DeltaNI: An Efficient Labeling Scheme for Versioned Hierarchical Data

Jan Finis* Robert Brunel* Alfons Kemper* Thomas Neumann* Franz Färber† Norman May†

* Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
† SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany

* *firstname.lastname*@cs.tum.edu † {franz.faerber, norman.may}@sap.com

ABSTRACT

Main-memory database systems are emerging as the new backbone of business applications. Besides flat relational data representations also hierarchical ones are essential for these modern applications; therefore we devise a new indexing and versioning approach for hierarchies that is deeply integrated into the relational kernel.

We propose the *DeltaNI* index as a versioned pendant of the nested intervals (NI) labeling scheme. The index is space- and time-efficient and yields a gapless, fixed-size integer NI labeling for each version while also supporting branching histories. In contrast to a naïve NI labeling, it facilitates even complex updates of the tree structure. As many query processing techniques that work on top of the NI labeling have already been proposed, our index can be used as a building block for processing various kinds of queries. We evaluate the performance of the index on large inputs consisting of millions of nodes and thousands of versions. Thereby we show that DeltaNI scales well and can deliver satisfying performance for large business scenarios.

Categories and Subject Descriptors

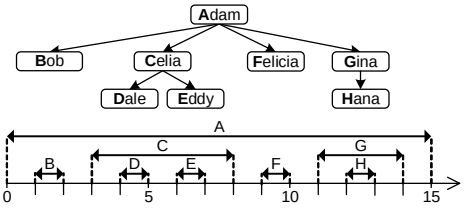
H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

Keywords

Database indexing; hierarchical data; hierarchy indexing; multiversion indexing; labeling schemes; nested intervals

1. INTRODUCTION

A lot of business use cases feature hierarchical data; but efficient support for this kind of data is still missing in most relational database systems. Especially in Enterprise Resource Planning (ERP) applications, various kinds of large hierarchies exist. For example, companies need to manage human resource (HR) hierarchies which model the relationship between their employees (cf. Figure 1), enterprise asset (EA) hierarchies which model all production-relevant assets and their parts (e. g., plants, machines, machine-parts, tools, equipment), or material hierarchies which model so-called "bills of materials" (BOM) that constitute a hierarchical arrangement of components to assemble an end product. These hierarchies (in particular EA) can become tremendously large: We obtained statistics of a major mechanical engineering company, which maintains an EA hierarchy of 59 million nodes in its ERP system. BOMs of large products can also consist of millions of nodes (e. g., a Boeing 747-400 consists of six million parts [3]). This data is also used for reporting purposes that feature complex OLAP-style queries over various recursive structural properties of the hierarchies. Querying these properties using the flat relational model (i. e., via recursive SQL) is very inefficient. For example, finding all descendants of a node requires a repeated self-join. Al-Khalifa et al. [1] have shown that such a join performs very poorly in comparison to structural joins that use tree-aware indexes. Consequently, such indexes are necessary to support hierarchical data as efficiently as modern database systems support relational data.



Employees		
Name	**Salary**	**...**
Adam	80,000	...
Bob	55,000	...
Celia	70,000	...
Dale	55,000	...
Eddy	45,000	...
Felicia	60,000	...
Gina	75,000	...
Hana	45,000	...

Figure 1: An HR hierarchy and its NI encoding

In order to deliver satisfying performance for complex queries on large hierarchies—such as the mentioned ERP use cases—we integrate hierarchy support with special tree-aware indexes into modern main-memory database systems. The hierarchies are to be integrated very tightly into the relational model, enabling hierarchies over the tuples of one or even more relations. For example, an HR hierarchy might consist of tuples from an employee relation and a district relation. Such a tight integration will allow customers to query and join relational and hierarchical data in one query, facilitating complex reporting queries over structural properties of the hierarchy *and* attributes of the tuples over which the hierarchy is built. For example, in the HR hierarchy from Figure 1, a query on relational and hierarchical data would be "sum up the salary of all subordinates of Celia".

For traceability and confirmability reasons, versioning is a central part of many ERP applications. Consequently, delivering satisfying query performance is even more difficult

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

since queries should also be able to work on former versions of the hierarchy. Efficient indexing of versioned data poses a major challenge, as indexes for non-versioned data are not directly applicable. Our goal is to develop a versioned, tree-aware index that can efficiently handle even very large hierarchies like the aforementioned use cases. Such a hierarchy is usually versioned on a daily basis for several years, so the result is a versioned hierarchy with millions of nodes and thousands of versions. Other applications may need even finer grained version control resulting in millions of versions.

Labeling schemes are prominent approaches for indexing hierarchical data that are used by most contributions in the field of XML query processing. Here, a constant number of labels is assigned to each node and certain queries can be answered by only considering the labels. A prominent labeling scheme is the nested intervals (NI) scheme, in which each node v is labeled with an interval $[lower, upper]$ that is a proper subinterval of the interval of the parent node of v.

We propose the *DeltaNI* index applying the NI scheme to versioned hierarchies. The index is efficient in space and time by representing only a base version as a fully materialized NI encoding; other versions are represented by deltas which transform the interval bounds between different versions. By grouping the deltas in an exponential fashion, the index allows executing queries in each version of a history of n versions while applying at most $\log n$ deltas. We also show how various update operations—even sophisticated ones such as moving or deleting ranges of nodes—can be reduced to a simple *swap* operation in the NI encoding. By proposing an efficient algorithm for *swap* on our delta representation, we achieve good update performance. By materializing additional base versions at carefully chosen points in the history, we further increase the performance and reduce the space consumption of the index.

2. HIERARCHIES IN RDBMS

This section discusses important topics for our approach including the relational storage of hierarchies, the NI labeling scheme, the queries and updates to be considered, and possible application areas for the index.

We define a hierarchy as a forest of rooted, ordered trees. Applications can also limit a hierarchy to be a tree instead of a forest (i. e., only one root exists) or may define that the order among siblings is not important and thus is not made visible to the user. Each node may carry a fixed number of attributes while edges may not carry any attributes.

Representation in RDBMSs. As it is our ambition to speed up queries which work on structural properties of the hierarchy (since other queries are already supported well by the relational model) we represent a hierarchy by a relation, in which each row represents one node storing its attributes, and an index which encodes the structure of the hierarchy—the DeltaNI index in our case. Our focus is the index which is responsible for storing the complete version history of the hierarchy structure. To provide versioning for the node attributes, well-researched techniques for relational databases can be used which are out of the scope of our contribution.

NI Encoding. Our index is built upon the NI labeling scheme (also referred to as interval encoding, NI encoding, or range-based encoding). Here, each node is represented by the integer interval $[lower, upper]$. The encoding can be obtained by a depth-first traversal in which the lower bounds are assigned in pre-order and the upper bounds are assigned in post-order from a global counter. The NI encoding of the hierarchy in the upper part of Figure 1 is shown below it. One can directly see the important property of the interval encoding: If a node v_2 is a descendant of another node v_1, its interval is a proper sub-interval of v_1's interval, i. e., $n_1.lower < n_2.lower$ and $n_1.upper > n_2.upper$.

Versioned Hierarchies. A version history $V_0, V_1, \ldots, V_n$ of a hierarchy depicts certain states of that hierarchy and allows queries in each version. Updates are only allowed in the latest version. Although we assume a linear version history for brevity, our approach also supports the branching of version histories. A new branch can be created based on any existing version and updates can be performed on the latest version of each branch. We place no restrictions on when a new version is created. Some applications might create a new version with each update while others might create new versions on a regular basis (e. g., daily).

Queries. Our index yields a fully-featured NI encoding for each version of the hierarchy. Consequently, all queries such an encoding can answer for a non-versioned hierarchy can be answered for versioned hierarchies with DeltaNI. For example, the index can be used as a basis for the staircase join[1] [15], the Stack-Tree join [1], or the TwigStack join[2] [5] in order to answer XPath-style queries. Refer to [7] for an overview of other prominent queries which can be executed efficiently on the NI scheme. For this paper, we focus on providing an efficient NI encoding for each version of the hierarchy. Aggregate or diff queries that span more than one version are out of the scope of this paper.

As our contribution is a low-level index, we will not present the execution of complex queries—this task is accomplished by a higher-level layer of the database (e. g., the staircase join). A simple query which can be directly answered by the index is the calculation of the size of the subtree rooted at a node v by calculating $(v.upper - v.lower - 1)/2$. For example, a subtree query in the hierarchy from Figure 1 may be: "How many employees are (transitively) supervised by Adam?" Using the interval encoding, the answer is $(15-0-1)/2 = 7$. In the versioned case, the query would be extended to work on a certain version, e. g., "How many employees are supervised by Adam in Version 42?". Although we only present such simple (yet useful) queries for brevity reasons, keep in mind that the index can be used by a database system to answer a wide range of complex queries efficiently.

Updates. Updating a hierarchy consists of adding, removing, or moving nodes in the hierarchy. The following update operations are to be supported:

- *insertBefore*(b): Inserts a new node before interval bound b.
- *moveSiblingRangeBefore*(v, v', b): Moves all siblings between v and v' (inclusively) and their descendants before bound b. v must be a left sibling of v' or $v = v'$.
- *deleteSiblingRange*(v, v'): Deletes all siblings between v and v' (inclusively) and their descendants. v must be a left sibling of v' or $v = v'$.

The defined set of update operations is very powerful, as it allows not only single node insertion and deletion—which

[1]The staircase join actually works on the pre/post encoding. However, the NI encoding is similarly expressive, so the staircase join also works on the NI encoding with only minor changes.

[2]Stack-Tree and TwigStag join require an additional *level* label for the child axis. This label can also be modeled by our deltas but is not discussed here for the sake of brevity.

most related work is restricted to—but also subtree deletion and the moving of nodes, subtrees, and even whole ranges of siblings. These operations are important in many use cases: For example, a division in an HR hierarchy receiving a new head (a comparatively frequent case) can be modeled by simply moving all nodes in that division below the new head with a single operation. In EA hierarchies, assets like equipment or vehicles (which form a subtree, since they consist of various parts) are relocated frequently: Relocations constituted almost 50% of all updates in some of the EA hierarchies of ERP customers we inspected.

With insert and delete only, a relocation would result in one delete and one insert per node in the range to be relocated. This would result in a very high update cost and the resulting delta would contain many operations also yielding increased space consumption. Consequently, such a powerful set of update operations is indispensable for the wide applicability of a hierarchy index. Our index supports all these updates in worst-case logarithmic time.

Application Areas for DeltaNI. The most obvious application area for the index is the version control of hierarchical data. Another possible use case are transaction-time temporal hierarchies. The index (as any other version control approach) can directly be used for this purpose. An additional lookup data structure (e. g., a search tree) which maps time intervals to versions has to be maintained, allowing to find the version that corresponds to a timestamp. We assume general hierarchies that subsume XML, so the index can also be used for managing versioned XML data.

The NI encoding is by default not dynamic (i. e., not efficiently updatable), since an update needs to alter $\mathcal{O}(n)$ bounds on average. Contrarily, the DeltaNI index can be used as an efficiently updatable NI labeling scheme for non-versioned databases: Gathering all incoming updates in a single delta is sufficient for making an NI encoding dynamic. Finally, the deltas in this approach can also be used for logging, as a delta accurately describes a set of changes.

3. INTERVAL DELTAS

In essence, our approach for efficiently storing the version history of a hierarchy consists of saving one or more base versions explicitly using the NI encoding and maintaining all other versions as interval deltas only. This allows for a space-efficient compression of the version history while still supporting efficient querying.

We define an *interval delta* $\delta : \mathbb{N} \to \mathbb{N}$ as a bijective function mapping interval bounds from a source version V to a target version V'. When necessary, we explicitly specify the source and target versions of a delta using the notation $\delta_{V \mapsto V'}$. Given an interval bound b of a node in V, $\delta_{V \mapsto V'}(b)$ yields the corresponding bound in V' and $\delta^{-1}_{V \mapsto V'}$ maps back from V' to V. We denote the interval encoding of the source version as *source space* and the one of the target version as *target space*. Thus, δ is a function mapping from the source to the target space.

Obviously, the full interval encoding of the target version can be obtained by applying δ to all interval bounds of the source version. However, the delta can also be used to answer queries without computing the target version intervals completely, as the delta allows transforming only the bounds which are relevant for a query.

There is one pitfall when using interval deltas to represent the version history of a hierarchy: Not all nodes may have existed in the base version V. These nodes do not have any bounds in the base version, thus computing their bounds in other versions V' using $\delta_{V \mapsto V'}$ is impossible. In addition, there might be nodes which were deleted in intermediate versions. To handle insertions and deletions consistently, we make the following enhancements, which we call *active region approach*: For each version V of the history, the maximum bound value in that version, denoted as $\max(V)$, is stored. By definition, any bound value greater than $\max(V)$ does not exist in version V (i. e., "is inactive"). In addition, for every base version V, we define $|V|$ as the number of bounds stored in V also including bounds greater than $\max(V)$. These enhancements allow us to model bounds that do not exist in a version. Consider a base version V and a version V' which adds a new node v with bounds $[v.lower, v.upper]$. This node insertion is modeled by adding the two bounds $b_1 = |V| + 1$ and $b_2 = |V| + 2$ into the base version V (which also increments $|V|$ by 2) but without increasing $\max(V)$, because b_1 and b_2 do not exist in V. To yield the correct result in V', the delta is adjusted correspondingly: $\delta_{V \mapsto V'}(b_1) = v.lower$ and $\delta_{V \mapsto V'}(b_2) = v.upper$. Finally, $\max(V')$ is incremented by 2 because this version now contains two more bounds. A node deletion in a version V' can simply be achieved by moving the bounds of the node to be deleted past $\max(V')$ and reducing $\max(V')$ accordingly. We denote the interval $[0, \max(V)]$ the *active region* of version V. The test whether a node v exists in a version V to which a delta δ exists is performed by checking whether the lower bound of v (and thus also the upper bound) is active, i. e., $\delta(v.lower) \leq \max(V)$.

Note that each node is uniquely identified by its bounds in the base version, since these bounds will never be updated. Thus, they constitute a durable numbering for the nodes in a versioned hierarchy. Given a bound b in a version V', one can obtain the node to which b belongs by applying reverse deltas to b, transforming the bound back to the base version and looking up the corresponding node there.

4. EFFICIENT DELTA REPRESENTATION

To render the interval delta approach feasible, the resulting delta representation must be efficient in space and time. A reasonable space complexity requirement for a delta δ is $\mathcal{O}(c)$ where c is the number of change operations which led from the source to the target version. In the worst case, this is also the best possible bound, because each change must be represented somehow, requiring at least a constant amount of space per change. A reasonable upper bound for the time complexity of δ, i. e., the time it takes to compute $\delta(b)$ for any interval bound b in the source version, is $\mathcal{O}(\log c)$. Any higher non-logarithmic bound would make the approach infeasible for deltas containing a large amount of changes. Our approach satisfies both mentioned complexity bounds. Note that the space and time complexities of our delta representation grow only with respect to the number of changes between the source and the target version. Especially, the complexities do not grow with the number of nodes or edges in the source or target version.

A first naïve delta representation would be to store all bounds which have changed between V and V'. However, a node insertion changes an average of $n/2$ bounds, yielding $\mathcal{O}(c \cdot n)$ space complexity.

Our technique for delta storage leverages the fact that each change introduces only a constant number of trans-

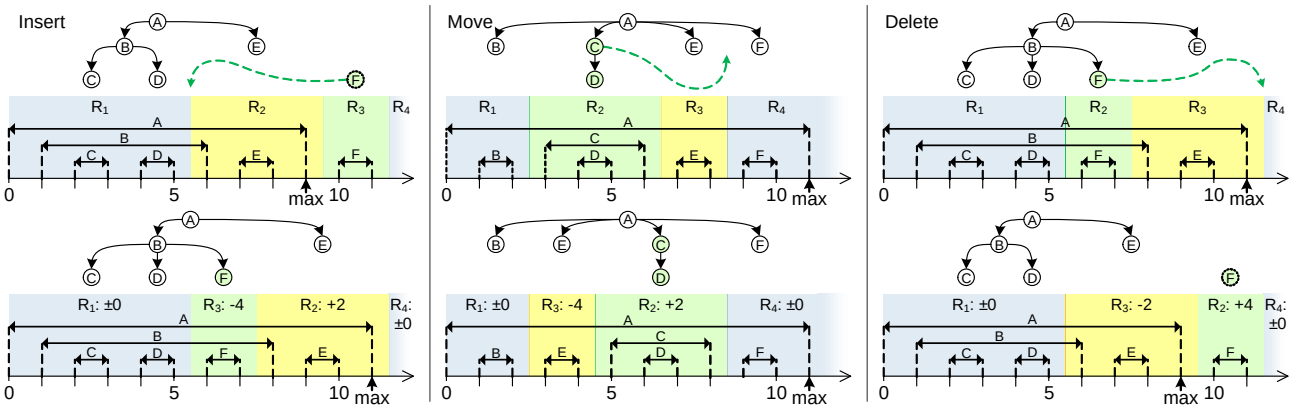


Figure 2: Updating a hierarchy with *insertBefore*(6) (left), *moveSiblingRangeBefore*$(C, C, 9)$ (middle), and *deleteSiblingRange*(F, F) (right). These operations are modeled by swapping R_2 with R_3 and updating max**.**

lations of ranges of interval bounds: Let $R_2 = [a, b]$ and $R_3 = [b{+}1, c]$ be two adjacent intervals and let $swap(R_2, R_3)$ be a function that swaps all bounds in R_2 with the bounds in R_3, that is, all bounds in R_2 are incremented (translated) by the size of R_3 and all bounds in R_3 are decremented by the size of R_2. We call the intervals R_2 and R_3 *translation ranges*, since they constitute ranges of bounds that are translated together. Since translation ranges are *intervals of interval bounds*, the name "bound" is confusing in this context: All values in a translation range are bounds, but the translation range has a lower and upper bound itself. For clarification reasons, we distinguish between bounds and borders: We call all values represented by a delta *bounds*. In contrast, we use lower/upper *border* when referring to the least/greatest bound that lies in a translation range.

The key observation is that each update of a tree, as defined in Section 2, can be modeled in the interval bound space by a *swap* of two adjacent translation ranges, followed by an update of the max value in case of insertion or deletion to adjust the size of the active region. Figure 2 depicts the implementation of the updates by swapping two ranges. The middle of the figure shows the relocation of the subtree rooted at node C to the right of node E. The hierarchy before the update with its NI encoding is shown on top and the resulting hierarchy below. This relocation is simply accomplished by a swap of the range $R_2 = [3, 6]$ (all bounds of the subtree C) and $R_3 = [7, 9]$ (all bounds between the subtree and the target position). The ranges $R_1 = [0, 3]$ and $R_4 = [10, \infty]$ do not take part in the swap and are not altered. The swap consists of translating all bounds in the range R_2 by $+2$ and all bounds in R_3 by -4. By storing only these translations, we achieve $\mathcal{O}(c)$ space complexity. Node insertion and subtree deletion are similar: The left side of the figure shows the insertion of a new node F as rightmost child of node B. The dashes around F depict that it is outside of the active region. Again, this insertion is accomplished by swapping regions $R_2 = [6, 9]$ and $R_3 = [10, 11]$ and incrementing the max value of the resulting version by $+2$ because a new node was added to the active region. The right side of the figure shows how F is deleted by swapping $R_2 = [6, 7]$ and $R_3 = [8, 11]$ and reducing max.

Formally, let $swap([a, b], [c, d])$ be the function that swaps the interval $[a, b]$ with the interval $[c, d]$ under the preconditions that $c = b{+}1$ (the intervals are adjacent and the second one is behind the first one), $a \leq b \wedge c \leq d$ (the intervals are well-formed, non-empty intervals). Let $relocate([x, y], z)$ be the function that inserts the non-empty interval $[x, y]$ before z under the precondition that $z \notin [x, y]$. The function *relocate* is implemented through a *swap*:

$$relocate([x,y],z) = \begin{cases} swap([z, x-1],[x,y]), & \text{if } z < x \\ swap([x,y],[y+1, z-1]), & \text{otherwise} \end{cases}$$

Using *relocate* and the active region approach, implementing all update operations is straightforward:

- *insertBefore*(b) :
  - *relocate*$([\max{+}1, \max{+}2], b)$
  - max := max $+2$
- *moveSiblingRangeBefore*(v, v', b) :
  - *relocate*$([v.lower, v'.upper], b)$
- *deleteSiblingRange*(v, v') :
  - *relocate*$([v.lower, v'.upper], \max{+}1)$
  - max := max $-(v'.upper - v.lower + 1)$

Since all update operations are now reduced to *swap*, updating a delta solely relies on an efficient implementation of this function. An efficient approach for implementing *swap* for our delta representation will be given in Section 5.2.

We represent version deltas compactly as the ordered set of all translation ranges that were introduced by updates that happened between the source and the target version (which is comparable to the XID-map approach used by Xyleme [21]). The ranges are represented by storing the value of their lower borders in the source and the target space. The value of the translation is computing by subtracting the source from target value. Because the translation ranges are densely arranged next to each other, it is sufficient to store only the lower borders of the ranges. The upper border can be inferred by looking up the lower border of the successive range and subtracting 1. The highest range is unbounded, i. e., its upper border is the positive infinity. Figure 3 illustrates how our approach represents the delta resulting from the node insertion depicted on the left of Figure 2. The vertical bars represent the lower borders of the translation ranges and the arrows depict to which position these borders are translated. An update introduces at most three new translation ranges: The two ranges R_2 and R_3 that are swapped and the range R_4 behind them. Since only the lower borders of translation ranges are stored, the range R_1 before the swapped ones has its upper border adjusted implicitly. We use the notation $R(s, t)$ to denote a translation range which maps from s in the source space to

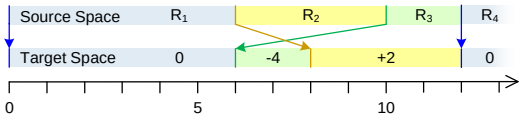


Figure 3: Model of translation ranges

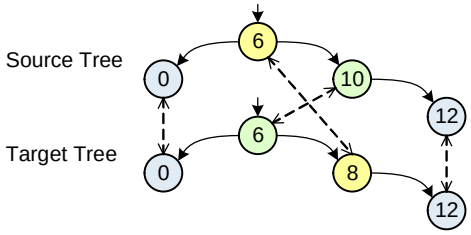


Figure 4: Representation of translation ranges

t in the target space. Thus, the delta depicted in Figure 3 is $\{R(0,0), R(6,8), R(10,6), R(12,12)\}$.

Using this representation, the delta function $\delta(b)$ is implemented as follows: Find the translation range $R(s,t)$ having the greatest s which is equal to or less than b and compute $\delta(b) = b+t-s$. In Figure 3, the bound 7 in the source space lies in $R_2 = R(6,8)$, so it is translated by $+8-6 = +2$, resulting in $\delta(7) = 9$. Note that this representation also allows to compute δ^{-1} similarly by applying the reverse translation. For example, the bound 6 in the target space lies in $R_3 = R(10,6)$. Therefore, $\delta^{-1}(6) = 6-(6-10) = 10$.

The representation shown in Figure 3 is only a conceptual model. A suitable data structure must allow the efficient computation of δ, δ^{-1}, and *swap*. Our implementation comprises two self-balancing binary search trees representing source and target space, called *source tree* and *target tree*. The keys in the trees are the lower borders of the translation ranges, and the payload is a pointer to the corresponding node in the other tree. Figure 4 shows the source and the target tree for the translation ranges from Figure 3.

Using the source/target tree representation, the implementation of $\delta(b)$ is straightforward: A usual search tree lookup in the source tree is used to find the translation range with the greatest lower border less or equal to b. By following the pointer to the corresponding node in the target tree and looking up its value there, the translation value is calculated. The implementation of $\delta^{-1}(b)$ is equally straightforward: Look up b in the target tree instead of the source tree and apply the negated translation value.

The size of the delta is in $\mathcal{O}(c)$ but is also bounded by the size of the hierarchy: The largest possible delta contains one translation range for each bound of the hierarchy. Note that repeated updates of a node or subtree (e. g., moving a tree around twice) do not create extra translation ranges but only update existing ones.

5. OBTAINING DELTAS

We have shown an approach for storing version deltas by representing translation ranges as nodes in two search trees which are linked with each other. The remaining challenge is to build this data structure efficiently. There are different possible scenarios for building a delta: One is that the source and the target version are available as usual NI encodings and the delta is to be inferred from them. A more dynamic scenario consists of building the delta incrementally: Whenever an update is performed on the hierarchy, the resulting *swap* is performed on the data structure. Handling this scenario efficiently requires specially augmented search trees.

5.1 Static Scenario

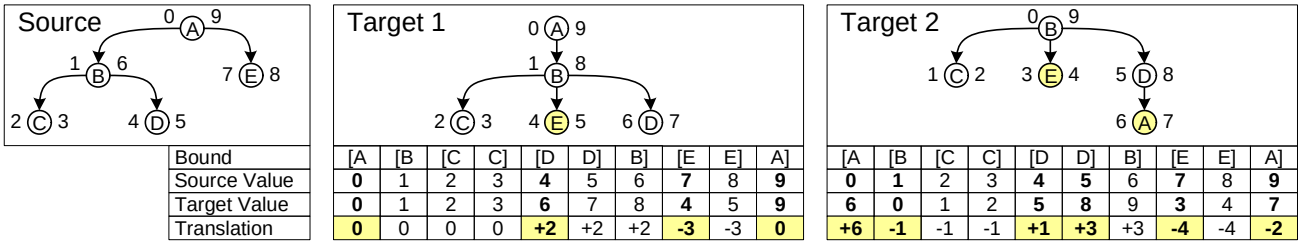
In this scenario we assume that the source and the target version for which to build a delta are available as NI encodings. This could be the case in applications where a user fetches a version from the database, edits it with a third-party program (e. g., a graphical tree editor) and then saves the result back to the database creating a new version. Another use case would be the periodic gathering of snapshots from the web [21]. The operations performed on the hierarchy are *not* known in this scenario, only the resulting NI encoding is available or is constructed on the fly. A matching of nodes must be available; such a matching is either implicit if the nodes carry unique identifiers (as in our example, and in many other use cases [6]), or a diff algorithm (e. g., [11]) must be used to match nodes in the two versions.

The algorithm for inferring the delta δ between two given hierarchy versions V and V' is as follows: Initialize t' with 0 and insert $R(0,0)$ into δ. Traverse V depth-first in pre/post order: For each node v, consider its lower bound before visiting its child nodes and its upper bound after visiting its child nodes. For each considered bound b, find the corresponding bound b' in V' by looking up the node v' that matches node v and retrieving its corresponding bound b'. Compute the translation t by subtracting b' from b. If $t \neq t'$, then the translation value has changed. Consequently, insert a new translation range $R(b,b')$ into δ. Set $t' = t$ and traverse the next bound until all bounds have been traversed.

Figure 5 shows the result of the algorithm comparing a source hierarchy (left) with two target hierarchies. The lower and upper bounds belonging to each node are displayed to its left and right, respectively. The middle of the figure shows a target hierarchy where only one update has occurred (node E was moved) while the right side shows a target with more updates. The table on the bottom of the figure shows the bounds which are traversed ([X denotes the *lower* bound of node X and X] the *upper* bound), their values in the source and target space, and the resulting translations. The delta is inferred by inserting a range for the first column and for each other column in which the translation value is different to the value of the previous column (highlighted in the figure). Thus, the resulting delta for the target hierarchy in the middle contains the four translation ranges $\{R(0,0), R(4,6), R(7,4), R(9,9)\}$. The right side of the figure shows a target hierarchy where more changes were introduced. Consequently, there are also more translation ranges (six) in the resulting delta.

5.2 Dynamic Scenario

The previous section introduced a scheme which bulk-builds a delta from two fully-materialized interval encodings. However, this approach requires that the full interval encoding of the target version is present, has a time complexity linear in the size of the hierarchy and cannot handle updates directly. It would be more appropriate if a version delta could directly be updated efficiently without having to infer any explicit NI encodings. As mentioned in Section 3, we model each atomic update by a swap of two consecutive bound intervals. Thus, an efficient update mechanism must perform this swap efficiently. The operation $swap(R_2 = [a,b], R_3 = [c,d])$ for a



Bound	[A	[B	[C	C]	[D	D]	B]	[E	E]	A]
Source Value	**0**	1	2	3	**4**	5	6	**7**	8	**9**
Target Value	**0**	1	2	3	**6**	7	8	**4**	5	**9**
Translation	**0**	0	0	0	**+2**	+2	+2	**-3**	-3	**0**

	[A	[B	[C	C]	[D	D]	B]	[E	E]	A]
Source Value	**0**	**1**	2	3	**4**	**5**	6	**7**	8	**9**
Target Value	**6**	**0**	1	2	**5**	**8**	9	**3**	4	**7**
Translation	**+6**	**-1**	-1	-1	**+1**	**+3**	+3	**-4**	-4	**-2**

Figure 5: Inferring deltas from source and target interval encoding

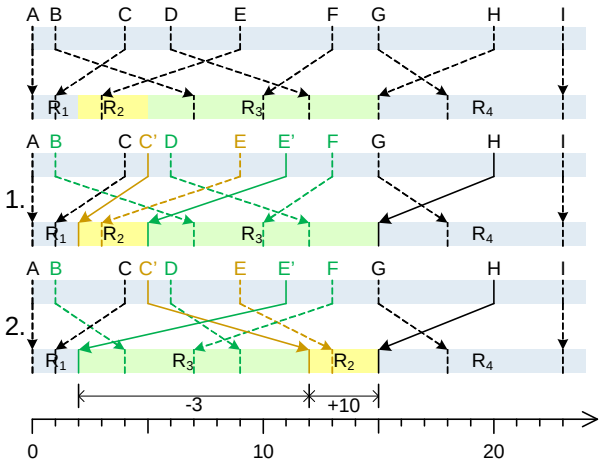


Figure 6: Updating a version delta by swapping translation range R_2 with R_3

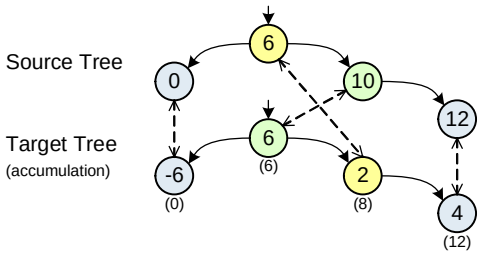


Figure 7: Using the accumulation tree as target tree

delta δ performs the swap in the target space. The bounds a, b, c, d are given as target space coordinates. Conceptually, the operation is implemented as follows:

1. Insert the lower borders of the swapped ranges $R_2 = R(\delta^{-1}(a), a)$ and $R_3 = R(\delta^{-1}(c), c)$, and of the range behind R_3 which is $R_4 = R(\delta^{-1}(d+1), d+1)$. If any of the ranges already exists, do not insert it again.
2. For all translation ranges $R(s,t)$ in δ with $t \in [a,b]$, translate t by the size of $[c,d]$ (i. e., by $d-c+1$). For all translation ranges $R(s,t)$ with $t \in [c,d]$, translate t backwards by the size of $[a,b]$.

The top of Figure 6 depicts a delta in which the ranges $R_2 = [2,4]$ and $R_3 = [5,14]$ are to be swapped. The delta already contains nine translation ranges $(A, ..., I)$. The middle of the figure shows the result after performing the first step of the algorithm: $C' = R(5,2)$, which is the lower border of R_2, and $E' = R(11,5)$, which is the lower border of R_3, are inserted. The lower border $H = R(20,15)$ of the range R_4 is already included in the delta and is reused. The bottom of the figure shows the delta after performing the second step of the algorithm: The target values of ranges that lie in R_2 in the target space are translated by +10 and the target values of those in R_3 are translated by -3.

The implementation of *swap* must adjust the target values of all borders in R_2 (marked orange in the figure) and R_3 (marked green). Since the target values are also keys in a search tree, the nodes in that tree also have to be rearranged to reflect the swap. On average, this results in a number of adjustments linear to the number n of ranges in the delta, which has an infeasible runtime if done naïvely. The swapping of nodes in the search tree by naïve deletion and reinsertion would even yield $\mathcal{O}(n \log n)$ time complexity.

To allow efficient updates of an interval delta in $\mathcal{O}(\log n)$, the search tree which models the target space has to be augmented to allow adjusting multiple keys at once and swapping ranges of search tree nodes efficiently.

Swapping Node Ranges: Split and Join. The efficient swapping of nodes can be accomplished by adding the *split* and *join* functionality to the self-balancing search tree: The *split*(T,k) function splits the search tree T before a key k, resulting in a tree that holds all keys $< k$ and one that holds all keys $\geq k$. Both resulting trees must be appropriately balanced. Given two search trees T_1 and T_2 where all keys in T_2 are greater than all keys in T_1, the *join*(T_1, T_2) function concatenates the trees, resulting in a new balanced tree which contains all their keys. Although both functions are quite uncommon since they are not needed by usual tree indexes, $\mathcal{O}(\log n)$ implementations exist for most common self-balancing search trees. We can swap two ranges of search tree nodes by splitting the tree at the borders of these ranges and then joining the resulting trees in a different order. One can imagine this as simply cutting the tree into smaller trees representing the different ranges and then gluing them together in the desired order. Such a swap consists of three splits and three joins and is therefore in $\mathcal{O}(\log n)$.

Adjusting Multiple Keys: Accumulation Tree. We achieve the adjustment of a key range in $\mathcal{O}(\log n)$ by replacing the ordinary search tree with a slightly adapted implementation which we call *accumulation tree*. An accumulation tree is a search tree in which each node only stores a part of its own key. The real key of a node v is obtained by adding (accumulating) all values on the path from v to the root. Since a search tree already traverses this path during the key lookup, the accumulation of the key of v is cheap. Figure 7 shows the delta from Figure 4 with an accumulation tree used as target tree. The resulting accumulated values (which are equal to the values of the original target tree in Figure 4) are shown in parenthesis below the nodes. For example, the rightmost node has a value of 12. This value is obtained by accumulating all values (6,2,4) on the path from the root.

Although the idea behind the accumulation tree is quite simple, it yields an important improvement: *All* keys in a

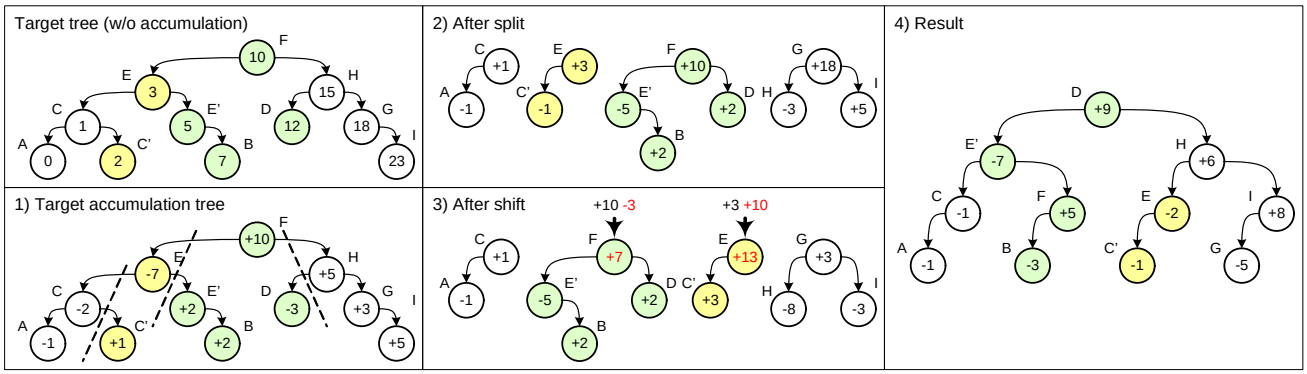


Figure 9: Steps of performing the *swap* operation on the (accumulation) target tree.

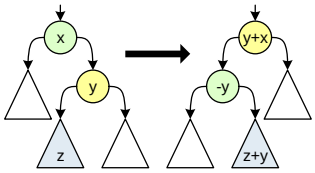


Figure 8: Left rotation in the accumulation tree

subtree rooted at a node v can be translated by simply adjusting the value of v, resulting in a time complexity which is constant instead of linear in the size of the subtree. However, the tree introduces a small maintenance overhead: Whenever performing rotations to restore the balance of the tree, the values of the two rotated nodes and the value of the root of the "middle" sub-tree below the nodes have to be adjusted. Otherwise, the rotation would alter the accumulated values. Figure 8 depicts the rules for updating the values after a left rotation. For example, the root of the subtree in the middle has $x + y + z$ as accumulated value before the rotation. Afterwards, it still has $(y + x) + (-y) + (z + y) = x + y + z$. Right rotation is similar.

Implementing *swap*. The first (simple) step of the algorithm consists of adding the borders of the swapped ranges. The second step of performing the swap from Figure 6 is depicted in Figure 9. On the top left of the figure, the target tree without accumulation is shown. The source tree is omitted, as it is not altered by a swap, except that range borders are added at appropriate positions. The lower left part of the figure (Step 1) shows the tree from the top, but now using accumulations. The dashed lines represent the positions where the tree is split. Step 2 of the figure shows the resulting trees after the splits are performed. Note that the split also rebalances the resulting trees. The next step (3) is to apply the translations to the two ranges. The accumulation tree allows this operation by simply adjusting the value in the root. The root of R_3 (F) is translated by -3 and the root of R_2 (E) is translated by $+10$. Finally, the trees are joined in the order C, F, E, G to yield the resulting tree, which is shown on the right of the figure. Since the time complexity of split and join is in $\mathcal{O}(\log n)$ and the complexity of the translation in the accumulation tree is in $\mathcal{O}(1)$, the resulting time complexity for the *swap* operation is $\mathcal{O}(\log n)$. As any kind of update is reduced to this operation, the index can execute all proposed updates in logarithmic time.

6. DELTA VERSION HISTORIES

A delta maps interval bounds from a version V to another version V' and vice versa. Now assume a large version history with n versions $V_0, \ldots, V_{n-1}$. To be able to answer queries for an arbitrary version V_i, one or more deltas must exist which eventually lead from a base version to V_i. We will now show how to efficiently build, manage, and query all deltas necessary for a complete history.

Without loss of generality, we will hereinafter assume a linear version history without any branches and with only one base version which is the eldest version V_0. The version indexes are sorted by the age of the version, so V_i is the version right before V_{i+1} and right after V_{i-1}. We define the *size* of a delta, written as $|\delta|$, as the number of versions covered by it. For example, the delta $\delta_{V_{20} \mapsto V_{30}}$ would have the size 10, because it covers all changes introduced in the ten versions versions $V_{21}, \ldots, V_{30}$. If a constant number of changes per version is assumed, the memory consumption of the delta is proportional to its size.

6.1 Querying the History

The interval bounds of each node are materialized for the base version V_0. Deltas are used to transform these bounds into any other version. The bounds in V_0 serve as durable identifiers for all the nodes, since they never change.

Let $\delta_1, \ldots, \delta_m$ be a sequence of deltas where each delta δ_i maps from a version to the version of the subsequent delta δ_{i+1}. If the first delta δ_1 maps from V_s and the last delta δ_m maps to V_t, then we can retrieve the bound b_t in V_t for a bound b_s in V_s by applying all deltas in the sequence:

$$b_t = \delta_m(\delta_{m-1}(\ldots \delta_2(\delta_1(b_s)) \ldots))$$

By applying the inverse deltas in the reverse order, we can also map back from V_t to V_s. By mapping a bound back to V_0, we can look up the node corresponding to that bound.

Assuming a constant number of changes per version, the time complexity of such a query is in $\mathcal{O}(m)$. So, for fastest query times, a sequence length of 1 would be best. This, however, implies that a delta from a base version to each other version must exist, resembling a star topology. In a linear version history, a change introduced in a version V_i will also be stored in the interval deltas for all versions which are more recent than V_i. When assuming a constant number of changes per version, maintaining deltas from the base version to each other version would require $\mathcal{O}(m^2)$ space in the worst and best case, because each change is contained in $m/2$ deltas on average. This is not feasible for hierarchies with millions of versions. Another extreme would be to store only the deltas from version V_i to V_{i+1}. Assuming a constant number of changes per version would yield $\mathcal{O}(m)$ space complexity, because each change is only stored

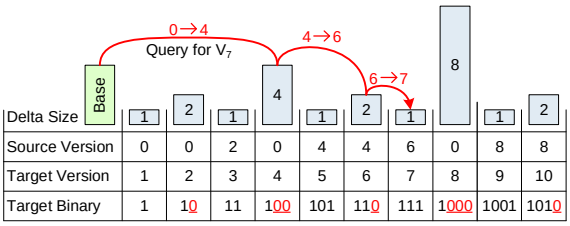


Figure 10: Using the number of trailing zeros in the binary representation of the target version for deciding delta sizes.

in the delta of the version in which it was introduced. This is the strategy with the least space consumption. However, a query in version V_i would then require i delta applications since all deltas of versions older than V_i have to be applied one by one. On average, this yields $\mathcal{O}(m)$ query complexity which is infeasible for large hierarchies, as well.

6.2 Exponential Deltas

We achieve a good space/time trade-off by enforcing an exponential distribution of the delta sizes. That is, some few deltas cover huge version ranges while most deltas cover only a few versions. The large deltas can be used to get "near" the target version quickly. Then, the small deltas are used to get exactly to the target version. This approach is comparable to the one of skip lists or to the finger tables in the Chord [30] peer-to-peer protocol.

Our approach uses the number of trailing zeros in the binary representation of the id of a version to determine the size of the delta leading to this version. Precisely, given a version V_i, the size of the delta δ which has V_i as target version is calculated as $|\delta| = 2^{tz(i)}$, where $tz(i)$ is the number of trailing zeros in the binary representation of i. For example, version 27 has the binary representation 11011_2. Since this binary string has no trailing zeros, this version will be represented by the delta $\delta_{V_{26}\to V_{27}}$, which has a size of 1. In contrast, version 36 corresponds to the binary string 100100_2, which has two trailing zeros. This results in the delta $\delta_{V_{32}\to V_{36}}$ of size $2^2 = 4$. Figure 10 depicts the size of the first ten interval deltas of a version history.

To query a version V_i using this technique, one has to start at the base version and execute "hops" which become smaller and smaller. The red arrow in Figure 10 shows how a query for version 7 is processed. The algorithm for finding the hops for version V_i simply consists of scanning the bit positions j of the binary representation of i from most-significant bit to least-significant bit. Whenever a 1 bit is found at position j, take one hop. The target version is i with all less significant bits than j zeroed out. For example, a query in version $i = 19 = 10011_2$ would be processed as follows: The highest 1 bit is $j = 4$ (j is counted from least- to most-significant bit, starting with zero for the least significant one), so the first hop is to version $10000_2 = 16$. The next one is at $j = 1$, resulting in the hop to $10010_2 = 18$. The final hop for the last 1 bit at $j = 1$ is 10011_2 which reaches the target version 19. The resulting deltas to be applied are $V_0 \to V_{16}$, $V_{16} \to V_{18}$, and $V_{18} \to V_{19}$.

Since the algorithm takes one hop per 1 bit of the version id i and version id bit lengths are logarithmic in the number of versions, the number of deltas to be applied to reach a version V_i is $\lceil \log_2(i) \rceil$ in the worst case (when the version id consists only of 1 bits) and 1 in the best case (when the version id is a power of 2). When maintaining a version history of n versions with n being a power of 2, each bit of a randomly chosen version id i is one or zero with the same probability, so the algorithm applies $\log_2(n)/2$ deltas on average.

A change introduced in version V_i is contained in the version delta for V_i and all version deltas of higher versions V_j where j is a power of 2. For example, a change introduced in version 7 is contained in the deltas $V_6 \to V_7$, $V_0 \to V_8$, $V_0 \to V_{16}$, $V_0 \to V_{32}$, and so on. Obviously, for a version history of n versions, there are logarithmically many versions which are a power of 2, so each change is contained in at most $1 + \lceil \log_2 n \rceil$ versions. Since one change needs a constant amount of space, a version history with n versions and constant number of changes per version can be stored using $\mathcal{O}(n \log n)$ space ($\mathcal{O}(n)$ changes in total, each being stored in $\mathcal{O}(\log n)$ versions).

As already shown, applying a delta of size s to a single bound has a time complexity of $\mathcal{O}(\log s)$. However, the computation of the value of a bound b in a version V_x usually needs to apply more than one delta. In the worst case, when the binary representation of x has only 1 bits in it, the algorithm must apply $\log_2 x$ deltas. The last delta covers one version and the number of covered versions doubles with each further delta, so the i-th delta covers 2^i versions. If we assume a constantly bounded number of changes per version, then the complexity of applying a delta covering n versions is $\mathcal{O}(\log n)$. Consequently, the complexity of applying all required deltas for reaching V_x is $\mathcal{O}(\sum_{i=0}^{\log_2 x} \log 2^i) = \mathcal{O}(\sum_{i=0}^{\log_2 x} i) = \mathcal{O}((\log_2 x)(1 + \log_2 x)/2) = \mathcal{O}(\log^2 x)$ in the worst case. In the best case, the version number is a power of 2 and only one delta has to be applied, yielding $\mathcal{O}(\log x)$.

Merging Deltas. During the generation of the exponential deltas, smaller deltas have to be merged to yield larger ones. For example, the delta $V_0 \to V_8$ is to be built by first merging the deltas $V_0 \to V_4$, $V_4 \to V_6$, and $V_6 \to V_7$, which yields the delta $V_0 \to V_7$. Now, there are two equally applicable strategies: One strategy is to apply the incoming changes for V_8 directly to the delta $V_0 \to V_7$, yielding the delta $V_0 \to V_8$ without further merges. Another strategy is to gather the changes for V_8 in a small delta $V_7 \to V_8$ and finally merge $V_0 \to V_7$ with $V_7 \to V_8$ to yield the final delta $V_0 \to V_8$. Regardless of the strategy used, an operation for merging two deltas is required.

Let, $\delta_{V\to V'}$ and $\delta_{V'\to V''}$ be two deltas which are connected via the version V', i. e., V' is the source of the one and the target of the respective other delta. We define the operation $merge(\delta_{V\to V'}, \delta_{V'\to V''})$ which merges the changes in the two deltas yielding the delta $\delta_{V\to V''}$. The resulting delta function must be the composition $\delta_{V\to V'} \circ \delta_{V'\to V''}$, i. e.:

$$\forall b \in \mathbb{N} \,.\, \delta_{V\to V''}(b) = \delta_{V'\to V''}(\delta_{V\to V'}(b))$$

The $merge(\delta_1, \delta_2)$ function can be implemented as follows: Start with an empty delta δ. For each translation range $R(s,t)$ in δ_1, compute $t' = \delta_2(t)$ and insert $R(s,t')$ into δ. Next, for each translation range $R(s,t)$ in δ_2, compute $s' = \delta_1^{-1}(s)$. If no translation rule with source value s' exists in δ, then add $R(s',t)$ to δ.

The implementation adjusts all translation ranges in the two deltas to incorporate the changes of the other delta, as well. Ranges in the prior delta δ_1 need their target values adjusted by δ_2, since the resulting delta maps to the target space of δ_2. The source values of the ranges in δ_2 need to

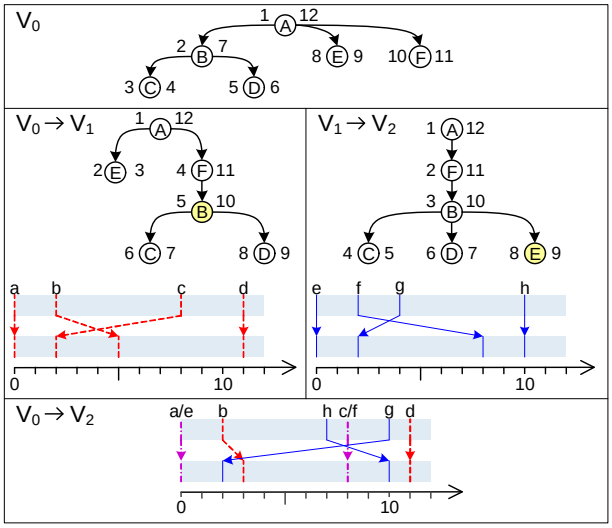


Figure 11: Merging two deltas.

be adjusted "backwards" by the inverse of δ_1, because the resulting delta maps from the source space of δ_1. Since each range in δ_1 and δ_2 adds at most one translation range to the resulting delta, the delta size $|\delta|$ is at most $|\delta_1| + |\delta_2|$. When the ranges of δ_2 are processed, they are only added if no delta with the same source value already exists. Thus, the resulting delta size may be smaller than $|\delta_1| + |\delta_2|$. A range is omitted if both versions transform the range. For example, if δ_1 moves a node X and δ_2 moves the same node again, then they will both contain a range starting at the lower bound of X. The resulting delta will only contain one rule for this lower bound.

Figure 11 shows an example for a merge. The source version V_0 is shown on top. In version V_1 (left), the subtree B was moved below F. In V_2 (right), node E was moved below B. The deltas δ_1 ($V_0 \to V_1$) and δ_2 ($V_1 \to V_2$) are displayed below the respective tree. A merge of these deltas results in the delta $V_0 \to V_2$ which is shown on the bottom of the figure. The letters a to h show which translation ranges in the resulting delta originate from which ranges in the source deltas. For example, the leftmost translation rule $R(0,0)$ is contained in both deltas as a and e, respectively, and is merged into one range a/e. Another example is the range c/f. The first delta has the rule c which is $R(8,2)$. When applying $\delta_2(2)$, the resulting target value is 8 (2 lies in range f which is translated by $+6$), so the resulting rule is $R(8,8)$ which is c/f. The second delta contains the rule f which is $R(2,8)$. The resulting source value for this rule is $\delta_1^{-1}(2) = 8$ (2 in the target space of δ_1 lies in rule c which has a translation of -6, so the inverse translation is $+6$). Since $R(8,8)$ already exists, no further range is added.

Since each translation range in the deltas has to be processed (linear) and for each range, a delta must be computed (log) and a range must be inserted (log), the resulting time complexity of the merge operation is $\mathcal{O}(n \log n)$, where n is the number of ranges in the merged deltas.

6.3 Optimizations

Query Routing. Merging two deltas takes $\mathcal{O}(n \log n)$ time and building a delta of size s requires $\log s$ merges, so building a large delta may take some seconds for large histories. Consequently, if a large delta for a version V_x is currently being built and a query is issued in V_x at that time, then a stall in the query processing is to be anticipated. To prevent this stall, the query is routed over the partial deltas as long as the merged delta is not fully built yet. Hence, *all* merging can be done in the background and no stall is to be anticipated for any query, even if the deltas for that query are not yet fully merged. As a delta is not used before it is thoroughly merged, multiple merges can even be executed concurrently by multiple low-priority background threads without any locking necessary. By using this optimization, the merging process has no influence on the query performance and queries can be answered efficiently in all versions, regardless of the time the delta merging takes.

Epochs. Until now, we have only considered a single base version, yielding $\mathcal{O}(\log^2 n)$ worst-case query complexity and $\mathcal{O}(n \log n)$ space consumption for a history with n versions. From time to time, we can fully materialize the NI encoding of a version, thus creating a new base version. Subsequent versions then start a new exponentially distributed delta history. We refer to a base version with its following delta history as *epoch*. With multiple epochs, a query in version V is executed by first finding the epoch E of V and then starting the hops from the base version of E. If we start a new epoch regularly after x versions, then we can find the epoch of a version with id u by simply calculating u/x. Additionally, each epoch only covers a constant number of deltas. Thus, the asymptotic query complexity becomes $\mathcal{O}(1)$ and the space consumption becomes $\mathcal{O}(n)$, assuming that the hierarchy size and the number of changes per version stay constantly bounded in all versions.

To achieve a reasonable space/time tradeoff, an epoch should not be created too often. As a rule of thumb, if a new delta would be larger than the materialization of a version, then a new epoch should be created. The figure to consider for determining the epoch length is $\frac{\textit{change frequency}}{\textit{hierarchy size}}$: The more changes are to be anticipated, the quicker the space consumption of deltas grows. The larger the hierarchy, the more memory a new base version consumes.

Again, the query routing optimization can be used: The materialization is done in the background and deltas are used for queries as long as the materialization is not finished. This prevents any possible stalls.

Note that epochs can also be used for efficient vacuuming of old versions, as an epoch is a self-contained piece of the version history. Hence, old epochs can easily be archived to disk or discarded to reduce memory consumption.

Static Deltas. All deltas but the latest one are static. Therefore, all these deltas do not need a data structure that supports the *swap* operation and can instead be represented by a read-optimized or even a read-only data structure such as an implicit complete binary tree.

7. EVALUATION

Baseline. To assess the performance of DeltaNI in comparison to other sophisticated versioned indexing schemes, we built a baseline consisting of a state-of-the-art labeling scheme backed up by a versioned index. For the labeling scheme, we chose the prominent path-based scheme ORDPATH [23] as it shows very good performance and low space consumption in comparison to other path-based schemes [28] and is practically used, for example, in the XML engine of Microsoft SQL Server. For the versioning, we chose the asymptotically optimal multiversion B-

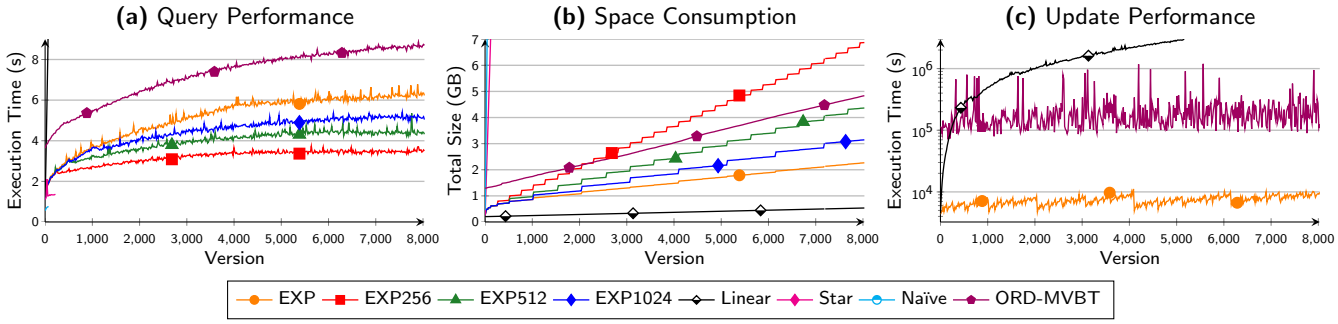


Figure 12: Time for executing one million queries (a), space consumption (b), and time for executing one million updates (c).

tree (MVBT) [2]. The combination of ORDPATH and the MVBT, which we call ORD-MVBT, is comparable to the data structure used in the MVBT-Twigstack approach of Woss and Tsotras [33]. By indexing the tuples and their ORDPATHs with the MVBT, the index is able to efficiently answer various kinds of queries in all former versions of the hierarchy and thus yields a promising baseline implementation to compare DeltaNI against.

Test Setup. The evaluation is based on a dataset derived from an EA hierarchy of a mechanical engineering company. Our obtained history starts with a snapshot containing 2.9 million nodes with an average depth of around 7 and a maximum depth of 16. The hierarchy is versioned on a daily basis for 22 years from 1990 to 2012 resulting in 8035 versions. The average number of updates per day is 638 and hence 5.1 million updates for the whole history. 36% of the updates are inserts, 35% are removes and 31% are subtree relocations. The size of the relocated trees is 8 nodes on average. To measure query performance, we chose the check whether two nodes lie on an XPath axis (ancestor, following, preceding, descendant) as query primitive for the following reasons: 1) The primitive is used for many important queries such as axis steps. 2) The query suites range-based (DeltaNI) and path-based (ORD-MVBT) schemes equally well in contrast to other queries which inherently favor either range-based (e. g., subtree size) or path-based (e. g., node level) ones. 3) The query assesses a recursive property of the hierarchy which is hard to evaluate with recursive SQL.

After each version V_i is built, the query is executed repeatedly for randomly chosen versions $V_{0..i}$ to measure the query performance in relation to the history length. We compare the baseline (ORD-MVBT) to DeltaNI with exponential hops without epochs (EXP) and with epochs of length 256 (EXP256), 512 (EXP512), and 1024 (EXP1024). We further measure the performance of the naïve delta grouping schemes which are the linear scheme (Linear), i. e., one delta between each successive version V_i and V_{i+1}, and the star scheme (Star), i. e., deltas from the base version V_0 to each other version. Finally, we also measure the naïve approach of materializing each version thoroughly (Naïve).

All tests were carried out on a HP Z600 Workstation with a 6-core Intel Xeon X5650 CPU at 2.66 GHz, 12 MB cache, and 24 GB RAM. The operating system is SuSE Linux Enterprise Server, kernel version 2.6.32.

Query Performance. Figure 12(a) shows the time in seconds to answer one million queries. The Linear scheme is obviously infeasible, while the other naïve schemes show good query performance but were aborted at version 88 (Naïve) and 231 (Star) since the test machine ran out of memory. Thus, they are infeasible for this hierarchy. The EXP scheme requires around 6.3 seconds in the final version (resulting in around 160,000 queries per second). The schemes with epochs are faster: The one with the most epochs—EXP256—is almost twice as fast as EXP (around 285,000 queries per second). Fewer epochs lead to longer histories thus decreasing performance. ORD-MVBT requires around 40% more time than EXP and around 150% more time than EXP256. This is due to the facts that 1) the MVB-tree also contains dead entries in its nodes while deltas only contain entries relevant for their version 2) integer comparisons in the deltas are faster than ORDPATH comparisons 3) B-tree variants such as the MVBT are optimized for fixed-size keys and require additional overhead for variable-sized keys such as ORDPATH. Obviously, specialized indexes like DeltaNI and ORD-MVBT outperform recursive SQL by orders of magnitude, as previously shown by Al-Khalifa et al. [1].

Memory Consumption. The memory consumption is shown in Figure 12(b). EXP uses 2.2 GB for the whole history. The schemes with Epochs use more memory. EXP256 uses three times as much memory (6.8 GB) due to the large number of fully-materialized versions, which are visible as "staircases" in the diagram. Note however that the schemes with hops grow linearly, while the one without grows log-linearly: The little step in the line for EXP at version 4096 shows the large delta that is built by this approach. The larger the power of 2, the larger these steps in the graph of EXP will become, while the steps in the schemes with epochs keep their constant size. Using epochs is a clear space/time tradeoff in this scenario, since the shorter the epochs, the higher the query performance (cf. Figure 12). For other scenarios where the hierarchy size is smaller in comparison to the number of changes, long epochs can even yield an advantage in time *and* space.

ORD-MVBT requires 4.8 GB of memory for the whole history. The fact that the line already starts at 1.4 GB is due to ORD-MVBT having no notion of a base version and thus needing to insert the 2.9 million starting nodes into the MVBT while DeltaNI can store them in a base version compactly. For a scenario that starts with an empty base version, in which the memory consumption of both approaches starts at zero, the memory consumption of ORD-MVBT is between the EXP512 and EXP256. Thus, the memory consumption of the two index structures is similar and varies only slightly for different use cases.

Update Performance. The time needed for one million updates is shown in Figure 12(c). The only exponen-

tial scheme displayed is EXP, since all exponential schemes have similar update behaviour (around 200,000 updates per second). This is because epochs are only created during the creation of a new version and thus have no influence on update performance. DeltaNI outperforms ORD-MVBT (around 10,000 updates per second) by around a factor of 20 since ORD-MVBT has to handle subtree relocations by repeated removes and inserts. We also measured the update performance for a synthetic dataset without relocations (not in the figure). Here, ORD-MVBT reaches around 35,000 updates which is still almost an order of magnitude less than DeltaNI, since the *swap* operation is extremely efficient compared to a MVBT insertion or deletion which has to compare various ORDPATHs to find the leaf to insert the new entry.

Besides the time consumption of the update operations, DeltaNI incurs an additional cost when creating a new version, which we also measured: The larger the power of 2, the longer the merging of the resulting delta takes, as more deltas need to be merged. Full materialization for a new epoch takes around 5 to 10 seconds. The merging of the largest delta, which is the delta 4096 for EXP, takes 20 seconds, while most smaller deltas need less than a second (380 microseconds for deltas of size 1). Since there is only one new version per day, delta building performance is absolutely sufficient. As the build process can be done in parallel in the background while the queries are routed via partial deltas (query routing optimization), a server with multiple cores could even handle several new versions per second.

In conclusion, DeltaNI shows superior update performance and an acceptable memory consumption, enabling the storage of a history of decades in main memory. The query times promise to yield a tremendous speedup compared to relational approaches. DeltaNI outperforms other contemporary approaches such as the ORD-MVBT used in this benchmark and is especially suited if the workload consists of more complex updates such as subtree relocations.

8. RELATED WORK

Hierarchy Support in RDBMS. Current systems can query hierarchical data only via recursive SQL. Many systems also provide support for XML data which constitutes a hierarchy representation. While these systems support columns with XML data, i. e., one XML hierarchy per tuple, we aim for one hierarchy over the tuples of one or more tables. No RDBMS supports these kinds of hierarchies efficiently, yet. Thus, the functionality we integrate is complementary to the XML support of recent database systems.

Labeling Schemes. A lot of labeling schemes have been proposed for XML, which are usually categorized into prefix-based and range- (or order-) based schemes. Among the most prominent ones is ORDPATH [23] in the prefix-based category. In the range-based category, various schemes that are conceptually equal to the NI encoding, such as pre/post [14], pre/size/level [4], or order/size [19], are used. This is only a small selection of schemes; many more exist—each with its own advantages and drawbacks—but are omitted here for brevity reasons. Labeling schemes are widely used for indexing non-versioned hierarchical data and can also be used as building blocks for indexing versioned hierarchical data, as shown in this paper.

Versioned Index Structures. Many of the traditional tree indexes used in relational databases have been augmented to be used for versioned indexing: The fully persistent B^+-tree [18], the Time-Split B-tree [20], the BT-tree [16], and the multiversion B-tree [2] to name a few. Like labeling schemes, these indexes cannot be directly applied to versioned hierarchies. Instead, they form building blocks used by various versioning approaches discussed below.

Hierarchy Version Management. Versioning of XML data has been a hot topic in the last decade. However, most of the (especially earlier) contributions in this field are not concerned with indexing but rather the fast reconstruction of a version or the difference between versions. Consequently, the resulting data structures are not useful for efficient query support. Examples for this are the early contributions of Chien et al. [10] which focus on version management. They consequently compare their approach to text-based version control systems like SCCS and RCS. Rusu et al. [26, 27] propose and compare different delta storage techniques. Marian et al. [21] are concerned with version management in an XML Warehouse in the Xyleme project. Their concepts like the XID-map and their diff algorithm [11] are important for our contribution. They also evaluate different delta storage techniques. Rosado et al. [25] present a version management technique storing the version history of an XML document in an XML document, thus allowing queries using usual XML technology. Buneman et al. [6] propose an archiving technique for scientific XML data.

Versioned Hierarchy Indexing. Considering more tree-aware version control of XML data, the more recent contributions of Chien et al. [8, 9] introduce the SPaR versioning scheme which is basically an adapted NI encoding with gaps. It relies on "durable" labels, i. e., labels that do not change even if new nodes are inserted. As noted in many publications (e. g., [34, 29]), encodings with gaps are problematic, because frequent insertions at the same positions quickly fill up the gaps. This makes relabeling necessary again and thus invalidates the durable labels. The SPaR authors suggest to mitigate this problem by replacing the integer labels with floats of arbitrary precision. However such techniques yield labels with a size of $\mathcal{O}(n)$ bits (proven in [12]) resulting in high memory consumption, costly label comparisons, and the complication of index structures relying on keys of a fixed size, such as B-tree variants. Our scheme can efficiently handle any number of insertions or relocations at any position and yields a *gapless* fixed-size integer NI encoding with all its benefits. It also includes durable fixed-size node identifiers that never need to be relabeled.

Cursory ideas were presented in workshop publications of Vagena, Tsotras, et al. covering XML versioning with the PathStack join on an NI encoding in conjunction with a document map [32] (no branching) and a BT-ElementList [31] (allows branching). More recently, Woss and Tsotras [33] carry on with the topic now using the Twigstack join on an ORDPATH encoding in conjunction with the MVBT tree. This approach neither allows branching histories (due to the MVBT) nor subtree relocations (due to ORDPATH).

The concept of versioning is closely related to the concept of transaction time in temporal databases. Therefore, work from the field of temporal XML can be applied to versioned hierarchies (and vice versa). Rizzolo et al. [22, 24] propose a temporal XML index for efficient TXPath query evaluation. It is based on so-called continuous paths which are timestamp-augmented label paths. Unfortunately, label paths are not generally applicable to hierarchies, as these do

not necessarily possess labels. Zhang et al. [35] propose a labeling scheme for temporal SQL, which, however, relies on schema information that is not available for hierarchies.

In conclusion, most related work from the XML field is only partially applicable to versioned hierarchies in general. Another drawback of almost all aforementioned contributions is that subtree relocations are not supported. While such a relocation scenario may not be important for XML, it is indeed important for other hierarchies (especially for EA hierarchies). By supporting subtree and even range relocations efficiently, the DeltaNI index is widely applicable as a general-purpose hierarchy index.

9. CONCLUSION

In this paper, we proposed a technique for efficiently storing and indexing versioned hierarchical data. Our index yields a nested intervals encoding for each version by maintaining exponential deltas leading to each version with a logarithmic number of hops. The deltas represent changes in a space- and time-efficient manner by storing only the *translation ranges* that are introduced by updates. Such updates are executed on the deltas using a special-purpose accumulation search tree which is able to swap ranges of keys in logarithmic time. By reducing all update operations to a *swap* operation, our index facilitates even complex updates like subtree or sibling range relocation efficiently. By using *epochs*, the index can be tuned further. Our evaluation shows that the index is able to handle even large use cases with very long version histories efficiently and outperforms alternative approaches in a relevant use case. Consequently, our index is a worthy addition for relational databases that need to handle dynamic versioned hierarchical data. To our best knowledge, DeltaNI is currently the only contribution that yields a gapless fixed-size labeling for versioned trees while still facilitating complex updates. Possible future work consists of introducing hierarchical data support to relational database systems by using the index in conjunction with a query engine that can handle hierarchical data and relational data in one query. We are currently integrating the approach into the two main-memory database systems HyPer [17] and SAP HANA [13].

10. REFERENCES

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4), 1996.

[3] Boeing. 747 fun facts. `http://www.boeing.com/commercial/747family/pf/pf_facts.html`.

[4] P. Boncz, S. Manegold, and J. Rittinger. Updating the pre/post plane in MonetDB/XQuery. In *XIME-P*, 2005.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[6] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *TODS*, 29(1), 2004.

[7] J. Celko. *Trees & Hierarchy in SQL for Smarties*. Morgan Kaufmann, 2004.

[8] S. Chien, V. Tsotras, C. Zaniolo, and D. Zhang. Efficient complex query support for multiversion XML documents. *EDBT*, 2002.

[9] S. Chien, V. Tsotras, C. Zaniolo, and D. Zhang. Supporting complex queries on multiversion XML documents. *TOIT*, 6(1), 2006.

[10] S. Chien, V. J. Tsotras, and C. Zaniolo. XML document versioning. *SIGMOD Record*, 30(3), 2001.

[11] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE*, 2002.

[12] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. *SIAM Journal on Computing*, 39(5), 2010.

[13] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4), 2012.

[14] T. Grust. Accelerating XPath location steps. In *SIGMOD*, 2002.

[15] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *VLDB*, 2003.

[16] L. Jiang, B. Salzberg, D. Lomet, M. Barrena, et al. The BT-tree: A branched and temporal access method. In *VLDB*, 2000.

[17] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[18] S. Lanka and E. Mays. Fully persistent B+-trees. In *SIGMOD*, 1991.

[19] Q. Li, B. Moon, et al. Indexing and querying XML data for regular path expressions. In *VLDB*, 2001.

[20] D. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, 1989.

[21] A. Marian, S. Abiteboul, G. Cobena, L. Mignet, et al. Change-centric management of versions in an XML warehouse. In *VLDB*, 2001.

[22] A. Mendelzon, F. Rizzolo, and A. Vaisman. Indexing temporal XML documents. In *VLDB*, 2004.

[23] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: insert-friendly XML node labels. In *SIGMOD*, 2004.

[24] F. Rizzolo and A. Vaisman. Temporal XML: modeling, indexing, and query processing. *VLDB Journal*, 17(5), 2008.

[25] L. Rosado, A. Márquez, and J. González. Representing versions in XML documents using versionstamp. *ECDM*, 2006.

[26] L. Rusu, W. Rahayu, and D. Taniar. Maintaining versions of dynamic XML documents. *WISE*, 2005.

[27] L. Rusu, W. Rahayu, and D. Taniar. Storage techniques for multi-versioned XML documents. In *DASFAA*, 2008.

[28] V. Sans and D. Laurent. Prefix based numbering schemes for XML: techniques, applications and performances. *PVLDB*, 1(2), 2008.

[29] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In *ICDE*, 2005.

[30] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *TON*, 11(1), 2003.

[31] Z. Vagena, M. Moro, and V. Tsotras. Supporting branched versions on XML documents. In *RIDE*, 2004.

[32] Z. Vagena and V. Tsotras. Path-expression queries over multiversion XML documents. In *WebDB*, 2003.

[33] A. Woss and V. Tsotras. Experimental evaluation of query processing techniques over multiversion XML documents. In *WebDB*, 2009.

[34] L. Xu, T. Ling, H. Wu, and Z. Bao. DDE: from Dewey to a fully dynamic XML labeling scheme. In *SIGMOD*, 2009.

[35] Y. Zhang, X. Wang, and Y. Zhang. A labeling scheme for temporal XML. In *WISM*, 2009.