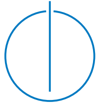
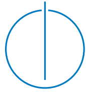


Technische Universität München
Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme

On Supporting Hierarchical Data in Relational Main-Memory Database Systems

Jan Peter Finis



Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

On Supporting Hierarchical Data in Relational Main-Memory Database Systems

Jan Peter Finis
Master of Science,
Master of Science with honours

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation: 1. Univ.-Prof. Alfons Kemper, Ph.D
2. Univ.-Prof. Nikolaus Augsten, Ph.D.
(Universität Salzburg, Österreich)
3. Univ.-Prof. Dr. Thomas Neumann

Die Dissertation wurde am 24.03.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.06.2016 angenommen.

For my parents,
who kept their faith in me even though
my elementary school teacher told them
that I would never succeed in life

Abstract

Hierarchical data is prevalent in a number of business use cases. However, most business data is stored in a relational database and hierarchical data is somehow encoded relationally. Since most queries on hierarchies also feature relational data, companies must continue maintaining their hierarchical data in a relational database system.

There are many challenges when storing and querying hierarchical data in a relational context, because tables are inherently flat data structures while hierarchies possess inherent recursive properties. This impedance mismatch calls for dedicated hierarchy support in relational database systems: A user-friendly front end for querying hierarchical data and a sophisticated back end for evaluating these queries efficiently. In addition, hierarchies are usually dynamic with a high rate of complex, structural updates being issued on them. The aim of this thesis is to take on all these challenges.

We propose a holistic framework for maintaining and querying hierarchical data in a modern main-memory relational database system. We design a data model for unifying hierarchical and relational data. Based on this model, we propose a query language, which blends seamlessly into SQL and allows the intuitive phrasing of queries working simultaneously on hierarchical and relational data. We then devise a family of indexing schemes to index hierarchical data effectively by enabling efficient queries and updates. We show how the proposed indexes can be used to answer queries in our SQL extension, thus providing all building blocks for our hierarchy framework. We then extend the framework with indexes for versioned hierarchical data, since many use cases require versioning for traceability and confirmability purposes. Finally, we propose an algorithm for the change recognition in hierarchies, which is important for versioning and completes our hierarchy framework.

As our experiments on real-world data of SAP customers show, our framework enables unprecedented performance and is the first approach to handle complex updates efficiently. The fact that parts of our framework have already been shipped with the latest release of the SAP HANA Vora in-memory query engine [85] proves that our work is not only of theoretical scientific interest but adds real business value to a relational system.

Kurzfassung

Hierarchische Daten sind weit verbreitet in geschäftlichen Anwendungsfällen. Jedoch werden die meisten Unternehmensdaten in relationalen Datenbanken gespeichert und hierarchische Daten werden relational codiert. Da die meisten Anfragen auf hierarchischen Daten auch relationale Daten beinhalten, müssen Unternehmen weiterhin hierarchische Daten in ihren relationalen Systemen halten.

Durch das Speichern und Anfragen von hierarchischen Daten im relationalen Kontext ergeben sich viele Herausforderungen, da Tabellen inhärent flach sind, während Hierarchien inhärent rekursive Eigenschaften besitzen. Dies erfordert dedizierte Hierarchieunterstützung in relationalen Systemen: Ein nutzerfreundliches Frontend zur Anfrage von hierarchischen Daten und ein ausgefeiltes Backend, um diese Anfragen effizient auszuwerten. Zusätzlich werden auf Hierarchien im Allgemeinen viele komplexe, strukturelle Änderungen durchgeführt. Ziel dieser Arbeit ist, all diese Herausforderungen anzugehen.

Wir schlagen ein holistisches Framework zur Haltung und Anfrage von hierarchischen Daten in einem modernen, relationalen Hauptspeicherdatenbanksystem vor. Wir entwerfen ein Datenmodell zur Vereinigung von hierarchischen und relationalen Daten. Basierend auf diesem Modell führen wir eine Anfragesprache ein, welche sich nahtlos in SQL integriert und dabei die intuitive Formulierung von Anfragen ermöglicht, welche gleichzeitig auf hierarchischen und relationalen Daten arbeiten. Dann entwerfen wir eine Familie von Indexstrukturen zur effektiven Indexierung von hierarchischen Daten. Wir zeigen wie Anfragen in unserer SQL-Erweiterung mit Hilfe unserer Indexe beantwortet werden können. Somit stellen wir alle nötigen Bausteine für unser Framework bereit. Als nächstes erweitern wir das Framework um Indexe für versionierte hierarchische Daten, da viele Anwendungsfälle Versionierung aus Rückverfolgbarkeits- und Beweisbarkeitsgründen benötigen. Als letztes beschreiben wir einen Algorithmus zur Erkennung von Änderungen in Hierarchien, welcher zur Versionierung benötigt wird und unser Framework abrundet.

Wie unsere Experimente mit echten Kundendaten von SAP belegen, stellt unser Framework beispiellose Leistung bereit und ist das erste Verfahren, welches komplexe Änderungsoperationen effizient unterstützt. Der Fakt, dass Teile unseres Frameworks bereits mit dem neuesten Release der SAP HANA Vora Hauptspeicherengine [85] ausgeliefert wurden zeigt, dass unsere Arbeit nicht nur von theoretischem, wissenschaftlichen Interesse ist, sondern auch zum wirtschaftlichen Nutzen eines relationalen Systems beiträgt.

Contents

Abstract vii

Kurzfassung ix

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 The Hierarchy/Table Impedance Mismatch . . . 2
- 1.2 Contribution and Outline . . . 5

2 A Data Model and Front End For Hierarchical Data 9
- 2.1 Requirements Review . . . 11
- 2.2 From Status Quo to Our Proposal . . . 13
- 2.3 Hierarchical Tables: Our Model . . . 19
- 2.4 Querying Hierarchies . . . 21
- 2.5 Creating and Manipulating Hierarchies . . . 23
  - 2.5.1 Deriving a Hierarchy from an Adjacency List . . . 23
  - 2.5.2 Hierarchical Base Tables . . . 26
  - 2.5.3 Manipulating Hierarchies . . . 27
- 2.6 Advanced Customer Scenarios . . . 28
- 2.7 Architecture and Implementation Aspects . . . 32
  - 2.7.1 Hierarchy Indexing Schemes . . . 32
  - 2.7.2 Hierarchy-Aware Join Operators . . . 34
  - 2.7.3 Bulk-Building . . . 35
- 2.8 Experiments . . . 39

2.9 Conclusion 41

3 Order Index: Indexing Highly Dynamic Hierarchical Data 43

3.1 Dynamic Hierarchies in Relational Systems 45
3.1.1 Challenges 45
3.1.2 Query Capabilities 47
3.1.3 Update Capabilities 50
3.2 Related Indexing Schemes 52
3.3 Hierarchical Query Processing 58
3.4 Order Indexes 63
3.4.1 The Order Index Concept 63
3.4.2 Order Index Structures 66
3.4.3 Back-Links in Block-Based Order Indexes 70
3.4.4 Updating Order Indexes 72
3.5 Order Index Extensions 84
3.5.1 Disk-Based Systems 84
3.5.2 Supporting ordinal primitives with the BO-Tree 85
3.6 Performance Evaluation 88
3.6.1 Test Setup 89
3.6.2 Block Size & Back-Link Representation 90
3.6.3 Comparison to Existing Schemes 92
3.7 Conclusion 100

4 DeltaNI: Indexing Versioned Hierarchical Data 103

4.1 Hierarchies in RDBMS 104
4.2 Interval Deltas 107
4.3 Implementing the Query Primitives 109
4.4 Efficient Delta Representation 110
4.5 Obtaining Deltas 115
4.5.1 Static Scenario 115
4.5.2 Dynamic Scenario 117
4.6 Delta Version Histories 120
4.6.1 Querying the History 122
4.6.2 Exponential Deltas 123
4.6.3 Optimizations 127
4.7 Evaluation 128

4.8 Related Work 132
4.9 Conclusion 134

5 RWS-Diff: Flexible Change Detection in Hierarchical Data 135
5.1 Tree Edit Scripts 137
5.2 Related Work 139
5.2.1 Tree Edit Distance Computation 139
5.2.2 Computing Diffs between Trees 140
5.3 The RWS-Diff Algorithm 142
5.3.1 Finding Simple Mappings 142
5.3.2 Random Walk Similarity Matching 143
5.3.3 Edit Script Generation 145
5.3.4 Complexity of RWS-Diff 147
5.4 Random Walk Similarity 148
5.4.1 Grams for Trees 148
5.4.2 Random Walk Distance 150
5.4.3 Weighting Grams 153
5.5 Evaluation 153
5.6 Conclusion 160

6 Conclusions 163

Bibliography 165

List of Figures

1.1 Example hierarchy and corresponding table 2

2.1 Example hierarchy and corresponding table 10
2.2 Using SQL/XML for XML generation and XQuery evaluation . . . 14
2.3 Example hierarchical query, expressed using two RCTEs. 17
2.4 Example from Figure 2.3 expressed using the proposed SQL extensions 18
2.5 Querying a heterogeneous hierarchy 30
2.6 Example hierarchy and its NODE column using different indexing schemes 33
2.7 Steps of the bulk build algorithm 36
2.8 Depth first traversal 37

3.1 Various labeling schemes and their changes when a subtree is relocated 46
3.2 Relocation updates on an example hierarchy: before and after . . . 50
3.3 An example query using the SQL extensions from [18] 59
3.4 A hierarchy with Order Index (AO-Tree) 63
3.5 Query evaluation in the Order Indexes 67
3.6 Using back-links to find entry [3 71
3.7 Inserting new leafs H and I in a BO-Tree 72
3.8 Relocating B below F in a BO-Tree 80
3.9 Relocating B below F in an O-List 82
3.10 Updating levels after left rotation in an AO-Tree [36] 83
3.11 Adjusting levels after a leaf block merge 83
3.12 Main memory and disk implementation of an O-List 85
3.13 Ordinal queries in the BO-Tree 86
3.14 BO-Tree performance for different back-links and block sizes 91
3.15 Performance of different query primitives 93
3.16 Compound query performance (scan[x]) 95

3.17 Bulk build and leaf update performance 95
3.18 (a) mixed_updates[p] (b) relocate_subtree[x] (c) relocate_range[y] . . 96
3.19 Performance over varying hierarchy size 98

4.1 An HR hierarchy and its NI encoding 104
4.2 Insertion, relocation, and deletion in a hierarchy, modeled by a swap 112
4.3 Model of translation ranges 114
4.4 Representation of translation ranges 114
4.5 Inferring deltas from source and target interval encoding 116
4.6 Updating a version delta by swapping translation range R_2 with R_3 117
4.7 Using the accumulation tree as target tree 119
4.8 Left rotation in the accumulation tree 120
4.9 Performing the swap operation on the (accumulation) target tree. . 121
4.10 Using the number of trailing zeros for deciding delta sizes 123
4.11 Merging two deltas 126
4.12 Execution time and space consumption measurements 130

5.1 Slightly different trees make top-down and bottom-up matching fail 136
5.2 Sibling order invariant subtree hashes 143
5.3 Algorithm for generating edit scripts 145
5.4 Edit mapping with implied edit operations 146
5.5 Partial construction of p,q-grams 149
5.6 Transforming bags of grams into corresponding random walks . . . 151
5.7 One change per parent on the nasa1 data set. 156
5.8 10 changes on the nasa1 data set. 157
5.9 Growing number of changes on the nasa1 data set. 158

List of Tables

2.1 Projecting hierarchy properties of `BOM` . . . 21
2.2 Bulk-building performance . . . 40
2.3 Query performance . . . 40

3.1 Essential query primitives on hierarchies . . . 47
3.2 Ordinal query primitives . . . 48
3.3 Update operations on hierarchies . . . 49
3.4 Asymptotic query complexities for various indexing schemes . . . 53
3.5 Asymptotic update complexities for various indexing schemes . . . 54
3.6 Implementing the query primitives . . . 65
3.7 Basic functions and fields used during range relocation . . . 75
3.8 Comparison of ordinal approaches . . . 88
3.9 Number of cache misses per operation . . . 94
3.10 memory consumption in bytes per node . . . 97

4.1 Implementing the Order Index interface for DeltaNI . . . 110

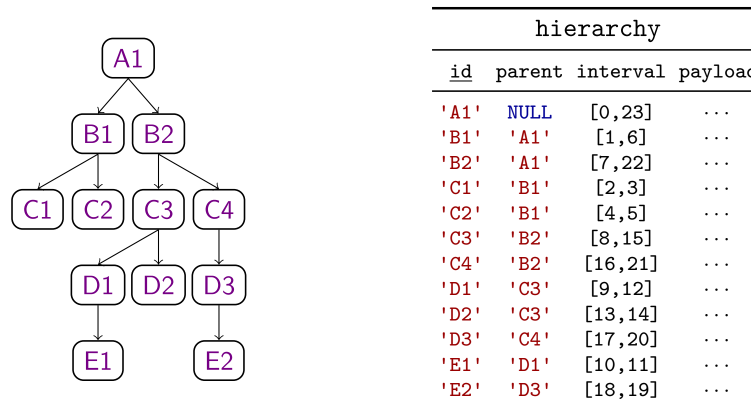
5.1 Comparison of indexes with 10,000 points . . . 154
5.2 Comparison of indexes with 100,000 points . . . 154
5.3 Result for the *bbc* data set . . . 159
5.4 Result for the *tagesschau* data set . . . 160

CHAPTER 1

Introduction

Hierarchical data, that is, data shaped as a rooted tree or forest, has always been ubiquitous in business, engineering, and science applications. In Enterprise Resource Planning (ERP) applications, various kinds of large hierarchies exist. For example, companies need to manage human resource (HR) hierarchies, which representing their organizational structures such as reporting lines or geographical divisions, 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 constitute a hierarchical arrangement of components to assemble an end product. Taxonomies, file systems, and hierarchies in the dimensions of data cubes are further examples for the prevalence of hierarchical data. The hierarchical data models JSON and XML have virtually become the language of the world wide web and have also received a lot of attention from the research community in the last decade. All in all, the importance of hierarchical data prevails in virtually all applications.

Although hierarchical data is so omnipresent, the relational model is still the predominant data model for modern database systems. Since such relational database systems (RDBMS) have been the core data storage for almost all mission critical corporate data in the last decades, it is very unlikely that they will be replaced in the near future. Moreover, even if companies were willing to install separate hierarchical databases, having coexisting relational and hierarchical systems would overcomplicate business analytics since users virtually always need relational *and* hi-



hierarchy			
id	parent	interval	payload
'A1'	NULL	[0,23]	...
'B1'	'A1'	[1,6]	...
'B2'	'A1'	[7,22]	...
'C1'	'B1'	[2,3]	...
'C2'	'B1'	[4,5]	...
'C3'	'B2'	[8,15]	...
'C4'	'B2'	[16,21]	...
'D1'	'C3'	[9,12]	...
'D2'	'C3'	[13,14]	...
'D3'	'C4'	[17,20]	...
'E1'	'D1'	[10,11]	...
'E2'	'D3'	[18,19]	...

Figure 1.1: Example hierarchy and corresponding table

erarchical data. Joining these data models would be cumbersome if they are stored in different systems. Another problem with dedicated hierarchical databases is that most legacy hierarchical corporate data is somehow encoded in relations. Manually migrating this implicitly represented hierarchical data to an own system would be a difficult and error-prone task. Consequently, the only reasonable approach for handling hierarchical corporate data is to use existing relational systems.

1.1 The Hierarchy/Table Impedance Mismatch

When trying to represent hierarchical data in relations, we face a classical impedance mismatch: Relations are flat tables, while hierarchies have inherent recursive properties. Figure 1.1 shows an example hierarchy on the left and a table representing it in two different ways on the right: Either through storing the key of the `parent` node of each node, or by storing a numeric `interval` for each node. Especially the former is a standard means of encoding a hierarchy in relational data.

A simple hierarchical query consists of finding all descendants of node B2 (which are C3, C4, D1, D2, D3, E1, and E2 in this example). When using the parent column to represent the hierarchy, this simple query cannot be answered in SQL-92, since we would have to self-join the table repeatedly to find all descendants. The number of joins depends on the depth of the hierarchy, so no query with a fixed number of joins is able to find all descendants in the general case. Instead, database users tend to implement this repeated join in application code, issuing one query

per level of the hierarchy. This approach is neither efficient nor user-friendly, as it requires query logic in the application code.

Recursive Common Table Expressions (RCTEs) [39, 3], which are part of the SQL-99 standard, can be used to express the query thoroughly:

```
WITH RECURSIVE descendants (id) AS (
   SELECT h.id FROM hierarchy h WHERE h.parent = 'B2'
   UNION ALL
   SELECT h.id FROM hierarchy h JOIN descendants d ON h.parent = d.id
)
SELECT id
  FROM descendants
```

While we are now able to at least express the query, the RCTE can still not be considered a good solution: First, it is still overly complex even for this extremely simple query. Next, the fact that this query represents all descendants of B2 is not obvious from the query; it is therefore hard to read and maintain. Finally, the recursive join specified with this statement is still an inferior evaluation technique for hierarchies and shows unsatisfying performance for large hierarchies.

Since no satisfying query evaluation techniques for hierarchies represented by a parent column exist, one may argue that such a column is simply a bad representation and the problems with it can be circumvented by picking a better suited hierarchy encoding. And indeed, the `interval` column shown in Figure 1.1 can answer the example query with one single join:

```
SELECT h.id FROM hierarchy h JOIN hierarchy r
    ON h.interval.lower > r.interval.lower
   AND h.interval.upper < r.interval.upper
 WHERE r.id = 'B2'
```

Although this join is easier to write and more efficient to evaluate than the RCTE, the fact that this query works on a hierarchy and extracts descendants is still hidden. From a bird's eye view, the query looks like a simple join over some relational data using a range predicate. In addition, using the interval column to encode the hierarchy structure makes updates very expensive, since the insertion of a node usually changes the interval of various other nodes. So while queries are easier and more efficient with this encoding, updates are much more difficult and less efficient.

The encodings shown here are only two examples of many existing ones. However, they are good representatives for showing that although it might seem simple to

represent hierarchical data in relations on first sight, each encoding has severe weaknesses which limit its applicability.

The example also shows that the challenges when blending hierarchical data into relational data are twofold: First, the chosen encoding must facilitate efficient query *and* update processing. Second, the query language must be user-friendly. It may not deceive about the fact that a hierarchy is being queried: The kind of hierarchical query issued must be obvious from the syntax. In addition, the syntax must blend well into SQL so that SQL users are able to learn it without much effort.

In conclusion, the hierarchy/table impedance mismatch demands adaptions in the front end (the query language) as well as in the back end (the query processing and encoding). Hence, only a holistic approach taking into account all database layers can yield thoroughly satisfying results.

Due to the prevalence of hierarchical data in business applications, almost all large database vendors have included some explicit hierarchy support into their relational products: Oracle Database provides Hierarchical Queries [2] as a proprietary SQL extension for traversing recursively structured data. Microsoft SQL Server 2008 introduced the `hierarchyid` [72, 1] data type whose values represent a position in a hierarchy. PostgreSQL supports hierarchical data with its ltree module [90]. The SQL/XML standard [4] for integrating XML support into SQL has also been implemented by prominent vendors [61, 75, 13]. Considering research, hierarchical data support has received a lot of attention, especially with the advent of XML. Dozens of papers have been dedicated to various aspects of hierarchical data support. In conclusion, a lot of approaches for supporting hierarchical data in relational systems exist in research as well as in commercial and open-source database products.

Since virtually every existing relational database system features some kind of hierarchy support and a lot of papers exist on the topic, it might seem that this problem may be considered solved. However, all existing systems suffer from various issues which we will investigate in the remainder of this thesis. Consequently, their general applicability is limited and a satisfactory solution is missing up to date. The reason for this is probably that most systems integrated hierarchy support because of growing customer demand while not treating this challenging problem with the due prudence: While the relational model needed decades of research to mature to the state in which it is today, most systems seem to have integrated the hierarchical

data model on a least effort basis. In contrast, a data model introducing such a severe impedance mismatch calls for a careful integration strategy. In addition, almost all dedicated hierarchy support in existing systems is about querying a hierarchy; efficient update support is not in the focus. Some system only support fully static hierarchies, while others provide some limited, often inefficient means of updating hierarchical data. Since there are indeed use cases featuring *very* dynamic hierarchies, efficient update support is of paramount importance.

In conclusion, all existing systems lack some aspect of satisfactory support of possibly-dynamic hierarchical data. The aim of this thesis is therefore a detailed and thorough consideration of the challenges arising in this context.

1.2 Contribution and Outline

All in all, our contribution is a fully-featured, efficient, and user-friendly framework for maintaining and querying hierarchical data in a relational main-memory database system. Supporting hierarchical data in such a system requires an easy-to-use front end as well as an efficient back end.

The thesis is structured as follows: Chapter 2 covers the front end considerations, while Chapter 3 covers the back end considerations. Afterwards, we explore the advanced topics hierarchy versioning (Chapter 4) and hierarchy change detection (Chapter 5). Finally, Chapter 6 draws conclusive remarks.

Our first contribution (Section 2.1) is a survey of the requirements a hierarchy framework in a relational setting must fulfill. The survey is based on various SAP customer scenarios and therefore accurately reflects the demands of business use cases.

A data model and front end for hierarchical data in a relational setting is our second contribution (Chapter 2). In this regard, we also investigate into the shortcomings of existing database systems and data models for hierarchical data. Based on our data model we propose a query language for hierarchical data which blends seamlessly into SQL and allows for queries featuring hierarchical and relational data simultaneously. We also propose data definition language (DDL) and data manipulation language (DML) constructs for hierarchical data. Therefore, we provide a fully-featured front end for hierarchical data in a relational database system.

As our third contribution (Section 2.7.3), we provide a highly-efficient operator for bulk-building a hierarchy from an existing relational representation, which is important for legacy use cases. This operator allows efficient ad-hoc hierarchy query execution on relationally-encoded hierarchical data.

Existing works on hierarchy indexing often only cover a small set of query and update primitives, which is not rich enough to answer important queries or execute necessary kinds of updates. In fact, we were unable to find any existing work on finding a suitable abstract interface for hierarchy indexes. Therefore, our fourth contribution (Section 3.1) consists of a generic interface of hierarchy query and update primitives. The primitives are carefully selected with regards to query and update needs of various business scenarios with the capabilities of existing indexing schemes in mind. Hence, this interface describes a necessary set of operations which each hierarchy indexing scheme must expose to be considered generally applicable.

A survey and a taxonomy of existing indexing schemes is our fifth contribution (Section 3.2). Our findings reveal that most existing schemes share similar properties and capabilities. They therefore can be summed up in a small number of categories with similar asymptotic behaviour. For all identified indexing categories we derive the asymptotic query and update runtimes of all methods from our proposed interface. We therefore draw an expressive outline of existing indexing techniques.

Using our proposed generic interface of query primitives, our sixth contribution (Section 3.3) is to show how indexes with this interface can be used in various hierarchy-aware relational operators to evaluate hierarchical queries efficiently. This forms the basis of our proposed back end for efficient query evaluation.

As our seventh contribution (Sections 3.4—3.6) we propose *Order Indexes* as a family of index structures for hierarchical data that is able to handle high rates of potentially-complex updates. To our best knowledge, Order Indexes are the first indexing technique which can handle all kinds of complex hierarchy updates efficiently. In addition, they support all proposed query primitives and offer query performance comparable to or even higher than existing static indexing schemes. Using Order Indexes in conjunction with hierarchy-aware relational operators is an efficient means for hierarchical query and update execution and therefore an efficient back end for our framework.

Our eighth contribution (Chapter 4) is *DeltaNI*, an index structure for *versioned* hierarchical data. Using this index structure allows for maintaining an unbounded number of versions of a hierarchy compactly. Updates are appended to the latest

version for a linear history of versions. In addition, the index also supports branching version histories where new branches can be created from any existing version and updates can be appended to any branch. The index is able to answer queries efficiently in *all* versions, irrespective of the age of the version. DeltaNI exposes the same interface as Order Indexes. Therefore, the relational operators used in conjunction with Order Indexes for efficient query processing can also be used with DeltaNI to enable query processing on versioned hierarchies without any changes in the operators. By using DeltaNI in conjunction with Order Indexes, our framework is able to support non-versioned and versioned hierarchical data–both with maximum performance–in the same database.

The *RWS-Diff* algorithm for finding the differences between two hierarchies is our ninth and final contribution (Chapter 5). Given two hierarchies, it generates an approximately cost-minimal edit-script, which contains a sequence of edit operations to transform the first hierarchy into the second. Thus, the edit script compactly represents the difference between the two hierarchies (the so-called edit distance). In our scenario, such edit scripts are useful when a hierarchy stored in the database is altered by a third party tool. Then, we can compute the edit script to trace the changes performed by the third party tool and to apply these changes onto the database.

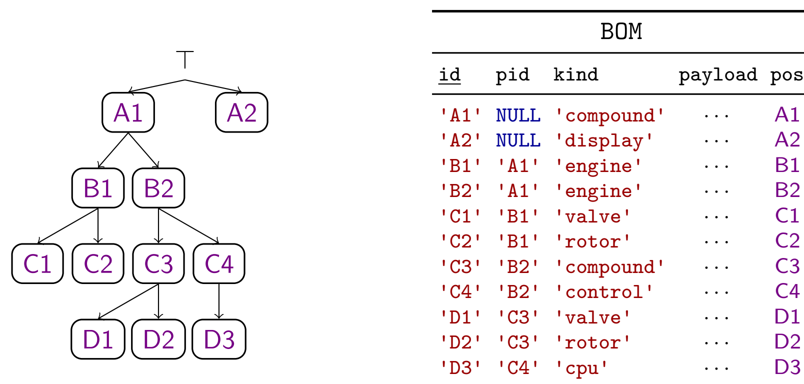
The practical applicability of our contributions in a modern setting is demonstrated by integrating our hierarchy handling framework into the relational main-memory database systems HyPer [51] and SAP HANA [35]. The feasibility for business scenarios is evidenced by basing our experiments on real business data of SAP customers. The fact that parts of our framework have already been shipped with the latest release of the SAP HANA Vora in-memory query engine [85] proves that our work is not only of theoretical scientific interest but adds real business value to a relational system.

CHAPTER 2

A Data Model and Front End For Hierarchical Data

Parts of this chapter have previously been published in [18].

Hierarchical relations appear in virtually any business application, be it for representing organizational structures such as reporting lines or geographical divisions, the structure of assemblies, taxonomies, marketing schemes, or tasks in a project plan. But as tree and graph structures do not naturally fit into the flat nature of the traditional relational model, handling such data today remains a clumsy and unnatural task for users of the SQL language. We have investigated a number of SAP applications dealing with hierarchies and collected their typical requirements (Section 2.1). Within SAP's Enterprise Resource Planning system, hierarchies are used in the human resources domain to model reporting lines of employees, in asset management to keep track of production-relevant assets and their functional locations (e. g., plants, machines, machine parts, tools, equipment), and in materials planning to represent an assembly of components into an end product, a so-called *bill of materials* (BOM). Due to the limitations of the relational model and SQL, logic for hierarchy handling within these applications has mostly been written in ABAP and therefore runs within the application server. We have identified almost a dozen different implementations of more or less the same hierarchy-handling logic, which is unfortunate not only from an interoperability and maintainability point of view. In most cases, hierarchies are represented in the database schema using a simple relational encoding, and converted into a custom-tailored format within the application, if needed. The most widespread encoding is a self-referencing



BOM				
id	pid	kind	payload	pos
'A1'	NULL	'compound'	...	A1
'A2'	NULL	'display'	...	A2
'B1'	'A1'	'engine'	...	B1
'B2'	'A1'	'engine'	...	B2
'C1'	'B1'	'valve'	...	C1
'C2'	'B1'	'rotor'	...	C2
'C3'	'B2'	'compound'	...	C3
'C4'	'B2'	'control'	...	C4
'D1'	'C3'	'valve'	...	D1
'D2'	'C3'	'rotor'	...	D2
'D3'	'C4'	'cpu'	...	D3

Figure 2.1: Example hierarchy and corresponding table

table resembling an adjacency list. It is well-known in the literature (e. g., [22]) under the term *Adjacency List model*. Figure 2.1 shows an example instance: table `BOM` represents a bill of materials. Field `id` uniquely identifies each part. The hierarchical relationship is established by a self-reference `pid` associating each row with its respective parent row, the part containing it. The resulting hierarchy is displayed on the left.

The only standard hierarchy-handling tools that SQL-based DBMSs today offer are *Recursive Common Table Expressions* (RCTEs) and some means to define custom stored procedures for working on adjacency lists like `BOM`. But these tools suffer from certain usability and performance deficiencies. The conventional alternative is to abandon the Adjacency List model and implement a more suitable encoding scheme manually, either on the relational level [22] or within the application (i. e., ABAP), relinquishing any kind of engine support either way. We consider neither RCTEs nor alternative encoding schemes nor any other purported solution we investigated sufficient to meet the common requirements that we have identified (Section 2.2). These requirements call for a solution that seamlessly integrates hierarchical and relational data by combining an expressive front end (new language constructs) with a powerful back end (indexing and query processing techniques), without departing farther than necessary from the philosophy of the relational model. Such a solution is missing to date.

That is not to say the research community has not given enough attention to the underlying technical challenges. In fact, the problem of storing and indexing hierarchical or recursively structured data has been studied deeply in the past decade, in the course of research efforts in the domains of XML and semi-structured

databases. While recognizing that many challenges of indexing and query processing have been successfully tackled previously (e. g., by RCTEs, encoding schemes, and structural joins), we believe there is an open opportunity to reconcile the ideas and techniques from the mentioned areas into a general framework for working with hierarchies in a relational context.

The core concept of our framework is to encapsulate structural information into a table column of a new special-purpose data type `NODE` (Section 2.3). Our extensions to the SQL query language comprise a small yet essential set of built-in functions operating on the `NODE` values (Section 2.4), plus DDL and DML constructs for obtaining and manipulating a `NODE` column in the first place (Section 2.5). Altogether the language elements cover the typical requirements and blend seamlessly with the look and feel of SQL, as we illustrate on some advanced scenarios (Section 2.6). The back end design we present is geared to the in-memory column store of SAP HANA [35]. From an early stage on, HANA has been envisioned as a multi-engine query processing environment offering abstractions for data of different degrees of structure, and the idea of hierarchical tables fits well in this spirit. That said, our framework is generic in such a way that it can be adapted for different RDBMSs, including classical row stores. Our HANA-based prototype provides insights into architectural and implementation aspects (Section 2.7) as well as a proof of concept and initial performance characteristics (Section 2.8).

2.1 Requirements Review

Through consultations with stakeholders at SAP we investigated a number of customer scenarios dealing with hierarchies. We restrained ourselves to “strict” hierarchies—basically *trees*—and precluded applications featuring general non-tree graphs. Hence, we identified the following requirements that a DBMS should fulfill to exhibit decent support for hierarchical data:

#1 *Tightly integrate relational and hierarchical data.* Today business data still resides mainly in pure relational tables, and virtually any query on a hierarchy needs to simultaneously refer to such data. First and foremost, any hierarchy support must harmonize with relational concepts, both on the data model side and on the language side. In particular, it is of major importance that *joining* hierarchical and relational data is straightforward and frictionless.

#2 *Provide expressive, lightweight language constructs.* Apart from expressiveness, the constructs must be intelligible in such a way that SQL programmers are able to intuitively understand, adopt, and leverage the functionality they provide. At the same time, an eye must be kept on light syntactic impact: Where appropriate, existing constructs should be reused or enhanced rather than replaced with new inventions. This not only minimizes training efforts for users who are familiar with SQL, but also reduces implementation complexity and adoption barriers for an existing system such as HANA.

#3 *Enable and facilitate complex queries* involving hierarchies, by offering convenient query language (QL) constructs and corresponding back end support. Typical tasks to be supported are: querying hierarchical properties such as the *level* of a node, and selecting nodes according to their properties; testing node relationships with respect to *hierarchy axes* such as *ancestor*, *descendant*, *parent*, *sibling*, or *child*; navigating along an axis starting from a given set of nodes; and arranging a set of nodes in either depth-first or breadth-first order.

#4 *Support explicit modeling of hierarchical data.* In a plain relational database, a schema designer initially has to decide on how to represent a hierarchy (e. g., as an adjacency list). Even though relational tree encodings are well understood nowadays [22], carefully choosing and implementing one still requires advanced knowledge. What's more, a tree encoding generally disguises the fact that the corresponding table contains a hierarchy. What is needed is a way to *explicitly* model hierarchies in the schema, using abstract, special-purpose data definition (DDL) constructs that hide the fiddly storage details.

#5 *Provide front and back end support for data manipulation.* Hierarchies in business scenarios are usually *dynamic*. In some cases manipulations involve only insertions or removals of individual leaf nodes, while in other cases more complex update operations are required, such as collective relocations of large subtrees. As an example, consider the enterprise assets (EA) hierarchy of an automotive company that contains a lot of machines and robots and relocates them whenever a new production line is established. Through a previous analysis of a customer's EA hierarchy, we found that as much as 31% of all recorded update operations were subtree relocations [36]. Consequently, the system must provide an interface (DML) and efficient back end support for both leaf and subtree manipulations.

#6 *Support legacy applications*, where modeling and maintaining hierarchies explicitly (#4) by extending the schema is not always an option. In existing schemas, hierarchies are necessarily encoded in a certain relational format—in the majority of cases the Adjacency List format. Users shall be enabled to take advantage of all QL functionality "ad-hoc" on the basis of such data *without* having to modify the schema. For that purpose, means to create a *derived hierarchy* from an adjacency list shall be provided.

#7 *Enforce structural integrity of a hierarchy.* To this end, the system must prevent the user from inserting non-tree edges. Furthermore, it must ensure that a node cannot be removed as long as it still has children, which would become orphans.

#8 *Cope with very large hierarchies* of millions of nodes. Every proposed language construct must yield an evaluation plan that is at least as fast as—and often considerably faster than—the best equivalent hand-crafted RCTE. To achieve maximum query performance, back end support in the form of special-purpose *indexing schemes* is indispensable. For hierarchies that are never updated—in particular derived hierarchies (#6)—read-optimized *static* indexing schemes are to be used. In contrast, dynamic scenarios (#5) demand for *dynamic* indexing schemes that support update operations efficiently.

In addition to these primary requirements, further requirements arise in some advanced customer scenarios: One example are multi-versioned or *temporal* hierarchies. Another example are "non-strict" hierarchies, that is, directed graphs that contain a few non-tree edges but shall be treated as trees anyway. Such hierarchies can be handled by systematically extracting a spanning tree from the graph, or by replicating subtrees that are reachable via non-tree edges. Although we already anticipate these advanced requirements in our prototype, they are not in the scope of this thesis.

2.2 From Status Quo to Our Proposal

Do we need yet another solution? The problem at hand is almost ancient in terms of DBMS history. *Hierarchical Queries* [2], a proprietary SQL extension for traversing recursively structured data in the Adjacency List format, have been a part of Oracle Database for about 35 years. In 1999, Recursive Common Table

```
SELECT
 XMLElement("PurchaseOrder", XMLAttributes(pono AS "pono"),
  XMLElement("ShipAddr", XMLForest(
   street AS "Street", city AS "City", state AS "State")),
  (SELECT XMLAgg(
    XMLElement("LineItem", XMLAttributes(lino AS "lineno"),
     XMLElement("liname", liname)))
   FROM lineitems l
  WHERE l.pono = p.pono)
 ) AS po
FROM purchaseorder p
```

```
SELECT top_price, XMLQUERY (
    'for $cost in /buyer/contract/item/amount
     where /buyer/name = $var1 return $cost'
    PASSING BY VALUE 'A.Eisenberg' AS var1, buying_agents
    RETURNING SEQUENCE BY VALUE )
FROM buyers
```

Figure 2.2: Using SQL/XML to generate XML from a relational table (top, [61]) and to evaluate XQuery (bottom, [34])

Expressions (RCTEs) [39, 3] brought standard SQL a general and powerful facility for recursive fixpoint queries. Furthermore, many alternative data models and languages incorporating hierarchies more or less natively have emerged, such as Multidimensional Expressions (MDX), XPath, XQuery, and SQL/XML. Upon closer inspection, however, neither of the existing approaches stands up to our requirements. In the following we discuss the assets and drawbacks of the strongest existing solutions we found.

XML. Storing and querying XML—an inherently hierarchical data format—and the idea of bridging the technology stacks of the relational and XML worlds have received a lot of attention in the research community. One research track pursues the idea of adapting RDBMSs for storing XML fragments and evaluating XPath and XQuery based on relational query processing techniques [15, 42]. Beyond that, the idea of joining the XML and relational data models in order to enable queries over both tables and XML documents has resulted in the SQL/XML standard [4], which integrates XML support into SQL. It has been implemented by prominent vendors [61, 75, 13]. Figure 2.2 depicts two example SQL/XML queries. So-called publishing functions enable the user to generate XML from relational input data (top example). Conversely, XQuery can be used within SQL statements to extract

data from an XML fragment and produce either XML or a relational view (bottom example). While SQL/XML is the tool of choice for working with XML within RDBMSs, it cannot hide the fact that the underlying data models were not designed for interoperability in the first place. "Casting" data from a relational to an XML format or vice versa induces a lot of syntactic overhead, as the many `XML...` clauses cluttering the top example of Figure 2.2 attest. In addition, SQL/XML requires users to know both data models and the respective query languages, which is a challenge to SQL-only users. Since our requirements #1 and #2 mandate that the data model and query language should blend seamlessly with SQL, we do not consider SQL/XML a candidate for general hierarchy support in a relational system. That said, we recognize that many techniques from the XML field, such as join operators and labeling schemes, can be leveraged for our purpose.

hierarchyid is a variable-length system data type introduced in Microsoft SQL Server 2008 [72, 1] whose values represent a position in an ordered tree using ORDPATH [74], a compact and update-friendly path-based encoding scheme. The feature is apparently a by-product of SQL Server's XML support and as such a good demonstration that XML technology can be leveraged for more general uses. The data type provides methods for working with nodes, such as `GetLevel` and `IsDescendantOf`. Nodes can be inserted and relocated by generating new values using, for example, methods `GetReparentedValue` and `GetDescendant`. From a syntax and data model perspective, this is the related work that is most similar to what we present in this chapter. However, there are several major differences to our design: The `hierarchyid` field is provided as a simple tool for modeling a hierarchy; yet, a collection of rows with such a field does *not* necessarily represent a valid hierarchy. It is up to the user to generate and manage the values in a reasonable way. By design, the system does not enforce the structural integrity of the represented tree. For example, it does not guarantee the uniqueness of generated values, and it does not prevent accidentally "orphaning" a subtree by deleting its parent node. In contrast, we require the system to ensure structural integrity at any time (Req. #7), so that queries on hierarchies will not yield surprising results. Of course, this design choice comes at a price, as consistency has to be checked on each update. As another difference, we opt to provide flexibility regarding the underlying indexing scheme. Rather than hardwiring a particular scheme such as ORDPATH, our design allows a scheme to be chosen according to the application scenario at hand. ORDPATH has

the inherent deficiency that relocating a subtree incurs changes to *all* `hierarchyid` values in that subtree (cf. Req. #5).

Hierarchical Queries in Oracle Database [2] extend the `SELECT` statement by the constructs `START WITH`, `CONNECT BY`, and `ORDER SIBLINGS BY`. The wording of these constructs and the related built-in functions and pseudo-columns such as `ROOT`, `IS_LEAF`, and `LEVEL` clearly hint at their intended use for traversing hierarchical data in the Adjacency List format. The underlying recursion mechanism is conceptually similar to RCTEs. Most functionality can in fact be expressed straightforwardly using RCTEs [68], so the discussion in the following paragraphs applies to Hierarchical Queries as well.

Recursive Common Table Expressions are a standard tool that can, among other things, be used for working with a table in the Adjacency List format. We refer to this particular combination as the "RCTE-based approach". As a generic mechanism, RCTEs do in fact have interesting uses far beyond traversing hierarchical data, and we by no means intend to render them obsolete. To convey an impression of how our design and the RCTE-based approach differ, we revisit the BOM example from Figure 2.1. Consider the following query, which is derived from a customer scenario:

"Select all combinations (e, r, c) of an engine e, a rotor r, and a compound part c, such that e contains r, and r is contained in c."

In the example BOM, the qualifying node triples are $(\mathrm{B2}, \mathrm{D2}, \mathrm{C3})$, $(\mathrm{B2}, \mathrm{D2}, \mathrm{A1})$, and $(\mathrm{B1}, \mathrm{C2}, \mathrm{A1})$. This query is not entirely trivial in that it involves not only two, but three nodes that are tested for hierarchical relationships. Figure 2.3 shows an RCTE-based solution. It selects the `id` and a `payload` of each node. We use two RCTEs: one starting from an engine e and navigating downwards from e to r, the other navigating upwards from r to c. Obviously, the statement is not particularly intelligible to readers, and somewhat tedious to write down in the first place. What's more, it is not the only way to solve the problem. An alternative would be to fully materialize all ancestor/descendant combinations (u, v) using a single RCTE and then work (non-recursively) on the resulting table. This option is generally inferior due to the large intermediate result, though it might be feasible if only a small hierarchy is involved. The point is that it is up to the user to choose the most appropriate strategy for answering the query. The first choice to make is the basic approach to use: one RCTE materializing all pairs versus two RCTEs as

```
WITH RECURSIVE ER (id, pl, r_id, r_pl, r_kind) AS (
   SELECT e.id, e.payload, e.id, e.payload, e.kind
     FROM BOM e
    WHERE e.kind = 'engine'
    UNION ALL
   SELECT e.id, e.pl, r.id, r.payload, r.kind
     FROM BOM r
     JOIN ER e ON r.pid = e.r_id
),
CER (e_id, e_pl, r_id, r_pl,
     c_id, c_pl, c_kind, pid) AS (
   SELECT id,pl,r_id,r_pl,r_id,r_pl,r_kind,r_id
     FROM ER
    WHERE r_kind = 'rotor'
    UNION ALL
   SELECT e.e_id, e.e_pl, e.r_id, e.r_pl, c.id, c.payload, c.kind, c.pid
     FROM BOM c
     JOIN CER e ON e.pid = c.id
)
SELECT e_id, e_pl, r_id, r_pl, c_id, c_pl
  FROM CER
 WHERE c_kind = 'compound'
```

Figure 2.3: Example hierarchical query, expressed using two RCTEs.

in the example. The second choice is the join direction to proceed in: towards the root versus away from the root. The query optimizer is tightly constrained by the approach the user is prescribing. In fact, the query statement is *imperative* rather than declarative: A bad choice from the user's side can easily result in incorrect answers or severe performance penalties.

Furthermore, as a direct consequence of the underlying Adjacency List model, navigation axes other than *descendant* and *ancestor* (e. g., the XPath axis *following*) are inherently difficult to express. And in order to query hierarchical properties such as the level of a node, the user must do the computation manually using arithmetics within the RCTE. Even though the flexibility of RCTEs allows the user to perform arbitrary computations, seemingly basic tasks, such as depth-first sorting, can be surprisingly difficult to express, let alone evaluate. All in all, writing RCTEs to express non-trivial queries is an "expert-friendly" and error-prone task in terms of achieving correctness, intelligibility, and robust performance.

In addition, the RCTE-based approach bears some inherent inefficiencies. We give two examples: First, note that even though in Figure 2.3 we are interested

```
SELECT e.id, e.payload, r.id, r.payload, c.id, c.payload
  FROM BOM e, BOM r, BOM c
 WHERE e.kind = 'engine'
   AND IS_DESCENDANT(r.pos, e.pos)
   AND r.kind = 'rotor'
   AND IS_ANCESTOR(c.pos, r.pos)
   AND c.kind = 'compound'
```

Figure 2.4: The example query from Figure 2.3, expressed using the proposed SQL language extensions.

only in qualifying ancestor/descendant pairs (e, r) and (r, c), but *not* in any nodes in between, the RCTE necessarily touches all intermediate nodes anyway. Second, attributes of interest to the user (`payload` in the example) must often be materialized early and carried along throughout the recursion, which is costly.

From RCTEs to our syntax. Figure 2.4 shows how the example query is expressed in the syntax we introduce in Section 2.3 and Section 2.4. In a nutshell, the `pos` field of type `NODE` identifies a row's position in the depicted hierarchy, which makes `BOM` a *hierarchical table*. Without delving into details, we illustrate three major cornerstones of our design: First, with the RCTE-based approach, the task of "discovering" and navigating the hierarchy structure on the one hand, and the task of actually using the hierarchy to compute hierarchical properties of interest on the other hand, are inseparably intertwined. By contrast, the query statement in Figure 2.4 makes use of an *available* hierarchical table exposing the `pos` field. The hierarchy structure is known ahead and persisted. The task of querying the hierarchy is cleanly separated from the task of specifying and building the hierarchy structure. Thus, any duplication of "discovery logic" in user code is avoided. Second, unlike the generic RCTE mechanism, our syntax is particularly tailored for working with hierarchies. This way we can provide increased intelligibility, user-friendliness, and expressiveness, and we can employ particularly tuned data structures and algorithms on the back end side. Third, our syntax states the hierarchical relationships clearly and in a *declarative* way (e. g., `IS_DESCENDANT`). This allows the query optimizer to reason about the user's intent and pick an optimal evaluation (i. e., join) strategy and direction.

The example shows how we achieve requirements #1, #2, and #3: Our syntax blends with SQL (#1), as we stick to joins and built-in functions to provide all required query support (#3). As a corollary, the syntactic impact is minimal (#2).

In fact, a hierarchy query does not need *any* extensions to the SQL grammar. Still, the syntax is highly expressive (#2): the query in Figure 2.4 reads just like the English sentence defining it.

2.3 Hierarchical Tables: Our Model

Basic Terms. We use the term *hierarchy* to denote an *ordered, rooted, labeled tree.* The tree in Figure 2.1 is an example. *Labeled* means each vertex in the tree has a label, which represents the attached data. *Rooted* means a specific node is marked as root, and all edges are conceptually oriented away from the root. We require that every hierarchy contains by default a single, virtual root node, which we denote by $\top$ and call the *super-root.* As a virtual node, $\top$ is hidden from the user. The children of $\top$ are the actual roots in the user data. Through this mechanism we avoid certain technical complications in handling empty hierarchies as well as hierarchies with multiple roots, so-called *forests.* Furthermore, a hierarchy is *ordered*, that is, a total order is defined among the children of each node. That said, for many applications the relative order of siblings is actually not relevant. While we recognize this use case by providing order-indifferent update operations, the system always maintains an internal order. This way order-based functionality such as pre-order ranking is well-defined and deterministic.

Hierarchical Tables. In a database context, a hierarchy is not an isolated object but rather closely tied to an associated table. A hierarchy has exactly *one* associated table. (Of course, additional tables can be tied to a hierarchy by using joins; cf. Section 2.6.) Conversely, a table might have multiple associated hierarchies. In Figure 2.1, for instance, table `BOM` has one associated hierarchy, which arranges its rows in a tree, thus adding a *hierarchical dimension* to `BOM`. We call a table with at least one associated hierarchy a *hierarchical table.* Let H be a hierarchy attached to a table T. Each row r of T is associated with *at most one* node v of H, so there may also be rows that do not appear in the hierarchy. Conversely, each node except for $\top$ is associated with *exactly one* row of T. The values in the fields of r can be regarded as labels attached to v or to the edge onto v. Besides the tree structure and a node–row association, H conceptually does not contain any data. A user never works with the hierarchy object H itself but only works with the associated

table T. Consequently, a row-to-node handle is required to enable the user to refer to the nodes in H. Such a handle is provided by a column of type `NODE` in T.

The `NODE` Data Type. A field of the predefined data type `NODE` represents the position of a row's associated node within the hierarchy. A table (row) can easily have two or more `NODE` fields and thus be part of multiple distinct hierarchies. Using an explicit column to serve as handle for a hierarchical dimension is a cornerstone of our design. We can expose all hierarchy-specific functionality through that column in a very natural and lightweight way. The following pseudo-code illustrates this for table `BOM` with its `NODE` column named `pos`:

```
SELECT id, ..., "level of pos"
  FROM BOM
 WHERE "pos is a leaf"
```

Compared to other conceivable approaches, such as introducing a pseudo-column for each property of interest (similar to the `LEVEL` column in Oracle Hierarchical Queries), or functions operating on table aliases (an idea mandated by early proposals for temporal SQL), the `NODE` field implicates minimal syntactic impact and also simplifies certain aspects: Transporting "hierarchy information" across a SQL view is a trivial matter of including the `NODE` column in the projection list of the defining `SELECT` statement. Furthermore, the functionality can be extended in the future by simply defining new functions operating on `NODE`.

Actual values of data type `NODE` are opaque and not directly observable; a naked `NODE` field must not be part of the output of a top-level query. The user may think of a `NODE` value as "the position of this row in the hierarchy". How this position is encoded is intentionally left unspecified. This leaves maximum flexibility and optimization opportunities to the back end.

The user works with a `NODE` column exclusively by applying hierarchical functions and predicates such as "level of" and "is ancestor of". Besides that, the `NODE` type supports only the operators `=` and `<>`. Other operations such as arithmetics and casts from other data types are not allowed. The system statically tracks the original hierarchy of each `NODE` column and ensures that binary predicates and set operations (e. g., `UNION`) do not mix `NODE` values from different hierarchies. `NODE` values can be `NULL` to express that a row is *not* part of the hierarchy. Non-null values always encode a valid position in the hierarchy. The handling of `NULL` values during query processing is consistent with SQL semantics.

ID	LEVEL	IS_LEAF	IS_ROOT	PRE_RANK	POST_RANK
'A1'	1	0	1	1	10
'B1'	2	0	0	2	3
'C1'	3	1	0	3	1
'C2'	3	1	0	4	2
'B2'	2	0	0	5	9
'C3'	3	0	0	6	6
'D1'	4	1	0	7	4
'D2'	4	1	0	8	5
'C4'	3	0	0	9	8
'D3'	4	1	0	10	7
'A2'	1	1	1	11	11

Table 2.1: Projecting hierarchy properties of BOM.

2.4 Querying Hierarchies

To meet Requirement #3 to support and facilitate complex queries, we enhance SQL's query language. As outlined in the previous section, a field of data type NODE serves as handle to the nodes in the associated hierarchy. For the following, we suppose a table with such a field (like BOM and pos) is at hand. How to obtain such a table—either a hierarchical base table or a derived hierarchy—is covered by Section 2.5.

We provide built-in scalar functions operating on a NODE value v to enable the user to query certain *hierarchy properties*:

LEVEL(v) — The number of edges on the path from $\top$ to v.

IS_LEAF(v) — Whether v is a leaf, i. e., has no children.

IS_ROOT(v) — Whether v is a root, i. e., its parent is $\top$.

PRE_RANK(v) — The pre-order traversal rank of v.

POST_RANK(v) — The post-order traversal rank of v.

Table 2.1 shows the result of projecting all these properties for BOM. The values of LEVEL, PRE_RANK, and POST_RANK are 1-based. There are certain more or less obvious equivalences. For example, IS_ROOT(v) is equivalent to LEVEL(v) = 1 and thus redundant, strictly speaking. However, for the sake of convenience and expressiveness we do *not* aim for a strictly orthogonal function set.

The following example demonstrates how hierarchy properties are used; it produces a table of all non-composite parts (i. e., leaves) and their respective levels:

```
SELECT id, LEVEL(pos) AS level
  FROM BOM
 WHERE IS_LEAF(pos) = 1
```

As mandated by SQL semantics, the order of the result rows is undefined. To traverse a hierarchy in a particular order, one can combine `ORDER BY` with a hierarchy property. For example, consider a so-called parts explosion for the BOM, which shows all parts in depth-first order, down to a certain level:

```
-- Depth-first, depth-limited parts explosion with level numbers
SELECT id, LEVEL(pos) AS level
  FROM BOM
 WHERE LEVEL(pos) < 5
 ORDER BY PRE_RANK(pos)
```

With `PRE_RANK`, parents are arranged before children (in *pre-order*); with `POST_RANK`, children are arranged before parents (in *post-order*). Sorting in breadth-first search order can be done using the `LEVEL` property:

```
-- Breadth-first parts explosion
SELECT id, LEVEL(pos) AS level
  FROM BOM
 ORDER BY LEVEL(pos)
```

Note that computing the actual pre- or post-order rank of a node is not trivial for many indexing schemes (e. g., ORDPATH). However, when `PRE_RANK` or `POST_RANK` appear only in the `ORDER BY` clause (which is their main use case), then there is no need to actually compute the values. For sorting purposes, pairwise comparison of the pre/post positions is sufficient, and all indexing schemes we use can handle this efficiently.

Besides querying hierarchy properties, a general task is to navigate from a given set of nodes along a certain hierarchy *axis*. Such axes can be expressed using one of the following *hierarchy predicates* (with u and v being `NODE` values):

`IS_PARENT(`u`,`v`)` — whether u is a parent of v.

`IS_CHILD(`u`,`v`)` — whether u is a child of v.

`IS_SIBLING(`u`,`v`)` — whether u is a sibling of v, i. e., has the same parent.

`IS_ANCESTOR(`u`,`v`)` — whether u is an ancestor of v.

`IS_DESCENDANT(`u`,`v`)` — whether u is a descendant of v.

`IS_PRECEDING(`u`,`v`)` — whether u precedes v in pre-order and is no ancestor of v.

`IS_FOLLOWING(`u`,`v`)` — whether u follows v in pre-order and is no descendant of v.

The task of axis navigation maps quite naturally onto a self-join with an appropriate hierarchy predicate as join condition. For example, the following lists all pairs (u, v) of nodes where u is a descendant of v:

```
SELECT u.id, v.id
  FROM BOM u
  JOIN BOM v
    ON IS_DESCENDANT(u.pos, v.pos)
```

As another example, we can use a join to answer the classic *where-used* query on a BOM. The query "Where is part D2 used?" corresponds to enumerating all ancestors of said node:

```
SELECT a.id
  FROM BOM p, BOM a
 WHERE IS_ANCESTOR(a.pos, p.pos)
   AND p.id = 'D2'
```

The different predicates are inspired by the axis steps known from XPath. Note that the *preceding* and *following* predicates are only meaningful in an ordered hierarchy, and thus of less interest in the general case.

The functions presented here are chosen based on the customer scenarios we have analyzed and the capabilities of the indexing schemes we have considered. Further functions might be added in the future.

2.5 Creating and Manipulating Hierarchies

The previous section describes query primitives that work on a field of type `NODE`. In this section, we show how such fields are declared and maintained.

2.5.1 Deriving a Hierarchy from an Adjacency List

According to Requirement #6, legacy applications demand for a means to *derive* a hierarchy from an available table in the Adjacency List format. Derived hierarchies enable users to take advantage of all query functionality "ad hoc" on the basis of relationally encoded hierarchical data, while staying entirely within the QL (and in particular, without requiring schema modifications via DDL). For this purpose we provide the `HIERARCHY` expression. It derives a hierarchy from a given adjacency-list-formatted *source_table*, which may be a table, a view, or the result of a subquery:

```
 HIERARCHY
 USING source_table AS source_name
[START WHERE start_condition]
 JOIN PARENT parent_name ON join_condition
[SEARCH BY order]
 SET node_column_name
```

This expression can be used wherever a table reference is allowed (in particular, a `FROM` clause). Its result is a temporary table containing the data from the *source_table* plus an additional `NODE` column named *node_column_name*. The expression is evaluated by first self-joining the *source_table* in order to derive a parent-child relation representing the edges, then building a temporary hierarchy representation from that, and finally producing the corresponding `NODE` column. The `START WHERE` subclause can be used to restrict the hierarchy to only the nodes that are reachable from any node satisfying *start_condition*. The `SEARCH BY` subclause can be used to specify a desired sibling order; if omitted, siblings are ordered arbitrarily. Conceptually, the procedure for evaluating the whole expression is as follows:

1. Evaluate *source_table* and materialize required columns into a temporary table T. Also add a `NODE` column named *node_column_name* to T.

2. Perform the join

   T `AS` C `LEFT OUTER JOIN` T `AS` P `ON` *join_condition*,

   where P is the *parent_name* and C is the *source_name*. Within the *join_condition*, P and C can be used to refer to the parent and the child node, respectively.

3. Build a directed graph G containing all row IDs of T as nodes, and add an edge $r_P \to r_C$ between any two rows r_P and r_C that are matched through the join.

4. Traverse G, starting at rows satisfying *start_condition*, if specified, or otherwise at rows that have no (right) partner through the outer join. If *order* is specified, visit siblings in that order. Check whether the traversed edges form a valid tree or forest, that is, there are no cycles and no node has more than one parent. Raise an error when a non-tree edge is encountered.

5. Build a hierarchy representation from all edges visited in Step 4 and populate the `NODE` column of T accordingly. The result of the `HIERARCHY` expression is T.

Note that the description above is merely conceptual; we describe an efficient implementation in Section 2.7.3. As described, an error is raised when a non-tree edge is encountered. This way we ensure the resulting hierarchy has a valid tree structure (Req. #7). In our prototype, we also support "non-strict" hierarchies by deriving a spanning tree over G, with various options controlling the way the spanning tree is chosen. We omit these advanced options for the sake of brevity.

The `HIERARCHY` syntax is intentionally close to an RCTE and even more so to Oracle Hierarchical Queries. (The self-join via *parent_name* is comparable to a `CONNECT BY` via `PRIOR` in a Hierarchical Query.) However, the semantics are quite different in that by design only a *single* self-join is performed on the input rather than a recursive join. As our experiments show, this allows for a very efficient evaluation algorithm compared to a recursive join. Furthermore, there is a major conceptual difference to the mentioned approaches: The `HIERARCHY` expression does nothing more than define a hierarchy. That hierarchy can be queried by wrapping the expression into a `SELECT` statement. In contrast, a RCTE both defines and queries a hierarchy in one convoluted statement. We believe that separating these two aspects greatly increases comprehensibility. As an example, consider again the BOM of Figure 2.1. The following statement uses a CTE to derive the `pos` column from `id` and `pid`, then selects the `id` and `level` of all parts that appear within part C2:

```
WITH PartHierarchy AS (
   SELECT id, pos
     FROM HIERARCHY USING BOM AS c
          JOIN PARENT p ON p.id = c.pid
          SET pos
)
SELECT v.id, LEVEL(v.pos) AS level
  FROM PartHierarchy u, PartHierarchy v
 WHERE u.id = 'C2'
   AND IS_DESCENDANT(v.pos, u.pos)
```

The mentioned separation of aspects is clearly visible. `PartHierarchy` could be extracted into a view and reused for different queries. One might argue that a RCTE or Hierarchical Query could as well be placed in a view, but that would still not result in a clear definition/query separation, because any potentially needed hierarchy properties (such as `LEVEL` in the example) would have to be computed in the view *definition* even though they are clearly part of the *query*. A query that does not need the level would still trigger its computation, resulting in unnecessary

overhead. In contrast, our design allows for deferring the selection of hierarchy properties to the query.

2.5.2 Hierarchical Base Tables

Derived hierarchies as discussed in the previous section are targeted mainly at legacy applications. For newly designed applications a preferable approach is to express and maintain a hierarchy explicitly in the table schema. We provide specific DDL constructs for this purpose (Req. #4). The user can include a *hierarchical dimension* in a base table definition:

```
CREATE TABLE T (
  ...,
  HIERARCHY name [NULL|NOT NULL] [WITH (option*)]
)
```

This implicitly adds a column named *name* of type `NODE` to the table, exposing the underlying hierarchy. Explicitly adding columns of type `NODE` is prohibited. A hierarchical dimension can also be added to or dropped from an existing table using `ALTER TABLE`. Like a column, a hierarchical dimension can optionally be declared nullable. If it is declared `NOT NULL`, the implicit `NODE` value of a newly inserted row is `DEFAULT`, making it a new root without children. A row with a `NULL` value in its `NODE` field is not part of the hierarchy.

A hierarchy that is known to be static allows the system to employ a read-optimized indexing scheme (cf. Req. #8). Therefore, we provide the user with a means of controlling the degree to which updates to the hierarchy are to be allowed. This is done through an *option* named `UPDATES`:

```
UPDATES = BULK|NODE|SUBTREE
```

`BULK` allows only bulk-updates; `NODE` allows bulk-updates and single-node operations, that is, relocating, adding, and removing single leaf nodes; `SUBTREE` allows bulk-updates, single-node operations, and the relocation of whole subtrees. A `BULK` dimension is basically *static*; individual updates are prohibited. We furthermore make a distinction between single-node and subtree updates, because subtree updates require a more powerful dynamic indexing scheme than single-node updates, with inevitable tradeoffs in query performance (cf. Section 2.7.1). Depending on the option, the system chooses an appropriate indexing scheme for the hierarchical dimension. The default setting is `SUBTREE`, so full update flexibility is provided unless restricted explicitly by the user.

2.5.3 Manipulating Hierarchies

For legacy application support (Req. #6), we aim to provide a smooth transition path from relationally encoded hierarchies (i. e., adjacency lists) to full-fledged hierarchical dimensions. In a first stage, we expect most legacy applications to rely entirely on views featuring `HIERARCHY` expressions on top of adjacency lists, thus avoiding any schema changes. Hence, bulk-building is, at least conceptually, used on *each* view evaluation; though it may be elided often in practice, since HANA employs view caching. In a second stage, a partly adapted legacy application might add a static (`UPDATES = BULK`) hierarchical dimension alongside an existing adjacency list encoding, and update the dimension periodically from the adjacency list via an explicit *bulk-update*. A bulk-update is issued by using a `HIERARCHY` expression as source table of a `MERGE INTO` statement. (We do not discuss this in detail for brevity reasons.) These two stages provide a way to gradually adopt hierarchy functionality in a legacy application, but they come at the cost of frequently performing bulk-builds whenever the hierarchy structure changes. Therefore, for green-field applications as well as for fully migrated legacy applications, a *dynamic* hierarchy (`UPDATES = NODE` or `SUBTREE`) supporting explicit, fine-grained updates via special-purpose DML constructs is preferable (Req. #5). Again, we strive for a minimally invasive syntax: We use ordinary `INSERT` and `UPDATE` statements operating on the `NODE` column of a hierarchical dimension to express updates.

Inserting. To specify the position where a new row is to be inserted into the hierarchy, we use an *anchor* value. Again, we refrain from extending the SQL grammar and define new built-in functions that take a `NODE` as input and yield an *anchor*. An anchor can be used as value for the `NODE` field in an `INSERT` statement. We support the following anchor functions:

`BELOW(`v`)` inserts the new row as child of v. The insert position among siblings is undefined.

`BEFORE(`v`)` or `BEHIND(`v`)` insert the new row as immediate left or right sibling of v.

For example, we can add a node B3 as new child of A2 into the hierarchy of Figure 2.1 like this:

```
INSERT INTO BOM (id, pos)
VALUES ('B3', BELOW(SELECT pos FROM BOM WHERE id = 'A2'))
```

The BELOW anchor is useful for unordered hierarchies, while the BEFORE and BEHIND anchors allow for precise positioning among siblings in hierarchies where sibling order matters.

The user can also use DEFAULT to make the new row a root, or NULL (for nullable dimensions) to omit it from the hierarchy.

Relocating. Relocating a node v is done by issuing an ordinary UPDATE on the NODE field of the associated row, again using an anchor to describe the node's target position. If v has any descendants, they are moved together with v, so the whole subtree rooted at v is relocated. Relocating a subtree is only allowed if option UPDATES = SUBTREE is used for the hierarchical dimension. In order to guarantee structural integrity, the system must prohibit relocation of a subtree below a node within that same subtree, as this would result in a cycle.

Removing. A node can be removed from a hierarchy by either deleting its row or setting the NODE field to NULL. However, these operations are prohibited if the node has any descendants that are not also removed by the same transaction. To remove a node with descendants, all children have to be relocated first or removed with that node. While this is necessary to ensure that removing nodes does not leave behind an invalid hierarchy, it is very restrictive: If a hierarchical dimension uses option UPDATES = BULK, the only rows that may be deleted are those whose NODE value is NULL; the user is prevented from deleting any rows that take part in the hierarchy. To make easy row deletion possible in this case, we allow truncating the whole hierarchy by setting the NODE value of *all* rows to NULL within the same transaction. Then, rows may be deleted at will, and subsequently the hierarchy can be rebuilt (bulk-built) from scratch. These rules ensure that the structure of the hierarchy remains valid at any time, thus satisfying Requirement #7.

2.6 Advanced Customer Scenarios

In this section we explore some advanced techniques for modeling entities that are part of multiple hierarchies, entities that appear in the same hierarchy multiple times, and inhomogeneous hierarchies that contain entities of various types. The queries are inspired by customer scenarios and demonstrate that our language extensions stand up to non-trivial, real-world queries.

Flexible Forms of Hierarchies. In certain applications an entity might be designed to belong to two or even more hierarchies. For example, an employee might have both a disciplinary superior as well as a line manager, and thus be part of two reporting lines. A straightforward way to model this is to use two hierarchical dimensions:

```
CREATE TABLE Employee (
   id INTEGER PRIMARY KEY, ...,
   HIERARCHY disciplinary,
   HIERARCHY line
)
```

A more complex case arises when a hierarchy shall contain certain rows more than once. Again, a bill of materials is a good example: A common part such as a screw generally appears multiple times within the same BOM, and we do not want to replicate its attributes each time. This is a typical $1:n$ relationship: one part can appear many times in the hierarchy. As our data model blends seamlessly with SQL, the solution is to model this case exactly as one would model $1:n$ relationships in SQL, namely by introducing two relations and linking them by means of a foreign key constraint. Thus, we separate the schema from Figure 2.1 into per-part data `Part` and a separate `BOM` table:

```
CREATE TABLE Part (
   id INTEGER PRIMARY KEY,
   kind VARCHAR(16),
   price INTEGER, ...      -- master per-part data
)

CREATE TABLE BOM (
   node_id INTEGER PRIMARY KEY,
   HIERARCHY pos,
   part_id INTEGER,    -- a node is a part (N:1)
   FOREIGN KEY (part_id) REFERENCES Part (id),
   ...   -- additional node attributes
)
```

Heterogeneous Hierarchies. Often, entities of different types are mixed in a single hierarchy. “Different types” means that the entities are characterized by different sets of attributes. Especially in XML documents, it is very common to have various node types (i. e., tags with corresponding attributes), and XPath expressions routinely interleave navigation with filtering by node type (so-called node tests). The SQL way of modeling multiple entity types is to define a separate table per

```
SELECT *
  FROM BOM c,          -- compound node
       Part cm,        -- compound master data
       BOM f,          -- fitting node
       Part fm,        -- fitting master data
       BOM e,          -- engine node
       Engine em       -- engine master data
 WHERE c.id = cm.id
   AND cm.kind = 'compound'
   AND IS_DESCENDANT(f.pos, c.pos)
   AND f.id = fm.id
   AND fm.kind = 'fitting'
   AND fm.manufacturer = 'X'
   AND IS_DESCENDANT(e.pos, f.pos)
   AND e.id = em.id
   AND em.power > 700
```

Figure 2.5: Querying a heterogeneous hierarchy

entity type, each with an appropriate set of columns. Returning to our BOM, we further enhance the `Part`–`BOM` data model with master data specific to engines:

```
CREATE TABLE Engine (
   id INTEGER PRIMARY KEY,
   FOREIGN KEY (id) REFERENCES Part (id),
   power INTEGER, ... -- master data
)
```

While `Part` contains master data common to all parts, `Engine` adds master data that is specific to parts of kind "engine". Both tables necessarily share their primary key domain (`id`). `BOM` is now a *heterogeneous* hierarchy in that each node has a type: it is either a general `Part` or an `Engine`. This design is extensible. Further part types can be added by defining further tables like `Engine` with $1:1$ relationships to `Part`.

While working with a BOM, the user can use type-specific part attributes for filtering purposes simply by joining in the corresponding master data. As an example, suppose that fittings by manufacturer X have been reported to outwear too quickly when used in combination with engines more powerful than 700 watts, and we need to determine the compounds that contain this hazardous combination in order to issue a recall. Figure 2.5 shows the solution. Note that the `BOM`–`Engine` join implies the test that node `e` is of kind `'engine'`.

Dimension Hierarchies. A major use case for hierarchies is arranging some keys that are used as dimensions for a fact table. Measures associated with the facts are to be aggregated alongside the dimension hierarchies. As an example, consider a sales table recording, besides a certain sales amount and other attributes, the store where each sale took place. Suppose stores are arranged in a geographic hierarchy. The schema is:

```
Sale :  {[ store_id, date, amount, ...  ]}
Store :  {[ id, location_id, ...  ]}
Location :  {[ id, pos, name, ...  ]}
```

By joining `Sale`—`Store`—`Location`, we can associate each sale with a `NODE` value (`Location.pos`) of the location hierarchy indicating where the sale took place. Suppose we would like to answer the query: "Considering only sales within *Europe*, list the total sales per sub-subregion." This query speaks, quite implicitly, of *three* distinct Location nodes: a reference node u, namely *Europe*; the set of nodes V two levels below u, corresponding to the sub-subregions; and the sets of nodes W_v below each $v \in V$, corresponding to locations of stores where a sale took place. We are explicitly interested in the nodes in V, but also need a name for a node $w \in W_v$ in order to specify the association of w to a sale, so that we can ultimately compute a sum over the sales amount. All in all, three self-joined instances of the hierarchical table are required:

```
SELECT v.id, SUM(sale.amount)
  FROM Location u, Location v, Location w, Store store, Sale sale,
 WHERE u.name = 'Europe'
   AND IS_DESCENDANT(v.pos, u.pos)
   AND LEVEL(v.pos) = LEVEL(u.pos) + 2
   AND IS_DESCENDANT(w.pos, v.pos)
   AND IS_LEAF(w.pos) = 1  -- store locations are leaves
   AND w.id = store.location_id
   AND store.id = sale.store_id
 GROUP BY v.id;
```

Note the straightforward reading direction, which intuitively matches the direction of navigation in the hierarchy. This example and the one from Figure 2.5 in particular show how our language extensions maintain the join "look and feel" of SQL, so even large queries look familiar to SQL programmers.

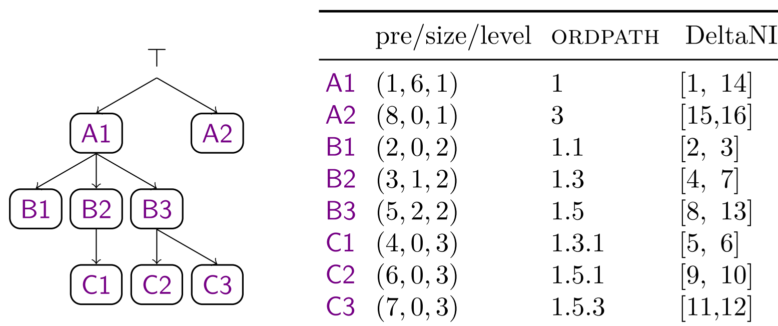
2.7 Architecture and Implementation Aspects

On the back end side, the foundation for implementing the functionality described in sections 2.4 and 2.5 is the *hierarchy indexing scheme* underlying each hierarchical dimension. As Requirement #8 anticipates, no single scheme can serve all application scenarios equally well; there is no "one size fits all" solution. Thus, our design leaves the system the choice among different indexing schemes. Each scheme comes with a set of built-in implementations of the hierarchy functions (e. g., `LEVEL`). For efficient query processing, we employ hierarchy-aware join operators that work well with all supported indexing schemes (cf. Section 3.3). The *bulk-building operation* is in large parts common to all indexing schemes. It is also particularly important for supporting derived hierarchies (Section 2.5.1) and thus legacy applications. Therefore, we cover this operation in detail (Section 2.7.3). Due to space constraints and since the primary focus of this chapter is on the data model and our language extensions, we omit certain technical details and refer to cited works. Our intention is to convey a general intuition of how our concepts can be implemented efficiently. Chapter 3 will then go into detail.

2.7.1 Hierarchy Indexing Schemes

In our framework, a hierarchy indexing scheme comprises the content of a `NODE` column and possibly an auxiliary data structure. It contains the hierarchy structure as non-redundant information. This is in contrast to traditional indexes such as B-trees, which are entirely redundant auxiliary data structures. What data is actually stored in the `NODE` column depends on the chosen indexing scheme. This is why we explicitly specify `NODE` as *opaque* to the user (cf. Section 2.3).

Indexing schemes of varying complexity and sophistication are conceivable: Among the simplest indexing schemes are those based on *labeling schemes*; they are "simple" in that the labels can be stored directly in the `NODE` column (and possibly indexed using ordinary database indexes); no special-purpose data structures are required. Labeling schemes have been studied extensively in the XML context. Two prominent subcategories are *order-based schemes* as studied by Grust et al. [41], and *path-based schemes* such as ORDPATH [74]. In our prototype we have implemented a simple yet effective order-based variant: the pre/size/level scheme (PSL) [15], where we label each node with its pre-order rank, subtree size, and level. We have



	pre/size/level	ORDPATH	DeltaNI
A1	$(1,6,1)$	1	[1, 14]
A2	$(8,0,1)$	3	[15,16]
B1	$(2,0,2)$	1.1	[2, 3]
B2	$(3,1,2)$	1.3	[4, 7]
B3	$(5,2,2)$	1.5	[8, 13]
C1	$(4,0,3)$	1.3.1	[5, 6]
C2	$(6,0,3)$	1.5.1	[9, 10]
C3	$(7,0,3)$	1.5.3	[11,12]

Figure 2.6: An example hierarchy and the contents of the `NODE` column using different indexing schemes

also implemented a path-based scheme comparable to ORDPATH. Figure 2.6 depicts the `NODE` column for an example hierarchy using these schemes.

We propose the more sophisticated dynamic indexing schemes Order Indexes and DeltaNI in Chapter 3 and Chapter 4, respectively. Both represent the hierarchy information in special-purpose data structures, and the `NODE` column contains handles into those structures. The figure shows a possible `NODE` column for DeltaNI but omits the auxiliary delta structures.

The choice among indexing schemes matters particularly with regard to their varying degrees of support for updates. For example, while the PSL scheme allows for an efficient evaluation of queries, it is totally *static*: Even a single-node update can, in general, necessitate changes to $\mathcal{O}(n)$ labels of other nodes. This is obviously not feasible for large hierarchies. More complex schemes, on the other hand, trade off query processing efficiency and in return support update operations to a certain degree. Our proposed dynamic indexing schemes support even complex update operations, such as relocating an entire subtree, in $\mathcal{O}(\log n)$ time.

The indexing scheme is meant to be chosen by the DBMS per hierarchical dimension, transparently to the user. The user indirectly influences the choice through the `UPDATES` option (Section 2.5). Our prototype decides as follows: For derived hierarchies, which are by design static, and for immutable hierarchical tables (`UPDATES = BULK`), the obvious choice is PSL. If the user requires support for complex updates (`SUBTREE`), we choose one of our Order Indexes (the BO-Tree). For system-versioned tables, we choose DeltaNI. For ordinary, non-versioned tables, and if the user settles for simple updates (`NODE`), we resort to a read-optimized Order Index (the O-List).

A deeper discussion of indexing is included in Chapter 3. Our main message here is: the design as presented is *extensible* and flexible in that it anticipates further indexing schemes to be plugged in. The user is not burdened with the decision for the optimal scheme; it is up to the DBMS to pick among the available alternatives.

Hierarchy Functions in the SQL statement are translated into operations on the underlying index. Consequently, every indexing scheme must provide the necessary operations. For example, consider the `LEVEL` function: with PSL, we can decode the result directly from the given `NODE` value; with a path-based scheme, we have to count the number of elements in the path. We have carefully chosen the set of functions to be supported such that all important use cases we identified are covered and, at the same time, it is possible to evaluate the functions efficiently on most existing indexing schemes proposed in the literature. All query functionality is built upon the generic query primitives devised in Section 3.1, which can be implemented efficiently for most existing indexing schemes. Most implementations are straightforward and covered in the cited publications. The implementations for our indexing schemes will be covered in the respective chapters.

Updates involving nodes are simply propagated to the index implementations, which update the `NODE` column and the auxiliary data structures accordingly. As we expect most existing applications to rely on derived hierarchies initially, we do not cover individual update operations any further in favor of a detailed discussion of bulk-building.

2.7.2 Hierarchy-Aware Join Operators

Like functions, binary predicates such as `IS_DESCENDANT` can be translated into invocations of the underlying index. But this is not adequate if they are used as join conditions, since the query optimizer would have to resort to nested-loops-based join evaluation. Therefore, we enhance the optimizer such that joins involving a hierarchy predicate are translated into efficient hierarchy-aware physical join operators. Various hierarchy-aware join operators have been proposed in the literature, mostly for XPath processing [6, 43, 19]. Basically any of these operators can be used in our setting, with slight adaptions to account for SQL semantics. An XPath axis step, for example, is implicitly a semi-join and performs duplicate elimination. With SQL, we have to support general joins, and duplicate elimination

is not necessary in the default case. The joins used in our prototype will be explained in Section 3.3.

2.7.3 Bulk-Building

As discussed in Section 2.5, we make extensive use of the bulk-building operation for derived hierarchies on one hand, and for bulk-updates via MERGE on the other hand. Our goal is an efficient implementation of the HIERARCHY expression, whose definition we revisit here:

```
 HIERARCHY
 USING source_table AS source_name
[START WHERE start_condition]
 JOIN PARENT parent_name ON join_condition
[SEARCH BY order]
 SET node_column_name
```

Virtually any indexing scheme we have investigated can be built straightforwardly during a depth-first traversal of the input hierarchy. Thus, the main task of the bulk-build algorithm is to transform the adjacency list from the input table into an intermediate representation that supports efficient depth-first traversal. Building the intermediate representation is common to all indexing schemes; only the final traversal is index-specific. Our prototype reuses existing relational operators for as many aspects as possible, adding as little new code as necessary. The algorithm proceeds as follows:

source_table is evaluated and the result is materialized into a temporary table T. For this we use an ordinary TEMP operator. To construct the hierarchy edges, we evaluate T AS C LEFT OUTER JOIN T AS P ON *join_condition*. The left join input C represents the child node and the right input P the parent node of an edge. Since it is an *outer* join, we also select children without a parent node. In the absence of a *start_condition*, these nodes are by default the roots of the hierarchy. We include the row IDs r_P and r_C of both join sides in the result for later use. r_P can be NULL due to the outer join. If *order* is specified, we use an ordinary SORT operator to sort the join result. Next, we remove all columns except for r_P and r_C, so what we have at this point is a stream of parent/child pairs (i. e., edges) in the desired sibling order.

Next, building and traversing the intermediate representation is taken over by a new operator, *hierarchy build* β. The steps performed for this operator to build

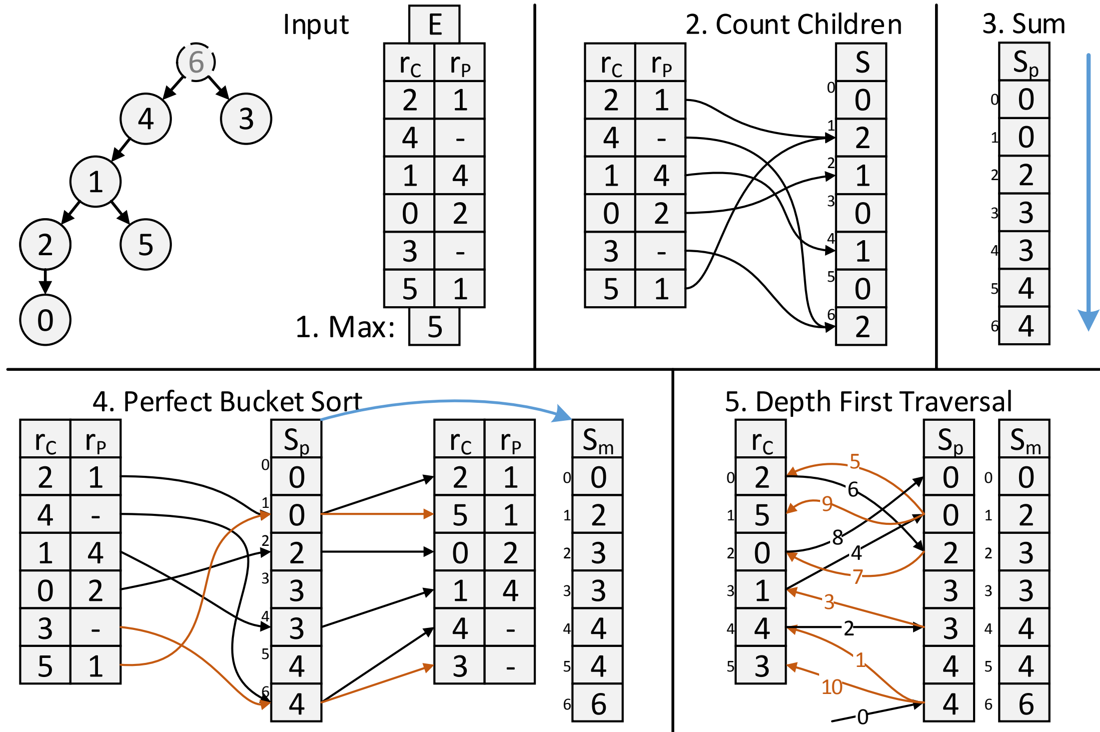


Figure 2.7: Steps of the bulk build algorithm

an example hierarchy are depicted in Figure 2.7. The hierarchy shown on the left depicts the hierarchy that is to be built. The numbers inside the nodes are their row-ids. The virtual root node 6 is not contained in the input. It is used during the algorithm to become the parent of all nodes that are roots in the input.

First, the operator materializes all edges into an edge array E. During this materialization, the operator also tracks the highest row-id m it has seen which is depicted "max" in the figure (Step 1). Afterwards, it counts the number of children each node has (Step 2) by allocating an array S of size $m+2$ and counting the children in it. Each entry $S[i]$ corresponds to the number of children of the node with row-id i. All edges that have `NULL` as parent are counted in the last slot $S[m+1]$. Once the counts are computed, the prefix sums S_p over array S are built (Step 3), i.e., $S_p[k] = \sum_{i=0}^{k-1} S[i]$. The sums can be computed in place as we no longer need S. Note that the sums are delayed one element, so $S_p[1]$ becomes 0 because $S[1] = 2$ is first counted in $S_p[2]$.

We can now use the prefix sums to perform a "perfect" bucket sort by r_P of array E. We iterate over E and look-up the target position of a row (r_C, r_P) by

```
1: i ← 0
2: A.push(m + 1)
3: while ¬T.isEmpty do
4:     c ← A.peek()
5:     if S_p[c] = S_m[c] then
6:         T.pop()
7:     else
8:         (r_C, r_P) ← E[S_p[c]]
9:         n ← addToIndex(r_C, r_P)
10:        R.append(r_C, n)
11:        A.push(r_C)
12:        S_p[c]++
```

Figure 2.8: Depth first traversal

computing $S_p[r_P]$. Afterwards we increment the value in $S_p[r_P]$. For example, the tuple $(2, 1)$ is sorted into row 0, because $S_p[1] = 0$. Row $(5, 1)$ is sorted into row 1 because $S_p[1] = 1$ now, as it was incremented when $(2, 1)$ was processed.

The bucket sort is extremely fast as it can easily locate the target row of each row. In addition, it has the nice property that rows which have the same parent stay in the same relative order, that is, the sort is stable. For example, row $(2, 1)$ is guaranteed to stay before $(5, 1)$. This is important, because otherwise the desired sibling order would be destroyed.

As the sort destroys the initial values of S_p we must make a copy of it before executing the sort. The old S_p that was transformed by the sort is kept as well, since we need it in the next step. We call it S_m.

We can now use S_p, S_m and the sorted E, to perform a depth first traversal to build the index. Figure 2.8 depicts how this traversal is performed. We maintain a stack A, which tracks the current position in the hierarchy. We start with the virtual node $m + 1$, which is the parent of all root nodes. Then, as long as the stack is not empty, we inspect its topmost element c. If $S_p[c] = S_m[c]$, then the node c either has no children or we have already visited its children. We therefore pop c from the stack and continue. If $S_p[c] < S_m[c]$, then c has children left to be visited. Thus, we retrieve its next child which is stored in $E[S_p[c]]$.

For each discovered child we call addToIndex, which adds it to the index being built. The method returns a NODE value that the index has chosen for the added node. Note that this method call is the only part of the bulk building process that is indexing scheme dependent; the rest of the algorithm is completely generic and can be applied to any indexing scheme. For example, for the PSL scheme we

track the current *pre-rank* and *level* for each visited node during the traversal (the *pre-rank* and *level* values are inserted before visiting children, the *size* after visiting children) and encode them into the corresponding `NODE` field. With DeltaNI, we add an entry to the auxiliary structure and insert a handle to this entry into the `NODE` field for each visited node.

After we have added the current child to the index, we append the pair (r_C, n) to the tuple stream R. This stream represents the result of the β operator that will be passed to an UPDATE operator to update the node column of the temporary table T. After updating R, we push r_C onto the stack to visit its children in a depth first manner. Finally, we increment $S_p[c]$, because a child of c has been visited, so $S_p[c]$ should point the next child.

The right side of Figure 2.7 depicts the order of steps that are taken by the traversal algorithm for the example input. Note that only the r_C part of E is shown, as r_P is not important here. The algorithm first visits the nodes 4,1,2,0 (arrows 0-7 in the figure) and pushes them onto the stack A. It then encounters that $S_p[0] = S_m[0] = 0$. It therefore pops 0 from A and checks 2. However, now also $S_p[2] = S_m[2] = 3$, so 2 is popped as well. Now, $S_p[1] = 1 \neq S_m[1]$ is considered. So the algorithm visits $E[S_p[1]] = (5, 1)$ (arrow 9). Finally, after popping 5,1, and 4, the algorithm also visits $E[S_p[6]] = (3, -)$ (arrow 10)> It has now visited all nodes in pre-order (4,1,2,0,5,3).

To check for non-tree edges during the depth-first traversal, we maintain a bitset that tracks for each r_C whether it has been visited (not displayed in algorithm and figure for brevity reasons). Once an r_C is visited more than once, we must have a non-tree edge and can abort or omit the non-tree edge.

Once the β operator has finished execution, the index (if there is one, i.e., if we do not only use a labeling scheme without index) is updated, but the node column is not yet updated. To update this column we use the usual UPDATE operator that is also used for usual SQL `UPDATE` statements. It receives the stream R of row-id/node tuples and updates the `NODE` column. Thus our final plan to execute the bulk build is UPDATE(β(SORT($T \rtimes T$))).

Handling `START WHERE`. The algorithm as described so far always builds the complete hierarchy even if a `START WHERE` clause is specified. Handling the clause is straightforward: Before executing β, we mark all rows satisfying *start_condition* σ. Then, during the traversal, we add only marked nodes and their descendants. All other nodes are visited but not added to the index.

Of course, this way the *whole* hierarchy is traversed even if only a few leaf nodes qualify for σ. A recursive variant of β that traverses *only* the qualifying nodes and their descendants is to first select all qualifying rows R_σ, and then perform a recursive join starting from rows in R_σ in order to enumerate all reachable nodes. However, as our experiments indicate (Section 2.8), a recursive join is much more expensive than an ordinary join, so the recursive variant should only be chosen if we can expect the sub-hierarchy H_σ spanned by R_σ to be very small in comparison to the full hierarchy H. This is not easy to predict, since the size of H_σ is not related to the size of R_σ: Suppose, for example, R_σ contains only a single node v_0, so a naïve query optimizer might choose the recursive algorithm. If v_0, however, happens to be the *only* root of H, then $H_\sigma = H$ and the optimizer's choice is bad. Our prototype therefore refrains from using the recursive algorithm.

Late Sorting. When a `SEARCH BY` term is specified, the algorithm as described performs a complete SORT before executing the bulk-build. However, sorting can also be deferred until *after* the bucket sort. This has the advantage that not *all* rows but only rows within each bucket have to be sorted, which speeds up sorting considerably. A disadvantage is that all columns appearing in the `SEARCH BY` term (rather than just r_C and r_P) must be maintained in the edge list, so the bucket sort is slowed down due to larger rows. Since `SEARCH BY` is only used for ordered hierarchies, which are uncommon in customer scenarios, we have not implemented late sorting. This way, our implementation of β remains compact and we can reuse the existing SORT operator.

2.8 Experiments

Although this chapter focuses on concepts rather than on the performance characteristics of alternative implementations, we conducted an experiment to demonstrate that an efficient evaluation of our language constructs is possible. We derived a BOM hierarchy from the materials planning data of a large SAP customer with a few million nodes/rows. The original non-hierarchical table encodes the hierarchy structure in the Adjacency List format. It contains an `INTEGER` primary key and an `INTEGER` column referencing the superordinate part. For the equivalent hierarchical table, we employ the PSL indexing scheme. In order to assess performance on varying hierarchy sizes, we scale the data by removing or replicating nodes, covering data sizes that easily fit into cache (10^3) as well as sizes that by far exceed cache

Hierarchy Size	10^3	10^4	10^5	10^6	10^7
a.) Hash Join	79 µs	1310 µs	12200 µs	170 ms	1660 ms
b.) Bucket Sort	5 µs	133 µs	2056 µs	31 ms	399 ms
c.) Traversal	21 µs	324 µs	2867 µs	23 ms	218 ms
Total	105 µs	1.77 ms	17.1 ms	224 ms	2.27 s
Recursive Join	125 µs	1.60 ms	20.6 ms	278 ms	4.47 s

Table 2.2: Bulk-building performance

Hierarchy Size	10^3	10^4	10^5	10^6	10^7
Result Size	0	2	9	59	1293
HAJoin	60 µs	431 µs	4 ms	42 ms	439 ms
RCTE	139 µs	4604 µs	70 ms	897 ms	14011 ms
HAJoin `CHAR(16)`	51 µs	484 µs	5 ms	52 ms	521 ms
RCTE `CHAR(16)`	183 µs	6205 µs	130 ms	251 ms	52797 ms

Table 2.3: Query performance

capacity (10^7). The benchmark is executed on an Intel Core i7-4770K CPU at 3.50 GHz, running Ubuntu 14.04.

Table 2.2 shows measurements for deriving a hierarchy from the adjacency list using our bulk-build algorithm: the times of the three steps of the algorithm on the top; the total time of all steps together below that; and lowermost, for purposes of comparison, the time of a recursive join over the super-part column. Such a recursive join could be used to implement the recursive variant of β, as outlined in the previous section. Note that the measured recursive join does not perform duplicate elimination (i. e., cycle elimination), which would make it considerably slower. The table unveils the most expensive step of the bulk-build process: the initial outer hash join building the edge list (a.). In contrast, all steps of the bulk-build operator β together (b. and c.) take only around one third of the time of the hash join. We therefore conclude that the proposed bulk-building mechanism is indeed very efficient. Furthermore, we see that executing an ordinary join is considerably faster than the recursive join, especially so for large hierarchies, so the recursive variant of β is in most cases inferior to the non-recursive variant.

We measured query performance by executing the query from Figure 2.4 on the hierarchy and comparing it with the equivalent RCTE from Figure 2.3 as baseline.

We use a 16 byte payload column, so the size of a result row containing 3 `INTEGER` keys and 3 payload fields is 60 byte. Table 2.3 includes the runtimes of the two algorithms and the sizes of the result sets. The last two rows show the results for analogous measurements with the `INTEGER` key replaced by a `CHAR(16)` key. As we see in the table, the hierarchy-aware join (HAJoin) easily outperforms RCTEs—for large hierarchies by a factor of over 30. There is a simple explanation for that huge speed-up, and it shows the general problem with RCTEs: Even though the result set is not too large, the recursive join must iterate over large subtrees of the hierarchy, yielding large intermediate results, only to find that there are almost no matching parts in these subtrees. By contrast, a hierarchy-aware join does not need to enumerate whole subtrees to find matching nodes; thus, the predicate can be pushed down to the table scans and only parts that meet the filter condition participate in the join in the first place. When we use `CHAR(16)` keys, the figures (last two lines) reveal one more advantage over RCTEs: Hierarchy-aware joins work on the join-optimized `NODE` column, while RCTEs must necessarily work on the key column. Consequently, an unwieldy key type whose values are expensive to compare hurts the performance of RCTEs, while hierarchy-aware joins do not suffer. Thus, the hierarchy-aware join outperforms the RCTE by two orders of magnitude in this scenario.

Note that a hierarchy-aware join is so fast that its execution is still much faster than the RCTE even if we always perform a complete bulk-build prior to executing the query. For example, for 10^7 nodes, bulk-building plus querying takes 2.71 seconds, while the RCTE takes 14 seconds, so the speed-up is still more than a factor of 5. We therefore conclude that migrating from an RCTE-based approach to a hierarchy dimension can yield a considerable query speed-up (Req. #8), even more so if the hierarchy is not always bulk-built before each query. Since HANA employs view caching, a bulk-build used in a view will not be re-executed unless the input tables change. Thus, even applications that simply issue a bulk-build for each query will run exceptionally fast most of the time, since the bulk-build will often be elided in favor of a cached result.

2.9 Conclusion

Our work has been motivated by customer demand and findings from our investigation of typical requirements of SAP applications featuring hierarchical data. Our

analysis leads us to conclude that the conventional approaches to handling such data—particularly recursive CTEs—are not fully satisfactory to meet the requirements. As a solution, we propose to enhance the relational model to incorporate hierarchies by means of a new data type `NODE`. This data type opaquely represents the hierarchy structure without mandating a specific encoding, in order to leave the system full flexibility in choosing the most appropriate indexing scheme. We introduce extensions to the SQL language that allow the user to specify queries over hierarchical data in a concise and expressive manner. The syntax extensions are minimal in that they rely mostly on built-in hierarchy functions and predicates operating on `NODE` values. Because of this, SQL programmers can adapt easily to the new syntax, and its integration into an existing RDBMS is straightforward. We propose efficient bulk-update operations for legacy applications, as well as fine grained update operations for greenfield applications. In conclusion, we propose a SQL based, user-friendly front end with means to build, alter, and query hierarchical data in a relational setting. However, a user-friendly query and data manipulation language is useless if it cannot be evaluated efficiently. Therefore, the following chapter will cover a back end that can evaluate queries efficiently while allowing a high rate of complex updates.

CHAPTER 3

Order Index: Indexing Highly Dynamic Hierarchical Data

Parts of this chapter have previously been published in [37]. The paper was invited for an extended version, which has been submitted for the Special Issue of VLDB Journal. The extended version is also part of this chapter.

Hierarchical data has always been ubiquitous in business and engineering applications, especially with the advent of the inherently hierarchical XML data format. Relational database systems (RDBMS) continue to be the predominant platform for such applications. These facts have repeatedly led to the challenge of representing hierarchical data in relational tables, or more specifically, encoding the structure of a hierarchy in a table such that a table row represents a hierarchy node. We solved parts of this challenge in the previous chapter by proposing a user-friendly, SQL-based front end, which encodes hierarchy information in columns of the abstract data type `NODE`. The performance of the system depends on the representation of a hierarchy node through a value of this data type. Therefore, we revisit the challenge of finding a representation that provides competitive query capabilities without sacrificing update performance. This classic trade-off strikes particularly hard with hierarchical data and numerous papers have been written on working around it. Unlike many of those works, we place our focus on update performance. In SAP's application scenarios we encounter fine-grained, complex updates—in particular relocations of large subtrees—and at the same time need to provide a certain set of primitive query operations where we cannot tolerate

significant performance losses. A comprehensive literature survey revealed that a robust, efficient solution is still missing to date.

Hereinafter we use *indexing scheme* as a collective term for any technique for representing a hierarchy with the aim of providing an acceptable tradeoff between query and update performance. Two major classes of schemes exist: *Labeling schemes* [7, 16, 21, 43, 45, 46, 48, 55, 56, 57, 58, 67, 74, 95, 101, 103, 105], on the one hand, attach a *label* to each node and answer queries by considering only these labels. The labels are maintained in one or more table columns, and those columns are usually augmented by one or more general-purpose database indexes, such as B-trees. *Index-based schemes* [36, 91], on the other hand, enhance the RDBMS by special-purpose index structures.

Especially labeling schemes are backed by a massive body of research on XML databases, on techniques for storing XML fragments in an RDBMS backend, and on evaluating query languages such as XPath and XQuery. However, in this thesis we make a case for index-based schemes. Our first contribution is an analysis of query versus update considerations (Section 3.1) and a taxonomy of existing indexing schemes in that light (Section 3.2). This leads us to conclude that previously proposed dynamic labeling schemes are unable to fulfill desired properties: they either ignore important query primitives, or they inherently suffer from certain problems inhibiting their update flexibility or robustness. Sophisticated index-based schemes can help us overcome the inherent problems of plain labeling schemes and support highly dynamic use cases efficiently. To show that the interface we propose for index-based schemes is powerful enough, we outline how it can be used to answer end-user queries (Section 3.3). Our main contribution is a family of index-based schemes called *Order Indexes* (Section 3.4). We propose three specific implementations, the AO-Tree, the BO-Tree, and the O-List, and discuss performance optimization techniques and possible extensions (Section 3.5). We conduct a number of experiments (Section 3.6) to assess the merits and drawbacks of the implementation variants, and we show that they support complex updates efficiently, avoid degeneration in case of unfavorable update patterns, and provide the query capabilities of labeling schemes with highly competitive performance. Order Indexes are a promising back end for the front end we proposed in Chapter 2. However, they are also applicable to non-relational systems such as XML databases.

3.1 Dynamic Hierarchies in Relational Systems

We define a hierarchy as a forest of rooted trees. In a relational context, each node in a hierarchy is represented by a row of an associated table, thus the table columns carry possible node attributes. The considerations for representing hierarchical data in relational systems differ somewhat from special-purpose systems, especially XML DBMS: In business applications such as SAP ERP, which we analyzed in the previous chapter, a hierarchy is rarely the primary dimension of a table. Therefore, clustering the associated table by hierarchy structure is usually infeasible or not preferable, and thus hierarchy indexes are generally *secondary* indexes in this setting. XML indexing techniques, on the other hand, commonly rely on clustering data by structure, for example by arranging it in pre-order and adding primary indexes.

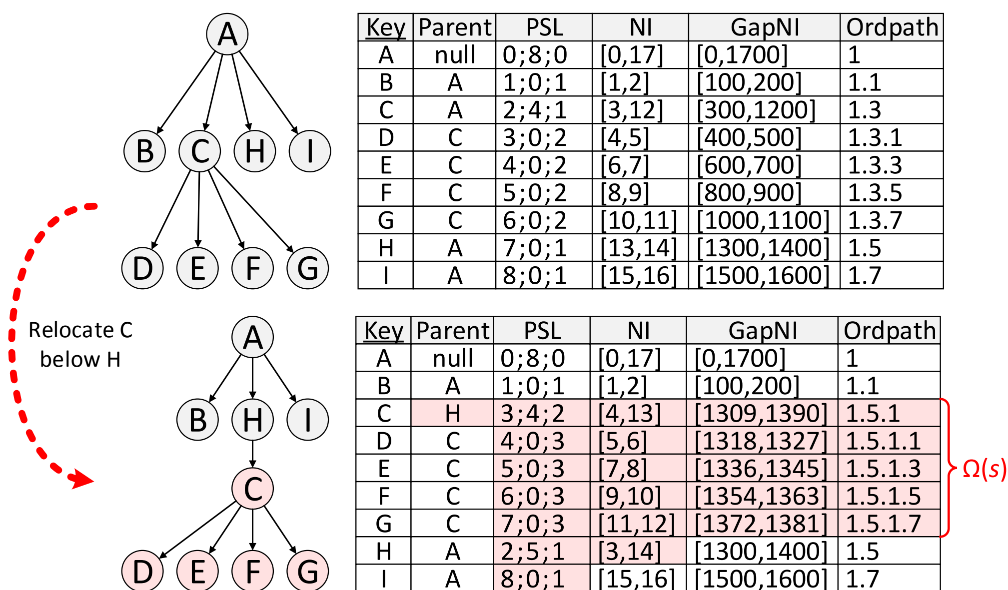
We base our considerations on the data model from Chapter 2. It represents a hierarchy by a `NODE` column plus a secondary index structure. The column contains the *labels* of the indexing scheme, that is, scheme-specific node identifiers. The name of the node column serves as a user handle to issue hierarchical queries on.

Depending on the application, the order among siblings may be meaningful (e. g., document order in XML), so a hierarchy can be *ordered*. However, even in the unordered case (e. g., human resources hierarchies), indexing schemes impose an internal storage order. We therefore focus on the ordered case for related work and our proposed indexes. Any index that supports ordered hierarchies can be used for both types of applications.

3.1.1 Challenges

Existing indexing schemes lack important capabilities for highly dynamic use cases. The three problems we identify in this section characterize these missing capabilities. The hierarchy from Figure 3.1 helps us exemplify the problems; it also illustrates several indexing schemes, which we explain later.

P1 *Lack of Query Capabilities.* Certain indexing schemes do support updates decently, but fail to offer query capabilities to evaluate even fundamental queries, which renders them infeasible for our use cases. An example is the adjacency list model, a common way of naïvely encoding a hierarchy in SQL by storing the primary key of the parent node (Parent in Figure 3.1): it cannot even handle the ancestor-



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

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

Figure 3.1: Various labeling schemes (top); labels that need to be changed when subtree C is relocated (bottom)

descendant relationship efficiently. In Section 3.1.2 we identify fundamental query primitives to be supported sine qua non.

P2 *Lack of Complex Update Capabilities.* Various ERP use cases demand for an indexing scheme that supports a rich set of update operations efficiently. However, most existing schemes are confined to *leaf* updates, that is, insertion or deletion of single leaf nodes, and fail to recognize more complex operations. Consider *subtree* relocation, where a subtree of a certain size s rooted in a specific node is moved in bulk to another location within the hierarchy. Ironically, the trivial adjacency list model naturally supports this. However, virtually all labeling schemes by design preclude an efficient implementation, because they inherently require relabeling *all* nodes in the relocated subtree at a cost of $\Omega(s)$. The bottom of Figure 3.1 shows the hierarchy from the top with the subtree rooted in C moved below H, and highlights the $\Omega(s)$ fields that need to be updated. In Section 3.1.3 we explore further conceivable complex update operations.

P3 *Vulnerability to Skewed Updates.* Certain dynamic labeling schemes crumble when confronted with skewed updates, such as when inserts are issued repeatedly at the same position. In some scenarios these updates are more frequent than is commonly acknowledged. For example, when inserting a new plant into an

BINARY PREDICATES	
is_descendant(a, b)	whether a is a descendant of b
is_child(a, b)	whether a is a child of b
is_before_pre(a, b)	whether a precedes b in a pre-order traversal
is_before_post(a, b)	whether a precedes b in a post-order traversal
NODE PROPERTIES	
level(a)	number of edges on the path from a root to a
is_root(a)	whether a is a root node
is_leaf(a)	whether a is a leaf node, i. e., has no children
INDEX ACCESS	
find(a)	a cursor c to node a
rowid(c)	the row id corresponding to the node at c
TRAVERSAL	
next_pre(c)	a cursor to the next node in pre-order
next_post(c)	a cursor to the next node in post-order
next_sibling(c)	a cursor to the next sibling
next_following(c)	a cursor to the next node in pre-order that is not a descendant of c

Table 3.1: Essential query primitives on hierarchies

enterprise asset hierarchy, many nodes will be added at one position. Fixed-length labeling schemes commonly indulge in excessive relabeling in this case, while variable-length schemes decay in their query performance and memory effectiveness due to overly growing labels.

3.1.2 Query Capabilities

Query primitives are the building blocks for answering high-level queries on hierarchies. Table 3.1 shows an essential set of primitives that are needed to answer fundamental queries. In this and the upcoming figures, a and b represent node labels (stored in a table column) and c represents a cursor pointing to an entry of the secondary index structure over these labels. An index that fails to support the depicted primitives cannot be considered a general-purpose hierarchy index. How to actually apply these primitives in an RDBMS will become clear in Section 3.3, which demonstrates that they are in fact sufficient to evaluate common end-user queries that appear in business scenarios.

ORDINAL PROPERTIES	
pre_rank(a)	a's rank in a pre-order traversal
post_rank(a)	a's rank in a post-order traversal
subtree_size(a)	number of nodes in the subtree rooted in a
range_size($[a, b]$)	number of nodes in all subtrees rooted in $[a, b]$
ORDINAL ACCESS	
select_pre(n)	a cursor to the n-th node in pre-order
select_post(n)	a cursor to the n-th node in post-order

Table 3.2: Ordinal query primitives

We distinguish between four kinds of primitives. *Binary predicates* test whether two nodes are related with respect to a certain axis. is_before_pre and is_before_post are useful for ordering nodes in a depth-first, either parent-before-child (*pre-order*) or child-before-parent (*post-order*) manner. Strictly speaking, the other two predicates are redundant: is_descendant can be expressed in terms of is_before_pre and is_before_post, and is_child in terms of is_descendant and level. However, we found having dedicated, potentially optimized implementations for these two primitives to be clearly beneficial, as most queries navigate along these axes. *Node properties* are used to filter nodes, for example, when the user wishes to restrict the result to leaf nodes or to nodes at certain levels. An example is the so-called explosion query often found in ERP applications, which consists of finding all descendants of a node up to a certain level. *Index access* primitives are used to navigate between the table (i.e., the labels) and the index. *Traversal* operations scan the index in various directions. They are useful to implement set-oriented operators that enumerate subsets of the hierarchy nodes, such as a scan.

We found this essential set of primitives to be expressive enough to cover most use cases, while at the same time allowing for efficient implementations in most existing indexing schemes. That said, one may well decide to add further primitives to support advanced applications, or just to improve performance through a redundant primitive for a special case. For instance, a group of advanced primitives are the *ordinal* query primitives shown in Table 3.2. They are based on the ordinal number of a node with respect to a certain traversal. In the figure, the syntax $[a, b]$ denotes a *sibling range* of nodes: b must be a right sibling of a (or a itself), and $[a, b]$ refers to all siblings between and including a and b. pre_rank and post_rank return the position of a node with respect to pre- or post-order traversal, respectively; given such ranks, pre_select and post_select return the corresponding cursor. Rank and

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

Table 3.3: Update operations on hierarchies

select are inverse: pre_rank(select_pre(n)) $= n$ and select_pre(pre_rank(a)) $= a$. With rank/select, we usually get support for subtree_size and range_size as a byproduct, since subtree_size(a) $=$ post_rank(a) $-$ pre_rank(a) $+$ level(a). Ranks are, for instance, used to create compact representations from subtrees, so-called tree signatures, for pattern matching purposes [104]. Use cases for select primitives are, for example, top-N queries with an offset for displaying parts of the hierarchy in a user interface. The subtree sizes provide useful statistics; they are also used in user interfaces to indicate the sizes of currently folded subtrees. Moreover, they can also be leveraged internally for cardinality estimations. In order to render Order Indexes applicable in more applications, we consider ordinal primitives for Order Indexes, too (Section 3.5.2). That said, we do not count them as essential. Their implementation complexity and memory and maintenance overhead at runtime will not pay off in every application.

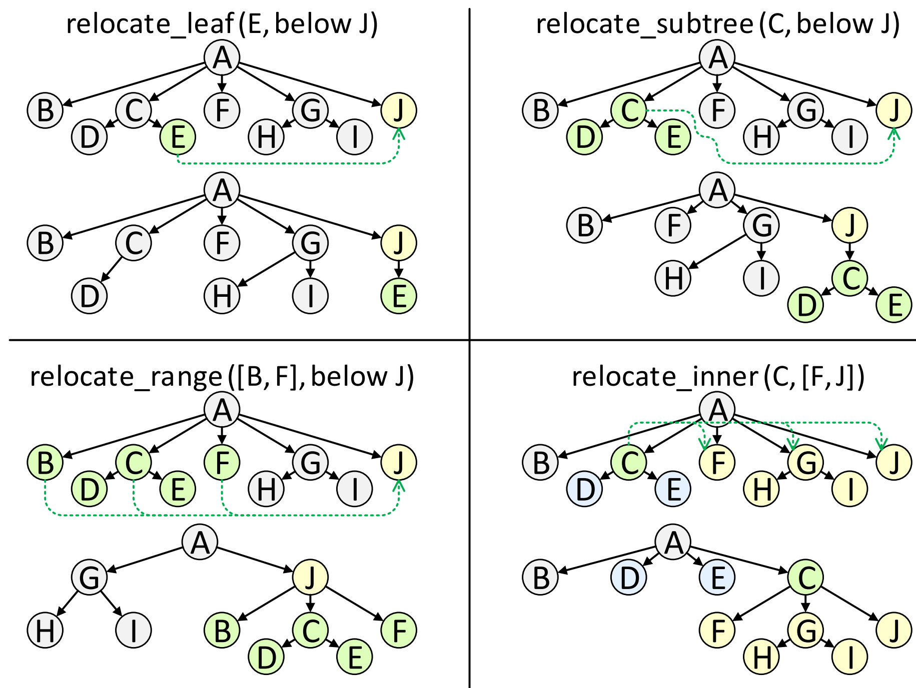


Figure 3.2: Various classes of relocation updates on an example hierarchy: before and after

is_before_pre, is_before_post, as well as any traversal and ordinal primitives are undefined for *unordered* hierarchies. These primitives will be less relevant for applications featuring such hierarchies. However, in many situations it is still desirable to have support for these operations in a deterministic way, even though the underlying order is implementation-defined.

3.1.3 Update Capabilities

In Table 3.3 we present a taxonomy of update operations that a dynamic indexing scheme shall support. p indicates a target position in the hierarchy. Depending on whether the sibling order is meaningful, p can have different values, such as "as first child of node x", "as direct left sibling of x", "below x", or "as a sibling of x". How exactly p is represented is not important here.

The first class of updates is bulk-building an index from another hierarchy representation. This is an important task when creating an indexing scheme on existing relational data. Fortunately, all indexing schemes—even static ones—can be

bulk-built efficiently using, for instance, the index-independent bulk-build operator from Section 2.7.3. The other classes are *leaf* node updates, *subtree* updates, sibling *range* updates, and *inner* node updates, each named after the entities involved in the update. Within each class, three kinds of updates are conceivable: *delete* updates, which delete existing nodes, *insert* updates, which insert new nodes, and *relocate* updates, which alter positions of existing nodes. Figure 3.2 illustrates the various classes with regard to the relocate kind on an example hierarchy. The other kinds, insert and delete, are similar; the only difference is that the updated entities (green in the figure) enter or leave the hierarchy, respectively, instead of being relocated. In a sense, the relocate kind subsumes the others: inserts and deletes are relocations into and out of the hierarchy, respectively. Thus, any index that handles relocation efficiently supports efficient insertion and deletion as well.

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

Our focus is particularly on *subtree* updates. As a leaf node is also a trivial subtree, they subsume the corresponding leaf updates. But since indexing schemes usually afford optimized operations for leaves, distinguishing between leaf and subtree operations is useful in practice. Most indexing schemes implement subtree operations naïvely through node-by-node processing, requiring at least s leaf updates for a subtree of size s. For small subtrees, $\Omega(s)$ update cost might be tolerable. However, real-world hierarchies—such as the hierarchies found in SAP's ERP applications—have a large average fan-out. Thus, even if a node that has only leaves as children is relocated, s will often be in the magnitude of thousands. Of course, updating larger subtrees will be detrimental to overall system performance only if a lot of such operations appear in the workload. But in many ERP use cases, a high percentage of updates (e. g., 31% in the enterprise asset hierarchy examined in [36]) are indeed subtree relocations. Furthermore, note that although subtree relocation may appear as an unnatural bulk operation in comparison to single leaf insertion or deletion, the operation is quite fundamental to SQL users: In the adjacency list model—the predominant format for representing hierarchies within an RDBMS—*any* change to a parent field corresponds to a subtree relocation. For example, we achieve the relocation in Figure 3.1 by simply setting the `Parent` of C to H in the table. In other words, every issued `UPDATE` statement touching the parent column incurs subtree relocations.

The *inner* node updates are useful for inserting a new level into the hierarchy, and for wrapping a subtree into a new root. As an example application, certain tree differentiation algorithms such as MH-Diff [23] emit edit scripts featuring these operations. An index that is being used for replaying such edit scripts has to support them.

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

Unlike the query primitives, the update primitives may be significantly more complex for ordered than for unordered hierarchies, since ordered hierarchies need the position among siblings maintained. Even if an index does support ordered hierarchies, using it for unordered hierarchies may allow optimizations because it can pick the sibling position which yields the cheapest update.

3.2 Related Indexing Schemes

In the following we explore existing indexing schemes and assess to which extent they suffer from one of the identified problems ***P1*** to ***P3***. It turns out that most indexing schemes suggested in the literature are variations of basic schemes with similar capabilities and asymptotic properties, so we can group them into a small taxonomy.

Table 3.4 shows the asymptotic amortized query complexities of all groups of schemes in our taxonomy, for all essential query primitives we presented in Table 3.1. All primitives work on labels, except the ones tagged with [′], which work on cursors to index entries. Depending on the access method, such cursors are already available during query execution or must first be obtained from the label with the find function. Note that many of the query operations in the table are not mentioned in the cited works, but inferring them is straightforward.

Table 3.5 shows the asymptotic amortized update complexities of all groups of schemes in our taxonomy. The columns represent the different classes of updates. Column skew$[u]$ represents skewed leaf node insertions; it depicts the complexity of

		ACCESS	TRAVERSAL			
		find	next_pre	next_post	next_sibling	next_following
naïve	Adjacency	1	–	–	–	–
	Linked	1	$1'$	$1'$	$1'$	$1'$
contain.	(Dyn-)NI	$\log n^{\ddagger}$	$1'$	$1'$	$1'$	$1'$
	(Dyn-)NI-Parent	$\log n^{\ddagger}$	$1'$	$1'$	$1'$	$1'$
	(Dyn-)NI-Level	$\log n^{\ddagger}$	$1'$	$1'$	$1'$	$1'$
pa.	(Dyn-)Dewey	$l \log n^{\ddagger}$	$1'$	$l \log n^{\ddagger}$	$l \log n^{\ddagger}$	$l \log n^{\ddagger}$
index	B-BOX[B]	B	$1'$	$1'$	$1'$	$1'$
	AO-Tree	1	$1'$	$1'$	$1'$	$1'$
	BO-Tree[B]	1	$1'$	$1'$	$1'$	$1'$
	O-List[B]	1	$1'$	$1'$	$1'$	$1'$
	DeltaNI	$\log u$	$\log u'$	$\log u'$	$\log u'$	$\log u'$

		BINARY PREDICATES			NODE PROPERTIES		
		is_desc.	is_before pre/post	is_child	level	is_root	is_leaf
naïve	Adjacency	l	–	1	l	1	1
	Linked	l'	l'	$1'$	l'^{*}	$1'$	$1'$
contain.	(Dyn-)NI	$1^{\ddagger}$	$1^{\ddagger}$	b'	–	b'	$1'/1^{\dagger}$
	(Dyn-)NI-Parent	$1^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	$l^{\ddagger *}$	1	$1'/1^{\dagger}$
	(Dyn-)NI-Level	$1^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	1	1	$1'/1^{\dagger}$
pa.	(Dyn-)Dewey	$l^{\ddagger}$	$l^{\ddagger}$	$l^{\ddagger}$	$l^{\ddagger}$	1	$l'^{\ddagger}$
index	B-BOX[B]	$\log_B n'$	$\log_B n'$	b'	–	b'	$1'$
	AO-Tree	$\log n'$	$\log n'$	$\log n'$	$\log n'^{*}$	$\log n'^{*}$	$1'$
	BO-Tree[B]	$\log_B n'$	$\log_B n'$	$\log_B n'$	$\log_B n'^{*}$	$\log_B n'^{*}$	$1'$
	O-List[B]	$1'$	$1'$	$1'$	$1'$	$1'$	$1'$
	DeltaNI	$1'$	$1'$	$1'$	$1'$	$1'$	$1'$

n: hierarchy size, l: level of node, b: number of siblings, u: number of updates, B: block size
–: not supported, $'$operates on cursors, †only in the static variant, $^{*}\mathcal{O}(1)$ during index scan
‡in the Dyn variant, performance may decrease over time due to growing labels
Qualitative rating: ☐ efficient ☐ mostly efficient ☐ inefficient or unsupported

Table 3.4: Asymptotic query complexities for various indexing schemes (amortized, average case)

		Update Operations				
		LEAF	SUBTREE	INNER	RANGE	SKEW[u]
naïve	Adjacency	1	1	c	c	1
	Linked	1	1	c	c	1
containment	NI	n	n	n	n	n
	Dyn-NI	1	s	1	s	u
	Dyn-NI-Parent	1	s	c	s	u
	Dyn-NI-Level	1	s	s	s	u
path	Dewey	lf	$l(f+s)$	$l(f+s)$	$l(f+s)$	$l(f+s)$
	Dyn-Dewey	l	ls	ls	ls	$l+u$
index	AO-Tree	1	$\log n$	$\log n$	$\log n$	1
	BO-Tree[B]	1	$B \log_B n$	$B \log_B n$	$B \log_B n$	1
	O-List[B]	1	$s/B+B$	$s/B+B$	$s/B+B$	u/B^2
	DeltaNI	$\log u$	$\log u$	$\log u$	$\log u$	$\log u$

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

Table 3.5: Asymptotic update complexities for various indexing schemes (amortized, average case)

a single skewed insertion after u other skewed insertions have taken place and thus expresses how skew-resilient an indexing scheme is. All updates operate on cursors.

The figures include two naïve hierarchy representations. They are pragmatic, easy-to-implement solutions, which, however, do not provide the efficient query capabilities of indexing schemes (***P1***). First, the Adjacency list introduced earlier. Second, Linked, a simple in-memory tree representation whose structure matches the hierarchy structure and which uses per-node pointers to the parent, the first and last child, and the previous and next sibling. The problem with these schemes is that the important query primitives level and is_descendant run in $\mathcal{O}(l)$ and also are likely to cause $\mathcal{O}(l)$ cache misses, because they have to walk up the tree. Adjacency uses a hash index for the key and parent columns, and Linked stores direct pointers into the hierarchy representation, so both execute find in $\mathcal{O}(1)$. Note that Adjacency does not maintain a sibling order. It is the only scheme that supports *only* unordered hierarchies, and therefore has no support for order-based traversal primitives.

Besides the naïve schemes, there are three major categories of indexing schemes: *containment-based* labeling schemes, *path-based* labeling schemes, and *index-based* schemes. Labeling schemes index their label columns with a B-tree that is also used for traversal operations, so the find operation runs in logarithmic time for them. Note that this B-tree also adds an $\mathcal{O}(B)$ term—where B is the block size of the B-tree—to all update operations for these schemes (and for BO-Tree and O-List), because the block in which the corresponding index entry lies has to be updated. We omitted this term in Table 3.5 for simplicity reasons and because $\mathcal{O}(B) = \mathcal{O}(1)$, since B does not depend on the hierarchy size.

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

Column NI in Figure 3.1 shows a nested intervals labeling that is commonly used in XML and other database applications, e. g. [105, 45, 48]. We can see that node E is a descendant of node A, because E's interval $[6, 7]$ is a proper subinterval of A's interval $[0, 17]$. A variation is the pre/post scheme [43], where each node is labeled with its pre- and post-order ranks. Considering updates, the mentioned schemes are *static* (***P2***). Their fundamental problem is that each insertion or deletion requires relabeling $\mathcal{O}(n)$ labels on average, as all interval bounds behind a newly inserted bound have to be shifted to make space. This group of static nested interval schemes is called NI in the tables. Considering queries, plain NI and pre/post have similar, limited capabilities: For example, we cannot test the important is_child predicate, because neither scheme allows us to compute the distance between a node and an ancestor. This severe limitation makes a nested intervals scheme without further fields useless (***P1***). It can be mitigated by either storing the level of a node or its parent in addition to the interval (NI-Level and NI-Parent in the tables, respectively).

Various mitigations for the nested intervals update problem have been proposed (Dyn-NI in the tables). Li et al. [58] suggest pre-allocating *gaps* between the interval bounds. Column GapNI in Figure 3.1 illustrates this. As long as a gap exists, new bounds can be placed in it and no other bounds need to be shifted; once a gap between two nodes is filled up, *all* bounds are relabeled with equally spaced values. The caveats are that relabelings are expensive, and skewed insertions may fill up

certain gaps overly quickly and lead to unexpectedly frequent relabelings (***P3***). In addition, all s nodes in a range or subtree being updated still need to be relabeled (***P2***).

Amagasa et al. [7] propose the QRS encoding based on pairs of floating-point numbers. Schemes along these lines are essentially gap-based as long as they rely on fixed-width machine representations of floats. Boncz et al. tackle the update problem using their pre/size/level [16] encoding (PSL, cf. Figure 3.1) by storing the pre values implicitly as a page offset, which yields update characteristics comparable to gap-based schemes. W-BOX [91] uses gaps but tries to relabel only locally using a weight-balanced B-tree; its skewed update performance is therefore superior to basic gap-based schemes. The Nested Tree [103] uses a nested series of nested interval schemes to relabel only parts of the hierarchy during an update and is therefore comparable to gap-based schemes.

Another idea to tackle the update problem for NI is to use variable-length data types to represent interval bounds: For example, the QED [55], CDBS [56], and CDQS [57] encodings by Li et al. are always able to derive a new label between two existing ones, and thus avoid relabeling completely. EXCEL [67] uses an encoding comparable to CDBS. It tracks the lower value of the parent for enhanced query capabilities. While these encodings never have to relabel nodes, they bear other problems: The variable-length labels cannot be stored easily in a fixed-size table column and comparing them is more expensive than comparing fixed-size integers. In addition, labels can degenerate and become overly big due to skewed insertion (***P3***). Cohen et al. [31] proved that for any labeling scheme that is not allowed to relabel existing labels upon insertion, an insertion sequence of length n exists that yields labels of size $\Omega(n)$. Thus, the cost of relabeling is traded in for a larger (potentially unbounded) label size. Query primitives that suffer from these degenerating labels are marked with [‡] in Table 3.4.

All gap-based and variable-length NI schemes can handle *inner node* updates decently by wrapping a node range into new bounds. For example, GapNI in Figure 3.1 is able to insert a parent node K above D, E, and F by assigning it the bounds $[350, 950]$. However, as soon as the node level [16] or its parent [67] (Dyn-NI-Level and Dyn-NI-Parent in the tables) are to be tracked explicitly—which is necessary for many queries (***P1***)—inner node updates turn expensive, as the parent of all c children of K (D, E, and F) or the levels of all s descendants change. Updates to subtrees or ranges of size s always alter all s labels. So, since all

containment-based schemes suffer from the problems ***P2*** and ***P3***, their use in highly dynamic settings is limited.

Path-based Labeling Schemes encode the path from the root down to a node into its label. Dewey [95] is a prominent example, and the basis of several more sophisticated schemes. It builds upon the sibling rank, that is, the 1-based position of a node among its siblings. Each node is labeled with the label of its parent node plus a separating dot plus its sibling rank. In the example hierarchy of Figure 3.1, node G receives the Dewey label 1.2.4, as G is the fourth child of C, which is the second child of A, which is the first root. Dewey is not dynamic (***P2***): We can easily insert a new node as rightmost sibling, but in order to insert a node a between two siblings, we need to relabel all siblings to the right of a and all their descendants (as indicated by factor f in Table 3.5). For unordered hierarchies, inserting rightmost siblings is sufficient, but for ordered hierarchies insertion between siblings is a desirable feature. Therefore several proposals try to enhance Dewey correspondingly: One prominent representative is Ordpath [74], which is used in Microsoft SQL Server for the *hierarchyid* data type. It is similar to Dewey, but uses only odd numbers to encode sibling ranks, while reserving even numbers for "careting in" new nodes between siblings. This way, Ordpath supports insertions at arbitrary positions without having to relabel existing nodes. In Figure 3.1, for example, inserting a sibling between C and H results in the label 1.4.1. Note that the dot notation for labels is only a human-readable surrogate; Ordpath stores it in a more compact binary format. A lot of further dynamic path-based schemes have been proposed: DeweyID [46] improves upon Ordpath by providing gaps that are possibly larger than the ones of Ordpath, thus resulting in less carets and usually shorter labels. CDDE [101] also aims to provide a Dewey encoding with shorter label sizes than Ordpath. The encoding schemes [55, 56, 57] can also be used for building dynamic path-based schemes. In the tables, Dewey refers to static path-based schemes such as Dewey itself and Dyn-Dewey refers to dynamic ones (e. g., Ordpath and CDDE). Dynamic path-based schemes are variable-length labeling schemes, and the proof of [31] holds as well, so they pay the price of potentially unbounded label sizes. In addition, all path-based schemes pay a factor l on all update and most query operations, since the size of each node's label is proportional to its level l.

Considering updates, the dynamic variants are able to insert leaf nodes, but cannot handle inner node updates efficiently, as the paths and thus the labels of

all descendants of an updated inner node would change. An exception to this is OrdpathX [21], which can handle inner node insertion without having to relabel other nodes. All path-based schemes inherently cannot handle subtree and range relocations efficiently, as the paths of all descendants have to be updated (***P2***). When used for ordered hierarchies, they are also vulnerable to skewed insertions (***P3***); however, update sequences that trigger worst-case behavior are much less common than for a containment-based scheme.

Index-based Schemes use special-purpose secondary index structures to evaluate queries rather than considering a label (all operations marked with [′] in Table 3.4). Their advantage is that they generally offer improved update support. B-BOX [91] uses a keyless B^+-tree to represent a containment-based scheme dynamically. It has the same update complexity as the BO-Tree in Table 3.5. However, it represents only lower and upper bounds but does not support level or parent information and thus has limited query capabilities (***P1***). Its find implementation runs in $\mathcal{O}(B)$, because it always scans a block of size B when searching for index entries. Our DeltaNI (Chapter 4) uses an index to represent a containment-based scheme with level support. As a *versioned* index, it captures a whole history of updates (factor u in the figures) and is able to handle time-travel queries. It can be used for unversioned hierarchies by simply keeping all data in a single version delta. While DeltaNI bears none of the three identified problems, its overall query performance is generally inferior to unversioned schemes, as our evaluation shows. Both DeltaNI and B-BOX can handle subtree and range relocations in logarithmic worst-case complexity, so they do not show any of the update problems.

3.3 Hierarchical Query Processing

We already mentioned example scenarios for the various *update* primitives we introduced in Section 3.1.3. This section sheds light on how the *query* primitives can be used for efficient processing of common end-user queries.

To get a feeling for what is needed for efficient query processing on hierarchical data, we first have to look at what kinds of queries are performed in this context. The most important functionality when querying hierarchical data is *navigating* through the hierarchy on certain axes. In a relational context, such navigation translates naturally into self-joins over the hierarchy nodes. Therefore, efficient join algorithms are crucial for overall system performance. Other conceivable operations

```
SELECT u.pos, v.pos, LEVEL(v.pos)
  FROM T as u
  JOIN T as v ON IS_DESCENDANT(v.pos, u.pos)
 ORDER BY PRE_RANK(u.pos), PRE_RANK(v.pos)
```

Figure 3.3: An example query using the SQL extensions from [18]

on hierarchies, such as sorting nodes in a specific order or aggregating information up or down the hierarchy, map to relational operators quite naturally as well. Hereinafter, we will exemplify certain efficient operators—especially joins—using our query primitives. By providing this set of abstract, index-indifferent query primitives, query processing can use the same algorithms irrespective of the actual values in the node column (in fact, our prototype is able to process queries with all nine indexing schemes we implemented for Section 3.6).

Our SQL extension from Chapter 2 actually does not extend the SQL grammar at all; it only adds the `NODE` data type and functions for working with it. Thus, it gets fully translated to relational algebra. In the following, we assume T is a hierarchical table exposing a node column named pos. H denotes the associated hierarchy index.

Figure 3.3 shows a simple example query associating each node with the level and identity of its descendants, ordered in pre-order. All hierarchy capabilities are encapsulated in a set of SQL functions: hierarchy properties and hierarchy predicates. This query uses two hierarchy properties (`LEVEL`, `PRE_RANK`) and a predicate (`IS_DESCENDANT`). The SQL functions from Chapter 2 can be translated straight into the query primitives of Section 3.1.2; for example,

`LEVEL(`a`)`	$H.\mathsf{level}(a)$
`IS_ANCESTOR(`a`,`b`)`	$H.\mathsf{is_descendant}(b, a)$
`IS_PRECEDING(`a`,`b`)`	$H.\mathsf{is_before_pre}(a, b) \wedge H.\mathsf{is_before_post}(a, b)$

The whole language could therefore naïvely be implemented by adding corresponding map operators to the plan. However, many hierarchy-centric optimizations are possible, as we will show hereinafter.

The two properties `PRE_RANK` and `POST_RANK` virtually always appear in conjunction with `ORDER BY`, as in the example query. An implementation may therefore reasonably decide to prohibit projecting `PRE_RANK` and `POST_RANK` and allow it only in order-by clauses—as our prototype currently does. Otherwise, we can employ the ordinal

query primitives from Table 3.2. As long as ranks only appear in an order-by clause, their evaluation boils down to ordinary sorting using is_before_pre and is_before_post as comparison predicates, respectively. Alternatively, the index can often even be used to enumerate the tuples in the desired order in the first place and get rid of the sort altogether. We call the corresponding physical operator *hierarchy index scan (HIS)*. Its basic task is to enumerate the tuples corresponding to a given subtree a in pre-order. This is where traversal primitives come into play:

```
Hierarchy Index Scan [pre-order] (T, H, a)
c ← H.find(a)
c′ ← H.next_following(c)
do
    yield T[H.rowid(c)]                ▷ table access
    c ← H.next_pre(c)
while c ≠ c′
```

First, we look up a in the index structure using find, and determine a's next following node as a delimiter for the scan ([74] suggested a similar technique based on a grdesc primitive comparable to next_following). The descendants of a are in the range between c and c'. We iterate over them using next_pre. A modification to post-order is straightforward, as are variants for other hierarchy axes. This makes the hierarchy index scan a versatile operator.

Similar to pre/post ranks, hierarchy predicates are rarely explicitly evaluated, as they primarily appear in join conditions. Nested loop joins work out of the box, but for decent performance the engine should provide specialized join operators. One must-have is the *hierarchy index nested loop join (HINLJ)*, which directly enumerates the matching right input tuples for each left input tuple it consumes by essentially running an index scan. This operator is quite appropriate for our example query, which features a hierarchy join over the descendant axis and a computation of the node level. The corresponding physical plan would be

$$\chi_{H.\text{level}}(\text{HIS}(T) \bowtie^{\text{HINLJ}}_{H.\text{is_descendant}} T)$$

A modern query compiler [71] could produce the following tight code fragment from this plan:

```
c_u ← first root node in H
while c_u ≠ null do                                              ▷ HIS
    c_v ← H.next_pre(c_u)
    c'_v ← H.next_following(c_v)
    while c_v ≠ c'_v do                                          ▷ HINLJ
        yield [c_u, c_v, H.level(c_v)]
        c_v ← H.next_pre(c_v)
    c_u ← H.next_pre(c_u)
```

The outer scan enumerates all c_u in pre-order and the nested scan enumerates their descendants c_v. No sorting is necessary, as the result already has the desired order.

Beyond the basic index join, more sophisticated join algorithms can be implemented in terms of the index primitives. The following pseudo-code performs what we call a *hierarchy merge join (HMJ)*. It consumes two input relations L and R that are sorted in pre-order and joins them along the ancestor axis. The left join side is pipelined through, the right side is accessed through an iterator. The order L–R is conveniently retained in the output.

```
Hierarchy Merge Join [IS_ANCESTOR] (L, R, H)
S ← ⟨⟩                                                   ▷ stack of R tuples
i ← 0                                                    ▷ position within R
for each l ∈ L do
    while S ≠ ⟨⟩ ∧ ¬H.is_descendant(l.pos, S.top().pos) do
        S.pop()
    while i ≠ R.size() do
        r ← R[i]
        if H.is_descendant(l.pos, r.pos) then
            S.push(r)
            i ← i + 1
        if H.is_before_pre(r.pos, l.pos) then
            i ← i + 1                                    ▷ r precedes l – ignore
        else
            break                                        ▷ r follows l; process l now!
    for each r ∈ S do
        yield l ∘ r
```

For each incoming L tuple l, the algorithm maintains the stack S in such a way that it contains exactly the join partners of l, by first popping obsolete tuples

from S and then pulling in further R tuples. The main virtue over HINLJ is that the HMJ can work on arbitrary inputs and thus can be used to construct bushy query plans. Supporting outer joins and other axes are straightforward extensions (as depicted in [6], for example).

The two joins we showed here are used for general-purpose hierarchy processing. In certain fields like XPath processing, more specialized join algorithms can be constructed using the query primitives. While most join algorithms in the literature are defined in conjunction with a specific labeling scheme, they can be easily adapted to work with our index-indifferent query primitives and thus remove the dependency on a specific scheme. As an example, the following code shows an adaption of the Staircase Join [43]—a prominent, optimized join for XPath processing—for the descendant axis:

```
 1: function STAIRCASEJOIN__DESC(doc, context)
 2:     for each successive pair (c1, c2) in context do
 3:         scanpartition_desc(c1, c2)
 4:     c ← last node in context
 5:     n ← end of doc
 6:     scanpartition_desc(c, n)
 -----------------------------------------------------------
 7: function SCANPARTITION__DESC(c1, c2)
 8:     for (c ← next_pre(c1); c ≠ c2; c ← next_pre(c)) do
 9:         if is_before_post(c, c1) then
10:             yield c
11:         else
12:             break                                    ▷ skip
```

Inferring this algorithm from the original definition—which is hard-coded to the pre/post labeling scheme—is quite straightforward (cf. Algorithm 2 and 3 in [43] for the original code). The same is true for other special-purpose join algorithms such as MPMGJN [105], Stack-Tree [6], and variations of TwigStack [19]. Therefore, our suggested query primitives are applicable to a broad range of existing applications and allow implementers to effectively separate implementations of indexing schemes from implementations of physical operators. Hence, we use them as the interface for our proposed indexing schemes, which we present next.

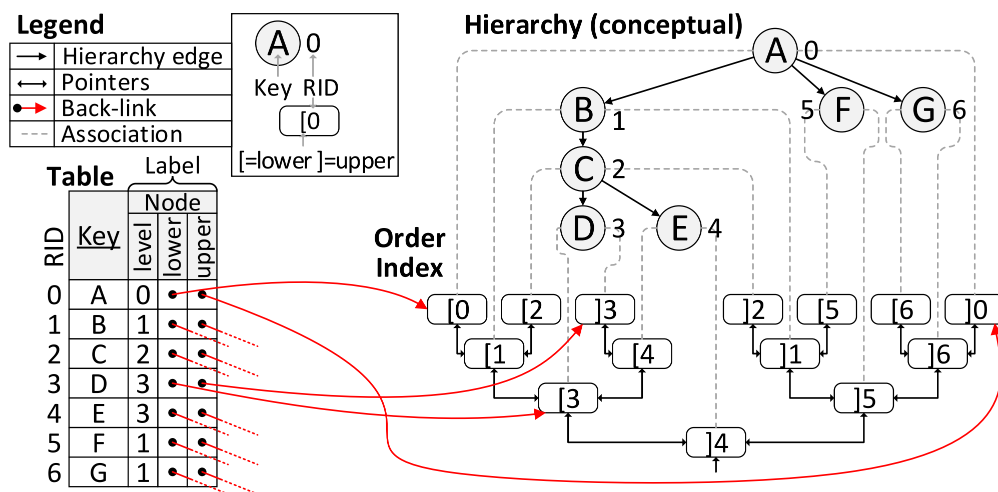


Figure 3.4: A hierarchy with Order Index (AO-Tree)

3.4 Order Indexes

To overcome the mentioned problems of prior works, we propose the concept of an *Order Index* and three specific data structures AO-Tree, BO-Tree, and O-List. They combine various ideas, from keyless trees (B-BOX) to accumulation trees (DeltaNI) through to gap allocation techniques (GapNI), to achieve increased update efficiency and robustness for dynamic workloads.

Order Indexes are index-based schemes and as such must be tightly integrated into the database system. They are designed as back ends for the hierarchy front end discussed in Chapter 2, which is integrated into TUM's HyPer [51] kernel and the SAP HANA Vora in-memory query engine [85]. Consequently, we discuss them in a main-memory context and cover disk-based systems only in a brief excursion (Section 3.5.1).

3.4.1 The Order Index Concept

An Order Index conceptually represents each hierarchy node by two interval bounds and its level. However, the bounds are not explicit numbers or other literals, but rather entries in an ordered data structure. Therefore, the index can be viewed as a dynamic representation of a containment-based scheme along the lines of NI, with *implicitly* represented bounds and explicitly maintained level. According to our model, an Order Index consists of one `NODE` table column plus the secondary index

structure. Each field in the node column is composed of three components lower, upper, and level. lower and upper are special links to the corresponding entries in the index structure. We call them *back-links*, as they refer back from a table row to an index entry, while common secondary indexes merely point from an index entry to a row through its row ID (RID).

EXAMPLE. Figure 3.4 shows an example hierarchy (top) and a pair of an Order Index and a table representing it. The index is actually an AO-Tree, which we explain in Section 3.4.2. A few exemplary back-links are shown as red arrows. An opening bracket denotes a lower and a closing bracket an upper bound; for example,]3 is the entry for the upper bound of row #3.

The index structure maintains the relative order of its entries (hence the term *Order* Index). Order Indexes provide the following interface:

entry(l) — a cursor to the entry for back-link l
rowid(c) — the id of c's associated row
is_lower(c) — whether c represents a lower bound
is_before(c_1, c_2) — whether c_1 is before c_2 in the entry order
next(c) — a cursor to the next entry in the entry order
adjust_level(c) — the *level adjustment* for c (see below)

Here, l is a back-link and c, c_1, and c_2 are cursors. A cursor is a reference to a specific entry in the index—either to a lower or an upper bound. The implementation of entry depends on how back-links and cursors are actually represented. Depending on the underlying data structure there are several options, which we discuss in Section 3.4.3. Regardless of which data structure is chosen, rowid and is_lower are implemented by storing the RID and an is_lower flag with each index entry; next corresponds to a basic traversal of the data structure. Therefore, we will only need to cover how entry, is_before, and adjust_level differ among our three implementations.

All query primitives introduced in Table 3.1 can be implemented in terms of the six index operations, as shown in Table 3.6. We use three helper functions: label looks up the label corresponding to an entry in the node column of the associated table. to_lower and to_upper yield the respective lower or upper bound of the same node, given a cursor to some entry.

The update primitives from Section 3.1.3 can be implemented as follows: insert_leaf corresponds to inserting a lower and an upper bound as adjacent entries into the data structure and storing the back-links and an initial level in the cor-

HELPER FUNCTIONS	
label(c)	table[rowid(c)].node
to_lower(c)	if is_lower(c) then c else entry(label(c).lower)
to_upper(c)	if is_lower(c) then entry(label(c).upper) else c
BINARY PREDICATES	
is_descendant(a, b)	$c \leftarrow$ entry(a.lower); is_before(entry(b.lower), c) $\wedge$ is_before(c, entry(b.upper))
is_child(a, b)	is_descendant(a, b) $\wedge$ level(a) $=$ level(b) $+ 1$
is_before_pre(a, b)	is_before(entry(a.lower), entry(b.lower))
is_before_post(a, b)	is_before(entry(a.upper), entry(b.upper))
NODE PROPERTIES	
level(a)	a.level $+$ adjust_level(entry(a.lower))
is_root(a)	level(a) $= 0$
is_leaf(a)	$c \leftarrow$ entry(a.lower); rowid(next(c)) $=$ rowid(c)
INDEX ACCESS	
find(a)	entry(a.lower)
rowid(c)	rowid(c)
TRAVERSAL	
next_pre(c)	$n \leftarrow$ to_lower(c); do $n \leftarrow$ next(n) until is_lower(n); n
next_post(c)	$n \leftarrow$ to_upper(c); do $n \leftarrow$ next(n) until $\neg$is_lower(n); n
next_sibling(c)	next(to_upper(c))
next_following(c)	next_pre(to_upper(c))

Table 3.6: Implementing the query primitives

responding table row. delete_leaf simply removes the two entries from the data structure and the table row. For relocate_range, we conceptually "crop" the corresponding range of bounds $[a, b]$, then alter the level adjustment—the value returned by adjust_level—for that range, and finally reinsert $[a, b]$ at the target position. As explained in Section 3.1.3, the other updates are implemented in terms of these operations, so we do not cover them explicitly. Section 3.4.4 will explain updates in full detail.

Level adjustments enable us to maintain level information dynamically. adjust_level is always added to the level stored in the table row (cf. level(a) in Table 3.6). This way we avoid having to alter the table in case of a range relocation; rather, we update the level adjustment of the relocated range. To do this efficiently we reuse a technique that we originally applied in DeltaNI [36] (cf. Chapter 4) for different

purposes: *accumulation*. Accumulation works for any hierarchically organized data structure that stores its entries in blocks, such as a B-tree or a binary tree (where the "blocks" are just nodes). The idea is to store a *block level* with each block. The level adjustment of an entry e is obtained by summing up the levels of all blocks on the path from e's block to the root block. This allows us to efficiently alter levels during a range relocation: After cropping the bound range $[a, b]$, we add the desired level delta δ to the block level of the root block(s) of that range, which effectively adds δ to the levels of *all* entries within $[a, b]$. Accumulation brings along the cost that level(a) becomes linear in the height of the data structure, usually $\mathcal{O}(\log n)$. However, during an index scan, the level adjustment can be tracked and needs to be refreshed only when a new block starts. This yields amortized constant time for level.

3.4.2 Order Index Structures

As specific Order Index structures, we propose the AO-Tree based on a keyless AVL tree, the BO-Tree based on a keyless B$^+$-tree, and the O-List based on a linked list of blocks. Since the BO-Tree and the O-List organize their entries into larger blocks of memory, we call these two *block-based* Order Indexes.

AO-Tree. Self-balancing binary trees, such as the AVL tree (our choice) or the red-black tree, offer logarithmic complexity for most operations, which makes them good candidates for an Order Index structure.

Because some required algorithms navigate from the bottom towards the root rather than the other way round, we must maintain pointers to parent blocks. Then, to compute adjust_level, for example, we sum up all block levels on the path from an entry to the root block, as outlined before. Since the trees are balanced, the worst-case complexity for this navigation is $\mathcal{O}(\log n)$.

EXAMPLE. The top of Figure 3.5 shows the AO-Tree from Figure 3.4 in more detail. The red numbers to the right of the entries are the block levels. The purple arrows show how to evaluate level for node E: We start with the value 2 from the table row #4 and sum up the block levels from [4 upwards to get $2 + 2 - 2 + 1 = 3$.

We check the entry order relation is_before(c_1, c_2) by simultaneously walking up the tree, starting at the two corresponding entries e_1 and e_2, to their least common ancestor e', and finally checking which of the two paths arrives at e' from the left.

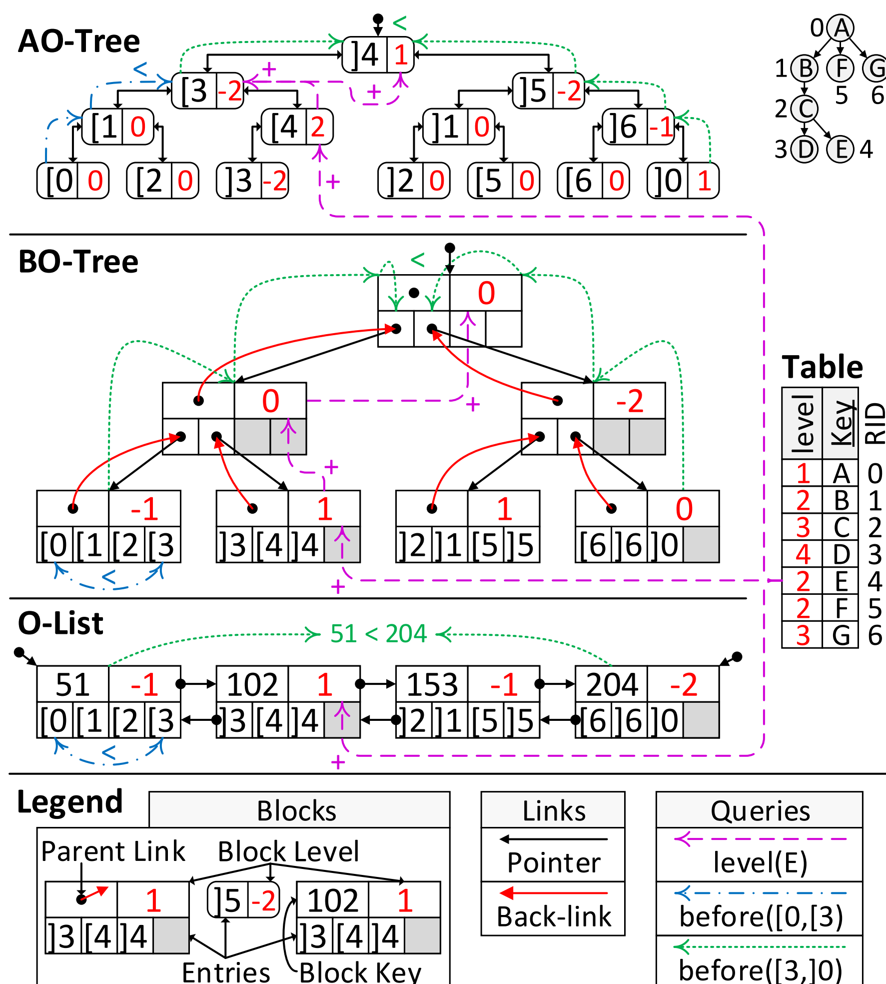


Figure 3.5: Query evaluation in the Order Indexes

EXAMPLE. In Figure 3.5, we evaluate is_descendant(D, A) by checking is_before([0, [3) (blue) and is_before([3,]0) (green).

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

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

a back-link to its parent block and a block level. In an inner block there are no separator keys but only child block pointers. An entry in a leaf block consists of a row ID and an is_lower flag. To save memory and allow some optimizations, we can store the is_lower flags separately from the row IDs in a bitfield. B^+-trees have pointers between neighboring leaf blocks for faster scans. The BO-Tree also has these pointers, but we omit them in our descriptions and figures for the sake of simplicity. Most B^+-Tree operations, including splitting and rebalancing, need almost no adaptions. Key search is no longer required since BO-Trees are keyless and the table stores back-links to leaf entries rather than keys (cf. Section 3.4.3). A cursor directly references an entry within a leaf block; more precisely, a BO-Tree cursor c consists of a pointer c.block to the block hosting the entry and a position index c.pos. Back-links to parent blocks are needed, because most operations involve leaf-to-root navigation. adjust_level(c), for instance, is computed by summing up all block levels on the path from the corresponding entry's leaf block to the root block.

EXAMPLE. The middle of Figure 3.5 shows a BO-Tree indexing the hierarchy from Figure 3.4. Back-links are displayed as red arrows, block levels as red numbers. The purple arrows show the level query for node E: We sum up the table level and the block levels on the path from [4 to the root, yielding $2 + 1 + 0 + 0 = 3$.

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

is_before(c_1, c_2) is evaluated as follows: If the corresponding entries e_1 and e_2 are located in the same leaf block, compare their positions within that block. Otherwise, walk up the tree to the least common ancestor lca of the two blocks containing e_1 and e_2; then determine which of the two paths enters lca from further left, by comparing the positions of the two corresponding pointers to the children through which the paths pass.

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

In the worst case, is_before must walk up the whole tree of height $\mathcal{O}(\log_B n)$ and then compare the positions of the child blocks. Section 3.4.3 shows how to

obtain these positions in $O(1)$ irrespective of the block size. Thus, the worst case complexity is $\mathcal{O}(\log_B n)$, which is a very good asymptotic bound due to the large logarithm base B. For example, a tree with $B = 1024$ is able to represent a hierarchy with 500 million nodes at a height of only 3, and thus needs at most 3 steps for a containment or level query. Since the root block will probably reside in cache, only 2 cache misses are to be anticipated in this case, so the BO-Tree is a very cache-efficient data structure.

While a large block size B is desirable for speeding up queries, it slows down updates. We can, however, take advantage of the fact that leaf blocks are updated more frequently than blocks further up in the tree, and enhance the BO-Tree with blocks of different sizes at different levels, based on the concepts described in [86]. As our evaluation shows, using small leaf blocks and larger inner blocks results in updates almost as fast as in trees with small B and queries almost as fast as in trees with large B and thus yields the best of both worlds.

O-List. Unlike AO-Tree and BO-Tree, the O-List is not a tree structure but merely a doubly linked list of blocks—hence its name. The bottom of Figure 3.5 shows an O-List for the example hierarchy. We use *block keys* to encode the order among the blocks, an idea borrowed from GapNI. Block keys are integers that are assigned using the whole key universe, while leaving gaps so that new blocks can be inserted between two blocks without having to relabel any existing blocks, as long as there is a gap. In addition to the block key, each block maintains a block level field for the level adjustment. The blocks are comparable to BO-Tree leaf blocks without a parent block, and we treat them in a similar manner: Inserting into a full block triggers a split; a block whose load factor drops below a certain percentage (40% in our implementation) is either refilled with entries from a neighboring block or merged with it. adjust_level(c) simply returns the level of the corresponding entry's block. is_before(c_1, c_2) first checks if the corresponding entries e_1 and e_2 are in the same block; if so, it compares their positions in the block, if not, it compares the keys of their blocks.

EXAMPLE. In Figure 3.5, we compute the level of node 4 by adding the block level 1 to the table level 2. is_before([0, [3) can be answered by an in-block position comparison. is_before([3,]0) holds because the block key 51 of [3 is less than the block key 204 of]0.

As both adjust_level and is_before reduce to a constant number of arithmetic operations, they are in $\mathcal{O}(1)$, which makes them even faster than for the BO-Tree.

But this query performance comes at the price of possibly non-logarithmic update complexity, as shown in Section 3.4.4.

3.4.3 Back-Links in Block-Based Order Indexes

We now cover possible designs for back-links in block-based Order Indexes. Back-links are used by entry(l) to locate an entry in the data structure and return a corresponding cursor. Apart from that, we use the same mechanism in the BO-Tree to look up positions of child blocks within their parent. In the AO-Tree back-links and cursors are simply direct pointers to the AVL tree nodes, since these never move in memory (so, entry(l) $= l$). For the BO-Tree and the O-List, however, entries are shifted around within their blocks or even moved across blocks by rotate, merge, and split operations. In these cases, any pointers to the entries would have to be adjusted. This causes a significant slowdown through random data access, as adjacent entries in blocks do not necessarily correspond to adjacent table tuples (recall that hierarchy indexes are secondary indexes). We investigate three approaches for representing back-links in these data structures: *scan*, *pos*, and *gap*. Figure 3.6 illustrates the three strategies for finding entry [3 in a BO-Tree (shown on the left). The relevant contents of the table are shown on the right. The address of a block is shown in its top-left corner. The table and block entries that are touched during this process are highlighted in red.

scan (top of Figure 3.6) is a simple strategy also used by B-BOX [91]. Back-links point only to the block containing the entry, which has to be scanned linearly for the row's ID to locate the entry. *scan* has the advantage that only entries that are migrated to another block during merges, splits, and rotations need their back-links updated. However, linear block scans add an unattractive $\mathcal{O}(B)$ factor to most queries and thus hinder us from using larger blocks.

pos (middle of Figure 3.6) represents back-links exactly like cursors: by a block pointer and the offset in the block. While this eliminates the $\mathcal{O}(B)$ factor and makes entry a no-op, it requires us to update back-links even when an entry is just shifted around within its block. As any insertion or deletion in a block involves shifting all entries behind the corresponding entry, this slows down updates considerably, especially for a larger B.

As a compromise, we propose ***gap*** (bottom of Figure 3.6), again using the idea of gaps from GapNI: Each entry is tagged with a *block-local key* (1 byte in the example) that is unique only within its block. A back-link consists of a block pointer and a

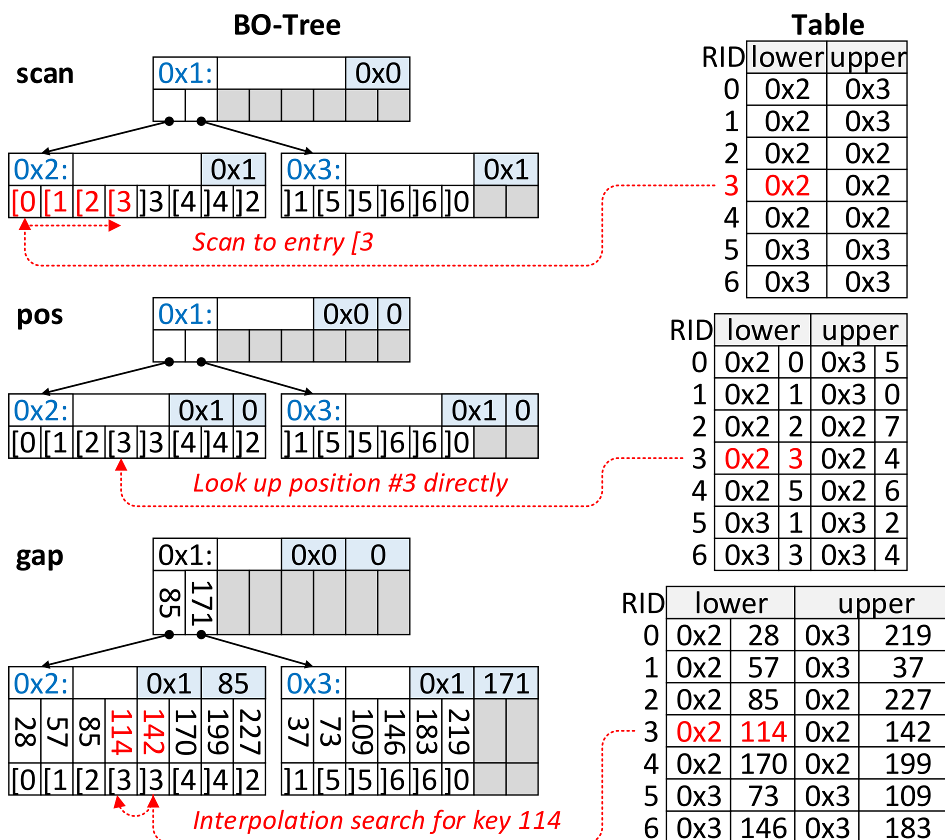


Figure 3.6: Using back-links to find entry [3

key. Initially the keys are assigned by dividing the key space equally among the entries in a block. When an entry is inserted, it is assigned the arithmetic mean of its neighbors; if no gap is available, all entries in the block are relabeled. The block-local keys are used to locate an entry using binary search or interpolation search. Interpolation search is very beneficial, as block-local keys are initially equally spaced and thus perfectly amenable for interpolation. A block may even be relabeled proactively once an interpolation search takes too many iterations, since this is a sign for heavily skewed keys. The occasional relabeling makes *gap* significantly cheaper than *pos*, which effectively relabels half a block, on average, on *every* update. Like with GapNI, an adversary can trigger frequent relabelings through repeated insertions into a gap. That said, even frequent relabelings would not pose a serious problem, as they are restricted to a single block of constant size B.

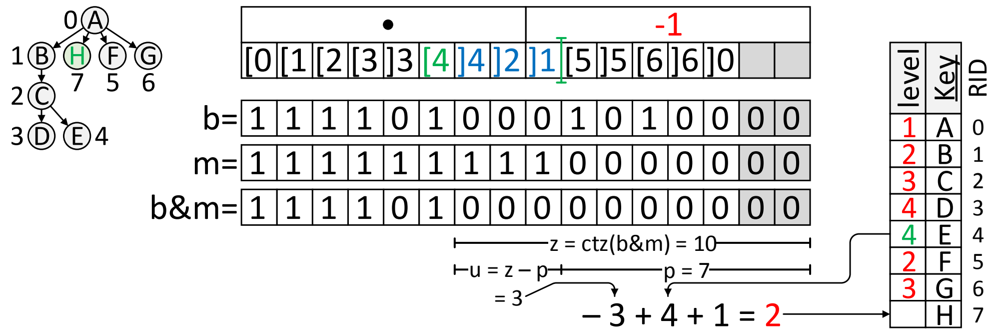


Figure 3.7: Inserting new leafs H and I in a BO-Tree

3.4.4 Updating Order Indexes

We now examine the update operations from Table 3.3 for AO-Tree, BO-Tree, and O-List. Most operations are adaptions of the standard algorithms of the underlying AVL- and B$^+$-trees. Note that the position of a node or range to be relocated or deleted, as well as the target position p of an insert or relocate, are always known a priori: Regardless of the data structure, they are simply represented as a cursor indicating a position in the bound sequence. Therefore, we can be even more efficient than the standard algorithms, which issue a key search first.

Leaf Updates

In general, insert_leaf and delete_leaf correspond to insertions and deletions of the two associated bounds in the index structure using adaptions of the standard algorithms; relocate_leaf is performed by deletion and subsequent reinsertion.

AO-Tree & BO-Tree. As the position of an insert or delete is given, we avoid the $\mathcal{O}(\log n)$ key search and achieve an amortized runtime of $\mathcal{O}(1)$ in the AO-Tree and $\mathcal{O}(B)$ in the BO-Tree. There is, however, one challenge when inserting a new leaf node at a level l: The level of the new node has to be set correctly to account for the effective level adjustment at the block where its lower bound is to be inserted. The top of Figure 3.7 shows a BO-Tree with $B = 16$ where a new leaf H is to be inserted on level 1. If we were to enter 1 into the level field in the table, the level adjustment of -1 would result in an erroneous reported level of 0. Therefore, insert_leaf first has to subtract the effective level adjustment from the real level l and enter this value into level (in this case 2). Unfortunately, the computation of adjust_level runs in $\mathcal{O}(\log n)$ for the AO-Tree and $\mathcal{O}(\log_B n)$ for the BO-Tree.

But we can circumvent the adjust_level call and achieve $\mathcal{O}(1)$ average-case insert complexity anyway, by inferring the level from a neighboring lower bound. We traverse the data structure either forward or backward from the insert position until we find a lower bound (recall that only lower bounds carry level information). If we find none after a constant number s of steps, we fall back to the logarithmic implementation. Otherwise we look up the level l in the table for this bound. Let u be the number of upper bounds we traversed. The level to enter into the table is then either $l + 1 - u$ if traversing backward or $l + u$ if traversing forward.

EXAMPLE. Figure 3.7 shows this process for scanning to the left. We traverse over $u = 3$ upper bounds (blue) and reach [4 so we take its level $l = 4$ from the table and calculate $4 + 1 - 3 = 2$.

Since every second bound is a lower bound on average, a small $s < 10$ works well. The BO-Tree with packed is_lower flags enables a significant optimization: Bitwise operations allow us to scan 8 bytes of this bitfield ($s = 64$ bounds) at once, as shown in Figure 3.7. We compute $z = \mathsf{ctz}(b \,\&\, m)$ for backward traversal or $z = \mathsf{clz}(b \,\&\, m)$ for forward traversal, where mask m has all bits set in the direction of the scan, and ctz/clz count the trailing/leading zeros (a hardware-accelerated operation on modern CPUs). Within only a few CPU cycles, we can obtain u by subtracting the insert position p from z, and also infer the index of the lower bound from z.

O-List. For the O-List, leaf updates have constant $\mathcal{O}(B)$ amortized average-case time complexity, though insert_leaf has a linear *worst-case* complexity: If a block overflows, it has to be split into two, and the new block needs an appropriate key. If no gap is available, the keys of all blocks are relabeled, yielding equally spaced keys again. For a block size of B, there are $\mathcal{O}(\frac{n}{B})$ blocks, so the complexity of relabeling is $\mathcal{O}(\frac{n}{B})$. Splitting a block is in $\mathcal{O}(B)$. Therefore, the worst-case complexity of a leaf insertion is $\mathcal{O}(\frac{n}{B} + B)$.

Although the O-List uses the same concept as GapNI for its block keys, it mitigates the relabeling problem in two ways: first, it reduces the time a relabeling takes by factor B; second, it also multiplies the minimal number of inserts that can possibly trigger a relabeling by that factor. In a basic GapNI encoding using 64-bit integers, an adversary can force a relabeling every $64 - \log n$ insertions by aiming all of them into a specific gap. So a hierarchy of one million nodes would need a relabeling every 44 inserts. For the O-List, this value is multiplied by B. For example, for a huge hierarchy of one billion bounds with $B = 1024$ and 64-bit block keys, the adversary would need roughly 35,000 insertions at the same position to trigger

relabeling, and this would touch only around 1 million blocks rather than 1 billion individual bounds. Thus, the amortized relabeling cost per skewed insertion is not just a factor B but a factor of B^2 smaller than for basic GapNI. When insertions happen in a less skewed manner, relabeling will rarely ever be triggered. To increase robustness even further, a wider data type for block keys (e. g., 128-bit integers) can be chosen without sacrificing too much memory, since only one key per block is required. We could even use a variable-length encoding such as CDBS [56] to avoid relabeling altogether. However, we settled for 64 bits in our implementation, since variable-length encodings are less processing-friendly than fixed-size integers, and relabeling happens rarely in sane scenarios.

Range relocation

Efficiently relocating ranges of nodes enables all other complex updates and is therefore a very important operation. Since the required algorithms are non-trivial and rarely found in AVL- and B^+-trees on which AO-Tree and BO-Tree are based, we provide a detailed discussion and pseudo code for our three indexes. In the pseudo code we make use of some basic functions depicted in Table 3.7. Note that in block-based Order Indexes, back-links must be updated whenever entries are migrated from one block to another, and also—in case the *pos* approach is used—whenever they are moved around in their block during an update. For brevity reasons, we omit this detail in the pseudo code. In addition, operations marked with [*] in Table 3.7 need to adjust the block levels so as to maintain a constant effective level adjustment for the involved entries (cf. Section 3.4.4).

We implement relocate_range$([a, b], p)$ in terms of move_range(c_s, c_e, c_t, δ). Cursors c_s and c_e represent the first entry [a and the entry behind the last entry]b to be relocated; cursor c_t represents the target before which the range is to be placed; δ is the level difference between the source and the target position. Each node in $[a, b]$ must have its level adjusted by this value.

AO-Tree. Range relocations are implemented in terms of the $\mathcal{O}(\log n)$ operations split and join as described in [36]: split splits a binary tree into two:

AO-TREE	
c.{left, right, parent}	left child / right child / parent block of c
c.level	block level of block c
rotate_{left, right}(c)*	performs a left / right rotation at c
rebalance(c)*	rebalances c if necessary, returns new root
unlink(c)*	unlinks c from its parent block
link_{left, right}(c, c_c)*	links c_c as left / right child of c
min(c)	a cursor to the leftmost block in the tree
remove(c)*	removes block c; rebalances if necessary
insert_before(c, c_b)*	inserts c before c_b and rebalance
BO-TREE	
b.{parent, level}	parent block / block level of block b
b.entries	array of entries in block b
b.size	number of entries stored in block b
find_pos(b)	position of block b in its parent block
is_leaf_block(b)	whether b is a leaf block
insert_block(b, b_c, p)	inserts block b_c as p-th child of b
move_entries(b, p, n, b', p')	moves n entries from block b at position p to block b' at position p'
rebalance(b)*	rebalances b, returns merged block or null
O-LIST	
first, last	first / last block in the O-List
bcount	number of blocks in the O-List
b.{key, level}	block key / block level of block b
b.{prev, next}	pointer to previous / next block
count_blocks(b, b')	number of blocks in the block range $[b, b']$

Table 3.7: Basic functions and fields used during range relocation

```
 1: function AO-Tree::split(c)
 2:     while c.parent ≠ null do
 3:         if c.parent.left = c then
 4:             c ← rotate_right(c.parent)
 5:             rebalance(c.right)
 6:         else
 7:             c ← rotate_left(c.parent)
 8:             rebalance(c.left)
 9:     c_l ← c.left; c_r ← c.right
10:     unlink(c_l); unlink(c_r)
11:     c_r ← insert_before(c, min(c_r))
12:     return (c_l, c_r)
```

We first rotate the block before which we want to split up the tree, until it is the root. After each rotation we restore the balance with an additional rotation if necessary (lines 2–8). Once the block is the root, we cut the left link to obtain the left tree. We cut the right link and reinsert the root as smallest entry into the right tree to obtain two balanced binary trees (lines 10–12). The inverse operation join concatenates two binary trees:

```
1: function AO-Tree::join(c_l, c_r)
2:     c ← min(c_r)
3:     remove(c)
4:     link_left(c, c_l); link_right(c, c_r)
5:     return rebalance(c)
```

We first remove the smallest entry c from the right tree c_r (lines 2–3). Then, we link c_l and c_r below c (line 4). Since c_l and c_r may have different heights, we must restore balance afterwards and return the root of the tree after rebalancing (line 5).

```
1: function AO-Tree::move_range(c_s, c_e, c_t, δ)
2:     (c_1, c_2) ← split(c_s); (c_3, c_4) ← split(c_e)
3:     join(c_1, c_4)
4:     c_3.level ← c_3.level + δ
5:     (c_5, c_6) ← split(c_t)
6:     c_7 ← join(c_5, c_3); c_8 ← join(c_7, c_6)
7:     return c_8
```

Based on split and join, we perform move_range by first cropping out the range $[c_s, c_e[$ into a tree rooted in c_3 (line 2) and re-joining the rest of the tree (line 3). Then we apply the level delta to c_3 (line 4). We split the tree at the target position, insert c_3 there, and return the merged tree (lines 5–7).

BO-Tree. To perform move_range(c_s, c_e, c_t, δ), we first crop out the range to be relocated into an own tree b' and rebalance underutilized blocks at the crop edges. Then, we insert b' at the target position and again rebalance underutilized blocks:

```
1: function BO-Tree::move_range(c_s, c_e, c_t, δ)
2:     (b', b_1, b_2) ← crop_rec(c_s.block, c_s.pos, null,
3:                    c_e.block, c_e.pos, null, δ)
4:     check_underflow({c_s.block, b_1, c_e.block, b_2})
5:     b'' ← insert_block_rec(b', c_t.block, c_t.pos, null)
6:     check_underflow({c_t.block, b''})
```

The cropping of the range crop_rec is performed by starting at the leaf blocks c_s.block and c_t.block and splitting blocks recursively upwards the tree until the least common ancestor block is encountered:

```
 1: function BO-Tree::crop_rec(b_s, p_s, b'_s, b_e, p_e, b'_e, δ)
 2:     if b_s ≠ b_e then
 3:         p_1 ← find_pos(b_s); p_2 ← find_pos(b_e)
 4:         b_1 ← split(b_s, p_s, b'_s); b_2 ← split(b_e, p_e, b'_e)
 5:         b ← crop_rec(b_s.parent, p_1, b_1, b_e.parent, p_2, b_2, δ)
 6:         return (b, b_1, b_2)
 7:     else
 8:         b ← new Block                                        ▷ LCA found
 9:         move_entries(b_s, p_s, p_e − p_s, b, 0)
10:         if ¬is_leaf_block(b_s) then
11:             insert_block(b, b'_s, 0)
12:             insert_block(b_s, b'_e, p_s)
13:         b.level ← block_level(b_s) + δ
14:         return (b, null, null)
15: function BO-Tree::block_level(b)
16:     if b = null then 0 else b.level + block_level(b.parent)
```

Once we are at the least common ancestor (LCA) when $b_s = b_e$, we create a new block b and move all entries in the range into it (lines 8–9). If this block is an inner block, then there are extra child blocks b'_s and b'_e which originated from splitting the blocks below the least common ancestor. We have to add these child blocks at their respective positions (lines 10–12). Block b is now the root of a new tree which contains all entries in the cropped range. By altering its block level by δ, we alter the effective levels of all nodes in the range (line 13).

The insertion insert_block_rec of the cropped block b is again done recursively by splitting up blocks at the target position until a block of the same height is reached:

```
1: function BO-Tree::insert_block_rec(b, b_t, p, b_c)
2:     if b.height > b_t.height then
3:         p_t ← find_pos(b_t)
4:         b' ← split(b_t, p, b_c)
5:         insert_block_rec(b, b_t.parent, p_t + 1, b')
6:         return b'
7:     else
8:         alter_entry_levels(b, b.level − block_level(b_t))
9:         insert_block(b_c, b_t, p)
10:        move_entries(b, 0, b.size, b_t, p)
11:        delete b
12:        return null
```

Once such a block b_t is reached, the levels of entries in b are adjusted by b.level − block_level(b_t) so that their effective level will stay the same in their new environment (line 8). The new child block b_c, which originated from the split of the lower blocks, is inserted into b_t and then all entries of b are moved into b_t (lines 9–10). Note that this may trigger a split of b_t. Now, all entries are incorporated in b_t, so b can be deleted. The altering of entries during insert_block_rec is performed as follows:

```
1: function BO-Tree::alter_entry_levels(b, δ)
2:     if is_leaf_block(b) then
3:         for each c in b do
4:             if is_lower(c) then
5:                 label(c).level ← label(c).level + δ
6:     else
7:         for each c in b do
8:             b' ← b.entries[c.pos]
9:             b'.level ← b'.level + δ
```

If b is a leaf block, we must alter the level column for lower bound entries (lines 3–5); otherwise, we must alter the block level of child blocks (lines 7–9).

Note that the used split function, which splits a block b at a position p, takes as an additional parameter a child block b_c split in the level below, which must be incorporated as first child of the new (right) block b'.

```
1: function BO-Tree::split(b, p, b_c)
2:     b' ← new Block; b'.level ← b.level; o ← 0
3:     if b_c ≠ null then
4:         b'.entries[0] ← b_c; o ← 1
5:     for each i between p and b.size do
6:         b'.entries[i − p + o] ← b.entries[i]
7:     b'.size ← b.size − p + o; b.size ← p
8:     return b'
```

Checking for block underflows is different from an underflow check in a standard B$^+$-tree. It must be performed recursively up to the root (line 12), because there may be underutilized inner blocks even if the leaf block below them has a valid capacity:

```
1: function BO-Tree::check_underflow(S)
2:     L ← ∅
3:     for each b in S do
4:         if b ≠ null ∧ ¬L.contains(b) then
5:             check_rec(b, L)
6:     for each b in L do delete b
7: function BO-Tree::check_rec(b, L)
8:     if b ≠ null then
9:         if b.size < underfill_limit then
10:            b_merged ← rebalance(b)
11:            if b_merged ≠ null then L.add(b_merged)
12:        check_rec(b.parent, L)
```

We need to check not only one leaf block but rather a whole set S of possibly neighboring leaf blocks (line 3). Therefore, we cannot instantly delete blocks that became obsolete due to a merge, since they may be in the set S to be processed subsequently. So, we first move them to a free set L (line 11) and delete them after S has been processed thoroughly (line 6).

EXAMPLE. Figure 3.8 shows how the subtree rooted in node B is relocated below node F in the example BO-Tree. We omit back-links and block levels (except for the root). After T_1 is cropped out (red lines), we apply $\delta = +1$ to its root, since the target parent F is one level higher than the old parent A. We rebalance the trees (green arrows), which shrinks T_2. Now we split T_2 behind [5 to create a gap for T_1 (blue lines). This increases

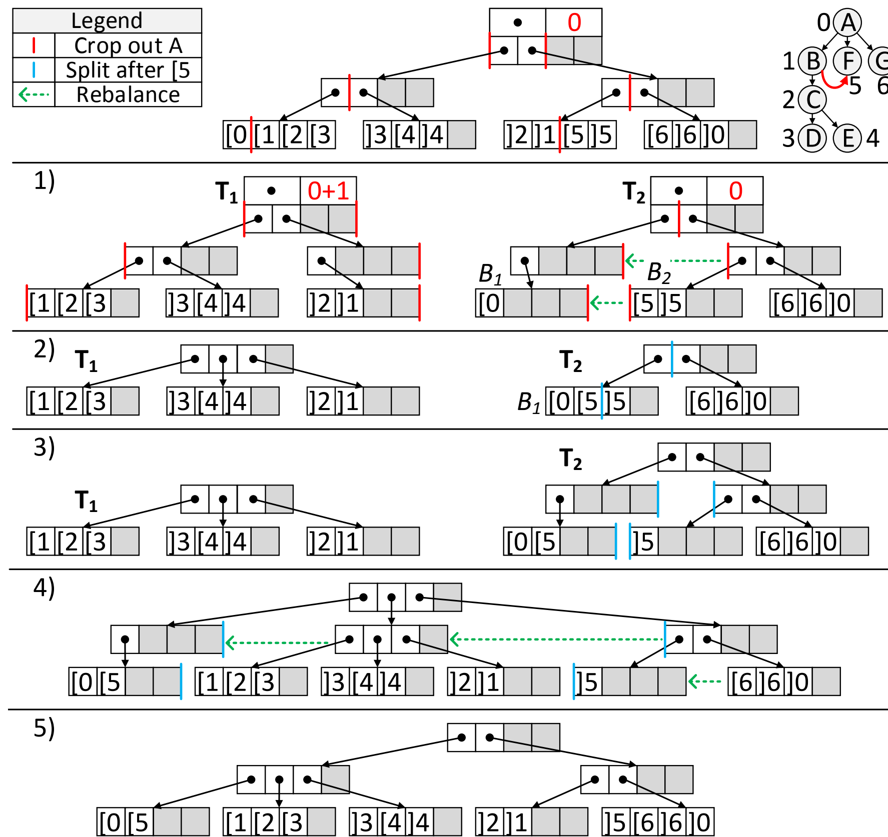


Figure 3.8: Relocating B below F in a BO-Tree

its height by one. Now, we add T_1 as second child of the new root of T_2. We rebalance underutilized blocks for a properly balanced result.

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

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

```
1: function O-List::move_range(c_s, c_e, c_t, δ)
2:     (b_1, b_2) ← split(c_s.block, c_s.pos)                          ▷ 1)
3:     (b_3, b_4) ← split(c_e.block, c_e.pos)
4:     b_1.next ← b_4; b_4.prev ← b_1
5:     check_underflow({b_1, b_4})                                     ▷ 2)
6:     (b_5, b_6) ← split(c_t.block, c_t.pos)                          ▷ 3)
7:     b_5.next ← b_2; b_2.prev ← b_5
8:     b_3.next ← b_6; b_6.prev ← b_3                                  ▷ 4)
9:     for each b from b_2 to b_3 do                                   ▷ 5)
10:        b.level ← b.level + δ
11:    g ← b_6.next.key − b_5.prev.key
12:    n ← count_blocks(b_5, b_6)
13:    if n < g then
14:        reassign_block_keys(b_5, b_6, g, n)
15:    else
16:        reassign_block_keys(first, last, MAX_INT, bcount)
17:    check_underflow({b_5, b_2, b_3, b_6})                           ▷ 6)
18: function O-List::split(b, p)
19:    b′ ← new BLOCK; b′.level ← b.level; b′.key ← b.key
20:    b′.next ← b.next; b.next.prev ← b′
21:    b′.prev ← null; b.next ← null
22:    move_entries(b, p, b.size − p, b′, 0)
23:    return (b, b′)
```

1) We crop out the range and 2) rebalance the crop edges. 3) We split at the target position and 4) link the cropped range in. 5) We update the block levels (lines 9–10) and the block keys (lines 11–16) by either relabeling only the cropped range, if it fits into the key gap g, or the whole O-List. Finally, 6) we rebalance the insert edges. Reassigning keys for n blocks is done by subdividing the available gap g into equal gaps of size g':

```
1: function O-List::reassign_block_keys(b_s, b_e, g, n)
2:     g′ ← n/(g + 1); k ← b_s.prev.key
3:     for each b from b_s to b_e do
4:         k ← k + g′; b.key ← k
```

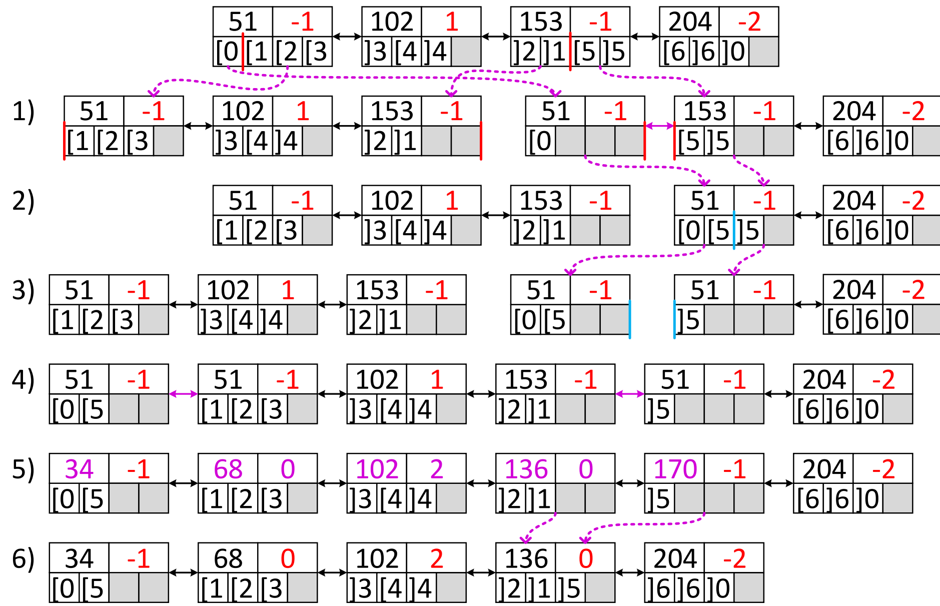


Figure 3.9: Relocating B below F in an O-List

EXAMPLE. Figure 3.9 shows the relocation of subtree B under node F. 1) The block range for node B is cropped out. 2) Underfull block 51 is merged with 153. 3) The list is split after [5 and 4) the cropped block range is linked in. 5) The block levels are adjusted by +1. The gap between 0 and 204 fits the five blocks, so their block keys are relabeled to divide it evenly. 6) Underfull block 136 is merged with 170.

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

Rebalancing and Level Adjustments

Order Indexes must be rebalanced like the data structures they are based on. AO-Tree maintains balance through rotations. BO-Tree and O-List fill underutilized

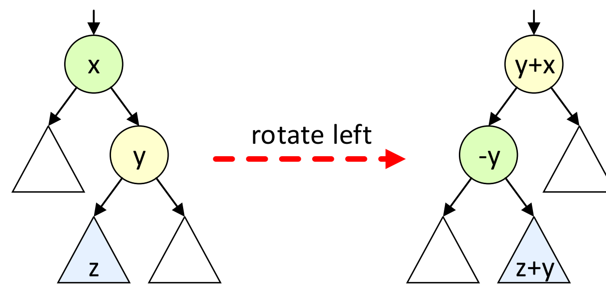


Figure 3.10: Updating levels after left rotation in an AO-Tree [36]

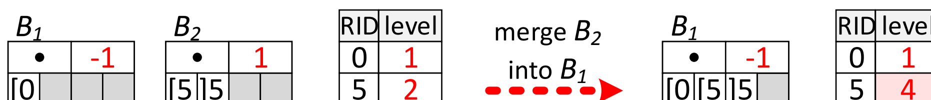


Figure 3.11: Adjusting levels after a leaf block merge

blocks by taking entries from neighboring blocks or merging two blocks. Overfull blocks must be split. An additional challenge in these cases is maintaining the level adjustment. For the AO-Tree, the adjustments to the levels of the rotated blocks is illustrated in Figure 3.10 (for a left rotation; right rotation is analogous). For the BO-Tree and the O-List any entry that is moved from a block B_2 to a neighboring block B_1 during rebalancing must have its level adjusted by B_2.level $-$ B_1.level. If B_1 and B_2 are inner blocks in the BO-Tree, the values to be adjusted are the block levels of their child blocks. If they are leaf blocks, the values to be adjusted are the level fields in the table, but only for lower bound entries.

EXAMPLE. Figure 3.11 shows the merging of the two leaf blocks B_1 and B_2 from Figure 3.8. Entry [5 is moved to B_1, so its table level becomes $2 + 1 - (-1) = 4$. No level update is performed for]5 because it is an upper bound.

When splitting a block, the new block can simply inherit the block level from the old block.

Bulk Building

Irrespective of the Order Index we use, bulk_build performs a single pre/post traversal of the given tree representation: when a node is pre-visited, its lower bound is appended to the index, and when it is post-visited, its upper bound is appended. This traversal effectively constructs the index structure from left to right. For the AO-Tree and the BO-Tree we use the standard $\mathcal{O}(n)$ algorithms

for bulk-building such data structures. For the O-List, the algorithm is also $\mathcal{O}(n)$ if we delay the assignment of evenly-spaced block keys until all blocks are filled. During bulk building, each block initially receives a block level of 0 and the absolute node levels are entered into the level component of the label column. To transform a relational hierarchy representation into a representation that can be pre/post traversed efficiently, we use the bulk-build operator described in Section 2.7.3. Thus we are able to bulk build all Order Indexes efficiently.

3.5 Order Index Extensions

This section presents two extensions for Order Indexes in disk-based systems and for ordinal query operations.

3.5.1 Disk-Based Systems

Even though we designed Order Indexes for main-memory database systems, they can be tuned for disk-based systems as well. On first sight, the block-based indexes BO-Tree and O-List seem particularly promising. By matching the block size with the disk page size, we can minimize the anticipated I/O operations. Using the BO-Tree in a disk-based system is straightforward, as it is based on a B^+-tree which is traditionally used for such systems. Therefore, we focus on the O-List in the following. As our evaluation shows, O-List generally outperforms BO-Tree for most queries, since all query primitives can be answered by considering only the blocks hosting the given entries, while BO-Tree often has to traverse the block structure upwards. Usually, leaf updates also require only one page access—the block hosting the corresponding bound entries—as long as no relabeling of block keys is necessary. However, once this happens, an expensive update to *all* blocks is needed. Moreover, when performing range relocations, all blocks in the relocated range need to be updated. These cases are painfully inefficient, since a lot of pages are loaded and written back to disk just for updating one or two values (block level and block key) within each page.

Therefore, we propose a variant of O-List for disk-based systems: We separate the header data of each block—i. e., the block level, the block key, and (logical) next and previous pointers—from the entry array, and condense all block headers in a separate layer of header pages. The bottom of Figure 3.12 illustrates this design.

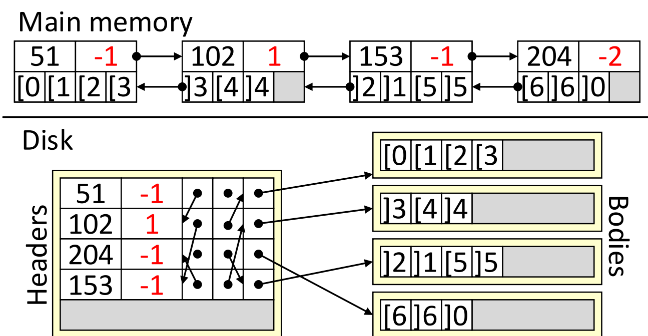


Figure 3.12: Main memory and disk implementation of an O-List

Each yellow frame represents one page. The order of headers within the pages does not have to match the list order. This allows us to perform range relocations efficiently by only adjusting a few pointers instead of shifting headers.

With this O-List variant, adjusting the header information of a range of blocks touches only one or a few header pages. The drawback is that we now need two page accesses to work with the entries of a block. However, this is somewhat alleviated as the header pages are frequently accessed and thus likely to be paged-in already. Relabeling a very large O-List will still touch many header pages and issue a lot of disk I/O, but the pages used during this relabeling are known in advance, so they can be prefetched. Therefore, O-List is an attractive choice for disk-based systems.

3.5.2 Supporting ordinal primitives with the BO-Tree

The BO-Tree can be enhanced to support the ordinal query primitives from Section 3.1.2. Ranks are basically aggregated bound counts: pre_rank(a) is equal to the number of *lower* bounds between the first entry in the index and a.lower, inclusively. Similarly, post_rank(a) is the number of *upper* bounds up to a.upper. Therefore, the basic idea is to add and maintain such bound counts within inner blocks, and let pre_rank and post_rank reuse the appropriate counts rather than count all bounds one by one. This is illustrated on the left side of Figure 3.13. In addition to the child block pointer, each inner block entry e now has num_lower and num_upper values counting the lower and upper bounds in all leaf blocks reachable through e.

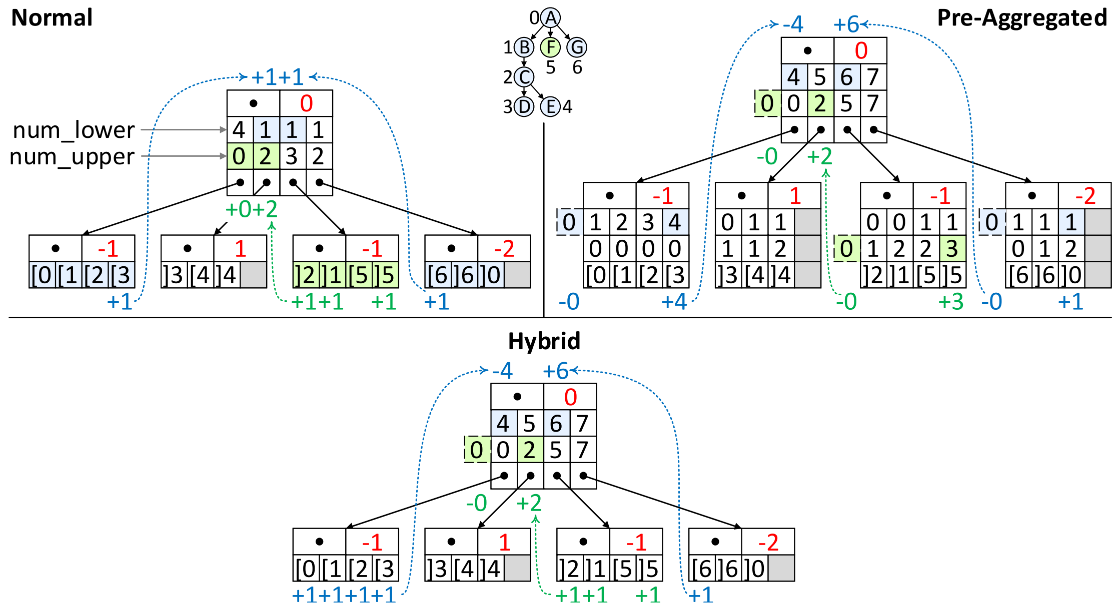


Figure 3.13: Ordinal queries in the BO-Tree

To evaluate post_rank(a), we first locate the upper entry $c =$ entry(a.upper) and count the upper bounds between c.pos and the beginning of c.block. Afterwards, we locate the inner entry e' pointing to c.block on the parent block and add up all num_upper values from the beginning of that block up to but excluding e'. We repeat this up to the root block to obtain the total count.

EXAMPLE. Figure 3.13 shows the query post_rank(F) in green. We start from the upper bound]5 of F and add up the three upper bounds]2,]1, and]5. Then we walk up to the parent block and add up the entries to the left, that is, 0 and 2. Thus, the result is $3 + 0 + 2 = 5$.

For the inverse operation select_post(n) we must do the opposite: We start by scanning the root block, subtracting the num_upper fields from n, until we reach the first entry e where num_upper $\geq n$; then we follow e's child pointer. We continue like this until we reach a leaf block. Finally, we determine the n-th upper bound within that block and return a cursor for the corresponding node.

EXAMPLE. Figure 3.13 shows the query post_rank(F) $= 5$, so we can visualize select_post(5) by doing the inverse. We start with 5 at the root and subtract -0 and -2. The next subtraction -3 would yield 0, so we go down to the third child. We scan the child for the third upper bound which is the correct entry]5.

pre_rank and select_pre are analogous. subtree_size(a) is just a shorthand for range_size($[a, a]$). To implement range_size($[a, b]$), we start from e_l = entry(a.lower) and e_r = entry(b.upper) and scan their leaf blocks to the right and to the left, respectively, adding up the lower bound counts. Then we scan the parent block in the same fashion (excluding the entries from which we came) while adding up the num_upper fields. Once we reach the least common ancestor block we scan the range between the entries and return the overall sum.

EXAMPLE. Figure 3.13 displays the query subtree_size(A) in blue. We start at the entries [0 and]0. In their leaf blocks, we have 4 and 1 lower bounds, respectively. We walk up to the parent block and add the two num_upper values 1 and 1 of the entries between. Thus, the final result is $4 + 1 + 1 + 1 = 7$.

Maintaining the num_lower and num_upper fields increases the complexity of leaf updates from $\mathcal{O}(B)$ to $\mathcal{O}(B + \log_B n)$, because one field per BO-Tree level has to be updated. The described rank/select algorithms run in $\mathcal{O}(B \log_B n)$, because we scan $\mathcal{O}(\log_B n)$ blocks of size B. range_size runs in $\mathcal{O}(B \log_B s)$ on average where s is the number of descendants, because we only need to go up $\log_B s$ levels on average. These runtimes are mediocre, especially for large block sizes B. If we are willing to sacrifice more leaf update performance, we can get rid of this factor: If we pre-aggregate the num_lower and num_upper fields such that each value is the partial sum over all entries to its left and add these pre-aggregated fields to the leaf blocks as well, we can save the block scans during the query. This is shown on the top right side of Figure 3.13. A sum of field v from position x to position y in a block can now be emulated by calculating $v[y] - v[x - 1]$ (with $v[-1]$ being 0 implicitly). For select_pre and select_post, we need to find the position where the sum first becomes greater or equal to n. This is done through a binary search in the block.

EXAMPLE. post_rank(F) is answered by calculating $-0 + 3$ for the scan in the leaf block and $-0+2$ in the root for the final result $-0+3-0+2 = 5$. subtree_size(A) is calculated by the three scans $-0 + 4$, $-4 + 6$, and $-0 + 1$ yielding $-0 + 4 - 4 + 6 - 0 + 1 = 7$. To answer select_post(5), we first use binary search to find the third entry whose num_lower field is ≥ 5. Then we subtract the value 2 from the second entry in the root and use binary search in the leaf block to locate the first entry ≥ 3, which is]5.

Pre-aggregation replaces block scans by constant computations and thus eliminates the factor B for rank and range_size. For select, the scans are replaced by $\mathcal{O}(\log B)$ binary searches, so the runtime is $\mathcal{O}(\log_B n \cdot \log B) = \mathcal{O}(\log n)$. Maintaining the

	rank	select	size	leaf update	memory
Basic BO	–	–	–	B	0
BO Ordinal	$B \log_B n$	$B \log_B n$	$B \log_B s$	$B + \log_B n$	$o(1)$
BO Pre-Agg.	$\log_B n$	$\log n$	$\log_B s$	$B \log_B n$	$2 + o(1)$
BO Hybrid	$B + \log_B n$	$B + \log n$	$B + \log_B s$	$B \log_B n$	$o(1)$
AO Ordinal	$\log n$	$\log n$	$\log n$	$\log n$	16
O-List Ordinal	$n/B + B$	$n/B + B$	$s/B + B$	B	$o(1)$

Table 3.8: Comparison of ordinal approaches

pre-aggregations, however, adds a factor B to leaf updates, because $B/2$ aggregates per block have to be updated on average for each field that changes. Note that the asymptotic runtimes for complex updates are not affected, because these are already $\mathcal{O}(B \log_B n)$ in the standard BO-Tree.

Concerning space utilization, the num_lower and num_upper fields do cost some extra memory. But this does not hurt when they are only stored in inner blocks, because we usually have few inner blocks due to the large fan-out of the BO-Tree. In the leaf blocks, memory can be saved by using narrow data types for the counts. For example, if leaf blocks hold at most 256 entries, one byte per value suffices. To reduce space consumption even further, we can dispense with leaf block aggregates and resort to scans (hybrid approach, bottom of Figure 3.13). Provided that is_lower flags are packed into bitfields, multiple bounds can be processed at once using bitwise instructions. This way the leaf block scan becomes significantly faster than a scan in the inner blocks would be. We found this to be a good memory/runtime tradeoff.

Table 3.8 summarizes all runtimes and the extra memory consumption (in extra bytes per entry). It also includes figures for the other Order Indexes, though their efficient support for these primitives is limited, since O-List is inherently linear and AO-Tree would require too much memory. Therefore, the BO-Tree should be chosen when support for ordinal primitives is required.

3.6 Performance Evaluation

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

3.6.1 Test Setup

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

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

bulk_build — Bulk-build hierarchy H using an edge list ordered in pre-order. This simulates a scenario in which a hierarchical table is bulk-loaded from an existing relational tree encoding.

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

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

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

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

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

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

Even though our focus is on dynamic hierarchies and thus update operations, we also measure query operations, because a highly dynamic index is useless if it cannot answer queries efficiently. We first assess the query primitives is_descendant, is_child, is_before_pre, is_before_post, level, is_leaf, and find, involving randomly selected nodes of H. In addition, we measure the performance of a full index scan using next_pre repeatedly, which corresponds to a pre-order traversal over H. Beyond isolated query primitives, we also measure the query from Figure 3.3, which uses a hierarchy index nested loop join over the descendant axis and evaluates the level. The performance of an index join varies with the size of the subtree that is scanned for each tuple from the left input u. Therefore, we again use H_x and pick random children of its root r as left input u. As the query basically executes a partial index scan over x entries for each u tuple, we call this measure **scan**[x]. It represents an important query pattern and thus gives us a realistic hint of the overall query processing performance we can expect.

3.6.2 Block Size & Back-Link Representation

We first want to determine a good block size B for BO-Tree and O-List, and assess the three back-link variants from Section 3.4.3. We vary B in powers of 4 from 4 to 1024. Block size "mix" refers to a multi-level scheme: 16 for leaf blocks, 64 for height 1 blocks, and 256 for height 2 blocks and above. Figure 3.14 shows results for various update and query operations on the BO-Tree. We omit the O-List figures as they lead to the same conclusions.

Concerning updates (top) it is clearly visible that a smaller block size B between 16 and 64 is most beneficial. Concerning queries (bottom), we see that larger blocks are very beneficial, especially for the important is_descendant query (is_child, is_before_pre, is_before_post behave similarly). We seek a B that provides a good trade-off between query and update performance. For the BO-Tree, our favorite is

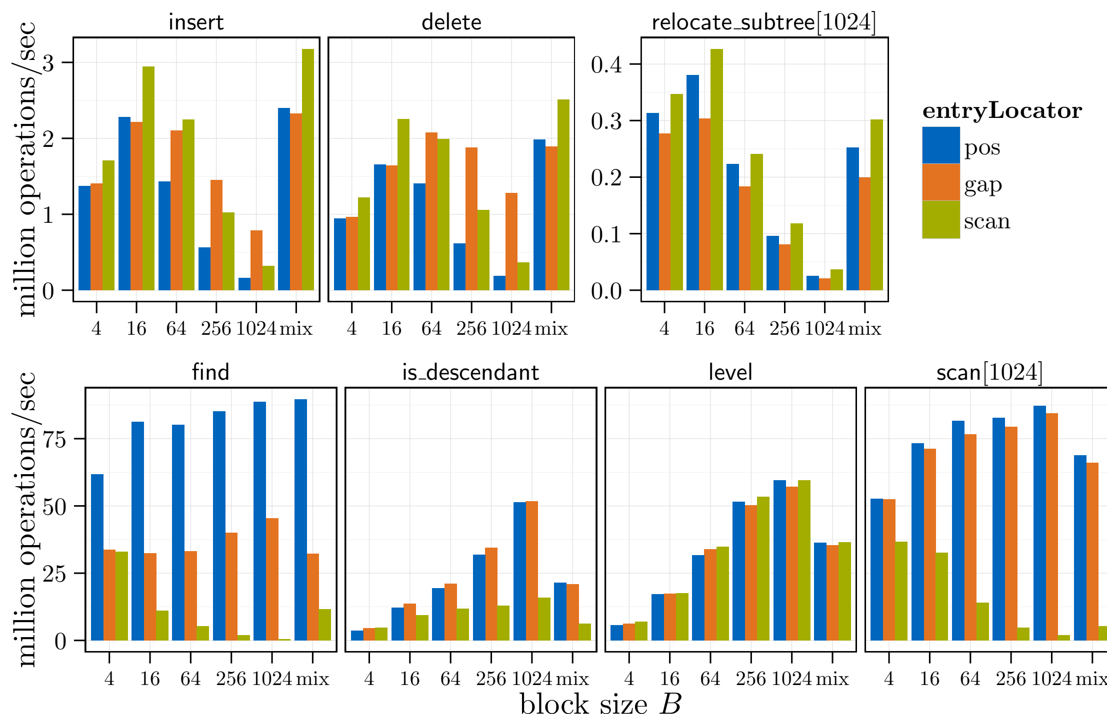


Figure 3.14: Comparing BO-Tree performance for different back-link representations and block sizes B

mix: queries are not as fast as for $B = 1024$, but still quite fast, especially on the compound scan[1024] query; in return, it is very efficient for updates, where large B such as 1024 suffer, especially when relocating subtrees (cf. relocate_subtree[1024]). For rather static data, 1024 can be a good choice nevertheless, as it maximizes query speed.

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

In conclusion, a mixed block size is the best trade-off for the BO-Tree. The *scan* back-links should be avoided; *pos* is preferable for small B and scenarios

with few updates, while *gap* is preferable for larger B and dynamic scenarios. For the remaining measurements we use *gap* back-links, B = mix for BO-Tree, and $B \in \{16, 64, 256\}$ for O-List.

3.6.3 Comparison to Existing Schemes

Our indexing schemes aim at highly dynamic settings where high update performance even for complex updates is desirable. To show that prior dynamic labeling schemes cannot support these settings efficiently, we compare our schemes AO-Tree, BO-Tree[B] and O-List[B] to promising contenders from different categories: Ordpath [74] as a representative for path-based variable-length schemes; CDBS [56] for containment-based variable-length schemes; GapNI [58] (with explicitly maintained level) for containment-based schemes with gaps. All three are backed up by a B-tree over the labels, which is used for scans where necessary (most queries can be answered by only considering the labels). Our fourth contender is DeltaNI [36] as a versioned, index-based scheme. Finally, we also measure the naïve schemes Adjacency and Linked. Note again that Adjacency does not represent ordered hierarchies, so the ordered query primitives are not defined (cf. Table 3.4). For comparison, we implemented them anyway by using the (unstable) order within the hash index buckets as sibling order.

To make a fair comparison, we implemented all contenders ourselves and tried to be as efficient as possible everywhere. For example, the cited papers for Ordpath and CDBS do not state how the variable-length labels are actually stored. Due to their variable length, storing them in the columns is tricky; out of place storage, however, introduces additional cache misses. We settle for a hybrid approach: We reserve 8 bytes per label and store small labels in place. Once the label size exceeds 8 bytes, we store it out of place and place a pointer to the location in the column. Due to their compact nature, we generally found most labels to fit in place for those schemes, except when skewed updates were assessed which bump up the label sizes.

Figure 3.15 shows the **query performance** for various types of queries. We first observe that the AO-Tree performs very poorly, as its data is scattered and the height of the AVL tree is large in comparison to a B-tree, so the index suffers from a high number of cache misses. In contrast, block-based Order Indexes perform well for all query types. For queries that labeling schemes can answer by considering only the labels, the Order Indexes are slower than labeling schemes because they

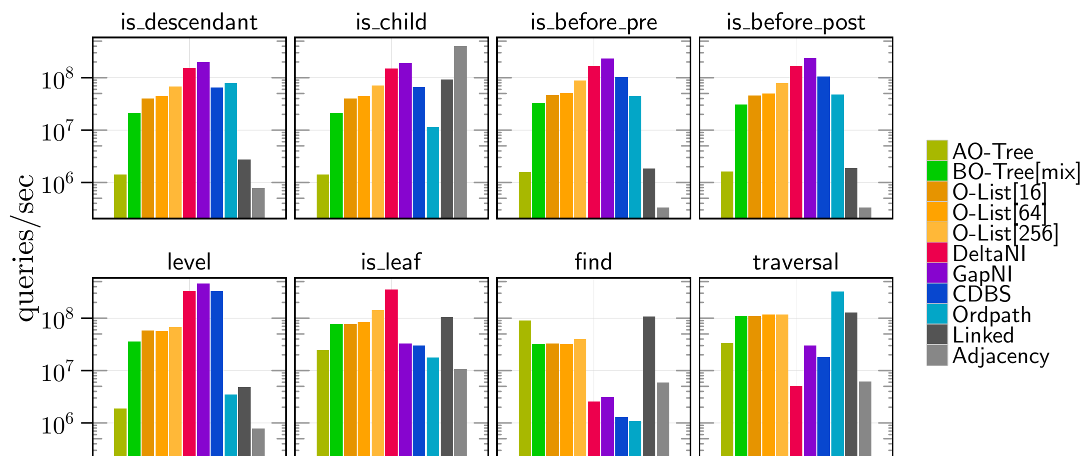


Figure 3.15: Performance of different query primitives

need to consider also the index and thus suffer at least one extra cache miss. For queries that need to access the index (e. g., is_leaf, traversal), the Order Indexes offer a very good performance. The find operation is where Order Indexes excel over labeling schemes, because they use fast $\mathcal{O}(1)$ back-links instead of $\mathcal{O}(\log n)$ B-tree key searches. DeltaNI appears to perform fine on most queries, but this is only the case because all the complexity is hidden in find, which DeltaNI must always perform before it can issue a query. The naïve schemes Linked and Adjacency show their greatest weakness here: they fail to offer robust query performance. While some queries such as is_child are fast, other important queries such as is_descendant and level are unacceptably slow.

As the queries in Figure 3.15 involve randomly selected nodes, Order Indexes are expected to be slow due to extra cache misses on accessing the index at random positions: The left side of Table 3.9 shows the average number of cache misses per operation for these queries. Note that the figure omits some queries for the sake of brevity. As we can see, block-based Order Indexes show around one to three cache misses per operation. In contrast, labeling schemes show only half a cache miss for these operations (except is_leaf and find), because Order Indexes need a random access into the index while the labeling schemes can infer the answer directly from the already-loaded labels. Thus, the performance numbers from Figure 3.15 already show a worst-case scenario for Order Indexes in comparison to labeling schemes. is_leaf also requires index access for labeling schemes, so they show more cache misses here. find uses efficient back-links for Order Indexes and thus only involves

	Random Position Queries						Scans		Updates			
	is_descendant	is_child	is_before_pre	level	is_leaf	find	traversal	scan[1024]	delete	insert	relocate_ subtree[1024]	relocate_ range[1024]
AO-Tree	20.7	21.7	20.6	10.4	2.8	1.2	0.42	0.38	60	21	133	182
BO-Tree[mix]	3.2	3.2	2.5	2.2	1.1	2.1	0.02	0.03	15	15	320	277
O-List[16]	2.5	2.5	2.1	1.8	1.1	2.0	0.02	0.03	13	13	329	234
O-List[64]	2.1	2.1	2.0	1.8	0.9	2.0	0.02	0.03	16	19	170	154
O-List[256]	1.0	1.1	0.9	1.4	0.4	1.7	0.02	0.02	21	37	309	179
DeltaNI	0.6	0.7	0.6	0.3	0.3	14.4	0.34	0.44	118	93	41	90
GapNI	0.4	0.4	0.4	0.2	3.0	13.3	1.94	0.19	23	27	31795	6467
CDBS	0.1	0.1	0.2	0.3	3.2	14.4	2.28	0.27	36	32	619	945
Ordpath	0.3	0.4	0.3	0.2	3.7	12.6	0.04	0.12	21	23	900	468
Linked	9.0	1.1	12.7	5.7	0.9	1.0	0.06	0.05	17	10	4	259
Adjacency	17.2	0.3	36.5	17.8	2.7	7.0	1.98	0.78	19	15	9	391

Table 3.9: Number of cache misses per operation

around two cache misses. In contrast, the B-tree lookup of labeling schemes costs over ten cache misses.

In contrast to the random position queries, when scanning the index, the current block is already in cache and level becomes an $\mathcal{O}(1)$ operation, so we expect our schemes to perform better in this case. Figure 3.16 shows the hierarchy index nested loop join performance scan$[x]$ with varying x. All schemes benefit from larger subtrees, because the initial find is the most expensive operation and always incurs a cache miss. The subsequent index scan often incurs no further cache misses due to prefetching. On this query, all block-based Order Index variants are superior. Linked is second fastest as it just has to chase pointers, which is fast but not as cache-friendly as scanning blocks. GapNI is also quite fast by virtue of its processing-friendly integer labels, while CDBS and Ordpath suffer from their variable-length labels. Ordpath additionally suffers from the level query that requires counting path elements; without this, it would be on a par with CDBS. DeltaNI is quite slow; it pays the price of a full-blown versioned scheme. The slowest of all is Adjacency, which has to use its hash indexes repeatedly to find child nodes to be scanned. When looking at the cache misses in the "Scans" columns of Table 3.9, it becomes clear why Order Indexes excel once the index is scanned: Block-based Order Indexes have almost no cache misses when scanning the index and can therefore outperform labeling schemes.

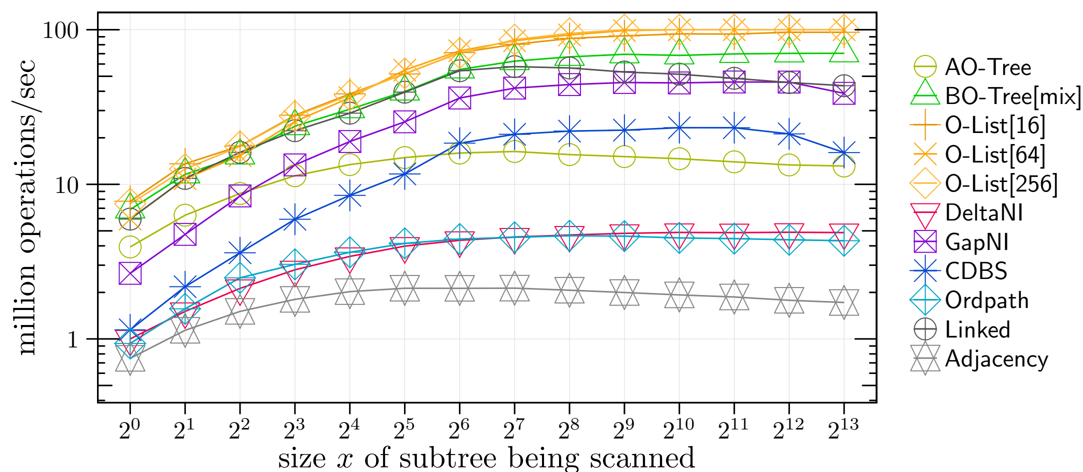


Figure 3.16: Compound query performance (scan[x])

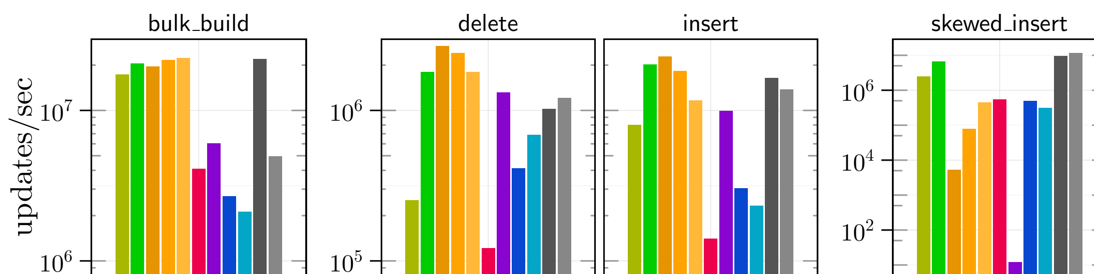


Figure 3.17: Bulk build and leaf update performance

We conclude that Order Indexes excel at a hierarchy index nested loop join. Since joins are at the core of common hierarchy queries, we anticipate outstanding overall query performance in real-world scenarios.

Figure 3.17 depicts the **update performance** for bulk building and leaf updates. As anticipated in Table 3.5, Order Indexes and the naïve schemes perform very well for most updates. For bulk building and non-skewed leaf updates, Order Indexes are superior to labeling schemes, though the contenders—in particular GapNI—also perform reasonably. Skewed insertions, however, force frequent relabelings, so GapNI's performance plummets. O-List has the same problem, as its block keys are also gap-based and skewed insertions fill up these gaps. Still, as anticipated, O-List significantly outperforms GapNI because new block keys are required less often; the larger B, the better. So, an O-List with sufficiently large blocks can handle skewed insertions quite well. BO-Tree even benefits from skewed insertions, as the blocks where the insertions happen will usually be in cache, and so it outperforms labeling

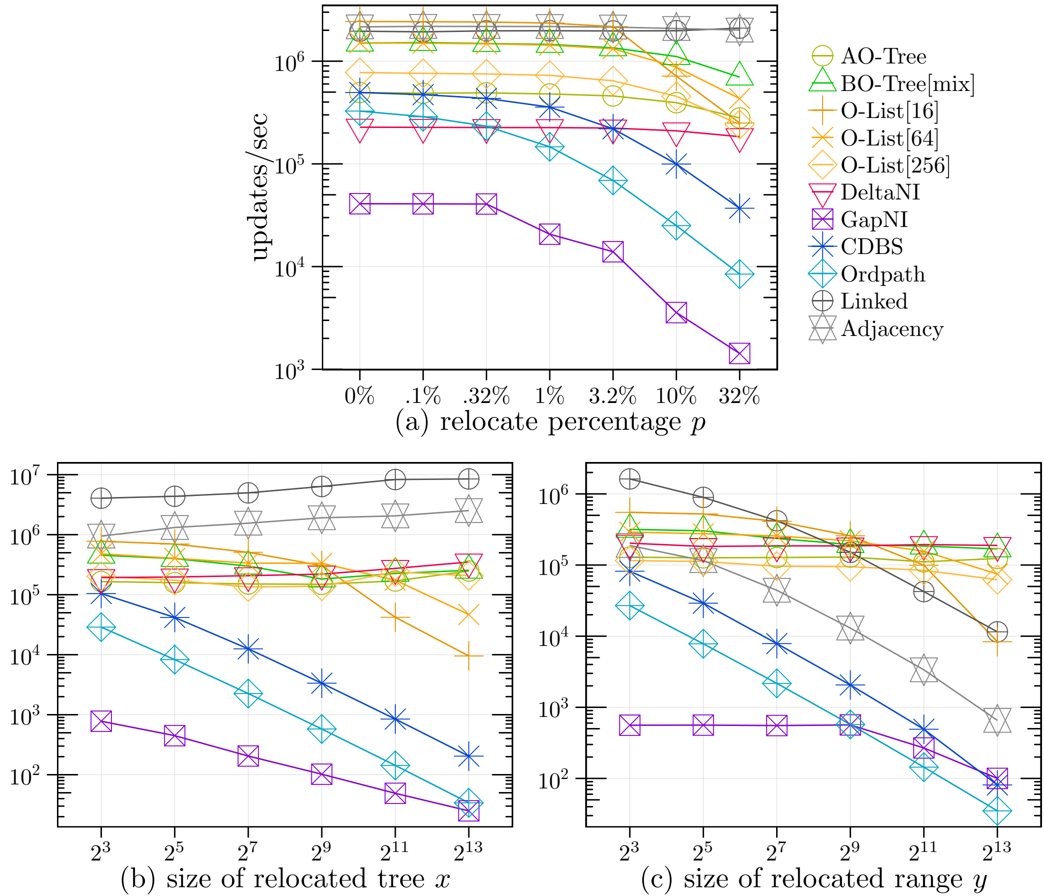


Figure 3.18: (a) mixed_updates[p] (b) relocate_subtree[x] (c) relocate_range[y]

schemes by a factor of around 20. Considering cache efficiency, block-based Order Indexes are generally more cache efficient than labeling schemes, as can be seen in columns delete and insert of Table 3.9.

Let us now consider complex updates. Figure 3.18 (a) shows the workload simulation mixed_updates[p] with p varying exponentially between 0% (no relocations) and 32%. While all indexing schemes perform acceptably for $p = 0\%$, the contenders drop rapidly on increasing p; starting at only $p = 0.32\%$ for labeling schemes. In this scenario the size of the relocated subtrees is “only” 107; larger sizes exacerbate the situation.

To investigate this, relocate_subtree[x] over varying x, the size of the relocated subtrees, is shown in Figure 3.18 (b). As we anticipate, all contenders but the naïve schemes drop linearly in x, as they have to relabel all x nodes. GapNI is particularly slow due to additional global relabelings. At $x = 8192$, labeling schemes can handle

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

Table 3.10: memory consumption in bytes per node

only around 100 updates per second, while Order Indexes remain fast at around 200,000 updates per second. The figures also attest that an O-List is just a list of blocks as opposed to a robust tree structure: it turns slow when too many blocks are involved (relabeled) in a relocation. The figure suggests a rule of thumb: Once x exceeds $64B$, performance drops noticeably (at 1024 for O-List[16] and 4096 for O-List[64]), so B should be chosen accordingly.

The naïve schemes handle subtree relocation exceptionally well by just updating single values or pointers. This advantage vanishes as soon as we consider the more potent sibling range updates in Figure 3.18 (c). Now, the naïve schemes have to process all siblings individually, so their performance drops with increasing range sizes. Only Order Indexes and DeltaNI handle these updates well, but again, O-List turns slow when the range becomes too large. Considering cache efficiency, Table 3.9 shows that Order Indexes cause around 100 to 300 cache misses per complex update, but these figures are independent of the size of the relocated range. In contrast labeling schemes cause more cache misses which also grow with the size of the relocated entity. Only DeltaNI can handle all kinds of updates with less than 100 cache misses.

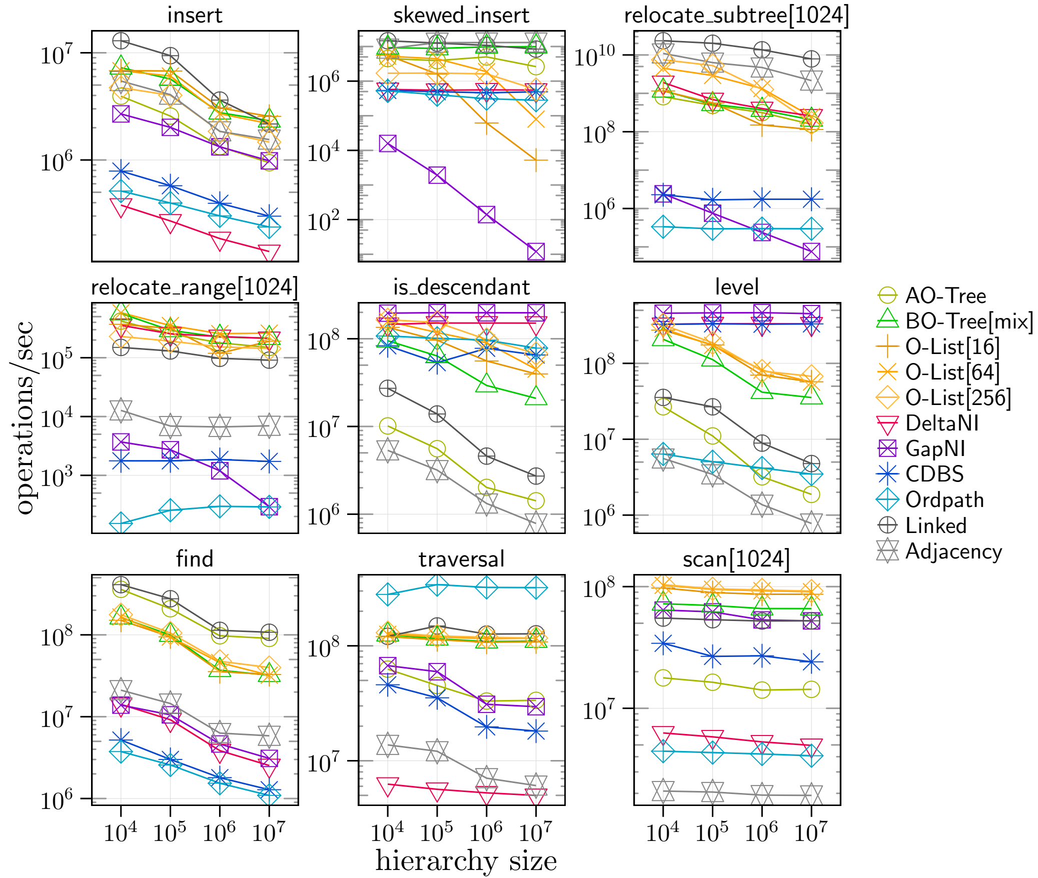


Figure 3.19: Performance over varying hierarchy size

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

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

Table 3.10 compares the **memory consumption** of all indexes in bytes per hierarchy node. The first column is the size after bulk-building H, the second

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

Until now, all experiments were conducted on a hierarchy of 10^7 nodes. To assess the **scalability** of the indexing schemes, we also measured on hierarchies of size 10^4, 10^5 and 10^6. Figure 3.19 shows the results for various kinds of updates and queries. For space reasons we omitted some queries and updates that show a comparable behaviour. Considering simple updates, Order Indexes and Linked suffer the highest performance losses when going from 10^5 to 10^6, because the index no longer fits into L3 cache. For the other contenders, performance always drops because their updates are in $\mathcal{O}(\log n)$. Considering complex and skewed updates, most contenders suffer almost no losses for larger hierarchies, as the operations are mostly compute-bound and the constant overhead is large in comparison to the $\log n$ factor some contenders possess. Only O-List and GapNI suffer more from larger hierarchies, because their occasional relabeling becomes more expensive.

Considering the random position queries is_descendant and level, schemes accessing the index incur extra cache misses and therefore suffer between 10^5 and 10^6, where L3 cache size is reached. The drop for find is comparable for all contenders, because they all need to access the index. However, Order Indexes and Linked keep their performance between 10^6 and 10^7 nodes while other indexes drop further. The reason for this is that the former use an $\mathcal{O}(1)$ find algorithm while the latter use a $\mathcal{O}(\log n)$ one. Finally, queries accessing the index linearly (traversal and scan[1024])

do not drop at all for all schemes which use a block-based index. Other indexes suffer a bit when the hierarchy does not fit into L3 cache.

In conclusion, most contenders scale exceptionally well on almost all operations and therefore can be used also for very large hierarchies. The only case where performance drops more noticeably is on skewed and complex updates for GapNI and O-Lists because of the relabeling, so the BO-Tree is preferable for very large hierarchies with a lot of skewed and complex updates.

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

3.7 Conclusion

In this chapter we have investigated indexing schemes for highly dynamic hierarchical data. Our first contribution is a set of abstract query and update primitives for dynamic indexing schemes. This carefully designed interface manages a balancing act between providing the required functionality for many common scenarios, and allowing for efficient implementations for most indexing schemes. Our discussion of applications in RDBMS and XML query processing demonstrates its usefulness. The primitives also provide the foundation for our analysis of existing indexing schemes with respect to the targeted dynamic scenarios. Our analysis leads us to the finding that existing schemes bear three main problems: lack of query capabilities, insufficient complex update support, and vulnerability to skewed updates. We therefore propose Order Indexes as an efficient indexing technique for highly dynamic settings. They can be viewed as a dynamic representation of a nested intervals labeling, using the concept of accumulation to maintain node levels while supporting even complex and skewed updates efficiently. Of our three

implementations AO-Tree, BO-Tree, and O-List, the latter two yield robust and competitive query and update performance. Order Indexes considerably outperform prior techniques when considering complex updates on subtrees, sibling ranges, and inner nodes. Our evaluation shows how carefully choosing a suitable back-link representation and block size can further optimize performance. The BO-Tree with varying block sizes yields a particularly attractive query/update tradeoff, making it a prime choice for indexing dynamic hierarchies.

By using Order Indexes as a storage back end in conjunction with the operators described in Section 3.3 and the front end proposed in Chapter 2, we can enrich a relational database system with efficient and user-friendly hierarchical data support.

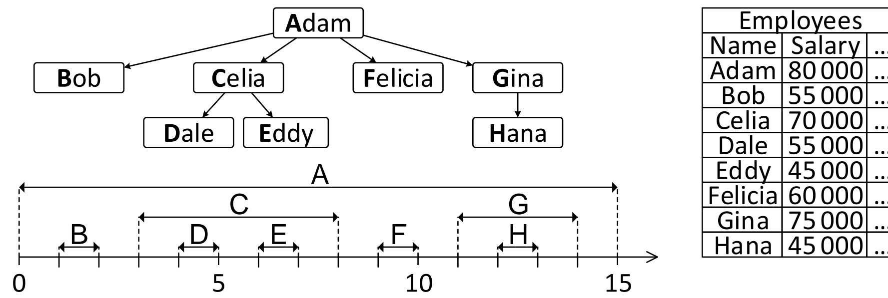
CHAPTER 4

DeltaNI: Indexing Versioned Hierarchical Data

Parts of this chapter have previously been published in [36].

Various kinds of large hierarchies exist in Enterprise Resource Planning (ERP) applications. For example, companies need to manage human resource (HR) hierarchies which model the relationship between their employees (cf. Figure 4.1), 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 bills of materials (BOM), which 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 [14]). This data is also used for reporting purposes that feature complex OLAP-style queries over various recursive structural properties of the hierarchies.

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



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 4.1: An HR hierarchy and its NI encoding

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.

4.1 Hierarchies in RDBMS

We base the work in this chapter upon the data model from Chapter 2. That is, we represent a hierarchy by a `NODE` table column plus a secondary hierarchy index.

While the Order Indexes proposed in Chapter 3 work on unversioned data, the DeltaNI index proposed in this chapter can handle versioned hierarchical data. Note however, that DeltaNI only versions the hierarchy structure. The attributes of the hierarchy nodes stored in the table must be versioned with usual relational versioning concepts, such as [50]. This relational versioning is out of the scope of this chapter.

NI Encoding. Like our Order Indexes from the previous chapter, DeltaNI is built upon the nested intervals (NI) labeling scheme. However, the two schemes use different techniques to make NI dynamic. While Order Indexes rely on representing interval bounds by entries in an ordered data structure, DeltaNI keeps the usual integer bounds but maintains extra data structures to track their value changes. 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 4.1 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. Especially, the index supports all query primitives we have gathered in Section 3.1.2, so it can also be used as a back-end for our hierarchy framework devised in the previous chapters. As described in Section 3.3, it can also be used to implement various other tree-aware join algorithms such as the staircase join [43], the Stack-Tree join [6], or the TwigStack join [19] in order to answer XPath-style queries. In this capter, we focus on providing an efficient NI encoding for each version of the hierarchy.

Therefore, aggregate or diff queries that span more than one version are out of the scope of this chapter.

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 $(v.\mathsf{upper} - v.\mathsf{lower} - 1)/2$. For example, a subtree query in the hierarchy from Figure 4.1 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, as shown in Section 3.3.

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

- $\mathsf{insertBefore}(b)$: Inserts a new node before interval bound b.
- $\mathsf{moveSiblingRangeBefore}(v, v', b)$: Moves all siblings between v and v' (inclusively) and their descendants before bound b. v must be a left sibling of v' or $v = v'$.
- $\mathsf{deleteSiblingRange}(v, v')$: 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 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. As described in Section 3.1.3, sibling range relocation subsumes all other kinds of updates, so DeltaNI is able to handle all complex kinds of updates efficiently (cf. Table 3.5).

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. However, Order Indexes should be preferred in the non-versioned case (cf. Section 3.6). Finally, the deltas in this approach can also be used for logging, as a delta accurately describes a set of changes.

4.2 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 $[v.\mathsf{lower}, v.\mathsf{upper}]$. 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: $\delta_{V \mapsto V'}(b_1) = v.\mathsf{lower}$ and $\delta_{V \mapsto V'}(b_2) = v.\mathsf{upper}$. 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., $\delta(v.\mathsf{lower}) \leq \max(V)$.

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.

4.3 Implementing the Query Primitives

Now that we have defined the notion of a delta, we show how we implement the query primitives from Table 3.1 (Page 47) with it. By implementing these primitives, we are able to plug DeltaNI into our hierarchy framework from the previous chapters. Indeed, we can reuse the interface of Order Indexes for DeltaNI. Thus, we must only implement the Order Index functions entry, rowid, is_lower, is_before, next, and adjust_level. Once we have these implementations, we can implement the query primitives as shown in Table 3.6 on Page 65.

Before we can implement the Order Index interface, we must first define which values are to be stored in the NODE column and what a back-link l and a cursor c in the context of DeltaNI is. Given a base version V_0, another version V_x, and a delta $\delta_{V_0 \mapsto V_x}$, we store the base version interval of the node in the NODE column. For a label a from this column, the integer values of this interval represent the back-links a.lower and a.upper. Order Indexes also need a level field in the column. For DeltaNI this field does not exist and is instead assumed to be always zero.

Since the back-links of a node are bound values of the base version V_0, back-links conceptually reside in the bound space of V_0. In contrast, a cursor c resides in the bound space of a specific version, which may be V_0 but may also be another version V_x for which a delta is available. Since Order Indexes base all operations on cursors, any query primitive executed on a cursor residing in V_x will automatically yield the correct results for this version. Of course, using cursors of different versions in the same query primitive is meaningless, but this will never happen since any query is issued in a specific version V_x and works only on cursors in V_x.

To implement all operations of the Order Index interface, we need some additional data structures:

A row mapping M — a mapping from bounds in the base version V_0 to the id of the corresponding table row. M can simply be implemented as a hash table. Note that bounds in a base version never change their value, so M needs no update if nodes are relocated in the hierarchy. Thus, node relocation stays efficient even if large ranges are moved. In fact, the maintenance of M does not change the asymptotic runtimes of any query or update operation.

A level delta $\Delta_{V_0 \mapsto V_x}$ — While $\delta_{V_0 \mapsto V_x}$ represents changes to the values of interval bounds between V_0 and V_x, $\Delta_{V_0 \mapsto V_x}$ represents changes to the values of node levels between these versions. When a node is newly inserted in V_x, its Δ is set

Function	Implementation
entry(l)	$\delta(l)$
rowid(c)	$M(\delta^{-1}(c))$
is_lower(c)	table[rowid(c)].node.lower $= \delta^{-1}(c)$
is_before(c_1, c_2)	$c_1 < c_2$
next(c)	$c + 1$
adjust_level(c)	$\Delta(\delta^{-1}(c))$

Table 4.1: Implementing the Order Index interface for DeltaNI

to its initial level in the hierarchy. We can use exactly the same data structures and algorithms for δ and Δ, so we will omit Δ in the following sections and instead explain all further concepts with δ only.

With M and Δ we can now implement the Order Index interface as shown in Table 4.1. entry performs the translation of a bound from V_0 to V_x. The rowid is obtained by transforming a bound in V_x (a cursor c) back to V_0 using δ^{-1} and then looking up the row id in M. By comparing a reverse-transformed bound $\delta^{-1}(c)$ to the lower bound stored in the table, we can check is_lower. The next function simply increments the input bound. The level adjustment is read from Δ. Since Δ tracks the level delta for a bound in V_0, we must first reverse transform the bound in V_x by applying $\delta^{-1}(c)$. Now, the query primitive implementations from Table 3.6 can be used, thus enabling DeltaNI as an index for our hierarchy framework.

Note that the cursor representation (a simple integer bound in V_x) has been chosen for simplicity reasons. In our implementation, a cursor consists of more than just the bound. For example, $\delta^{-1}(c)$ is used in many primitives. Thus, we could store this value in the cursor once we have computed it, so that it does not have to be computed twice for one cursor. In addition, we can store a direct pointer into the delta data structure to speed up further calculations. Aside from all such possible optimizations, applying DeltaNI to our hierarchy framework is conceptually straightforward.

4.4 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, that is, 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 translations of ranges of interval bounds: Let $R_2 = [a, b]$ and $R_3 = [b + 1, c]$ be two adjacent intervals and let $\mathsf{swap}(R_2, R_3)$ 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 4.1, 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 4.2 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 the left and the resulting hierarchy on the right. 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

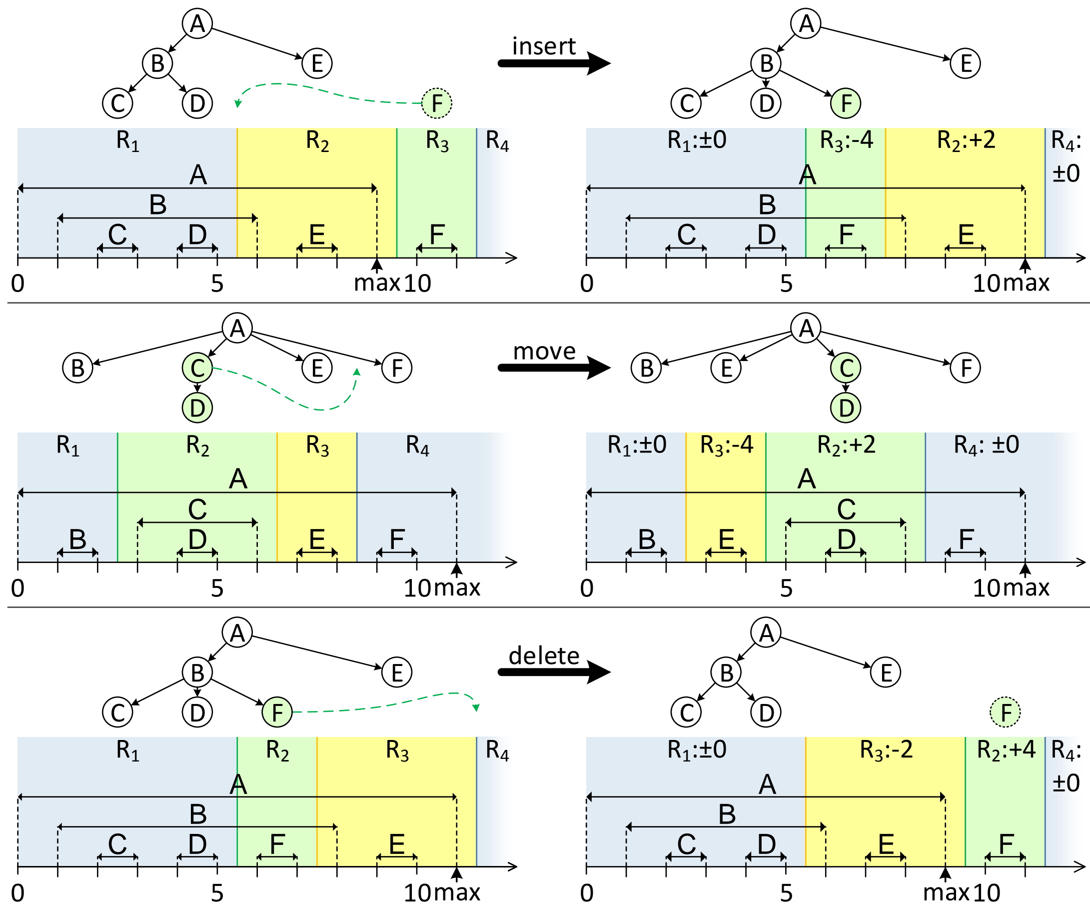


Figure 4.2: Updating with insertBefore(6) (top), moveSiblingRangeBefore$(C, C, 9)$ (middle), and deleteSiblingRange(F, F) (bottom). These operations are modeled by swapping R_2 with R_3 and updating max.

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 top 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 bottom of the figure shows how F is deleted by swapping $R_2 = [6, 7]$ and $R_3 = [8, 11]$ and reducing max.

Formally, let $\mathsf{swap}([a, b], [c, d])$ 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 $\mathsf{relocate}([x, y], z)$ 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:

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

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

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

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 4.5.2.

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 [65]). 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 computed 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 4.3 illustrates how our approach represents the delta resulting from the node insertion depicted on the top of Figure 4.2. 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

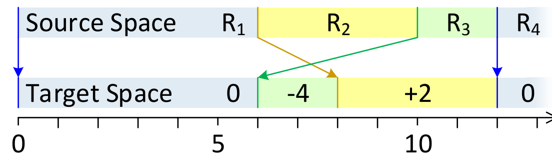


Figure 4.3: Model of translation ranges

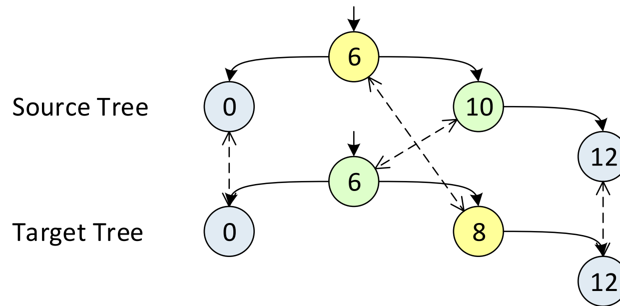


Figure 4.4: Representation of translation ranges

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 t in the target space. Thus, the delta depicted in Figure 4.3 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 4.3, 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 4.3 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 4.4 shows the source and the target tree for the translation ranges from Figure 4.3.

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.

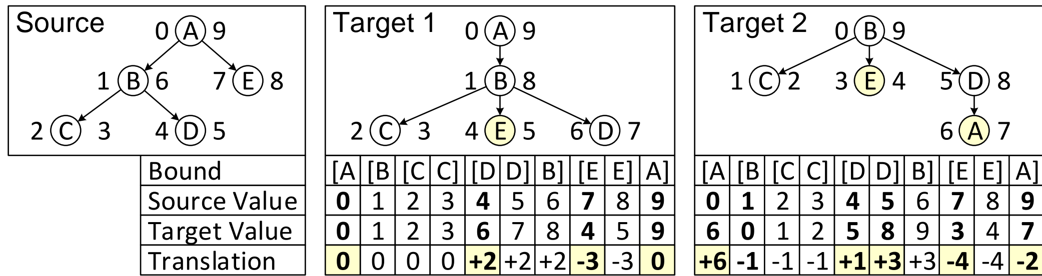
The size of the delta is in $\mathcal{O}(c)$ but is also bounded by the size of the hierarchy: The largest possible delta contains one translation range for each bound of the hierarchy. Note that repeated updates of a node or subtree (e. g., moving a tree around twice) do not create extra translation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

4.5 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.

4.5.1 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 [65]. 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 [20]), or 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**

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	**6**	**0**	1	2	**5**	**8**	9	**3**	4	**7**
Translation	**+6**	**-1**	-1	-1	**+1**	**+3**	+3	**-4**	-4	**-2**

Figure 4.5: Inferring deltas from source and target interval encoding

diff algorithm such as [30] or our RWS-Diff, which we will propose in Chapter 5, 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 4.5 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.

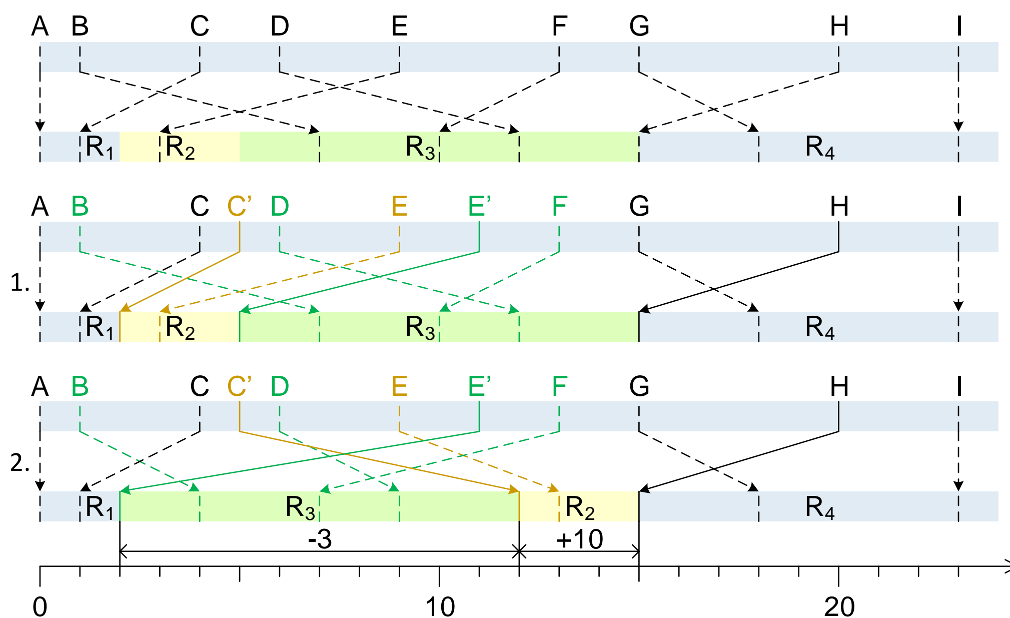


Figure 4.6: Updating a version delta by swapping translation range R_2 with R_3

4.5.2 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 4.2, 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 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 4.6 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. In fact, Section 3.4.4 contains split and join implementations for the AO-Tree, which can also be used for usual AVL trees and thus for deltas.

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 $\mathcal{O}(\log n)$.

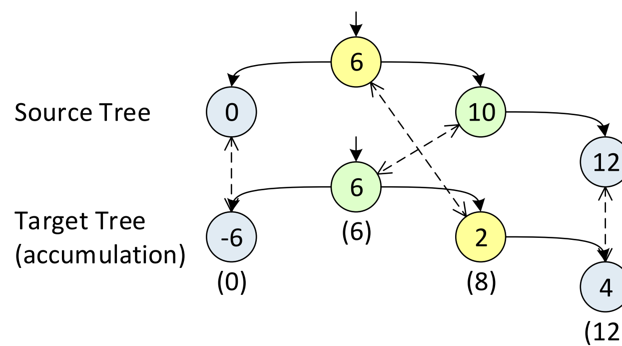


Figure 4.7: Using the accumulation tree as target tree

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* (cf. Section 3.4.1). 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 4.7 shows the delta from Figure 4.4 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 4.4) 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, and 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 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 4.8 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

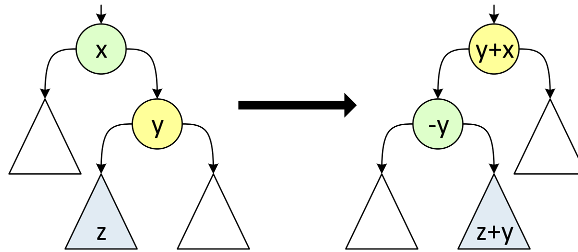


Figure 4.8: Left rotation in the accumulation tree

Figure 4.6 is depicted in Figure 4.9. On the top 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. Next, 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 bottom 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.

4.6 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. Although we only use bound deltas δ in this chapter, all findings also hold for level deltas Δ.

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

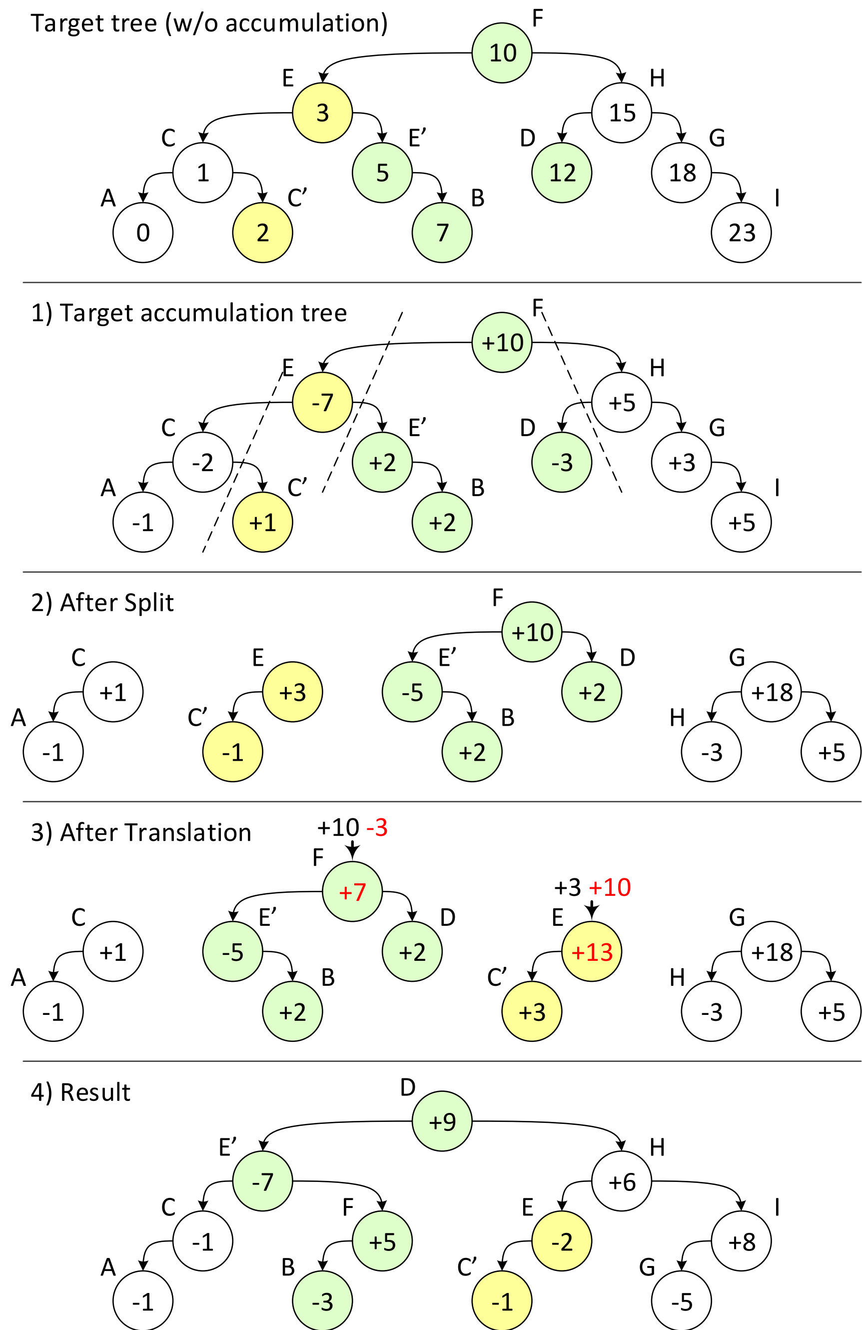


Figure 4.9: Performing the swap operation on the (accumulation) target tree.

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.

4.6.1 Querying the History

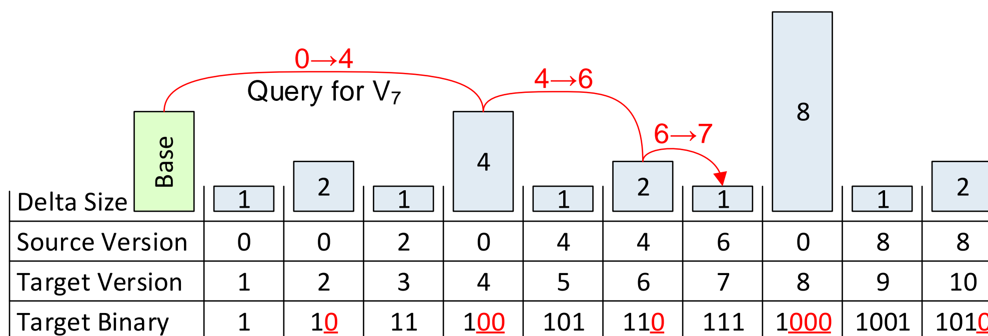
The interval bounds of each node are materialized for the base version V_0. Deltas are used to transform these bounds into any other version. The bounds in V_0 serve as durable identifiers for all the nodes, since they never change.

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 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



Delta Size	1	2	1	4	1	2	1	8	1	2
Source Version	0	0	2	0	4	4	6	0	8	8
Target Version	1	2	3	4	5	6	7	8	9	10
Target Binary	1	10	11	100	101	110	111	1000	1001	1010

Figure 4.10: Using the number of trailing zeros in the binary representation of the target version for deciding delta sizes.

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.

4.6.2 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 [93] 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 $|\delta| = 2^{\mathsf{ctz}(i)}$, where $\mathsf{ctz}(i)$ 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 4.10 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 4.10 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 $V_0 \rightarrow V_{16}$, $V_{16} \rightarrow V_{18}$, and $V_{18} \rightarrow V_{19}$.

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 $V_6 \rightarrow V_7$, $V_0 \rightarrow V_8$, $V_0 \rightarrow V_{16}$, $V_0 \rightarrow V_{32}$, 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', that is, V' is the source of the one and the target of the other delta. We define the operation $\mathsf{merge}(\delta_{V \to V'}, \delta_{V' \to V''})$ 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 $\mathsf{merge}(\delta_1, \delta_2)$ 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 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 4.11 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

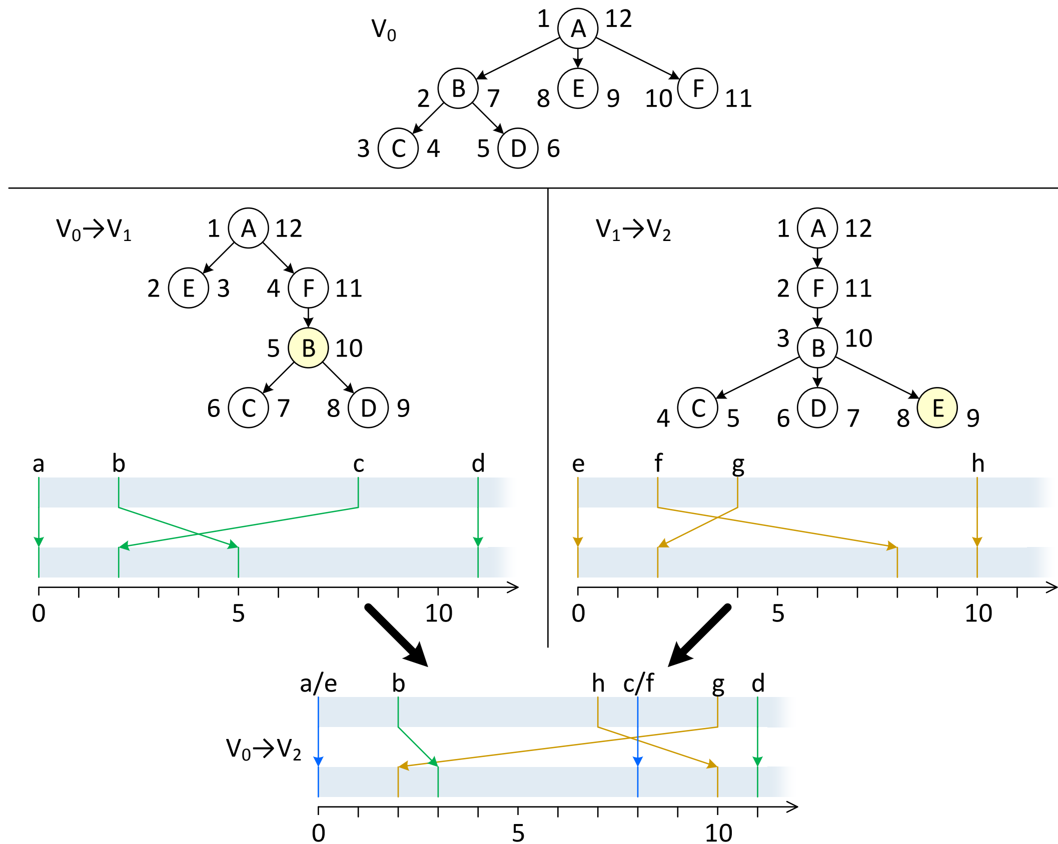


Figure 4.11: Merging two deltas

the respective tree. A merge of these deltas results in the delta $V_0 \rightarrow V_2$ 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.

4.6.3 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 or a B-tree.

4.7 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 [74] as it shows very good performance and low space consumption in comparison to other path-based schemes [84] and is practically used, for example, in the XML engine of Microsoft SQL Server. For the versioning, we chose the asymptotically optimal multiversion B-tree (MVBT) [11]. 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 [99]. 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., parent node retrieval) ones. 3) The query assesses a recursive property of the hierarchy which is hard to evaluate with recursive SQL (cf. experiments in Section 2.8).

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), that is, one delta between each successive version V_i and V_{i+1}, and the star scheme (Star), that is, 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 4.12(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. [6].

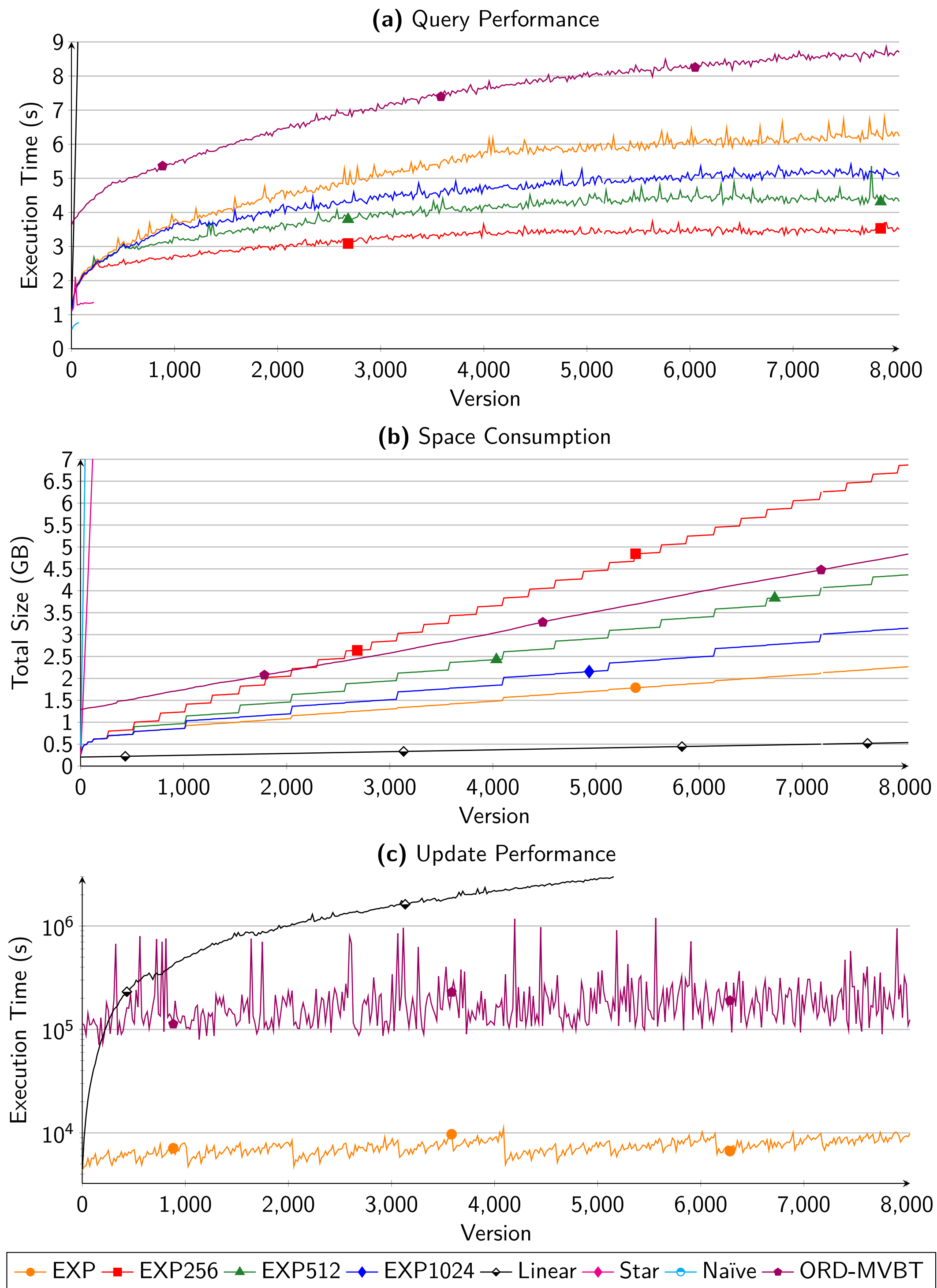


Figure 4.12: Time for executing one million queries (a), space consumption (b), and time for executing one million updates (c).

Memory Consumption. The memory consumption is shown in Figure 4.12(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 4.12). 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 compactly in a base version. 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 4.12(c). The only exponential 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. It therefore aligns nicely into the spirit of Order Indexes which show their greatest merits in dynamic scenarios featuring complex updates.

4.8 Related Work

Related work about general relational hierarchy support and indexing schemes in general has already been covered in Sections 2.2 and 3.2, respectively. Therefore, we will focus on related work about hierarchy versioning here.

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 [53], the Time-Split B-tree [62], the BT-tree [49], and the multiversion B-tree [11] 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. [29], which focus on version management. They consequently compare their approach to text-based version control systems like

SCCS and RCS. Rusu et al. [81, 82] propose and compare different delta storage techniques. Marian et al. [65] are concerned with version management in an XML Warehouse in the Xyleme project. Their concepts like the XID-map and their diff algorithm [30] are important for our contribution. They also evaluate different delta storage techniques. Rosado et al. [80] 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. [20] 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. [27, 28] 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., [101, 91]), 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 [31]) 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 [97] (no branching) and a BT-ElementList [96] (allows branching). More recently, Woss and Tsotras [99] 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 efficient complex updates (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. [66, 78] 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. [108] 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 complex updates are not supported efficiently. 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.

4.9 Conclusion

In this chapter, 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 also the only indexing scheme that yields a gapless fixed-size integer labeling for versioned trees while still facilitating complex updates. By adding DeltaNI to our potpourri of indexing schemes, we enhance our framework to versioned hierarchies.

CHAPTER 5

RWS-Diff: Flexible Change Detection in Hierarchical Data

Parts of this chapter have previously been published in [38].

When tree data changes or versions of a data item are independently modified, it is necessary to compute the difference to reconcile or display the changes. The changes are often expressed as a so-called *edit script*: a compact sequence of operations that transforms one tree into the other. Computing edit scripts has many important applications. Consider, for example, revision control systems that deal with trees like XML data warehousing [5], source code revision control [24], or HTML warehousing [25]. The goal in these applications is to compute a compact and intuitive representation of the history. Computing compact tree diffs is also crucial for various other applications, for example, data synchronization [60], genomic and proteomic data [92, 47], RNA secondary structures [107], or image analysis [17]. As outlined in the previous chapter, our hierarchy versioning approach also requires tree diffs in some scenarios.

Our goal is to compute compact edit scripts for very large trees, for example, two file systems with tens or hundreds of thousands of nodes. Furthermore, operations that lead to short and intuitive edit scripts are to be supported. In particular, not only edits on individual nodes (e.g., deletion of a node), but on whole subtrees should be considered (e.g., moving of a subtree). Again, this fits into our overall goal of enabling complex updates on hierarchies. As an example, consider a diff for synchronizing a remote file system. A locally moved subdirectory requires sending all files one-by-one to be inserted into the remote file system if moves are not

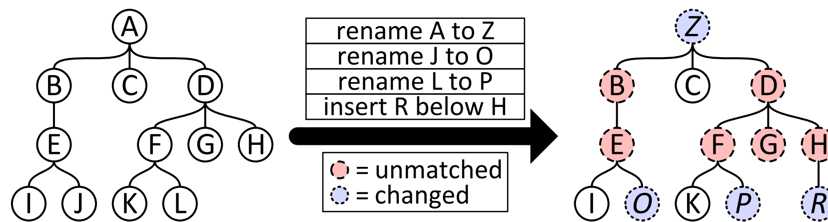


Figure 5.1: Slightly different trees make top-down and bottom-up matching fail

detected. Our approach should work for both ordered and unordered trees to be generally applicable. An example of ordered trees are HTML documents, were the order of the paragraphs matters; file systems are unordered trees.

Ideally, a tree difference algorithm computes a *minimal* edit script. Unfortunately, for unordered trees, the problem is MAX-SNP hard [106] even for a limited set of node operations; for ordered trees, exact solutions require $O(n^3)$ time in the number of nodes and thus do not scale either. Our approach is to *approximate* the minimal edit script. The result is a small (albeit not necessarily minimal) edit script that is correct, that is, it turns the first input tree into the second.

Most previous attempts to approximate the minimal edit script run in $O(n^2)$ and consequently do not scale to large trees. The few solutions that run in $O(n \log n)$ use comparably simple matching algorithms which start either at the root (top-down) or at the leaves (bottom-up). The matching is continued as long as identical nodes or subtrees are found. These approaches rely on large subtrees that can be matched exactly and fail otherwise, as illustrated in Figure 5.1: Top-down cannot match any node since the root labels differ. Bottom-up subtree matching can only match single leaf nodes (I, C, K), although many inner nodes are unchanged. The changes in the leaves alter the containing subtrees and prevent them from matching. Changes in the leaf nodes are a frequent scenario, for example, files in a file system and text values in XML.

Our solution is RWS-Diff (Random Walk Similarity Diff), a novel robust algorithm for tree differences. RWS-Diff supports both node and subtree edits. It is *robust* because it does not rely on exact subtree matching, but is also able to match similar subtrees. Similarity computations are costly and the challenge is to find similar subtrees efficiently. We present a new technique which represents each subtree by a d-dimensional feature vector using random walks. This allows us to

use well-established indexes for similarity search in d-dimensional space to find similar subtrees.

RWS-Diff is the first algorithm that runs efficiently in $O(n \log n)$ and deals well (due to subtree similarity matching) with all kinds of node edits for which top-down and bottom-up based approaches with the same asymptotic runtime fail. RWS-Diff does not rely on any application specific assumptions (like node identifiers) that simplify the matching. It is configurable to work on both ordered and unordered trees and supports a large set of edit operations. Our evaluation using synthetic and real-world data shows substantial gains in the matching quality (up to ten times smaller edit scripts on average compared to other quasi-linear methods) and robustness (more than 200 times smaller edit scripts for certain real-world trees) and confirms the scalability of RWS-Diff.

5.1 Tree Edit Scripts

For a pair of input trees A and B, a tree difference algorithm (diff) computes a sequence of edit operations (called *edit script*) that transforms A into B. Tree diffs vary in the kinds of supported operations and the underlying tree definition.

Tree Definition. Our diff algorithm works on all rooted, labeled trees with a (non-strict) order defined on the labels (i.e., the labels can be sorted and compared for equality) and a hash function that maps labels to numeric values. Our algorithm can be configured for both ordered trees (where the sibling order matters) and unordered trees. The sibling order is not related to the label order.

The labels carry application specific data. XML nodes, for example, may be labeled with element tags or text content, but also more complex labels are possible. Our flexible tree definition suites a wide range of applications such as HTML documents, XML, file systems, or RNA secondary structures, and sets us apart from many other works with restricted input trees.

Edit Operations. We allow edit operations on both *nodes* and *subtrees*. Let a and a' be two nodes in tree A; the following edit operations are allowed in an edit script:

- rename(a, l) Change the label of node a to l.
- insertLeaf(l, a, i) Insert a new leaf node with label l as a new child of node a before the i-th child of a.
- deleteLeaf(a) Remove leaf node a.

- insertSubtree(S, a, i) Insert a new subtree S before the i-th child of node a.
- deleteSubtree(a) Remove the subtree rooted in node a, that is, a and all its descendants.
- move(a, a', i) Remove the subtree rooted in a and insert it before the i-th child of a'.
- copy(a, a', i) Insert a copy of the subtree rooted in a before the i-th child of a'.

The child position i is omitted in the case of unordered trees. The root node is extended with a dummy parent node to allow all edit operations also on the root node. Subtree edit operations lead to more compact and intuitive edit scripts that can be applied fast. For example, moving a chapter of a document is faster and more expressive than deleting all sections and paragraphs and reinserting them at the target position individually. Node insertion and deletion are defined on leaf nodes; an inner node is deleted by first moving all its children (with their subtrees) to its parent.

Our tree difference algorithm is flexible as it can be configured to work with either the ordered or the unordered version of the edit operations. Furthermore, the operations for copying, inserting, and deleting subtrees can be switched off, in which case they are expressed by other operations.

A cost is assigned to each edit operation and the edit script is the better the lower the accumulated costs of the contained operations are. RWS-Diff does not imply a specific cost model. The only restriction is that the cost of each operation must be less than the cost of a sequence of other operations that can emulate it, because otherwise the operation would be useless. For example, deleting a subtree using deleteSubtree must be cheaper than deleting the same subtree node by node using deleteLeaf. The cost for subtree insertion should be a function of the subtree size, as it must encode the whole subtree S to be inserted; otherwise the best edit script would always consist of deleting tree A and inserting tree B if the trees are large enough.

Edit Mapping. An *edit mapping* maps nodes between two trees and is used to express the difference between the trees; intuitively, two nodes are mapped if they correspond to each other. We produce an edit mapping in the first step and infer the edit script from the mapping in a second step.

We define the edit mapping M between a tree A that should be transformed into tree B to be a function from the nodes of B to the nodes of A. The mapping

function is partial and neither injective nor surjective, that is, not all nodes of B or A need to be mapped and a node of A can be the image of multiple nodes of B.

5.2 Related Work

Edit scripts have been discussed from two points of view. Works on the edit distance compute the similarity between trees, where two trees are considered similar if a short edit script can transform one tree into the other. Tree diff algorithms are interested in the edit script itself.

5.2.1 Tree Edit Distance Computation

The tree edit distance is defined as the minimal cost of an edit script that transforms one tree into the other. The classical algorithm by Zhang and Shasha [107] for *ordered trees* only allows the *node* edit operations discussed in Section 5.1. For these operations, the exact distance for two trees T_1 and T_2 is computed in $\mathcal{O}(n_1 n_2 \min(d_1, l_1) \min(d_2, l_2))$ time and $O(n_1 n_2)$ space, where n_1 (n_2) is the number of nodes, l_1 (l_2) is the number of leaf nodes, and d_1 (d_2) the depth of T_1 (T_2). Thus, for trees with $\mathcal{O}(n)$ leaves and depth $\mathcal{O}(n)$, the runtime complexity is $O(n^4)$. Klein et al. [52] and Dulucq et al. [33] improve the runtime to $\mathcal{O}(n^3 \log n)$. Demaine et al. [32] present an algorithm that runs in $\mathcal{O}(n^3)$ time and show that this is the best worst case complexity that can be achieved. Unfortunately, the worst case is a frequent scenario in this algorithm, rendering it slower than the classical algorithm by Zhang and Shasha for many practical scenarios. Recently, the RTED [77] algorithm solved this problem; it maintains the optimal worst case complexity and runs as fast or faster than any of the previously proposed algorithms. For each of these algorithms the minimal edit script for the edit distance can be computed within the same complexity bound [107].

Overall, computing the *minimal* edit script requires $O(n^3)$ time and $\Theta(n^2)$ space for ordered trees, even when only node edit operations are allowed. An extension of Zhang and Shasha's algorithm with deleteSubtree and insertSubtree runs in $O(n^4)$ time [10]. With the *move* operation that we allow in our approximation, the edit distance problem is NP-complete even for the case of flat strings [88].

Approximations of the tree edit distance that run more efficiently have been proposed. Guha et al. [44] propose an upper bound for the tree edit distance by

computing the *string* edit distance between the pre-order (or post-order) sequences of the tree node labels in $\mathcal{O}(n^2)$ time. An edit script can be computed using this method.

With p,q-grams [9] Augsten et al. propose a concept of "q-grams for trees". The grams are constructed using the ancestor relationship (configurable by p) and the sibling relationship (q). The method decomposes the input trees into small, besom-shaped subtrees with depth p and q leaves. Each of these small subtrees is then serialized to a string and hashed. A list of these hashes represents the data in the tree and its hierarchical relationships. The algorithm calculates the p,q-grams in $\mathcal{O}(n)$ time and space. Our approach also makes use of p,q-grams for finding similar subtrees but adds a dimensional reduction step to speed up similarity search. Similar to p,q-grams, the binary branch technique [102] splits trees into small subtrees, but binary branches keep less structure information than p,q-grams [9]. Neither p,q-grams nor binary branches compute edit scripts.

For unordered trees, finding the exact tree edit distance is MAX SNP-hard [106] since the matching algorithm can not rely on the sibling order. Zhang et al. [89] propose an exact, enumeration-based algorithm for unordered trees which runs in $\mathcal{O}(n^3 16^n)$ and a heuristic solution based on searching in the enumeration space which runs in $\mathcal{O}(n^2)$. By sorting siblings lexicographically by label the concept of p,q-grams can be adapted to unordered trees. The p,q-gram approximation runs efficiently in $\mathcal{O}(n \log n)$ and is shown to work well in practice [8]. We use this technique for supporting random walk similarity on unordered trees.

5.2.2 Computing Diffs between Trees

Numerous approaches for computing approximately cost-minimal edit scripts have been proposed, but most of them either suffer from a prohibitive runtime of at least $\mathcal{O}(n^2)$, are restricted to very specific types of data, or do not show a robust behaviour.

Chawathe et al. propose LaDiff [26], which imposes restrictions on the hierarchical order between labels: An example are LaTeX documents, where a subsection is always within a section. This bottom-up algorithm uses a heuristic optimized for text. As a bottom-up tree edit distance, it is sensitive to changes in the leaf nodes. LaDiff combined with another method [24] is implemented in the tree-diff tool DiffXML [69]. Le et al. [54] remove the hierarchical order restriction from LaDiff. The family of algorithms based on LaDiff runs in $O(ne)$ time, where e is

the size of the edit script. In the worst case, when the trees are very different, the runtime is $O(n^2)$ and the approach does not scale. In our experiments we compare to DiffXML as a representative of these algorithms.

In [59, 60] a three-way merging algorithm for XML is described, which includes an algorithm for calculating diffs between XML documents (3DM). It works in a bottom-up fashion, mapping trees using their content. It also uses the neighborhood of tree nodes to produce mappings, for example, when the left and right siblings of a node are mapped, a mapping for the node in between is inferred. The algorithm has worst-case complexity $\mathcal{O}(n^2)$ and runs in $\mathcal{O}(n \log n)$ if the changes between trees are small.

The MH-DIFF algorithm [23] allows the operations insert, rename, delete, move, and copy. Here, insert and delete work on single inner or leaf nodes. In the first step, all possible mappings between the nodes of two trees are considered and mappings that can only increase the cost are pruned in the second step. The problem is then solved as a bipartite weighted matching problem by assigning an approximate cost to each mapping between a pair of node. The overall runtime complexity is $\mathcal{O}(n^2 \log n)$.

Wang et al. [98] only allow insertLeaf and removeLeaf. Furthermore, only nodes with the same path to the root are mapped. The algorithm produces large edit scripts if these assumptions are not met. When the maximum number of children of all nodes is assumed to be a constant (i.e., independent of the tree size), $\mathcal{O}(n^2)$ runtime complexity is achieved. [92] improves the average runtime of this algorithm for hierarchical biological data without altering the worst case complexity.

The KF-Diff+ algorithm [100] is specific to a particular kind of XML documents, in which each node has a key that is unique between all siblings. In this case, a diff which allows *move* operations only between nodes with the same parent can be computed in $\mathcal{O}(n)$ time.

The only algorithm without strong assumptions that runs in less than quadratic time is XyDiff [64]. XyDiff uses tree hashes that are invariant to the sibling order [98] to efficiently find and map moved subtrees. In the next step, nodes in the vicinity of mapped subtrees are mapped. The overall algorithm runs in $\mathcal{O}(n \log n)$ and produces good results if large unchanged subtrees are present. In addition to the operations supported by XyDiff, our algorithm also supports subtree deletion. Subtree deletion is useful to remove surplus data from one of the trees, for example, the citation elements present in some DBLP entries that otherwise dominate the

edit distance. We experimentally compare our algorithm to XyDiff and show that (even without subtree deletion) our algorithm produces significantly smaller edit scripts with a similar runtime.

5.3 The RWS-Diff Algorithm

A good edit mapping is one that maps as many nodes as possible and maps nodes which are very similar to each other. The better the mapping, the smaller the generated edit script will be. A perfect mapping would be one that produces a cost-minimal edit script. However, such a mapping is extremely hard to compute (MAX SNP-hard for unordered trees even if only node operations are allowed, cf. Section 5.2). Thus, RWS-Diff is an approximate method which tries to find a good–but not always perfect–mapping. However, our focus is on finding a better mapping than previous approximate approaches by using an elaborated similarity measure to find non-obvious mappings.

This section introduces RWS-Diff which constructs the approximate cost-minimal edit mapping and then creates an edit script from it. Our method can roughly be separated into five steps:

1. A simple matching step which tries to find obvious common structures in both versions of the tree. The nodes mapped in this step do not have to be considered in subsequent matching steps and thus significantly improve their speed.
2. Construction of feature vectors for unmapped subtrees of both trees, that is, small fixed-length vectors which are similar if subtrees are similar. The squared euclidean distance between the vectors constitutes our random walk similarity measure.
3. Creation of appropriate index structures for nearest neighbors queries among the feature vectors.
4. Mapping of previously unmapped subtrees by looking up possible candidates using nearest neighbors queries.
5. Generation of the edit script from the edit mapping.

5.3.1 Finding Simple Mappings

In the first step, we try to match large parts of the trees rapidly. The goal is not to find all possible mappings but to find only the obvious ones which are easy to

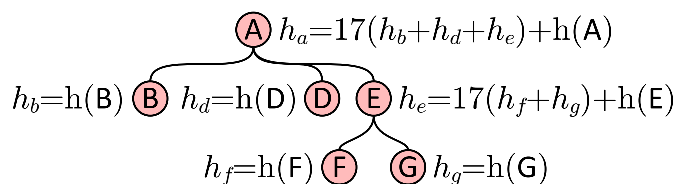


Figure 5.2: Sibling order invariant subtree hashes

compute. We use methods and concepts already described in the literature and successfully applied. These are top-down matching [87, 100] and matching using subtree hashes [64].

Top-down Matching. The top-down matching starts at the roots of the trees to be compared and maps nodes with the same label to each other. If a node is mapped and is not a leaf node, the same matching method is recursively applied to its children. If more than one sibling has the same label, we do not map it in this step, since we might map the wrong pairs of nodes. When using this method, if a node's label is changed, the whole subtree rooted in this node is not mapped anymore.

Hash Matching. Hash matching between trees A and B is performed bottom up by calculating a hash value of each unmapped subtree in A and inserting it into a hash table. Then, the unmapped subtrees of B are hashed as well and the hash table is probed to find equivalent subtrees in A. For unordered trees, we have to use a position-independent hash function. Simply adding the hash of the node label and the hash values of all children multiplied with a prime number is position-independent since addition is a commutative operator (cf., Figure 5.2). For ordered trees it suffices to multiply the hash of each child with a prime number that is different for each child position.

5.3.2 Random Walk Similarity Matching

Using hash matching and top-down matching, we can usually map a large portion of the tree nodes if there is a moderate number of differences (which is usually the case). The top-down matching finds all paths from the root that have not changed. The hash matching finds smaller subtrees which have not changed. However, the quality of the resulting mapping is usually not sufficient because these simple matching methods are not robust at all. For example, renaming the root totally

disables top-down matching and a single renaming in a node or insertion of a node disables hash matching for all subtrees that contain this node. Since even such "trivial", non-structural edit operations disable the simple matching methods, we need a method for finding trees that are similar but not equal.

Our core contribution finds trees that are not necessarily equal but similar and thus are missed by the simple matching approaches. The idea is to represent each subtree by a d-dimensional feature vector (with fixed d) that constitutes a random walk in d-dimensional space. The random walks are generated in a way that ensures that their squared euclidean distance, which we call *random walk distance (RWD)* is approximately proportional to the edit distance of the corresponding trees. How these random walks are generated in detail will be discussed in Section 5.4.

We find similar subtrees in trees A and B by generating all feature vectors for subtrees in A and inserting these vectors into an index structure for d-dimensional nearest neighbors queries. If the copy operation is allowed, we insert all subtrees, because even already mapped subtrees could be mapped again for a copy. Otherwise, we only insert subtrees with an unmapped root. Then, we generate feature vectors for all unmapped subtrees in B. Next, we iterate over tree B in pre-order and for each unmapped subtree b, we use its feature vector to probe into the index structure to find the ℓ (with fixed ℓ) nearest neighbors which are candidates for being similar. We retrieve $\ell > 1$ neighbors, because a low RWD does not always (but often) imply a similarity in the subtrees (i.e., false positives are possible). Therefore, the ℓ nearest neighbors in the feature vector space are merely used as mapping candidates and we use the one with the least edit distance or none if all are false positives (which should happen very infrequently due to the stochastic properties of the RWD). For the similarity comparison of the ℓ mapping candidates, we use an iterative deepening top-down matching that stops after a fixed number of compared nodes and is thus in $O(1)$. Although the premature stopping might reduce approximation quality, it is important to meet the desired log-linear runtime bounds. If the copy operation is not allowed we skip candidates which have an already mapped root. Once the best candidate subtree a for subtree b is determined, we map the roots of a and b and perform an ordinary top-down matching starting from a and b to map cheaply as many descendants as possible. Afterwards, we continue the pre-order iteration over B to map remaining subtrees.

We can use standard index structures to efficiently find nearest neighbors in d-dimensional space. We focus on prominent indexing schemes which are *k-d trees*,

```
 0: function GENERATEEDITSCRIPT(A, B, M)
 1:     for each nodes b of B in pre-order do
 2:         if ∄a.(b, a) ∈ M then
 3:             Emit insertLeaf(label(b), M(parent(b)), pos(b))
 4:         else
 5:             a ← M(b)
 6:             if ∃b′.(b′, a) ∈ M ∧ pre(b′) < pre(b) then
 7:                 Emit copy(a, M(parent(b)), pos(b))
 8:             else if M(parent(b)) ≠ parent(a) then
 9:                 Emit move(a, M(parent(b)), pos(b))
10:             if label(b) ≠ label(a) then
11:                 Emit rename(a, label(b))
12:     for each nodes a of A in post-order do
13:         if ∄b.(b, a) ∈ M then
14:             Emit deleteLeaf(a)
```

Figure 5.3: Algorithm for generating edit scripts

k-means locality-sensitive hashing (KLSH), and *hierarchical k-means (HKM)*. These methods have been shown to be useful in practice [63, 73, 76]. However, note that any scheme for finding nearest neighbors in d-dimensional space can be used. The k-d tree is an established multidimensional index structure which repeatedly separates the space by hyperplanes. The exact nearest neighbors algorithm on the k-d tree is quite costly, so we use the approximate *best bin first (BBF)* [12] algorithm. Hierarchical k-means clustering has been successfully used to cluster high dimensional data [73]. It works by recursively finding k centroids for the data point clusters and then arranging those clusters into a tree structure. K-means locality-sensitive hashing [76] uses k-means clustering to convert space coordinates into locality-sensitive hashes.

5.3.3 Edit Script Generation

An edit script is a sequence of edit operations that transforms tree A into tree B. The algorithm shown in Figure 5.3 produces the edit script from an edit mapping (which maps nodes of B to nodes of A). Figure 5.4 shows an example for each kind of edit operation produced by the algorithm. The algorithm traverses all nodes of B in pre-order, that is, parents are visited before their children. If a node b in B has no mapping (b, a) in M, b is inserted (Lines 2-3, node H in figure). If a mapping (b, a) exists, we check whether we already have visited a node b' which

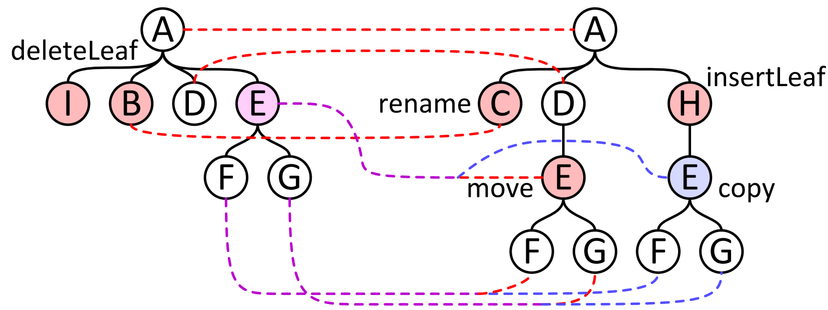


Figure 5.4: Edit mapping with implied edit operations

maps to a as well. The node b' is already visited if its pre-order rank pre(b') is smaller than the one of b (Lines 4-6). If such a node exists, we already have "used" subtree a and must thus copy it (Line 7, right node E in figure); otherwise, we check if the parents of the mapped nodes differ. If they do, we move node a to its new parent (Lines 8-9, left node E in figure). In addition to updating the node position, we must also rename node a if the labels of the mapped nodes differ (Lines 10-11, node $B \to C$ in figure). After the pre-order iteration over B, we perform a post-order iteration over A and delete all nodes that are not mapped (Lines 12-14, node I in figure).

Whenever an edit operation is emitted, it is applied to tree A and subsequent edit operations are defined on the new version of A. In addition, the mapping M is updated after each insertLeaf and copy operation with mappings to the newly added node(s). After executing the algorithm, the sibling order is adjusted for ordered trees, which is separately discussed below.

Line 8 of the algorithm requires that M maps the parent of the current node b to some node of A. This is guaranteed by traversing B in pre-order and updating the mapping after each insert operation. In the second phase of the algorithm (Lines 12-14), unmapped nodes in A are removed using the deleteLeaf operation, which requires the nodes to be leaves at the time of being removed. This holds since (a) the mapped child a of an unmapped node in A satisfies the condition in Line 8, that is, the subtree rooted in a is moved to a mapped parent in the first phase of the algorithm; (b) the nodes of A are traversed in post-order, thus unmapped children are removed before their unmapped parents.

The algorithm does not produce subtree insertions and deletions. By generating the insertLeaf and deleteLeaf operations in pre-order and post-order, respectively,

we ensure that all inserts and deletes that belong to the same subtree are adjacent in the edit script. We merge sequences of leaf insertions and deletions into subtree insertions and deletions in a simple postprocessing step. By omitting this step, we can switch off subtree insertion and deletion. If subtree copy is switched off, the mapping is injective, so the condition in Line 6 is never true.

After executing the algorithm, A is identical to B except for the sibling order, and all nodes of A and B are mapped. We use the approach of XyDiff [64] to fix the sibling order. The c children of each node in A are numbered with the sibling positions of the respective (mapped) nodes in B, that is, each child in A gets assigned a position between 1 and c. We compute the longest increasing sub-sequence X of the position numbers in $\mathcal{O}(c \log c)$ time [40]; all nodes that are not in X are moved to the right position by emitting moveSubtree operations.

5.3.4 Complexity of RWS-Diff

RWS-Diff must have an $\mathcal{O}(n \log n)$ worst case runtime complexity in order to yield a scalable solution. The simple matching methods that are also applied in existing $\mathcal{O}(n \log n)$ methods obviously fall into this bound. The generation of the random walk feature vectors for all subtrees in a tree is in $\mathcal{O}(n)$ (cf. next Section). Since there may be $O(n)$ subtrees in both trees that must be mapped by the RWS, mapping one subtree may only cost $O(\log n)$. A nearest neighbors lookup is usually in $O(\log n)$ in the index structures. We adjust the index structures to yield even worst case $O(\log n)$ behaviour by simply decreasing the approximation quality in pathological cases. For example, HKM has its height limited to $O(\log n)$ and if there are more than ℓ candidates in the final Voronoi cell, only the ℓ first are considered. Although the approximation becomes worse in some cases, our evaluation shows that the overall quality is still good. For finding the best candidate between the ℓ candidates, we use the constantly bounded iterative deepening top-down matching which is in $O(1)$, so a single RWS mapping stays in $O(\log n)$. An insertion or lookup in the mapping M is in $O(1)$ since dense integers can be assigned to each node in tree B and M can be implemented as an array indexed by these integers. Finally, the edit script generation loops only twice over both trees and is thus in $O(n)$, so we meet the desired overall complexity bound of $\mathcal{O}(n \log n)$.

5.4 Random Walk Similarity

To find similar subtrees rapidly, we reduce the information content of each subtree to a fixed-size d-dimensional feature vector and use indexed d-dimensional nearest neighbors queries. To obtain the feature vector, we first serialize each subtree into a bag of p,q-gram hashes. The more similar two trees are, the more hashes are equal. Each of the hashes becomes a step in a d-dimensional random walk and the final feature vector is the endpoint of the walk. Consequently, the more similar two trees are, the more steps in their random walks are equal and thus the distance of the random walk end points is the smaller the more similar two trees are. We prove certain stochastic properties of the proposed random walk to show that it is indeed a valid approximate similarity measure.

5.4.1 Grams for Trees

Grams (also shingles or tokens) are tree summaries that represent a tree by a set of small excerpts. Using such grams, the problem of finding similar subtrees is reduced to the problem of finding bags of grams with large intersections. This approach has been widely applied to strings before and has shown to be useful also for trees.

We use *p,q-grams*, which are besom-shaped subtrees consisting of q leaf nodes (called *base*) and a chain of p non-leaf nodes (called stem). In the original tree, the base nodes are siblings and the stem nodes their p closest ancestors. p,q-grams capture both ancestor and sibling relationships and can be made invariant to small order changes. In addition, they have already been successfully applied to tree similarity computations in various scenarios [9, 8].

The p,q-grams for all subtrees of a tree of size n can be generated in $\mathcal{O}(n)$ time. The p,q-gram construction is illustrated in Figure 5.5 for an example tree ($p = q = 2$). In the first step, the tree is sorted lexicographically by labels (the sort order of identical labels is irrelevant for the p,q-gram construction). Next, for each node a in the tree, a window of size $w \geq q$ is slidden over the children of the node and p,q-grams are produced. The bases are formed by the first node of each window and any sub-sequences of the remaining nodes in the window. If a node does not have enough ancestors or enough children, dummy nodes (labeled with an asterisk in the figure) are used to produce the p,q-grams. In Figure 5.5, the sorting changes the order of the children of the node with label "A". The window size is

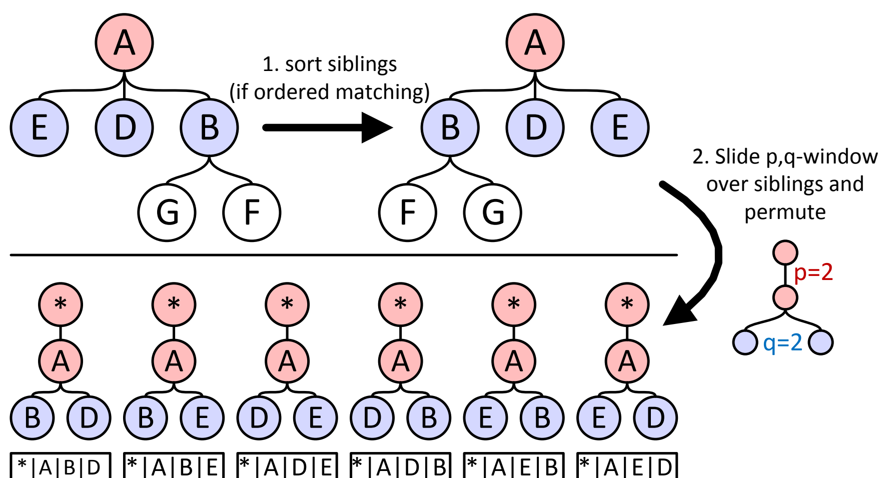


Figure 5.5: Partial construction of p,q-grams

$w = 3$ and six bases are formed for three window positions (the window is wrapped around at the right border).

Invariance to order changes is obtained through sorting siblings. Augsten et al. show in [8] that this way the permutation of a constant amount of siblings changes only a constant amount of p,q-grams. The construction of the base using a window makes the p,q-grams robust to modifications that change the sort order of children, called "children error" in [8], while still capturing sibling relationships. The "stem" captures ancestor relationships in the p,q-grams. If sibling permutations should not be allowed (ordered trees), the trees are not sorted and windows of size q are used.

The p,q-grams are finally serialized into arrays of size $p + q$, which is straightforward due to the fixed shape of the p,q-grams. The bottom of Figure 5.5 shows the serializations of the respective p,q-grams.

The similarity of two trees can now be expressed over their bags of p,q-grams. Let b_A and b_B be the bags of grams of tree A and B, respectively, then the symmetric bag difference $D(A, B)$ is defined as $|S_A \uplus S_B| - 2|b_A \cap b_B|$. This difference directly reflects the number of elements we have to remove from A and add to B if we want to transform A to B and as such approximately reflects the required edit operations. It is a distance measure, that is, the distance $D(A, B)$ between identical sets is zero while the distance between entirely different sets is $|A| + |B|$.

5.4.2 Random Walk Distance

Even though the comparison of trees is now easier, the actual size of the tree representations has gone up. If there are a and b unmapped subtrees in tree A and B, respectively, we have to compute $a \times b$ bag differences to find the best matches. Each of these computations has linear runtime in the bag sizes, which would clearly violate the $\mathcal{O}(n \log n)$ runtime bound. To speed up the similarity search, we do not explicitly calculate the bag difference between any two bags. Instead, we compress each bag of grams to a fixed-size d-dimensional vector and then use a nearest neighbors search in the d-dimensional space to find mapping candidates.

The d-dimensional feature vector for a tree A which has a bag of grams b_A is generated as follows: First, compute a hash value h_g for each p,q-gram g in the bag b_A. Then, use h_g to generate a random point v_g on the d-dimensional unit sphere (e.g., use h_g as seed for a random number generator that generates the vector components). To get the final feature vector v_A for A, add up all the vectors v_g. To approximate the symmetric bag difference, we use the d-dimensional squared euclidean distance. The vector v_A constitutes the end point of a d-dimensional random walk with $|b_A|$ steps of length one. Therefore, we call the resulting distance *random walk distance (RWD)*:

$$D(A, B) \approx \mathsf{RWD}(A, B) = ||v_A - v_B||^2 = v_A \cdot v_B$$

By using the random walk distance, we reduce the problem of finding similar subtrees to the problem of finding points which are close in euclidean space. It is intuitive that this is a valid similarity measure: The more grams differ between the bags b_A and b_B, the more steps from which v_A and v_B are assembled differ.

All grams that are in both bags b_A and b_B yield the exact same steps in the random walk. Consequently, these steps do not alter the distance at all. The number of remaining grams is $x = |b_A \backslash b_B|$ and $y = |b_B \backslash b_A|$ which constitute the two random walks whose squared euclidean distance is the RWD. The distance between the end points of two random walks with x and y steps is equal to the distance between the origin and an end point of a random walk with $z = D(A, B) = x + y$ steps. Figure 5.6 shows the transformation of two bags of grams b_A and b_B into corresponding two-dimensional random walks v_A and v_B. The numbers below the

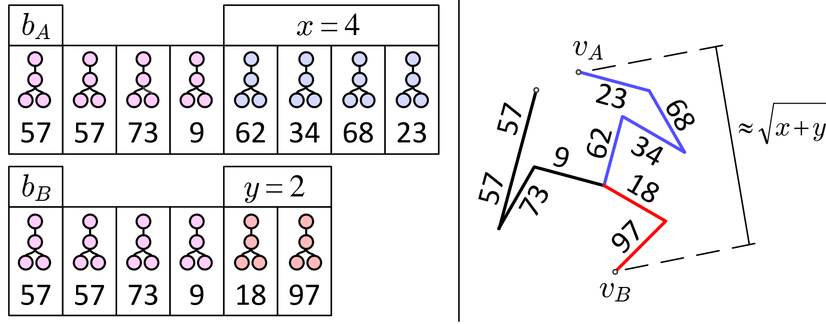


Figure 5.6: Transforming bags of grams into corresponding two-dimensional random walks

grams show their hash values. In this example, the trees have 4 common grams and $2 + 4 = 6$ different grams. Thus, the expected euclidean distance is $\sqrt{6}$.

Of course, the RWD is only an approximation since random walks with totally different steps might end up at points that are close to each other. To argue that the measure is useful indeed, we examine its stochastic properties: Let $v = v_1 + \ldots + v_z$ be the endpoint of a random walk starting from the origin and taking z steps in d-dimensional space and let RWD_z^d be the squared euclidean distance between the origin and v. For each step v_i, we have $E[v_i] = \vec{0}$, $\|v_i\|^2 = 1$ with all v_i being independent random variables. Then we have $\text{RWD}_z^d = \|v\|^2 = v \cdot v$. By expansion of the dot product we obtain $\text{RWD}_z^d = \sum_{i=1}^{z}(v_i)^2 + \sum_{k \neq \ell} v_k \cdot v_\ell$. With $E[(v_i)^2] = 1$ and $E[v_k \cdot v_\ell] = E[v_k] \cdot E[v_\ell] = 0$ for every $k \neq \ell$, we have

$$E[\text{RWD}_z^d] = z = D(A, B) \tag{5.1}$$

Thus, the RWD is indeed an approximation of the symmetrical bag distance regardless of the number of dimensions. The squared RWD is as follows:

$$(\text{RWD}_z^d)^2 = \underbrace{z^2}_{a} + \underbrace{2z \sum_{k \neq \ell} v_k \cdot v_\ell}_{b} + \underbrace{(\sum_{k \neq \ell} v_k \cdot v_\ell)^2}_{c}$$

and thus for the variance:

$$\text{Var}[\text{RWD}_z^d] = \underbrace{E[a]}_{=z^2} + \underbrace{E[b]}_{=0} + E[c] - \underbrace{E[\text{RWD}_z^d]^2}_{=z^2} = E[c]$$

and for $E[c]$:

$$E[c] = \sum_{k \neq \ell} \sum_{i \neq j} E[(v_k \cdot v_\ell)(v_i \cdot v_j)]$$

All terms with $\{k, \ell\} \neq \{i, j\}$ in this summation are zero. The remaining $2z(z-1)$ terms have a mean of $m = E[(v_i \cdot v_j)^2]$. Let the ℓ-th component of vector v_i be v_i^ℓ. Due to independence of v_i and v_j and because v_i and v_j are equally distributed, we can expand the dot product and simplify to $m = \sum_{\ell=1}^{d} E[(v_i^\ell)^2]^2$. Since all d components v_i^ℓ are identically distributed with $\|v_i\|^2 = 1$, we have $E[(v_i^j)^2] = \frac{1}{d}$ by symmetry and thus $m = \frac{1}{d}$. Consequently

$$\mathrm{Var}[\mathrm{RWD}_z^d] = 2z(z-1)m = \frac{2z(z-1)}{d} \tag{5.2}$$

The first consequence of Equation 5.2 is that the number of dimensions reduces the variance and thus makes the approximation more precise. For an infinite number of dimensions, the RWD would even be exactly the symmetric bag difference. This implies that a high number of dimensions is desirable. However, a high number of dimensions makes computations more costly and renders the index structures that we use for the nearest neighbors search ineffective due to the curse of dimensionality. Hence, too many dimensions are prohibitive as well. We obtained best results with $10 \leq d \leq 20$. The second consequence of Equation 5.2 is that the variance is proportional to $z(z-1)$. Thus, the larger the distance, the less reliable the approximation is. In contrast, the RWD is a very reliable approximation for small distances. This fact is extremely beneficial for our application: As we want to execute a nearest neighbors search, we are especially interested in points with a small distance. For these points, the RWD is very precise, so there are no false negatives. As mentioned, the problem of false positives is mitigated by choosing the best of ℓ nearest neighbors. In conclusion, the stochastic properties of the d-dimensional random walk make the RWD an excellent approximate distance measure for our purposes. Note that while the RWD is defined as the squared euclidean distance since this is a direct approximation for the bag distance, the index structures use the usual euclidean distance. As we are not interested in the value of the RWD itself but only in the nearest neighbors, this is not an issue.

5.4.3 Weighting Grams

Until now, we assumed that each step in the random walk has a length of one. However, we can also weight the grams and multiply the step length by that weight to give certain grams more or less significance. A general assumption we can make is that having less frequent grams in common is more significant than having frequent grams in common. If we look, for example, at HTML documents, two subtrees having a *br* element in common are not that rare, while having a long text node in common which appears infrequently in the document is a strong indication of a correct mapping. Therefore, we use the *inverse gram frequency*, that is, the reciprocal value of the number of times a gram occurs in both trees as weight. Although this is a quite simple heuristic, it improved the edit script quality noticeably in our tests. Of course, more elaborate heuristics could be used as for example proposed in [83]. What we want to emphasize here is not the concrete choice of heuristic, but the fact that the random walk similarity can be tuned easily by such a heuristic.

5.5 Evaluation

Index Structure Comparison. In order to find out which of the feature vector indexes presented in Section 5.3.2 is best suited we compare them with test data. We generate that data by taking 10,000/100,000 random subtrees with 10 to 100 nodes from various freely available XML files [94]. For each subtree S_i, we generate a feature vector f_i with $d = 10$ dimensions, modify the tree randomly, and calculate another feature vector f_i'. We then insert the feature vectors f_i into an index. Afterwards, we issue l-nearest neighbors queries with $l = 10$ for 1000 randomly chosen f_i's. We measure *precision*, that is, the fraction of returned nodes which are correct l-nearest neighbors, *average distance* to query point, *setup time* (i.e., time for generating the index) and *runtime* for the 1000 queries. All figures are averaged over 20 runs on identical hardware (Core i5 M460).

We assess exact approaches and approximate approaches. The exact approaches are a *linear scan* (comparison to all points) and a *k-d tree* with *exact* querying. The approximate approaches are *KLSH*, *HKM*, and a *k-d tree with best-bin-first (BBF)* querying.

Method	Precision	AVG dist	Runtime	Setup time
Linear scan	1	3.45	625 ms	0 ms
Exact k-d tree	1	3.45	1197 ms	8 ms
BBF k-d tree	0.46	3.85	54 ms	8 ms
KLSH	0.74	3.60	30 ms	131 ms
HKM	0.54	3.75	13 ms	261 ms

Table 5.1: Comparison of indexes with 10,000 points

Method	Precision	AVG dist	Runtime	Setup time
Linear scan	1	3.02	6454 ms	0 ms
Exact k-d tree	1	3.02	15041 ms	160 ms
BBF k-d tree	0.33	3.55	88 ms	160 ms
KLSH	0.76	3.15	470 ms	1988 ms
HKM	0.40	3.39	21 ms	3461 ms

Table 5.2: Comparison of indexes with 100,000 points

Table 5.1 shows the result for 10,000 subtrees and Table 5.2 for 100,000 subtrees. Of course, the exact methods have a precision of 1. They also show that the "perfect" average distance, that is, the distance of the real 10-nearest neighbors is 3.44 and 3.02 (for 10,000 nodes and 100,000 nodes, respectively). The runtime of the exact methods is prohibitively long even for a small dataset consisting of only 10000 points. The approximate methods, in contrast, show good results. Although the precision is not too good (46%–74% and 33%–76%), the average distance of 3.60–3.85 and 3.15–3.55 shows that the wrongly selected nodes are still close to the perfect average distance and thus are still good mapping candidates (in comparison, the distance of randomly selected points was 8.51 and 8.63). In conclusion, the approximate indexing methods, especially HKM and KLSH, are well suited for RWS-Diff. HKM is very fast even for larger datasets while KLSH has very precise results close to the exact solution. Since HKM is much faster than KLSH while its average distance is not much worse, we use HKM for the further experiments.

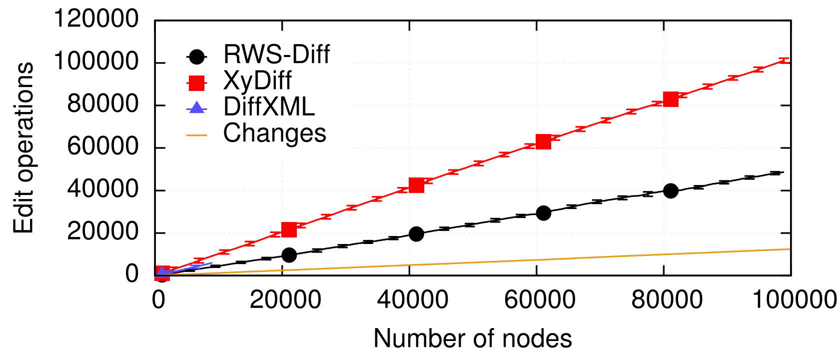
Comparison to Other Methods. To evaluate the quality of our method we compare the results with other proposed solutions. Here, the problem is that most methods have only a limited set of operations or they work only on unordered or ordered data. Even when they are comparable they often suffer from excessive

runtime or enormously large edit scripts. Other comparative studies have found these shortcomings as well [79, 47]. Both studies have also found that XyDiff [64] is the only serious contender. In addition to XyDiff, we measure DiffXML [69] as an example for a widely used open source tool for XML change detection [70] with a worst-case complexity of $\mathcal{O}(n^2)$.

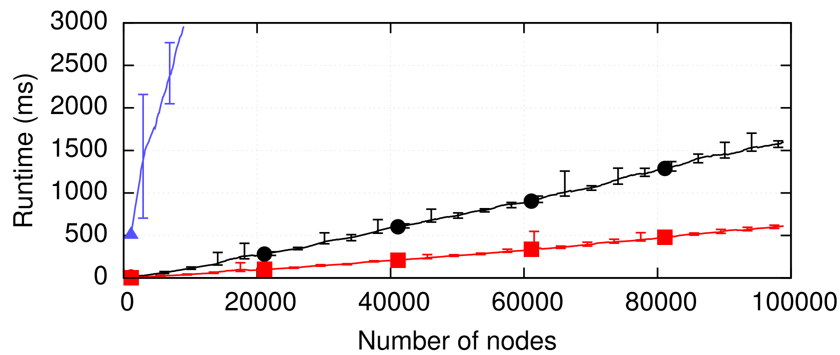
To make the comparison fair we switch off insertSubtree, deleteSubtree, and copy for our method as XyDiff does not support them. DiffXML uses deleteSubtree, but we grant that advantage. We use a uniform cost model, so the cost of the resulting edit script is equal to its length. Both contenders work in ordered mode. Since this can introduce additional work and edit operations, we use our method in ordered mode as well. The measured time does not include reading and parsing the XML/HTML trees or writing the edit script to a file.

Synthetic Changes. To show the runtime and quality for different tree sizes and change patterns, we first measure the results for synthetically changed trees. The trees are generated by extracting an increasing amount of nodes from the real-world dataset *nasa1* [94], which contains astronomical XML data, and modifying the extracted tree. The size of the extracted tree ranges from 100 to 100,000 nodes, but we stop DiffXML early after 10,000 nodes because of its tremendous runtime. The modification consists of (a) either random renames of one child of every node (one change per parent) or (b) of 10 random inserts, deletes, renames, or moves within the tree. The former change pattern represents a scenario where many small changes are introduced across the whole tree. Since one child of every node is modified simple methods might fail in this scenario. The latter change pattern of 10 random changes represents a scenario where only a comparably small part of the tree is changed. Of course, a very short edit script is anticipated in this case. Finally (c), we also measure an increasing amount of changes on a tree of constant size (20,000 nodes). This change pattern shows how the methods behave for a growing number of changes.

Figures 5.7—5.9 show the resulting number of edit operations (top) and the runtime (bottom) for the three different change patterns on the *nasa1* dataset. The data points are smoothed by calculating the moving average of 20 points. The error bars depict the minima and maxima smoothed away. For the two change patterns with a growing number of nodes in Figure 5.7 and Figure 5.8 the horizontal axis shows the tree size, while for the change pattern with a growing number of changes



(a) *Edit script size*



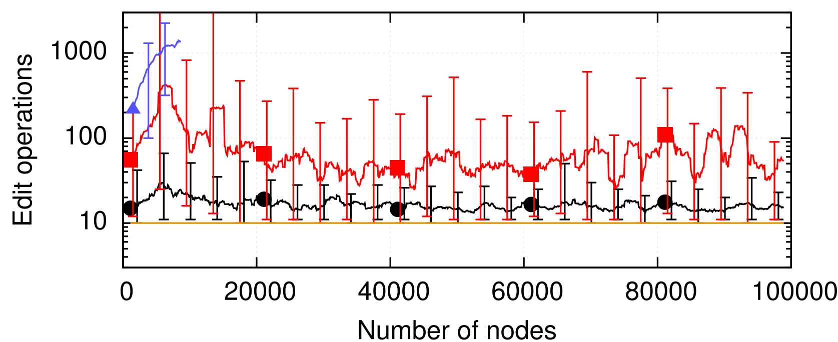
(b) *Runtime*

Figure 5.7: One change per parent on the nasa1 data set.

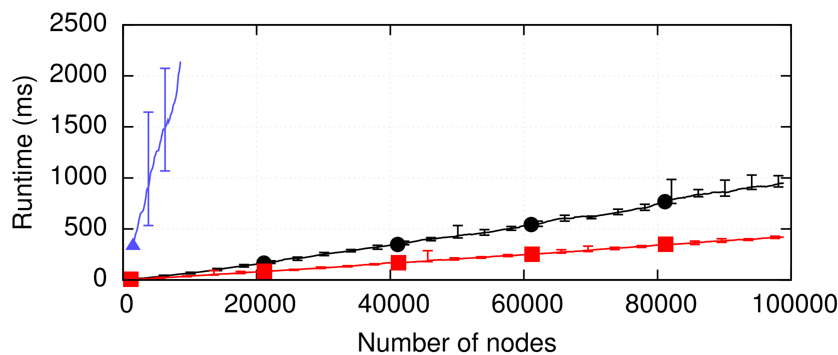
in Figure 5.9 the horizontal axis shows the number of changes. Note the logarithmic vertical axis for the number of edit operations in Figure 5.8(a).

The runtime plots at the bottom of the figures show that RWS-Diff is around two to three times slower than a simple matching approach like XyDiff. The runtime of RWS-Diff always stays in the same order of magnitude as XyDiff and therefore qualifies for the same application scenarios. The runtime of both XyDiff and our method is almost linear which backs up the claimed $\mathcal{O}(n \log n)$ runtime bound. As anticipated, the $\mathcal{O}(n^2)$ runtime of DiffXML is infeasible for larger scenarios.

The plots for the number of edit operations at the top of the figures depict the quality gain of the similarity-based matching: For the case with one change per parent (Figure 5.7), the edit script of RWS-Diff is only around half as long as the one of XyDiff. Surprisingly, it is also slightly smaller than the more complex approach of DiffXML. Especially the scenario with only 10 changes (Figure 5.8) shows the quality and robustness gains of our solution: XyDiff emits 74.5 changes on average



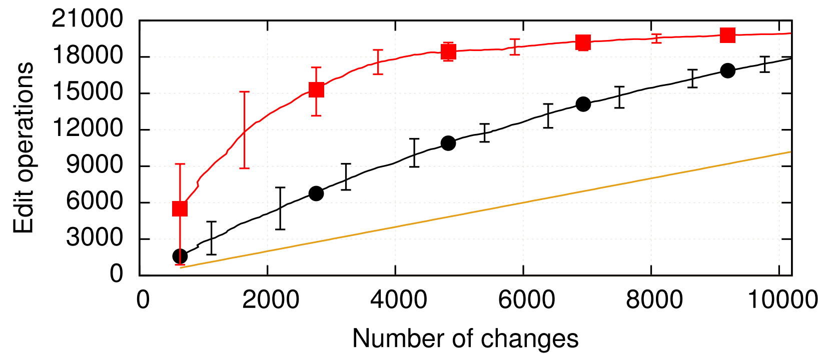
(a) *Edit script size*



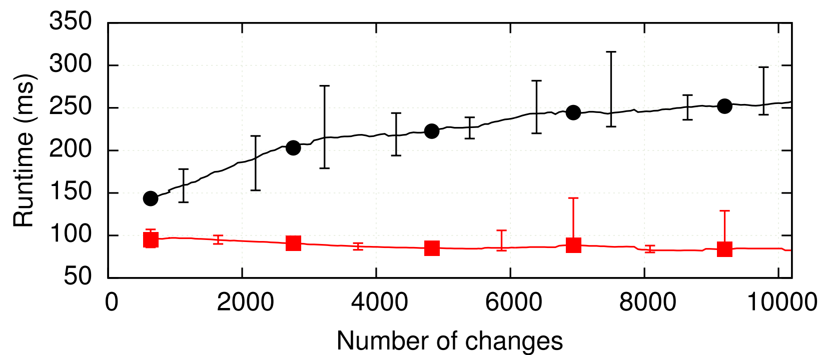
(b) *Runtime*

Figure 5.8: 10 changes on the nasa1 data set.

while RWS-Diff emits only 16.6 on average which is only about 50% more than the exact solution and around 4.5 times less than XyDiff. In addition, our method is very robust as it never emits more than 68 operations, that is, 6.8 times more than the exact solution. In comparison, XyDiff sometimes yields comparably good results but often creates edit scripts with 100–1000 edit operations. Its largest edit script even consists of 3255 operations which is more than 300 times longer than the exact solution. Such an edit script is almost useless and demonstrates the huge robustness problem of simple matching approaches. DiffXML is even much worse than XyDiff for these few changes: It generated the astronomical amount of 850.8 changes on average and a peak number of 2477 changes. Consequently, the experiment shows that especially for common scenarios with few changes per version, similarity-based matching can lower the edit script size and increase robustness significantly—not only in comparison to simple matching methods but also in comparison to more expensive methods like DiffXML.



(a) *Edit script size*



(b) *Runtime*

Figure 5.9: Growing number of changes on the nasa1 data set.

When examining a growing number of changes in Figure 5.9, our method always shows a considerable edit script quality gain. For a very large number of changes (in this case 8000+ changes for 20000 nodes), the edit script quality of both methods becomes more similar. This is due to the large amount of changes that alter the subtrees so much that they are no longer similar (i.e., do not share any p,q-grams anymore) and thus also render similarity-based approaches like ours ineffective. However, using edit scripts for such a large number of changes (almost half the tree size!) is very likely to be inferior to saving all versions explicitly anyway. Note that we did not measure DiffXML here as the tree consisted of too many nodes to yield a reasonable runtime.

Website Data. We inspected the two news websites *www.bbc.co.uk/news* and *www.tagesschau.de* in 20-minute intervals. These websites were selected because they change frequently. After collecting 900 different versions, we used our approach, XyDiff, and DiffXML to calculate the difference between each consecutive pair

Method	AVG(#Edits)	σ(#Edits)	AVG(Runtime)	σ(Runtime)
RWS-Diff	81.41	42.41	120.93 ms	9.61 ms
XyDiff	1034.14	405.46	37.12 ms	6.32 ms
DiffXML	359.12	278.37	2649.49 ms	244.99 ms

Table 5.3: Result for the *bbc* data set

of versions. DiffXML was sometimes not up to the task and aborted with an exception; the following averages thus only considered the runtimes and results where it did not crash. While our previous experiment only revealed the behavior on a synthetic set of changes, this experiments shows how the methods perform in a real scenario.

Tables 5.3 and 5.4 show the results for the two datasets. In this real-world scenario, the quality gain by using our similarity approach is even higher than for the synthetic changes: While XyDiff is between 4 and 5 times faster than our method because the number of changes is comparably high, the quality of its edit script is highly inferior to our method: For the bbc data set, our method produces on average an edit script that is almost 13 times smaller than the one of XyDiff. In addition, our edit script is also around 4.5 times smaller than the one of DiffXML—even though DiffXML is an $\mathcal{O}(n^2)$ method and should therefore yield better results. For the tagesschau data set, the edit script length of our method is around 6 times smaller than those of XyDiff and DiffXML. The low standard deviation of our method for both data sets shows that similarity-based matching drastically increases the robustness of the method and thus leads to a more constant edit script quality. In contrast, the methods without similarity sometimes produce very inflated edit scripts: XyDiff produced 2083 edit operations between two versions of the bbc dataset while our method produced only 7 for these versions which is around 238 times less.

Our experiments revealed that the theoretical advantage of similarity matching is indeed also a practical one. The edit script quality is considerably increased (even in comparison to an $\mathcal{O}(n^2)$ method) while the runtime stays comparable to simple matching approaches. The similarity also increases the robustness by drastically reducing the variance of the edit script quality. Consequently, similarity-based edit script generation is a viable tool for scenarios where a short edit script is desired but runtime is still important.

Method	AVG(#Edits)	σ(#Edits)	AVG(Runtime)	σ(Runtime)
RWS-Diff	61.32	87.37	126.58 ms	10.76 ms
XyDiff	381.02	736.58	25.15 ms	8.39 ms
DiffXML	313.39	299.90	1399.85 ms	309.64 ms

Table 5.4: Result for the *tagesschau* data set

Although the runtimes of our algorithm and XyDiff are comparable, XyDiff is still faster which was to be expected since similarity computations are more expensive than simple matching computations. In contrast, edit script quality and especially robustness is consistently improved a lot by the similarity matching. In almost all applications, this quality/runtime tradeoff is in favor RWS-Diff, since an edit script is read more often than it is generated: A smaller edit script makes applying that script faster. Since applying an edit script to go back to a former version is the core of most version control systems that use edit scripts, the additional generation runtime will pay off by a reduced runtime for applying the script. Also when changes are used to reconcile or visualize changes, a shorter script uses less bandwidth and yields a better visualization of the changes and is thus always preferable. In conclusion, similarity-based matching is usually worth its increased runtime.

5.6 Conclusion

We proposed a method for rapidly generating an approximately cost-minimal edit script between two trees. Our approach uses subtree similarity for finding a comparably good mapping in cases where simple matching methods like top-down or hash matching fail. Nevertheless, it retains the quasi-linear runtime of such simple methods. The similarity matching is executed by first summarizing each unmapped subtree by a bag of p,q-grams, hashing each of the grams, and generating a random d-dimensional vector from each hash. Then, the vectors are added generating a random walk feature vector. The random walks possess the property that the squared euclidean distance of two walks approximates the symmetric bag distance of the corresponding bags and is therefore a suitable similarity measure. By using index structures, we perform a rapid nearest neighbors search on the feature vectors

to complete the edit mapping. The proposed algorithm is flexible as it can handle various types of edit operations and works for ordered and unordered trees.

Our evaluation has shown that the similarity search is able to decrease the length of the edit scripts up to an order of magnitude while the runtime stays similar to previously published simple matching approaches. This constitutes an important advancement, since a short edit script is extremely beneficial for all applications. In addition to the overall decrease in edit script length, the chance that the matching fails—leading to a huge edit script—is drastically reduced by the similarity matching, so the quality of the generated edit script is far more constant than the quality of previous contributions. RWS-Diff is thus the first generally applicable robust tree diff algorithm with log-linear runtime complexity.

CHAPTER 6

Conclusions

In this thesis, we have investigated in the challenges of integrating hierarchical data support into relational database systems. We depicted shortcomings of existing techniques and proposed a holistic framework for handling hierarchical data in the front end and in the back end of a relational database system.

We started by proposing a data model for integrating hierarchical data into the relational model. Based on this model, we created a query, DML, and DDL language which blends seamlessly into SQL and can be evaluated using traditional relational algebra without any new logical operators. We demonstrated the expressiveness and conciseness of our query language with various real-world customer queries.

Next, we laid our focus on indexing of hierarchical data and efficient hierarchical query evaluation. We elaborated a generic interface, which general purpose hierarchy indexes must possess. Based on this interface, we conducted a study of existing hierarchy indexing techniques and found that most techniques share similar asymptotic properties and expressiveness. We also concluded that no existing indexing scheme is able to handle dynamic hierarchies satisfactorily. As a mitigation of this shortcoming, we proposed Order Indexes as an efficient indexing scheme with unprecedented update and competitive query performance. Our experiments showed that Order Indexes are indeed the first indexing technique feasible for our scenarios. In order to expand our hierarchy support to versioned hierarchies, we proposed the DeltaNI indexing scheme. We demonstrated its unprecedented

performance in the versioned case and showed that it is also the first versioned technique to handle complex updates.

Finally, we proposed the RWS-Diff algorithm for finding the approximately cost-minimal edit script. We introduced the concept of random walk similarity as a general concept for measuring the similarity between objects that can be decomposed into small excerpts. Therefore, our findings do not only apply to hierarchies but can be used in all areas where similarity of decomposable objects is required. Since RWS-Diff runs in quasi-linear time, it is feasible for comparing even huge hierarchies. Our conducted experiments demonstrate that the approximation quality of RWS-Diff is even superior to algorithms with a higher asymptotic runtime complexity.

In conclusion, we proposed a holistic framework for handling hierarchical data in relational systems and showed that it is superior to existing approaches in terms of expressiveness, user-friendliness and especially query and update performance. Our indexing techniques are the first ones to allow for very large hierarchies in highly dynamic settings. Especially, all our contributions enable the efficient use of complex updates, which no existing approach can handle efficiently. Our query language is expressive and yet blends so well with SQL that it does not even require additions to the SQL grammar. We evidenced the feasibility in business scenarios by basing our experiments on real business data of SAP customers. In fact, the necessity of our proposed framework for SAP's customers is witnessed by the fact that large parts of the framework have already been enabled for customer use in the latest release of the SAP HANA Vora in-memory query engine [85]. We are therefore certain that our findings are not only of theoretical scientific interest but are also applicable in practice.

Bibliography

[1] Books Online for SQL Server 2014 – Database Engine – Hierarchical Data.

[2] Oracle Database SQL language reference 12c release 1 (12.1). E17209-15.

[3] Information technology — database languages — SQL, 2011.

[4] Information technology — Database languages — SQL, Part 14: XML-Related Specifications (SQL/XML), 2011.

[5] S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, S. Cluet, G. Cobena, G. Corona, G. Ferran, A. Galland, et al. Xyleme, a dynamic warehouse for XML data of the web. In *IDEAS*, 2001.

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

[7] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A robust numbering scheme for XML documents. In *ICDE*, 2003.

[8] N. Augsten, M. Böhlen, C. Dyreson, and J. Gamper. Approximate joins for data-centric XML. In *ICDE*, 2008.

[9] N. Augsten, M. Böhlen, and J. Gamper. The pq-gram distance between ordered labeled trees. *TODS*, 35(1), 2005.

[10] D. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Queen's University, Kingston, 1995.

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

[12] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, 1997.

[13] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, et al. System RX: One part relational, one part XML. In *SIGMOD*, 2005.

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

[15] P. Boncz, T. Grust, M. Van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.

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

[17] L. Boyer, A. Habrard, and M. Sebban. Learning metrics between tree structured data: Application to image recognition. In *ECML*, 2007.

[18] R. Brunel, J. Finis, G. Franz, N. May, A. Kemper, T. Neumann, and F. Faerber. Supporting hierarchical data in SAP HANA. In *ICDE*, 2015.

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

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

[21] J. Cai and C. K. Poon. OrdPathX: Supporting two dimensions of node insertion in XML data. In *DEXA*, 2009.

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

[23] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, 1997.

[24] S. S. Chawathe. Comparing hierarchical data in external memory. In *VLDB*, 1999.

[25] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *ICDE*, 1998.

[26] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD*, 1996.

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

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

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

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

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

[32] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *TALG*, 6(1), 2009.

[33] S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *CPM*, 2003.

[34] A. Eisenberg and J. Melton. Advancements in SQL/XML. *SIGMOD Rec.*, 33(3), 2004.

[35] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database—an architecture overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[36] J. Finis, R. Brunel, A. Kemper, T. Neumann, F. Faerber, and N. May. DeltaNI: An efficient labeling scheme for versioned hierarchical data. In *SIGMOD*, 2013.

[37] J. Finis, R. Brunel, A. Kemper, T. Neumann, N. May, and F. Faerber. Indexing highly dynamic hierarchical data. In *VLDB*, 2015.

[38] J. Finis, M. Raiber, N. Augsten, R. Brunel, A. Kemper, and F. Faerber. RWS-Diff: flexible and efficient change detection in hierarchical data. In *CIKM*, 2013.

[39] S. J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. *ANSI Document X3H2-96-075r1*, 1996.

[40] M. L. Fredman. On computing the length of longest increasing subsequences. *DM*, 11(1), 1975.

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

[42] T. Grust, J. Rittinger, and J. Teubner. Why off-the-shelf RDBMSs are better at XPath than you might expect. In *SIGMOD*, 2007.

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

[44] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *SIGMOD*, 2002.

[45] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. Naughton, and D. DeWitt. Mixed mode XML query processing. In *VLDB*, 2003.

[46] M. Haustein, T. Härder, C. Mathis, and M. Wagner. DeweyIDs—the key to fine-grained management of XML documents. In *SBBD*, 2005.

[47] C. Hedeler and N. W. Paton. A comparative evaluation of XML difference algorithms with genomic data. In *SSDBM*, 2008.

[48] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB Journal*, 11(4), 2002.

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

[50] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*, 2013.

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

[52] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA*, 1998.

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

[54] K.-H. Lee, Y.-C. Choy, and S.-B. Cho. An efficient algorithm to compute differences between structured documents. *TKDE*, 16(8), 2004.

[55] C. Li and T. W. Ling. QED: A novel quaternary encoding to completely avoid re-labeling in XML updates. In *CIKM*, 2005.

[56] C. Li, T. W. Ling, and M. Hu. Efficient processing of updates in dynamic XML data. In *ICDE*, 2006.

[57] C. Li, T. W. Ling, and M. Hu. Efficient updates in dynamic XML data: From binary string to quaternary string. *VLDB Journal*, 17(3), 2008.

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

[59] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *MobiDE*, 2003.

[60] T. Lindholm. A three-way merge for XML documents. In *DocEng*, 2004.

[61] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery processing in Oracle XMLDB. In *SIGMOD*, 2005.

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

[63] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, 1999.

[64] A. Marian. Detecting changes in XML documents. In *ICDE*, 2002.

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

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

[67] J.-K. Min, J. Lee, and C.-W. Chung. An efficient XML encoding and labeling method for query processing and updating on dynamic XML data. *JSS*, 82(3), 2009.

[68] K. Morton, K. Osborne, R. Sands, R. Shamsudeen, and J. Still. *Pro Oracle SQL*. Apress, second edition, 2013.

[69] A. Mouat. XML diff and patch utilities. BSc thesis, Heriot-Watt University, Edinburgh, Scotland, 2002. `http://prdownloads.sourceforge.net/diffxml/dissertation.ps`.

[70] A. Mouat. DiffXML, 2013. `http://diffxml.sourceforge.net/`.

[71] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[72] P. Nielsen and U. Parui. *Microsoft SQL Server 2008 Bible*. John Wiley & Sons, 2011.

[73] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *CVPR*, 2006.

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

[75] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, et al. XQuery implementation in a relational database system. In *VLDB*, 2005.

[76] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *PRL*, 31(11), 2010.

[77] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *PVLDB*, 5(4), 2011.

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

[79] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards XML version control of office documents. In *DocEng*, 2005.

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

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

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

[83] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *IPM*, 24(5), 1988.

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

[85] SAP SE. Solutions — Data Management — SAP HANA Vora. *http://go.sap.com/germany/product/data-mgmt/hana-vora-hadoop.html*, Dec. 2015.

[86] S. Sasaki and T. Araki. Modularizing B$^+$-trees: Three-level B$^+$-trees work fine. In *ADMS*, 2013.

[87] S. M. Selkow. The tree-to-tree editing problem. *IPL*, 6(6), 1977.

[88] D. Shapira and J. A. Storer. Edit distance with move operations. In *CPM*, 2002.

[89] D. Shasha, J. T. L. Wang, K. Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *TSMC*, 24(4), 1994.

[90] T. Sigaev and O. Bartunov. ltree, 2002. `http://www.postgresql.org/docs/9.1/static/ltree.html`.

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

[92] Y. Song and S. S. Bhowmick. BioDIFF: an effective fast change detection algorithm for genomic and proteomic data. In *CIKM*, 2004.

[93] 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.

[94] D. Suciu. XML data repository, 2012. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[95] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

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

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

[98] Y. Wang, D. J. DeWitt, and J. yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *ICDE*, 2003.

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

[100] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. KF-Diff+: Highly efficient change detection algorithm for XML documents. In *ODBASE*, 2002.

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

[102] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, 2005.

[103] J.-H. Yun and C.-W. Chung. Dynamic interval-based labeling scheme for efficient XML query and update processing. *JSS*, 81(1), 2008.

[104] P. Zezula, F. Mandreoli, and R. Martoglia. Tree signatures and unordered XML pattern matching. In *SOFSEM*, 2004.

[105] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

[106] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *IPL*, 49(5), 1994.

[107] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6), 1989.

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