DeltaNI: An Efficient Labeling Scheme for Versioned Hierarchical Data

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

Technische Universität München

SAP AG
Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!
Enterprise-Resource-Planning (ERP) systems use a lot of **dynamic** hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)
Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)
- Asset hierarchies
  - 10 — 100 million nodes
  - A lot of subtree moves (50% or more)
Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)
- Asset hierarchies
  - 10 — 100 million nodes
  - A lot of subtree moves (50% or more)

- **Problem:** Current indexing approaches do not support subtree moves!
Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)

- Asset hierarchies
  - 10 — 100 million nodes
  - A lot of subtree moves (50% or more)

- Problem: Current indexing approaches do not support subtree moves!

- Challenge: Versioning required for accountability
Hierarchical Data

Hierarchical Relationship over tuples of a table

<table>
<thead>
<tr>
<th>Name</th>
<th>Boss</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>NULL</td>
<td>80,000</td>
</tr>
<tr>
<td>Bob</td>
<td>Adam</td>
<td>55,000</td>
</tr>
<tr>
<td>Celia</td>
<td>Adam</td>
<td>70,000</td>
</tr>
<tr>
<td>Dale</td>
<td>Celia</td>
<td>55,000</td>
</tr>
<tr>
<td>Eddy</td>
<td>Celia</td>
<td>45,000</td>
</tr>
<tr>
<td>Felicia</td>
<td>Adam</td>
<td>60,000</td>
</tr>
<tr>
<td>Gina</td>
<td>Felicia</td>
<td>75,000</td>
</tr>
<tr>
<td>Hana</td>
<td>Gina</td>
<td>45,000</td>
</tr>
</tbody>
</table>
Hierarchical Data

- Hierarchical Relationship over tuples of a table

<table>
<thead>
<tr>
<th>Name</th>
<th>Boss</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>NULL</td>
<td>80,000</td>
</tr>
<tr>
<td>Bob</td>
<td>Adam</td>
<td>55,000</td>
</tr>
<tr>
<td>Celia</td>
<td>Adam</td>
<td>70,000</td>
</tr>
<tr>
<td>Dale</td>
<td>Celia</td>
<td>55,000</td>
</tr>
<tr>
<td>Eddy</td>
<td>Celia</td>
<td>45,000</td>
</tr>
<tr>
<td>Felicia</td>
<td>Adam</td>
<td>60,000</td>
</tr>
<tr>
<td>Gina</td>
<td>Felicia</td>
<td>75,000</td>
</tr>
<tr>
<td>Hana</td>
<td>Gina</td>
<td>45,000</td>
</tr>
</tbody>
</table>

- Queries over structural properties, e.g., subtree
  
  ```
  SELECT name, salary FROM /Employee[name='Celia']/*
  ```
Hierarchical Data

- Hierarchical Relationship over tuples of a table

<table>
<thead>
<tr>
<th>Name</th>
<th>Boss</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>NULL</td>
<td>80,000</td>
</tr>
<tr>
<td>Bob</td>
<td>Adam</td>
<td>55,000</td>
</tr>
<tr>
<td>Celia</td>
<td>Adam</td>
<td>70,000</td>
</tr>
<tr>
<td>Dale</td>
<td>Celia</td>
<td>55,000</td>
</tr>
<tr>
<td>Eddy</td>
<td>Celia</td>
<td>45,000</td>
</tr>
<tr>
<td>Felicia</td>
<td>Adam</td>
<td>60,000</td>
</tr>
<tr>
<td>Gina</td>
<td>Felicia</td>
<td>75,000</td>
</tr>
<tr>
<td>Hana</td>
<td>Gina</td>
<td>45,000</td>
</tr>
</tbody>
</table>

- Queries over structural properties, e.g., subtree
  
  SELECT name, salary FROM /Employee[name='Celia']/*

- **Scope:** Index the hierarchy structure
Hierarchical Data

- Hierarchical Relationship over tuples of a table

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>80,000</td>
</tr>
<tr>
<td>Bob</td>
<td>55,000</td>
</tr>
<tr>
<td>Celia</td>
<td>70,000</td>
</tr>
<tr>
<td>Dale</td>
<td>55,000</td>
</tr>
<tr>
<td>Eddy</td>
<td>45,000</td>
</tr>
<tr>
<td>Felicia</td>
<td>60,000</td>
</tr>
<tr>
<td>Gina</td>
<td>75,000</td>
</tr>
<tr>
<td>Hana</td>
<td>45,000</td>
</tr>
</tbody>
</table>
```

- Queries over structural properties, e.g., subtree
  
  ```sql
  SELECT name, salary FROM /Employee[name='Celia']/; /*
  ```

- **Scope:** Index the hierarchy structure
Versioned Hierarchical Data

- Multiple versions of a hierarchy (1000+)
  - Updates at latest version create new version
  - Versioning of the table out of scope
  - Possibly branching history

Versioned Queries

```sql
SELECT name, salary FROM /Employee[name=‘Celia’]/* IN V2
```
Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).
Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

- Efficient queries in all versions
Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

- Efficient queries in all versions
- Low space consumption
  - Large hierarchies
  - Long histories
  - Main-memory database
Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

▶ Efficient queries in all versions
▶ Low space consumption
   ▶ Large hierarchies
   ▶ Long histories
   ▶ Main-memory database
▶ Efficient updates in latest version
   ▶ Insert, delete, and subtree move
Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

- Efficient queries in all versions
- Low space consumption
  - Large hierarchies
  - Long histories
  - Main-memory database
- Efficient updates in latest version
  - Insert, delete, and subtree move
- Allow branching histories
Indexing Hierarchies: Labeling Schemes

- Widely used hierarchy indexing: Labeling Schemes
Widely used hierarchy indexing: Labeling Schemes

- Each node carries fixed set of labels
- Queries can be answered by only considering labels
Widely used hierarchy indexing: Labeling Schemes

- Each node carries fixed set of labels
- Queries can be answered by only considering labels
- Widely applied in, e.g., XPath processing
- Examples: pre/post, ORDPATH, nested intervals (NI)
Widely used hierarchy indexing: Labeling Schemes

- Each node carries fixed set of labels
- Queries can be answered by only considering labels
- Widely applied in, e.g., XPath processing
- Examples: pre/post, ORDPATH, nested intervals (NI)
Widely used hierarchy indexing: Labeling Schemes

- Each node carries fixed set of labels
- Queries can be answered by only considering labels
- Widely applied in, e.g., XPath processing
- Examples: pre/post, ORDPATH, nested intervals (NI)
Widely used hierarchy indexing: Labeling Schemes

- Each node carries fixed set of labels
- Queries can be answered by only considering labels
- Widely applied in, e.g., XPath processing
- Examples: pre/post, ORDPATH, nested intervals (NI)
Widely used hierarchy indexing: Labeling Schemes

- Each node carries a fixed set of labels
- Queries can be answered by only considering labels
- Widely applied in, e.g., XPath processing
- Examples: pre/post, ORDPATH, nested intervals (NI)

/Employee[name="Celia"]/* ⇒ “All nodes in [3,8]”
Challenge 1: Efficient Query Support
Challenge 1: Efficient Query Support
- NI can be used to answer queries efficiently ✓
Challenges

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
Challenges

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓

- Challenge 2: Efficient Update Support
  - NI not dynamic (O(n) bounds change per update) 😞
Challenges

- **Challenge 1: Efficient Query Support**
  - NI can be used to answer queries efficiently ✓
- **Challenge 2: Efficient Update Support**
  - NI not dynamic ($O(n)$ bounds change per update) 😞
Challenges

- **Challenge 1: Efficient Query Support**
  - NI can be used to answer queries efficiently ✓

- **Challenge 2: Efficient Update Support**
  - NI not dynamic ($O(n)$ bounds change per update) 😞

$O(n)$ bound changes
Challenges

- **Challenge 1: Efficient Query Support**
  - NI can be used to answer queries efficiently ✓

- **Challenge 2: Efficient Update Support**
  - NI not dynamic ($O(n)$ bounds change per update) 😞

\( O(n) \) bound changes
Challenges

- **Challenge 1: Efficient Query Support**
  - NI can be used to answer queries efficiently ✓

- **Challenge 2: Efficient Update Support**
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) 😞

- **Challenge 3: Space Consumption of Histories**

$\mathcal{O}(n)$ bound changes
Challenges

- **Challenge 1: Efficient Query Support**
  - NI can be used to answer queries efficiently ✓

- **Challenge 2: Efficient Update Support**
  - NI not dynamic (\(\mathcal{O}(n)\) bounds change per update) 😞

- **Challenge 3: Space Consumption of Histories**
  - \(\mathcal{O}(n)\) bounds change per update need to be stored 😞

\(\mathcal{O}(n)\) bound changes
Observation: Each update can be represented by a swap of two ranges of bounds
Swaps

- Observation: Each update can be represented by a swap of two ranges of bounds
- Insert Node: Before

![Diagram showing swaps and range boundaries]
Observation: Each update can be represented by a swap of two ranges of bounds

Insert Node: After
Observation: Each update can be represented by a swap of two ranges of bounds

Delete Node: Before
Observation: Each update can be represented by a swap of two ranges of bounds

Delete Node: After
Observation: Each update can be represented by a swap of two ranges of bounds

Delete Subtree: Before
Observation: Each update can be represented by a swap of two ranges of bounds

Delete Subtree: After
Observation: Each update can be represented by a swap of two ranges of bounds

Move Subtree: Before
Observation: Each update can be represented by a swap of two ranges of bounds

Move Subtree: After
Storing Swaps

Observation: Each update can be represented by a swap of two ranges of bounds
Observation: Each update can be represented by a swap of two ranges of bounds

Idea: Simply store that swap instead of the changed bounds
Observation: Each update can be represented by a swap of two ranges of bounds

Idea: Simply store that swap instead of the changed bounds
Observation: Each update can be represented by a swap of two ranges of bounds

Idea: Simply store that swap instead of the changed bounds

![Diagram showing before and after swaps of ranges R1, R2, R3, R4 in the source space.](image-url)
Observation: Each update can be represented by a swap of two ranges of bounds

Idea: Simply store that swap instead of the changed bounds
Representing Swaps

- **Representation:** Two balanced (binary) search trees ("double tree")
- **Node content:** Lower border and link to other tree

![Diagram showing source and target trees with nodes 0, 6, 10, 12 and arrows indicating swaps between them. The source space is depicted as R1, R2, R3, R4 and the target space as 0, -4, +2, 0.](image_url)
The double tree represents a function $\delta : \mathbb{N} \mapsto \mathbb{N}$
The double tree represents a function $\delta : \mathbb{N} \leftrightarrow \mathbb{N}$.

$\delta$ maps interval bounds from source space to target space.
The double tree represents a function $\delta : \mathbb{N} \leftrightarrow \mathbb{N}$

$\delta$ maps interval bounds from source space to target space

Let $b$ be a bound in source space, then $\delta(b)$ is equivalent bound in target space
The double tree represents a function \( \delta : \mathbb{N} \mapsto \mathbb{N} \).

\( \delta \) maps interval bounds from source space to target space.

Let \( b \) be a bound in source space, then \( \delta(b) \) is equivalent bound in target space.

Given an NI encoding in version \( V_i \) and a delta \( \delta_{V_i \rightarrow V_j} \) from version \( V_i \) to another version \( V_j \), we can answer queries in \( V_j \).
Computing $\delta$ on the Double Tree

- Computing $\delta(b)$ is easy:

  - Find greatest node in source tree less than $b$.
  - Usual search-tree lookup.
  - Apply translation of that node.

  ![Diagram of source and target trees with arrows indicating the computation of $\delta(b)$](attachment:image.png)
Computing $\delta$ on the Double Tree

- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than $b$
    - Usual search-tree lookup

![Diagram showing source and target trees with nodes labeled and arrows indicating transformations.](image)

Source Tree

Target Tree
Computing $\delta$ on the Double Tree

- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than $b$
    - $\Rightarrow$ Usual search-tree lookup
  - Apply translation of that node
Computing $\delta$ on the Double Tree

- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than $b$
    - Usual search-tree lookup
  - Apply translation of that node

$\delta(11) = 7$: 

![Diagram of source and target trees with node translations marked]
Computing $\delta$ on the Double Tree

- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than $b$
    - Usual search-tree lookup
  - Apply translation of that node

$\delta(11) = 7$:

- Computation of $\delta^{-1}(b)$ similar
Double tree feasible?

Does the double tree delta solve the problems?
Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support

\[ n = \text{number of nodes, } c = \text{number of changes in delta} \]
Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✔

\[ n = \text{number of nodes}, \quad c = \text{number of changes in delta} \]
Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
  - Calculating $\delta(b)$ (search tree lookup) is in $O(\log c)$ ✓

$n = \text{number of nodes, } c = \text{number of changes in delta}$
Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption

\[ n = \text{number of nodes}, \ c = \text{number of changes in delta} \]
Double tree feasible?

Does the double tree delta solve the problems?

- **Challenge 1: Efficient Query Support ✓**
- **Challenge 2: Space Consumption**
  - Storing all changed bounds: $O(n)$ space 😞

$n =$ number of nodes, $c =$ number of changes in delta
Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption
  - Storing all changed bounds: $O(n)$ space 😞
  - Storing only range borders: $O(c)$ space 😊

$n = \text{number of nodes}, \ c = \text{number of changes in delta}$
Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption ✓
- Challenge 3: Efficient Update Support

\[ n = \text{number of nodes}, \quad c = \text{number of changes in delta} \]
Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption ✓
- Challenge 3: Efficient Update Support ?

\[ n = \text{number of nodes}, \ c = \text{number of changes in delta} \]
Until now, we only considered deltas with one change.
Until now, we only considered deltas with one change.

How to build deltas with more changes?
Updating Deltas
Task: Swap range $R_2$ with $R_3$
Updating Deltas

- **Step 1:** Insert range borders

- **Search tree insert:** $O(\log c)$ ✓

![Diagram showing step 1 of inserting range borders and search tree insert notation](image)
Updating Deltas

Step X:

- R₁
- R₂
- R₃
- R₄

A B C C' D E E' F G H I
Step 2: Swap borders in $R_2$ and $R_3$
Updating Deltas

- **Step 2:** Swap borders in $R_2$ and $R_3$

- **Naive:** Delete and reinsert all changed borders: $\mathcal{O}(c \log c)$ 😞

- ➞ Better approach required
Efficient Border Swap

Observation: Only target space changes
Efficient Border Swap

Observation: Only target space changes
Efficient Border Swap

- **Observation**: Only target space changes
- **Steps**: Adjust keys $O(c)$ keys, swap $O(c)$ nodes
Efficient Border Swap

- **Observation:** Only target space changes

![Diagram showing the process of efficient border swap]

- **Before:**
  -...
- **After:**
  -...

- **Steps:**
  - Adjust keys
  - swap nodes

- **Diagram Elements:**
  - Nodes labeled A, B, C, D, E, F, G, H, I
  - Edges connecting the nodes
  - Highlighted sections indicating border swaps
How to swap $O(c)$ nodes in a search tree in $O(\log c)$?

Solution: Split and join?

Split: Split a tree into two new balanced trees
Join: Concatenate two trees to one balanced one

Both operations run in $O(\log c)$.
How to swap $O(c)$ nodes in a search tree in $O(\log c)$?

Solution: Split and join

Split: Split a tree into two new balanced trees

Join: Concatenate two trees to one balanced one

Both operations run in $O(\log c)$
How to swap $O(c)$ nodes in a search tree in $O(\log c)$?

Split: Split a tree into two new balanced trees

Join: Concatenate two trees to one balanced one

Both operations run in $O(\log c)$
How to swap $O(c)$ nodes in a search tree in $O(\log c)$?

- **Split**: Split a tree into two new balanced trees
- **Join**: Concatenate two trees to one balanced one
Efficient Node Rearrangement

- How to swap $\mathcal{O}(c)$ nodes in a search tree in $\mathcal{O}(\log c)$?

Split: Split a tree into two new balanced trees

Join: Concatenate two trees to one balanced one

Both operations run in $\mathcal{O}(\log c)$ ✓
Split and join can rearrange nodes efficiently

But: Keys are not updated $\Rightarrow$ search tree condition violated!

Updating one by one would require $O(c)$
Split and join can rearrange nodes efficiently
But: Keys are not updated ⇒ search tree condition violated!
Updating one by one would require $O(c)$
Solution: Accumulation tree
⇒ Node key: Sum of all keys on path to root
Split and join can rearrange nodes efficiently
But: Keys are not updated ⇒ search tree condition violated!
Updating one by one would require $O(c)$
Solution: Accumulation tree
⇒ Node key: Sum of all keys on path to root
Split and join can rearrange nodes efficiently

But: Keys are not updated ⇒ search tree condition violated!

Updating one by one would require $O(c)$

Solution: Accumulation tree

⇒ Node key: Sum of all keys on path to root

Changing all keys in a subtree: $O(1)$
The Swap Algorithm

- Using *split/join* and the *accumulation tree*, updating in $O(\log c)$ is possible
- Target tree before update:
Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible.

Target tree with accumulation before update:
The Swap Algorithm

- Using **split/join** and the **accumulation tree**, updating in $O(\log c)$ is possible
- **Step 1**: Split tree ($O(\log c)$)
The Swap Algorithm

- Using split/join and the accumulation tree, updating in $O(\log c)$ is possible
- **Step 1:** Split tree ($O(\log c)$)
The Swap Algorithm

- Using *split/join* and the *accumulation tree*, updating in $O(\log c)$ is possible
- Rearrange trees (no-op)
The Swap Algorithm

- Using split/join and the accumulation tree, updating in $O(\log c)$ is possible
- Rearrange trees (no-op)
The Swap Algorithm

- Using **split/join** and the **accumulation tree**, updating in $O(\log c)$ is possible
- **Step 2**: Translate keys ($O(1)$)
The Swap Algorithm

- Using split/join and the accumulation tree, updating in $O(\log c)$ is possible
- Step 3: Join trees ($O(\log c)$)
The Swap Algorithm

- Using **split/join** and the **accumulation tree**, updating in $O(\log c)$ is possible
- **Final result:**

```
  ▶ -1  ▶ -7  ▶ +5  ▶ -1  ▶ -3  ▶ -1
  ▶ -7  ▶ +9  ▶ -2  ▶ -1  ▶ -5  ▶ -1
```

```
A B C E' F C' D H E I G
```

```
-1 -1 -7 +9 +6 +8
```

```
A B C E' F C' D H E I G
```
What we have shown:

- Efficient Queries (NI Encoding)
- Efficient Updates (Swap Algorithm)
- Low Space Consumption ($O(c)$)
What we have shown:

- Double tree delta efficiently represents the changes in a version
  - Efficient Queries (NI Encoding)
  - Efficient Updates (Swap Algorithm)
  - Low Space Consumption ($\mathcal{O}(c)$)
What we have shown:
- Double tree delta efficiently represents the changes in a version
  - Efficient Queries (NI Encoding)
  - Efficient Updates (Swap Algorithm)
  - Low Space Consumption ($\mathcal{O}(c)$)

What is missing:
Overview

▶ What we have shown:
  ▶ Double tree delta efficiently represents the changes in a version
    ▶ Efficient Queries (NI Encoding)
    ▶ Efficient Updates (Swap Algorithm)
    ▶ Low Space Consumption ($O(c)$)

▶ What is missing:
  ▶ How to represent whole version histories efficiently?
Assume:

- Linear history of $n$ versions $V_0, \ldots, V_{n-1}$
- Constantly bounded number of changes $c$ per version

What we need:

- $V_0$ has a fully materialized NI encoding
- We need deltas that lead to each other version (transitively)
- E.g., $\delta_{0 \rightarrow 3}$ and $\delta_{3 \rightarrow 5}$ lead to $V_5$ by applying $\delta_{3 \rightarrow 5}(\delta_{0 \rightarrow 3}(b))$

Which deltas to store in order to...?

- minimize space consumption?
- minimize query runtime?
Simple Schemes

- Minimize space consumption: linear topology

![Diagram showing linear topology with nodes labeled 0 to 10 and arrows indicating connections.](image)

- Minimize query time: star topology

![Diagram showing star topology with a central node labeled 'Base' and nodes 1 to 10 radiating outwards.](image)
Simple Schemes

- Minimize space consumption: linear topology
  ⇒ $O(n)$ space consumption ✓
Simple Schemes

- Minimize space consumption: **linear topology**
  - $O(n)$ space consumption ✓
  - $O(n)$ query time 😞
Simple Schemes

- Minimize space consumption: **linear topology**
  - $O(n)$ space consumption ✓
  - $O(n)$ query time 😞

- Minimize query time: **star topology**
Simple Schemes

- Minimize space consumption: linear topology
  - $O(n)$ space consumption ✓
  - $O(n)$ query time 😞

- Minimize query time: star topology
  - $O(\log n)$ query time ✓
Simple Schemes

- Minimize space consumption: **linear topology**
  - \( O(n) \) space consumption ✓
  - \( O(n) \) query time 😞

- Minimize query time: **star topology**
  - \( O(\log n) \) query time ✓
  - \( O(n^2) \) space consumption 😞
We need a better space/time tradeoff!
We need a better space/time tradeoff!

**Solution:** Exponential scheme

![Exponential Scheme Diagram](image)
We need a better space/time tradeoff!

Solution: Exponential scheme

\[ O(n \log n) \] space consumption

\[ O(\log_2 n) \] best case query time

\[ O(\log n) \] worst case query time
We need a better space/time tradeoff!

Solution: Exponential scheme

- $O(n \log n)$ space consumption ✓
- $O(\log n)$ best case query time ✓
We need a better space/time tradeoff!

**Solution:** Exponential scheme

- $O(n \log n)$ space consumption
- $O(\log n)$ best case query time
- $O(\log^2 n)$ worst case query time
Evaluation Baseline

- Baseline: Currently strongest algorithms
Baseline: Currently strongest algorithms
  ▶ Labeling with ORDPATH
    ▶ No relabeling ⇒ efficient updates
    ▶ Efficient queries
Evaluation Baseline

- **Baseline:** Currently strongest algorithms
  - Labeling with ORDPATH
    - No relabeling $\Rightarrow$ efficient updates
    - Efficient queries
  - Versioning with Multiversion B-Tree (MVBT)
    - Asymptotically optimal query time and space consumption
Baseline: Currently strongest algorithms ORD-MVBT
  - Labeling with ORDPATH
    - No relabeling $\Rightarrow$ efficient updates
    - Efficient queries
  - Versioning with Multiversion B-Tree (MVBT)
    - Asymptotically optimal query time and space consumption
Evaluation Baseline

- Baseline: Currently strongest algorithms ORD-MVBT
  - Labeling with ORDPATH
    - No relabeling ⇒ efficient updates
    - Efficient queries
  - Versioning with Multiversion B-Tree (MVBT)
    - Asymptotically optimal query time and space consumption
- Improvements with DeltaNI
  - Support of subtree relocation and deletion
  - Branching histories
  - Simple integer comparisons instead of bytewise comparisons
Evaluation: Query Performance

Time for one million queries

Execution Time (s)

Version

DeltaNI
ORD-MVBT
Evaluation: Space Consumption

Space consumption

<table>
<thead>
<tr>
<th>Version</th>
<th>Total Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>DeltaNI</td>
</tr>
<tr>
<td>26</td>
<td>0.5</td>
</tr>
<tr>
<td>29</td>
<td>1.5</td>
</tr>
</tbody>
</table>

- ORD-MVBT

Graph showing the increase in space consumption with version.
Evaluation: Update Performance

Time for one million updates

Execution Time (s)

Version

DeltaNI
ORD-MVBT
Core observation: All updates reducable to range swap in the NI encoding
Core observation: All updates reducable to range swap in the NI encoding

- Double tree interval deltas make NI encoding dynamic
  - $O(c)$ space consumption
  - $O(\log c)$ update complexity
  - Even complex updates supported (subtree relocation)
Conclusion

- Core observation: All updates reducable to range swap in the NI encoding
- Double tree interval deltas make NI encoding dynamic
  - $O(c)$ space consumption
  - $O(\log c)$ update complexity
  - Even complex updates supported (subtree relocation)
- Versioning via exponential delta-packing scheme
  - Yields reasonable space/time tradeoff
Thank you for your attention!

Any questions?