Workload-Aware Contention-Management in Indexes for Hierarchical Data

Kevin Wellenzohn[1] Michael H. Böhlen[2] Sven Helmer[3] Marcel Reutegger[4]

Abstract: Queries in hierarchical databases (HDBs) often combine predicates referring to values of node properties with path predicates relating to the structure, which are called property-and-path (PP) queries. Usually, PP indexes are used to support these types of queries efficiently. In an environment in which HDBs are updated concurrently, we encounter conflicts which may lead to transaction aborts. We identify *preventable aborts* caused by conflicts in the index, while the operations in the actual database are executed without any problems. These index conflicts are due to the deletion of a path in the index concurrently taking place with an insertion underneath a node on the deleted path. We leverage recent workload information to detect and suspend the deletion of substructures in PP indexes that are likely to conflict with concurrent insertions. However, the suspension of these deletions has a detrimental effect on the query performance, which means this becomes a tradeoff between the number of transaction aborts and the speed of the query evaluation. We implement our approach in Apache Jackrabbit Oak and FOEDUS, experimentally investigate the tradeoff, and show how to balance the effects to maximize the transactional throughput for a given workload.

Keywords: hierarchical databases; structural indexes; concurrency control

1 Introduction

A lot of the data in business and engineering applications, such as bills of materials [Fi13], enterprise asset hierarchies [Fi13], and business rules [Lo15], is organized in a hierarchical way. Additionally, many NoSQL content stores manage hierarchical data, e.g. in the form of JSON. Similar to relational databases, though, in which we index a subset of attributes in a relation to speed up query evaluation, in hierarchical databases (HDBs), we also often index a subset of nodes in a hierarchy relevant for frequent queries. This set of nodes is application-dependent and we assume that a user flags these nodes, which are then indexed by the system.

Clearly, in a multi-user environment, node indexes can become a bottleneck if nodes are frequently updated concurrently. This leads to conflicts not just on the node level, but may also result in path conflicts on common ancestor nodes of updates. We show how to prevent path conflicts in node indexes that would otherwise lead to transaction aborts. Figure 1a

[1] University of Zurich, Dept of Informatics, Binzmühlestrasse 14, 8050 Zurich, Switzerland wellenzohn@ifi.uzh.ch
[2] University of Zurich, Dept of Informatics, Binzmühlestrasse 14, 8050 Zurich, Switzerland boehlen@ifi.uzh.ch
[3] University of Zurich, Dept of Informatics, Binzmühlestrasse 14, 8050 Zurich, Switzerland helmer@ifi.uzh.ch
[4] Adobe Systems, Barfusserplatz 6, 4051 Basel, Switzerland mreutegg@adobe.com

doi:10.18420/BTW2023-03

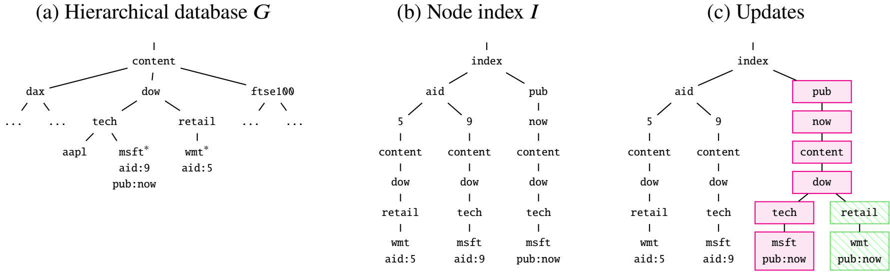


Fig. 1: Transactions T_i and T_j conflict when T_i deletes index node `msft` and its ancestors upwards (red nodes), while T_j adds a new child to a deleted ancestor (green hatched nodes).

shows an example of a content management system (CMS), such as Adobe Experience Manager [Ad23] or Magnolia [Ni06], built on top of an HDB. In this application scenario, users change webpages in a private workspace of the CMS and, when finished, flag them as being publishable: this adds a property `pub` with the value `now` to publishable nodes. For example, in Figure 1a, the node `msft` is ready to be published. Eventually, the CMS pushes the changes to the webserver and removes the property `pub:now` from the node `msft`. We marked the indexed nodes in the HDB with an asterisk (*) to make them easier to spot and Figure 1b shows the corresponding index, containing the flagged nodes and their ancestors. We now run the two transactions T_i and T_j on this database: T_i removes the property `pub` from the node `msft`, while T_j concurrently adds `pub:now` to the node `wmt`. These updates need to be propagated to the index (Figure 1c). T_i's deletion of index node `msft` propagates upwards: the empty path `pub/now/content/dow/tech/msft` is deleted (red nodes) since we do not have any nodes with the property `pub` anymore. Concurrently, T_j needs to add the green nodes `retail/wmt` below `dow` to update the index. This results in a path conflict between T_i and T_j since T_i wants to delete a path, while T_j wants to insert a branch on this path. The conflict arises at the shared ancestor nodes. In our example, the nodes `msft` and `wmt` in the index have the lowest common ancestor `dow` and share the path from `dow` to the root. We propose a technique identifying problematic regions in node indexes leading to conflicts, i.e., regions in which the same shared ancestor nodes are frequently inserted and deleted. In our example, if we had kept the path from `dow` upwards in the index when removing node `msft`, anticipating an insertion, we could have inserted node `wmt` without any problems. However, this comes at a price: we temporarily keep purposeless nodes in the index, slowing down query evaluation. We experimentally show how to balance reducing contention with query performance to maximize the throughput.

In summary, we make the following contributions:

- We describe and define *preventable aborts*, which are aborts caused by propagated node insertions and deletions in node indexes for hierarchical databases.

- We introduce the notion of *node volatility*, which allows us to identify nodes that are repeatedly involved in preventable aborts.
- We develop the *robust node index* (RNI), which detects and suspends the deletion of volatile nodes to decrease the number of preventable aborts significantly.
- We implemented RNI in Apache Jackrabbit Oak [Ap22] and FOEDUS [Ki15] and evaluated it experimentally with different workloads and datasets. We show that making the node index *robust* is more effective than (a) alternative concurrency-control protocols reducing aborts [WK16] and (b) lazy techniques for node deletions in indexes [Lo04, LS97], increasing the throughput by up to a factor of six.

2 Related Work

High-contention workloads are a significant bottleneck for database systems [Ap17, Ha17, RFA16, RTA14, Ti18]. We discuss approaches that deal with contention (a) on the level of the concurrency-control protocol and (b) on the level of the index.

The most similar approach to RNI among the protocol-level approaches is MOCC [WK16] (that is the reason why we chose it for the experimental evaluation). It starts by using optimistic concurrency-control (OCC) to synchronize accesses to records. However, it also monitors the number of aborts caused by a record due to concurrent accesses. If this number reaches a certain threshold, the record is regarded as hot and MOCC switches to a pessimistic locking protocol to reduce the number of aborts. Like MOCC, in RNI we monitor the load on heavily contentious nodes and switch to a different mode when necessary. In the following, we briefly describe other protocol-level approaches. Yuan et al. [Yu16] reduce the number of aborts in OCC by aborting a transaction only if an *essential pattern* exists between transactions, which is more restrictive than the read-write conflict OCC checks for. Similarly, Bumper [DR13] only aborts a transaction if a so-called *triad* (conceptually similar to an essential pattern) is detected. Tian et al. [Ti18] propose a contention-aware locking scheme that reduces the overall lock-waiting times. They choose which transaction T to grant a lock to based on the number of other transactions that depend on T's progress. Johnson et al. [JPA09] reduce contention in the lock manager by passing hot locks directly from transaction to transaction, without releasing and re-acquiring them. QURO [YC16] analyzes program code and reorganizes the code within transactions to reduce contention by acquiring a lock as late as possible. Deterministic concurrency-control has been proposed to reduce synchronization in replicated databases [TA10]. A transaction acquires all locks at its start, which means that transactions competing for exclusive access to a contended record must execute in serial order [Th12]. This prevents conflicts and aborts due to deadlocks at the expense of concurrency (especially under contention). Calvin [Th12] and other deterministic systems require that the read/write sets of transactions be known a priori [Ha17], which is not the case in our application scenario.

Frequently inserting and deleting nodes into and from indexes is a known concurrency bottleneck [LY81, LS97]. Lomet et al. [Lo04, LS97] propose to defer node deletions during updates in B-trees. During deletion, the key is (eagerly) removed from the correct leaf and if it becomes underutilized, the node deletion is deferred and processed later. When to exactly process deferred operations is not specified [LS97], though. Even though there is a lot of work on indexing hierarchical data [HL11, Sh15], concurrency control (CC) specifically for indexes in HDBs has received little attention in comparison to CC for HDBs in general [Be15, Be11, Fi02, HHL06]. For instance, there are path indexes only considering the structure, such as DataGuides [GW97] and APEX [CMS02], and indexes that consider the structure and values, such as IndexFabric [Co01] and CAS (content-and-structure) indexes [Ma15, WBH20]. However, none of these papers discuss concurrency control and we believe there is still untapped potential in this area. For example, node deletions in HDB indexes can be suspended as long as the indexed values are removed during the deletion. In general, this is not possible for CC in the HDB itself, as the actual removal of the node is part of a transaction's semantics. Workload-aware indexing has been shown to improve index query and/or update performance [CMS02, Id11, TYJ09]. APEX [CMS02] optimizes frequently queried paths in XML databases. QU-Trade [TYJ09] uses the recent workload to balance the cost of writing/reading frequently updated/queried objects. Again, none of these approaches discuss concurrency control. In contrast, adaptive indexing incrementally sorts and refines an index during query execution [Id11], sketching ideas on how to realize CC in the future work section. This promise is delivered in [Gr14], which provides more details on concurrency control. Queries can cause contention if they concurrently attempt to optimize overlapping query-ranges. In that case adaptive indexing forgoes the chance to optimize the index and skips the optional optimization.

3 Background

3.1 Data Model

We model a database G as an unordered tree that is defined as a set of nodes $G = \{n_1, n_2, \ldots\}$.[5] A node $n = /\lambda_1/\ldots/\lambda_x$ is uniquely identified by the *node labels* λ_i on the path from the root node to node n. The last node label in this sequence, λ_x, is n's label. The label of the root node is the empty string.

Example 1 *Consider the database G in Fig. 1a. Node n =* `/content/dow/tech/msft` *has label* `msft`. *If no ambiguity arises, we shorten node labels by using only the initials, hence n =* `/c/d/t/m`*. G consists of the labels of all its nodes* {`/`, `/c`, `/c/d`, `/c/d/t`, `/c/d/r`, `/c/d/t/a`, `/c/d/t/m`, `/c/d/r/w`, ...}. *From now on, we denote a node by its label λ; the full ID can be derived from its ancestors' labels.* □

[5] Based on Apache Jackrabbit Oak's data model [Ap22].

A node may have an arbitrary number of properties. We define the *property set* $P(G)$ of tree G as a set of triples (n, k, v), which denote that node $n \in G$ has property k set to value $v \neq \epsilon$. We use the notation $n[k]_{(G)} = v$ iff $(n, k, v) \in P(G)$, and $n[k]_{(G)} = \epsilon$ iff $\nexists v((n, k, v) \in P(G))$ to denote that node n does not have property k in G. If it is clear from the context which tree we are referring to, we omit the subscript and write $n[k] = v$ or $n[k] = \epsilon$. A node n is an ancestor of node m (and m is a descendant of n) iff $n = /\lambda_1/\ldots/\lambda_x$ is a prefix of $m = /\lambda_1/\ldots/\lambda_x/\ldots/\lambda_y$ or, stated shortly, prefix(n, m). A node is an ancestor and descendant of itself, i.e., prefix(n, n) is true for every node n.[6]

Example 2 *Consider Fig. 1 and let n =* `/c/d/t/m`*. We have* $n[\texttt{pub}]_{(G)} = \texttt{now}$ *before running transaction* T_i *and* $n[\texttt{pub}]_{(G)} = \epsilon$ *after running* T_i*. Node* `/c/d` *is an ancestor of n, since prefix(*`/c/d`*,* `/c/d/t/m`*) is true.* □

Typically, queries in HDBs are property-and-path (PP) queries, meaning we need to provide a property, a value to compare to, and a path. As we only consider paths (and not twigs), the order of the siblings in a tree does not matter.

Definition 1 *(*PP Query*) A PP query* $Q = (k, v, m)$ *returns the set of nodes with property k equal to value v that are descendants of m, i.e.,* $\{d \mid d[k] = v \wedge \textit{prefix}(m, d)\}$*.* □

3.2 The Property-and-Path (PP) Index

A property-and-path (PP) index I is used to efficiently query all nodes in a subtree that have a property k set to a value v. Essentially, a PP index is modeled as an unordered tree, similar to an HDB as described in Section 3.1. The first label of every path in I is called `index`, the second is the name of a property k, and the third is a value v for k. This is then followed by paths to all nodes in the indexed database G that have a property k with a value v (cf. Figure 1b). In a typical application, a node can have many properties (e.g., author ID `aid` and other metadata), but usually only some are indexed.

Querying: Evaluating PP query $Q = (k, v, /\lambda_1/\ldots/\lambda_x)$ with index I translates to navigating down the path `/index/`$k/v/\lambda_1/\ldots/\lambda_x$, traversing all descendants of λ_x, searching for index nodes n with $n[k] = v$, and returning their corresponding content nodes. These are obtained by truncating the three leading node labels of n. For example, for index node `/i/p/n/c/d/t/m` the corresponding content node is `/c/d/t/m`.

Example 3 *Assume we want to find pages under* `dow` *that are ready for publication, i.e., we run the query* $Q = (\texttt{pub}, \texttt{now}, \texttt{/c/d})$ *on G. Using the index I in Figure 1b, we descend to*

[6] This is similar to the ancestor-or-self and descendant-or-self axes in XPath.

node `/i/p/n/c/d` *and check if any descendant contains the property-value pair* `pub:now`. *This is the case for node* $m =$ `/i/p/n/c/d/t/m`*, thus the query returns* {`/c/d/t/m`}. *(As we will see later, there may be* unproductive nodes *in the index missing the property-value pair* `pub:now`. *These nodes are currently not active and will not be returned.)* □

Insertion: An insertion into index I is described by a triplet $(k, v, n = /\lambda_1/\ldots/\lambda_x)$, where k is a property, v is a value, and m is a node (that now has a property k set to value v). The insertion is executed as follows. First, the system traverses the nodes along path $n = \texttt{/index/}k/v/\lambda_1/\ldots/\lambda_x$ or creates them if they do not exist yet. Then, the system sets $n[k] = v$.

Example 4 *When transaction* T_j *adds property* `pub:now` *to node /c/d/r/w in G (cf. Figure 1a), we add a branch* `/r/w` *underneath* `/i/p/n/c/d` *in index I in Figure 1b, setting the property* `pub` *in node* `wmt` *to* `now`. □

Deletion: A deletion is also described by a triplet $(k, v, n = /\lambda_1/\ldots/\lambda_x)$. During the deletion, we first descend to node $n = \texttt{/index/}k/v/\lambda_1/\ldots/\lambda_x$ and remove property k by setting $n[k] = \epsilon$. However, it does not stop there. We prune n and all its ancestors one by one as long as they are a leaf and do not have the property k (we do not prune the index definition `/index`).

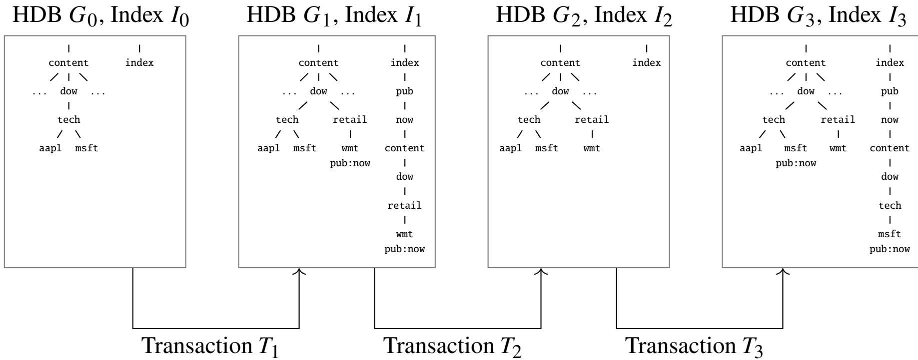
Example 5 *When transaction* T_i *removes property* `pub` *from node* `/c/d/t/m` *in G, the property* `pub` *is removed from index node* $n =$ `/i/p/n/c/d/t/m`*. As n is now a leaf node without a property, it is deleted. The pruning continues up to* `index`*, essentially removing the path* `p/n/c/d/t/m` *from index I (cf. Figure 1b).* □

4 Conflicts and Aborts

This section describes concurrent operations that lead to conflicts in HDBs. We assume multi-version concurrency control (MVCC) with snapshot isolation. MVCC resolves conflicts by aborting transactions.

4.1 Snapshots

The state of an HDB logically progresses from one snapshot of the database to the next as transactions commit. A *history* is a sequence $H = \langle \ldots, G_i \rangle$ of databases (including any indexes) ordered by commit time. A committed HDB $G_i \in H$ is an immutable *snapshot*. A new transaction T_j logically creates a mutable copy G_j of the last committed snapshot



Each transaction's changes to the content subtree:
▹ Transaction T_1: An author adds a webpage `wmt` with property `pub` set to `now`
▹ Transaction T_2: The CMS removes property `pub` from `wmt` after pushing it to the webserver
▹ Transaction T_3: An author publishes webpage `msft` by setting its property `pub` to `now`

Fig. 2: A typical CMS-workload in which authors repeatedly publish webpages.

$G_i \in H$ with snapshot G_i being the *base snapshot* of T_j. Transaction T_j applies all its read and write operations on G_j.

Example 6 *Our running example (see Figure 2) shows an initial HDB G_0 with a corresponding (empty) index I_0.*[7] *After running the transactions T_1 to T_3, one after the other, we have the history H with the committed snapshots G_0 to G_3: $H = \langle G_0, G_1, G_2, G_3 \rangle$.* □

4.2 Conflict Detection and Handling

Before going into the details of resolving conflicts between concurrent transactions, we define basic notions of transactions in HDBs. A transaction T_j can change a database with two primitives: node-write operations $\mathsf{wn}(n)$, to insert or delete nodes, and property-write operations $\mathsf{wp}(n, k)$, to add, delete, or change a property k of node n.

Definition 2 *(*Write Set*) The write set ΔT_j of a transaction T_j is the set of node- and property-write operations in tree G_j. Let G_i be T_j's base snapshot. ΔT_j contains:*

1. *Node-write operations* $\mathsf{wn}(n)$*:*

$$\mathsf{wn}(n) \in \Delta T_j \Leftrightarrow (n \in G_i - G_j) \vee (n \in G_j - G_i)$$

[7] For the sake of simplicity, we dropped the property `aid`.

2. *Property-write operations* $\mathsf{wp}(n,k)$*:*

$$\begin{aligned} \mathsf{wp}(n,k) \in \Delta T_j \Leftrightarrow &(n \in G_j - G_i \wedge n[k]_{(G_j)} \neq \epsilon) \vee \\ &(n \in G_i - G_j \wedge n[k]_{(G_i)} \neq \epsilon) \vee \\ &(n \in G_i \cap G_j \wedge n[k]_{(G_i)} \neq n[k]_{(G_j)}) \end{aligned}$$
□

Example 7 *The write set* ΔT_1 *of transaction* T_1 *in Figure 2 contains the following operations:* wn(`/c/d/r` *and* `/c/d/r/w`)*, creating the node* `retail` *and then the node* `wmt`*, and* wp(`/c/d/r/w`, `pub:now`)*, adding the property* `pub` *with the value* `now` *to the node* `wmt`*. Moreover, it also includes the operations* wn(`/i/p`)*,* wn(`/i/p/n`)*,* wn(`/i/p/n/c`)*,* wn(`/i/p/n/c/d`)*,* wn(`/i/p/n/c/d/r`)*,* wn(`/i/p/n/c/d/r/w`)*, and* wp(`/i/p/n/c/d/r/w`, `pub:now`)*, updating the index.* □

We have to distinguish different types of conflicts between two concurrent transactions T_i and T_j: *path conflicts* and *property conflicts*. Path conflicts include at least one wn operation that inserts or deletes a node and are denoted by wn-wn, wn-wp, and wp-wn. We encounter a wn-wn conflict if one transaction adds/deletes a node while the other adds/deletes one of its descendants, i.e., the label of one node is a prefix of the other. A wn-wp or wp-wn conflict exists if one transaction deletes a node, while the other adds, changes, or deletes any property on the *same* node. Property conflicts (wp-wp conflicts) occur when T_i and T_j simultaneously try to change the *same* property on the *same* node.

Definition 3 *(*Path Conflict*) We have a path conflict between concurrent transactions* T_i *and* T_j *iff at least one of the following conflicts occurred:*

1. wn-wp *conflict:* $\exists n, k(\mathsf{wn}(n) \in \Delta T_i \wedge \mathsf{wp}(n,k) \in \Delta T_j)$
2. wp-wn *conflict:* $\exists n, k(\mathsf{wp}(n,k) \in \Delta T_i \wedge \mathsf{wn}(n) \in \Delta T_j)$
3. wn-wn *conflict:* $\exists n, m(\mathsf{wn}(n) \in \Delta T_i \wedge \mathsf{wn}(m) \in \Delta T_j \wedge (\mathit{prefix}(n,m) \vee \mathit{prefix}(m,n)))$ □

Definition 4 *(*Property Conflict*) A property conflict, i.e.,* wp-wp *conflict, exists between concurrent transactions transactions* T_i *and* T_j *iff* $\exists n, k(\mathsf{wp}(n,k) \in \Delta T_i \wedge \mathsf{wp}(n,k) \in \Delta T_j)$ □

Example 8 *Assume that transactions* T_4 *and* T_5 *start concurrently in HDB* G_3 *(see Figure 3), hence* G_3 *becomes* T_4*'s and* T_5*'s base snapshot.* T_4 *and* T_5 *run into a path conflict (*wn-wn*), since the former deletes a node under which the latter adds a child. The conflicting operations in the write sets of* T_4 *and* T_5*,* wn(`/i/p/n/c/d`) $\in \Delta T_4$ *and* wn(`/i/p/n/c/d/r`) $\in \Delta T_5$*, are highlighted in red in Figure 3.* □

When a transaction T_j attempts to commit, a verification phase checks whether T_j conflicts with a concurrent transaction. If a conflict is detected one of the involved transactions

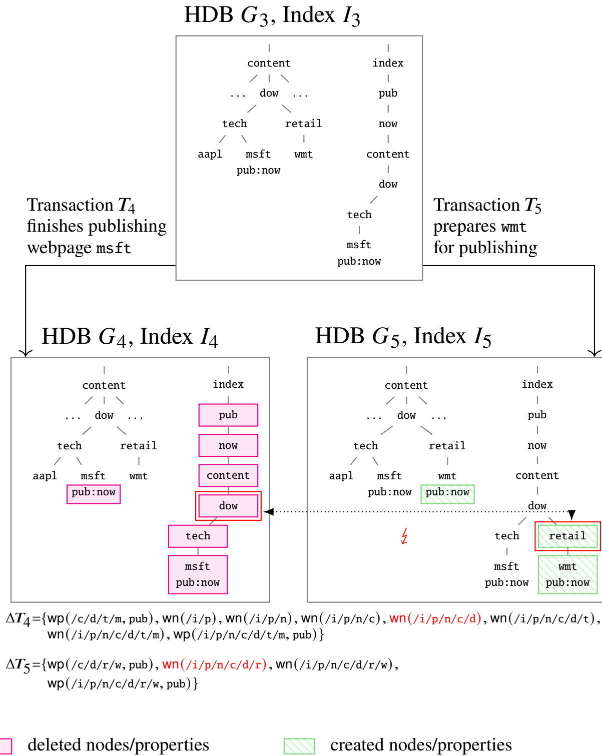


Fig. 3: Transactions T_4 and T_5 conflict, because T_4 deletes index node dow, while T_5 adds child `retail`.

has to abort. Oak implements the first-committer-wins rule [Be95], which means that the transaction that issues the commit first is allowed to commit, while the other is aborted (other policies, such as timestamp-based priority to favor older transactions are also possible). In our running example T_4 commits first and therefore T_5 must abort due to the conflict shown above.

Clearly, if there is a conflict caused by operations in the database, one transaction has to abort. Two different transactions concurrently changing the same node at the same time are just not compatible. However, what is particularly interesting in Example 8 is that the conflict is caused by operations updating the index. The operations updating the actual database G_3 are perfectly fine, as they update properties in two completely different nodes. It turns out that if a conflict occurs only in the index, we sometimes have options to avoid such an abort. We take a closer look at this in the following section.

5 The Robust Node Index (RNI)

5.1 Volatile Nodes

Path conflicts occur frequently in index hotspots where transactions insert and delete nodes sharing a large number of ancestors. We call nodes that are repeatedly inserted and deleted *volatile*. These are a main source for path conflicts in indexes. We propose the *robust node index (RNI)* that detects and manages volatile index nodes. RNI suspends the deletion of a volatile index node, as we expect the node to be inserted again soon. Not repeatedly deleting and inserting a volatile node n means that node-write operations on n, $\mathsf{wn}(n)$, are avoided, reducing contention and, consequently, the number of aborting transactions.

We define the volatility of a node n as the number of times n was inserted or deleted. This corresponds to checking the number of $\mathsf{wn}(n)$ operations that have been executed (cf. Definition 2). In order to do so, we look at the recent transactional workload, which is defined by a sliding window $\mathsf{SW}(H, L)$ of length L over history H. $\mathsf{SW}(H, L)$ denotes the set of transactions that committed over the last $L \geq 0$ time units. Let t_{now} be the current time and $t(T)$ be the commit time of transaction T, then $\mathsf{SW}(H, L) = \{T_j \mid t(T_j) \in (t_{\mathsf{now}} - L, t_{\mathsf{now}}] \wedge G_j \in H\}$.

Definition 5 *(*Volatile Node*) A node n is* volatile *in history H iff the number of transactions in sliding window* $\mathsf{SW}(H, L)$ *that executed a* $\mathsf{wn}(n)$ *operation is at least equal to the volatility threshold* τ*, i.e.,*

$$|\{T \mid T \in \mathsf{SW}(H, L) \wedge \mathsf{wn}(n) \in \Delta T\}| \geq \tau \qquad \square$$

Example 9 *Consider index node* $n =$ `/i/p/n/c/d` *in index* I_3 *in Figure 2. Assuming time* $t_{\mathsf{now}} = 11$ *and sliding window length* $L = 10$*, Figure 4 shows the commit times* $t(T_1)$*,* $t(T_2)$*, and* $t(T_3)$ *of the transactions we ran on our HDB. Since all commit times lie in the sliding window,* $\mathsf{SW}(H_1, L) = \{T_1, T_2, T_3\}$*. All these transactions either insert or delete n, thus* $\forall T \in \mathsf{SW}(H_1, L) : \mathsf{wn}(n) \in \Delta T$ *and for a volatility threshold* $\tau \leq 3$*, node n is volatile.* $\square$

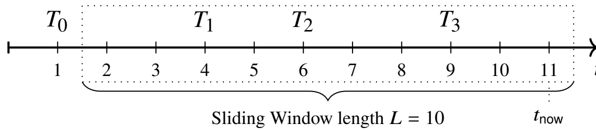


Fig. 4: Transactions T_1 through T_3 from Figure 2 are contained in the sliding window $SW(H_1, 10)$.

RNI checks for volatile nodes during the pruning of nodes in the index. The deletion of pair (v, m) with path $m = /\lambda_1/\ldots/\lambda_x$ and value v from index I is described in Algorithm 1. RNI descends to index node $n =$ `/index/`$k/v/\lambda_1/\ldots/\lambda_x$, deletes n's property k, and tries to prune n. RNI only prunes a node if three conditions are satisfied: (a) it is (or has become) a leaf, (b) it does not have property k, and (c) it is *not* volatile.[8]

[8] It also does not prune the topmost node, `index`.

Algorithm 1: Deletion in RNI

Input: Index I, pair (v, m), and history H. k is a property, v a value, and $m = /\lambda_1/\lambda_2/\ldots/\lambda_x$ a node.

1 $n \leftarrow$ `/index/`$k/v/\lambda_1/\lambda_2/\ldots/\lambda_x$
2 $n[k] \leftarrow \epsilon$
3 **while** $n \neq$ `/index` $\wedge$ isLeaf(n) $\wedge$ $n[k] = \epsilon$ $\wedge$ $\neg$volatile(n) **do**
4 $\quad u \leftarrow n$
5 $\quad n \leftarrow$ parent of n
6 $\quad$ Delete node u

Example 10 *RNI prunes node u_1 =* `/i/p/n/c/d/t/m` *in response to transaction T_4 deleting property* `pub` *from node* `/c/d/t/m`*. The node can be deleted since u_1 is not volatile. The pruning propagates to u_1's parent node u_2 =* `/i/p/n/c/d/t`*, which can also be pruned. However, the parent of u_2, u_3 =* `/i/p/n/c/d`*, is not pruned and the deletion is not propagated farther up the index, because u_3 is volatile (cf. Example 9). Since* `/i/p/n/c/d` *is no longer deleted by T_4, T_5's insertion of a child node is not a conflict anymore.* □

Currently, we have implemented the tracking of volatile nodes in a naive fashion, i.e., we just count the number of insertions and deletions executed on each node. However, the performance of volatility tracking can be improved considerably by employing algorithms from stream processing for finding frequent items. For our purposes, we do not need exact numbers, so an approximation is enough, which improves the performance even more. For instance, Cormode and Hadjieleftheriou use a sketch algorithm for finding frequent items in data streams [CH10].

5.2 Preventable Aborts

As we have seen in Example 10, we can avoid aborting a transaction when a path conflict occurs in the index by not deleting volatile nodes. We now take a closer look at these *preventable aborts*.

Definition 6 *(*Preventable Abort*) Let T_j be a transaction that is aborted due to a conflict with transaction T_i. T_j's abort is* preventable *iff each conflict with T_i is a path conflict in the index.* □

Lemma 1 *Let T_i and T_j be two concurrent transactions. T_i's and T_j's write operations on an existing node n in a RNI index cannot cause a preventable abort if n is volatile or has a volatile descendant.* □

Proof T_i and T_j can only cause a path conflict if both contain operations changing the same property k. If they change different properties, the index updates take place in completely

separate branches of I. Let node n be a node in RNI I. Let d be a volatile descendant of n (recall that n is a descendant of itself). Since d is volatile, neither T_i nor T_j can prune d or any of its ancestors, including n. Therefore $\mathsf{wn}(n) \notin \Delta T_i$ and $\mathsf{wn}(n) \notin \Delta T_j$. As a consequence, we can rule out any path conflict (i.e., wn-wn, wn-wp, and wp-wn conflicts). The only possible conflict between T_i and T_j is a wp-wp property conflict on property k. However, this is a property conflict, i.e., $\mathsf{wp}(n, k) \in \Delta T_i$ and $\mathsf{wp}(n, k) \in \Delta T_j$, for which an abort is *not* preventable. □

5.3 Unproductive Nodes

While not deleting volatile nodes reduces the number of aborting transactions, this slows down query evaluation, thus it is a trade-off. We call non-deleted volatile nodes *unproductive*, as they have to be traversed during query evaluation, but do not contribute to the result set of the query. A characteristic of an unproductive node in an RNI is that neither the node itself nor any of its descendants have a value for a property.

Definition 7 *(*Unproductive Node*) An index node n is* unproductive *in tree G iff no descendant of n has any property:*

$$\forall d\big((d \in G \wedge \mathit{prefix}(n, d)) \Rightarrow \nexists k(d[k] \neq \epsilon)\big) \qquad \square$$

Example 11 *After running T_4 in Example 10, the index node* `/i/p/n/c/d` *and its ancestors are unproductive because they do not have any properties. Nevertheless, during query evaluation, we still traverse these nodes.* □

5.4 Parameterization

Volatility Threshold τ Let us consider two extreme values for τ. With $\tau = \infty$, RNI is identical to a basic PP index as described in Section 3.2, since a node never becomes volatile. With $\tau = 0$, index nodes are never pruned and the performance of queries deteriorates, because many nodes will be unproductive. Additionally, the index will keep growing, meaning we also waste a lot of space. Our goal is to choose threshold τ that best balances the number of path conflicts and query runtime to maximize the throughput.

This tradeoff is workload-dependent; a write-heavy workload calls for small values of τ to reduce the number of aborts, while a read-heavy workload benefits from larger values of τ so that query performance does not suffer too much. In a balanced workload, moderate values of τ are most promising. Nodes in mostly static subtrees with few updates and few conflicts, which constitute the largest part of the index, are pruned and queries perform well. Nodes in dynamic subtrees that are repeatedly inserted and deleted are already retained after a small number of updates, minimizing the number of aborts. We investigate this tradeoff in our experimental evaluation (Section 6).

Sliding Window Length L Parameter L determines how much of the recent workload is used to determine whether a node is volatile. If we set $L = 0$, the sliding window is empty and a node cannot become volatile unless $\tau = 0$. As we increase L, a node is more likely to be classified as volatile, because we consider a larger portion of history H. The sliding window length L is naturally upper-bounded by the time frame that history H covers. For instance, Oak periodically runs a garbage collection to delete old snapshots in H and snapshots are retained for a minimum amount of time (the default is 24 hours). We choose $L = 24$ hours to use all workload information that Oak provides.

6 Experimental Evaluation

Our experimental evaluation considers synthetic and real-world datasets (see Section 6.1) as well as different workloads (see Section 6.2), i.e., read-heavy and write-heavy scenarios. We organize the evaluation as follows:

1. In Section 6.3 we show how to calibrate RNI. We experimentally determine the optimal threshold τ that balances query performance and number of aborts. We also look at the impact of the length of the sliding window.
2. In Section 6.4, we compare RNI to an enhanced basic PP index running the concurrency-control protocol MOCC [WK16] that was specifically designed to reduce the number of aborts. We demonstrate that a basic PP index modified with MOCC still suffers from many path conflicts and show that RNI provides a better throughput.
3. Finally, in Section 6.5, we investigate an approach deferring node deletions to improve concurrency during updates (proposed by Lomet et al. [Lo04, LS97]). However, this only delays the conflicts: the deferred deletions often clash with regular user transactions later on and RNI's performance is still better.

6.1 Preliminaries

We use real-world and synthetic datasets in our experimental evaluation. The real-world dataset is the Dell website[9] and contains 12,244,893 nodes. A node has an average (maximum) depth of 13.68 (24) and an average (maximum) fanout of 2.88 (1729). The synthetic dataset is a binary tree of depth 19 and contains $2^{20}-1 \approx 1$M nodes. Using a binary tree increases the likelihood of path conflicts, so this dataset simulates a kind of worst-case scenario.

[9] `https://dell.com`; Dell uses AEM [Ad23] as CMS and Oak as HDB for its website. The Dell dataset has been extracted from a dump of Oak.

Each experiment was run for five minutes. At the beginning of an experiment, the index is pre-populated with 10% of the nodes from the dataset. The experiments are conducted on virtual machines, each having 8 CPU cores and 32GB of RAM. Unless stated otherwise, we use 8 threads that run concurrent transactions.

6.2 Workloads

We use two types of transactions, *writers* and *readers*, that simulate the publishing of webpages. Each writer picks a set of 50 content nodes, adds a property, and updates the index accordingly. A subsequent writer removes this property from the same content nodes and updates the index. A reader simulates the background process that looks for publishable webpages by executing a PP query against the index. We use three variations of this workload that differ in the ratio between writer and reader transactions (cf. Table 1).

Workload	Abbrev.	Writer:Reader Ratio
Write-Intensive	WI	5:1
Balanced	BA	1:1
Read-Intensive	RI	1:5

Tab. 1: The three considered workloads.

Writers We generate the writer transactions to provoke preventable aborts. We do so by splitting the database G into the same number of partitions as concurrent transactions (or threads) and assigning them to writers. Writers only randomly modify properties of nodes in their partition, i.e., there are no conflicts in the database itself, we can only have path conflicts in the index. The partitioning is done as follows. We assign each node n a unique rank r, $1 \leq r \leq N$, with N being the number of nodes in the tree G. The rank of each node is determined by an inverse level-order traversal of G, i.e., the first leaf has rank 1 and the root has rank N. A node with rank r belongs to partition $p = r \mod P$, where P is the number of partitions, thus each partition contains $\lfloor N/P \rfloor$ nodes. When determining the write set of a write transaction, the j-th node in a partition is picked with a probability of $\mathit{Zipf}(j, \lfloor N/P \rfloor, s_w)$, where s_w is the skew (of the write transactions). The Zipfian distribution $\mathit{Zipf}(j, J, s)$ is equal to $(j^s \sum_{i=1}^{J} \frac{1}{i^s})^{-1}$, where J is the number of elements, j the position of an element ($1 \leq j \leq J$), and s the skew ($s = 0$ being the uniform distribution). The default value of s_w in our experiments is 1.

Readers A reader executes a single PP query $Q = (k, v, /\lambda_1/\ldots/\lambda_d)$. The root of the traversed subtree is randomly chosen among all nodes at a certain depth d (in our experiments we choose $d = 8$). For our synthetic dataset, which is a binary tree with depth 19, a PP query with $d = 8$ traverses a subtree with at most $2^{19-8+1} = 4096$ nodes. Let N_d be the number of nodes at depth d. The j-th node among all nodes at depth d is picked with

probability $Zipf(j, N_d, s_r)$, where parameter s_r is the reader skew. The default value of s_r in our experiments is 1.

6.3 Calibration of RNI and Comparison with Basic PP Index

We begin with the calibration of the threshold τ and show how it affects the tradeoff between contention (expressed as the number of preventable aborts) and the query performance (expressed as the number of nodes read). The results of the experimental evaluation in Figure 5 also serve as a comparison of RNI with a basic PP index. The measurements at the far right-hand side of every diagram ($\tau = 1000$) represent the performance of a basic PP index: all the curves flatten off at that point and continue on the same level for even larger values of τ.

6.3.1 Volatility Threshold τ

Volatile Nodes The first column of diagrams in Figure 5 shows the percentage of index nodes that are volatile, depending on the threshold τ, at the end of an experimental run. Clearly, the smaller τ, the more volatile nodes there are. For $\tau = 1$, between 40% and 70% of all index nodes are volatile for the synthetic dataset, while the numbers are lower for the Dell dataset at around 10% to 20% (this dataset is larger and, thus, each individual node is inserted and deleted less frequently). Also, write intensive workloads have more volatile nodes than read-intensive ones, as they contain more insert and delete operation. With increasing τ the percentage of volatile nodes eventually reaches zero, which is equivalent to a basic PP index: it has no volatile nodes.

Abort Ratio The second column of diagrams in Figure 5 illustrates the impact of τ on the abort ratio of transactions. For small values of τ, we can eliminate almost all preventable aborts, as many nodes become volatile and path conflicts occur rarely. With increasing τ, the abort ratio increases. For write-intensive workloads, the abort ratio reaches 50%, while for read-intensive workloads, this ratio is much lower, at about 10%, since read transactions do not conflict with each other. In summary, this confirms that RNI is able to detect and retain index nodes that are responsible for preventable aborts, which are detrimental to the performance of a basic PP index.

Number of Read Nodes The third column of diagrams in Figure 5 shows the flip side: while a small value of τ reduces the number of aborted transactions, the query performance suffers, as many unproductive volatile nodes have to be traversed. For $\tau = 1$, a PP query visits two-and-a-half to four times as many nodes during query evaluation for the synthetic dataset compared to the number of nodes for a large value of τ. Due to the larger size of the

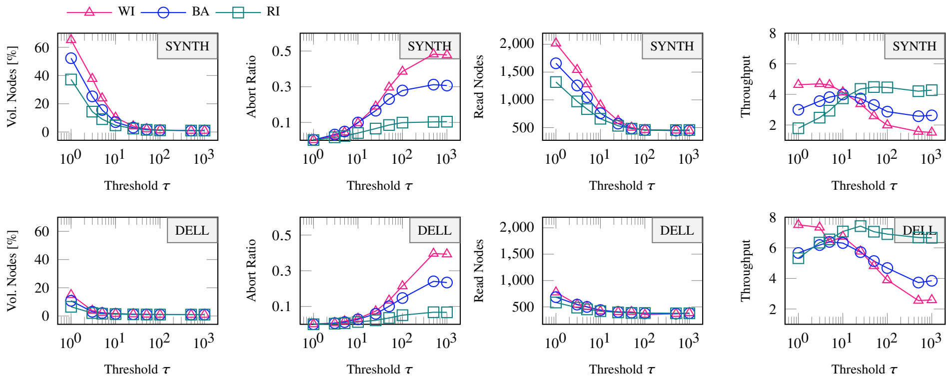


Fig. 5: Threshold τ trades query performance and abort ratio to increase the throughput.

Dell dataset and (therefore) fewer volatile nodes, these numbers are smaller. For increasing τ, the number of read nodes eventually levels off at just under 500, which is the number of nodes that have to be accessed to answer a query in a basic PP index.

Throughput We report normalized values here to make the results comparable to those in Sections 6.4 and 6.5. Normalizing means calculating the ratio between the serial execution of the basic PP index (as a baseline) and the concurrent execution of RNI. The last column of diagrams in Figure 5 shows the results for our experiments on throughput. As already mentioned, RNI trades the reduction of contention against query performance. However, the situation is not quite that simple. For write-intensive workloads, aborts are the major performance bottleneck as opposed to query performance, so $\tau = 1$ yields the best throughput (recall that we run the experiments on an eight-core machine, so a throughput of eight means perfect parallelization). We observe the most pronounced effect for balanced workloads: here values of around 10 for τ feature the best performance, with smaller and larger values showing significantly less performance. For read-intensive workloads, the optimal throughput performance is not as distinctive as for balanced workloads. Moreover, the optimal value for τ is shifted to the right, as query performance plays a more important role. Nevertheless, by using an appropriate value of τ, we can always achieve a better performance with RNI compared to a basic PP index.

6.3.2 Sliding Window Length L

Finally, we look at the impact of the length L of the sliding window. $L = 0$ mirrors the case $\tau = \infty$: in both cases no node can be become volatile, so a number greater than zero has to be chosen to see any kind of effect. As we see in Figure 6, the measured numbers for the throughput stabilize quickly.

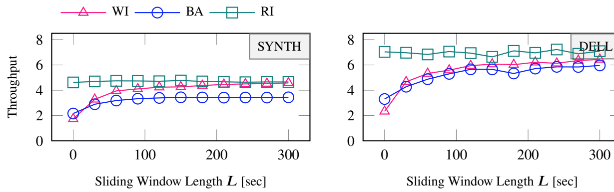


Fig. 6: RPP's throughput is insensitive to L as long as $L > 0$.

6.4 Comparison with MOCC

Next, we investigate whether a basic PP index running the MOCC protocol [WK16], which we call MOCC$_{PP}$, is able to compete with our approach RNI. We ran the MOCC$_{PP}$ experiments in the in-memory system FOEDUS [Ki15], in which MOCC is natively implemented. Table 2 illustrates the differences in terms of (normalized) throughput between MOCC$_{PP}$ and RNI (the best result per dataset and approach is shown in boldface). For RNI, we show two rows with results. The first row (optimized) uses the optimal value of τ for each of the different workloads. In practice, it will be difficult to tune RNI for every individual workload, so the second row ($\tau = 10$) shows the results for a configuration employing a common value of $\tau = 10$ for all the workloads.

workload	SYNTH			DELL		
approach	WI	BA	RI	WI	BA	RI
MOCC$_{PP}$	1.48	3.45	**5.94**	1.63	3.30	**5.80**
RNI (optimized)	**4.69**	4.00	4.48	**7.51**	6.39	7.42
RNI ($\tau = 10$)	**4.15**	4.00	3.74	6.73	6.31	**7.07**

Tab. 2: Comparison of normalized throughput between MOCC$_{PP}$ and RNI.

We make a couple of observations here. The higher the ratio of read transactions, the better MOCC$_{PP}$ performs. This does not come as a surprise: for highly contentious workloads, MOCC runs an optimistic concurrency protocol with a low overhead, i.e., no locks are used, and during a validation phase transactions that are in conflict with other transactions have to abort. In read-intensive workloads, conflicts rarely occur. However, when faced with heavy contention, MOCC switches to a pessimistic lock-based protocol to avoid a large number of transactions to abort during the validation phase. While this does bring down the number of aborted transactions, it introduces an overhead in the form of lock management and in a write-intensive workload with many conflicts, transactions have to wait for the release of locks. In case of a deadlock, we may even have to abort transactions. The performance of RNI is much more balanced across the different workloads: it can handle environments with a lot of write conflicts much better than MOCC$_{PP}$. In summary, there is only one scenario, the read-intensive synthetic workload, for which MOCC$_{PP}$ performs better than RNI. In all other cases, RNI outperforms MOCC$_{PP}$.

6.4.1 MOCC$_{\text{RNI}}$: Combining MOCC with RNI

Since RNI and MOCC use orthogonal principles, we can combine the two to obtain an even better approach by running MOCC with volatile nodes. We call this protocol MOCC$_{\text{RNI}}$. Figure 7 shows the results for calibrating the parameter τ for MOCC$_{\text{RNI}}$. Comparing these results to the last two diagrams in the bottom row of Figure 5 (illustrating the tuning of RNI), we see that the performance of MOCC$_{\text{RNI}}$ is worse than that of RNI for write-intensive and balanced workloads (i.e., workloads with a higher proportion of write transactions). In these cases, the advantage of using a small value of τ is offset by the overhead of using a pessimistic locking protocol. Consequently, we should never use small values of τ for MOCC$_{\text{RNI}}$.

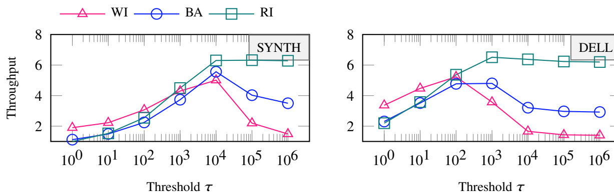


Fig. 7: Optimal thresholds τ for MOCC$_{\text{RNI}}$.

6.4.2 Comparing MOCC$_{\text{RNI}}$ with MOCC$_{\text{PP}}$

We now take a closer look at the performance of MOCC$_{\text{RNI}}$ versus that of MOCC$_{\text{PP}}$. As combining MOCC with volatile nodes aims at improving the performance of MOCC for write-heavy scenarios, we focus on the WI and BA workloads.

First, we investigate how well MOCC$_{\text{PP}}$ and MOCC$_{\text{RNI}}$ handle skewed workloads with hotspots, i.e., nodes that are accessed very frequently. The first two columns of Figure 8 depict the results for varying the skewedness (determined by the parameter s of the Zipfian distribution). In the first column, we alter the writer skew s_w, in the second column the reader skew s_r. We can see clearly, that MOCC$_{\text{PP}}$ cannot cope with high writer skew at all. As soon as s_w increases beyond 0.5, the performance of MOCC$_{\text{PP}}$ deteriorates drastically. Due to the high contention, MOCC$_{\text{PP}}$ switches to a pessimistic lock-based protocol. This keeps transactions from aborting, but introduces waiting times for the release of locks, because a lot of transactions want to access the same data items in a skewed workload. The only case for which MOCC$_{\text{PP}}$ performs better is a uniformly distributed workload on the synthetic dataset. However, this case is the least relevant one in practice: real-world workloads are rarely uniformly distributed. The picture changes, when we look at the reader skew s_r (second column of Figure 8). MOCC$_{\text{RNI}}$'s performance degrades slightly for a higher reader skew, as the skewed read operations traverse unproductive nodes more often. MOCC$_{\text{PP}}$, on the other hand, shows constant performance for the synthetic dataset and even profits a bit for the Dell dataset. Nevertheless, MOCC$_{\text{RNI}}$ maintains an edge over MOCC$_{\text{PP}}$.

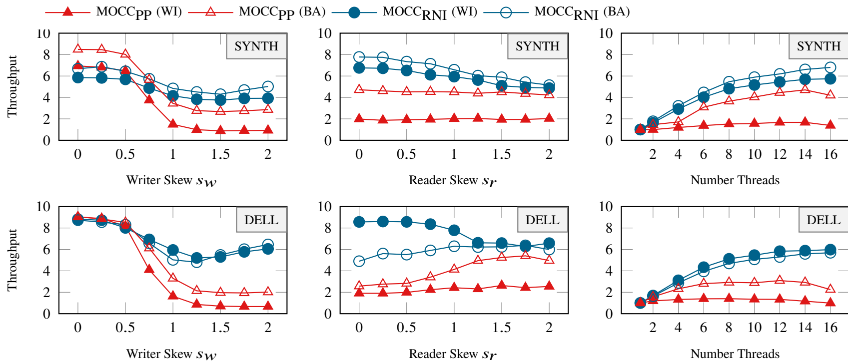


Fig. 8: MOCC_{RNI} has a higher throughput than MOCC_{PP}.

Second, we illustrate how MOCC_{RNI} and MOCC_{PP} compare for different degrees of concurrency (third column in Figure 8, s_w and s_r are set to the default value of 1). We increase the number of transactions that are running concurrently to see how well the two approaches can adapt to higher levels of concurrency. MOCC_{RNI} scales much better, since it avoids many path conflicts from the outset with the use of volatile nodes.

6.5 Comparison with Deferred Node Deletions

In the next set of experiments, we compare RNI with an approach that defers node deletions as proposed by Lomet et al. [Lo04, LS97]. We implement deferred node deletions in PP as follows. When a user transaction attempts to delete an index node, the indexed property is removed (so that query results are correct) but the node deletion is deferred and the node is added to a queue. A background process periodically polls this queue and attempts to batch-prune the queued index nodes. If a background transaction fails due to a conflict, the index nodes are re-enqueued. We call this approach DeferredPP. Table 3 shows a comparison of the (normalized) throughput of DeferredPP with RNI. We conducted these experiments in Oak.

workload	SYNTH			DELL		
approach	WI	BA	RI	WI	BA	RI
DeferredPP	**2.90**	2.44	1.70	**4.62**	4.41	4.13
RNI (optimized)	**4.69**	4.00	4.48	**7.51**	6.39	7.42
RNI (τ = 10)	**4.15**	4.00	3.74	6.73	6.31	**7.07**

Tab. 3: Comparison of normalized throughput between deferred node deletion and RNI.

The first interesting observation is that DeferredPP's performance goes down with an increasing ratio of read transactions. For read-intensive workloads, deferring the deletions comes with a drawback. Essentially, the nodes scheduled for deletion are unproductive

nodes that have to be traversed by queries, driving down the query performance. The more read transactions we have, the more pronounced this effect is. More generally, when pruning a batch of nodes in background transactions, these transactions can clash with other transactions running in the system. While we always roll back a background transaction in a conflict (i.e., the regular transactions have precedence), this still consumes system resources and further reduces the throughput. Thus, DeferredPP is worse than RNI for all workloads. A scenario for which DeferredPP could potentially work is a system with write-intensive workloads that experiences phases of calm with a light load, e.g. during the night, in which the pruning takes place with a low probability of causing conflicts.

7 Conclusion

We investigated a problem that property-and-path (PP) indexes are faced with in hierarchical databases: the occurrence of path conflicts in the index when nodes with the same property (on different paths but with common ancestors in the database) are concurrently inserted and deleted. While the operations in the database go ahead without any issues, due to the propagation of deletes to ancestor nodes in the index, this causes a conflict and aborts the whole transaction. However, these aborts are preventable by leaving volatile nodes, i.e., nodes that are frequently inserted and deleted, in the index.

We propose the robust node index (RNI) that detects volatile nodes and prevents path conflicts due to the propagation of deletes. However, leaving volatile nodes in the index has a cost attached to it. The index becomes larger than it has to be and traversing additional, unproductive nodes during query evaluation has a negative impact on the performance. We experimentally evaluated the tradeoff between reducing the number of aborts and increasing query execution time and show how to tune RNI to maximize the throughput. This is done by only keeping volatile nodes in the index if their volatility is above a threshold τ, i.e., if a node is inserted and deleted more than τ times during a certain timeframe. Comparisons with other approaches, such as MOCC [WK16] and deferred delete [Lo04, LS97], confirm that RNI is able to significantly reduce the abort ratio from around 50% to below 10% for write-heavy workloads, thereby increasing the throughput up to a factor of five.

Bibliography

[Ad23] Adobe: , Adobe Experience Manager. https://www.adobe.com/marketing-cloud/experience-manager.html, 2023. [Online; accessed January 2023].

[Ap17] Appuswamy, Raja; Anadiotis, Angelos; Porobic, Danica; Iman, Mustafa; Ailamaki, Anastasia: Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. PVLDB, 11(2):121–134, 2017.

[Ap22] Apache: , Apache Jackrabbit Oak. https://jackrabbit.apache.org/oak/, 2022. [Online; accessed January 2023, last updated November 2022].

[Be95] Berenson, Hal; Bernstein, Philip A.; Gray, Jim; Melton, Jim; O'Neil, Elizabeth J.; O'Neil, Patrick E.: A Critique of ANSI SQL Isolation Levels. In: SIGMOD. 1995.

[Be11] Bernstein, Philip A.; Reid, Colin W.; Wu, Ming; Yuan, Xinhao: Optimistic Concurrency Control by Melding Trees. PVLDB, 4(11), 2011.

[Be15] Bernstein, Philip A.; Das, Sudipto; Ding, Bailu; Pilman, Markus: Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. SIGMOD, 2015.

[CH10] Cormode, Graham; Hadjieleftheriou, Marios: Methods for finding frequent items in data streams. The VLDB Journal, 19(1):3–20, 2010.

[CMS02] Chung, Chin-Wan; Min, Jun-Ki; Shim, Kyuseok: APEX: An adaptive path index for XML data. In: SIGMOD. ACM, pp. 121–132, 2002.

[Co01] Cooper, Brian F.; Sample, Neal; Franklin, Michael J.; Hjaltason, Gísli R.; Shadmon, Moshe: A Fast Index for Semistructured Data. In: VLDB. 2001.

[DR13] Diegues, Nuno Lourenco; Romano, Paolo: Bumper: Sheltering Transactions from Conflicts. In: IEEE SRDS. 2013.

[Fi02] Fiebig, Thorsten; Helmer, Sven; Kanne, Carl-Christian; Moerkotte, Guido; Neumann, Julia; Schiele, Robert; Westmann, Till: Anatomy of a native XML base management system. VLDB J., 11(4):292–314, 2002.

[Fi13] Finis, Jan; Brunel, Robert; Kemper, Alfons; Neumann, Thomas; Färber, Franz; May, Norman: DeltaNI: An Efficient Labeling Scheme for Versioned Hierarchical Data. In: SIGMOD. pp. 905–916, 2013.

[Gr14] Graefe, Goetz; Halim, Felix; Idreos, Stratos; Kuno, Harumi A.; Manegold, Stefan; Seeger, Bernhard: Transactional support for adaptive indexing. VLDB J., 23(2), 2014.

[GW97] Goldman, Roy; Widom, Jennifer: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB. 1997.

[Ha17] Harding, Rachael; Van Aken, Dana; Pavlo, Andrew; Stonebraker, Michael: An Evaluation of Distributed Concurrency Control. PVLDB, 2017.

[HHL06] Haustein, Michael Peter; Härder, Theo; Luttenberger, Konstantin: Contest of XML Lock Protocols. In: VLDB. 2006.

[HL11] Haw, Su-Cheng; Lee, Chien-Sing: Data storage practices and query processing in XML databases: A survey. Knowledge-Based Systems, 24(8):1317–1340, 2011.

[Id11] Idreos, Stratos; Manegold, Stefan; Kuno, Harumi A.; Graefe, Goetz: Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. PVLDB, 4(9):585–597, 2011.

[JPA09] Johnson, Ryan; Pandis, Ippokratis; Ailamaki, Anastasia: Improving OLTP Scalability using Speculative Lock Inheritance. PVLDB, 2(1):479–489, 2009.

[Ki15] Kimura, Hideaki: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In: SIGMOD. 2015.

[Lo04] Lomet, David B.: Simple, Robust and Highly Concurrent B-trees with Node Deletion. In: ICDE. pp. 18–27, 2004.

[Lo15] Loro, Alessandra; Gruenheid, Anja; Kossmann, Donald; Profeta, Damien; Beaudequin, Philippe: Indexing and Selecting Hierarchical Business Logic. PVLDB, 8(12):1656–1667, 2015.

[LS97] Lomet, David B.; Salzberg, Betty: Concurrency and Recovery for Index Trees. VLDB J., 6(3):224–240, 1997.

[LY81] Lehman, Philip L.; Yao, s. Bing: Efficient Locking for Concurrent Operations on B-trees. ACM TODS., 6(4):650–670, December 1981.

[Ma15] Mathis, Christian; Härder, Theo; Schmidt, Karsten; Bächle, Sebastian: XML indexing and storage: fulfilling the wish list. Computer Science - R&D, 30(1), 2015.

[Ni06] Nicolaisen, Thomas Ferris: The Use of Open Source and Open Standards in Web Content Management Systems. Master's thesis, University of Oslo, Oslo, Norway, May 2006.

[RFA16] Ren, Kun; Faleiro, Jose M.; Abadi, Daniel J.: Design Principles for Scaling Multi-core OLTP Under High Contention. In: SIGMOD. 2016.

[RTA14] Ren, Kun; Thomson, Alexander; Abadi, Daniel J.: An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. PVLDB, 2014.

[Sh15] Shukla, Dharma; Thota, Shireesh; Raman, Karthik; Gajendran, Madhan; Shah, Ankur; Ziuzin, Sergii; Sundaram, Krishnan; Guajardo, Miguel Gonzalez; Wawrzyniak, Anna; Boshra, Samer; Ferreira, Renato; Nassar, Mohamed; Koltachev, Michael; Huang, Ji; Sengupta, Sudipta; Levandoski, Justin J.; Lomet, David B.: Schema-Agnostic Indexing with Azure DocumentDB. PVLDB, 2015.

[TA10] Thomson, Alexander; Abadi, Daniel J.: The Case for Determinism in Database Systems. PVLDB, 2010.

[Th12] Thomson, Alexander; Diamond, Thaddeus; Weng, Shu-Chun; Ren, Kun; Shao, Philip; Abadi, Daniel J.: Calvin: Fast Distributed Transactions for Partitioned Database Systems. In: SIGMOD. 2012.

[Ti18] Tian, Boyu; Huang, Jiamin; Mozafari, Barzan; Schoenebeck, Grant: Contention-Aware Lock Scheduling for Transactional Databases. PVLDB, 2018.

[TYJ09] Tzoumas, Kostas; Yiu, Man Lung; Jensen, Christian S.: Workload-Aware Indexing of Continuously Moving Objects. PVLDB, 2009.

[WBH20] Wellenzohn, Kevin; Böhlen, Michael H.; Helmer, Sven: Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data. Proc. VLDB Endow., 13(10):1641–1653, 2020.

[WK16] Wang, Tianzheng; Kimura, Hideaki: Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. PVLDB, 10(2):49–60, 2016.

[YC16] Yan, Cong; Cheung, Alvin: Leveraging Lock Contention to Improve OLTP Application Performance. PVLDB, 9(5):444–455, 2016.

[Yu16] Yuan, Yuan; Wang, Kaibo; Lee, Rubao; Ding, Xiaoning; Xing, Jing; Blanas, Spyros; Zhang, Xiaodong: BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases. PVLDB, 9(6):504–515, 2016.