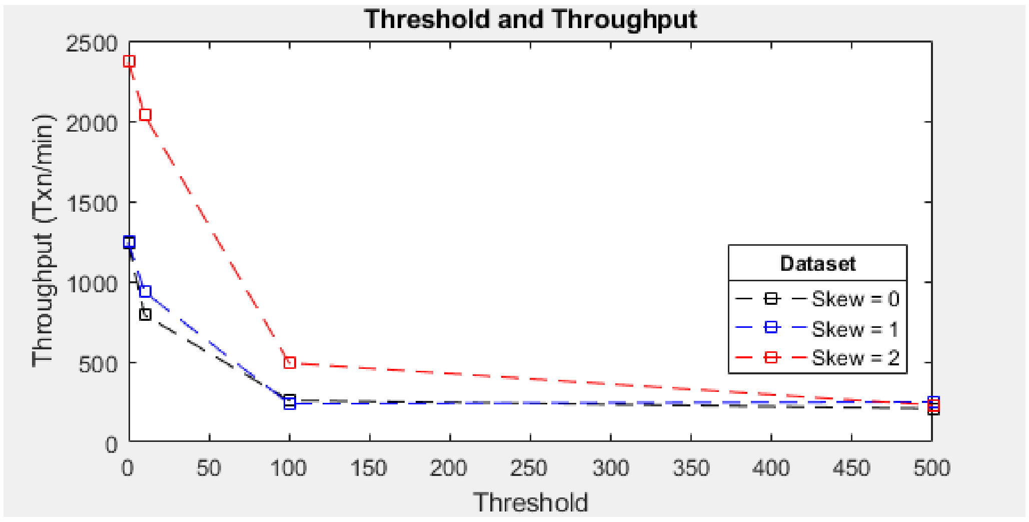


Figure 4: Throughput (Transactions per Minute)

## 5 Conclusion

An RPP Index's main function is to find the correct amount of compromise between the volatility threshold and query performance. A higher threshold theoretically improves query speed, but consequently comes at the expense of a higher abort ratio as more nodes are bound to conflict with each other. On the other hand, a lower volatility threshold decreases the abort ratio at the expense of query speed.

There are many considerations to make when designing an RPP-Index. The main one is to understand whether the workload is more read-intensive or write-intensive. By nature, a workload that is write-intensive will experience most of the benefits of a well-defined RPP-Index as it will prevent inevitably more frequent path conflicts which cause transactions to abort. On the other hand, in a read intensive workload where aborts would virtually never happen, a low threshold is not so important and the RPP-Index would not see as many benefits in comparison to a regular PP-index. Finding the correct parametrization is the key challenge for implementing a successful RPP-Index that will reduce contention and increase throughput.

## References

K. Wellenzohn, M. H. Böhlen, Sven Helmer, Marcel Reutegger, Sherif Sakr *Workload-Aware Contention-Management in Indexes for Hierarchical Data*, (2019).