Indexing Highly Dynamic Hierarchical Data

Jan Finis* Robert Brunel*
Alfons Kemper* Thomas Neumann* Norman May† Franz Faerber†
* Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
† SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
* *firstname.lastname*@cs.tum.edu † *firstname.lastname*@sap.com

ABSTRACT

Maintaining and querying hierarchical data in a relational database system is an important task in many business applications. This task is especially challenging when considering dynamic use cases with a high rate of complex, possibly skewed structural updates. Labeling schemes are widely considered *the* indexing technique of choice for hierarchical data, and many different schemes have been proposed. However, they cannot handle dynamic use cases well due to various problems which we investigate in this paper. We therefore propose our dynamic *Order Indexes*, which offer competitive query performance, unprecedented update efficiency, and robustness for highly dynamic workloads.

1. INTRODUCTION

Hierarchical data has always been ubiquitous in business and engineering applications, especially with the advent of the inherently hierarchical XML data format. Relational database systems (RDBMS) continue to be the predominant platform on which such applications are built. These facts have inevitably and repeatedly led to the challenge of representing hierarchical data in relational tables, or more specifically, encoding the structure of a hierarchy in a table such that a table row represents a hierarchy node. We revisit the challenge of finding a representation that provides decent query capabilities without sacrificing update performance. This classic trade-off strikes particularly hard with hierarchical data and vast amounts of papers have been written on working around it. Unlike many of those works, we place our focus on update performance. In SAP's application scenarios we encounter fine-grained, complex updates—in particular relocations of large subtrees—and at the same time need to provide a certain set of primitive query operations where we cannot tolerate significant performance losses. After thorough searching, we have come to the conclusion that a robust, efficient solution is still missing to date.

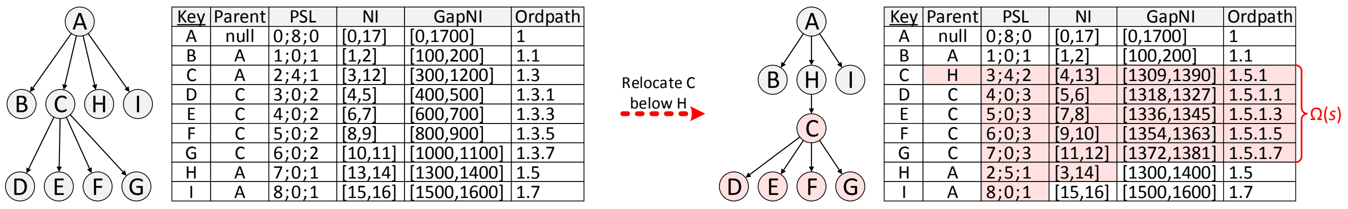
In this paper we use *indexing scheme* as a collective term for any technique for representing a hierarchy with the aim of providing an acceptable tradeoff between query and update performance. Two major classes of schemes exist: *Labeling schemes* [1, 2, 4, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 23], on one hand, attach a *label* to each node and answer queries by considering only these labels. The labels are maintained in one or more table columns, and those columns are in practice usually augmented by one or more general-purpose database indexes, such as B-trees. *Index-based schemes* [7, 19], on the other hand, enhance the RDBMS by special-purpose index structures.

Especially labeling schemes are backed by a massive body of research on XML databases, on techniques for storing XML fragments in an RDBMS backend, and on evaluating query languages such as XPath and XQuery. However, in this paper we make a case for index-based schemes. Our first contribution is an analysis of query versus update considerations (Sec. 2) and a taxonomy of existing indexing schemes in that light (Sec. 3). It leads us to conclude that previously proposed dynamic labeling schemes are unable to fulfill desired properties: they either ignore important query primitives, or they inherently suffer from certain problems inhibiting their update flexibility or robustness. Sophisticated index-based schemes can help us overcome the inherent problems of plain labeling schemes and support highly dynamic use cases efficiently. Our main contribution is a family of index-based schemes called *Order Indexes* designed to achieve this (Sec. 4). We propose three specific implementations, the AO-Tree, the BO-Tree, and the O-List, and discuss performance optimization techniques for each. We conduct a number of experiments (Sec. 5) to assess the merits and drawbacks of the implementation variants, and we show that they support complex updates efficiently, avoid degeneration in case of unfavorable update patterns, and at the same time provide the query capabilities of common labeling schemes with highly competitive performance. We recently proposed a SQL language extension [3] for seamless handling of hierarchical data in SAP HANA and apply Order Indexes as a promising back-end for it. However, they are also applicable to non-relational systems such as XML databases.

2. HANDLING DYNAMIC HIERARCHIES

The considerations for representing hierarchical data in relational systems differ somewhat from special-purpose systems. For example, XML indexing techniques commonly rely on clustering data by structure, e. g., arranging it in pre-order. In business applications such as SAP ERP, which we analyzed in [3], a hierarchy is rarely the primary dimension of a table. Therefore, clustering by hierarchy structure is usually

Proceedings of the VLDB Endowment, Vol. 8, No. 10
Copyright 2015 VLDB Endowment 2150-8097/15/06.



Key	Parent	PSL	NI	GapNI	Ordpath
A	null	0;8;0	[0,17]	[0,1700]	1
B	A	1;0;1	[1,2]	[100,200]	1.1
C	A	2;4;1	[3,12]	[300,1200]	1.3
D	C	3;0;2	[4,5]	[400,500]	1.3.1
E	C	4;0;2	[6,7]	[600,700]	1.3.3
F	C	5;0;2	[8,9]	[800,900]	1.3.5
G	C	6;0;2	[10,11]	[1000,1100]	1.3.7
H	A	7;0;1	[13,14]	[1300,1400]	1.5
I	A	8;0;1	[15,16]	[1500,1600]	1.7

Key	Parent	PSL	NI	GapNI	Ordpath
A	null	0;8;0	[0,17]	[0,1700]	1
B	A	1;0;1	[1,2]	[100,200]	1.1
C	H	3;4;2	[4,13]	[1309,1390]	1.5.1
D	C	4;0;3	[5,6]	[1318,1327]	1.5.1.1
E	C	5;0;3	[7,8]	[1336,1345]	1.5.1.3
F	C	6;0;3	[9,10]	[1354,1363]	1.5.1.5
G	C	7;0;3	[11,12]	[1372,1381]	1.5.1.7
H	A	2;5;1	[3,14]	[1300,1400]	1.5
I	A	8;0;1	[15,16]	[1500,1600]	1.7

Figure 1: Various labeling schemes (left); labels that need to be changed when subtree C is relocated (right)

infeasible or not preferable; in other words, hierarchy indexes are generally *secondary* indexes. We employ the model from [3] representing a hierarchy by a *node* column plus a secondary index structure. The column contains the labels of the indexing scheme; the index structure indexes the values in that column. Depending on the use case, the order among siblings may be meaningful, so a hierarchy can either be ordered (e. g., in XML) or unordered. Even in the unordered case, indexing schemes usually impose an internal storage order and treat both ordered and unordered hierarchies equally. Therefore, we assume ordered hierarchies hereinafter, but note that our findings apply to both settings.

In Sec. 1 we claim that previously proposed indexing schemes lack certain important capabilities in the light of highly dynamic use cases. The three problems we identify in this section characterize these missing capabilities. The hierarchy from Fig. 1 helps us exemplify the problems; it also illustrates several indexing schemes, which we explain later.

P1 *Lack of Query Capabilities.* Certain indexing schemes do support updates decently, but fail to offer query capabilities to answer even fundamental queries, which renders them infeasible for our use cases. An example is the adjacency list model, a common way of naïvely encoding a hierarchy in SQL by storing the primary key of the parent node (Parent in Fig. 1): it cannot even handle the ancestor-descendant relationship efficiently. In Sec. 2.1 we identify fundamental query primitives to be supported sine qua non.

P2 *Lack of Complex Update Capabilities.* Various ERP use cases demand for an indexing scheme that supports a rich set of update operations efficiently. However, most existing schemes are confined to *leaf* updates, that is, insertion or deletion of single leaf nodes, and fail to recognize more complex operations. Consider *subtree* relocation, where a subtree of a certain size s rooted in a specific node is moved in bulk to another location within the hierarchy. Ironically, the trivial adjacency list model naturally supports this. However, virtually all labeling schemes by design preclude an efficient implementation, because they inherently require relabeling *all* nodes in the relocated subtree at a cost of $\Omega(s)$. The right side of Fig. 1 shows the hierarchy from the left with the subtree rooted in C moved below H, and highlights the $\Omega(s)$ fields that need to be updated. In Sec. 2.2 we explore further conceivable complex update operations.

P3 *Vulnerability to Skewed Updates.* Certain dynamic labeling schemes crumble when confronted with skewed updates, such as when inserts are issued repeatedly at the same position. In some scenarios these updates are more frequent than is commonly acknowledged. For example, when inserting a new plant into an enterprise asset hierarchy, many nodes will be added at a position. Fixed-length labeling schemes commonly indulge in excessive relabeling in this case, while variable-length schemes decay in their query performance and memory effectiveness due to overly growing labels.

BINARY PREDICATES	
is_descendant(a, b)	whether a is a descendant of b
is_child(a, b)	whether a is a child of b
is_before_pre(a, b)	whether a precedes b in a pre-order traversal
is_before_post(a, b)	whether a precedes b in a post-order traversal
NODE PROPERTIES	
level(a)	the number of edges on the path from a root to a
is_root(a)	whether a is a root node
is_leaf(a)	whether a is a leaf node, i. e., has no children
TRAVERSAL	(c is a cursor of the secondary index structure)
find(a)	a cursor c to node a
next_pre(c)	a cursor to the next node in pre-order
next_post(c)	a cursor to the next node in post-order
next_sibling(c)	a cursor to the next sibling

Figure 2: Essential query primitives on hierarchies

2.1 Query Capabilities

In [3] we explore end-user queries on hierarchies that commonly appear in business scenarios. Query primitives are the building blocks for such high-level queries. Fig. 2 shows an essential set of primitives that are needed to answer fundamental queries. An index that fails to support these primitives cannot be considered a general-purpose hierarchy index. We distinguish between three kinds of primitives. *Binary predicates* compare two nodes. They are often used for self-joining the hierarchy to navigate along certain axes. is_descendant and is_child are of utmost importance, as most queries navigate along these axes. is_before_pre and is_before_post are useful for ordering nodes in a depth-first, either parent-before-child (e.g., document order in XML) or child-before-parent manner. *Node properties* are used to filter nodes, for example, when the user wishes to restrict the result to leaf nodes or to nodes at certain levels. An example is the so-called explosion query often found in ERP applications, which consists of finding all descendants of a node up to a certain level. *Traversal* operations are useful to implement an index-nested-loop join or an index scan. Such an operation commonly starts with a find, that is, the position of a node in the index structure is looked up. From there, the traversal functions are used to scan the index in various directions. The following SQL code uses the language extensions from [3] on a table H exposing a node column named pos. It shows a simple, meaningful query making use of all three kinds of primitives:

```
SELECT S.pos, R.pos, LEVEL(R.pos)
  FROM H AS S JOIN H AS R ON IS_DESCENDANT(R.pos, S.pos)
```

The query features a hierarchy join over the descendant axis and computes the node level. It can be answered efficiently using an index-nested-loop join. Fig. 3 shows the pseudo code in terms of the query primitives.

```
for all $s \in S$ do
    $r \leftarrow$ next_pre(find($s$))
    while is_descendant($r, s$) do
        emit ($s, r$, level($r$))
        $r \leftarrow$ next_pre($r$)
```

Figure 3: A hierarchy index-nested-loop join

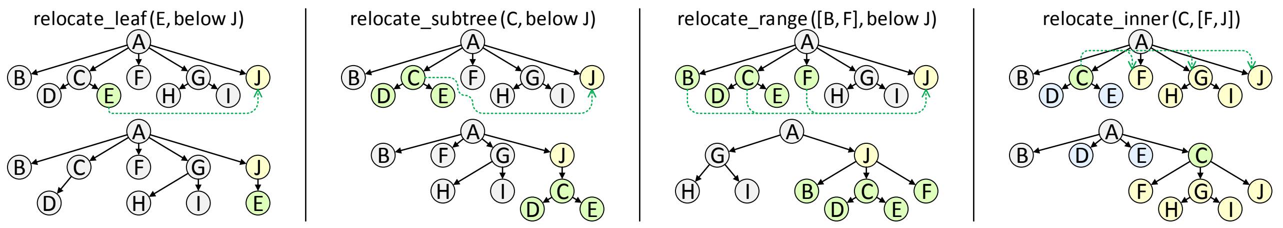

relocate_leaf (E, below J)
relocate_subtree (C, below J)
relocate_range ([B, F], below J)
relocate_inner (C, [F, J])

Figure 4: Various classes of relocation updates on an example hierarchy: before (top) and after (bottom)

BULK UPDATES	
bulk_build(T)	bulk-builds the hierarchy from a tree representation T
LEAF UPDATES	alter a single leaf node
delete_leaf(a)	deletes a leaf node a
insert_leaf(a, p)	inserts the new leaf node a at position p
relocate_leaf(a, p)	relocates a leaf node a to position p
SUBTREE UPDATES	alter a subtree
delete_subtree(a)	deletes the subtree rooted in a
insert_subtree(a, p)	inserts the new subtree rooted in a at position p
relocate_subtree(a, p)	relocates the subtree rooted in a to position p
RANGE UPDATES	alter subtrees rooted in a range of siblings
delete_range($[a, b]$)	deletes all subtrees rooted in range $[a, b]$
insert_range($[a, b], p$)	inserts all subtrees rooted in range $[a, b]$ at position p
relocate_range($[a, b], p$)	relocates all subtrees rooted in range $[a, b]$ to position p
INNER NODE UPDATES	alter an inner node
delete_inner(a)	deletes node a from the hierarchy; the former children of a become children of a's parent
insert_inner($a, [b, c]$)	inserts the new node a as child of the parent of b and c; nodes in range $[b, c]$ become children of a
relocate_inner($a, [b, c]$)	makes all children of a children of a's parent; a becomes the parent of all nodes in range $[b, c]$, and the child of the previous parent of b and c

Figure 5: Update operations on hierarchies

For each tuple s from the left input S we obtain a cursor to s in the index. Then, we iterate over the index in pre-order using next_pre as long as there are further descendants of s.

2.2 Update Capabilities

In Fig. 5 we present a taxonomy of update operations that a dynamic indexing scheme shall support. p indicates a target position in the hierarchy. Depending on whether the sibling order is meaningful, p can have different values, such as "as first child of node x", "as direct left sibling of x", "below x", or "as a sibling of x". How exactly p is represented is not important here. The syntax $[x, y]$ denotes a *sibling range* of nodes: y must be a right sibling of x (or x itself), and $[x, y]$ refers to all siblings between and including x and y.

The first class of updates is bulk-building a hierarchy, using for example the bulk build operator from [3]. Fortunately, all indexing schemes—even static ones—support this efficiently. The other classes are *leaf* node updates, *subtree* updates, sibling *range* updates, and *inner* node updates, each named after the entities involved in the update. Within each class, three update kinds are conceivable: *delete* updates, which delete existing nodes, *insert* updates, which insert new nodes, and *relocate* updates, which alter the positions of existing nodes. Fig. 4 illustrates the various classes with regard to the relocate kind on an example hierarchy. The other kinds, insert and delete, are similar; the only difference is that the entities being updated (green in the figure) enter or leave the hierarchy, respectively, instead of being relocated. In a sense, the relocate kind subsumes the other two: inserts and deletes are relocations into and out of the hierarchy, respectively. Thus, any index that handles relocation efficiently supports efficient insertion and deletion as well.

Most related works consider only *leaf* updates, which are most common in XML. In addition, they are the simplest update class and implementing them efficiently is rather easy in comparison to the other classes of updates.

Our focus is particularly on *subtree* updates. As a leaf node is also a trivial subtree, they subsume the corresponding leaf updates. But since indexing schemes usually afford optimized operations for leaves, distinguishing between leaf and subtree operations is useful in practice. Most indexing schemes implement subtree operations naïvely through node-by-node processing, requiring at least s leaf updates for a subtree of size s. For small subtrees, $\Omega(s)$ update cost might be negligible. However, real-world hierarchies—such as the hierarchies found in SAP's ERP applications—have a large average fan-out. Thus, even if a node that has only leaves as children is relocated, s will often be in the magnitude of thousands. Of course, the fact that updating larger subtrees is a slow operation for labeling schemes will be detrimental to overall system performance only if a lot of such operations appear in the workload. However, in many ERP use cases, a high percentage of updates are indeed subtree relocations. For example, in [7] we found that 31% of all updates performed on an enterprise asset hierarchy are subtree relocations. Furthermore, note that although subtree relocation may appear as an unnatural bulk operation in comparison to single leaf insertion or deletion, the operation is quite fundamental to SQL users: In the adjacency list model—the predominant format for representing hierarchies within an RDBMS—*any* change to a parent field corresponds to a subtree relocation. For example, we achieve the relocation in Fig. 1 by simply setting the Parent of C to H in the table. In other words, every issued `UPDATE` statement touching the parent column incurs a certain number of subtree relocations.

The *inner* node updates are useful for inserting a new level into the hierarchy, and for wrapping a subtree into a new root. As an example application, certain tree differentiation algorithms such as MH-Diff [5] emit edit scripts featuring these operations. Therefore, a hierarchy index that is being used for replaying such edit scripts has to support them.

The sibling *range* update might seem obscure at first sight, but is in fact very powerful: It subsumes subtree and leaf updates, because a subtree rooted in a or a leaf a are trivial sibling ranges $[a, a]$. It also subsumes inner node updates, because moving all children of an inner node to another position makes this node a leaf, so a simple leaf update can delete or relocate it. Thus, range updates subsume *all* other update operations, and indexes that supports them—specifically, our Order Indexes—can implement all mentioned operations in terms of range relocation.

3. RELATED INDEXING SCHEMES

In the following we explore existing indexing schemes and assess to which extent they suffer from one of the problems ***P1*** to ***P3*** we identify in Sec. 2. It turns out that most contributions are variations of basic schemes with similar

Scheme		Update Operations				
		leaf	subtree	inner node	range	skew[u]
naïve	Adjacency	🟩 1	🟩 1	🟨 c	🟨 c	🟩 1
	Linked	🟩 1	🟩 1	🟨 c	🟨 c	🟩 1
containment	NI	🟥 n	🟥 n	🟥 n	🟥 n	🟥 n
	Dyn-NI	🟩 1	🟥 s	🟩 1	🟥 s	🟥 u
	Dyn-NI-Parent	🟩 1	🟥 s	🟨 c	🟥 s	🟥 u
	Dyn-NI-Level	🟩 1	🟥 s	🟥 s	🟥 s	🟥 u
path	Dewey	🟥 lf	🟥 $l(f+s)$	🟥 $l(f+s)$	🟥 $l(f+s)$	🟥 $l(f+s)$
	Dyn-Dewey	🟩 l	🟥 ls	🟥 ls	🟥 ls	🟥 $l+u$
index	AO-Tree	🟩 1	🟩 $\log n$	🟩 $\log n$	🟩 $\log n$	🟩 1
	BO-Tree[B]	🟩 1	🟩 $B \log_B n$	🟩 $B \log_B n$	🟩 $B \log_B n$	🟩 1
	O-List[B]	🟩 1	🟨 $s/B+B$	🟨 $s/B+B$	🟨 $s/B+B$	🟨 u/B^2
	DeltaNI	🟩 $\log u$	🟩 $\log u$	🟩 $\log u$	🟩 $\log u$	🟩 $\log u$

n: hierarchy size, l: level of node, c: number of children
s: number of descendants, u: number of updates, B: block size
f: number of following siblings plus their descendants
Qualitative rating: 🟩 efficient 🟨 mostly efficient 🟥 inefficient

Figure 6: Asymptotic update complexities for various indexing schemes (amortized, average case)

capabilities and asymptotic properties, so we can group them into a small taxonomy. Fig. 6 shows the asymptotic amortized update complexities of all groups of schemes in our taxonomy. The columns represent the different classes of updates. Column skew[u] represents skewed leaf node insertions; it depicts the complexity of a single skewed insertion after u other skewed insertions have taken place and thus expresses how skew-resilient an indexing scheme is. The figure includes two naïve representations. They are pragmatic, easy-to-implement solutions, which, however, do not provide the efficient query capabilities of indexing schemes (***P1***). First, the Adjacency list introduced earlier. Second, Linked, a simple in-memory tree representation whose structure matches the hierarchy structure and which uses per-node pointers to the parent, the first and last child, and the previous and next sibling. Besides these two, there are three major categories of indexing schemes: *containment-based* labeling schemes, *path-based* labeling schemes, and *index-based* schemes. Note that labeling schemes are usually indexed by a B-tree and thus an additional $\log n$ factor applies (not displayed in Fig. 6).

Containment-based Labeling Schemes, also referred to as *order-based* or *nested intervals* schemes, label each node with a [lower, upper] interval or a similar combination of values. As the term "nested" alludes to, their main property is that a node's interval is nested in the interval of its parent node. Queries can be answered by testing the intervals of the involved nodes for containment relationships.

Column NI in Fig. 1 shows a nested intervals labeling that is commonly used in XML and other database applications, e. g. [23, 9, 11]. We can see that node E is a descendant of node A, because E's interval $[6, 7]$ is a proper subinterval of A's interval $[0, 17]$. A variation is the pre/post scheme [8], where each node is labeled with its pre- and post-order ranks. Plain NI and pre/post have similar, limited query capabilities: For example, we cannot test the important is_child predicate, because neither scheme allows us to compute the distance between a node and an ancestor. This severe limitation makes a nested intervals scheme without further fields useless (***P1***). It can be mitigated by either storing the level of a node or its parent in addition to the interval. Considering updates, the mentioned schemes are *static* (***P2***). Their fundamental problem is that each insertion or deletion requires relabeling $\mathcal{O}(n)$ labels on average, as all interval bounds behind a newly inserted bound have to be shifted to make space. This group of static nested interval schemes is called NI in Fig. 6.

Various mitigations for the nested intervals update problem have been proposed (Dyn-NI in Fig. 6). Li et al. [15] suggest pre-allocating *gaps* between the interval bounds. Column GapNI in Fig. 1 illustrates this. As long as a gap exists, new bounds can be placed in it and no other bounds need to be shifted; as soon as a gap between two nodes is filled up, *all* bounds are relabeled with equally spaced values. The caveats are that relabelings are expensive, and skewed insertions may fill up certain gaps overly quickly and lead to unexpectedly frequent relabelings (***P3***). In addition, all s nodes in a range or subtree being updated still need to be relabeled (***P2***).

Amagasa et al. [1] propose the QRS encoding based on pairs of floating-point numbers. Schemes along these lines are essentially gap-based as long as they rely on fixed-width machine representations of floats. In [2], Boncz et al. tackle the update problem using their pre/size/level encoding (PSL, cf. Fig. 1) by storing the pre values implicitly as a page offset, which yields update characteristics comparable to gap-based schemes. W-BOX [19] uses gaps but tries to relabel only locally using a weight-balanced B-tree; its skewed update performance is therefore superior to basic gap-based schemes. The Nested Tree [22] uses a nested series of nested interval schemes to relabel only parts of the hierarchy during an update and is therefore comparable to gap-based schemes.

Another idea to tackle the update problem for NI is to use variable-length data types to represent interval bounds: For example, the QED [12], CDBS [13], and CDQS [14] encodings by Li et al. are always able to derive a new label between two existing ones, and thus avoid relabeling completely. EXCEL [16] uses an encoding comparable to CDBS. It tracks the lower value of the parent for enhanced query capabilities. While these encodings never have to relabel nodes, they bear other problems: The variable-length labels cannot be stored easily in a fixed-size relational column and comparing them is more expensive than comparing fixed-size integers. In addition, labels can degenerate and become overly big due to skewed insertion (***P3***). Cohen et al. [6] have proved that for any labeling scheme that is not allowed to relabel existing labels upon insertion, an insertion sequence of length n exists that yields labels of size $\Omega(n)$. Thus, the cost of relabeling is traded in for a larger (potentially unbounded) label size.

All gap-based and variable-length NI schemes can handle *inner node* updates decently by simply wrapping a node range into new bounds. For example, GapNI in Fig. 1 is able to insert a parent node K above D, E, and F by assigning it the bounds $[350, 950]$. However, as soon as the node level [2] or its parent [16] (Dyn-NI-Parent and Dyn-NI-Level in Fig. 6) are to be tracked explicitly—which is necessary for many queries (***P1***)—the inner node update turns expensive, as the parent of all c children of K (D, E, and F) or the levels of all s descendants change. Subtree and range updates of size s always require all s labels to be altered. So, since all containment-based schemes suffer from the problems ***P2*** and ***P3***, their use in a highly dynamic setting is limited.

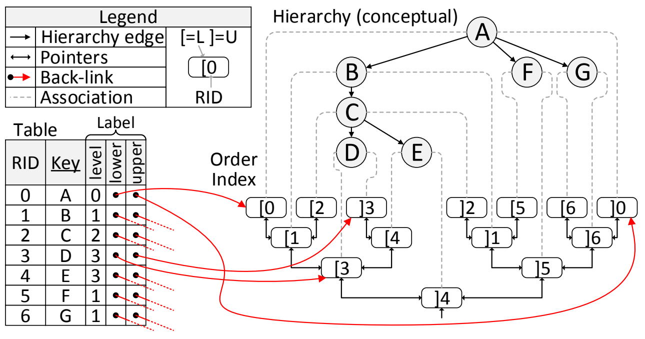
Path-based Labeling Schemes encode the path from the root down to a node into the label. Dewey [20] is a prominent example, and the basis of several more sophisticated schemes. It builds upon the sibling rank, that is, the 1-based position of a node among its siblings. Each node is labeled with the label of its parent node plus a separating dot plus its sibling rank. In the example hierarchy of Fig. 1, node G receives the Dewey label 1.2.4, as G is the fourth child of C, which is the second child of A, which is the first root. Dewey is not dynamic (***P2***): We can easily insert a new node as rightmost sibling, but in order to insert a node a between two siblings, we need to relabel all siblings to the right of a and all their descendants (f in Fig. 6). Since insertion between siblings is a desirable feature, several proposals try to enhance Dewey correspondingly: One prominent representative is Ordpath [17], which is used in Microsoft SQL Server for the *hierarchyid* data type. It is similar to Dewey, but uses only odd numbers to encode sibling ranks, while reserving even numbers for "careting in" new nodes between siblings. This way, Ordpath supports insertions at arbitrary positions without having to relabel existing nodes. In Fig. 1, for example, inserting a sibling between C and H results in the label 1.4.1. Note that the dot notation for labels is only a human-readable surrogate; Ordpath stores it in a more compact binary format. A lot of further dynamic path-based schemes have been proposed: DeweyID [10] improves upon Ordpath by providing gaps that are possibly larger than the ones of Ordpath, thus resulting in less carets and usually shorter labels. CDDE [21] also aims to provide a Dewey encoding with shorter label sizes than Ordpath. The encoding schemes [12, 13, 14] can also be used for building dynamic path-based schemes. In Fig. 6, Dewey refers to static path-based schemes such as Dewey itself and Dyn-Dewey refers to dynamic ones (e. g., Ordpath and CDDE). Dynamic path-based schemes are variable-length labeling schemes, and the proof of [6] holds as well, so they pay the price of potentially unbounded label sizes. In addition, all path-based schemes pay a factor l on all update and most query operations, since the size of each node's label is proportional to its level l.

Considering updates, the dynamic variants are able to insert leaf nodes, but cannot handle inner node updates efficiently, as the paths and thus the labels of all descendants of an updated inner node would change. An exception to this is OrdpathX [4], which can handle inner node insertion without having to relabel other nodes. All path-based schemes inherently cannot handle subtree and range relocations efficiently, as the paths of all descendants have to be updated (***P2***). They are also vulnerable to skewed insertions (***P3***); however, an update sequence that triggers worst-case behavior is much less common than one for a containment-based scheme.

Index-based Schemes use special-purpose secondary index structures to answer queries rather than considering a label. Their advantage is that they generally offer improved update support. B-BOX [19] uses a keyless B$^+$-tree to represent a containment-based scheme dynamically. It has the same update complexity as the BO-Tree in Fig. 6. However, it represents only lower and upper bounds but does not support level or parent information and thus has limited query capabilities (***P1***). Our DeltaNI [7] uses an index to represent a containment-based scheme with level support. As a *versioned* index, it is able to handle time-travel queries. It can be used for unversioned hierarchies by simply keeping all data in a single version delta. While DeltaNI bears none of the three identified problems, its overall query performance is poor in comparison to an unversioned scheme, as our evaluation shows. Both DeltaNI and B-BOX can handle subtree and range relocations in logarithmic worst-case complexity, so they do not show any of the update problems.

4. ORDER INDEXES

We propose the concept of an *Order Index* and three specific data structures AO-Tree, BO-Tree, and O-List. By combining



RID	Key	level	lower	upper
0	A	0	●	●
1	B	1	●	●
2	C	2	●	●
3	D	3	●	●
4	E	3	●	●
5	F	1	●	●
6	G	1	●	●

Figure 7: A hierarchy with Order Index (AO-Tree)

various ideas, from keyless trees (B-BOX) to accumulation trees (DeltaNI) through to gap allocation techniques (GapNI), we avoid the mentioned problems of prior works.

Order Indexes are index-based schemes and as such must be integrated into the database system. They are designed as back-ends for the hierarchy framework devised in [3], which is integrated into the HyPer kernel and is currently being integrated into HANA as well. Consequently, we discuss them in a main-memory context and do not cover disk-based systems, although adapting the concepts is straightforward for BO-Tree and O-List. Order Indexes can be integrated straightforwardly into any database engine that implements hierarchy-aware operators in terms of the query and update primitives from Fig. 2 and 5, such as the framework from [3].

4.1 The Order Index Concept

An Order Index conceptually represents each hierarchy node by a lower bound, an upper bound, and its level. However, the bounds are not explicit numbers or other literals, but rather entries in an ordered data structure. Therefore, the index can be viewed as a dynamic representation of a containment-based scheme along the lines of NI, with *implicitly* represented bounds and explicitly maintained level. Altogether, an Order Index generally consists of three table columns lower, upper, and level, plus the secondary index structure. lower and upper contain pointers into the index structure. We call these pointers *back-links*, as they point back from a table row to an index entry, while common secondary indexes merely point from an index entry to a row through its row ID (RID).

EXAMPLE. Fig. 7 shows an example hierarchy (top) and a pair of an Order Index and a table representing it. The index is actually an AO-Tree, which we explain in Sec. 4.2. A few exemplary back-links are shown as red arrows. An opening bracket denotes a lower and a closing bracket an upper bound; for example,]3 is the entry for the upper bound of row #3.

The index structure maintains the relative order of its entries (hence the term *Order* Index) and provides the following interface (e is an index entry, l a back-link):

find(l) — the entry e to which l points
rid(e) — the id of e's associated row
lower(e) — whether e represents a lower bound
before(e_1, e_2) — whether e_1 is before e_2 in the entry order
next(e) — the next entry in the entry order
adjust_level(e) — the level adjustment for e (see below)

Regardless of which data structure is actually chosen, rid and lower can be implemented straightforwardly by storing the RID and a lower flag with each index entry; next corresponds to a basic traversal of the data structure. Only adjust_level, find, and before differ among our three implementations.

Operation	Implementation
is_descendant(a, b)	$e \leftarrow$ find$(a.$lower$)$; before(find$(b.$lower$), e) \wedge$ before$(e,$ find$(b.$upper$))$
is_child(a, b)	is_descendant$(a, b) \wedge$ level$(a) =$ level$(b) + 1$
is_before_pre(a, b)	before(find$(a.$lower$)$, find$(b.$lower$))$
is_before_post(a, b)	before(find$(a.$upper$)$, find$(b.$upper$))$
level(a)	$a.$level + adjust_level(find$(a.$lower$))$
is_root(a)	level$(a) = 0$
is_leaf(a)	$e \leftarrow$ find$(a.$lower$)$; rid(next(e)) = rid(e)
find(a)	find$(a.$lower$)$
next_pre(e)	$n \leftarrow$ next(e); if lower(n) then n else next_pre(n)
next_post(e)	$n \leftarrow$ next(e); if $\neg$lower(n) then n else next_post(n)
next_sibling(a)	next(find$(a.$upper$))$

Figure 8: Implementing the query primitives

All query primitives introduced in Sec. 3 can be implemented in terms of the six index operations, as shown in Fig. 8. Considering update operations, insert_leaf corresponds to inserting a lower and an upper bound as adjacent entries into the data structure and storing the back-links and an initial level in the corresponding table row. delete_leaf simply removes the two entries from the data structure and the table row. For relocate_range, we conceptually "crop" the corresponding range of bounds $[a, b]$, then alter the *level adjustment*—the value returned by adjust_level—for that range, and finally reinsert $[a, b]$ at the target position. As explained in Sec. 2.2, the other updates are implemented in terms of these operations, so we do not have to cover them explicitly.

Level adjustments enable us to maintain level information dynamically. adjust_level is always added to the level stored in the table row (cf. level(a) in Fig. 8). This way we avoid having to alter the table in case of a range relocation; rather, we update the level adjustment of the relocated range. To do this efficiently we reuse a technique that we originally applied in DeltaNI [7] for different purposes: *accumulation*. Accumulation works for any hierarchically organized data structure that stores its entries in blocks, such as a B-tree or a binary tree (where the "blocks" are just nodes). The idea is to store a *block level* with each block. The level adjustment of an entry e is obtained by summing up the levels of all blocks on the path from e's block to the root block. This allows us to efficiently alter levels during a range relocation: After cropping the bound range $[a, b]$, we add the desired level delta δ to the block level of the root block(s) of that range, which effectively adds δ to the levels of *all* entries within $[a, b]$. Accumulation brings along the cost that level(a) becomes linear in the height of the data structure, usually $\mathcal{O}(\log n)$. However, during an index scan, the level adjustment can be tracked and needs to be refreshed only when a new block starts. This yields amortized constant time for level.

4.2 Order Index Structures

As specific Order Index structures, we propose the AO-Tree based on a keyless AVL tree, the BO-Tree based on a keyless B$^+$-tree, and the O-List based on a linked list of blocks.

AO-Tree. Self-balancing binary trees, such as the AVL tree (our choice) or the red-black tree, offer logarithmic complexity for most operations, which makes them good candidates for an Order Index structure. We must maintain pointers to parent nodes, because the required algorithms navigate from the bottom towards the root rather than the other way round. To compute adjust_level, for example, we sum up all block levels on the path from an entry to the root, as outlined above. Since the trees are balanced, the worst-case complexity for navigating upwards is $\mathcal{O}(\log n)$.

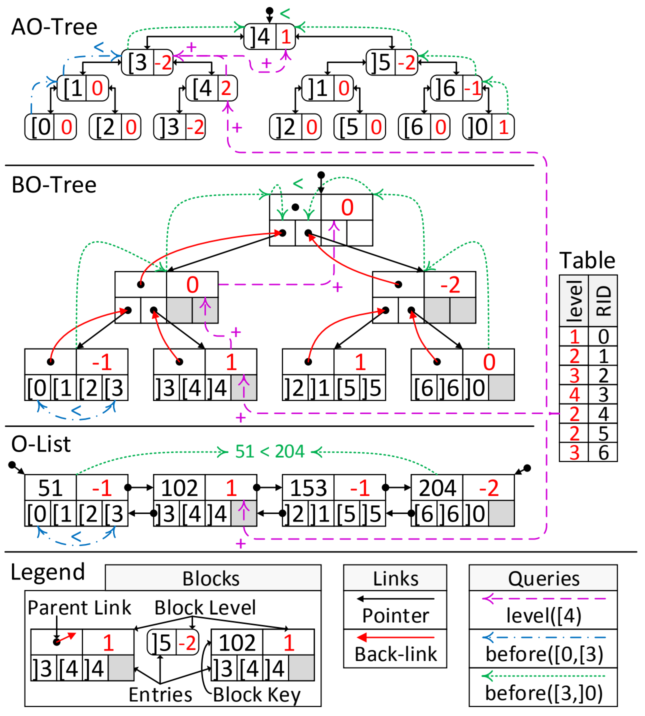


Figure 9: Query evaluation in the Order Indexes

EXAMPLE. The top of Fig. 9 shows the AO-Tree from Fig. 7 in more detail. The red numbers to the right of the entries are the block levels. The purple arrows show how to evaluate level for node 4: We start with the value 2 from the table row #4 (back-links omitted in the figure) and sum up the block levels from [4 upwards to get $2 + 2 - 2 + 1 = 3$.

We check the entry order relation before(e_1, e_2) by simultaneously walking up the tree, starting at e_1 and e_2, to their least common ancestor e', and finally checking which of the two paths arrives at e' from the left.

EXAMPLE. In Fig. 9, we evaluate is_descendant(3, 0) by checking before([0, [3) (blue arrows) and before([3,]0) (green arrows).

Leaf updates correspond to basic binary tree insert and delete operations, with rebalancing if necessary. Range relocations are implemented in terms of the $\mathcal{O}(\log n)$ operations split and join as described in [7]: split splits a binary tree into two, while join concatenates two binary trees. We perform relocate_range$([a, b], p)$ by first splitting the tree before [a and behind]b into a tree T_1 containing all entries smaller than [a, a tree T_2 containing the range $[a, b]$, and a tree T_3 containing all entries greater than]b. We then apply the desired level delta δ to the root of T_2; join T_1 with T_3; split the resulting tree at p, resulting in trees T_4 and T_5; and finally join the remaining trees in the order T_4, T_2, T_5.

Even though all desired operations can be implemented straightforwardly and yield $\mathcal{O}(\log n)$ worst-case runtime, using a binary tree has certain disadvantages: It uses a lot of space for storing three pointers (left, right, parent) and a block level per entry; and it is not particularly cache-friendly, as its entries are scattered in memory, so a traversal will usually incur many cache misses.

BO-Tree. B$^+$-trees are based on blocks of size B rather than single-entry nodes, which greatly improves their cache-friendliness over binary trees. A BO-Tree can be implemented by adapting a B$^+$-tree as follows: each block additionally maintains a back-link to its parent block and a block level.

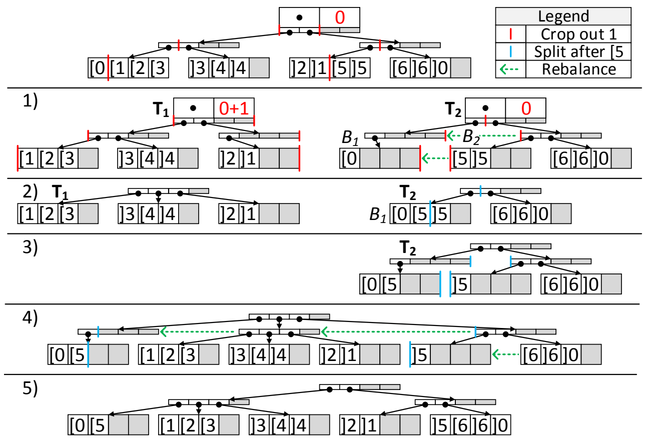


Figure 10: Relocating 1 below 5 in a BO-Tree

In an inner block there are no separator keys but only child block pointers. An entry in a leaf block consists of a row ID and a lower flag. Most B$^+$-Tree operations, including splitting and rebalancing, need almost no adaptions. Key search is no longer required since BO-Trees are keyless and the table stores *back-links* to leaf entries rather than keys (cf. Sec. 4.3). Back-links to parent blocks are needed, because most operations involve leaf-to-root navigation. adjust_level(e), for instance, is computed by summing up all block levels on the path from e's leaf block to the root.

EXAMPLE. The middle of Fig. 9 shows a BO-Tree indexing the hierarchy from Fig. 7. Back-links are displayed as red arrows, block levels as red numbers. The purple arrows show the level query for node 4: We sum up the table level and the block levels on the path from [4 to the root, yielding $2 + 1 + 0 + 0 = 3$.

Since the tree height is in $\mathcal{O}(\log_B n)$, that is the worst- and best-case complexity for computing the level. The wider the blocks in the BO-Tree, the faster level can be computed. Note that a level query does not need to locate the corresponding entry within its block; only the block level is accessed.

before(e_1, e_2) is evaluated as follows: If e_1 and e_2 are located in the same leaf block, compare their positions within that block. Otherwise, walk up the tree to the least common ancestor *lca* of the two blocks containing e_1 and e_2; then determine which of the two paths enters *lca* from further left, by comparing the positions of the two corresponding pointers to the children through which the paths pass.

EXAMPLE. In the figure, entries [0 and [3 are on the same leaf block, so we compare their positions (blue arrow). To evaluate before([3,]0) we walk up to the least common ancestor, which happens to be the root block (green arrows). The [3 path enters the root through child 0 and the]0 path enters through child 1, so [3 is indeed before]0.

To determine the position of a block within its immediate parent block, and to locate an entry within its leaf block, we have to scan that block linearly. Note, however, that the before algorithm needs to determine the child positions within the *lca* block, but *not* within any of the other blocks on the path to *lca*. In the before([3,]0) case, for example, we neither need the positions of e_1 and e_2 within their leaf blocks nor the positions of the leaf blocks within their parents. Thus, we need only one linear block scan per query—rather than one per visited block. Therefore, the worst case-complexity is $\mathcal{O}(B)$ for the scan and $\mathcal{O}(\log_B n)$ for moving up the tree, so $\mathcal{O}(B + \log_B n)$ overall (rather than $\mathcal{O}(B \log_B n)$). If we choose a large B, the query time will be dominated by B.

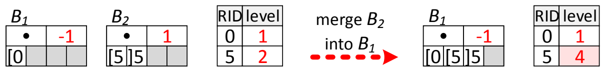


Figure 11: Adjusting levels after a leaf block merge

Sec. 4.3 shows how we can get rid of the linear scan, yielding $\mathcal{O}(\log_B n)$, which is a very good asymptotic bound due to the large logarithm base B. For example, a tree with $B = 1024$ is able to represent a hierarchy with 500 million nodes at a height of only 3, and thus needs at most 3 steps for a containment or level query. Since the root block will probably reside in cache, only 2 cache misses are to be anticipated in this case, which makes the BO-Tree a very cache-efficient data structure.

Leaf updates correspond to simple B$^+$-tree insertions or deletions without prior key search and have an amortized runtime of $\mathcal{O}(1)$. relocate_range($[a, b], p$) is performed as follows: 1) Simultaneously split blocks upwards starting from the positions of [a and]b within their respective leaf blocks. As soon the least common ancestor block is reached, crop out the entry range between the two split child blocks and place it into a newly allocated block. This results in a cropped-out tree T_1 containing the entry range from [a to]b and a tree T_2 containing all other entries. If a level delta δ is to be applied, adjust T_1's root block level by δ. 2) Rebalance blocks split in Step 1 to get rid of underfull blocks. 3) Split T_2 starting at the leaf block of p and continue to split upwards until reaching a block L that is higher than the root of T_1; the height of T_2 may have to be increased during this step to obtain L. 4) Now, simply insert T_1 as a new child block of L. 5) Rebalance blocks split in Step 3.

EXAMPLE. Fig. 10 shows how the subtree rooted in node 1 is relocated below node 5 in the example BO-Tree. We omit back-links and block levels (except for the root). After T_1 is cropped out (red lines), we apply $\delta = +1$ to its root, since the target parent 5 is one level higher than the old parent 0. We rebalance the trees (green arrows), which shrinks T_2. Now we split T_2 behind [5 to create a gap for T_1 (blue lines). This increases its height by one. Now, we add T_1 as second child of the new root of T_2. We rebalance underfull nodes for a properly balanced result.

When entries are moved from a block B_2 to another block B_1 during rebalancing, they must have their level adjusted by B_2.level $- B_1$.level. If B_1 and B_2 are inner blocks, the values to be adjusted are the block levels of their child blocks. If they are leaf blocks, the values to be adjusted are the level fields in the table, but only for lower bound entries.

EXAMPLE. Fig. 11 shows the merging of the two leaf blocks B_1 and B_2 from Fig. 10. Entry [5 is moved to B_1, so its table level becomes $2 + 1 - (-1) = 4$. No level update is performed for]5 because it is an upper bound.

In the worst case, we perform two block splits per level up to the least common ancestor and as many block merges for rebalancing, so we perform up to $\mathcal{O}(\log_B n)$ splits and merges in total. Each block split and merge touches $\mathcal{O}(B)$ entries in the worst case. Thus, the overall worst-case complexity of range relocation is $\mathcal{O}(B \log_B n)$. The smaller the relocated range, the higher the chance that the least common ancestor block has a lower height. So, relocation is faster for small ranges; $\mathcal{O}(B \log_B s)$ in the best case.

While a large block size B is desirable for speeding up queries, it slows down updates. We can, however, take advantage of the fact that leaf blocks are updated more frequently than blocks further up in the tree, and enhance the BO-Tree with support for blocks of different sizes at different levels, based on the concepts described in [18]. As our evaluation shows, using small leaf blocks and larger inner blocks results in a tree that updates almost as fast as trees with small B but queries almost as fast as trees with large B and thus yields the best of both worlds.

O-List. Unlike AO-Tree and BO-Tree, the O-List is not a tree structure but merely a doubly linked list of blocks—hence its name. The bottom of Fig. 9 shows an O-List for the example hierarchy. We use *block keys* to encode the order among the blocks, an idea borrowed from GapNI. Block keys are integers that are assigned using the whole key universe, while leaving gaps so that new blocks can be inserted between two blocks without having to relabel any existing blocks, as long as there is a gap. In addition to the block key, each block maintains a block level field for the level adjustment. The blocks are comparable to BO-Tree leaf blocks without a parent block, and we treat them in a similar manner: Inserting into a full block triggers a split; a block whose load factor drops below a certain percentage (40% in our implementation) is either refilled with entries from a neighboring block or merged with it. When moving entries from one block to another, their levels have to be adjusted (cf. Fig. 11). adjust_level(e) simply returns the level of e's block. before(a, b) first checks if a and b are in the same block; if so, it compares their positions in the block, if not, it compares the keys of their blocks.

EXAMPLE. In Fig. 9, we compute the level of node 4 by adding the block level 1 to the table level 2. before([0, [3) corresponds to an in-block position comparison. before([3,]0) holds because the block key 51 of [3 is less than the block key 204 of]0.

As both adjust_level and before reduce to a constant number of arithmetic operations, they are in $\mathcal{O}(1)$, which makes them even faster than for the BO-Tree. But the improved query performance comes at a price: While leaf updates have $\mathcal{O}(1)$ amortized average-case time complexity, leaf insertion has a linear worst-case complexity. If a block overflows, it has to be split into two, and the new block needs an appropriate key. If no gap is available, the keys of all blocks are relabeled, yielding equally spaced keys again. For a block size of B, there are $\mathcal{O}(\frac{n}{B})$ blocks, so the complexity of relabeling is $\mathcal{O}(\frac{n}{B})$. Splitting a block is in $\mathcal{O}(B)$. Therefore, the worst-case complexity of a leaf insertion is $\mathcal{O}(\frac{n}{B} + B)$.

Range relocations are performed similarly to the BO-Tree, with the difference that only one level of blocks has to be split and rebalanced and the block keys of the moved bound range have to be updated.

EXAMPLE. Fig. 12 shows the relocation of subtree 1 under node 5. 1) The block range for node 1 is cropped out. 2) Underfull block 51 is merged with 153. 3) The list is split after [5, and the block keys of the split blocks are updated to fill the gap evenly. 4) The cropped block range is linked in and its block levels and keys are updated: 1 is added to all block levels. The gap between 68 and 136 fits the three blocks, so their keys are relabeled to divide it evenly. 5) Underfull block 136 is merged with with 119.

Splitting and merging blocks is in $\mathcal{O}(B)$; relabeling all cropped blocks is in $\mathcal{O}(\frac{s}{B})$. Thus, the runtime of subtree relocation is in $\mathcal{O}(\frac{s}{B} + B)$ if the gap fits the cropped range. Otherwise a total relabeling is performed, yielding $\mathcal{O}(\frac{n}{B} + B)$ worst-case runtime. Although the runtime is linear in s, or even linear in n when relabeling, the O-List still performs well in practice. In particular, it is much more dynamic than GapNI, from which the idea of block keys with gaps originated. Its strong point is the divisor of B in the complexity. By choosing a sufficiently large B, e. g., 256 or 1024, we can minimize the cost of relabeling to a point where relabeling becomes feasible even for very large hierarchies. Small and average-size

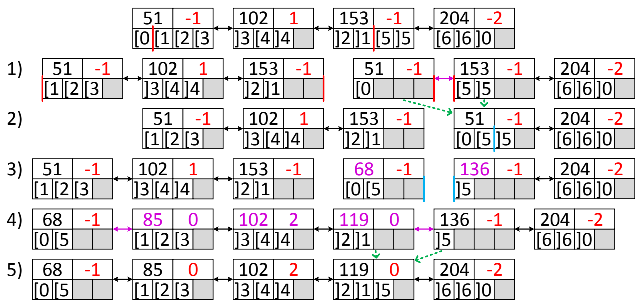


Figure 12: Relocating 1 below 5 in an O-List

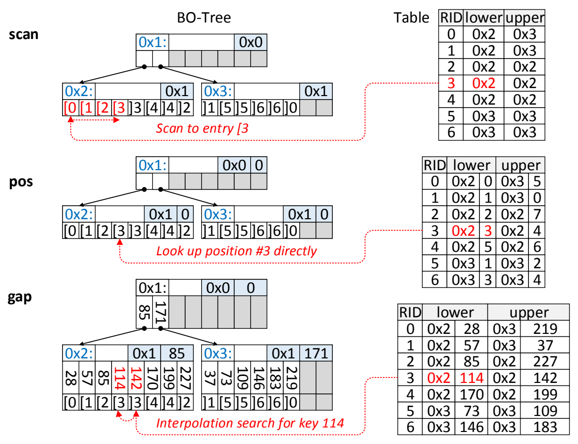
subtrees span only a few blocks in such O-Lists, so relocating them is very efficient.

The O-List mitigates the relabeling problem in two ways: not only does it reduce the time a relabeling takes by factor B, it also multiplies the minimal number of inserts that can possibly trigger a relabeling by that factor. In a basic GapNI encoding using 64-bit integers, an adversary can force a relabeling every $64 - \log n$ insertions by aiming all of them into a specific gap. So a hierarchy of one million nodes would need a relabeling every 44 inserts. For the O-List, this value is multiplied by B. For example, for a huge hierarchy of one billion bounds with $B = 1024$ and 64-bit block keys, the adversary would need roughly 35,000 insertions at the same position to trigger relabeling, and this would touch only around 1 million blocks rather than 1 billion individual bounds. Thus, the amortized relabeling cost per skewed insertion is not just a factor B but a factor of B^2 smaller than for basic GapNI. When insertions happen in a less skewed manner, relabeling will rarely ever be triggered. To increase robustness even further, a wider data type for block keys (e. g., 128-bit integers) can be chosen without sacrificing too much memory, since only one key per block is required. We could even use a variable-length encoding such as CDBS [13] to avoid relabeling altogether. However, we settled for 64 bits in our implementation, since variable-length encodings are less processing-friendly than fixed-size integers, and relabeling happens rarely in sane scenarios.

4.3 Representing Back-Links

We now cover the efficient implementation of find(l), which locates an entry in the Order Index given a back-link l. Apart from locating entries, back-links are also used in the BO-Tree to look up positions of child blocks within their parent.

The implementation of find depends on how back-links are actually represented. This is trivial for the AO-Tree: we can use direct pointers to the AVL tree nodes, as they never move in memory. For the BO-Tree and the O-List, however, entries are shifted around within their blocks or even moved across blocks by rotate, merge, and split operations. In these cases, any pointers to the entries would have to be adjusted. This causes a significant slowdown through random data access, as adjacent entries in blocks do not necessarily correspond to adjacent table tuples (recall that hierarchy indexes are secondary indexes). We investigate three approaches for representing back-links in these data structures: *scan*, *pos*, and *gap*. Fig. 13 illustrates the three strategies for finding entry [3 in a BO-Tree (shown on the left). The relevant contents of the table are shown on the right. The address of a block is shown in its top-left corner. The table and block entries that are touched by find are highlighted in red.


scan
BO-Tree
Scan to entry [3
pos
Look up position #3 directly
gap
Interpolation search for key 114
Table

RID	lower	upper
0	0x2	0x3
1	0x2	0x3
2	0x2	0x2
3	0x2	0x2
4	0x2	0x2
5	0x3	0x3
6	0x3	0x3

RID	lower		upper	
0	0x2	0	0x3	5
1	0x2	1	0x3	0
2	0x2	2	0x2	7
3	0x2	3	0x2	4
4	0x2	5	0x2	6
5	0x3	1	0x3	2
6	0x3	3	0x3	4

RID	lower		upper	
0	0x2	28	0x3	219
1	0x2	57	0x3	37
2	0x2	85	0x2	227
3	0x2	114	0x2	142
4	0x2	170	0x2	199
5	0x3	73	0x3	109
6	0x3	146	0x3	183

Figure 13: Using back-links to find entry [3

scan (top of Fig. 13) is a simple strategy also used by B-BOX [19]. Back-links point only to the block containing the entry, which has to be scanned linearly for the row's ID to locate the entry. *scan* has the advantage that only entries that are migrated to another block during merges, splits, and rotations need their back-links updated. However, linear block scans add an unattractive $\mathcal{O}(B)$ factor to most queries and thus hinder us from using larger blocks.

pos (middle of Fig. 13) represents back-links by a block pointer and the offset in the block. While this eliminates the $\mathcal{O}(B)$ factor and grants $\mathcal{O}(1)$ find performance, it necessitates relabeling even when an entry is only shifted around within its block. As any insertion or deletion in a block involves shifting all entries behind the corresponding entry, this slows down updates considerably, especially for a larger B.

As a compromise, we propose ***gap*** (bottom of Fig. 13), again using the idea of gaps from GapNI: Each entry is tagged with a *block-local key* (1 byte in the example) that is unique only within its block. A back-link consists of a block pointer and a key. Initially the keys are assigned by dividing the key space equally among the entries in a block. When an entry is inserted, it is assigned the arithmetic mean of its neighbors; if no gap is available, all entries in the block are relabeled. The block-local keys are used to locate an entry using binary search or interpolation search. Interpolation search is very beneficial, as block-local keys are initially equally spaced and thus perfectly amenable for interpolation. A block may even be relabeled proactively once an interpolation search takes too many iterations, since this is a sign for heavily skewed keys. The occasional relabeling makes *gap* significantly cheaper than *pos*, which effectively relabels half a block, on average, on *every* update. Like with GapNI, an adversary can trigger frequent relabelings through repeated insertions into a gap. That said, even frequent relabelings are not a serious problem, as they are restricted to a single block of constant size B.

5. PERFORMANCE EVALUATION

We compare our Order Index implementations from the HyPer kernel with several contending indexing schemes implemented in C++. All measurements are executed on an Intel Core I7-4770K CPU running Ubuntu 14.10 with Kernel 3.16.0-23.

5.1 Test Setup

We use a test hierarchy H whose structure is derived from a real-world materials planning application of an SAP customer. As scenarios featuring very large hierarchies are most critical, we expand the size of H to 10^7 nodes by replicating subtrees in such a way that the structural properties, especially the average depth of 10.33, remain equal. Due to the large size, the indexes exceed L3 cache, yielding "worst case" uncached results. We also measured all figures for smaller H sizes, but omit the results as they lead to the same conclusions. For experiments involving subtrees of a specific size, we derive a family of hierarchies H_x containing 10^7 nodes like H, but with a different shape: There is a single root r and all of its children are roots of subtrees of size x. We obtain each subtree by choosing a random subtree of size $\geq x$ from H and then removing random nodes to downsize it to x. Using H_x, we can easily pick a subtree of size x among r's children.

In order to get a complete picture of the assets and drawbacks of the indexing schemes, we measure various important update and query operations. The following update scenarios assess single operations as well as mixed workloads.

bulk_build — Bulk-build hierarchy H using an edge list ordered in pre-order. This simulates the scenario described in [3], where an index is built from scratch from an existing tree representation.

insert — Build H by issuing leaf inserts in a random order, with the constraint that a node has to be inserted before any of its children. This simulates a scenario where the hierarchy is built over time through unskewed inserts.

delete — Delete all nodes from H by issuing randomly chosen leaf deletions. This is the counterpart of insert.

skewed_insert — Choose a node a from H and then insert 10,000 nodes as children of a. This represents a scenario where updates are issued at a single position, and assesses the ability of an indexing scheme to handle skewed inserts.

relocate_subtree$[x]$ — Start with H_x and then relocate 10,000 subtrees of size x to other positions below the root. We vary x in powers of 4 from 8 to 8192. This scenario assesses the performance of complex subtree updates.

relocate_range$[y]$ — Start with H_8 and relocate 10,000 sibling ranges consisting of $y/8$ siblings (and hence y nodes) to other positions below r. Again we vary y in powers of 4 from 8 to 8192. This scenario assesses the performance of this most expressive, most complex class of updates.

mixed_updates$[p]$ — Start with H and issue 100,000 mixed random updates, consisting of either a subtree relocate (probability p) or a leaf insert or delete (probability $1-p$). As the number of inserts and deletes are roughly equal, the hierarchy size does not change much over time. The size of the subtrees chosen for relocation is 107.46 on average. It is derived from the update pattern we observed in the materials planning hierarchy on which H is based. By varying p, we simulate real-world update patterns with a varying relocation rate.

Even though our primary focus is on dynamic hierarchies and thus update operations, we also measure query operations, because a highly dynamic index is useless if it cannot answer queries efficiently. We first assess the query primitives is_descendant, is_child, is_before_pre, is_before_post, level, is_leaf, and find described in Sec. 2.1, involving randomly selected nodes of H. In addition, we measure the performance of a full index scan using next_pre repeatedly, which corresponds to a pre-order traversal over H. Beyond isolated query primitives, we also measure the query from Fig. 3, which uses an index-nested-loop join over the descendant axis

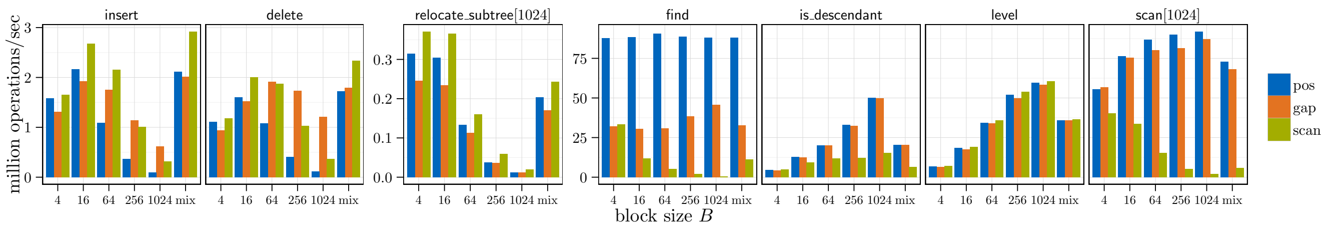


Figure 14: Comparing BO-Tree performance for different back-link representations and block sizes B

and evaluates the level. The performance of an index join varies with the size of the subtree that is scanned for each tuple from the left input S. Therefore, we again use H_x and pick random children of its root r as left input S. As the query basically executes a partial index scan over x entries for each S tuple, we call this measure **scan**$[x]$. It represents an important query pattern and thus gives us a realistic hint of the overall query processing performance we can expect.

5.2 Block Size & Back-Link Representation

We first conduct experiments to determine a good block size B for BO-Tree and O-List, and assess the three back-link variants from Sec. 4.3. We vary B in powers of 4 from 4 to 1024. Block size "mix" refers to a multi-level scheme: 16 for leaf blocks, 64 for height 1 blocks, and 256 for height 2 blocks and above. Fig. 14 shows results for various update and query operations on the BO-Tree. We omit the O-List figures as they lead to the same conclusions.

Concerning updates (left side) it is clearly visible that a smaller block size B between 16 and 64 is most beneficial. Concerning queries, we see that larger blocks are very beneficial, especially for the important is_descendant query (is_child, is_before_pre, is_before_post behave similarly). We seek a B that provides a good trade-off between query and update performance. For the BO-Tree, our favorite is mix: queries are not as fast as for $B = 1024$, but still quite fast, especially on the compound scan[1024] query; in return, it is very efficient for updates, where large B such as 1024 suffer, especially when relocating subtrees (cf. relocate_subtree[1024]). For rather static data, 1024 can be a good choice nevertheless, as it maximizes query speed.

Concerning the back-link representations, we observe that *scan* is infeasible: queries are too slow, and the increased update speed it offers is not significant, as *pos* and especially *gap* perform well enough. For the important scan[1024] query, *scan* is up to 50 times slower than the other representations. A query where *scan* seems acceptable is level, but this is only due to the fact that this query does not involve back-links at all. *pos* performs fine for smaller B and is preferable there, while *gap* becomes almost mandatory for $B \geq 256$, where *pos* loses too much insert and delete performance.

In conclusion, a mixed block size is the best trade-off for the BO-Tree. The *scan* back-links should be avoided; *pos* are preferable for small B and scenarios with few updates, while *gap* is preferable for larger B and dynamic scenarios. For the remaining measurements we use *gap* back-links, $B =$ mix for BO-Tree, and $B \in \{16, 64, 256\}$ for O-List.

5.3 Comparison to Existing Schemes

Our indexing schemes aim at highly dynamic settings where high update performance even for complex updates is desirable. To show that prior dynamic labeling schemes cannot support these settings efficiently, we compare our schemes AO-Tree, BO-Tree[B] and O-List[B] to promising contenders from different categories: Ordpath [17] as a representative for path-based variable-length schemes; CDBS [13] for containment-based variable-length schemes; GapNI [15] (with explicitly maintained level) for containment-based schemes with gaps. All three are backed up by a B-tree over the labels, which is used for scans where necessary (most queries can be answered by only considering the labels). Our fourth contender is DeltaNI [7] as a versioned, index-based scheme. Finally, we also measure the naïve schemes Adjacency and Linked. Adjacency is indexed by a hash index on both the key and the parent column. Linked does not need any index as it can access the nodes directly via pointers.

Fig. 15 shows the **query performance** for various types of queries. We first observe that the AO-Tree performs very poorly, as its data is scattered and the height of the AVL tree is large in comparison to a B-tree, so the index suffers from a high number of cache misses. In contrast, the BO-Tree and the O-List perform well for all query types. For queries that labeling schemes can answer by considering just the labels, the Order Indexes are slower than labeling schemes because they need to consider also the index and thus suffer at least one extra cache miss. For queries that need to access the index (e. g., is_leaf, traversal), the Order Indexes offer a very good performance. The find operation is where Order Indexes excel over labeling schemes, because they use fast $\mathcal{O}(1)$ back-links instead of $\mathcal{O}(\log n)$ B-tree key searches. DeltaNI appears to perform fine on most queries, but this is only the case because all the complexity is hidden in find, which DeltaNI must always perform before it can issue a query. The naïve schemes Linked and Adjacency show their greatest weakness here: they fail to offer robust query performance. While some queries such as is_child are fast, other important queries such as is_descendant and level are unacceptably slow.

As the queries in Fig. 15 involve randomly selected nodes, Order Indexes are expected to be slow due to extra cache misses on accessing the index at random positions. In contrast, when scanning the index, the current block is already in cache and level becomes an $\mathcal{O}(1)$ operation, so we expect our schemes to perform better in this case. Fig. 16 shows the index-nested-loop join performance scan$[x]$ with varying x. All schemes benefit from larger subtrees, because the initial find is the most expensive operation and always incurs a cache miss. The subsequent index scan often incurs no further cache misses due to prefetching. On this query, all Order Index variants except for AO-Tree are superior. Linked is second fastest as it just has to chase pointers, which is fast but not as cache-friendly as scanning blocks. GapNI is also quite fast by virtue of its processing-friendly integer labels, while CDBS and Ordpath suffer from their variable-length labels. Ordpath additionally suffers from the level query that requires counting path elements; without this, it would be on a par with CDBS. DeltaNI is quite slow; it pays the price of a full-blown versioned scheme. The slowest of all is Adjacency,

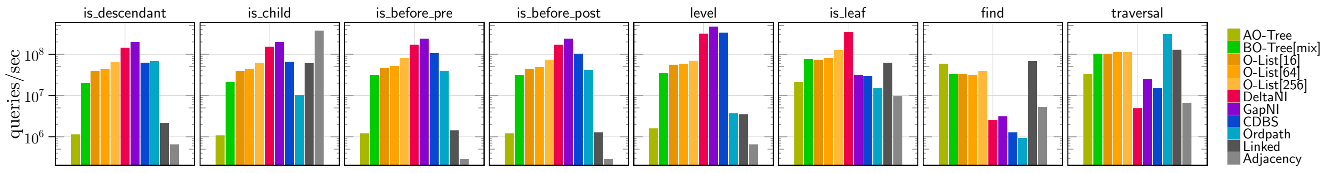


Figure 15: Performance of different query primitives

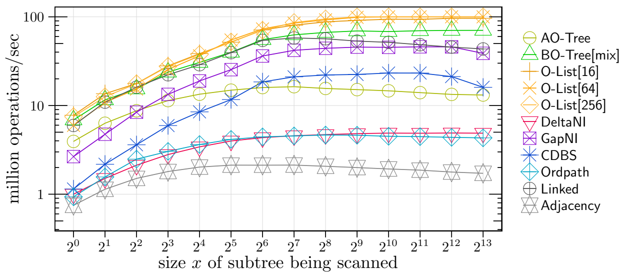


Figure 16: Compound query performance (scan[x])

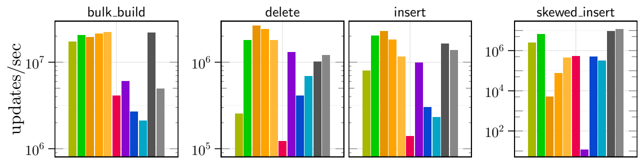


Figure 17: Bulk build and leaf update performance

which has to use its hash indexes repeatedly to find child nodes to be scanned. We conclude that Order Indexes excel at an index-nested-loop-based hierarchy join. Since joins are at the core of common hierarchy queries, we anticipate outstanding overall query performance in real-world scenarios.

Fig. 17 depicts the **update performance** for bulk building and leaf updates. As anticipated in Fig. 6, Order Indexes and the naïve schemes perform very well for most updates. For bulk building and non-skewed leaf updates, Order Indexes are superior to labeling schemes, though the contenders—in particular GapNI—also perform reasonably. Skewed insertions, however, force frequent relabelings, so GapNI's performance plummets. O-List has the same problem, as its block keys are also gap-based and skewed insertions fill up these gaps. Still, as anticipated, O-List significantly outperforms GapNI, because new block keys are required less often; the larger B, the better. So, an O-List with sufficiently large blocks can handle skewed insertions quite well. BO-Tree even benefits from skewed insertions, as the blocks where the insertions happen will usually be in cache, and so it outperforms labeling schemes by a factor of around 20.

Let us now consider complex updates. Fig. 18 (a) shows the workload simulation mixed_updates[p] with p varying exponentially between 0% (no relocations) and 32%. While all indexing schemes perform acceptably for $p = 0\%$, the contenders drop rapidly on increasing p; starting at only $p = 0.32\%$ for labeling schemes. In this scenario the size of the relocated subtrees is "only" 107; larger sizes exacerbate the situation. To investigate this, Fig. 18 (b) shows relocate_subtree[x] over varying x, the size of the relocated subtrees. As we anticipate, all contenders but the naïve schemes drop linearly in x, as they have to relabel all x nodes. GapNI is particularly slow due to additional global relabelings. At $x = 8192$, labeling schemes can handle only around 100 updates per second, while Order Indexes remain fast at around 200,000 updates per second. The figures also attest that an O-List is just a list of blocks as opposed to a robust tree structure: it turns slow when too many blocks are involved (relabeled) in a relocation. The figure suggests a rule of thumb: Once x exceeds $64 \cdot B$, performance drops noticeably (at 1024 for O-List[16] and 4096 for O-List[64]), so B should be chosen accordingly.

The naïve schemes handle subtree relocation exceptionally well by just updating single values or pointers. This advantage vanishes as soon as we consider the more potent sibling range updates in Fig. 18 (c). Now, the naïve schemes have to process all siblings individually, so their performance drops with increasing range sizes. Only Order Indexes and DeltaNI handle these updates well, but again, O-List turns slow when the range becomes too large.

For brevity reasons we do not include figures for inner node updates; they are comparable to sibling range updates, because the implementation is similar: Labeling schemes need costly relabelings for all nodes below the updated inner node a. GapNI and CDBS need to update the levels of these nodes, other schemes need to update the whole label. The naïve schemes have to relabel all children of a, as for range relocation. Order Indexes and DeltaNI implement inner node updates in terms of range relocations.

We conclude that while all contenders handle leaf inserts and deletes well, only Order Indexes and DeltaNI can handle all kinds of complex updates efficiently. Thus, Order Indexes offer large benefits for use cases featuring complex updates.

Fig. 19 compares the **memory consumption** of all indexes in bytes per hierarchy node. The first column is the size after bulk-building H, the second column after building H from random inserts; the third column is the average size increase per skewed insertion after 10000 skewed insertions have taken place. We see that *gap* back-links take the most space, 8 bytes extra: 2 bytes per key in the entry and in the label, times two because each node has a lower and an upper bound. *pos* back-links require only 2 extra bytes: 1 byte for the position stored in the label, per bound. Smaller blocks incur a small overhead over larger ones. Apart from that, the sizes are comparable to the contenders, with the exception of Ordpath, which stores only one path label as opposed to two bound labels. Note however, that the ordpaths in all our tests contained almost no carets, so this represents a favorable case for this scheme. CDBS shows its greatest demerit: Skewed insertions blow up the label size; each new label is 2 bits longer than the previous one. The sizes for DeltaNI and AOTree are constantly large, since both use space-costly binary trees with parent pointers. Adjacency's memory usage is dominated by its hash indexes. Linked constantly consumes $7 \times 8 = 56$ bytes per node: 5 pointers plus a row ID, plus 1 pointer from the table to each node.

Alltogether, our experiments show that our proposed Order Indexes handle queries and updates competitively. Their largest benefit over the contenders is robustness: Especially BO-Tree performs well throughout all disciplines, while each

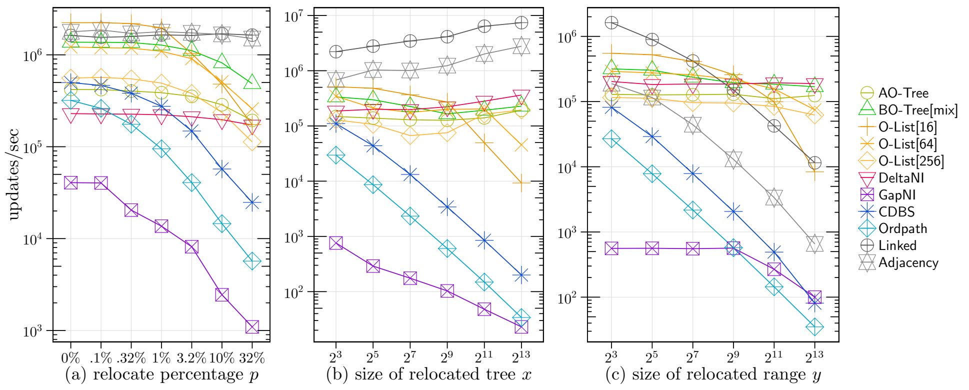


Figure 18: (a) mixed_updates[p] (b) relocate_subtree[x] (c) relocate_range[y]

Index	**bulk**	**insert**	**skewed**
AO-Tree	96.0	96.0	96.0
BO-Tree[mix, pos]	43.3	55.8	72.0
BO-Tree[mix, gap]	50.6	66.1	89.1
BO-Tree[mix, scan]	41.3	52.3	68.9
O-List[16, pos]	41.8	51.5	65.5
O-List[64, pos]	38.9	48.0	59.9
O-List[256, pos]	38.2	47.1	58.7
O-List[16, gap]	47.8	60.8	79.5
O-List[64, gap]	44.9	57.4	73.9
O-List[256, gap]	44.2	58.7	72.7
O-List[16, scan]	39.8	49.5	63.5
O-List[64, scan]	36.9	46.0	57.9
O-List[256, scan]	36.2	45.1	56.7
DeltaNI	90.6	211.2	167.2
GapNI	53.8	66.1	54.0
CDBS	71.9	97.9	2579.0
Ordpath	27.0	33.9	41.5
Linked	56.0	56.0	56.0
Adjacency	81.4	81.4	40.0

Figure 19: bytes/node

contender has a problem in at least one of the disciplines. As we anticipate, Order Indexes yield the largest gains over prior techniques in settings featuring complex updates. AO-Tree performs poorly; it is thus only interesting in theory due to its conceptual simplicity. BO-Tree with mixed block sizes is an excellent all-round index structure with full robustness for all update operations; it should be the first choice when the update pattern is unknown. O-List with sufficiently large block size outperforms BO-Tree in queries by around 50%, but it is less robust in dealing with skewed insertions and relocations of large subtrees and ranges.

6. CONCLUSION

In this paper we have investigated into indexing schemes for highly dynamic hierarchical data. Our analysis leads us to the finding that existing indexing schemes bear three main problems: lack of query capabilities, insufficient complex update support, and vulnerability to skewed updates. We therefore propose Order Indexes as an efficient indexing technique for highly dynamic settings. They can be viewed as a dynamic flavor of a nested intervals labeling, using the concept of accumulation to maintain node levels while supporting even complex and skewed updates efficiently. Of our three implementations AO-Tree, BO-Tree, and O-List, the latter two yield robust and competitive query and update performance. Order Indexes considerably outperform prior techniques when considering complex updates on subtrees, sibling ranges, and inner nodes. Our evaluation shows how carefully choosing a suitable back-link representation and block size can further optimize performance. The BO-Tree with varying block sizes yields a particularly attractive query/update tradeoff, making it a prime choice for indexing dynamic hierarchies.

7. REFERENCES

[1] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A robust numbering scheme for XML documents. In *ICDE*, pages 705–707, 2003.
[2] P. Boncz, S. Manegold, and J. Rittinger. Updating the pre/post plane in MonetDB/XQuery. In *XIME-P*, 2005.
[3] R. Brunel, J. Finis, G. Franz, N. May, A. Kemper, T. Neumann, and F. Faerber. Supporting hierarchical data in SAP HANA. In *ICDE*, pages 1280–1291, 2015.
[4] J. Cai and C. K. Poon. OrdPathX: Supporting two dimensions of node insertion in XML data. In *DEXA*, pages 903–908, 2009.
[5] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. *SIGMOD Rec.*, 26(2):26–37, 1997.
[6] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. *SIAM Journal on Computing*, 39(5):2048–2074, 2010.
[7] J. Finis, R. Brunel, A. Kemper, T. Neumann, F. Faerber, and N. May. DeltaNI: An efficient labeling scheme for versioned hierarchical data. In *SIGMOD*, pages 905–916, 2013.
[8] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a relational DBMS to watch its (axis) steps. In *VLDB*, pages 524–535, 2003.
[9] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. Naughton, and D. DeWitt. Mixed mode XML query processing. In *VLDB*, pages 225–236, 2003.
[10] M. Haustein, T. Härder, C. Mathis, and M. Wagner. DeweyIDs—the key to fine-grained management of XML documents. In *SBBD*, pages 85–99, 2005.
[11] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.
[12] C. Li and T. W. Ling. QED: A novel quaternary encoding to completely avoid re-labeling in XML updates. In *CIKM*, pages 501–508, 2005.
[13] C. Li, T. W. Ling, and M. Hu. Efficient processing of updates in dynamic XML data. In *ICDE*, 2006.
[14] C. Li, T. W. Ling, and M. Hu. Efficient updates in dynamic XML data: From binary string to quaternary string. *VLDB Journal*, 17(3):573–601, 2008.
[15] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, 2001.
[16] J.-K. Min, J. Lee, and C.-W. Chung. An efficient XML encoding and labeling method for query processing and updating on dynamic XML data. *JSS*, 82(3):503–515, 2009.
[17] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, pages 903–908, 2004.
[18] S. Sasaki and T. Araki. Modularizing B^+-trees: Three-level B^+-trees work fine. In *ADMS*, pages 46–57, 2013.
[19] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In *ICDE*, pages 285–296, 2005.
[20] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, pages 204–215, 2002.
[21] L. Xu, T. W. Ling, H. Wu, and Z. Bao. DDE: From Dewey to a fully dynamic XML labeling scheme. In *SIGMOD*, pages 719–730, 2009.
[22] J.-H. Yun and C.-W. Chung. Dynamic interval-based labeling scheme for efficient XML query and update processing. *JSS*, 81(1):56–70, 2008.
[23] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 30(2):425–436, 2001.