Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

Enhancing Relational Database Systems for Managing Hierarchical Data

DISSERTATION

Robert Brunel

Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

Enhancing Relational Database Systems for Managing Hierarchical Data

Robert Anselm Brunel

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: *Prof. Dr. Helmut Seidl*
Prüfer der Dissertation:
1. *Prof. Alfons Kemper, Ph. D.*
2. *Prof. Dr. Torsten Grust*

Die Dissertation wurde am 07.08.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.11.2017 angenommen.

Abstract

Hierarchical structures permeate our daily lives, and as such are also featured in many of the software applications we work with. Unfortunately, maintaining and querying hierarchical data in the underlying database systems remains a clumsy and inconvenient task even today, almost 50 years after the relational data model was first conceived. Now that modern in-memory engines are becoming more capable than ever, we take the opportunity to revisit the challenge of making these systems truly "hierarchy-aware." Our approach is based on modeling hierarchies in relational tables in an intuitive and light-weight way by means of a built-in abstract data type. This opens up new opportunities both on the level of the SQL query language as well as at the heart of the database engine, which we fully exploit: We extend SQL with concise but expressive constructs to represent, bulk-load, manipulate, and query hierarchical tables. An accompanying set of relational algebra concepts and algorithms ensure these constructs can be translated into highly efficient query plans. We design these algorithms in terms of a carefully crafted generic framework for representing and indexing hierarchies, which enables us to freely choose among a variety of sophisticated indexing schemes at the storage layer according to the application scenario at hand. While we strive to leverage the decades of existing research on indexing and processing semi-structured data in our framework, we further push the limits when it comes to robustly indexing and querying very large and highly dynamic hierarchical datasets.

The result is an unprecedented degree of native support and versatility for managing hierarchical data on the level of the relational database system. This benefits a great many real-world scenarios which routinely deal with hierarchical data in enterprise software, scientific applications, and beyond.

Contents

1 Introduction 1

1.1 Contributions . . . 2
1.2 Publications . . . 5
1.3 Outline . . . 6

2 Hierarchical Data in Relational Databases 7

2.1 Basic Concepts of Hierarchies . . . 7
2.2 Application Scenarios . . . 10
2.2.1 Common Data Models . . . 11
2.2.2 Our Focus . . . 12
2.2.3 Example Scenarios . . . 13
2.2.4 Typical Queries on Hierarchies . . . 15
2.2.5 Summary of Requirements and Challenges . . . 20
2.3 Status Quo: Integrating Hierarchical Data and DBMS . . . 22
2.3.1 The Adjacency List Model . . . 22
2.3.2 Recursion in SQL . . . 23
2.3.3 Hierarchical Queries in Oracle Database . . . 24
2.3.4 Recursive Common Table Expressions . . . 24
2.3.5 Hierarchical Computations via `ROLLUP` . . . 27
2.3.6 Encoding Hierarchies in Tables . . . 28
2.3.7 `hierarchyid` in Microsoft SQL Server . . . 29
2.3.8 `ltree` in PostgreSQL . . . 31
2.3.9 XML Databases and SQL/XML . . . 31
2.3.10 Multidimensional Databases and MDX . . . 33

3 A Framework for Integrating Relational and Hierarchical Data 35

3.1 Overview . . . 35
3.2 The Hierarchical Table Model . . . 38
3.2.1 Hierarchical Tables . . . 38
3.2.2 The `NODE` Data Type . . . 39
3.3 QL Extensions: Querying Hierarchies . . . 40
3.3.1 Hierarchy Functions . . . 40
3.3.2 Hierarchy Predicates . . . 42
3.3.3 Hierarchical Windows . . . 43
3.3.4 Recursive Expressions . . . 46
3.3.5 Further Examples . . . 47
3.4 DDL Extensions: Creating Hierarchies . . . 50
3.4.1 Hierarchical Base Tables . . . 50
3.4.2 Derived Hierarchies: The `HIERARCHY` Expression . . . 51
3.4.3 Deriving a Hierarchy from an Adjacency List . . . 51
3.5 DML Extensions: Manipulating Hierarchies . . . 53

3.6 Advanced Examples . . . 56
3.6.1 Flexible Forms of Hierarchies . . . 56
3.6.2 Heterogeneous Hierarchies (I) . . . 57
3.6.3 Heterogeneous Hierarchies (II) . . . 57
3.6.4 Dimension Hierarchies . . . 59
3.6.5 A Complex Report Query . . . 60
3.6.6 Hierarchical Tables and RCTEs . . . 61

4 The Backend Perspective of Hierarchical Tables 63
4.1 Hierarchy Indexing Schemes . . . 63
4.1.1 Basic Concepts . . . 64
4.1.2 A Common Interface for Hierarchy Indexes . . . 66
4.1.3 Elementary Index Primitives . . . 67
4.1.4 Index Primitives for Queries . . . 68
4.1.5 Index Primitives for Updates . . . 71
4.2 Implementing Indexing Schemes: State of the Art . . . 73
4.2.1 Challenges in Dynamic Settings . . . 75
4.2.2 Naïve Hierarchy Representations . . . 77
4.2.3 Containment-based Labeling Schemes . . . 78
4.2.4 Path-based Labeling Schemes . . . 80
4.2.5 Index-based Schemes . . . 80
4.3 Schemes for Static and Dynamic Scenarios . . . 81
4.4 PPPL: A Static Labeling Scheme . . . 82
4.5 Order Indexes: A Family of Dynamic Indexing Schemes . . . 83
4.5.1 Order Index Interface . . . 84
4.5.2 The AO-Tree . . . 85
4.5.3 Block-based Order Indexes . . . 85
4.5.4 Representing Back-Links . . . 87
4.5.5 Implementing the Query Primitives . . . 88
4.5.6 Implementing the Update Primitives . . . 89
4.6 Building Hierarchies . . . 91
4.6.1 A Practical Intermediate Hierarchy Representation . . . 92
4.6.2 A Generic build() Algorithm . . . 94
4.6.3 Implementing Derived Hierarchies . . . 97
4.6.4 Transforming Adjacency Lists . . . 98
4.6.5 Transforming Edge Lists to Hierarchy IR . . . 100
4.7 Handling SQL Data Manipulation Statements . . . 103

5 Query Processing on Hierarchical Data 107
5.1 Logical Algebra Operators . . . 107
5.1.1 Preliminaries . . . 107
5.1.2 Expression-level Issues and Normalization . . . 108
5.1.3 Map . . . 111
5.1.4 Join . . . 112

5.1.5 Binary Structural Grouping 114
5.1.6 Unary Structural Grouping 116
5.1.7 Unary Versus Binary Grouping 118
5.2 Physical Algebra Operators 120
5.2.1 Hierarchy Index Scan 121
5.2.2 Hierarchy Join: Overview 123
5.2.3 Nested Loop Join 125
5.2.4 Hierarchy Index Join 125
5.2.5 Hierarchy Merge Join 126
5.2.6 Hierarchy Staircase Filter 134
5.2.7 Structural Grouping: Overview 136
5.2.8 Hierarchy Merge Groupjoin 137
5.2.9 Hierarchical Grouping 140
5.2.10 Structural Grouping: Further Discussion 141
5.2.11 Hierarchy Rearrange 143
5.3 Further Related Work and Outlook 146

6 Experimental Evaluation **153**
6.1 Evaluation Platform 153
6.2 Test Data 154
6.3 Experiments 156
6.3.1 Hierarchy Derivation 156
6.3.2 Index Primitives for Queries 159
6.3.3 Hierarchy Traversal 161
6.3.4 Hierarchy Joins 164
6.3.5 Hierarchical Windows 169
6.3.6 Hierarchical Sorting 171
6.3.7 Pattern Matching Query 174
6.3.8 Complex Report Query 177
6.4 Summary 179

7 Conclusions and Outlook **181**

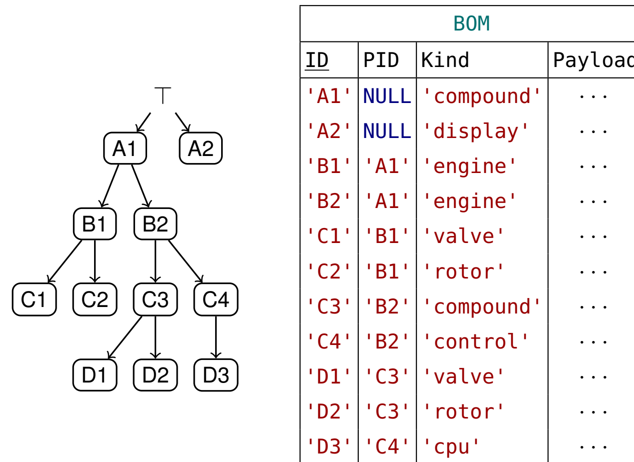
Bibliography **185**

1 Introduction

Hierarchies are infused into our daily lives. Nearly any system in the world can be arranged in a hierarchy, be it political, socioeconomic, technical, biological, or nature itself. As a consequence, hierarchies also appear in software applications throughout. Central concepts in computing are inherently hierarchical, such as file systems, access control schemes, or semi-structured document formats like XML and JSON. If we look specifically at enterprise software, we see hierarchies being used for modeling geographically distributed sites, marketing schemes, financial accounting schemes, reporting lines in human resources, and task breakdowns in project plans. In manufacturing industries, so-called bills of materials, which describe the hierarchical assembly structure of an end product, are a central artifact. And last but not least, in business analytics, hierarchies in the dimensions of data cubes help data analysts to effectively organize even vast amounts of data and guide their analysis to the relevant levels of granularity.

A crucial component in virtually all of the larger mission-critical applications is the database layer, where the hierarchies are ultimately stored. Given the important role that hierarchies play in so many commercial applications, it comes as no surprise that the first recognized database model, devised by IBM for their Information Management System in the 1960s, was hierarchical in nature. One of its intended purposes was to manage bills of materials for the construction of the Apollo spacecraft [5]. Many other early database systems focused primarily on manufacturing scenarios, such that "BOM processor" became a household term. Today, however, virtually all mainstream database systems are based on Codd's relational data model [21].

While the relational data model has been a runaway success for many good reasons, its flat nature makes it particularly hard to accommodate hierarchies and their inherently recursive structure: A hierarchy must be manually mapped to a flat table in the database schema, where *rows* of the table correspond to *nodes* of the hierarchy, and a designated set of columns represents the structure by means of a particular *encoding scheme*. Choosing and manually implementing such an encoding scheme is a major challenge for application developers: The encoding must be expressive enough to be able to answer all relevant types of queries against the hierarchy, and at the same time it must support all types of structural updates the users may wish to perform. A poorly chosen encoding can render certain queries and updates prohibitively expensive to execute, if not impossible to express in the first place. The developer further has to consider the physical design and add supporting database indexes as necessary in order to find a suitable tradeoff between storage utilization, support for updates,



BOM			
ID	PID	Kind	Payload
'A1'	NULL	'compound'	···
'A2'	NULL	'display'	···
'B1'	'A1'	'engine'	···
'B2'	'A1'	'engine'	···
'C1'	'B1'	'valve'	···
'C2'	'B1'	'rotor'	···
'C3'	'B2'	'compound'	···
'C4'	'B2'	'control'	···
'D1'	'C3'	'valve'	···
'D2'	'C3'	'rotor'	···
'D3'	'C4'	'cpu'	···

Figure 1.1: An example bill of materials hierarchy and how it would typically appear in a relational database table.

and query performance. Beyond that, an eye has to be kept on keeping the database logic maintainable, and on foreseeing future update operations and query workloads that may come up in an evolving application. All this is a major burden to developers and very hard to get right. Somewhat surprisingly, today's off-the-shelf relational database products do not offer much support to their users, even though there has been popular demand ever since the relational data model has existed. A solution that application developers thus commonly resort to is to use a trivial data model for representing the hierarchy at the database layer (Figure 1.1) and outsource most logic into the application layer. This situation is especially unsatisfactory in the context of highly capable modern in-memory technology, where software architects are looking for ways to push more logic down to the database layer in order to utilize the available powerful hardware.

This research project therefore takes on the challenge of making database systems truly hierarchy-aware. Our aim is a general and user-friendly framework for storing, manipulating, and querying hierarchical datasets in a relational database context.

1.1 Contributions

Our contributions towards providing comprehensive support for hierarchies in emerging in-memory relational database systems can be summarized as follows:

A holistic approach. We consider all relevant aspects across the layers of a database system architecture: the logical modeling of hierarchical data, the SQL front-end, the translation of our SQL constructs into relational algebra, necessary operators and algorithms for processing updates and queries efficiently, and last but not least the storage layer where the hierarchies are represented and indexed. To our knowledge, we are the first to tackle such a holistic approach. Previous work often starts off on

top of the SQL level (e. g., by adding functionality through user-defined data types or stored procedures) without touching the backend. Our invasive method enables a deeper and more comprehensive integration of hierarchical and relational data and results in unprecedented usability and performance.

A design based on real-world use cases. Our research has been conducted in cooperation with the HANA Database group [29] of SAP SE based in Walldorf. As a major vendor of enterprise solutions, SAP has a plethora of relevant use cases. Academic literature sometimes focuses on techniques that are theoretically powerful but turn out to lack usability in practice. By contrast, we consider a selected set of real-world use cases and let them guide us in finding the "right" degree of flexibility and functionality to support. This ensures our research can be of immediate benefit to the industry.

An analysis of conventional approaches to hierarchies in RDBMS. We thoroughly investigate the present state of support for hierarchies in today's off-the-shelf database systems. We also briefly cover more distantly related functionality and research tracks. While many relevant query processing techniques—e. g., for recursive common table expressions and structural joins—have been tackled by prior research, our analysis from a practical point of view reveals that a robust and efficient solution that would optimally meet the requirements of our application scenarios is still missing to date. Our research thus generalizes and reconciles many of the prior ideas and techniques into a more comprehensive, flexible, and user-friendly framework for handling hierarchies specifically in a relational context.

A concept to allow explicitly modeling hierarchies as "first-class citizens". The fundamental idea of our approach is to allow users to model *hierarchical tables* within the database by means of a built-in abstract data type `NODE`. This intuitive and light-weight concept makes the fact that a hierarchy is being modeled explicit, and ensures that hierarchical and relational data can interact seamlessly. The `NODE` type supports a well-defined hierarchy topology: irregular forest structures with heterogeneous node types. It also gives access to well-defined functionality with guaranteed performance characteristics.

Carefully crafted extensions to the SQL query language. As a consequence of being unable to model hierarchies explicitly in the past, SQL has no real support for queries on hierarchies, which results in unintelligible and unmaintainable query statements. Our `NODE` data type can serve as a "syntactic handle" to the hierarchy and thus provides the foundation for attractive language features. We add concise but expressive constructs to create and populate hierarchical tables, to manipulate the hierarchy structure, and to effectively express queries on the hierarchies. Our constructs interact seamlessly and intuitively with well-established SQL concepts such as views, joins, and windowed tables, and they are light-weight and orthogonal in that no significant grammar extensions are required. Thus, the original spirit of SQL is maintained.

Novel functionality regarding hierarchical computations. Our SQL extensions allow users to conveniently express a class of practically relevant queries from typical application scenarios. When it comes to aggregation-like hierarchical computations involving

structural recursion, they go beyond what can practically be expressed in SQL today. Many applications—in particular in business analytics—depend on such functionality and will thus be able to push logic down to the database layer.

Engine support for query processing. Lacking data modeling and query language support, adding engine support for queries on hierarchies has not been a consideration in today's database systems. However, to be able to evaluate complex real-world queries with high performance and scalability, such support is indispensable. In our design the backend is aware of the hierarchical structure of the involved tables and thus many opportunities are opened up in that regard. We describe a comprehensive system of generic and asymptotically efficient algebra operators and algorithms for implementing the functionality that our SQL constructs expose. We also cover considerations on the relational algebra level and issues around generating efficient query plans.

A flexible framework for hierarchy indexes. Our algebra operators are designed in terms of a carefully crafted generic framework for representing and indexing hierarchies at the storage layer. A *hierarchy indexing scheme* comes with a definition of the abstract `NODE` data type and is encapsulated by a common index interface, which exposes low-level query primitives and update operations with suitable performance guarantees. A broad range of existing sophisticated techniques for encoding hierarchies fit into this design. This provides users with the flexibility to choose among a variety of indexing schemes according to the application scenario at hand.

A survey of indexing techniques. The technical challenges of storing and indexing recursively structured data have received a fair amount of attention from the research community in the past two decades, although that work has not primarily been in the context of relational databases but in the domains of XML and semi-structured databases. A plethora of tree encoding schemes have been proposed that can equally be applied to hierarchies in a relational schema. We contribute a comprehensive survey of this area, and design our framework to embrace the prior art.

Pushing the limits of hierarchy indexing. As part of our research project a family of indexing schemes called *order indexes* has also been proposed, which further advances the state of the art when it comes to indexing hierarchies in relational databases. Unlike many prior works, their emphasis is particularly on coping with very large and at the same time highly dynamic datasets, on supporting complex update operations, and on offering robustness in case of unfavorable "skewed" update patterns. While their design and implementation is not in the scope of this thesis, we cover how they are fitted into our framework and demonstrate their feasibility in our benchmarks.

Working with existing hierarchical data. Providing a migration path for legacy applications is of high practical relevance but easily overlooked in green-field designs. We place an emphasis on this topic by providing language constructs to derive hierarchical tables from existing relational data. We also cover the necessary algorithms to efficiently bulk-load hierarchy indexing schemes from the data.

Prototype. Our research prototype provides a proof of concept for the mentioned algorithms and algebra operators. It also incorporates a considerable selection of indexing scheme implementations that fit our generic design. We use it to conduct an experimental evaluation to demonstrate the performance characteristics of our framework.

1.2 Publications

Some ideas and figures presented in this thesis have appeared previously in the following publications. Some were written jointly with collaborators from TUM and SAP.

- Robert Brunel and Jan Finis. "Eine effiziente Indexstruktur für dynamische hierarchische Daten". In: *Datenbanksysteme für Business, Technologie und Web (BTW) 2013. Workshopband P-216* (Mar. 2013), pp. 267–276
- Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Franz Färber, and Norman May. "DeltaNI: An efficient labeling scheme for versioned hierarchical data". In: *Proc. 2013 ACM SIGMOD Int. Conf. Management of Data*. ACM, June 2013, pp. 905–916
- Jan Finis, Martin Raiber, Nikolaus Augsten, Robert Brunel, Alfons Kemper, and Franz Färber. "RWS-Diff: Flexible and efficient change detection in hierarchical data". In: *Proc. 22nd ACM Int. Conf. Information and Knowledge Management (CIKM)*. ACM, Oct. 2013, pp. 339–348
- Robert Brunel, Jan Finis, Gerald Franz, Norman May, Alfons Kemper, Thomas Neumann, and Franz Färber. "Supporting hierarchical data in SAP HANA". In: *Proc. 31st Int. Conf. Data Engineering (ICDE)*. IEEE, Apr. 2015, pp. 1280–1291
- Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. "Indexing highly dynamic hierarchical data". In: *Proc. VLDB Endowment* 8.10 (Aug. 2015), pp. 986–997
- Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. "Order Indexes: Supporting highly dynamic hierarchical data in relational main-memory database systems". In: *The VLDB Journal* 26.1 (Feb. 2017), pp. 55–80
- Robert Brunel, Norman May, and Alfons Kemper. "Index-assisted hierarchical computations in main-memory RDBMS". In: *Proc. VLDB Endowment* 9.12 (Aug. 2016), pp. 1065–1076

1.3 Outline

The remainder of this thesis is organized as follows:

CHAPTER 2: Hierarchical Data in Relational Databases

This chapter provides our foundation and motivation. It introduces all relevant concepts concerning hierarchical data in relational databases. We characterize the forms of hierarchical data we target, motivate our focus by citing typical scenarios and queries from the enterprise applications of SAP, and state specific requirements a relational database system has to fulfill to be called "hierarchy-aware." A survey of the state of the art positions us against the various related work in the area.

CHAPTER 3: A Framework for Integrating Relational and Hierarchical Data

This chapter covers our framework for hierarchical data in SQL-based database systems, beginning with its core concept of encapsulating the structure of a hierarchy by an abstract data type `NODE`. Based on this we motivate and define extensions to SQL for creating, querying, and modifying a *hierarchical table*. A few advanced examples illustrate that the language elements cover typical usage scenarios and blend seamlessly with the "look and feel" of SQL.

CHAPTER 4: The Backend Perspective of Hierarchical Tables

This chapter discusses how hierarchical tables and the `NODE` type are implemented and indexed under the hood, what exactly makes up a *hierarchy indexing scheme*, and our common abstract index interface. We also thoroughly survey a broad range of existing hierarchy encoding techniques that fit into this design, and describe how it accommodates our proposed *order indexes* family of indexing schemes. Finally, we discuss how to implement update statements as well as efficient bulk-building from existing hierarchical data.

CHAPTER 5: Query Processing on Hierarchical Data

This chapter covers the translation of our query language extensions into relational algebra and from there into executable query plans. We propose a comprehensive set of hierarchy-aware physical algebra operators, analyze their properties in detail, and discuss their generic implementation in terms of the query primitives of our index interface.

CHAPTER 6: Experimental Evaluation

In this chapter we demonstrate the feasibility of our framework by exercising the proposed indexing and query processing techniques against synthetic and real-world data based on a prototypical execution engine. Our experiments comprise our own family of indexing schemes and physical algebra operators as well as typical alternative approaches and ad-hoc solutions.

CHAPTER 7: Conclusions and Outlook

This chapter concludes this thesis, wraps up the key properties of our solution, and outlines possible directions for future research.

2

Hierarchical Data in Relational Databases

This chapter sheds light on our motivation and the challenges we tackle in this thesis. We first introduce the necessary fundamental concepts around hierarchies in a relational database context (§ 2.1). This foundation allows us to precisely characterize the forms of hierarchies we intend to support with our framework (§ 2.2.1, § 2.2.2). We motivate our focus by looking at a number of scenarios in enterprise applications (§ 2.2.3), which in particular gives us an understanding of the common types of queries (§ 2.2.4). Based on this discussion, we summarize the requirements a database system has to meet in order to exhibit decent support for managing hierarchical data (§ 2.2.5). Finally, we survey the state of the art of how users deal with such data today, which reveals the existing functionality gaps that our framework intends to fill (§ 2.3).

2.1 Basic Concepts of Hierarchies

Hierarchical data appears in database applications in various forms. Much like an ordinary relation in the relational data model, a *hierarchy* is a collection of real-world entities such as objects, names, or categories. The gist is a particular relationship that is defined between the entities, which gives meaning to terms such as one entity being "above," "below," or "at the same level as" another. Different applications place different constraints on how exactly the hierarchical arrangement may look. Figure 2.1 displays two examples. In hierarchy (a) each entity has at most one superordinate entity, and there are no cycles in the "is below" relationships. Hierarchies like this are often called *strict*. Hierarchy (b) has *two* top-level entities, (A) and (F), and allows entity (D) to have multiple superordinates: it is "directly below" (B), (C), and (F). A hierarchy like this is sometimes referred to as an *overlapping hierarchy*. Figure 2.1c shows the common data model in the high-level entity/relationship notation [18]: Each entity is identified by a key and described by certain attributes, and there is an "is below" relationship between entities. The multiplicity N of the relationship can be 1 in order to model a strict hierarchy, or it can be $*$ to allow for an overlapping hierarchy. In a relational database, the $N = 1$ model can be implemented as a single flat table:

```
CREATE TABLE Entity (
   ID INTEGER NOT NULL PRIMARY KEY,
   PID INTEGER NULL REFERENCES Entity (ID),  -- "is below"
   ...  -- additional attributes
)
```

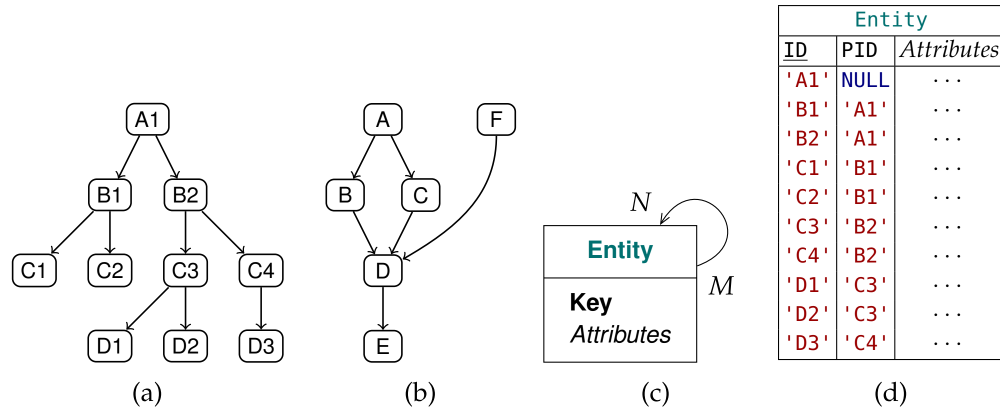


Figure 2.1: Two example hierarchies, and how they may appear in entity/relationship models (c) and relational database tables (d).

The hierarchical relationship is established by the foreign key self-reference, which associates each entity (row) with the respective superordinate entity. Figure 2.1d shows example data representing hierarchy (a).

In practice, the basic multiplicities of E/R diagrams and `REFERENCES` constraints of SQL are not expressive enough to precisely capture the semantics of the modeled hierarchies. Most applications have additional constraints on the structure of the hierarchies. For example, the above models permit cyclic "is below" relationships (an entity could even be a subordinate of itself!), which is usually meaningless.

The Structure of a Hierarchy. To characterize hierarchical structures precisely we need to bring in the terminology of graph theory (as covered, e.g., in [26]). We can describe a data model such as Figure 2.1c as a *vertex-labeled directed graph*, where each graph node is associated with exactly one entity (table row). The mathematical model of a *directed graph* is a tuple $G = (V, E)$, where V is some set of objects, the *nodes* (or vertices), and where $E \subseteq V \times V$ is a set of ordered pairs of elements of V representing the directed *edges*. By their nature as elements of a set, the nodes are distinguishable, so we may call the graph vertex-labeled.

In our setting, the labels (the contents of V) are the key values that identify the modeled entities. The edge set E thus contains pairs of keys and directly represents the "is below" relationship. Note that an edge is defined solely by the two nodes it connects: In hierarchies the focus is typically on the entities themselves rather than on their relationship. Treating the edges as *weak* entities without an own identity is therefore usually sufficient. Furthermore, neither the E/R model nor the relational model would allow the same edge to exist *multiple* times between a particular pair of nodes. In line with this, our graph model defines E as a set rather than a multiset, precluding so-called "multiple edges" by design. Beyond this, our relational model does not impose any further structural constraints that may hold in applications. The terminology of graph theory helps us to express these. Common restrictions are:

- *Connected graphs:* In a connected graph, every pair of nodes is connected by at least one path. There are no unreachable nodes.

- *Acyclic directed graphs (DAGs):* A directed graph is *acyclic* if there are no directed cycles. A *directed cycle* is a closed walk over the nodes using only edges in E, with no repetitions of nodes and edges (other than the end node matching the start node). Disallowing directed cycles also precludes edges that connect a node to itself, so-called *loops*. However, a DAG may still have an *undirected* cycle, a closed walk that ignores the direction of the edges.

- *Rooted trees:* A rooted tree is a directed graph with the following properties: (1.) It is connected. (2.) It is *rooted*: There is a designated *root* node $\top \in V$ which determines the direction of the edges. All edges are oriented away from the root[1]. (3.) It is acyclic. There are neither directed nor undirected cycles.

The example (a) of Figure 2.1 is a tree with root A1, whereas (b) is a connected DAG.

Further Terminology for Trees. Besides being connected and acyclic, trees have several further characteristic properties, which render them comparatively simpler to represent and process algorithmically: The absence of cycles means that any node is reachable from the root through a *unique* path. It also implies that a tree of $|V|$ nodes has exactly $|E| = |V| - 1$ edges (assuming $|V|$ is finite). For example, the hierarchy in Figure 2.1a has 10 vertices and, by its nature as a tree, 9 edges.

The concept of a unique root path furthermore gives rise to a rich and intuitive terminology that we will heavily use in this thesis. The most important terms are:

- If the unique path from $\top$ to node v passes through node u, node u is called an *ancestor* of v, and v is called a *descendant* of u.

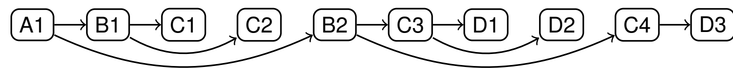
- The *parent* of a node v is the ancestor that is directly connected to v. Every node except the root has a unique parent. A *child* of a node v is a node of which v is the parent. A node that has no children is called a *leaf*. A node that is not a leaf is called an *inner* node (also *internal* or *branch* node).

- The *depth* of a node v is the number of nodes on the path from $\top$ to v. In the example of Figure 2.1a the depth of D1 is 4. (By contrast, in the DAG of Figure 2.1b, the depth of D is ambiguous: it can be seen as either 3 or 2.) A hierarchy in which all leaves have the same depth is called a *balanced hierarchy*.

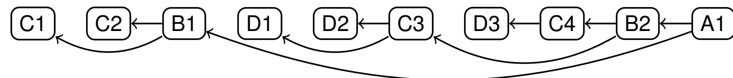
[1]Equivalently, all edges could be considered as being oriented *towards* the root. This interpretation is essentially up to the application. We consistently stick to the former, more common interpretation, as it matches the natural direction of traversal. Also note that as $\top$ determines the direction of the edges, there is no need to explicitly manifest that direction in the mathematical model. In the literature, a *tree* is therefore more commonly defined as a 3-tuple $(V, E, \top)$ where (V, E) is an *undirected* graph, whose edges E are unordered pairs (2-sets) of nodes.

Ordered Hierarchies. In some applications the relative order of entities with a common superordinate is meaningful, although neither E/R models nor SQL have built-in means for expressing this. In structured documents like XML, for example, a "document order" is inherent. In other scenarios, a particular attribute may represent the order. Even if that is not the case, we want to treat such hierarchies as being ordered when storing them in a database. Although maintaining the order incurs space and runtime overhead, queries such as "enumerate all nodes in their natural hierarchy order" could otherwise not be answered deterministically. Mathematically, we are thus dealing with *ordered* graphs: An ordered graph is an unordered graph where the outgoing edges of each node are totally ordered.

Two concepts that exist only with ordered trees are *preorder* and *postorder*. They refer to a linear ordering of the nodes that can be produced by "tracing" an ordered depth-first traversal. In a preorder listing, the root of each subtree is listed just before the nodes in its subtree:



In a postorder listing, each root appears right *after* the nodes in its subtree:



If we assign numbers to the nodes reflecting their preorder and postorder ranks, we can characterize the terms *ancestor* and *descendant* via these numbers:

$$\begin{aligned} v \text{ is a descendant of } u \;&:\Longleftrightarrow\; \mathrm{pre}(v) > \mathrm{pre}(u) \;\wedge\; \mathrm{post}(v) < \mathrm{post}(u) \\ v \text{ is an ancestor of } u \;&:\Longleftrightarrow\; \mathrm{pre}(v) < \mathrm{pre}(u) \;\wedge\; \mathrm{post}(v) > \mathrm{post}(u) \end{aligned}$$

The other two possible combinations are:

$$\begin{aligned} v \text{ precedes } u \;&:\Longleftrightarrow\; \mathrm{pre}(v) < \mathrm{pre}(u) \;\wedge\; \mathrm{post}(v) < \mathrm{post}(u) \\ v \text{ follows } u \;&:\Longleftrightarrow\; \mathrm{pre}(v) > \mathrm{pre}(u) \;\wedge\; \mathrm{post}(v) > \mathrm{post}(u) \end{aligned}$$

In our example, C3 follows B1, C1, and C2, but precedes C4 and D3. Note the symmetry in the ancestor/descendant and preceding/following relationships.

In the terminology of XPath [110], binary relationships of this kind are called *axes*, and further axes beyond ancestor, descendant, preceding, and following are defined. However, in the context of "pure" hierarchies the ancestor and descendant axes matter the most. These concepts will play a central role in later chapters, in particular § 5.

2.2 Application Scenarios

In our cooperation with SAP, we consulted with stakeholders of various enterprise applications and investigated a number of scenarios that can benefit from first-class support for hierarchical datasets. Armed with the concepts introduced in the previous section, we can now precisely characterize the modeled types of hierarchies and clarify the scenarios we particularly focus on.

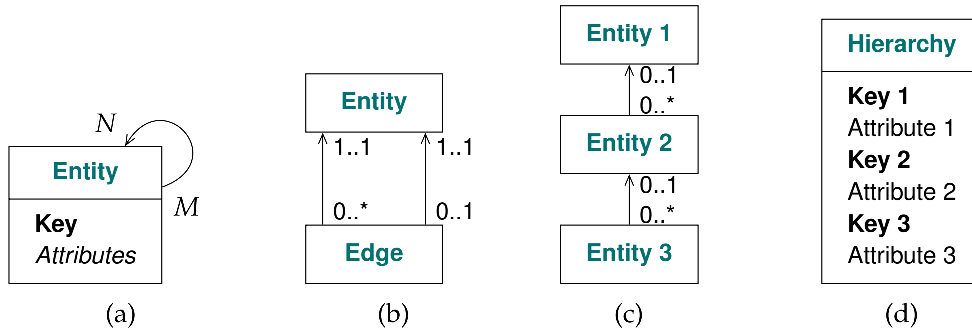


Figure 2.2: Common entity/relationship models of hierarchical data

2.2.1 Common Data Models

Figure 2.2 shows, again from an abstract entity/relationship point of view, different models of hierarchical data that are commonly seen in practice:

(a) This is the basic "self-referencing entity" model we have already seen in Figure 2.1c. It is capable of representing directed graphs, but they may be disconnected or cyclic. If $N = 1$ and cycles are disallowed, it closely corresponds to our notion of a tree.

(b) This variant is equivalent to (a) with $N = 1$, but gives edges a distinct identity.

A design in the fashion of (b) is necessary when a general graph is modeled and the edges are to be labeled. However, labeled edges are untypical for "pure" hierarchies where the focus is primarily on the entities (the nodes). Also note that for a tree the distinction between node attributes and edge attributes is technically not necessary: Since every entity has a unique superordinate, there is a 1 : 1 correspondence between a node v, its incoming edge (u, v), and transitively the whole path $\top, \ldots, v$ from the root to v. This allows us to "join" the Edge relation into the Entity relation, bringing us back to variant (a). The interpretation of whether an attribute is a node label or an edge label is then up to the application.

(c) Besides having a single self-referencing entity, another common pattern is a series of multiple entities in $1 : N$ relationships, directed from subordinate entity to superordinate entity, respectively.

This models a *homogeneous* hierarchy where the number of levels is fixed and all entities on a level are of the same type. These hierarchies are usually also balanced. A common example is geographic data:

Continent : {[Name]}	= {(Europe), ...}
Country : {[Name, Continent]}	= {(Germany, Europe), ...}
State : {[Name, Country]}	= {(Bavaria, Germany), ...}
Region : {[Name, State]}	= {(Upper Bavaria, Bavaria), ...}
City : {[Name, Region]}	= {(Munich, Upper Bavaria), ...}

Although this model is widespread in practice, it does not make sense for every application. The rigid homogeneous structure creates modeling problems when entities on certain levels do not actually exist (e. g., states and regions in tiny countries). To account for such cases one would have to either insert a dummy region or distort the application logic accordingly.

Model (c) can be transformed into model (a) by introducing a common key (e. g., an integer ID) with a shared domain across all involved tables, and then arranging these IDs in a separate self-referencing relation according to model (a).

(d) In this model there is a single "denormalized" relation with a number of dedicated attributes $a_1, \ldots, a_k$ which represent the hierarchy structure somewhat implicitly: Attribute a_i (as a whole) represents level i of the hierarchy, and each distinct a_i value represents a particular node on that level. A tuple of a_i values corresponds to a complete path from the root to a leaf node.

Model (d) is a variant of model (c) where all relations are joined into one. In practice many database tables exhibit this design or can be interpreted as a hierarchy in this way. For example, every table containing a date field could be viewed as a Year–Quarter–Month–Day hierarchy.

2.2.2 Our Focus

Our goal is to make working with data models such as those shown in Figure 2.2 as convenient and efficient as possible. We primarily focus on hierarchy structures of the following two categories:

1. Rooted trees. We refer to these as *strict* hierarchies. They may be inhomogeneous (unlike models c and d), unbalanced, arbitrarily wide and deep, and ordered.

2. *Non-strict hierarchies* that can reasonably be viewed as strict hierarchies.

Although the data models of Figure 2.2a and b can accommodate them, we do not intend to support arbitrary directed graphs. An often-cited example for this kind of scenario is flight planning (e. g., [82]): In a flight route graph, entities of type Airport are connected via Flight edges, forming a complex and inherently cyclic directed graph. Flights may be associated with attributes such as a departure time, duration, and cost, and a common query may be: "For every pair of cities, count the potential flights in a given time range." To express such queries effectively one requires a graph-centered query language [106], and evaluating them involves systematic graph traversals that can handle encountered cycles. This appears to be better tackled by dedicated graph databases (see, e. g., [89]). We therefore decided to exclude any graphs that cannot reasonably be interpreted as strict hierarchies from our scope. Our goal is a light-weight solution in the world of SQL that tightly integrates with relational database backends, where most business data today lives. This targeted focus allows us to provide features that are both user-friendly and "performance-friendly" to a degree that would be hard to achieve by a one-size-fits-all solution for graph-like data.

That said, deviations from a strict hierarchy structure are sometimes necessary or desirable in real-world applications such as those of SAP. Acyclic directed graphs with a limited number of edges violating the strictness condition are not as problematic. We found that these scenarios can usually be handled by working on a tree view of the graph, on which all operations of strict hierarchies are well-defined. Such a view can be obtained either by eliminating edges to extract a spanning tree, or by replicating any subtrees that are reachable via multiple paths. We anticipate these advanced requirements in our framework.

2.2.3 Example Scenarios

We now look at typical applications featuring strict hierarchies, mostly in the context of enterprise software. To gain insights into which features and semantics we require from our framework, we consider both the data modeling perspective as well as the queries that are commonly run against the data.

Materials Planning. A *bill of materials* (BOM) is perhaps *the* classic use case for databases in business applications. It is defined by the German standard DIN 199 [27] as "a complete, formally structured list of the components that make up a product or assembly." The BOM is a central artifact in enterprise resource planning systems, where it can drive the whole value chain from engineering to manufacturing through to sales and maintenance. From a data modeling perspective, a BOM contains entities of type Part, and there is a Part "is composed of" Part relationship, possibly associated with a quantity denoting how often the part appears within the composite. The relative order of parts usually does not matter.

By far the most common materials planning procedure is the *parts explosion*, a structured (indented) list that breaks apart an assembly into the components that are needed to build it. Explosions can be created in various ways: The list may be in depth-first or (more rarely) breadth-first order; it may show only direct subassemblies (single-level explosion) or a limited number of levels. *Where-used queries* view the BOM in a bottom-up manner, going from raw materials to composites at increasingly higher levels. Often there are additional filters, or accumulated quantities are displayed with the parts. Example queries are: "What are the total production costs for assembly X?"; "List subassemblies whose accumulative costs are larger than X"; "Which raw materials need to be ordered in which quantity?"; "What is the total quantity of each component required to build X, grouped by component type?" (summarized explosion); and "In how many different assemblies does a subpart of type t appear?"

Human Resources. In large organizations, department divisions and reporting lines usually form hierarchies. These comprise two entity types, Employee and Department, in different relationships: A department "subdivides" another department, and each employee is "assigned to" a department and "reports to" a supervising employee. The sibling order does not matter. In a more complex model an employee could be partially assigned to multiple departments (creating an $N : M$ relationship), where each assignment is labeled with the contribution in percent. Queries are primarily about

listing and inspecting the organization structure: “Which employees work under manager X?”; or “Display the reporting line for Y.” Beyond that, simple computations may be performed: “Count all employees within each level-k department”; or “List managers who are responsible for more than n employees.”

Enterprise Asset Management. An *asset hierarchy* helps to keep track of production-relevant assets (such as machine parts, tools, and auxiliary equipment) and their functional locations (plants, rooms, assembly lines). It contains *two* types of potentially nested entities: Functional Location and Equipment. A piece of equipment is “installed at” a functional location and may be “composed of” another piece of equipment. Alternatively, two separate location and equipment hierarchies could be modeled—the latter resembling a BOM. The sibling order usually does not matter.

Profit Center Accounting. Cost and profit centers are typically arranged in a strict hierarchy with a Cost Center “is composed of” Cost Center relationship. The sibling order reflects the numbering scheme of the accounts. Queries often involve computations, such as “Sum up the costs for each level-k cost center.”

Project Planning. Planning and issue tracking systems often structure their items into milestones and each milestone into tasks and subtasks. The primary relationship is Task “depends on” Task. A task is considered complete when all of its subtasks are complete. The sibling order may model the priorities or the planned order of completion. More complex project plans may additionally contain cross-links between tasks, resulting in an acyclic directed graph. However, strictly hierarchical project plans are often preferred, as they are easier to display, navigate, and maintain.

Documents. Semi-structured text, most notably XML [108] and JSON [52], is by nature strictly hierarchical and ordered. A document consists of entities of type Document Node, with subtypes such as Text Block, Paragraph, Section, Page, et cetera. There may be auxiliary cross-references within a document. Queries involve searching for document nodes of certain types, matching simple patterns against the document, and summing up metrics such as word counts along the document structure.

Many other artifacts in computing and software technology could be counted into this category, such as file repositories, package dependencies, and abstract syntax trees.

Business Intelligence (BI). In analytic applications, hierarchies are routinely used to explore large *fact tables*. Such a table characterizes facts (e. g., sales items) by a number of *dimension attributes* (e. g., the sold product) and associates them with numeric *measures* of interest (e. g., the price). Many dimensions can be organized as hierarchies. For example, consider a large enterprise that uses a hierarchical scheme for its product portfolio (e. g., Business Area–Product Line–Product–Version). We may have a table of sales and an auxiliary table storing the products:

$$\text{Sale} : \{[\underline{\text{ID}}, \underline{\text{Item}}, \text{Customer}, \text{Product}, \text{Date}, \text{Amount}]\}$$
$$\text{Product} : \{[\underline{\text{ID}}, \text{Name}, \text{Category}, \text{Size}, \text{Price}]\}$$

If Category identifies a node in the product hierarchy, Product can act as a hierarchical dimension of the sales table. Other common examples are geographic dimensions such as Continent–Country–Region–City, or date/time dimensions such as Year–Quarter–Month–Day or Year–Calendar Week–Day. BI products use such dimensions to interactively filter and group the facts. The user may begin his analysis on the top-most level—at the most coarse level of detail—and then "drill" deeper as required. From published material on BI technology and multidimensional data models [50, 54, 86, 101] we can learn about the typical characteristics of dimension hierarchies: They are usually *strict*, as directed graphs are difficult to handle in user interfaces and pose semantic issues when summarizing and accumulating the data. The sibling order is usually meaningless for discrete dimension values, or the required order may depend on the type of report (e. g., products ordered by name or by decreasing revenue). For simplicity, the hierarchies are also often *balanced* and treated primarily as a fixed set of named levels, so they can be conveniently referred to in the user interface. (Although this is not always the most appropriate modeling, as we noted in § 2.2.1.) The lowest level has a distinguished role, as the leaves in each dimension represent the specific items the facts are mapped to (e. g., the actual products), whereas the inner nodes represent abstract groups or summarized ranges of items.

Multiple hierarchies may be in place for a dimension, as the various examples of subdividing a date/time period show. A design one sometimes encounters is to mix multiple divisions into a single *overlapping* hierarchy. For example, when hierarchies for both Fiscal Year and Calendar Year are defined, sharing the lower Month–Day levels instead of replicating them in two hierarchies seems convenient. But this is not clean, as Day 1 of January is in fact a different entity than Day 1 of February. Turning the hierarchy into a DAG needlessly complicates matters. As two different divisions are rarely used in the same query, it is virtually always preferable to treat them as different (strict) hierarchies. We therefore do not intend to support overlapping hierarchies.

Queries involving dimension hierarchies are generally computation-intensive. They apply various filters to the fact data and aggregate the measures along the hierarchy structure. Examples are: "Compute the total/average sales amount per region"; or "List regions with total sales below average." A challenge is that *multiple* hierarchical dimensions are sometimes involved: "Display a crosstab of total sales per country/region and per year/month." Having multiple dimensions essentially multiplies the number of possible paths through the hierarchies for aggregation.

2.2.4 Typical Queries on Hierarchies

By looking at the scenarios discussed in the previous section, we see that queries involving hierarchies can broadly be classified into three categories:

1. *Basic node queries* produce a potentially ordered listing of a selection of hierarchy nodes and project basic properties such as their depths.

2. *Structural pattern matching queries* filter and match entities based on their positions in the hierarchy.

3. *Hierarchical computations* associate a subset of the hierarchy nodes with numeric values and then propagate or aggregate them along the hierarchy structure.

Due to the limitations of the relational model and SQL, hierarchies are in practice often represented using very basic relational encodings, which preclude implementing such queries at the database layer. While investigating the applications of SAP, we observed several implementations of largely equivalent hierarchy-handling logic written in ABAP (running within the application server) or stored procedures. These approaches have significant shortcomings regarding performance, interoperability, and maintainability. Our goal, therefore, is to be able to express all relevant queries directly using appropriate extensions to SQL. To achieve this we need to examine the three categories of queries and clarify what features exactly are missing from the language.

Basic Node Queries. Certain properties of the hierarchy nodes are needed frequently and should therefore be easily accessible. The most interesting properties of a node are its depth, whether it is a root or a leaf node, and its number of children. Besides querying these properties, the user may wish to filter nodes according to their values, and produce a listing of the nodes in a particular order. These requirements all map fairly straightforwardly onto the basic `SELECT–FROM–WHERE–ORDER` framework of SQL, so no major conceptual enhancements are needed. We only have to extend the expression language of SQL appropriately. For example:

```
SELECT ID, depth of the node FROM Hierarchy WHERE depth of the node <= 3
 ORDER BY depth-first hierarchy order
```

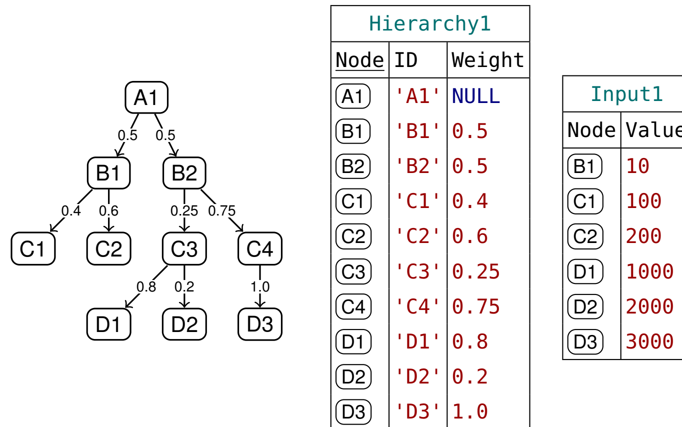
Structural Pattern Matching Queries. Pattern matching queries involve a hierarchical pattern, which can be as simple as "v is below u," where u and v are placeholders for nodes. The user usually either wants a list of all combinations of nodes that match the given pattern, or test a particular list of combinations against the pattern and see those combinations that match. To illustrate a more complex pattern matching query, we revisit the bill of materials table from Figure 1.1 (p. 2):

> "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."

The qualifying combinations are (B2, D2, C3), (B2, D2, A1), and (B1, C2, A1).

The challenge with pattern matching queries lies in the recursive nature of the "is below" relationship. We want to be able to match, for instance, a node v with a particular ancestor u *without* having to—from a language perspective—explicitly mention or "loop" over the intermediate nodes between u and v in the query statement. It turns out these kinds of queries fit quite well into SQL's declarative paradigm. The most appropriate mechanisms to express pattern matching are filters and structural joins on hierarchy axes. For example, the simple "v is below u" pattern becomes:

```
SELECT u.ID, v.ID FROM Hierarchy u, Hierarchy v WHERE v's node is a descendant of u's node
```



Hierarchy1

Node	ID	Weight
A1	'A1'	NULL
B1	'B1'	0.5
B2	'B2'	0.5
C1	'C1'	0.4
C2	'C2'	0.6
C3	'C3'	0.25
C4	'C4'	0.75
D1	'D1'	0.8
D2	'D2'	0.2
D3	'D3'	1.0

Input1

Node	Value
B1	10
C1	100
C2	200
D1	1000
D2	2000
D3	3000

Figure 2.3: A hierarchy with weighted edges represented by table Hierarchy1, and a table Input1 of numeric input values attached to some of the nodes.

Note that the hierarchy may potentially contain a large number of *intermediate nodes* between any two nodes matching this pattern. Intermediate nodes merely provide connectivity between the nodes of interest; they otherwise are not relevant to the query. This leads us to an important design goal concerning the evaluation of such queries: The evaluation of a pattern should not be required to "touch" (or iterate over) the intermediate nodes in any way. In particular, the number of intermediate nodes should be insignificant to the overall performance. To achieve this goal, we require appropriate hierarchy indexes that efficiently maintain the connectivity information, as well as join operators that can leverage those indexes. Many relevant indexing and query processing techniques have been studied previously in the domain of XML databases [11, 41, 53, 118]. The problem of efficiently evaluating pattern matching queries can thus be considered as largely solved already—albeit in a different context.

Hierarchical Computations. When it comes to the third category of queries, things are less obvious. Consider the example hierarchy in Figure 2.3, where the edges are weighted and some nodes are associated with numeric values given by table Input1. If Hierarchy1 would model a bill of materials, for example, ID could be a foreign key referencing the part master data, Weight the part quantity, and Value the prices of selected components. Now, suppose we want to compute weighted sums of those values bottom up—how can we state a SQL expression that correctly incorporates the weights? To see how this can work, we need to take a closer look at the intended semantics.

In general, a *hierarchical computation* propagates and accumulates data—usually numeric values—along the hierarchy edges. Data flow can happen either in the direction towards the root (*bottom up*) or away from the root (*top down*). The computation input may include the "static" labels stored with the entities themselves (e. g., ID and Weight in Hierarchy1). However, in realistic queries the input is generally the result of an arbitrary subquery, which associates the hierarchy nodes with "dynamic" input values

Input1	
Node	Value
B1	10
C1	100
C2	200
D1	1000
D2	2000
D3	3000

(a)

Output1
Node
A1
B1
C1

(b)

Result1	
Node	Result
A1	6310
B1	310
C1	100

(c)

Input2			
Node	ID	Weight	Value
C1	'C1'	0.4	100
C2	'C2'	0.6	200
B1	'B1'	0.5	10
D1	'D1'	0.8	1000
D2	'D2'	0.2	2000
C3	'C3'	0.25	NULL
D3	'D3'	1.0	3000
C4	'C4'	0.75	NULL
B2	'B2'	0.5	NULL
A1	'A1'	NULL	NULL

(d)

x
100
200
310
1000
2000
3000
3000
3000
6000
6310

(e)

Figure 2.4: Example tables. — (a/b) input/output nodes for binary grouping; (c) result of a bottom-up rollup based on Input1; (d) combination of Hierarchy1 and Input1 for unary grouping; (e) result of a bottom-up rollup based on Input2

(like table Input1). Not all nodes potentially carry an input value, and not all nodes will be equally interesting in the result. Consider an analytic scenario as in § 2.2.3, with a fact table of sales data and a dimension hierarchy of products and product groups. Here, the computation input could be the (pre-aggregated) sales amounts that are associated with the products (the leaf nodes) via join. A common task in scenarios of this kind is to sum up the revenue of a selection of products—say, "type A"—along the hierarchy and report these sums for certain product groups visible in the user interface—say, the three uppermost levels. This example represents a type of hierarchical computation with two particular characteristics: First, only a subset of nodes carry an input value. We call these the *input nodes*. Second, the set of input nodes is comparatively small and mostly disjunct from the nodes that after the computation carry a result we are interested in. We call those the *output nodes*. As input and output nodes are naturally determined by two separate subqueries, we refer to this type of hierarchical computation as *binary structural grouping*. "Structural" here alludes to the role the hierarchy structure plays in forming groups of tuples: In the top-down case, the group of each node consists of its ancestors; in the bottom-up case, it consists of the descendants. In our example we are computing a simple sum over each group.

In both SQL and relational algebra, binary grouping queries exhibit a join–group–aggregate pattern, with an inner or left outer join on a hierarchy axis such as *descendant*, and subsequent grouping of the outer side. The following example statement computes a bottom-up rollup based on Hierarchy1, given the input nodes Input1 and the output nodes Output1 in Figure 2.4b (but neglecting the edge weights for now):

```
SELECT t.*, SUM(u.Value) AS Result
  FROM Output1 t LEFT OUTER JOIN Input1 u
       ON u.Node = t.Node OR u's node is a descendant of t's node in Hierarchy1
 GROUP BY t.*
```

In our example this yields table Result1 (Figure 2.4c). From an expressiveness point of view, join–group–aggregate statements are a sufficient and to most SQL users a fairly intuitive way of specifying a hierarchical computation. They are not fully satisfactory, though: they lack conciseness, since conceptually a table of tuple pairs representing the group associations must be assembled by hand prior to grouping, and the fact that a top-down or bottom-up hierarchical computation is being done is somewhat disguised. Regarding their evaluation, join–group–aggregate query plans perform acceptably when the aggregation (in the example: `SUM`) is cheap to compute and the set of output nodes is rather small. However, there is a major efficiency issue: for each output node, the computation naïvely sums up the values of *all* matching input nodes, while ideally we would reuse results from previously processed output nodes. In our example, to compute the sum for (A1) we can save a few arithmetic operations by reusing the sum of (B1) and adding just the input values of (D1), (D2), and (D3). To enable such reuse, hierarchy-aware algorithms are needed. The binary grouping operators in our framework process output nodes (i. e., the left join input) in the natural top-down or bottom-up order—which ensures each output node is processed after any of its covered nodes—and memorize any results that can be reused for upcoming output nodes. Thereby they overcome the mentioned efficiency issues.

Binary structural grouping is not suitable for all types of hierarchical computations, however. In cases where there is no clear distinction between input and output nodes, *unary structural grouping* is more natural. Unary structural grouping works on a *single* table and inherently yields a result for every tuple. Every node essentially acts as both an input and an output node. This is comparable to a binary grouping query where the output and input nodes and the input values have been fused into one table and a self-join is performed. For example, Input2 in Figure 2.4d combines the nodes of Hierarchy1 with Input1. Since only input nodes carry meaningful values, “output-only” nodes are assigned `NULL` as a neutral Value. Figure 2.4e shows the new column resulting from a unary bottom-up rollup computing the (unweighted) `SUM` on that table. In SQL, there is currently no more direct way to express this than a self-joining join–group–aggregate statement. What we would ideally be able to write is:

```
SELECT t.*, t.Value + SUM(u.Value over all covered nodes u) AS x
  FROM Input2 t with bottom-up unary structural grouping based on Hierarchy1
```

Besides lending themselves well for certain computations, unary grouping queries also afford more powerful types of aggregations: In theory, we can evaluate a *structurally recursive* aggregation formula against the data. Such a formula, when evaluated for a node v in the table (say, (B2) in Input2) is conceptually first evaluated in a recursive manner for its directly covered nodes ((C3) and (C4)), and the evaluation results become available in the computation for v itself. Using structural recursion we can state the above rollup in yet another way:

```
SELECT t.*, t.Value + SUM(u.x over directly covered nodes u) AS x
  FROM Input2 t with bottom-up unary structural grouping based on Hierarchy1
```

Note that the computation for each node does (in the bottom-up case) no longer go over *all* descendants as in our previous binary grouping example; only *direct* input

nodes are included in the sums. For B1 the value x would be computed as $10 +$ (x of C1) + (x of C2), and overall this would yield the same result column x as in Figure 2.4e. Finally, we can now express the weighted rollup of our opening example by simply multiplying in the Weight within the SUM expression.

Thus, unary structural grouping combined with structurally recursive aggregation expressions goes way beyond what basic join–group–aggregate statements are able to express. However, it requires the system to "know" the hierarchy structure, so it can apply the expression in a bottom-up manner. Given this ability, enhancing SQL by a corresponding syntax turns out to be surprisingly straightforward, as we will see in § 3. Regarding the evaluation, note that—unlike binary grouping—unary grouping with structural recursion makes reusing the results of covered nodes *explicit* and thus inherently translates into an efficient evaluation approach.

2.2.5 Summary of Requirements and Challenges

Following the discussion of application scenarios in this section, we can now summarize the requirements that a database system needs to fulfill in order to exhibit decent support for hierarchical data.

#1 *Tightly integrate relational and hierarchical data.* First and foremost, any support for hierarchies in databases must harmonize with relational concepts, both on the data model side and on the query language side. Business data today still resides mainly in pure relational tables, and from the examples in this section it becomes clear that queries on hierarchies will routinely need to refer to such "pure" data as well. In particular, it is of major importance that *joining* hierarchical and relational data works in a straightforward and frictionless way.

#2 *Provide expressive and intuitive extensions to SQL.* The language constructs must be expressive enough to support all typical tasks of defining, manipulating, and querying hierarchies. Apart from expressiveness, the resulting statements must be intelligible, so that 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 mechanisms of SQL should be reused or enhanced rather than replaced with new inventions. This not only minimizes training efforts for users who are already familiar with SQL, but also reduces implementation complexity and adoption barriers for existing relational database systems.

#3 *Support explicit modeling of hierarchical data.* When designing a relational database schema from scratch, a database engineer initially has to decide on a data model to represent a hierarchy and in particular to encode its structure—for example, using a self-referencing table as we have seen in § 2.1. Even though relational tree encodings have been thoroughly studied in the literature (cf. § 2.3), carefully choosing and implementing one of them still requires advanced knowledge. What's more, the non-trivial encodings tend to clutter the table schema and disguise the hierarchical nature of the

data. What is required is a way to *explicitly* model a hierarchical table in the schema, using abstract data definition constructs that hide the gory implementation details.

#4 *Enable and facilitate non-trivial queries* involving hierarchies, by offering convenient query language constructs and corresponding backend support. Besides basic queries on hierarchy nodes, we discussed two types of queries in § 2.2.4 that must be supported in particular: structural pattern matching and hierarchical computations.

#5 *Support updates to the hierarchy structure*. Hierarchies in transactional databases are usually *dynamic* and undergo regular updates. In some applications only basic insertions or removals of individual leaf nodes are performed, while in other cases complex operations such as bulk-relocations of arbitrarily large subtrees are required. As an example, consider the asset management scenario of § 2.2.3. In an automotive company, such a hierarchy would contain a large number of machines, robots, and mechanical tools. The assets would need to be relocated in bulk when a new production line is established. In [31] we conducted an analysis of the asset hierarchy of an SAP ERP customer, and found that as much as 31% of the recorded update operations were subtree relocations. The database system must therefore provide a data manipulation interface that supports various types of structural changes—including both leaf and subtree manipulations—, and appropriate backend support for their efficient execution.

#6 *Enforce structural integrity.* Support for manipulating the hierarchy structure goes hand in hand with the requirement of ensuring its integrity. To this end, the system must prevent the user from inserting edges that would violate the strictness properties of the hierarchy (see § 2.2.2). Furthermore, it must ensure that an entity cannot be removed as long as it still has subordinate entities, which would become orphans in the hierarchy.

#7 *Support legacy applications*. In live applications, modeling and maintaining hierarchies explicitly (#3) by designing a green-field database schema or by extending an existing schema is not always an option. In existing databases, hierarchies are necessarily represented in variants of the data models we discussed in § 2.2.1. An important requirement thus is to allow users to take advantage of the extended functionality for hierarchies in an ad-hoc way on the basis of existing data, without forcing them to modify the schema. For that purpose, means to create a *derived* hierarchy from common data formats are required. This process must be supported in the backend by efficient transformation and bulk-building operations.

#8 *Cope with very large hierarchical datasets* comprising millions of entities. All provided language constructs must have strict performance guarantees and allow themselves to be translated into highly efficient execution plans. To achieve high query performance, backend support in the form of sophisticated hierarchy indexing schemes and physical algebra operators is indispensable. The backend also needs to adapt to different scenarios: For hierarchies that are never updated—in particular derived hierarchies (#7)—read-optimized *static* indexing schemes may be used. In contrast, dynamic

scenarios (#5) demand *dynamic* indexing schemes that provide an adequate tradeoff between query and update performance.

A system fulfilling these requirements will enable application engineers to implement all logic for handling hierarchies directly at the database layer, or to push parts of the logic in an existing application down to the database. This promises significant maintainability benefits, makes it easier to ensure the consistency of the hierarchical data, allows for novel kinds of queries on the hierarchies, and enables the applications to handle even large-scale datasets with excellent performance.

2.3 Status Quo: Integrating Hierarchical Data and DBMS

At this point we have a fair understanding of the scenarios we intend to support. In terms of DBMS history, the problem of representing, manipulating, and querying hierarchical data is very old and has been studied many times in the past. We now survey the state of the art with a focus on the off-the-shelf functionality available in today's relational database systems, where hierarchies are stored as basic tables, but we also strive various related areas of database research and technology. We cover the strongest solutions we found and assess them against our requirements stated in § 2.2.5.

2.3.1 The Adjacency List Model

In a relational database, every hierarchy modeled in the application becomes a flat table which encodes the structure in terms of one or more table columns of basic SQL data types. Although there is a lot of freedom to how this can be done, in practice one very frequently encounters the basic self-referencing table model we introduced in § 2.1 (Figure 2.1d), or minor variants thereof. In the literature (e. g., [13]) it is also referred to under the term *adjacency list model*—although that is somewhat misleading, as the resulting table is more comparable to an "edge list." To support accessing tuples by their ID or PID, two (usually B-tree-based) indexes can be defined. Furthermore, to represent a sibling order, a dedicated numeric field is sometimes added. Under the name *edge mapping*, the indexed and ordered variant has for example been applied to storing XML documents in RDBMS (see [36, 39, 40]).

The adjacency list model is perhaps the most effortless approach to store a hierarchy in a table, which explains its popularity. However, one pays for its simplicity when it comes to updating and querying the table. Having two indexes is not economical in space—especially when the keys are of a string type—and costly to maintain (not to speak of the Order field). Regarding queries, the foreign key self-reference only gives us access to the direct parent and children of a node. Anything beyond that is not directly supported and requires iteration or recursion (i. e., a level-by-level search); see the following sections. Due to the many drawbacks, practical guidelines such as [13] and academic works often recommend more sophisticated encodings, which represent the "is below" relationship in a more implicit way along with other information that helps in answering common queries. We will cover many of those in § 2.3.6 and § 4.2.

2.3.2 Recursion in SQL

The structure of hierarchical data is recursive by nature. Thus, queries against a hierarchy sometimes require recursion as well. In particular when the structure is encoded as a trivial adjacency list (§ 2.3.1), even basic tasks—such as determining all descendants of a node—require iteration or recursion. But even with more powerful encodings, the user may wish to evaluate a structurally recursive expression against the hierarchy, as we explored in § 2.2.4.

Not all database systems offer any support for recursive queries. Users therefore often resort to application-level logic as workarounds, which is of course unsatisfying from an expressiveness, efficiency, and maintainability point of view. One mechanism that some systems do provide is the ability to define custom stored procedures, which may use iterative loops or even recursive calls. The following example uses standard SQL/PSM syntax [97] and is adapted from § 2.4.1 of [13]. Starting from a given node, a `WHILE` loop moves up the hierarchy toward the root and performs an action for each encountered node:

```
CREATE PROCEDURE TraverseUpwards (IN n INTEGER) LANGUAGE SQL DETERMINISTIC
WHILE EXISTS (SELECT * FROM Hierarchy WHERE Node = n) DO BEGIN
   CALL AnotherProcedure(n);    -- process the current node
   SET n = (SELECT Parent FROM Hierarchy WHERE Node = n);
END WHILE;
```

Stored procedures are an improvement over application-level code, as they get executed much closer to the actual data they operate on. No per-iteration data transfers between the application layer and the database layer are necessary. However, an unfortunate practical issue is that many database systems use a proprietary syntax for stored procedures, and each offers a slightly different feature set. As in our example, the procedures usually need to be hard-coded against a particular table. Furthermore, they are often executed as a black box, which hinders optimization and creates friction losses. To achieve better performance, it is desirable to have a mechanism that is deeply integrated into the execution engine. We will examine two such mechanisms: Hierarchical Queries (§ 2.3.3) and Recursive Common Table Expressions (RCTEs, § 2.3.4). Unlike stored procedures, RCTEs can straightforwardly be translated into holistic execution plans and are thus (in practical terms) more amenable to optimization. Refer to Ordonez et al. ([82] and [83]) for a recent survey of optimization techniques.

A merit of all three mentioned approaches is that they are general, flexible and computationally powerful in the sense that they can potentially generate entirely new pieces of data and recur on them. (Some authors refer to this as *generative* recursion as opposed to *structural* recursion [30].) Working with adjacency-list-encoded hierarchies (§ 2.3.1) is therefore just one of their many use cases. But the power and generality of recursive stored procedures and RCTEs is also connected to two major drawbacks: They are difficult for the user to write and maintain, and they are comparatively hard for the system to optimize. Also, for the purposes of hierarchical computations (§ 2.2.4), the computational power of generative recursion is less relevant: It turns out that *structural* recursion—where the recursion tree is essentially predetermined, in our case by

the hierarchy itself—suffices for typical tasks. As this concept does not exist in SQL yet, we add language extensions that focus on structural recursion. They enable computations that even go beyond what is practically possible with RCTEs, while minimizing the syntactic overhead. As an added benefit, the simple nature of structural recursion leaves more room for optimizations and efficient evaluation techniques, as we will see in § 5.1.7.

2.3.3 Hierarchical Queries in Oracle Database

Hierarchical Queries are a proprietary SQL extension for "connecting" and traversing recursively structured data. They have been a part of Oracle Database for about 35 years (§ 9.3 of [80]), and a part of IBM DB2 for i since 2001 [45]. A hierarchical query is an extended `SELECT` statement using the constructs `START WITH`, `CONNECT BY`, and `ORDER SIBLINGS BY`. The following example is slightly adapted from [80]:

```
    SELECT ID, LEVEL, SYS_CONNECT_BY_PATH(ID, '/') "Path"
      FROM Employees
     WHERE LEVEL <= 3
     START WITH Name = 'King'
   CONNECT BY PRIOR ID = PID
     ORDER SIBLINGS BY Name
```

The wording of the constructs clearly hints at their intended use for traversing hierarchical data in the adjacency list format. They inform the database engine about the "is below" relationship in the table, the intended roots, and the sibling order. To work with the so-created hierarchy, the projection list may include special pseudo columns such as `LEVEL` and built-in functions such as `CONNECT_BY_PATH` (for obtaining a string representation of the root path) and `CONNECT_BY_ROOT` (for accessing the root row).

The underlying recursion mechanism is conceptually similar to RCTEs. Most functionality of hierarchical queries can be expressed straightforwardly using RCTEs [75], and Oracle Database appears to execute them using a similar technique (a `CONNECT BY PUMP` operator that iteratively adds rows to a union). The following discussion therefore applies to hierarchical queries as well.

2.3.4 Recursive Common Table Expressions

Recursive Common Table Expressions (RCTEs) are a mechanism to compute the transitive closure of a recursively defined table using iterative fixpoint semantics. They were introduced to standard SQL in 1999 [35, 97], and superseded Hierarchical Queries (§ 2.3.3) and several previous proposals for recursion support such as Recursive Union (also critiqued in [35]). While RCTEs are general and powerful in theory, an approach relying only on the adjacency list model and RCTEs to query hierarchies would leave a lot to be desired, as we will see. The following example shows how basic navigation works. It starts at node (A1) of the hierarchy in Figure 2.1d (p. 8) and produces an (unordered) list of its descendants:

```
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 2.5: A structural pattern matching query, expressed using two RCTEs.

```
WITH RECURSIVE RCTE AS (
   SELECT * FROM Entity WHERE ID = 'A1'
    UNION ALL
   SELECT v.* FROM RCTE u INNER JOIN Entity v ON v.PID = u.ID
) SEARCH DEPTH FIRST BY ID SET Ord
SELECT * FROM RCTE ORDER BY Ord
```

The result includes all fields of the original Entity table, plus a generated column Ord, which can be used to arrange the rows in depth-first order. The de facto standard technique to evaluate an RCTE is the *semi-naïve algorithm* described in [35]: It first evaluates the non-recursive part of the `UNION`, then iteratively evaluates the recursive query expression and the `UNION` operation until no new rows are generated, which means a fixpoint has been reached.

To see RCTEs in action in a less trivial example, we revisit the pattern matching query from § 2.2.4, which determines combinations of three nodes (e, r, c) in a particular hierarchical relationship. Figure 2.5 shows the RCTE-based solution. It includes the ID and the Payload of each of the three nodes in the result. The statement uses two RCTEs: one starting from an engine e and navigating downwards from e to a rotor r, the other navigating upwards from r to a compound c. This example clearly conveys the major downsides of an RCTE-based approach: The statements are convoluted and tedious to write down, and not intelligible to their readers. In our examples, it is not immediately obvious that a hierarchy is being traversed, or which direction the traversal proceeds in. Seemingly basic tasks are surprisingly difficult to express (let alone evaluate): In order to determine basic node properties such as the depth of a node, the user must manually specify the computation using arithmetics within the RCTE. As a consequence of the trivial adjacency list model, structural patterns other than the basic *descendant* and *ancestor* axes are not supported.

The reason why RCTEs become bulky very quickly is that they combine three separate tasks into one statement: First, imposing a hierarchy structure onto the base table (by specifying a join condition that identifies the relevant ID and PID fields); second, "discovering" the hierarchy nodes on the fly by navigating it via joins; and third, computing the actual data of interest. These tasks are inseparably intertwined.

In addition, there are two less obvious issues that render RCTEs virtually infeasible when it comes to hierarchical computations: First, to enable semi-naïve fixpoint evaluation (i. e., to guarantee *stratification*), several SQL constructs are disabled within the recursive query expression of an RCTE (see [35] for details). In particular, `GROUP BY` is forbidden, as aggregation crossing recursion can violate monotony in subtle ways. The `GROUP BY` therefore has to be placed outside of the RCTE definition, which essentially results in a join–group–aggregate statement with a very expensive recursive join. Second, RCTEs inherently require any hierarchical computation to be phrased in an *iterative* way. This means that, even if we ignore any concerns about readability and performance, it is virtually impossible to use RCTE-based recursion with `GROUP BY` to achieve the semantics of structural grouping, in particular the *bottom-up* case. Such a computation would have to be stated in a bottom-up iterative way, starting at the "lowermost" tuples in the input, then sweeping breadth-first over the input by iteratively joining in the "next higher" tuples. However, neither "lowermost" nor "next higher" can be reasonably expressed with the adjacency list model (and would require unwieldy `EXISTS` subqueries even with more powerful encodings). Even if that were possible *and* `GROUP BY` were allowed within the RCTE, the grouping would not necessarily capture all relevant covered nodes within a single iteration: In an irregular hierarchy the "lowermost" nodes will already be on different levels, to begin with.

What we have discussed so far are mainly issues of expressiveness. When it comes to performance, an RCTE-based approach also bears some inherent inefficiencies:

- Repeated self-joins can be expensive. Each join can potentially produce a large intermediate result, and the overall number of iterations depends on the height of the hierarchy. (Our evaluation in § 6 quantifies this further.)
- Often, attributes of interest to the user (Payload in our example) must be materialized early and carried along through the recursion, which is costly.
- The RCTE necessarily has to navigate over all intermediate nodes between the actual nodes of interest. In the example of Figure 2.5, we are interested only in qualifying ancestor/descendant pairs (e, r) and (r, c), but not in any nodes in between. Touching the intermediate nodes would violate our requirement of § 2.2.4 that the query runtime should not depend on the number of intermediate nodes; it should ideally be linear in the number of e, r, and c candidates.

Finally, RCTE statements have an *imperative* aspect that violates SQL's declarative nature. Consider again Figure 2.5. An alternative solution to the same problem would be to use a different join order: start with r, then navigate to e and c from there. Yet another option would be to first materialize all possible ancestor/descendant combinations (u, v) using a single RCTE, and then use two non-recursive joins on the resulting

table to match the pattern. For large hierarchies this option is usually inferior due to the larger intermediate result. The point is that it is up to the user to choose the most appropriate strategy for answering the query, and the query optimizer is tightly constrained by that choice. This can easily result in severe performance penalties.

All in all, we do not believe RCTEs to be a practical solution for our use cases. Writing RCTEs is an "expert-friendly" and error-prone task in terms of achieving correctness, intelligibility, and robust performance. As a general mechanism, RCTEs do have interesting uses beyond traversing hierarchical data, and we do not intend to render them obsolete. That said, applications relying on our framework will never have to resort to RCTEs for typical tasks.

2.3.5 Hierarchical Computations via `ROLLUP`

In Figure 2.2d (p. 11) we showed a common data model for hierarchies that is based on a denormalized table. This model is quite restrictive in that it accommodates only homogeneous and balanced hierarchies. Nevertheless, it seems to be dominant today for modeling dimension hierarchies in data warehouses [93, 100]. The Common Warehouse Metamodel [23] refers to this model under the term *level-based* hierarchy.

Indeed, there is a particular use case where tables of this type shine: bottom-up hierarchical rollups computing simple aggregations such as `COUNT` and `SUM`. This is a routine task in analytic applications (see § 2.2). Computations of this type can easily be expressed on level-based tables using SQL's basic grouping mechanisms `GROUP BY` and `GROUPING SETS`. SQL in fact has another feature that is targeted at exactly this scenario: the `ROLLUP` construct [38, 81, 97]. Given a sales table with a hypothetical geographic dimension hierarchy as outlined in § 2.2.3, we could for example write:

```
SELECT Country, State, Region, SUM(Sale)
  FROM Sale
 GROUP BY ROLLUP (Country, State, Region)
```

Technically, the construct is merely syntactic sugar for `GROUPING SETS`. A `ROLLUP` with k dimensions $(C_1, \ldots, C_k)$ is equivalent to a `GROUPING SETS` clause with $k+1$ grouping sets, corresponding to all prefixes of the sequence:

```
GROUP BY GROUPING SETS ( (C_1,...,C_k), (C_1,...,C_{k-1}), ..., (C_1,C_2), (C_1), () )
```

Thus, it does not enable any computations that are not possible otherwise. A merit of using the existing grouping operators is that they tend to be heavily optimized in commercial DBMS. If the aggregates are algebraic or distributive [38], they are also well-suited to parallelization, and the results for higher levels can be derived from the lower levels. Rollup computations can therefore yield high performance.

On the other hand, there are tight constraints on both the structure of the underlying hierarchy and the class of computations that can be expressed. When the hierarchy is inhomogeneous or exhibits an irregular structure—where the depth is unbounded and nodes on a level may be of different types—, or when the computations are more involved than simple rollups, the approach fails. Many of the scenarios of § 2.2.3 do

indeed feature such complex hierarchies. We therefore believe that, while ROLLUP is attractive due to its simplicity and performance, users would welcome less restrictive data models and enhanced functionality for hierarchical computations. Our framework is able to provide both.

2.3.6 Encoding Hierarchies in Tables

In the previous sections we have seen the shortcomings of approaches based on the adjacency list model (§ 2.3.1) or the level-based model (§ 2.3.5). Some of these shortcomings can be overcome by representing the hierarchy structure more cleverly in the database schema. Such representations, which rely entirely on the means of the relational model and standard SQL (as opposed to specific engine support), are commonly referred to as *labeling schemes*. In general, a labeling scheme represents a hierarchy primarily as a set of nodes, and attaches to each node a unique *label*. A label is a fixed number of data items—in our setting, table fields. The edges are not represented explicitly. Labeling schemes have been studied extensively [2, 7, 12, 41, 44, 46, 48, 60–63, 73, 79, 99, 113, 115, 118]. Especially in the context of XML, a large body of relevant research exists (see also § 2.3.9). We can categorize the various approaches as follows:

- **Naïve labeling schemes** use trivial labels. An example is the adjacency list model, where the label is simply the key of the parent. They are easy to implement and thus widespread in practice. However, in comparison to "full-blown" labeling schemes they do not provide sufficient support for queries and updates.
- **Containment-based labeling schemes**, also known as *order-based* schemes, label each node with a [lower, upper] interval or similar values. As the term "containment" alludes to, their main property is that the interval of each node is contained in (or *nested* into) the interval of its parent node.
- **Path-based labeling schemes**, sometimes called *prefix-based codes*, label each node with the full path from the root to the node using variable-length labels.

Figure 2.6 shows three examples: (a) is a simple and often-cited *nested intervals* labeling [44, 48, 63, 104, 105, 114, 118]. We can see that node D3 is a descendant of node A1, because its interval $[16, 17]$ is a proper subinterval of $[1, 20]$. Example (b) shows the conceptually similar *pre/post* labeling scheme, as for example studied in [39–41]. Here each node is labeled with its preorder and postorder ranks. Example (c) shows a *Dewey* labeling [99], which is perhaps the most basic path-based scheme. It builds upon the sibling rank, the 1-based position of a node among its siblings. A label consists of the label of the parent node, a separating dot, and the sibling rank. In the example, the Dewey label of C4 is 1.2.2, as it is the second child of B2, which is the second child of A1, which is the first root. Note how all three schemes naturally represent the sibling order. We will study further types of labeling schemes in § 4.2. For now, we are mainly interested in their basic properties.

In general, all *queries* against the hierarchy are answered by considering only the labels. With containment-based schemes, this means testing whether the intervals of the involved nodes overlap (a constant-time operation); with path-based schemes, this means decomposing and comparing the variable-length path strings. Labeling schemes obviate the need for recursion to perform basic axis checks, which is a big gain in usability over the naïve adjacency list model. For example, we can straightforwardly translate the abstract "v is a descendant of u" condition of § 2.2.4 into SQL:

Nested Intervals:	`v.lower > u.lower AND v.upper < u.upper`
Pre/Post:	`v.pre > u.pre AND v.post < u.post`
Dewey:	`u.path` *is a substring of* `v.path`

Finding and enumerating the labels can be further supported by defining B-tree or hash indexes over the label columns. *Updates* to the hierarchy structure are generally executed by *relabeling* the affected nodes and updating the indexes accordingly.

It is a major challenge to design a labeling scheme that supports all anticipated operations well while balancing the classic tradeoff between query and update performance, especially when considering large hierarchies and high rates of non-trivial update operations in potentially skewed patterns. We cover how state-of-the-art labeling schemes attempt to tackle this challenge in § 4.2. In any case, solutions based on labeling schemes are always rather specialized. They need to be carefully chosen and manually implemented for the application scenario at hand. The choice largely dictates (or restricts) the types of hierarchies that can be represented and the types of queries and updates that can be expressed conveniently. As this is a non-trivial task, entire books such as [13] have been written to provide recipes to SQL developers. They usually come with sets of hard-wired SQL snippets that implement particular schemes and their supported query and update operations.

Ideally, we want the database system to take over these tasks for us. Our approach therefore is to provide an abstract data type that hides the underlying encoding details completely. It guarantees a decent feature set regardless of the actual representation, and thus relieves the user from dealing with the complexities and limitations of a custom implementation. At the same time, the database kernel is given the means to employ special-purpose algorithms and data structures, which allow it to easily outperform most hand-implemented "pure" labeling schemes.

2.3.7 `hierarchyid` in Microsoft SQL Server

The built-in `hierarchyid` data type introduced in SQL Server 2008 [70, 78] is a comparatively recent example of an RDBMS vendor investing into enhancements to simplify working with hierarchical data. The hierarchyid type represents a path string. A collection of hierarchyid values can thus be used to represent an ordered tree. A node can effectively be inserted and relocated by assigning the row a new hierarchyid value. Such values can be generated from a string representation (e. g., `'/2/1/1/'`) or from existing values using methods such as GetReparentedValue and GetDescendant. For

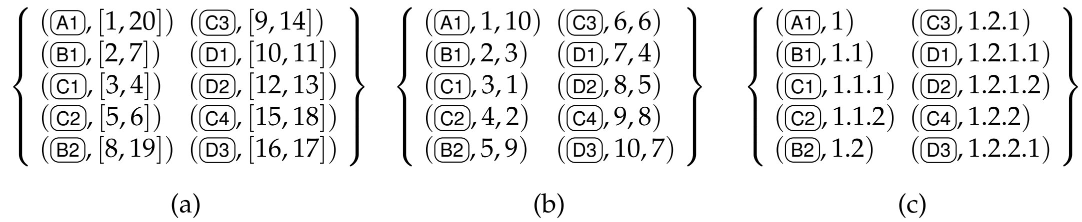


Figure 2.6: Example labeling schemes representing the hierarchy of Figure 2.2a.

querying nodes, the data type provides self-explanatory methods such as GetLevel, IsDescendantOf, GetAncestor, CommonAncestor, GetRoot, and conversions from and to strings and VARBINARY. The following example procedure lists all nodes below (B2):

```
DECLARE @v hierarchyid
SELECT @v = Node FROM Hierarchy WHERE ID = 'B2';
SELECT *, Node.ToString() AS Position, Node.GetLevel() AS Depth
  FROM Hierarchy
 WHERE Node.IsDescendantOf(@v) = 1;
```

To speed up queries, the reference manual [70] recommends users to add an ordinary B-tree index to maintain the rows in depth- or breadth-first hierarchyid order, and a unique index to guarantee the hierarchy actually forms a tree. Overall, this design is comparable to our hierarchical table model from a syntax and data model perspective. However, there are also several major differences:

First, hierarchyid is basically a plug-in based on SQL Server's Common Language Runtime (CLR). To our knowledge, it does not come with deeper SQL language and engine integration. We go a step further by employing special-purpose hierarchy indexes instead of ordinary B-tree indexes, as well as physical algebra operators integrated into the execution engine.

Second, the hierarchyid type is provided as a simple tool for modeling a hierarchy; yet, a table with such a field does *not* necessarily represent a valid hierarchy. 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, so multiple rows could represent the same node. It also does not require the direct parent of a node to exist, so one could accidentally "orphan" a subtree by deleting its parent. It is up to the user to generate and manage the values in a reasonable way. In contrast, we are more stringent in that regard and guarantee structural integrity at any time (Requirement #6 of § 2.2.5). Of course, this design choice comes at a price, as consistency has to be checked on each update, but it helps to avoid surprising results from queries. Our indexes make these consistency checks comparatively cheap.

Third, as another difference, we opt to provide flexibility regarding the underlying indexing scheme. The encoding scheme used behind the hierarchyid type is Ordpath [79], a path-based scheme. It is hard-wired into the data type; in particular, conversions from and into a path string are directly supported, which would be hard to

efficiently compute with encoding schemes that are not path-based. While it is claimed to be "compact and update-friendly" [79], Ordpath has a few inherent deficiencies that are common to all path-based schemes: First, to relocate a subtree one has to update *all* hierarchyid values in that subtree. Second, the representation is variable-length (and allowed to grow up to 892 bytes), which makes hierarchyid values less compact and processing-friendly than certain alternative schemes. Due to these reasons, our philosophy is to avoid manifesting a particular encoding scheme into the design; we rather intend the scheme to be chosen flexibly according to the application scenario at hand.

2.3.8 `ltree` in PostgreSQL

PostgreSQL has a module called `ltree` for representing "hierarchical tree-like structures" [87]. It comes with a data type ltree that stores a so-called *label path*, much comparable to the hierarchyid type discussed in the previous section. The module also comes with a general path pattern matching language, which is needed to formulate queries against a hierarchy. For example:

```
SELECT *, NLEVEL(PATH) AS Depth FROM Hierarchy WHERE Path <@ 'A1.B2';
```

Also noteworthy is the use of special indexes that fit into the framework of Generalized Search Trees (GiST, [47]). Thus, ltree goes a step further in terms of language syntax and backend support when compared to hierarchyid. Apart from that, most of the remarks we made in the previous section also apply to ltree.

2.3.9 XML Databases and SQL/XML

In the late 1990s the Extensible Markup Language (XML) [108] became popular as a versatile format for data transport. There were high hopes that it would also lend itself well to data storage and access. To allow users to conveniently extract nodes and data from XML documents, the powerful XPath [110] and XQuery [111] languages were developed. The existing database products around these technologies can roughly be divided into two classes: On the one hand, there are "native" XML stores, whose internal physical structures were designed from the ground up for XML. On the other hand, there are systems that rely on relational databases as the backend for storing XML fragments and executing queries by translating XPath and XQuery to SQL [6, 42]. As documents written in XML are inherently hierarchical, the latter class is particularly interesting in our context. To speed up the resulting queries, suitable RDBMS-based XML encodings and indexing schemes have been studied, e. g. [1, 7, 22, 24, 36, 39–43, 56, 63, 66, 67, 72, 79, 95, 99, 102, 118]. If one ignores some XML specifics, much of this work is applicable to our setting as well. We therefore thoroughly survey many of the proposed indexing techniques in § 4.2.

Another, more invasive approach is to turn existing RDBMS into XML-enabled systems by "fusing" the two data models together and integrating XML support directly into SQL. This idea resulted in the SQL/XML standard [98] and has been implemented by prominent vendors [4, 64, 84]. SQL/XML enables queries over both tables *and* XML

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

Figure 2.7: Using SQL/XML to generate XML fragments from relational data (top, from [64]), and to evaluate XQuery on an XML fragment, producing a mix of relational and XML data (bottom, from [28]).

documents, and as such is clearly the tool of choice for working with XML in a relational context. One may therefore ask whether this technology can be leveraged for our scenarios by representing hierarchies as XML fragments, and whether the design of the XML-enabled SQL dialect can serve as a blueprint for our language extensions.

To this end, consider Figure 2.7, which shows the interaction between relational and XML data by means of two example queries. The top example illustrates how so-called publishing functions enable the user to convert relational input data into XML fragments. Note how a single field of type `XML` can store a complete hierarchy of XML nodes. This is a major difference to our approach, which associates one table row with one node. The bottom example illustrates how XQuery is used within SQL statements to extract data from an XML fragment and produce either XML or a relational view. Note the heavy "machinery" for passing values in and out of the query.

Without going into details, these two examples clearly show that SQL/XML makes no attempt to bridge the duality between the two types of data. It requires users to know both models and the respective query languages, which is a challenge to SQL-only users. Furthermore, converting data from relational to XML or vice versa induces a lot of overhead. This overhead is not only "syntactic," as the many `XML...` clauses cluttering Figure 2.7 attest; there is also a significant runtime cost. In summary, the heavyweight approach of drawing in the complete technology stack of XML, XML Schema, XPath, and XQuery into the database would contradict our requirements #1 and #2, which mandate that the data model and query language must blend seamlessly with SQL. We therefore cannot consider SQL/XML as a fruitful design blueprint.

2.3.10 Multidimensional Databases and MDX

The multidimensional data model is a variation of the relational model that organizes data as a collection of multidimensional data cubes. Its applications are mainly in the business intelligence tools we already mentioned in § 2.2.3, where the cubes are sliced-and-diced and aggregated along their dimensions and presented in visual or crosstab-style reports. To produce the data underlying these reports, Multi-Dimensional Expressions (MDX) is often used, a declarative query language for data cubes. It is a proprietary technology by Microsoft [71], who first released it in 1997 as part of their OLE DB for OLAP specification, but has been embraced by a majority of OLAP products since. For a general overview of BI technology and multidimensional database concepts, refer to [16], [54], and the survey articles by Jensen and Pedersen [50, 86]. In this section we are mainly interested in multidimensional databases and MDX for the primary role they give to hierarchical dimensions, and how they can inspire us in extending SQL for hierarchies. As a simple example, assume a dimension [Store] for which a hierarchy [Location] of three levels has been defined, referred to in the path-like notation of MDX by:

[Store].[Location].[Continent]   [Store].[Location].[Country]   [Store].[Location].[City]

Having defined this upfront, one can output the full hierarchy as a tree by simply "selecting" it, much like one would select a column in SQL:

```
SELECT [Measures].[Sales Amount] ON COLUMNS,
       [Store].[Location].Members ON ROWS
  FROM [Sales Cube]
```

This already associates the nodes with the corresponding sums of sales amounts. To demonstrate further features related to hierarchies, Figure 2.8 shows a less trivial query taken from the MDX Language Reference [71]. The Adventure Works cube has a Customer dimension, which in turn has a geographic hierarchy arranging the customers. In the `SELECT` part, the query navigates the geographic hierarchy using the Descendants() function: It determines the descendants of the Australia node on the State-Province level, that is, all state-provinces in Australia. This set of members is placed on the rows of the report. In the first column, the Internet Sales Amount for each province is displayed. In the second column, the [Measures].X calculation is displayed. This calculation computes the percentage of the Internet Sales Amount in a State-Province relative to the aggregated total Internet Sales Amount in the country (i. e., Australia). How the percentages are computed is specified in the `WITH` clause. The Ancestors() function determines the Country to which CurrentMember belongs; the query execution engine will successively bind CurrentMember to each State-Province value in the report rows. Item() is needed for technical reasons: it extracts the first (and only) tuple from the set returned by Ancestors(). Although this example (and arguably MDX as such) is not exactly easy to understand, we can learn several things:

- MDX works on a rich data model that has been specified upfront using some proprietary tooling (e. g., XML for Analysis [109]). It is a language for querying

```
WITH MEMBER [Measures].X AS
   [Measures].[Internet Sales Amount] / ([Measures].[Internet Sales Amount],
      Ancestors
         ([Customer].[Customer Geography].CurrentMember,
          [Customer].[Customer Geography].[Country]
         ).Item(0)
   ), FORMAT_STRING = '0%'
SELECT
   {Descendants(
      [Customer].[Customer Geography].[Country].&[Australia],
      [Customer].[Customer Geography].[State-Province],
      SELF)} ON ROWS,
   {[Measures].[Internet Sales Amount], [Measures].X} ON COLUMNS
FROM [Adventure Works]
```

Figure 2.8: An example MDX query

only; it supports neither data definition nor data manipulation, whereas our SQL extensions cover these aspects as well.

- MDX attempts to look like SQL, but its mechanics work quite differently. To SQL users its behavior, such as the fact that aggregations of the numeric measures in the cube happen implicitly, may come as a surprise. While MDX makes it possible to express multidimensional hierarchical aggregations very concisely, it deviates too far from the SQL look and feel for our purposes.

- Hierarchies (and their levels) are named first-class objects, and can be handled much like an ordinary column in SQL. Again, this is a step too far for our purposes. Our goal is to treat hierarchies as ordinary database tables instead of as distinct objects. Also, a hierarchy is treated as a fixed set of homogeneous levels, and this limitation is deeply built into the language. As already noted in § 2.2, we want to allow for more flexibility regarding the hierarchy structure.

- The Ancestors() and Descendants() functions used in the query are part of an extensive set of functions for hierarchy navigation that can be used in query statements and calculations. The general approach to navigation is to apply *set-valued* functions to *sets* of nodes. This functional style contradicts the declarative nature of SQL. Our preferred way to express navigation is an ordinary SQL join (see § 2.2.4).

We thus conclude that the concepts of MDX, although promising upon first sight, are mostly inapplicable to our setting.

3 A Framework for Integrating Relational and Hierarchical Data

This chapter covers our framework for supporting hierarchical data in relational database systems. Based on its core concepts of a hierarchical table and an abstract `NODE` data type (§ 3.2), we describe language constructs for querying hierarchies (§ 3.3), for defining new hierarchies or deriving them from existing data (§ 3.4), and for updating their structure (§ 3.5). The language elements are designed to cover typical requirements and blend seamlessly with the look and feel of SQL, which we illustrate on some advanced scenarios (§ 3.6).

3.1 Overview

Before delving into the details, we outline the central concepts of our framework and demonstrate its language features by some simple examples. As there is virtually no application that consists *solely* of hierarchical data, our central design goal has been to seamlessly blend all functionality into existing SQL concepts. In particular, we refrain from treating hierarchies as a distinct type of object in the database schema, as this would counteract the idea of a frictionless interaction between hierarchical data and coexisting tables. Instead, we tie hierarchies very closely to an associated table that contains the actual data that is hierarchically arranged. This leads us to our core concept of a *hierarchical table*, which is a table containing at least one column of the abstract data type `NODE`. In a nutshell, a field of this type represents the position of the row in the hierarchy. It is backed by an index that represents the structure, but these details are completely hidden from the user. The user can create a hierarchical table either from scratch or *derive* such a table from existing data. For example, the bill of materials table from the opening chapter (Figure 1.1, p. 2) can be transformed into a hierarchical table based on the information in its ID and PID fields. This is a one-time process, after which the system becomes fully aware of the hierarchy structure: The resulting `NODE` column and its accompanying index can be cached or even persisted and then leveraged in all subsequent queries on the same table.

From the user's perspective, all queries against the hierarchy are expressed in terms of the readily available `NODE` field. Our extensions to SQL comprise a small yet essential set of built-in functions that conceptually operate on values of type `NODE`. To see them

This chapter is primarily based on the material published in [9]. The concepts and language constructs for hierarchical computations were previously published in [10].

```
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.Node, e.Node)
   AND r.Kind = 'rotor'
   AND IS_ANCESTOR(c.Node, r.Node)
   AND c.Kind = 'compound'
```

Figure 3.1: The query from Figure 2.5, expressed in terms of our SQL extensions.

in action, let us first consider structural pattern matching. In § 2.2.4 we implemented a non-trivial example query as a recursive common table expression (Figure 2.5, p. 25). Figure 3.1 shows the query in our syntax. This example exhibits three cornerstones of our design: First, there is a clear separation between the task of creating or deriving a hierarchy on the one hand, and the task of actually querying it on the other hand. In contrast, the RCTE necessarily "explores" the hierarchy structure through cumbersome, user-prescribed joins on the ID and PID fields, and queries it at the same time; the two aspects are inseparably intertwined. Second, unlike the general-purpose facilities for recursion (see § 2.3.2), our syntax is tailored for working with hierarchies, as the two used predicates `IS_DESCENDANT` and `IS_ANCESTOR` attest. This specialization allows us to increase user-friendliness and expressiveness, as well as to use special-purpose data structures and algorithms at the backend. Third, hierarchical relationships such as "descendant of" are stated in a direct and *declarative* way. The recursive nature of a hierarchy is mostly hidden, so the user does not have to craft a recursive solution. This again benefits the backend, as the query optimizer can reason about the user's intent and pick an optimal evaluation strategy (i. e., join direction). Finally, the example also shows how we meet the requirements #1, #2, and #4 stated in § 2.2.5: Our syntax blends with SQL (#1), as we stick mostly to joins and built-in functions to provide the required query support (#4). As a corollary, the syntactic impact—in terms of required extensions to the SQL grammar—is low (#2). Still, the syntax is highly expressive (#2): the query in Figure 3.1 reads just like the English sentence defining it.

Structural Grouping. A significant part of our language extensions is dedicated to rendering *structural grouping* as convenient as possible. § 2.2.4 motivated such queries from a high-level point of view. We saw that binary structural grouping translates into join–group–aggregate queries that combine a join on a hierarchy axis (as in Figure 3.1) with a subsequent `GROUP BY` of the outer side. This way, a useful class of hierarchical computations can be expressed fairly intuitively out of the box. The following example corresponds to the analytic query of § 2.2.4, which sums up the revenue of "type A" products bottom up and reports the sums for the three uppermost levels:

```
WITH Input1 AS (                                                    I-a
   SELECT p.Node, s.Amount AS Value
     FROM Hierarchy1 p JOIN Sale s ON p.Node = s.Product
    WHERE p.Type = 'type A' )
```

```
SELECT t.*, SUM(u.Amount) AS Total
  FROM Hierarchy1 t LEFT OUTER JOIN Input1 u ON IS_DESCENDANT_OR_SELF(u.Node,t.Node)
 WHERE DEPTH(t.Node) <= 3
 GROUP BY t.*
```

However, we already noted a lack of convenience and conciseness in such join–group–aggregate queries. The user has to assemble a table of tuple pairs representing the group associations by hand prior to grouping. This disguises the fact that a top-down or bottom-up hierarchical computation is being done. It becomes tedious especially when the output and input nodes largely overlap or are even identical, as in:

```
SELECT t.Node, SUM(u.Value)                                                    II-a
  FROM Input1 AS t
       LEFT OUTER JOIN Input1 AS u ON IS_DESCENDANT_OR_SELF(u.Node, t.Node)
 GROUP BY t.*
```

This query works on only one table, but we have to mention it twice in the query. A small yet effective extension to the windowed table mechanism of SQL will allow us to rewrite such queries into a *unary* structural grouping form:

```
SELECT Node, SUM(Value) OVER (HIERARCHIZE BY Node) FROM Input1                 II-b
```

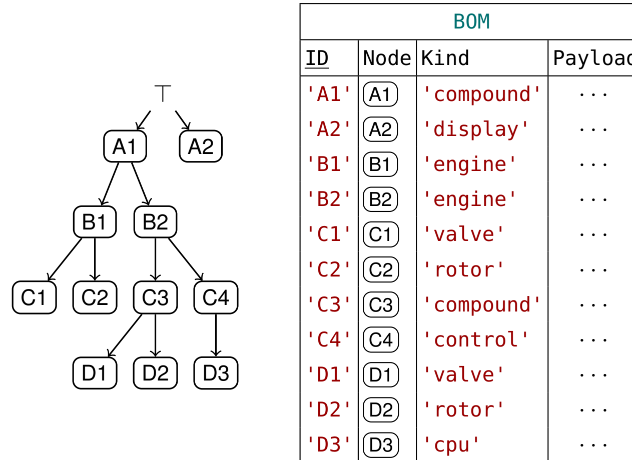
The `HIERARCHIZE BY` clause forms windows over the current table in a hierarchical way, which in this case achieves exactly the desired bottom-up rollup semantics. In general, rewriting binary to unary structural grouping queries will often result in more concise and intuitive statements. But beyond that, the same mechanism offers us another attractive language opportunity: support for structural recursion. Using a structurally recursive SQL expression we can state the rollup in statements II-a and II-b from above in yet another way:

```
SELECT Node, RECURSIVE INT (Value + SUM(x) OVER w) AS x                        II-c
  FROM Input1 WINDOW w AS (HIERARCHIZE BY Node)
```

The expression for x works just like in our original pseudo code from § 2.2.4: it sums up the readily computed sums x of all tuples that are covered by the current tuple. Besides allowing us to state certain binary hierarchical computations (which are in principle already supported in SQL today) in a simpler and more direct way, unary grouping also enables us to state significantly more complex computations based on structural recursion with remarkable conciseness. For example, we can now straightforwardly take the edge weights from Input2 (Figure 2.4d, p. 18) into account in our rollup:

```
SELECT Node, RECURSIVE DOUBLE (Value + SUM(Weight * x) OVER w) AS x            III
  FROM Input2 WINDOW w AS (HIERARCHIZE BY Node)
```

Such computations could not be expressed (let alone evaluated) convincingly in the past. With our hierarchical windows they appear very natural.



BOM			
ID	Node	Kind	Payload
'A1'	A1	'compound'	···
'A2'	A2	'display'	···
'B1'	B1	'engine'	···
'B2'	B2	'engine'	···
'C1'	C1	'valve'	···
'C2'	C2	'rotor'	···
'C3'	C3	'compound'	···
'C4'	C4	'control'	···
'D1'	D1	'valve'	···
'D2'	D2	'rotor'	···
'D3'	D3	'cpu'	···

Figure 3.2: The bill of materials table from Figure 1.1 as a hierarchical table.

3.2 The Hierarchical Table Model

3.2.1 Hierarchical Tables

A *hierarchical table* is a table with at least one *hierarchical dimension*. A hierarchical dimension is a strict hierarchy that is associated with the table and arranges its tuples. The hierarchy conceptually does not contain any data besides the structural information and the node–tuple associations. A table may have multiple hierarchical dimensions, but a hierarchy can have exactly *one* associated table. (If additional tables need to be "tied" to a hierarchy, ordinary foreign key references to the hierarchical table can of course be used; see § 3.6.) Table BOM in Figure 3.2 is an example; it has only one associated hierarchy.

In our model, the topology of a strict hierarchy is an *ordered, rooted, labeled tree* as defined in § 2.1. Much of the functionality we describe in this chapter thus does not (and cannot) apply for non-strict hierarchies. To still support such data to some degree, we provide the means to transform data representing a directed graph into a proper strict hierarchy (see § 3.4.2). Note also that we rely on an *ordered* tree model. The system maintains a deterministic internal order even if no meaningful sibling order is prescribed by the user. This in particular ensures that every node has a well-defined rank in the preorder or postorder sequence of all nodes; for example, B1 in Figure 2.3 always has pre rank 2 and post rank 3. These ranks enable certain order-based language features, and they are the foundation for the evaluation techniques we discuss in § 5.

A single hierarchical table may be used to store multiple logical hierarchies of the same type. For example, the BOM table could store *all* bills of materials that have ever been entered into the system. We therefore do not require the hierarchy to form a single connected tree. This means there may be multiple root nodes. To avoid complications in handling such forests (as well as issues with empty hierarchies), we always maintain

a single, virtual root node, which we denote by ⊤ and call the *super root*. The actual roots of the individual hierarchies in the user data become the children of ⊤. This way, the hierarchy can be represented as a connected tree internally, but will appear to the user—who never sees the virtual ⊤ node—as a forest.

Let us now consider the tuple–node relationship, given a hierarchy H attached to a table T. We require each tuple of T to be associated with *at most one* node of H. Thus, some tuples may not appear in the hierarchy. Conversely, we require each node ν except for ⊤ to be associated with *exactly one* tuple t of T. Logically, a node is not associated with any further data: The *label* of node ν consists exactly of the fields of t. Although we associate the label with the node, recall from § 2.1 that for trees a 1 : 1 association between a node and its incoming edge can be made. Each field value can thus be interpreted as a label on either the node ν or the edge onto ν. This is up to the application. For instance, in Figure 2.3 (p. 17) we viewed the Weight column of table Hierarchy1 as an edge label and the ID column as a node label.

A user never works with the hierarchy object H itself, only with the associated table T. Consequently, a "syntactic handle" is required to enable the user to refer to a node in H given the corresponding tuple. Such a handle is provided by an attribute of type `NODE` in T, whose name can be chosen freely by the user.

3.2.2 The `NODE` Data Type

A field of the built-in data type `NODE` serves as the primary handle to a specific hierarchical dimension H. Intuitively, a `NODE` value represents the position of the associated node of a tuple in that hierarchy. For our example table BOM with its `NODE` field named `Node`, this would look as follows:

```
SELECT h.ID, ..., "depth of" h.Node FROM BOM AS h WHERE h.Node "is a leaf"
```

These node handles are a cornerstone of our design. There are in fact other conceivable approaches to making such a handle available in the query language, such as exposing built-in pseudo columns (similar to Hierarchical Queries in Oracle, see § 2.3.3), operating on the table alias `h` itself (a style mandated by early proposals for temporal SQL [25]), or operating on explicit named hierarchy objects. However, after careful consideration, we found that the proposed design allows us to expose all desired functionality in the most natural way while incurring the least "syntactic impact." It also simplifies some technical aspects: Transporting "hierarchy information" across a SQL view becomes 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 operators and functions on the `NODE` type.

Similar to an abstract data type, we want the actual values of the `NODE` data type to be opaque to the user. To prevent them from being observed directly, a naked `NODE` field may not be part of the output of a top-level query. The user may think of a `NODE` value as a position in a hierarchy, but how this position is represented is not relevant to the user. (In fact, even at the backend it can mostly be treated as `BINARY` data of variable-

length size; the actual properties of the type and its representation and manipulation are ultimately up to an underlying indexing scheme, as we discuss in § 4.)

The user works with a `NODE` field mostly 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.

`NODE` values can be `NULL` to express that a tuple is *not* part of the hierarchy. A non-null value always encodes a valid position in the hierarchy. The handling of `NULL` values during query processing must be consistent with SQL semantics. For example, a predicate such as "is ancestor of" applied to two `NODE` values where at least one value is `NULL` must evaluate to `UNKNOWN`. Likewise, grouping (via `GROUP BY`) must treat `NULL` nodes as equivalent and place them into one group.

We already noted that each `NODE` field is associated to a different, specific hierarchy. We provide no way to add more than one `NODE` for the same hierarchical dimension to a hierarchical table. This guarantees a tuple can only appear once in any hierarchy, which is in line with our fundamental hierarchy model. However, a table can very well have two or more `NODE` fields, and thus a tuple can be part of multiple *distinct* hierarchies. Similarly, the user can of course join hierarchical tables together and thus assemble tuples with arbitrarily many different `NODE` fields. In these situations, `NODE` values from *different* hierarchies could potentially be applied to a binary predicate such as "is ancestor of," or even mixed into a single column via set operations such as `UNION`. This is not a feature we support, as such mixing is usually meaningless and likely an error in the program logic. We therefore require the system to prohibit this at query compilation time by tracking the underlying hierarchy of each `NODE` field as part of its type, and raising a SQL error when two different `NODE` types are mixed. Effectively, each hierarchy H has its own *distinct* `NODE` type, although this is not exposed to the user. In the remainder we will occasionally use the notation $\texttt{NODE}^H$ to express the `NODE` type that is specific to hierarchy H.

3.3 QL Extensions: Querying Hierarchies

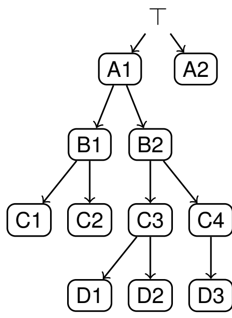
We now cover our enhancements to SQL's query language to support and facilitate complex queries (our Requirement #4). As outlined previously, queries require a `NODE` field that serves as handle to the nodes in the associated hierarchy. For the following, we assume a table with a field ℓ of data type $\texttt{NODE}^H$ is at hand. How to obtain such a table—either a hierarchical base table or a derived hierarchy—is covered in § 3.4.

3.3.1 Hierarchy Functions

To enable the user to directly query commonly required properties of the nodes, we provide some built-in scalar functions that operate on a `NODE` value ℓ:

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

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



Node	IS_LEAF	IS_ROOT	DEPTH	SIZE	DEGREE	HEIGHT	PRE_RANK	POST_RANK
A1	FALSE	TRUE	1	10	2	4	1	10
B1	FALSE	FALSE	2	3	2	2	2	3
C1	TRUE	FALSE	3	1	0	1	3	1
C2	TRUE	FALSE	3	1	0	1	4	2
B2	FALSE	FALSE	2	6	2	3	5	9
C3	FALSE	FALSE	3	3	2	2	6	6
D1	TRUE	FALSE	4	1	0	1	7	4
D2	TRUE	FALSE	4	1	0	1	8	5
C4	FALSE	FALSE	3	2	1	2	9	8
D3	TRUE	FALSE	4	1	0	1	10	7
A2	TRUE	TRUE	1	1	0	1	11	11

Figure 3.3: Projecting basic properties of the nodes in the BOM table.

`DEPTH(ℓ)` — The number of edges on the path from ⊤ to ℓ.

`SIZE(ℓ)` — The number of nodes in the subtree rooted at ℓ.

`DEGREE(ℓ)` — The number of children of ℓ.

`HEIGHT(ℓ)` — The height of the subtree rooted at ℓ, i.e., the depth of the deepest node in ℓ's subtree.

`PRE_RANK(ℓ)` — The preorder traversal rank of ℓ.

`POST_RANK(ℓ)` — The postorder traversal rank of ℓ.

Figure 3.3 shows the result of projecting all these properties for the BOM table. Note the values of `DEPTH`, `HEIGHT`, `PRE_RANK`, and `POST_RANK` are 1-based. The following simple query shows the node properties in action:

```
SELECT ID, DEPTH(Node) AS "Depth" FROM BOM WHERE IS_LEAF(Node) = TRUE
```

It produces a list of all non-composite parts (i.e., leaves) and their respective depths.

These functions are primarily offered for the sake of expressiveness and convenience. Therefore, they are deliberately not strictly orthogonal. For example, `IS_ROOT(`v`)` is equivalent to `DEPTH(`v`) = 1` and thus redundant. (We will explore more equivalences of this kind in § 5.1.2.) Furthermore, the user could also compute these functions manually, using for example joins or hierarchical windows. It is therefore acceptable if a particular implementation chooses to not expose all of the functions. `HEIGHT` and `DEGREE`, in particular, are used rather infrequently but may be expensive to compute depending on which indexing scheme is employed, so we regard them as optional.

As the semantics of SQL mandate, the order of the result tuples in the above query is undefined. To traverse a hierarchy in a particular order, one can combine `ORDER BY` with a node property. As an example, consider a parts explosion for the BOM (cf. § 2.2.3), which shows the parts in depth-first order, down to a certain maximum depth:

```
-- Depth-first, depth-limited parts explosion with depths
SELECT ID, DEPTH(Node) FROM BOM WHERE DEPTH(Node) < 5 ORDER BY PRE_RANK(Node)
```

In fact, the primary use case for PRE_RANK and POST_RANK is to perform such topological sorting. With PRE_RANK, parents will be arranged before children (preorder); with POST_RANK, children will be arranged before parents (postorder). A breadth-first search order can be achieved with DEPTH:

```
-- Breadth-first parts explosion
SELECT ID, DEPTH(Node) FROM BOM ORDER BY DEPTH(Node), ID
```

3.3.2 Hierarchy Predicates

Besides querying node 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 binary *hierarchy predicates* (where ℓ_1 and ℓ_2 are NODE values):

IS_PARENT(ℓ_1,ℓ_2) — whether ℓ_1 is a parent of ℓ_2.

IS_CHILD(ℓ_1,ℓ_2) — whether ℓ_1 is a child of ℓ_2.

IS_SIBLING(ℓ_1,ℓ_2) — whether ℓ_1 is a sibling of ℓ_2, i. e., has the same parent.

IS_ANCESTOR(ℓ_1,ℓ_2) — whether ℓ_1 is an ancestor of ℓ_2.

IS_ANCESTOR_OR_SELF(ℓ_1,ℓ_2) — whether $\ell_1 = \ell_2$ or ℓ_1 is a ancestor of ℓ_2.

IS_DESCENDANT(ℓ_1,ℓ_2) — whether ℓ_1 is a descendant of ℓ_2.

IS_DESCENDANT_OR_SELF(ℓ_1,ℓ_2) — whether $\ell_1 = \ell_2$ or ℓ_1 is a descendant of ℓ_2.

IS_PRECEDING(ℓ_1,ℓ_2) — true iff ℓ_1 precedes ℓ_2 in preorder and is not an ancestor of ℓ_2.

IS_FOLLOWING(ℓ_1,ℓ_2) — true iff ℓ_1 follows ℓ_2 in preorder and is not a descendant of ℓ_2.

As we already noted in § 2.2.4, the task of axis navigation maps quite naturally onto a self-join with an appropriate hierarchy predicate as the join condition. For example, the following lists the IDs of all pairs (ℓ_1, ℓ_2) of nodes where ℓ_1 is a descendant of ℓ_2:

```
SELECT u.ID, v.ID FROM BOM u JOIN BOM v ON IS_DESCENDANT(u.Node, v.Node)
```

As another example, we can use a join to answer the classic *where-used* query on a bill of materials (cf. § 2.2.3). The query "Where is part D2 used?" corresponds to enumerating all ancestors of the corresponding node:

```
SELECT u.ID FROM BOM u, BOM v WHERE IS_ANCESTOR(u.Node, v.Node) AND v.ID = 'D2'
```

Here we formulated the join as a filter.

The set of predicates is somewhat aligned with the axis steps of XPath (see § 2.1). Note that the *preceding* and *following* predicates are only meaningful in an ordered hierarchy, and will thus rarely be used in order-indifferent applications. Also, the set of predicates again has some redundancy for the user's convenience; for example, IS_ANCESTOR and IS_DESCENDANT are symmetric, and the OR_SELF variants are a simple shorthand for hand-written OR expressions. In § 5.1.2 these properties are explored further.

```
Sale : {[ID, Item, Customer, Product, Date, Amount]}
SELECT Customer, Date, SUM(Amount) OVER w
  FROM Sale
WINDOW w AS (
          PARTITION BY Customer                               -- window partition clause
          ORDER BY Date                                       -- window ordering clause
          RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
             EXCLUDE NO OTHERS                                -- window frame clause
       )
```

Figure 3.4: An example window specification for a sales table.

The set of functions and predicates presented thus far covers everything that is needed in the common application scenarios we encountered, while being aligned with the capabilities of the hierarchy indexing schemes we considered for the backend. It is of course conceivable (and straightforward!) to further extend this set in order to cover more specialized use cases.

3.3.3 Hierarchical Windows

While node properties and hierarchy predicates are useful in particular for pattern matching queries, hierarchical windows are designed to facilitate complex hierarchical computations based on the concept of unary structural grouping we introduced in § 2.2.4. Unlike binary grouping, unary structural grouping is a novel concept to SQL that could not be expressed in the past. Following our informal overview in § 3.1, we now cover the syntax and semantics of our extensions for unary grouping.

Windowed Tables and Hierarchies. *Windowed tables* are a convenient and powerful means for aggregations and statistical computations on a single table, which otherwise would require unwieldy correlated subqueries. Their implicitly self-joining nature makes them a natural fit for structural grouping. We therefore add the concept of *hierarchical* windows to this mechanism.

Let us first briefly review the standard terminology and behavior of windowed tables. (We assume the reader to already be somewhat familiar with these concepts; refer to e. g. [116].) A window is formed on the result table of a `SELECT`/`FROM`/`WHERE` query according to a *window specification*. Note the table underlying a window may not be grouped. That is, in case `GROUP BY` is used in the same SQL block, the statement will be syntactically transformed by the query compiler so that the `GROUP BY` is encapsulated in a nested subquery. This ensures that these two related features do not get into each other's way. A standard *window specification* may comprise a *window partition clause*, a *window ordering clause*, and a *window frame clause*. As an example, consider again our sales table from § 2.2.3: Figure 3.4 shows how we may annotate the table with per-customer sales totals running over time. The used frame clause is:

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

This happens to be the implicit default and thus could be omitted. Conceptually, the query is evaluated as follows: (1) the sales are partitioned by Customer; (2) each partition is sorted by Date; (3) within each sorted partition, each tuple t is associated with a group of tuples relative to t, its *window frame* as determined by the frame clause, in this case: all sales up to t; (4) the window function (`SUM`) is evaluated for that group and its result appended to t.

The frame is always a subsequence of the current ordered partition. The order imposed by the ordering clause is generally a partial order, since tuples may not be distinct with respect to the `ORDER BY` fields. Tuples in t's frame that match in these fields are called *peers* or `TIES`.

Hierarchical Window Specification. For unary structural grouping, our windowed table will be a collection of nodes; that is, there is a `NODE` column whose values are drawn from a hierarchical base table. (Table Input1 from the scenario of § 2.2.4 is an example.) We extend the standard window specification with a new `HIERARCHIZE BY` clause specifying a *hierarchical window*. This clause may take the place of the ordering clause behind the partitioning clause. That is, partitioning happens first as usual, and hierarchizing replaces ordering. While window ordering turns the partition into a partially ordered sequence, hierarchizing turns it into an acyclic directed graph derived from the hierarchy. We begin our discussion with a minimal hierarchical window specification, which omits partitioning and the frame clause (so the above default applies):

```
HIERARCHIZE BY ℓ [BOTTOM UP|TOP DOWN]
```

The clause determines the `NODE` field ℓ, its underlying hierarchy H, and the direction of the intended data flow (bottom up by default), giving us all information we need to define an appropriate $<$ predicate on the partition:

top-down: $u < t \;:\Longleftrightarrow\; \texttt{IS_ANCESTOR}(u.\ell, t.\ell)$
bottom-up: $u < t \;:\Longleftrightarrow\; \texttt{IS_DESCENDANT}(u.\ell, t.\ell)$

We additionally need the notion of *covered* elements we already used informally in § 2.2.4. An element u is said to be *covered* by an element t if no third element lies in between:

$$u <: t \;:\Longleftrightarrow\; u < t \;\wedge\; \neg\,\exists\, u' : u < u' < t.$$

Using $<:$, we can identify the immediate $<$ neighbors (descendants/ancestors) of a tuple t within the current partition. Note that in case *all* hierarchy nodes are contained in the current partition, the "tuple u is covered by t" relationship is equivalent to "node $u.\nu$ is a child/parent of $t.\nu$." However, we need the general $<:$ notion because the current partition may well contain only a *subset* of the nodes. The $<:$ predicate helps us establish a data flow between tuples even when the intermediate nodes do not appear in the input.

A tuple u from the current partition can be related in four relevant ways to the current tuple t:

(a) $u < t$ (b) $t < u$ (c) $u.\nu = t.\nu$ (d) neither of those

```
SELECT SUM(Value) OVER (HIERARCHIZE BY Node) AS x FROM Input3
```

Input3		Window Frame							Result
Node	Value		0	1	2	3	4	5	x
C1	100	0	$\hat{=}$						100
C2	200	1		$\hat{=}$					200
D1	1000	2			$\hat{=}$				1000
D3	3000	3				$\hat{=}$			3000
B2	20	4			$<:$	$<:$	$\hat{=}$		4020
A1	1	5	$<:$	$<:$	$<$	$<$	$<:$	$\hat{=}$	4321

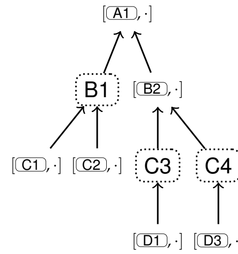


Figure 3.5: Applying a bottom-up hierarchical window to a table.

To reuse the syntax of the standard window frame clause without any modifications, we have to reinterpret three concepts accordingly: `PRECEDING` tuples are those of category (a); `FOLLOWING` tuples are those of category (b); `TIES` are tuples of category (c).

In the bottom-up case, `PRECEDING` tuples correspond to descendants and `FOLLOWING` tuples to ancestors of $t.\nu$. These terms are intuitively understood if we view the tuples in the window frame as a sequence partially ordered by $<$. They are not to be mixed up with the *preceding* and *following* hierarchy axes. Tuples on those axes, as well as tuples where ν is `NULL`, fall into category (d) and are always excluded from the frame.

The default frame clause includes categories (a), (c), and the current row. The handling of (c) tuples can be controlled independently via the `EXCLUDE` clause. The options are to include them (`NO OTHERS`, default) or to exclude the `CURRENT ROW`, the `TIES`, or the whole `GROUP`. Note that distinguishing between category (c) and (d) is specific to hierarchizing; standard window ordering puts *all* tuples for which neither $u < t$ nor $t < u$ holds into t's peer group (with $<$ defined according to the ordering clause).

Example. As a basic example, consider Figure 3.5, where we apply a bottom-up hierarchical window to table Input3 and compute $x =$ `SUM(Value)`, just as in Statement II-b from our initial overview in § 3.1. The matrix indicates the relationships of the tuples. Since our window uses the default frame clause, the frames comprise exactly the $<$ and $<:$ tuples, plus the current row. (Note there are no ties in this example data.) Summing over them yields the x values shown to the right. Note that although the table does not include the intermediate nodes B1/C3/C4, the input values of C1/C2 do still count into A1, and likewise for D1/D3 and the B2 tuple, as illustrated by the data flow graph to the right. As said, unary grouping does not require all intermediate nodes to be present in the input. Thus, it behaves precisely like the alternative binary approach based on an `IS_DESCENDANT_OR_SELF` join (Statement II-a from § 3.1).

For basic rollups, which are by far the most common type of hierarchical computation, the implicit window frame clause does exactly the "right thing"—thanks to our definitions of $<$ and the `PRECEDING`/`FOLLOWING` concepts—and it is hard to imagine a more concise and readable way of expressing them in SQL.

3.3.4 Recursive Expressions

Thus far, hierarchical windows are merely a shorthand; the computations we have seen can equivalently be expressed using join–group–aggregate statements. Structural recursion, however, significantly extends their expressive power. To enable recursive expressions, we recycle the SQL keyword `RECURSIVE` and allow wrapping it around expressions containing one or more window functions:

```
RECURSIVE [τ] (expr) AS c
```

This makes a field c of type τ accessible within any contained window function, and thus provides a way to refer to the computed *expr* value of any tuple in the window frame. If c is used anywhere in *expr*, τ must be specified explicitly, and an implicit `CAST` to τ is applied to *expr*. Automatic type deduction in certain cases is a possible future extension, but it is not generally possible without ambiguity.

The following additional rules apply: First, if *expr* contains one or more window function expressions of the form "$expr_i$ `OVER` w_i," all used hierarchical windows w_i must be equal (that is, match in their partitioning and `HIERARCHIZE` clauses, including the `NODE` field and the direction).

Second, the frame of each window w_i is restricted as follows: only the *covered* tuples ("`RANGE 1 PRECEDING`") can potentially be included in the frame, and in particular `EXCLUDE GROUP` is enforced. That is, the frame clause of every window function within *expr* effectively becomes:

```
RANGE BETWEEN 1 PRECEDING AND CURRENT ROW EXCLUDE GROUP
```

This in particular ensures that the window frame will not contain the `CURRENT ROW`, any `TIES`, or any `FOLLOWING` tuples. If any of those were contained in the frame, any access to field c within *expr* would create a cyclic dependency. (It is conceivable to loosen the restrictions somewhat and give the user more control via a custom frame clause. However, as there are no obvious use cases for this functionality, we do not consider frame clauses at this point.)

Third, the field c may appear only *within* one of the window function expressions $expr_i$; for instance, in combination with an aggregate function *AGG*:

```
RECURSIVE τ (...  AGG(expr′) OVER w ...) AS c
```

Mentioning c outside a window function would implicitly access the current tuple, which is forbidden, whereas according to SQL's rules, mentioning c within $expr'$ implicitly accesses the *frame row* (`FRAME_ROW`), which thanks to our restrictive window frame can only be a covered tuple for which the c value is available. While this standard behavior is what is usually intended and quite convenient, SQL has a way to override the implicit frame row access. We could for example refer to the current tuple even within *AGG* by using a so-called *nested window function*:

```
RECURSIVE τ (...  AGG(...VALUE_OF(c AT CURRENT_ROW)...) OVER w ...) AS c
```

```
SELECT * FROM (                                                          I-b
   SELECT h.*,
          SUM(Amount) FILTER (WHERE Type = 'type A') OVER w
     FROM Hierarchy1 h LEFT OUTER JOIN Sale s ON Node = s.Product
   WINDOW w AS (HIERARCHIZE BY Node)
) WHERE DEPTH(Node) <= 3
```

Figure 3.6: Statement I-a from § 3.1, expressed using hierarchical windows.

We prohibit this for *c* (which, as said, may only be accessed "`AT FRAME_ROW`"), but allow it for any other field.

Returning to Figure 3.5, we can now equivalently apply the *recursive* rollup expression from our example Statement II-c in § 3.1 to Input3:

```
RECURSIVE INT (Value + SUM(x) OVER w) AS x
```

The window frames are now restricted to the covered $<:$ tuples. Since Input3 is already ordered suitably for bottom-up evaluation—i. e., postorder—we can fill in the x result column in a single pass and always have the x values of our frame rows at hand when we need them.

3.3.5 Further Examples

Even with non-recursive expressions, hierarchical windows are already an attractive alternative to verbose join–group–aggregate statements. Consider again our opening Statement I-a from § 3.1. A little-known but useful feature of the aggregation functions in SQL is that the input values to each function can be further restricted by specifying a `FILTER` condition. This allows us to state this query as shown in Figure 3.6. Since a self-join is now implicitly covered through the hierarchical window, we save one join over Statement I-a. Note also that the outer join may yield tuples where Amount is `NULL`, but these are conveniently ignored in the `SUM`; therefore, no further preprocessing is necessary.

Altogether there are three conceivable points in the evaluation of a `SELECT` statement where the user might want to add `WHERE` conditions: a priori (*before* windows are formed); as `FILTER` (restricting the computation input but not affecting the table); and a posteriori (restricting the output). To achieve the latter, two selections must be nested, as SQL currently has no `HAVING` equivalent for windowed tables.

Figure 3.7 shows further meaningful expressions. They all are based on the bottom-up or top-down hierarchical window specified by the following query:

```
Input : {[Node, ID, Weight, Value]}

SELECT Node, expr FROM Input                                             IV
WINDOW td AS (HIERARCHIZE BY Node TOP DOWN),
       bu AS (HIERARCHIZE BY Node BOTTOM UP)
```

The input table here models a weighted hierarchy with additional values attached to the nodes (like in Figure 2.4d, p. 18). Some of the example expressions are based

```
(1a)  SUM(Value) OVER bu
(1b)  RECURSIVE INT (Value + SUM(x) OVER bu) AS x
(2a)  PRODUCT(Weight) OVER td  -- non-standard
(2b)  RECURSIVE INT (Weight * COALESCE(FIRST_VALUE(x) OVER td, 1)) AS x
(3a)  SUM(Value) OVER (bu RANGE 1 PRECEDING EXCLUDE GROUP)
(3b)  RECURSIVE (SUM(Value) OVER bu)
(4a)  RECURSIVE DOUBLE (Weight * (Value + SUM(x) OVER bu)) AS x
(4b)  RECURSIVE DOUBLE (Value + Weight * (SUM(x) OVER bu)) AS x
(4c)  RECURSIVE DOUBLE (Value + SUM(Weight * x) OVER bu) AS x
(4d)  RECURSIVE DOUBLE (Value + SUM(VALUE_OF(Weight AT CURRENT_ROW) * x) OVER w) AS x
 (5)  RECURSIVE VARCHAR ( COALESCE(FIRST_VALUE(x) OVER td, '') || '/' || ID) AS x
(6a)  COUNT(*) OVER td
(6b)  RECURSIVE INT (COALESCE(FIRST_VALUE(x) OVER td, 0) + 1) AS x
(7a)  COUNT(*) OVER bu
(7b)  RECURSIVE INT (COALESCE(FIRST_VALUE(x) OVER td, 0) + 1) AS x
 (8)  RECURSIVE INT (1 + COALESCE(MAX(x) OVER bu, 0)) AS x
(9a)  COUNT(*) OVER (bu RANGE 1 PRECEDING EXCLUDE GROUP)
(9b)  RECURSIVE (COUNT(*) OVER bu)
(10)  RECURSIVE (MY_FUNC(ARRAY_AGG(ROW(ID, x)) OVER w)) AS x
```

Figure 3.7: SQL examples for unary computations

on the assumption that Input contains *all* nodes of the underlying hierarchy, that is, $\Pi_{\text{Node}}(\text{Input}) = \Pi_{\text{Node}}(T)$, if T is the hierarchical base table. For computations that can be stated using either an ordinary expression or a `RECURSIVE` expression, Figure 3.7 shows both alternatives. (And there would be yet another alternative to express these computations: a self-joining join–group–aggregate statement, i. e., binary structural grouping; but we omit these verbose statements in the figure.)

(1) is our familiar rollup. Besides `SUM`, the operation in expression 1a could also be `AVG`, `MIN`, `MAX`, `COUNT` (cf. example 7), `EVERY`, `ANY`, or `ARRAY_AGG` to simply collect all values in an array. The recursive variant would have to be appropriately adapted. Duplicate input values can be eliminated as usual using `DISTINCT`, as opposed to the implicit `ALL` semantics. SQL's `FILTER` construct adds further expressiveness. For example, in a bill of materials we may count the distinct types of subparts of a certain manufacturer that each part is built of:

```
COUNT(DISTINCT Type) FILTER (WHERE Manufacturer = 'A') OVER bu
```

(2) is a top-down counterpart to the previous example (1). This computation yields the effective weights by multiplying over all tuples on the root path. Expression 2a uses a hypothetical `PRODUCT` aggregation function, which is curiously missing from standard SQL. Expression 2b works around that via recursion, aptly taking advantage of the special `FIRST_VALUE` aggregation function. To understand the example, note that for a top-down recursive computation, the window frame

can be either empty—making `FIRST_VALUE` yield `NULL`—or contain one covered ancestor. In our bill of materials the weight could be the part's multiplicity ("how often?") within its super-part. Here the product would tell us how often the part appears in total in the assembly.

(3) is a variant of example (1) summing over only the covered tuples rather than all descendants. In expression 3b we access only Value but not the actual expression result (thus, its type τ can be auto-deduced); still, the semantics are those of recursive evaluation. Under the assumption that Input2 happens to contain all hierarchy nodes, the cover relation $<:$ becomes equivalent to the `IS_CHILD` predicate as noted earlier; so the same could as well be achieved via join–group–aggregate.

(4) are variants of a weighted rollup; they sum up Value while taking Weight into account. Expression 4a immediately applies t.Weight to every computed $t.x$ value, whereas expression 4b applies it only to the $u.x$ values of the input tuples u. Expression 4c applies the u.Weight of any input tuple u to the $u.x$ value of that tuple. (Therefore, during the computation of $t.x$, both the x and the Weight fields of the frame rows are accessed.) Expression 4d is mostly equivalent to expression 4b, but brings it into a form similar to expression 4c using a nested window function to access the Weight of the current row within the `SUM`.

In general, such weighted rollups cannot be performed without (structural) recursion. A non-recursive expression such as "`SUM(Weight * Value) OVER bu`" would apply the Weight only to each tuple's Value, rather than the accumulated value, which is not what we intend. A non-recursive workaround that is sometimes applicable is to "multiply out" the expression according to the distributivity law and use *two* separate computations: First expression 2a, yielding absolute weights `w` for each tuple, then `SUM(w * Value)` bottom up. The result is then mostly equivalent to expression 4a.

(5) constructs a path-based representation of the hierarchy using the same technique as example (2). It builds a string from the ID values on the root path, for example `/A1/B1/C1` for (C1). Related hierarchy-handling tools such as Hierarchical Queries (§ 2.3.3), hierarchyid (§ 2.3.7), and ltree (§ 2.3.8) often provide similar functionality via built-in functions. This example demonstrates how one can easily emulate that functionality by a custom computation, but with added flexibility.

(6–9) compute properties of the data flow graph over the input table. As Input2 can be viewed as an extension of the hierarchical base table (containing *all* nodes of the hierarchy), they are equal to the node's (6) depth, (7) subtree size, (8) subtree height, and (9) degree. In general (7) gives us the size of the window frame and (9) the number of covered tuples. Note how expression 9b does not actually access any frame tuple at all. `COUNT(*)` counts tuples but does not actually access any attributes.

(10) Finally, if we need to go beyond the capabilities of SQL's aggregate functions

and expression language, we can use ARRAY_AGG to collect data from the covered tuples and pass it to a user-defined function.

Yet another option for (10) would be user-defined aggregates, which can be implemented in some RDBMS via proprietary extension mechanisms (e. g. [38]). Using these mechanisms arbitrarily complex computations could be plugged in.

3.4 DDL Extensions: Creating Hierarchies

The previous section covered our query language constructs which work on fields of type NODE. In this section, we show how hierarchical tables containing such fields are defined and maintained.

3.4.1 Hierarchical Base Tables

For newly designed applications we intend to provide specific DDL constructs to express and maintain hierarchies explicitly in the table schema (Requirement #3 from § 2.2.5). Because the underlying object is a persistent base table, updates are supported. In this scenario the user starts out with an empty hierarchy and then incrementally adds and updates its nodes using the DML constructs we discuss in § 3.5.

A base table definition may specify a *hierarchical dimension* as follows:

```
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:

```
ALTER TABLE T ADD HIERARCHY Node NULL
ALTER TABLE T DROP HIERARCHY Node
```

Like a column, a hierarchical dimension can explicitly be declared nullable, or a "NOT NULL" constraint can be added. If it is declared NOT NULL, the implicit NODE value of a newly inserted row becomes 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 whose structure is known to rarely or never change allows the system to employ a read-optimized indexing scheme (cf. Requirement #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, which makes the NODE column essentially immutable; 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. These restrictions are strict in the sense that an SQL error is raised when a prohibited DML operation is attempted. The default setting is SUBTREE, so full update flexibility is provided unless there is an explicit restriction from the user. Depending on the specified option, the system can choose an appropriate indexing scheme. The motivation behind distinguishing between single-node and subtree updates is that the latter require a more powerful dynamic indexing scheme than the former, with inevitable tradeoffs in query performance (see § 4.2).

Note that a table with one or more hierarchical dimensions might additionally have one or more *temporal* dimensions, specified via standard PERIOD definitions [97]. The defined hierarchies would then by default become temporal hierarchies. Although we do not cover this topic any further, the orthogonal way in which a hierarchical dimension is specified would allow our framework to fairly straightforwardly accommodate scenarios featuring temporal-hierarchical data in the future (see § 7).

3.4.2 Derived Hierarchies: The HIERARCHY Expression

While hierarchical base tables are a preferred choice for newly designed applications, derived hierarchies are targeted mainly at legacy applications. According to our Requirement #7 from § 2.2.5, legacy applications demand for a means to *derive* a hierarchy from an available table using a certain relational hierarchy encoding. 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 query language—and in particular, without requiring schema modifications via DDL.

For this purpose we provide the *derived hierarchy* construct, which syntactically resembles an invocation of a built-in table-valued function named HIERARCHY. It derives a hierarchy from a given *source table*, which may be a table, a view, or the result of an arbitrary subquery:

```
SELECT ..., name
  FROM HIERARCHY (
        USING source table AS source name
        ...            -- transformation specification
        SET name       -- NODE column name
      )
```

This expression can be used wherever a table reference is allowed, in particular a FROM clause. It is comparable to a table-valued function: Its result is a temporary table containing the data from the *source table* and additionally a new NODE column named *name* and an associated hierarchy as metadata. The transformation specification is a description of the specific relational input format. A system might support various common input formats, such as those we discussed in § 2.2. We next cover the proposed syntax of the transformation specification for the most common type of input, the adjacency list format (§ 3.4.3).

3.4.3 Deriving a Hierarchy from an Adjacency List

The syntax to derive a hierarchy from a table in the adjacency list format is as follows:

```
HIERARCHY (
   USING source table AS source name
  [START WHERE start condition]
   JOIN PRIOR parent name ON join condition
  [SEARCH BY order]
   SET node column name
)
```

Conceptually, this table-valued 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 optional `START WHERE` subclause can be used to restrict the hierarchy to only the nodes that are reachable from any node satisfying *start condition*. The optional `SEARCH BY` subclause can be used to specify a desired sibling order; if omitted, siblings are ordered arbitrarily. In more detail, the conceptual procedure for evaluating the complete expression is as follows:

1. Evaluate *source table* and materialize the required columns into a temporary table T. Also add a `NODE` column named *node column name* to T and initialize it with `NULL` values.

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 during Step 4 and populate the `NODE` column of T with a corresponding `NODE` value for each visited node, accordingly. The result of the `HIERARCHY` expression is T.

Note that this description is merely conceptual; we describe an efficient implementation in §4.6.

In Step 4, an error is raised when an edge is encountered that leads to an already visited node. The resulting hierarchy is therefore guaranteed to have a strict tree structure (Requirement #6). Alternatively, such edges could simply be ignored. This would effectively derive a spanning tree over the graph. Yet another alternative would be to revisit the previously visited node again but producing *new* tree nodes for that branch in the output. This would have the effect of "multiplying out" the whole graph. Which

```
WITH PartHierarchy AS (
   SELECT ID, Node
     FROM HIERARCHY ( USING BOM AS c JOIN PRIOR p ON p.ID = c.PID SET Node )
)
SELECT v.ID, DEPTH(v.Node) AS Depth
  FROM PartHierarchy u, PartHierarchy v
 WHERE u.ID = 'C2' AND IS_DESCENDANT(v.Node, u.Node)
```

Figure 3.8: Deriving a hierarchy from the BOM table and querying it ad hoc.

option is most suitable depends on the application to hand. Either option ensures that the derived hierarchy will be strict.

The `HIERARCHY` syntax may be reminiscent of Hierarchical Queries (§ 2.3.3) or RCTEs (§ 2.3.4). In particular, the self-join via *parent name* is roughly comparable to a `CONNECT BY` via `PRIOR` in a Hierarchical Query. However, the semantics of `HIERARCHY` are quite different in that by design only a *single* self-join is performed on the input rather than a recursive join. This allows for a very efficient evaluation algorithm compared to a recursive join. And there is another major conceptual difference to the mentioned approaches: Note that the `HIERARCHY` expression does nothing more than define a hierarchy; that hierarchy can then be queried by wrapping the expression into a `SELECT` statement. In contrast, an RCTE both defines *and* queries a hierarchy in one convoluted statement. Separating these two aspects greatly increases comprehensibility. As an example, consider again our familiar BOM of Figure 1.1 (p. 2). The statement in Figure 3.8 uses a CTE featuring a `HIERARCHY` expression to derive the Node column from ID and PID, then selects the IDs and depths of all parts that are used within part C2. The mentioned separation of aspects is clearly visible. PartHierarchy could be extracted into a SQL view and reused for different queries. One might argue that an RCTE or Hierarchical Query could be wrapped into a view as well, but that still would not clearly separate the definition and querying aspects: The user would have to compute any node properties that might potentially be needed in later queries (such as `DEPTH` in the example) upfront in the view *definition*, even though they are clearly part of the *query*. A query that does not need the depths would still trigger their computation, resulting in unnecessary overhead. In contrast, our design allows the user to cleanly defer the selection of node properties of interest to the actual query.

3.5 DML Extensions: Manipulating Hierarchies

In order to accommodate legacy applications (Requirement #7), we aim to provide a smooth transition path from relationally encoded hierarchies 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 relational encodings such as adjacency lists, as this approach allows them to avoid any schema changes. In such a scenario, *each* view evaluation conceptually derives a new hierarchical table from scratch (although that may often be elided if the system supports view caching). As the `NODE`

column of a derived hierarchy is immutable, the only way the hierarchy structure can be modified is to manipulate the original encoding.

In a second stage, a partly adapted legacy application might add a static hierarchical dimension (`UPDATES=BULK`) alongside the existing adjacency list encoding, and update the dimension periodically from the adjacency list by performing an explicit *bulk update*. This can be done using SQL's standard `MERGE INTO` statement, by specifying a `HIERARCHY` expression as the source to merge and the existing hierarchical table as the target.

These two stages provide a way to gradually adopt hierarchy functionality in a legacy application, but they come at the cost of requiring frequent bulk builds whenever the hierarchy structure changes. Therefore, for both green-field applications as well as fully migrated legacy applications it may be preferable to use a *dynamic* hierarchy (`UPDATES=NODE` or `SUBTREE`). Such a hierarchical table supports fine-grained updates via explicit DML constructs. Again, our `NODE` abstraction allows us to straightforwardly add support for expressing structural updates while using a minimally invasive syntax: Update operations naturally translate into ordinary `INSERT` and `UPDATE` statements operating on the `NODE` column of a hierarchical dimension.

Node Anchors. To specify the position where a row is to be inserted into the hierarchy, we use an *anchor* value. A node anchor acts as a pseudo value for the `NODE` field in an `INSERT` or `UPDATE` statement. It is essentially an existing `NODE` value plus an indication of where the node or subtree being manipulated is to be placed relative to that node. Again, we refrain from extending the SQL grammar and instead use simple built-in functions which take a `NODE` value as input and yield a node anchor as output:

- `BELOW(`ℓ`)` inserts the new row as a child of ℓ. The insert position among the existing children is undefined.
- `BEFORE(`ℓ`)` or `BEHIND(`ℓ`)` insert the new row as the left or right sibling of ℓ.
- `ABOVE(`ℓ`)` inserts the new row above ℓ. The parent of ℓ becomes the parent of the new node, and ℓ itself becomes the child of the new node.

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

Inserting Nodes. When we insert a new row into a hierarchical base table using `INSERT`, we can add a corresponding node to the hierarchy by specifying a node anchor for the `NODE` field. Obtaining an appropriate node anchor typically requires a sub-select. For example, we can add a node (B3) as new child of (A2) into our familiar BOM hierarchy like this:

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

or

```
INSERT INTO BOM (ID, Node)
SELECT 'B3', BELOW(Node) FROM BOM WHERE ID = 'A2'
```

To make the new row a root, we can specify the `DEFAULT` value. To omit the row from the hierarchy, we can specify `NULL`, provided the `NODE` column is nullable:

```
INSERT INTO T (..., Node) VALUES (..., DEFAULT)  -- insert as root node
INSERT INTO T (..., Node) VALUES (..., NULL)  -- do not insert a node
```

In case no value is provided in the `INSERT` statement, `DEFAULT` and `NULL` are the implicit defaults for non-nullable and for nullable dimensions, respectively. A row with a `NULL` node can of course be added to the hierarchy later on via an `UPDATE`.

Relocating Nodes. A node can be relocated by updating the `NODE` field of the associated row using an ordinary `UPDATE` statement, again specifying an anchor to indicate the target position:

```
UPDATE T SET Node = ( SELECT BELOW(Node) FROM T WHERE ... ) WHERE ...
```

If the node to be relocated has any descendants, the whole subtree rooted at the node is relocated together with it. However, such a subtree relocation is only allowed if option `UPDATES=SUBTREE` is used for the hierarchical dimension. In order to guarantee structural integrity, the system must raise a SQL error when a subtree is attempted to be relocated below a node within that same subtree, as this would result in a cycle.

Removing Nodes. A node can be removed from a hierarchy either by deleting its row or by updating its `NODE` field to `NULL`:

```
UPDATE T SET Node = NULL WHERE ...
DELETE FROM T WHERE ID = '...'  -- by attribute/key value
DELETE FROM T WHERE Node IN (...)  -- by NODE value
```

However, these operations are prohibited if the node has any descendants that are not also removed during the same transaction. In other words, in order to remove the root of a subtree, all its children have to be relocated first or removed together with that node. This rule is quite restrictive, but it is necessary to ensure that removing nodes does not leave behind an invalid hierarchy. Also note that in case the hierarchical dimension uses option `UPDATES=BULK`, nodes cannot be individually inserted, relocated, or removed. This renders it difficult to delete any rows that are part of the hierarchy, that is, whose `NODE` value is not `NULL`. One operation that is allowed on a `BULK` dimension is to truncate the whole hierarchy by unconditionally setting the `NODE` values of *all* rows to `NULL`: "`UPDATE T SET Node = NULL`." Thus, a way to delete rows with associated nodes is to first truncate the hierarchy, then delete the rows at will, and subsequently rebuild the hierarchy from scratch. If any of the mentioned rules are violated, a SQL error is raised. Although this is restrictive, it ensures that the structure of the hierarchy remains valid at any time, thus satisfying Requirement #6 of § 2.2.5.

3.6 Advanced Examples

We now explore several more advanced scenarios involving hierarchical tables. This includes modeling entities that are part of multiple hierarchies or entities that appear in the same hierarchy multiple times (§ 3.6.1), as well as creating inhomogeneous hierarchies that contain entities of various types (§ 3.6.2). These examples are inspired by customer scenarios at SAP and demonstrate that our language extensions stand up to real-world scenarios.

3.6.1 Flexible Forms of Hierarchies

In certain applications an entity may be designed to belong to two or even more hierarchies at the same time. 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 add *multiple* hierarchical dimensions to a table:

```
CREATE TABLE Employee (
   ID INTEGER PRIMARY KEY,
   ...,
   HIERARCHY Disciplinary,
   HIERARCHY Line
)
```

A more complex case arises when a single hierarchy shall be able to contain the same entity multiple times. Again, a bill of materials can serve as an example: A common part such as a screw may generally appear multiple times within the same BOM, and we may not want to replicate its attributes each time. This is a typical $1:N$ relationship: one part may appear N times in the hierarchy. The solution is to model this case exactly as one would usually model $1:N$ relationships in the relational model, namely by introducing two tables and linking them by means of a foreign key constraint. In our example, we would split the original BOM schema from Figure 3.2 (p. 38) into a Part table storing per-part data and a correspondingly simplified BOM table:

```
CREATE TABLE Part (
   ID INTEGER PRIMARY KEY,
   Kind VARCHAR,
   -- per-part master data
   ...
)
```

```
CREATE TABLE BOM (
   NodeID INTEGER PRIMARY KEY,
   HIERARCHY Node,
   PartID INTEGER,   -- a node is a part (N:1)
   FOREIGN KEY (PartID) REFERENCES Part (ID),
   -- additional node or edge attributes
   ...
)
```

The generalized pattern of "factoring out" a hierarchy into a satellite table is as follows:

```
CREATE TABLE T ( PK INTEGER PRIMARY KEY, ... )
CREATE TABLE U ( HIERARCHY Node, FK INTEGER UNIQUE REFERENCES T (PK), ... )
```

The `UNIQUE` constraint on FK ensures that U–T is a $1:1$ relationship. Thus, the hierarchy effectively arranges the rows from both T and U. Fields that logically belong to a node or edge should be placed in U in order to keep T uncluttered. Rows from T that are

currently not a part of the hierarchy do not need to actually appear in U. To remove a node, one would rather delete the corresponding U row altogether instead of setting its `Node` value to `NULL`.

With this pattern, one can add any number of distinct T-hierarchies by simply creating more satellite tables like U. Furthermore, if we relax the U–T relationship by dropping the `UNIQUE` constraint, a T entity can effectively appear any number of times within the same hierarchy.

3.6.2 Heterogeneous Hierarchies (I)

In many scenarios, entities of various types are arranged in a single hierarchy. Different types of entities are characterized by different sets of attributes. Especially in structured documents such as XML, various types of document nodes (i. e., tags with corresponding attributes) are interleaved in arbitrary ways, and XPath expressions routinely combine hierarchy 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 type, each with an different set of columns as appropriate. Returning to our BOM example from § 3.6.1, we can elaborate the Part–BOM data model further by introducing entities of type "engine":

```
CREATE TABLE Engine (
   ID INTEGER PRIMARY KEY,
   FOREIGN KEY (ID) REFERENCES Part (ID),
   Power INTEGER,
   ... -- engine-specific master data
)
```

While Part contains master data that is common to all kinds of parts, Engine adds master data that is specific to engines. Note that the two tables must have a common primary key domain (ID). The foreign key constraint enforces a 1 : 1 relationship to Part. BOM is now a *heterogeneous* hierarchy in that each node can be either of type Part or of the specific subtype Engine. This design is extensible. Further part types can be added by defining further tables like Engine.

When querying the BOM, type-specific part attributes can be accessed by simply joining in the corresponding master data. Such a join acts as a filter by type. As an example query, 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 3.9 shows the solution. Note in particular how the BOM–Engine join implicitly ensures that node e is of kind "engine," so we do not need a separate test.

3.6.3 Heterogeneous Hierarchies (II)

We now consider another example from a Human Resources use case, where entities of two entirely different types are mixed into a single hierarchy. Suppose we have

```
SELECT *
  FROM BOM c, Part cm,          -- compound node and master data
       BOM f, Part fm,          -- fitting node and master data
       BOM e, Engine em         -- engine node and master data
 WHERE c.ID = cm.ID AND cm.Kind = 'compound'
   AND IS_DESCENDANT(f.Node, c.Node)
   AND f.ID = fm.ID AND fm.Kind = 'fitting' AND fm.Manufacturer = 'X'
   AND IS_DESCENDANT(e.Node, f.Node)
   AND e.ID = em.ID AND em.Power > 700
```

Figure 3.9: Querying the heterogeneous bill of materials hierarchy.

```
SELECT e.Emp AS "Employee", d.Dep AS "Department"
  FROM DepEmp e, DepEmp d
 WHERE e.Emp = 'Bob'
   AND IS_ANCESTOR(d.Node, e.Node)   -- e is within d
   AND d.Dep IS NOT NULL             -- d is a department
   AND NOT EXISTS (                  -- no department in between
          SELECT * FROM DepEmp v
           WHERE IS_ANCESTOR(d.Node, v.Node) AND IS_ANCESTOR(v.Node, e.Node)
             AND v.Dep IS NOT NULL
       )
```

Figure 3.10: Querying the Department–Employee hierarchy: “Where does Bob work?”

an existing Department table and an existing Employee table, and we want to arrange the departments and employees in a combined hierarchy. In this hierarchy, each department node would have one child of type employee indicating the department manager, as well as any number of children for the subdepartments; each manager node would form the root of a subhierarchy of employees. One way to model this is as follows:

```
CREATE TABLE DepEmp (
   NodeID INTEGER PRIMARY KEY,
   HIERARCHY Node,
   Dep INTEGER UNIQUE REFERENCES Department (ID),
   Emp VARCHAR UNIQUE REFERENCES Employee (ID),
   CONSTRAINT C1 CHECK (Dep IS NULL OR Emp IS NULL)
)
```

Every node in this table references either a department or an employee, but never both at the same time. Additional attributes on the entities can be joined in from the Department and Employee tables when needed.

To query, for example, the department a given employee e works in, we need to determine the closest department node above e. Figure 3.10 shows a possible solution. A bigger challenge is to query the manager of e: If e has a parent of type employee, that is the manager; if, however, e is a department head, the manager is the head of the parent department. To solve this query we would need a much more complex pattern.

An advantage of the DepEmp design is that it is not invasive: We do not need to modify the original Department and Employee tables, nor do we need to add any

```
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.Node, u.Node)
   AND DEPTH(v.Node) = DEPTH(u.Node) + 2
   AND IS_DESCENDANT(w.Node, v.Node)
   AND IS_LEAF(w.Node) = TRUE  -- store locations are leaves
   AND w.ID = store.LocationID
   AND store.ID = sale.StoreID
 GROUP BY v.ID
```

Figure 3.11: A rollup query based on a geographic dimension hierarchy.

satellite tables or have a shared key domain between Department and Employee, unlike the approaches of § 3.6.1 and § 3.6.2. However, it is somewhat inconvenient that an employee node cannot have both a parent employee (manager) *and* a parent department, as this would violate the strictness constraints. Our example works around this by interleaving the nodes such that employees are only indirectly connected to their department via the manager. This of course results in more complex queries, as the above example shows.

3.6.4 Dimension Hierarchies

As we noted in § 2.2.3, dimension hierarchies of fact tables are an important use case. In this example we consider a variant of the sales scenario, where the sales table records the store where each sale took place, and the stores are arranged in a common geographic hierarchy. The schema is:

$$\text{Sale} : \{[\text{StoreID}, \text{Date}, \text{Amount}, \ldots]\}$$
$$\text{Store} : \{[\underline{\text{ID}}, \text{LocationID}, \ldots]\}$$
$$\text{Location} : \{[\underline{\text{ID}}, \text{Node}, \text{Name}, \ldots]\}$$

By joining Sale–Store–Location, we can associate each sale with a Node value corresponding to the location of the store.

Now, let us answer the following example query: "Considering only sales within Europe, list the total sales per sub-subregion." The prose form of this query speaks, quite implicitly, of *three* distinct Location nodes: a reference node u, namely Europe; a node v two levels below u, corresponding to a sub-subregion; and a leaf node w below v, corresponding to the location of a store where a sale took place. We can use a join–group–aggregate statement to answer the query. While we are mainly interested in the values for v, we also need names for u and w in the statement. All in all, three self-joined instances of the hierarchical table are required. Figure 3.11 shows the solution.

Note the straightforward reading direction of the query: it intuitively matches the direction of navigation in the hierarchy. This example and the one from Figure 3.9 are good examples of how our language extensions maintain the original "look and feel" of SQL, so even more complex queries will be intuitive to experienced SQL users.

3.6.5 A Complex Report Query

In our previous example, we were able to express a non-trivial hierarchical computation in terms of a simple join–group–aggregate statement. We now investigate a more elaborate query whose solution requires hierarchical windows. Similar to the setup of § 2.2.4, it is based on an abstract hierarchical table and a separate table of numeric input values attached to some of the nodes:

$$\text{Hierarchy} : \{[\underline{\text{ID}}, \text{Node}, \text{Payload}]\} \qquad \text{Input} : \{[\text{Node}, \text{Value}]\}$$

At the heart, the purpose of this query is to compute a bottom-up sum of the measure values, much like in the basic examples of § 3.1. But there are some additional requirements: First, before we can do the rollup we need an initial join between the given Input values and the Hierarchy nodes, which gives us a combined input table for unary structural grouping. Second, each node in the hierarchy carries 128 bytes of further payload, which we want to include in the result and thus have to carry through the computation. Third, in addition to the actual sum x for each node, we want to also show the contribution p in percent of each node's x value to the total of its parent. Fourth, we want to report only the three upper levels of the hierarchy, with the nodes arranged in preorder. Fifth, we want to visualize the nodes' positions using Dewey-style path strings (see § 2.3.6). If the x value of B1 is 1250, an example result line for C2 may be

```
['/A1/B1/C2', 125, 10%, payload]
```

The additional "stress factors" in this example are commonly found in real-world queries. In particular, the contribution percentage is an often-requested measure in financial applications we saw at SAP. To obtain both the contribution percentages and the Dewey path strings, we need a bottom-up *and* a top-down hierarchical computation. With the help of hierarchical windows this is expressed as follows:

```
-- bottom-up rollup
WITH T1 (Node, ID, Payload, x) AS (
  SELECT h.Node, h.ID, h.Payload,
         SUM(in.Value) OVER (HIERARCHIZE BY h.Node BOTTOM UP)
    FROM Hierarchy h LEFT OUTER JOIN Input in ON h.Node = in.Node
),
-- top-down computation of the contribution p and the path
T2 (Node, ID, Payload, x, p, Path) AS (
  SELECT Node, ID, Payload, x,
         RECURSIVE ( 100.0 * x / FIRST_VALUE(x) OVER w ),
         RECURSIVE VARCHAR(255) (
            COALESCE(FIRST_VALUE(p) OVER w, '') || '/' || ID) AS p,
    FROM T1 WINDOW w AS (HIERARCHIZE BY Node TOP DOWN)
)
-- final filtering and odering
SELECT Path, x, p, Payload FROM T2 WHERE DEPTH(Node) <= 3
 ORDER BY PRE_RANK(Node)
```

Without our language extensions this query would be prohibitively difficult to express. By contrast, the above statement is fairly intuitive, considering the complexity of the

task at hand. Moreover, it can also be translated into a very efficient execution plan, as our evaluation in § 6.3.7 will show.

3.6.6 Hierarchical Tables and RCTEs

In § 2.3.4 we mentioned that hierarchical tables are not intended to completely replace or obviate RCTEs. In fact, these two features can interact. We illustrate this by the example of a quantities calculation in a bill of materials. Suppose the payload of a Part contains a Quantity field indicating how often the part is used within its respective parent part. The following query computes the required quantity of each part by multiplying the quantities of the nodes on the root path:

```
WITH RECURSIVE RCTE (Node, Quantity) AS (
   SELECT Node, Quantity FROM Part WHERE DEPTH(Node) = 1
UNION ALL
   SELECT v.Node, u.Quantity * v.Quantity
     FROM RCTE u JOIN Part v ON IS_CHILD(v.Node, u.Node)
)
SELECT * FROM RCTE
```

This particular query could of course be expressed more easily in terms of a hierarchical window. (A join–group–aggregate statement over the ancestors of each node is *not* possible, but only because there is no `PRODUCT` aggregation function in standard SQL.) However, what the example shows is that RCTEs can navigate hierarchical tables just like adjacency list tables, using, for example, `IS_CHILD` as the join condition within the recursive query expression. Legacy queries relying on RCTEs can therefore straightforwardly be rewritten when the underlying tables are migrated to hierarchical tables. An RCTE like in the example could be considered comparatively readable: The general structure of the recursion—start at level 1, iteratively move along the child axis—is quite obvious in the statement. An expression of the form `IS_CHILD(`v`, `u`)` expresses the navigation direction more clearly than an ordinary join such as `v.PID = u.ID` would.

4 The Backend Perspective of Hierarchical Tables

In the hierarchical table model we introduced in the previous chapter, a hierarchy appears to the user as an ordinary relational table with an added column of type `NODE`. From the user's perspective, the `NODE` field is opaque; it is merely a "syntactic handle" for expressing queries and updates on the hierarchy. At the backend, this fundamental design gives us the flexibility to choose between different encoding schemes for the structure, which we call *hierarchy indexing schemes* (§ 4.1). This flexibility is critical: As Requirement #8 from § 2.2.5 anticipates, there is no "one size fits all" encoding that could serve all applications equally well. All data structures that make up a specific type of indexing scheme are encapsulated by an abstract index interface (§ 4.1.2). The interface exposes the required primitives to implement our SQL extensions with suitable worst-case asymptotic performance guarantees. Thus, the backend can be mostly agnostic about the layout and manipulation of the `NODE` values and auxiliary data structures. Each implementation of an indexing scheme comes with a definition of the abstract `NODE` type and a number of related data types, plus usually one or more auxiliary data structures. We survey a broad range of existing schemes that fit into our abstract model (§ 4.2), and recommend suitable default options for common scenarios (§ 4.3) and in particular static settings (§ 4.4). We also include an overview of our novel *order indexes* family of indexing schemes (§ 4.5), which target highly dynamic scenarios. Besides the query primitives, indexing schemes also need to support *bulk building* (§ 4.6) as well as SQL update statements (§ 4.7). Efficient bulk-building is particularly important for supporting legacy applications that derive hierarchies from existing data; we therefore cover the necessary algorithms in much detail.

4.1 Hierarchy Indexing Schemes

A *hierarchy indexing scheme* is a specific implementation of a hierarchy encoding on top of the abstract `NODE` column design. We begin our discussion by considering the fundamental components that make up such an implementation.

This chapter builds upon the publications [9], [32], and [33]. As the implementation of our proposed order indexes is not in the focus of this thesis, we only summarize their relevant characteristics in § 4.5 and refer to those publications for further information. § 4.6 provides a much extended discussion of the bulk-building techniques first described in [9].

4.1.1 Basic Concepts

With most types of indexing schemes, the `NODE` column is backed by a more or less sophisticated auxiliary data structure. The *run-time* representation of a hierarchy therefore generally consists of two logical data objects: the complete contents of the `NODE` column, and an instance of the auxiliary data structure. Unlike the `NODE` column, the index structure is entirely transparent to the user: It is not an explicit, user-accessible object in the database catalog. It is created or destroyed, kept up to date, and leveraged for query processing by the backend alone without the user ever noticing. In some ways, this is comparable to a traditional index on a table column, such as a B-tree or hash table. However, there is an important difference: The hierarchy indexing scheme contains the hierarchy structure as *non-redundant* information. Traditional indexes, by contrast, are redundant and could be dropped at any time without losing information.

In § 4.2 we explore specific types of indexing schemes. Each type of indexing scheme consists of implementations of the following general components:

- The auxiliary index structure (and possible sub-structures).
- A `Label` data type, which is the actual data that is stored per row in the abstract `NODE` field. We call these scheme-specific, SQL-level node identifiers the *labels* of the indexing scheme.
- A `Node` data type and a `Cursor` data type. A `Node` or `Cursor` object can be obtained from a `Label`. These objects are *internal* node handles as opposed to the labels, which are exposed at SQL level. They are obtained and manipulated through the interface and used in the backend during query processing. `Cursor` is a subtype of `Node`, that is, cursors can be used wherever the `Node` type is required. While `Node` is a basic handle to a single node, a `Cursor` can additionally be used to traverse the hierarchy structure in various directions.
- An implementation of the abstract hierarchy index interface based on the `Label`, `Node`, and `Cursor` types and the auxiliary index structure.

Some update operations require a specification of an intended *target position* in the hierarchy. This concept is based on the `Node` data type: A target position is simply a `Node` object plus an anchor, which can be “before” or “behind” (meaning: as a direct left or right sibling), “above,” or “below.” The unspecific “below” is used when the sibling order is not meaningful; its semantics are up to the indexing scheme.

A hierarchy indexing scheme has to maintain a *bidirectional* association between the table rows and the internal representation of the nodes. If an auxiliary data structure is used, the `Label` essentially works as a link from the row to an entry representing the node in the index structure. Defining the mechanism for this is up to the indexing scheme. The link from a hierarchy node to its associated row, on the other hand, is realized via ordinary row IDs. The row ID can be obtained from a `Node` or `Cursor` handle via the index interface. This “linking” via row IDs is analogous to how ordinary database indexes such as B-trees work. The row IDs are usually stored directly in the

index structure entries. How a row ID is actually represented may differ significantly between row stores, column stores, disk-based systems, and in-memory systems, but this is not relevant in the following.

The motivation for having *three* data types rather than just the labels is that the process of locating an internal entry (i. e., obtaining a `Node`) in the auxiliary index structure given only a `Label` as a starting point may have a non-trivial cost, which we want to avoid where possible. Furthermore, the capability to perform systematic traversals of the hierarchy structure may require a `Cursor` to maintain a non-trivial state.

The actual definitions of the `Label`, `Node`, and `Cursor` data types heavily depend on the design of the auxiliary data structure. A `Label` may be a simple pointer, a structured set of values, or a non-trivial binary code. The `Node` and `Cursor` types are a form of pointer to an entry in the data structure, which can be dereferenced in constant time.

Any `Label`, `Node`, and `Cursor` values are manipulated solely through the index interface and are treated by the rest of the backend as opaque bytes. In that regard, the `NODE` type is comparable to `BINARY` or `VARBINARY`. That is, the `Label` of a row can be accessed individually and treated as opaque bytes. Whether its length is fixed or variable depends on the indexing scheme, but the size limit is known at data definition time.

***Static* and *Dynamic* Indexing Schemes.** The choice among indexing schemes matters particularly with regard to their varying degrees of support for updates. We differentiate static and (more or less) dynamic schemes. *Static indexing schemes* cannot incorporate incremental updates and are invalidated whenever nodes are inserted, relocated, or removed. An invalidated indexing scheme must be rebuilt at the latest when it is accessed by a transaction to which the changes are visible. Static schemes generally can employ significantly more compact data structures and efficient algorithms than their dynamic counterparts. But they are only feasible when the hierarchy structure does not change often, or when changes only happen in batches. Two cases where static schemes are a perfect choice are derived hierarchies, which are by design immutable, and snapshots of historic states in system-time temporal tables. *Dynamic indexing schemes*, on the other hand, trade off query processing performance and simplicity of data structures in return for more efficient and more powerful update operations. They are significantly more complex to design and implement.

Clustering. In physical storage, database systems usually maintain the rows of a table in a purposeful order, such as by primary key or in a way that maximizes compression rates or temporal locality. In many scenarios (such as the SAP ERP use cases we analyzed in [9]), a hierarchy is not the primary dimension of a table, and clustering the associated table by hierarchy structure is infeasible or not preferable. Hierarchy indexes are therefore *non-clustered* indexes: The existence of a hierarchical dimension does not enforce a particular ordering of the rows in memory or persisted storage. That said, the user may still direct the system to cluster a hierarchical table by the hierarchy order, that is, to arrange the rows in pre-, post-, or level-order. This can significantly speed up certain workloads, such as when nodes are enumerated first via the

CONSTRUCTION

$H.\text{build}(H')$
$H.\text{relink}(\ell, r, r')$

BASIC ACCESS

$\nu \leftarrow H.\text{node}(\ell)$
$c \leftarrow H.\text{cursor}(\nu)$
$\ell \leftarrow H.\text{label}(\nu)$
$r \leftarrow H.\text{rowid}(\nu)$

ORDINAL ACCESS

$i \leftarrow H.\text{pre-rank}(\nu)$
$c \leftarrow H.\text{pre-select}(i)$
$i \leftarrow H.\text{post-rank}(\nu)$
$c \leftarrow H.\text{post-select}(i)$

NODE PROPERTIES

$H.\text{depth}(\nu)$
$H.\text{is-root}(\nu)$
$H.\text{is-leaf}(\nu)$
$H.\text{size}(\nu)$
$H.\text{range-size}([\nu_1, \nu_2])$

BINARY PREDICATES

$H.\text{is-before-pre}(\nu_1, \nu_2)$
$H.\text{is-before-post}(\nu_1, \nu_2)$
$H.\text{axis}(\nu_1, \nu_2)$
$H.\text{is-parent}(\nu_1, \nu_2)$

TRAVERSAL

$c' \leftarrow H.\varsigma\text{-prev}(c), H.\varsigma\text{-next}(c)$
$c' \leftarrow H.\varsigma\text{-begin}(c), H.\varsigma\text{-end}(c)$
$c' \leftarrow H.\text{next-sibling}(c)$
$c' \leftarrow H.\text{next-following}(c)$

LEAF UPDATES

$\nu \leftarrow H.\text{insert-leaf}(r, p)$
$H.\text{relocate-leaf}(\nu, p)$
$H.\text{remove-leaf}(\nu)$

SUBTREE UPDATES

$H.\text{relocate-subtree}(\nu, p)$
$H.\text{remove-subtree}(\nu)$

RANGE UPDATES

$H.\text{relocate-range}([\nu_1, \nu_2], p)$
$H.\text{remove-range}([\nu_1, \nu_2])$

INNER UPDATES

$\nu \leftarrow H.\text{insert-inner}(r, [\nu_1, \nu_2])$
$H.\text{relocate-inner}(\nu, [\nu_1, \nu_2])$
$H.\text{remove-inner}(\nu)$

Figure 4.1: Essential query and update primitives on hierarchies.

hierarchy index but additional table columns are subsequently accessed. We do not consider these orthogonal optimization techniques further, but note that our design can easily incorporate them, as hierarchy indexes use ordinary row IDs to reference the associated rows.

4.1.2 A Common Interface for Hierarchy Indexes

Figure 4.1 gives an overview of the primitives for querying and updating hierarchies that are provided by our common index interface. While many further primitives are conceivable, our design goal has been to devise a coherent set of essential primitives that is aligned with the functionality of our SQL language constructs (and foreseeable future extensions of it), but that at the same time allows for practical and efficient indexing scheme implementations. Therefore, our discussion in § 4.1.4 omits query primitives that are clearly too specific (e. g., *least common ancestor*) or that few schemes can support (e. g., *subtree height*). Likewise, the update operations we present in § 4.1.5 can be reasonably supported by any sophisticated dynamic indexing scheme. Of course, implementers may still decide to add special primitives for advanced applications, or redundant primitives for special cases in order to improve performance further.

Many operations in our interface (in particular the traversal and ordinal primitives) depend on a well-defined and deterministic sibling order. We therefore consider only indexing schemes that maintain *ordered* hierarchies (§ 2.1). Even if the user hierarchy is unordered, they impose a deterministic, implementation-defined internal order.

We use a few conventions in the following discussion:

- The variable T refers to the hierarchical table containing the `NODE` column at hand, which is assumed to be named "Node." The variable H refers to the associated

instance of the hierarchy indexing scheme. The notation $|H|$ refers to the number of nodes in H. The notation $T[r]$ accesses a row of table T by its row ID r. For instance, $T[r]$.Node retrieves the label of row r.

- ℓ is a variable of type `Label`, that is, a value stored in some `NODE` column such as T.Node. Labels directly support basic equality comparisons $\ell_1 = \ell_2$ and $\ell_1 \neq \ell_2$. Variables ν and ν_i are of type `Node`. Variables c and c' are of type `Cursor`. Note again that a `Cursor` can be used in any place where a `Node` is required.

- The syntax $[\nu_1, \nu_2]$ denotes a *sibling range* of nodes: ν_2 must be a right sibling of ν_1 (or ν_1 itself), and $[\nu_1, \nu_2]$ refers to all siblings between and including ν_1 and ν_2.

- p indicates a target position in the hierarchy, used for updating.

All primitives that accept arguments of type `Label`, `Node`, `Cursor`, or a target position require that these arguments represent valid values, that is, a node with label ℓ or node handle ν or cursor c must exist in H. They are undefined otherwise. In particular, `Label` values are expected to be non-`NULL`. (`NULL` arguments are handled at the level of the SQL functions, which yield `NULL` when any argument is `NULL`.) These implicit preconditions are not repeated in the following.

4.1.3 Elementary Index Primitives

Before we proceed to the individual primitives for queries and updates, we first consider how an indexing scheme is constructed and how its fundamental index/table association can be accessed and maintained. We begin with construction:

H.build(H') — Builds the hierarchy from another hierarchy representation H'.

This operation can also be used to "bulk-update" an existing indexing scheme, even if it is static. It clears the contents of H and the associated `NODE` column and reconstructs the index structure and `NODE` column to reflect the structure of H'. The given hierarchy H' must be tied to the same hierarchical table T; that is, it must contain a subset of the rows in T. The underlying representation of H' may be entirely different from H. We require the worst-case asymptotic runtime complexity of build() to be in $\mathcal{O}(|H'| \log |H'|)$. For many indexing schemes it can even be in $\mathcal{O}(|H'|)$.

An efficient operation for bulk-building is useful in several situations: The main use case is deriving a new hierarchical table from existing relational data in another format, such as an adjacency list, using our `HIERARCHY()` construct. A related use case are bulk-updates via `MERGE`, which replaces the existing hierarchy structure in a hierarchical table by a newly derived structure. Another scenario is creating a snapshot from a temporal hierarchy index. Finally, serialization mechanisms can use bulk-building to efficiently reconstruct an index from a serial format.

§ 4.6 considers the implementation of build() in detail, and also discusses a practical data structure for an intermediate hierarchy representation H'.

Relinking Nodes. In general, executing any of the query or update primitives may require access to arbitrary other index entries and labels of T, even if the primitive itself logically affects only a single node. Therefore, an important precondition for all primitives is that all node–row associations are valid at the time of invocation. However, since indexing schemes reference rows through ordinary row IDs, these associations may be affected by database operations that are not direct manipulations of the hierarchy structure, such as DML statements against the hierarchical table T or a physical reorganization. We therefore need the database engine to cooperate: Whenever the system "moves" a row of T that is also referenced in H (i. e., a row whose `NODE` value is not `NULL`) in a way that affects its row ID, it needs to notify the indexing scheme to update its node–row association. The following primitive serves this purpose:

$H.\text{relink}(\ell, r, r')$ — Update the row associated with the node given by label ℓ.

This tells the index that the row with former ID r now has a new row ID r'.

Basic Access. A few elementary primitives are needed in order to obtain an index handle from a given table row, and vice versa:

$\nu \leftarrow H.\text{node}(\ell)$ — The node with label ℓ.

$c \leftarrow H.\text{cursor}(\nu)$ — A cursor to node ν.

$\ell \leftarrow H.\text{label}(\nu)$ — The label of node ν.

$r \leftarrow H.\text{rowid}(\nu)$ — The row ID corresponding to the given node ν.

Most query and update primitives require at least `Node` objects or even cursors as arguments. A label ℓ as such is therefore not very useful; the corresponding *node* has to be located first in the index structure using node(ℓ). Vice versa, given a `Node` handle, label() returns the corresponding `Label` object, that is, node() and label() are inverse:

$$H.\text{label}(H.\text{node}(\ell)) = \ell$$

The rowid() function returns the ID of the corresponding table row holding the label, which allows us to access the whole table row rather than just the label. Thus, label(ν) is logically just a short-hand for $T[H.\text{rowid}(\nu)]$.Node.

4.1.4 Index Primitives for Queries

Query primitives are the building blocks for answering high-level queries on hierarchies. We distinguish between four kinds of query primitives: node properties, binary predicates, traversal operations, and primitives for ordinal access.

Node Properties compute characteristic properties of a node:

$H.\text{depth}(\nu)$ — The number of edges on the path from the super-root $\top$ to ν.

H.is-root(ν) — Whether ν is a root node, i. e., a child of $\top$.

H.is-leaf(ν) — Whether ν is a leaf node, i. e., has no children.

H.size(ν) — The number of nodes in the subtree rooted in ν.

H.range-size($[\nu_1, \nu_2]$) — The number of nodes in all subtrees rooted in $[\nu_1, \nu_2]$.

These primitives are primarily needed for queries that explicitly request the node properties via the corresponding SQL functions, but they may also be leveraged internally, for example to gather statistics for cardinality estimations. Let us examine some of their properties: Regarding depth(), note that user-level roots have a depth of 1. The depth of the super-root would be 0, but this node can never appear in any user data. The integral functions depth(), size(), and range-size() yield values in the range $[1; |H|]$, but are neither injective nor surjective. Finally, the two unary predicates are redundant, as we have the equivalences

$$H.\text{is-root}(\nu) \Leftrightarrow H.\text{depth}(\nu) = 1 \quad \text{and} \quad H.\text{is-leaf}(\nu) \Leftrightarrow H.\text{size}(\nu) = 1.$$

However, they may be significantly cheaper to compute directly, depending on the indexing scheme in use.

Binary Predicates allow us to test the relationship of two given nodes.

$\nu_1 = \nu_2$ — Whether ν_1 and ν_2 are the same node.

H.is-before-pre(ν_1, ν_2) — Whether ν_1 precedes ν_2 in a preorder traversal.

H.is-before-post(ν_1, ν_2) — Whether ν_1 precedes ν_2 in a postorder traversal.

H.axis(ν_1, ν_2) — Returns the axis of ν_1 with respect to ν_2.

H.is-parent(ν_1, ν_2) — Whether ν_1 is the parent of ν_2.

There are exactly five basic, disjunct axes in which two given nodes can possibly be positioned (see § 2.1):

$$\text{Axes} = \{\text{preceding}, \text{ancestor}, \text{self}, \text{descendant}, \text{following}\}.$$

The `Node` equality check "=" has the usual properties of an equivalence relation. The is-before-pre() and is-before-post() predicates form a *total* and *strict* order relation on the set of `Node` objects—that is, they are irreflexive, asymmetric, and transitive. In particular, exactly one of the following terms is always true (where ς is either "pre" or "post"):

$$H.\text{is-before-}\varsigma(\nu_1, \nu_2) \quad \vee \quad (\nu_1 = \nu_2) \quad \vee \quad H.\text{is-before-}\varsigma(\nu_2, \nu_1).$$

Most of the order-based physical hierarchy operators we discuss in § 5 rely on "=" and is-before-ς() alone, which makes them the most important primitives for query

processing. The other two predicates are redundant. is-parent() can be expressed in terms of axis() and depth():

$$H.\text{is-parent}(\nu_1, \nu_2) \Leftrightarrow (H.\text{axis}(\nu_1, \nu_2) = \text{ancestor}) \wedge (H.\text{depth}(\nu_2) = H.\text{depth}(\nu_1) + 1).$$

A *self* axis check "$H.\text{axis}(\nu_1, \nu_2) = \text{self}$" is equivalent to "$\nu_1 = \nu_2$." The other axes can be dissected into elementary is-before-ς() checks:

$$\begin{aligned}
H.\text{axis}(\nu_1, \nu_2) &= \text{preceding} && \Leftrightarrow && H.\text{is-before-pre}(\nu_1, \nu_2) \wedge H.\text{is-before-post}(\nu_1, \nu_2).\\
H.\text{axis}(\nu_1, \nu_2) &= \text{ancestor} && \Leftrightarrow && H.\text{is-before-pre}(\nu_1, \nu_2) \wedge H.\text{is-before-post}(\nu_2, \nu_1).\\
H.\text{axis}(\nu_1, \nu_2) &= \text{descendant} && \Leftrightarrow && H.\text{is-before-pre}(\nu_2, \nu_1) \wedge H.\text{is-before-post}(\nu_1, \nu_2).\\
H.\text{axis}(\nu_1, \nu_2) &= \text{following} && \Leftrightarrow && H.\text{is-before-pre}(\nu_2, \nu_1) \wedge H.\text{is-before-post}(\nu_2, \nu_1).
\end{aligned}$$

It follows from these definitions that the *preceding*, *ancestor*, *descendant* and *following* axis checks are also strict order relations, albeit not total.

Despite their redundancy, we found dedicated and potentially optimized implementations for is-parent() and axis() to be clearly beneficial to performance, as navigation along these axes is featured in many typical queries.

Traversal Operations navigate along the hierarchy structure in various directions, in particular preorder ($\varsigma = \text{pre}$) and postorder ($\varsigma = \text{post}$).

$c' \leftarrow H.\varsigma\text{-prev}(c)$ — A cursor to the previous node in ς-order (if $c \neq H.\varsigma\text{-begin}()$).

$c' \leftarrow H.\varsigma\text{-next}(c)$ — A cursor to the next node in ς-order (if $c \neq H.\varsigma\text{-end}()$).

$c \leftarrow H.\varsigma\text{-begin}()$ — A cursor to the first node in ς-order.

$c \leftarrow H.\varsigma\text{-end}()$ — A cursor to the one-behind-the-last node in ς-order.

$c' \leftarrow H.\text{next-sibling}(c)$ — A cursor to the next sibling.

$c' \leftarrow H.\text{next-following}(c)$ — A cursor to the next node in preorder that is on the *following* axis of the node indicated by c.

Traversal operations are useful to implement scan operators that enumerate certain subsets of the nodes. The seemingly obscure next-following() retrieves the first preorder successor outside of the subtree rooted in c, which is often a suitable scan delimiter.

Ordinal Access Primitives are based on the position of a node with respect to a systematic traversal of the hierarchy H in ς-order, where ς can be "pre" or "post."

$i \leftarrow H.\varsigma\text{-rank}(\nu)$ — The numerical position of ν in the ς-ordered sequence of nodes.

$\nu \leftarrow H.\varsigma\text{-select}(i)$ — The i-th node in the ς-ordered node sequence (if $i \in [1, |H|]$).

pre-rank() and post-rank() return the position of a node with respect to preorder or postorder traversal, respectively. Given such ranks, pre-select() and post-select() return the corresponding nodes.

The ς-rank() functions are counterparts to the is-before-ς() order relations. As such, they are bijective, produce a dense numbering from 1 to $|H|$, and

$$H.\text{is-before-}\varsigma(\nu_1, \nu_2) \Leftrightarrow H.\varsigma\text{-rank}(\nu_1) < H.\varsigma\text{-rank}(\nu_2).$$

ς-rank() and ς-select() are inverse:

$$H.\varsigma\text{-rank}(H.\varsigma\text{-select}(i)) = i \quad \text{and} \quad H.\text{select-}\varsigma(H.\varsigma\text{-rank}(\nu)) = \nu.$$

With the help of ς-rank() we can establish a relationship between depth(), pre-rank(), post-rank(), and size():

$$H.\text{pre-rank}(\nu) - H.\text{post-rank}(\nu) + H.\text{size}(\nu) - H.\text{depth}(\nu) = 0.$$

Thanks to this relationship, only three out of four functions have to be explicitly computed, and the fourth can be derived using simple arithmetics. In particular, by implementing rank() the support for size() and even range-size() comes as a byproduct.

Although few applications require the full power of these primitives, they can be very convenient. Ranks are, for instance, used to create compact representations of subtrees, so-called tree signatures, for pattern matching purposes [117]. A use cases for select primitives are top-k queries with an offset for displaying parts of the hierarchy in a user interface. Unlike the other primitives we discussed, most indexing schemes require additional implementation efforts, storage space, and update overhead to support them efficiently. This will not pay off in every application. In particular, it is often sufficient to use is-before-ς() comparisons instead of evaluating the ς-rank() values. We therefore rely only on the is-before-ς() primitives in our algorithms.

4.1.5 Index Primitives for Updates

We now consider a set of update primitives to manipulate the hierarchy structure represented by an indexing scheme. Note that these primitives work only on index level; they do not modify the hierarchical *table* apart from potentially updating the affected labels stored in the `NODE` column. They are, however, usually triggered by table updates, such as when a row is deleted. How exactly SQL update statements are translated into invocations of the primitives is covered in § 4.7.

The update primitives can be grouped into four classes: *leaf updates*, *subtree updates*, *range updates*, and *inner updates*, named after the affected entities. Within each class, three kinds of updates are conceivable: *inserting* nodes, *relocating* nodes, and *removing* nodes. Unlike relocate and remove, insert primitives add *new* nodes to the hierarchy, so they do not take an existing `Node` object as their argument; instead, their argument is a row ID r, and their return value is the new `Node` object. The row r must not yet exist in H, that is, its `NODE` value must be `NULL`.

Figure 4.2 illustrates the various update classes with regard to the relocate kind on an example hierarchy. The other kinds, insert and remove, are similar; the only difference is that the updated entities enter or leave the hierarchy, respectively, instead

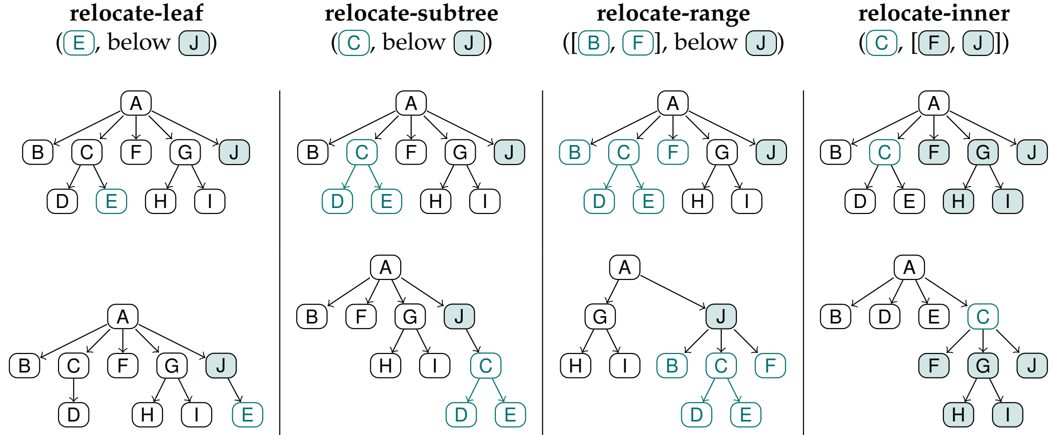


Figure 4.2: The four classes of relocation updates on an example hierarchy: before (top) and after (bottom).

of just being relocated. In a sense, the relocate kind subsumes the others: insertions and removals are relocations into and out of the hierarchy, respectively. Thus, any index that handles relocation efficiently can support efficient insertion and removal as well.

Leaf Updates alter a single leaf node.

$v \leftarrow H.\text{insert-leaf}(r, p)$ — Inserts a node for row r as a leaf node at position p.
Precondition: p is a valid position in H, and does not use an *above* anchor.

$H.\text{relocate-leaf}(v, p)$ — Relocates node v to position p (if $H.\text{is-leaf}(v)$).

$H.\text{remove-leaf}(v)$ — Removes node v (if $H.\text{is-leaf}(v)$).

Subtree Updates relocate or remove a subtree.

$H.\text{relocate-subtree}(v, p)$ — Relocates the subtree rooted in v to position p.

$H.\text{remove-subtree}(v)$ — Removes the subtree rooted in v.

Range Updates alter all subtrees rooted in a range of siblings.

$H.\text{relocate-range}([v_1, v_2], p)$ — Relocates all subtrees rooted in range $[v_1, v_2]$ to p.

$H.\text{remove-range}([v_1, v_2])$ — Removes all subtrees rooted in range $[v_1, v_2]$.

Inner Updates alter an inner node.

$v \leftarrow H.\text{insert-inner}(r, [v_1, v_2])$ — Inserts a node v for row r in place of range $[v_1, v_2]$ and makes $[v_1, v_2]$ children of v.

H.relocate-inner(ν, $[\nu_1, \nu_2]$) — Puts the children of ν into ν's place and moves ν to the place of $[\nu_1, \nu_2]$, which become children of ν.

H.remove-inner(ν) — Removes node ν and puts its children into its place below its former parent.

Note that insert-subtree() and insert-range() are missing from the set, as our SQL extensions have no syntax for inserting a whole range of nodes at once (or "copying" over a range of nodes from another hierarchical table).

Leaf updates have been the main focus of many prior publications. In comparison to the other classes, they are easiest to implement. However, we found that many scenarios require strong support for the more complex *subtree updates*. Here, a subtree of an arbitrary size is removed as a whole, or moved in bulk to another location. As an example, consider an enterprise asset hierarchy where an assembly line is relocated to another plant, or a machine or robot is relocated to another assembly line. Since the machine consists of parts and subparts and the assembly line consists of various machines, robots, and equipment, these are large subtree relocations. Note that subtree updates subsume the corresponding leaf updates, as every leaf is a trivial subtree. But since most indexing schemes can employ optimizations for leaves, distinguishing between those classes is still useful in practice. The *range updates* might seem obscure at first sight, but they are in fact very powerful: They subsume subtree and leaf updates, because a subtree rooted in ν is a trivial sibling range. Finally, the *inner updates* are useful for injecting 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 [17] emit edit scripts featuring these operations. An index that is being used for replaying such edit scripts has to support them. Inner updates are subsumed by range updates as well: We can use range relocation to move all children of an inner node to another position, making this node a leaf, then use a leaf update to relocate (or remove) it. Thus, indexes that support range updates—such as our order indexes—can implement *all* update primitives in terms of range updates.

Unlike the query primitives, the update primitives may be significantly more complex for *ordered* than for unordered hierarchies, as the sibling order must be maintained. Vice versa, allowing an ordered indexing scheme to neglect the sibling order may be an interesting optimization. For example, some schemes may be able to leverage the unspecific "below" target position and pick the sibling position that yields the cheapest update, while for others, "below x" may safely default to, e. g., "as last child of node x" with no performance loss.

4.2 Implementing Indexing Schemes: State of the Art

In § 2.3.6 we already noted that there is a tremendous amount of prior work on representing hierarchies in databases. Many of the techniques discussed in those works can be used to implement our interface for indexing schemes. Each design provides a different tradeoff between query and update performance. However, it turns out that

	Query Primitives											
	ACCESS	PREDICATES			PROPERTIES				TRAVERSAL			
	node cursor	before	axis	parent	depth	root	leaf	size	pre	post	sibling	following
Naïve Schemes												
Adjacency	1	—	l	1	l	1	1	—	—	—	—	—
Linked	1	l'	l'	$1'$	l'^{*}	$1'$	$1'$	s	$1'$	$1'$	$1'$	$1'$
Containment-based Labeling Schemes												
NI	$\log n$	1	1	b'	—	b'	1	1	$1'$	$1'$	$1'$	$1'$
Dyn-NI	$\log n^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	b'	—	b'	$1'$	s	$1'$	$1'$	$1'$	$1'$
(Dyn-)NI-Parent	$\log n^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	$l^{\ddagger *}$	1	$1'/1^{\dagger}$	s	$1'$	$1'$	$1'$	$1'$
(Dyn-)NI-Level	$\log n^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	$1^{\ddagger}$	1	1	$1'/1^{\dagger}$	s	$1'$	$1'$	$1'$	$1'$
Path-based Labeling Schemes												
Dewey	$l \log n$	l	l	l	l	1	l'	s	$1'$	$l \log n$	$l \log n$	$l \log n$
Dyn-Dewey	$l \log n^{\ddagger}$	$l^{\ddagger}$	$l^{\ddagger}$	$l^{\ddagger}$	$l^{\ddagger}$	1	$l'^{\ddagger}$	s	$1'$	$l \log n^{\ddagger}$	$l \log n^{\ddagger}$	$l \log n^{\ddagger}$
Index-based Schemes												
B-BOX[B]	B	$\log_B n'$	$\log_B n'$	b'	—	b'	$1'$	s	$1'$	$1'$	$1'$	$1'$
AO-Tree	1	$\log n'$	$\log n'$	$\log n'$	$\log n'^{*}$	$\log n'^{*}$	$1'$	$\log n'^{*}$	$1'$	$1'$	$1'$	$1'$
BO-Tree[B]	1	$\log_B n'$	$\log_B n'$	$\log_B n'$	$\log_B n'^{*}$	$\log_B n'^{*}$	$1'$	$\log_B n'^{*}$	$1'$	$1'$	$1'$	$1'$
O-List[B]	1	$1'$	$1'$	$1'$	$1'$	$1'$	$1'$	1	$1'$	$1'$	$1'$	$1'$
DeltaNI	$\log u$	$1'$	$1'$	$1'$	$1'$	$1'$	$1'$	$\log u'$	$\log u'$	$\log u'$	$\log u'$	$\log u'$

	Update Primitives				
	LEAF	SUBTREE	INNER	RANGE	SKEW[u]
Naïve Schemes					
Adjacency	1	1	c	c	1
Linked	1	1	c	c	1
Containment-based Labeling Schemes					
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-based Labeling Schemes					
Dewey	lf	$l(f+s)$	$l(f+s)$	$l(f+s)$	$l(f+s)$
Dyn-Dewey	l	ls	ls	ls	$l+u$
Index-based Schemes					
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/depth of node
- b number of siblings
- u number of updates
- B block size
- c number of children
- s number of descendants
- f number of following siblings incl. descendants
- — not supported
- $'$ operates on `Node` or `Cursor` objects
- $\dagger$ only in the static variant
- $\ddagger$ in the Dyn variant, performance may decrease over time due to growing labels
- $*$ $\mathcal{O}(1)$ during index scan

Qualitative rating:

- ☐ efficient
- ☐ mostly efficient
- ☐ inefficient or unsupported

Figure 4.3: Asymptotic query and update complexities for various groups of indexing schemes (amortized, average case).

many proposals are variations of basic schemes with similar capabilities and asymptotic properties. We can therefore group the prior art into a small taxonomy.

In this taxonomy we primarily distinguish between *labeling* schemes and *index-based* schemes. The former class of schemes was already introduced in § 2.3.6, together with the subcategories of *naïve*, *containment-based*, and *path-based* labeling schemes. In our terminology of hierarchy indexing schemes, labeling schemes store the essential information encoding the hierarchy structure entirely in the `Label` data type, and use common database indexes for the auxiliary index structure. These schemes are comparatively simple and non-invasive in that they can be implemented entirely in terms of SQL. *Index-based schemes*, by contrast, enhance the RDBMS backend by special-purpose auxiliary index structures. These indexes contain the bulk of the hierarchy information, and the `Label` objects mainly act as handles into those structures. Modifying the backend is of course a much more invasive approach.

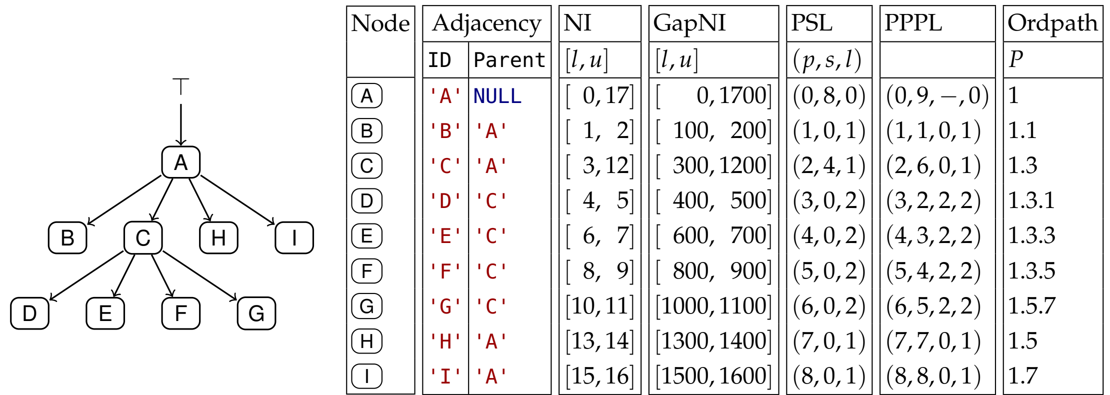
Generally speaking, labeling schemes allow for executing certain queries very efficiently by merely considering the labels at hand, whereas index-based schemes suffer from the overhead of their auxiliary index structure. Regarding updates, however, a common drawback of many labeling schemes is that structural updates can easily become very expensive. Most attempts to mitigate this bring along major drawbacks like large memory footprints, a vulnerability to skewed updates, or no longer supporting all desired query and update primitives. With our work on order indexes (§ 4.5) we therefore make a case for index-based schemes.

Figure 4.3 is a compact overview of the different groups of schemes in our taxonomy. It shows the amortized asymptotic time complexities for the most important primitives and the different classes of updates of Figure 4.1. Many of our index primitives are not discussed in the cited works, but inferring their implementation is straightforward.

For entries tagged with a red ′ in the figure we assume the `Node` object of the involved node to be readily available; for the other entries we assume a `Label` argument. The update operations in general are given `Node` objects to indicate a desired target position. Depending on the access path to the index during query execution, the `Node` objects may first need be obtained from the labels via the node() function—which can have non-negligible costs, as the table also shows—, or they might happen to be readily available from previous operations. A case where the `Node` objects are already available "for free" is when the traversal primitives are used to scan the hierarchy, or when a `Node` was obtained from a prior update at the same position. For the asymptotic complexities of updates we therefore do *not* include the time for obtaining the `Node` handles.

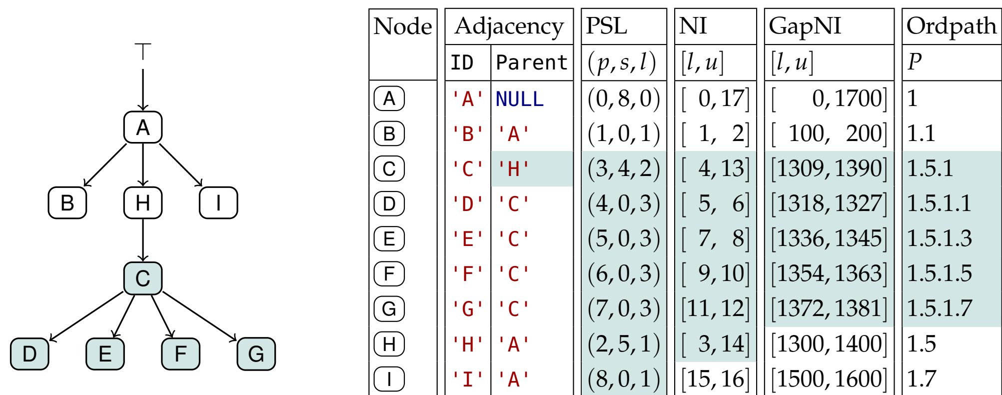
4.2.1 Challenges in Dynamic Settings

A particular emphasis in our following analysis is on highly dynamic use cases. We will see that existing dynamic indexing schemes lack important capabilities: they either ignore important query primitives, or they inherently suffer from certain problems inhibiting their update flexibility or robustness. To characterize these missing capabilities, we identify three problems *P1* to *P3* in this section. Figure 4.4 illustrates different indexing schemes—the familiar adjacency list model, as well as PSL, NI, GapNI, and



Node	Adjacency		NI	GapNI	PSL	PPPL	Ordpath
	ID	Parent	$[l,u]$	$[l,u]$	(p,s,l)		P
A	'A'	NULL	[0, 17]	[0, 1700]	(0, 8, 0)	(0, 9, −, 0)	1
B	'B'	'A'	[1, 2]	[100, 200]	(1, 0, 1)	(1, 1, 0, 1)	1.1
C	'C'	'A'	[3, 12]	[300, 1200]	(2, 4, 1)	(2, 6, 0, 1)	1.3
D	'D'	'C'	[4, 5]	[400, 500]	(3, 0, 2)	(3, 2, 2, 2)	1.3.1
E	'E'	'C'	[6, 7]	[600, 700]	(4, 0, 2)	(4, 3, 2, 2)	1.3.3
F	'F'	'C'	[8, 9]	[800, 900]	(5, 0, 2)	(5, 4, 2, 2)	1.3.5
G	'G'	'C'	[10, 11]	[1000, 1100]	(6, 0, 2)	(6, 5, 2, 2)	1.5.7
H	'H'	'A'	[13, 14]	[1300, 1400]	(7, 0, 1)	(7, 7, 0, 1)	1.5
I	'I'	'A'	[15, 16]	[1500, 1600]	(8, 0, 1)	(8, 8, 0, 1)	1.7

Figure 4.4: An example hierarchy and the corresponding data stored in the `Label` objects for different indexing schemes.



Node	Adjacency		PSL	NI	GapNI	Ordpath
	ID	Parent	(p,s,l)	$[l,u]$	$[l,u]$	P
A	'A'	NULL	(0, 8, 0)	[0, 17]	[0, 1700]	1
B	'B'	'A'	(1, 0, 1)	[1, 2]	[100, 200]	1.1
C	'C'	'H'	(3, 4, 2)	[4, 13]	[1309, 1390]	1.5.1
D	'D'	'C'	(4, 0, 3)	[5, 6]	[1318, 1327]	1.5.1.1
E	'E'	'C'	(5, 0, 3)	[7, 8]	[1336, 1345]	1.5.1.3
F	'F'	'C'	(6, 0, 3)	[9, 10]	[1354, 1363]	1.5.1.5
G	'G'	'C'	(7, 0, 3)	[11, 12]	[1372, 1381]	1.5.1.7
H	'H'	'A'	(2, 5, 1)	[3, 14]	[1300, 1400]	1.5
I	'I'	'A'	(8, 0, 1)	[15, 16]	[1500, 1600]	1.7

Figure 4.5: Relocating subtree C in the example from Figure 4.4. Labels that need to be changed are highlighted.

Ordpath, which we cover later—to help us exemplify the problems.

[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, which cannot even handle the ancestor-descendant relationship efficiently. The fundamental query primitives we identified in § 4.1.4 are the minimum we require. An implementation that fails to support them cannot be considered a general-purpose hierarchy indexing scheme.

[P2] Lack of Complex Update Capabilities. Various 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 removal of single leaf nodes, and fail to recognize more complex operations.

Consider *subtree* relocation: The trivial adjacency list model happens to naturally

support this quite well. However, virtually all labeling schemes by design preclude an efficient implementation, because they inherently require relabeling *all* nodes in the relocated subtree. In general, subtree and range operations have to be implemented naïvely through node-by-node processing, requiring at least s leaf updates for a subtree of size s. Figure 4.5 shows the hierarchy from Figure 4.4 with the subtree rooted in (C) moved below (H), and highlights the $\Omega(s)$ fields that need to be updated.

For small subtrees, an update complexity of $\Omega(s)$ might be tolerable. However, real-world hierarchies—such as those found in SAP ERP datasets—tend to 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 certain use cases, a high percentage of updates (e. g., 31% in the enterprise asset hierarchy we examined in [31]) are indeed subtree relocations.

Furthermore, note that although subtree relocation may appear as an unnatural bulk operation in comparison to single leaf insertion or removal, the operation is quite heavily exercised in SQL statements: Our language extensions make it easy to relocate a subtree by simply updating the `NODE` field of its root. Likewise, in the adjacency list table shown in Figure 4.5, the relocation is performed by simply setting the Parent field of (C) to `'H'`. A single `UPDATE` statement may in fact provoke an arbitrary number of subtree relocations *at once*. Our assessment of prior work therefore places an emphasis on the complex update operations we identified in § 4.1.5.

[P3] Vulnerability to Skewed Updates. Certain dynamic labeling schemes crumble when they are confronted with skewed updates, such as when inserts are issued repeatedly at the same position. In some scenarios these updates are more frequent than is commonly acknowledged. For example, when inserting a new plant into an enterprise asset hierarchy, many nodes will be added at 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. We therefore point out such vulnerabilities in our assessment. In Figure 4.3, column skew[u] represents "skewed" leaf node insertions. It depicts the complexity of a single skewed insertion after u other skewed insertions have taken place. This is an indication of how severely an indexing scheme can be impacted by skewed insertion patterns in the worst case.

We proceed to explore the indexing schemes in our taxonomy with a focus on dynamic settings, and assess to which extent they suffer from the three identified problems.

4.2.2 Naïve Hierarchy Representations

Figure 4.3 includes two "naïve" indexing schemes. They are comparatively easy to implement, but cannot stand up to the full-blown indexing schemes in terms of query capabilities and efficiency (*P1*).

The first, Adjacency, is an incarnation of the familiar adjacency list scheme (§ 2.3.1). We assume a hash index for the ID and PID columns, so that node(), which performs a

lookup by label, can run in $\mathcal{O}(1)$. As Adjacency does not maintain a sibling order, it has no support for any of the order-based query primitives. The extreme simplicity of the scheme happens to make subtree relocations and the is-parent() predicate remarkably efficient. On the other hand, it is unsurprisingly very weak in answering most of the query primitives (*P1*).

The second naïve representation is Linked, a simple in-memory node structure reminiscent of a classic linked list, whose structure matches the hierarchy structure. Each node has links—in the form of direct pointers—to the parent, the first and last child, and the previous and next sibling. A label stores direct pointers into the hierarchy representation, so node() also runs in $\mathcal{O}(1)$.

The limitations of these two naïve schemes can be clearly seen with the important query primitives depth() and axis(): Because their implementations have to walk up the tree, they run in $\mathcal{O}(l)$ and are also likely to cause $\mathcal{O}(l)$ cache misses.

4.2.3 Containment-based Labeling Schemes

The first major category of non-naïve schemes are labeling schemes. We begin with the subcategory of *containment-based* labeling schemes. Our discussion of labeling schemes in general assumes that the label column is indexed with a B-tree, where B is the block size of the B-tree. We further make use of that B-tree for the traversal operations, as they could otherwise not be implemented efficiently. Under this assumption, the node() operation runs in logarithmic time, because it has to perform a B-tree lookup to locate the corresponding index entry. Note that the B-tree also adds an $\mathcal{O}(B)$ term to all update operations for these schemes, because the block in which the corresponding index entry lies has to be updated. For simplicity reasons, and because B is a constant independent of the hierarchy size, we omit this term in Figure 4.3.

Column NI in Figure 4.4 shows the classic nested intervals labeling [44, 48, 118], and a variation, the Pre/Post scheme [41], where each node is labeled with its preorder and postorder ranks. We already introduced both in § 2.3.6. These schemes are static (*P2*). Their fundamental weakness is that each insertion or removal requires relabeling $\mathcal{O}(|H|)$ labels on average, as all interval bounds behind a newly inserted bound have to be shifted to make space. Static nested interval schemes of this kind are grouped under the name NI in the tables.

Considering queries, the plain NI and Pre/Post schemes have similar, limited capabilities: For example, we cannot test the important is-parent() predicate, because neither scheme allows us to compute the distance between a node and an ancestor. This severe limitation renders a nested intervals scheme without further fields useless (*P1*). It can be mitigated by either storing the depth of a node or its parent in addition to the interval. In the tables, these variants are named NI-Level and NI-Parent, respectively.

Various mitigations for the nested intervals update problem have been proposed. These are collectively grouped under the name Dyn-NI in the tables.

- Li et al. [63] suggest pre-allocating *gaps* between the interval bounds. Column GapNI in Figure 4.5 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 problem is that relabelings are expensive, and skewed insertions may fill up certain gaps overly quickly and lead to unexpectedly frequent relabelings (*P3*). Also, all *s* nodes in a range or subtree being updated still need to be relabeled (*P2*).

- Amagasa et al. [2] 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. [7] tackle the update problem with their Pre/Size/Level encoding (PSL, see Figure 4.4) by storing the pre-rank() values implicitly as a page offset, which yields update characteristics comparable to gap-based schemes.

- W-BOX [96] uses gaps but tries to relabel only locally using a weight-balanced B-tree. Its skewed update performance is superior to basic gap-based schemes.

- The Nested Tree [115] 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 as well.

Another idea to tackle the update problem for NI is to use variable-length data types to represent interval bounds: For example, the QED [60], CDBS [61], and CDQS [62] encodings by Li et al. are always able to derive a new label between two existing ones, and thus avoid relabeling completely. EXCEL [73] 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. [22] 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 tagged with ‡ in Figure 4.3.

All gap-based and variable-length NI schemes handle *inner node* updates decently by wrapping a node range into new bounds. For example, with GapNI in Figure 4.4 we could insert a parent node (K) above (D), (E), and (F) by assigning it the bounds $[350, 950]$. However, as soon as the node's depth [7] or parent [73] are to be tracked (these variants are named Dyn-NI-Level and Dyn-NI-Parent in the tables), inner node updates turn expensive, as the depths of all s descendants change, and the parent references of all c children of (K) (namely (D), (E), and (F)) need to be adjusted. Updates to subtrees or ranges of size s always require us to modify s labels. Unfortunately, tracking at least the depth is necessary for many queries (*P1*). Thus, containment-based schemes suffer from *P2* and *P3*, which limits their use in dynamic settings.

4.2.4 Path-based Labeling Schemes

We next consider path-based labeling schemes. From this subcategory we already discussed the Dewey [99] labeling scheme (§ 2.3.6), which is the basis of several more sophisticated schemes. Figure 4.5 repeats the example data.

Dewey is not dynamic (*P2*): We can easily insert a new node as rightmost sibling, but in order to insert a node ν between two siblings, we need to relabel all siblings to the right of ν and all their descendants (as indicated by factor f in Figure 4.3). For unordered hierarchies, inserting rightmost siblings is sufficient, but for ordered hierarchies insertion between siblings is a desirable feature. Therefore, several proposals of dynamic path-based schemes try to enhance Dewey correspondingly:

- One prominent representative is Ordpath ([79], see also § 2.3.7). 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 4.4, for example, inserting a sibling between (C) and (H) would result in the label 1.4.1. Note that the dot notation is only a human-readable surrogate; Ordpath actually stores labels in a more compact binary format.

- 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 [113] also aims for a Dewey encoding with shorter label sizes than Ordpath.

- The techniques of [60–62] can also be used to build 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 [22] 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 depth.

Considering updates, the dynamic variants 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 [12], 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*). 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.

4.2.5 Index-based Schemes

To evaluate queries, index-based schemes use special-purpose auxiliary index structures rather than considering just the `Label` (all operations tagged with $'$ in Figure 4.3).

As such they must be tightly integrated into the database system. Their advantage is that they generally offer much improved support for updates.

B-BOX [96] uses a keyless B^+-tree to represent a containment-based scheme dynamically. It has the same update complexity as the BO-Tree in Figure 4.3. However, it represents only lower and upper bounds but does not add information to efficiently determine a node's depth() or parent. It thus has limited query capabilities (*P1*). Its node() implementation runs in $\mathcal{O}(B)$, because it always scans a block of size B when searching for index entries.

DeltaNI, one of the contributions of this research project [31], uses an index to represent a containment-based scheme with depth() support. DeltaNI supports even complex update operations, such as relocating an entire subtree, in $\mathcal{O}(\log |H|)$ time. As an index for *temporal* hierarchies, it is able to capture a whole history of updates (factor u in the figures) and can answer time-traveling queries. It can be used for non-temporal hierarchies as well 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 the evaluation in [33] shows.

Order indexes are a family of index-based schemes we developed in order to overcome the mentioned problems of prior works. An order index in essence represents a nested intervals labeling in a dynamic, skew-resistant data structure. In [32] and [33] we describe three specific implementations of this concept: the AO-Tree based on the AVL tree, the BO-Tree based on the B^+-tree, and the O-List based on a linked list of blocks. 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. The tree-based implementations are able to handle all presented update operations efficiently in logarithmic worst-case time. They avoid degeneration in case of unfavorable update patterns, and provide the query capabilities of labeling schemes with highly competitive performance. By adjusting the configurable parameters, the BO-Tree and the O-List can be tuned towards lower query times or lower update costs. This allows its applications to choose a tradeoff which is appropriate for the application at hand. As we recommend order indexes as the standard dynamic indexing schemes for our hierarchy framework, we give a more detailed overview in § 4.5.

B-BOX, DeltaNI, AO-Tree, and BO-Tree can handle subtree and range relocations with logarithmic worst-case complexity and thus do not show any of the update problems.

4.3 Schemes for Static and Dynamic Scenarios

In order to accommodate a wide variety of application scenarios, we expect any database system that implements our framework to offer a selection of multiple alternative hierarchy indexing schemes. In principle, a suitable scheme could be chosen individually for each hierarchical table. However, the user should not be burdened with the choice; we rather intend the DBMS to pick the optimal scheme *automatically* among the available implementations. The user may more or less indirectly influence the choice

through additional parameters to the hierarchical dimension, in particular the critical `UPDATES` option (§ 3.4). We propose a decision scheme along the following lines:

- For derived hierarchies, which are inherently static, and for hierarchical base tables with `UPDATES = BULK`, the obvious choice is a static labeling scheme. In § 4.4 we introduce a suitable scheme called *Pre/Post/Parent/Level scheme* (PPPL), which is particularly tailored to the requirements of our index interface.
- If the user requires support for complex updates (`UPDATES = SUBTREE`), the BO-tree we discuss in § 4.5 is a safe all-round choice. To further improve query performance without sacrificing update support, the O-List may be a good alternative. A path-based scheme may also perform well, in particular when the user settles for simple updates (`UPDATES = NODE`). However, this choice has to be made by considering the workload at hand.
- If the hierarchical table also has an application-time or system-time temporal dimension, and this dimension is accessed non-trivially by queries in the workload, a temporal indexing scheme like DeltaNI is recommendable. However, the implementation and performance of such a scheme heavily depends on the basic design for temporal dimensions chosen by the specific RDBMS. This is not in scope of this thesis.

More adaptive approaches to choosing the most suitable scheme according to the workload are conceivable, but this is part of future work (see § 7). We proceed with a detailed discussion of the proposed PPPL (§ 4.4) and order indexes (§ 4.5) schemes.

4.4 PPPL: A Static Labeling Scheme

The *Pre/Post/Parent/Level scheme* (PPPL) is an extension of the Pre / Post containment-based labeling scheme which we propose as the default built-in indexing scheme for static or read-mostly scenarios. In these scenarios the hierarchy is loaded once and rarely changed. Except for bulk-loading, no updates are supported.

The `Label` of PPPL is a 4-tuple [pre, post, parent-pre, level] consisting of the pre-rank(), post-rank(), the pre-rank of the parent, and the depth() of the corresponding node. Additionally, the hierarchical table is indexed on pre-rank() and on post-rank() using two simple direct lookup tables, pre-to-rowid[] and post-to-rowid[]. No additional information is needed for `Node` or `Cursor`, so these types are identical to `Label`. Figure 4.4 shows the `NODE` column of an example hierarchy.

Let us examine how the query primitives of our index interface can be implemented with PPPL. The basic access primitives are almost no-ops:

$$
\begin{aligned}
&H.\text{node}(\ell),\ H.\text{cursor}(\ell) && \equiv\ \ell \\
&H.\text{label}(\nu) && \equiv\ \nu \\
&H.\text{rowid}(\nu) && \equiv\ \text{pre-to-rowid}[\nu.\text{pre}]
\end{aligned}
$$

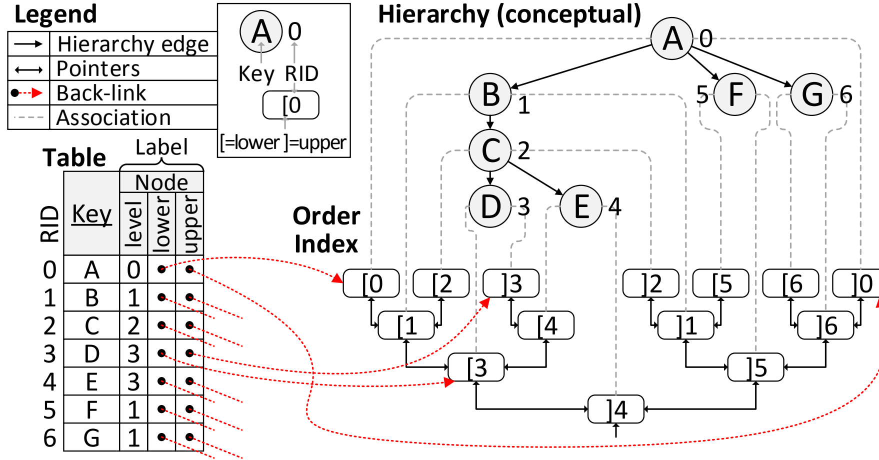


Figure 4.6: A hierarchy represented by an AO-Tree order index [33].

The pre and post fields are sufficient for answering the critical is-before-pre(), is-before-post(), and axis() predicates. The parent-pre field adds support for is-parent().

$$
\begin{aligned}
&H.\text{is-before-pre}(\nu_1, \nu_2) &&\equiv\quad \nu_1.\text{pre} < \nu_2.\text{pre}\\
&H.\text{is-before-post}(\nu_1, \nu_2) &&\equiv\quad \nu_1.\text{post} < \nu_2.\text{post}\\
&H.\text{axis}(\nu_1, \nu_2) &&\equiv\quad \textit{(via is-before)}\\
&H.\text{is-parent}(\nu_1, \nu_2) &&\equiv\quad \nu_2.\text{parent-pre} = \nu_1.\text{pre}
\end{aligned}
$$

The level field adds support for the properties depth(), size(), is-root(), and is-leaf().

$$
\begin{aligned}
&H.\text{depth}(\nu) &&\equiv\quad \nu.\text{level}\\
&H.\text{is-root}(\nu) &&\equiv\quad H.\text{depth}(\nu) = 1\\
&H.\text{is-leaf}(\nu) &&\equiv\quad H.\text{size}(\nu) = 1\\
&H.\text{size}(\nu) &&\equiv\quad H.\text{depth}(\nu) + \nu.\text{post} - \nu.\text{pre}
\end{aligned}
$$

The pre-to-rowid[] and post-to-rowid[] indexes add support for traversal in preorder and postorder, and for the ordinal access primitives.

$$
\begin{aligned}
&H.\varsigma\text{-next}(c) &&\equiv\quad T[\varsigma\text{-to-rowid}[\nu.\varsigma + 1]].\text{Node}\\
&H.\text{next-sibling}(c) &&\equiv\quad H.\text{next-following}(c)\\
&H.\text{next-following}(c) &&\equiv\quad T[\text{pre-to-rowid}[\nu.\text{post} + \nu.\text{level} + 1]].\text{Node}\\
&H.\varsigma\text{-rank}(\nu) &&\equiv\quad \nu.\varsigma\\
&H.\varsigma\text{-select}(i) &&\equiv\quad T[\varsigma\text{-to-rowid}[i]].\text{Node}
\end{aligned}
$$

As these definitions show, most primitives boil down to cheap $\mathcal{O}(1)$ arithmetics on the label itself. This makes PPPL essentially as fast as an indexing scheme can get.

4.5 Order Indexes: A Family of Dynamic Indexing Schemes

The basic idea of order indexes is to represent nested intervals in a dynamic data structure. A central observation on which they are based is that in a nested intervals

encoding, the actual *numerical value* of each lower or upper bound has little importance; what really counts is the *relative order* among the bounds. For example, in Figure 4.4 (p. 76), we can infer that node (F) is a descendant of node (A), because its interval $[8, 9]$ is a proper subinterval of (A)'s interval $[0, 17]$, that is, $0 < 8$ and $9 < 17$. Thus, instead of representing the numbers explicitly, we can use a data structure that directly represents the order relation ($<$) among bounds. *Ordered* data structures such as the classic AVL- or B-trees maintain an order relation among their entries by design. Consequently, ordered data structures can be adapted to represent nested interval encodings *implicitly*, which is exactly what order indexes do.

An *order index* conceptually represents each hierarchy node by two interval bounds and its depth value. These components [lower, upper, level] are stored in the `Label`. However, a lower and upper bound is not an explicit number or other literal, but rather a special link to a corresponding entry in the ordered data structure. We call these links *back-links*, as they refer back from a table row to an index entry, while common secondary indexes merely point the other way from an index entry to a row (through its row ID). The main task of the ordered data structure is to maintain the relative order of its entries—hence the term "*order* index." Each *entry* represents either a lower or an upper bound of a node. Regardless of the underlying ordered data structure, the information an entry conceptually stores is the rowid of the corresponding table row and an is-lower flag determining whether it is the lower or the upper bound of the row. An order index *cursor* is a direct reference (e. g., a pointer) to a specific entry in the ordered data structure. A *cursor* can be obtained from a back-link, although this incurs a non-trivial cost. The `Node` and `Cursor` types are identical (i. e., no further distinction is necessary) and simply wrap an order index cursor.

4.5.1 Order Index Interface

We now discuss the low-level interface an order index O provides for querying. In the following, l is a back-link, and c, c', c_1, and c_2 are order index cursors.

$c \leftarrow O.\text{entry}(l)$ — Returns an order index cursor to the entry for back-link l.

$r \leftarrow O.\text{rowid}(c)$ — Returns the ID of c's associated row.

$O.\text{is-lower}(c)$ — Returns true if c represents a lower bound.

$O.\text{before}(c_1, c_2)$ — Returns true if c_1 is before c_2 in the entry order.

$c' \leftarrow O.\text{next}(c)$ — Returns a cursor c' to the next entry in the entry order.

$O.\text{adjust-level}(c)$ — Returns the *level adjustment* for c.

The starting point for working with an order index is a back-link l, that is, a value from the lower or upper field of a `Label`. Using $O.\text{entry}(l)$, a back-link can be followed to obtain a cursor. (This is in fact the only functionality back-links support.) $O.\text{rowid}(c)$ and $O.\text{is-lower}(c)$ access the information of the entry to which c points. next() is used

to traverse the ordered data structure. The most central primitive is before(), which reflects the order relation implied by the order index.

The level adjustments returned by adjust-level() are a mechanism to maintain depth() information dynamically. The level stored in the Label of a node ν is a relative value. Its actual depth $H.\text{depth}(\nu)$ can be obtained by adding the level adjustment for its lower bound to the stored level. We will elaborate further on this mechanism below.

While the implementations of rowid() and is-lower() are trivial, the implementations of entry(), next(), before(), and adjust-level() differ among the three order index implementations we propose. entry() depends on how back-links are actually represented. next() corresponds to a basic forward traversal of whatever ordered data structure is used. Similarly, before() has to traverse the data structure to detect the relative positions of the two given entries.

4.5.2 The AO-Tree

Figure 4.6 shows an example hierarchy (upper right) and its representation as a hierarchical table and an associated AO-Tree order index. The AO-Tree (AVL-Order-Tree) is one of our three proposed implementations. It is based on a keyless AVL tree.

A few exemplary back-links are shown as red, dotted arrows. An opening bracket denotes a lower and a closing bracket an upper bound; for example,]3 is the entry for the upper bound (is-lower is false) of row #3 (rowid is 3). The more "left" an entry in the AO-Tree is in the figure, the lower its place in the order relation is. For example,

$$\ldots <]4 <]2 <]1 < [5 < \ldots,$$

and $O.\text{before}(]2,]5)$ is true.

Do not confuse hierarchy edges ($\longrightarrow$) and AO-Tree edges ($\longleftrightarrow$): The purpose of the AO-Tree edges is to maintain an ordered, balanced tree. Only the ordering that is implied by this tree is meaningful in the sense that it constitutes an implicit nested intervals encoding. Thus, for example, we could also rotate the entries]3 and [4 in such a way that]3 becomes the right child of [3 and [4 the right child of]3. The implied ordering [3 <]3 < [4 would stay the same.

Note that strictly speaking the is-lower flag would not have to be stored explicitly in each entry, as

$$O.\text{is-lower}(c) \equiv (O.\text{entry}(H.\text{label}(c).\text{lower}) = c).$$

However, implementing is-lower() this way would incur a round trip to the table on each call, with a potential cache miss. We therefore prefer to store is-lower explicitly.

4.5.3 Block-based Order Indexes

Self-balancing binary trees such as AVL trees or red/black trees offer logarithmic complexity for most operations. While this makes them good candidates for an order index structure, their performance in practice is dwarfed by their cache-unfriendliness. We therefore devised two more sophisticated order index implementations: the BO-Tree

and the O-List. Rather than storing one entry per tree node like the AO-tree, they are based on larger blocks of memory that fit B entries each. We therefore call them *block-based* order indexes and B the *block size*.

Block-based data structures afford a technique called *accumulation*. This technique allows us to efficiently maintain the level adjustments. (We originally conceived it in the context of DeltaNI [31] for different purposes.) Accumulation works for any hierarchically organized data structure that stores its entries in blocks, such as a B-tree or even a classic binary tree (where the "blocks" are just single-entry 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 brings along the cost that O.adjust-level(c) becomes linear in the height of the data structure, usually $\mathcal{O}(\log |H|)$. But in return, even for range relocations at most $\mathcal{O}(\log |H|)$ block levels need to be adjusted in order to update the depths of the involved nodes. An as optimization, during an index scan, the level adjustment can be tracked and needs to be refreshed only when a new block starts. This would yield amortized constant time for depth() in this case.

The **BO-Tree** (B^+-Order-Tree) is basically a B^+-tree that has been adapted as follows: The entries are stored in a series of leaf blocks, which each have pointers to the neighboring leaf blocks for faster scans. An entry in a leaf block consists of a row ID and an is-lower flag. (Recall that no keys are needed.) In an inner block, there are no separator keys but only child block pointers. Each block additionally maintains a back-link to its parent block and a block level.

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. 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 c.pos. The back-links to parent blocks are needed because most operations involve leaf-to-root navigation. adjust-level(), for instance, is computed by summing up all block levels on the path from the corresponding leaf entry's leaf block to the root block.

The worst- and best-case complexity of adjust-level() and similar operations is proportional to the tree height, which is in $\mathcal{O}(\log_B |H|)$. Thus, the wider the blocks in the BO-tree, the faster these primitives work. Due to the large logarithm base B, a worst-case complexity of $\mathcal{O}(\log_B |H|)$ is a very favorable asymptotic bound. 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 binary predicate or a depth() query. Since the root block will probably reside in cache, only 2 cache misses are to be anticipated in this case, which makes the BO-Tree a very cache-efficient data structure.

The simpler **O-List** is not a tree structure but merely a doubly linked list of blocks. *Block keys* 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 bocks without having to relabel, 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 treated in a similar manner: Inserting into a full block triggers a split; a block whose load factor drops below a certain percentage (e. g., 40%) is either refilled with entries from a neighboring block or merged with it. adjust-level() 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, and if not, it compares the keys of their blocks. 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 a possibly non-logarithmic update complexity.

A more detailed discussion of the implementation of the AO-Tree, BO-Tree, and O-List indexes is not in scope of this thesis; the reader is referred to [32] and [33].

4.5.4 Representing Back-Links

In block-based order indexes, different designs for the representation of back-links are conceivable. Recall that back-links are references to bounds stored in a `Label`, whose only purpose is that they can be passed to entry() to obtain a corresponding cursor. The task of entry() is to locate an entry in the data structure given the information stored in the back-link.

In the AO-Tree, back-links and order index cursors are identical: they are simply direct pointers to the AVL tree nodes, so $O.\text{entry}(l) \equiv l$. This is only feasible because AVL tree entries never move in memory. For the BO-Tree and the O-List, however, entries are shifted around within their blocks or even moved across blocks by rotate, merge, and split operations.

Suppose we would represent back-links exactly like cursors: by a block pointer and the offset in the block. We call this the ***pos*** approach. It makes entry() a no-op. However, the back-links (i. e., cursors) stored in the `Label` objects (i. e., the `NODE` column) would have to be kept up to date whenever entries are moved out of their place, even when an entry is just shifted around within its block. As any insertion or removal in a block involves shifting all entries behind the corresponding entry, this slows down updates considerably, especially for a larger B. As adjacent entries in the order index blocks are not necessarily linked to adjacent tuples in the table (recall that hierarchy indexes are non-clustered indexes), this would incur random data accesses and thus cause a significant slowdown.

We suggest two alternative approaches for representing back-links in block-based data structures: *scan* and *gap*.

- With the simple ***scan*** approach, a back-link consists only of a [block pointer] indicating the block containing the entry. To actually locate an entry, entry() has to scan the block linearly for the row ID. These block scans add an unattractive $\mathcal{O}(B)$ factor to most queries and thus hinder us from using larger blocks. On the

other hand, *scan* has the advantage that back-links need to be updated only when entries are migrated to other blocks. The *scan* approach is used by B-BOX [96].

- The ***gap*** approach is a compromise between *pos* and *scan*. It again leverages ideas of GapNI: Each entry is tagged with a *block-local key* (e. g., 1 byte) that is unique within the block. A back-link is a [block pointer, key] pair. Initially the keys are assigned by dividing the key space equally among the entries in the 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. entry() uses binary search or interpolation search to locate the entry by its block-local key.

gap is significantly cheaper than *pos*, which effectively relabels half a block, on average, on *each* update. Like with GapNI, an adversary can trigger relabelings by repeatedly inserting 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. Our evaluation in [33] therefore suggests *gap* as a good all-round approach, but order indexes might be configured to use *pos* or *scan* to further optimize certain applications.

4.5.5 Implementing the Query Primitives

All query primitives of our hierarchy index interface can be implemented in terms of the six order index operations, as we show in the following. We begin with the basic access primitives.

$$
\begin{aligned}
&H.\text{node}(\ell),\ H.\text{cursor}(\ell) &&\equiv\ O.\text{entry}(\ell.\text{lower})\\
&H.\text{label}(\nu) &&\equiv\ T[H.\text{rowid}(\nu)].\text{Node}\\
&H.\text{rowid}(\nu) &&\equiv\ O.\text{rowid}(\nu)
\end{aligned}
$$

To understand these and the following definitions, recall that both a `Node` ν and a `Cursor` c of the hierarchy index interface are identical to an order index cursor. However, the order index cursor of a `Node` is guaranteed to always point to the entry of the *lower* bound of the node. In other words, $O.\text{is-lower}(\nu)$ holds for a valid `Node` ν, but not necessarily for a `Cursor`. Some definitions in the following rely on this invariant.

The helper functions to-lower() and to-upper() are given an order index cursor c to some entry, which may represent either the upper or the lower bound of some node ν, and yield a cursor to respective lower or upper bound of node ν:

$$
\begin{aligned}
&O.\text{to-lower}(c) &&\equiv\ \textbf{if}\ O.\text{is-lower}(c)\ \textbf{then}\ c\ \textbf{else}\ O.\text{entry}(H.\text{label}(c).\text{lower})\\
&O.\text{to-upper}(c) &&\equiv\ \textbf{if}\ O.\text{is-lower}(c)\ \textbf{then}\ O.\text{entry}(H.\text{label}(c).\text{upper})\ \textbf{else}\ c
\end{aligned}
$$

Note that all hierarchy index primitives that accept a `Node` also accept a `Cursor`. The following definitions, however, expect a `Node`. If a `Cursor` c_i is actually given, $O.\text{to-lower}(c_i)$ has to be applied first to make sure it points to a *lower* bound.

Binary predicates mainly rely on before():

$$
\begin{aligned}
&H.\text{is-before-pre}(\nu_1, \nu_2) && \equiv && O.\text{before}(\nu_1, \nu_2) \\
&H.\text{is-before-post}(\nu_1, \nu_2) && \equiv && O.\text{before}(O.\text{to-upper}(\nu_1), O.\text{to-upper}(\nu_2)) \\
&H.\text{axis}(\nu_1, \nu_2) = \text{desc.} && \equiv && O.\text{before}(\nu_2, \nu_1) \wedge O.\text{before}(\nu_1, O.\text{to-upper}(\nu_2)) \\
&H.\text{is-parent}(\nu_1, \nu_2) && \equiv && \quad H.\text{axis}(\nu_2, \nu_1) \ = \ \text{descendant} \\
& && && \wedge\ H.\text{depth}(\nu_2) = H.\text{depth}(\nu_1) + 1
\end{aligned}
$$

We next consider the node properties. As noted above, depth() needs to take into account the level adjustment that applies to the block containing ν's lower bound:

$$
H.\text{depth}(\nu) \quad \equiv \quad H.\text{label}(\nu).\text{depth} + O.\text{adjust-level}(\nu)
$$

is-root() is based on depth(). A leaf is detected if the lower bound of the node is immediately followed by an upper bound:

$$
\begin{aligned}
&H.\text{is-root}(\nu) && \equiv && H.\text{depth}(\nu) = 0 \\
&H.\text{is-leaf}(\nu) && \equiv && \neg O.\text{is-lower}(O.\text{next}(\nu))
\end{aligned}
$$

The traversal primitives mainly rely on next():

$$
\begin{aligned}
&H.\text{pre-next}(c) && \equiv && c \leftarrow O.\text{to-lower}(c); \\
& && && \textbf{do } c \leftarrow O.\text{next}(c) \textbf{ until } O.\text{is-lower}(c); \\
& && && c \\
&H.\text{post-next}(c) && \equiv && c \leftarrow O.\text{to-upper}(c); \\
& && && \textbf{do } c \leftarrow O.\text{next}(c) \textbf{ until } \neg O.\text{is-lower}(c); \\
& && && c \\
&H.\text{next-sibling}(c) && \equiv && O.\text{next}(O.\text{to-upper}(c)) \\
&H.\text{next-following}(c) && \equiv && H.\text{pre-next}(O.\text{to-upper}(c))
\end{aligned}
$$

The above discussion omits the size(), rank(), and select() primitives, whose implementation is more involved. Refer to [33] for detailed coverage.

4.5.6 Implementing the Update Primitives

We now discuss how the update primitives of our index interface can be implemented from a high-level point of view. Our aim is to outline how order indexes achieve their logarithmic update characteristics, without diving into the technicalities of any specific implementation. (These are covered in our publications [32] and [33].) The specific algorithms need to manipulate the low-level blocks and entries making up the ordered data structure. They of course differ much between AO-Tree, BO-Tree, and O-List, but are mostly straightforward adaptions of the standard algorithms of the underlying AVL- and B^+-tree data structures. In fact, there is one big advantage the order index adaptions have over the standard algorithms, which allows them to be simpler and more efficient: They *never* have to perform key searches—which would be $\mathcal{O}(\log N)$ for balanced trees of size N—in order to locate entries. Rather, the positions of a node ν or range $[\nu_1, \nu_2]$ to be relocated or removed, as well as the target position p of an inserted node, are always given *a priori*: as cursors indicating positions in the sequence of bounds.

Two things complicate matters, however: First, we must take care of the back-links, the links between the labels in the `NODE` column and their associated order index entries. With 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. Second, some operations need to adjust the block levels so as to maintain a constant effective level adjustment for any entries that are moved out of their place.

Also, order indexes require rebalancing just like the data structures they are based on. The AO-Tree maintains balance through rotations. BO-Tree and O-List split overfull blocks and fill underutilized blocks by taking entries from neighboring blocks or merging two blocks, carefully maintaining the level adjustments in the process.

Implementing the leaf update operations is fairly straightforward: insert-leaf() inserts a lower and an upper bound as adjacent entries into the ordered data structure (using individual insert-before() inserts, which are adaptions of the standard algorithms) and stores a `Label` comprising the two back-links and an initial level in the corresponding table row.

$$
\begin{aligned}
H.\text{insert-leaf}(r, \text{before } c) \quad \equiv \quad & l \leftarrow H.\text{depth}(O.\text{to-lower}(c)); \\
& c_2 \leftarrow O.\text{insert-before}(c, r, \text{false}); \\
& c_1 \leftarrow O.\text{insert-before}(c_2, r, \text{true}); \\
& l \leftarrow l - O.\text{adjust-level}(c_1); \\
& T[r].\text{Node} \leftarrow [c_1, c_2, l];
\end{aligned}
$$

remove-leaf() removes the two entries from the data structure and sets the `Label` in the table row to `NULL`.

$$
\begin{aligned}
H.\text{remove-leaf}(\nu) \quad \equiv \quad & r \leftarrow H.\text{rowid}(\nu); \\
& O.\text{remove}(H.\text{label}(\nu).\text{lower}); \\
& O.\text{remove}(H.\text{label}(\nu).\text{upper}); \\
& T[r].\text{Node} \leftarrow \texttt{NULL};
\end{aligned}
$$

Finally, relocate-leaf() can be done in terms of removal and subsequent reinsertion. Leaf updates have an amortized runtime of $\mathcal{O}(1)$ with the AO-Tree and $\mathcal{O}(B)$ with the BO-Tree (though occasionally some rebalancing work is necessary). With the O-List, leaf updates have constant $\mathcal{O}(B)$ amortized average-case time complexity, but insert-leaf() has a linear worst-case complexity due to its gap-based block keys.

To implement range updates, a move-range() update primitive operating on ranges of bounds is useful:

O.move-range($[c_1, c_2], c_t, \delta$). — Moves a range of entries. Cursors c_1 and c_2 represent the first and last entry in the range 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. Cursor c_t may not be in $[c_1, c_2]$.

move-range() is aware of the level adjustment—the value returned by adjust-level()—for the entries in the range $[c_1, c_2]$. It adjusts the effective level of each entry in the range by δ. With move-range(), relocating a range is straightforward:

$$
\begin{aligned}
H.\text{relocate-range}([\nu_1, \nu_2], \text{before } \nu') \quad \equiv \quad & c_1 \leftarrow H.\text{label}(\nu_1).\text{lower}; \\
& c_2 \leftarrow H.\text{label}(\nu_2).\text{upper}; \\
& c_t \leftarrow H.\text{label}(\nu').\text{lower}; \\
& \delta \leftarrow H.\text{depth}(\nu') - H.\text{depth}(\nu_1); \\
& O.\text{move-range}([c_1, c_2], c_t, \delta);
\end{aligned}
$$

As noted, the target position is simply an order index cursor c_t. In the example p is "before ν'" so c_t is set to a cursor to the lower bound of ν'.

For order indexes based on balanced trees, move-range() can generally be implemented with logarithmic time complexity by relying on two low-level tree operations split() and join(): The former splits a given tree into two; the latter concatenates two trees to one. Both operations are in $\mathcal{O}(\log N)$ for balanced trees, and can maintain their balance. (Since split() and join() algorithms are rarely discussed in textbooks about AVL- and B⁺-trees, we point the reader to their thorough in our publication [33].) With the help of these operations, move-range() can first crop out the range $[c_1, c_2]$ into a separate tree T'; then add the desired level delta δ to the block level of the root block(s) of T', which effectively adds δ to the levels of *all* nodes within the range; rejoin the remaining parts of the original tree; rebalance underutilized blocks at the crop boundaries; reinsert T' at the target position; and finally rebalance again. Note in particular that during move-range() both the number of `level` values of `Label` objects as well as the number of block levels that need to be touched are bounded. The level adjustment mechanism thus enables us to efficiently update the effective levels of all nodes involved in the range relocation.

We have now discussed the *leaf* update primitives as well as relocate-range(). The algorithm for remove-range() is a straightforward variation of relocate-range(). relocate-subtree() and remove-subtree() can trivially be implemented via relocate-range() and remove-range(). Finally, the *inner* update primitives can be expressed in terms of the range operations as described in § 4.1.5.

4.6 Building Hierarchies

In § 4.1.3 we pointed out several situations in which hierarchy indexing schemes are constructed or re-built from scratch. A major use case is the *derived* hierarchical table, where rebuilding is potentially necessary whenever a row in the original source table is updated or a new row is inserted. To avoid painful stalls of the database system in these situations, all stages of indexing scheme construction must be as efficient as possible. In this section we cover this topic in much detail. We first consider the build(H') primitive and describe a practical data structure for its input hierarchy representation H'. Based on that we discuss a largely generic implementation of build(). We then cover the complete process of transforming input data from the adjacency list format into such an intermediate representation H' in order to implement derived hierarchies.

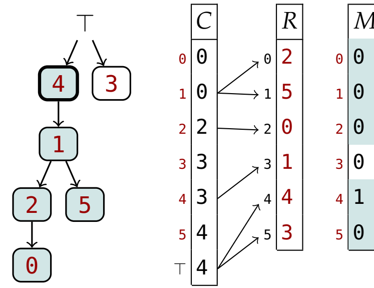


Figure 4.7: The basic format of an Intermediate Hierarchy Representation.

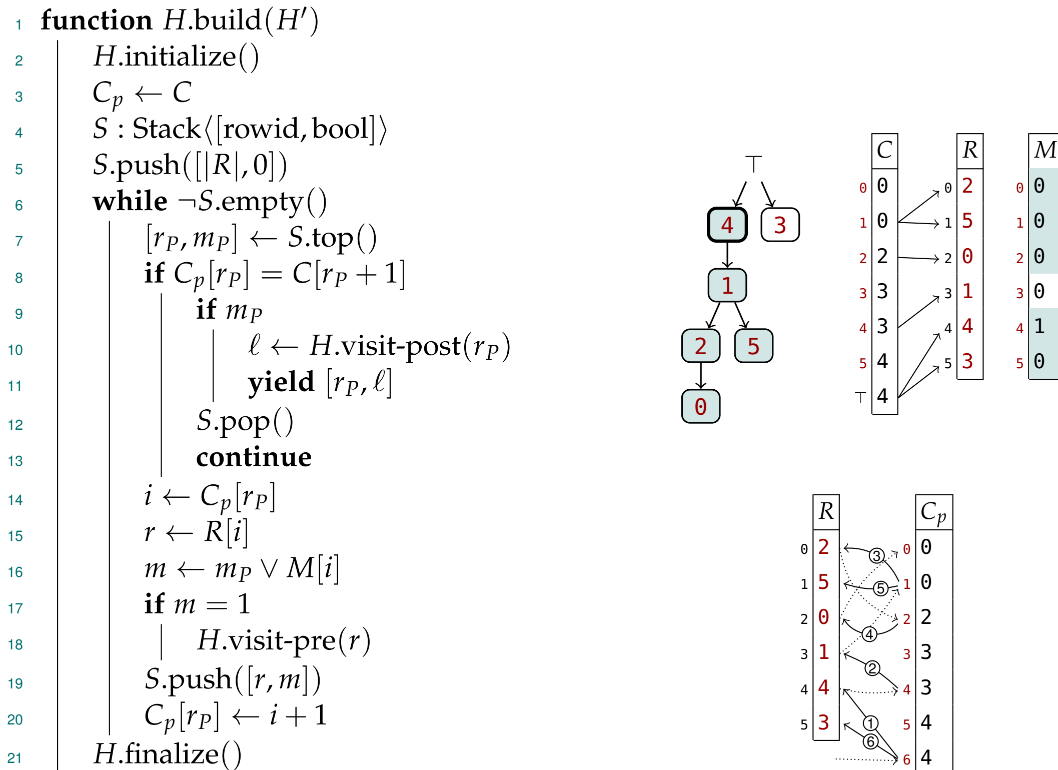
4.6.1 A Practical Intermediate Hierarchy Representation

While the data structure used for H' in the $\text{build}(H')$ operation could be any of the available full-blown hierarchy indexing schemes, this would often be overkill, since H' is usually just used in a throw-away manner as an intermediate representation for bulk building. Upon its construction, there is in fact only one task it has to support well: performing a full depth-first traversal in $\mathcal{O}(|H'|)$ time. We therefore developed a simple and space-efficient special-purpose data structure, the *Intermediate Hierarchy Representation* (Hierarchy IR), which we found to be very effective in practice.

Similar to a full-blown indexing scheme, a Hierarchy IR references the rows of an associated table via row IDs. It arranges these rows in a directed acyclic graph; that is, each row can have one or more parent rows. The IR is thus less restrictive than most of the indexing schemes we explored. This is helpful in particular for input data that does not form a strict tree: We can bring the data into an IR representation first, and then remove non-tree edges to transform it into a strict hierarchy according to a specified given by the user (see § 3.4.3).

At its heart, the data structure is a vector of the row IDs of the hierarchical table, which is sorted and indexed by the respective row IDs of the *parent* rows of its entries. Let T be the associated table. Let $N := |T|$ be the number of rows in T. We assume that the row IDs of T are assigned densely from 0 to $N-1$, although this is not a strict requirement. The representation consists of three vectors C, R, and M. Figure 4.7 shows an example hierarchy and its representation. Here, the table size $|T|$ is 6. The nodes are labeled by the IDs of the associated rows, which range from 0 to 5. The figure also displays the virtual super-root $\top$; it is the parent of all rows that do not have an associated parent row otherwise. The three vectors are defined as follows:

- R is a vector of row IDs of size N. It arranges the row IDs of T by the row IDs of their respective *parent* rows: first the children of row 0 (none in the example), then the children of row 1 (2 and 5), and so on. Rows without a parent (4, 3) appear at the end. The internal ordering of a group of row IDs with the same parent row is significant: it implicitly represents the sibling order of the hierarchy.
- C is a vector of size $N+1$ storing indices of R, that is, it indexes R. Its purpose is to enable us to determine the children of a given row. Its own indices are the



(a) The algorithm. (b) Steps taken for the example Hierarchy IR.

Figure 4.8: A generic algorithm for build(), performing a depth-first traversal over H'.

row IDs of T: For $i < N$, $C[i]$ gives us the index of the R entry that indicates the *first* child of row i, if any. Thus, the children of row i can be found in the range $R[C[i], \ldots, C[i+1] - 1]$. The last entry, $C[N]$, points us to the children of $\top$, that is, all rows without a parent row: they are in the range $R[C[N], \ldots, N-1]$. Note that C is sorted. This is because the row IDs in R are arranged by parent row ID.

- Vector M is optional. It is a bit vector of size N whose indices are the row IDs of T (like C). Its purpose is to mark a subset of T's rows in cases where the hierarchy H' shall be restricted to only those nodes reachable via a given set of *start nodes*. (One such case are derived hierarchies with a specified `START WHERE` clause.) If $M[i]$ is set, row i is a start node. In the example, only (4) is a start node; thus, (3), though represented in H', it not considered to be part of the hierarchy.

A Hierarchy IR can be efficiently constructed from an unordered list of edges, as we will discuss in § 4.6.5.

4.6.2 A Generic build() Algorithm

We now assume a valid Hierarchy IR instance H' is given, and present an algorithm for the H.build(H') operation based on a systematic traversal of this data structure. The implementation of build() is the only part of the construction process that is specific to each indexing scheme. That said, the algorithm we present is still largely generic; it can be used for most indexing schemes we investigated.

The algorithm performs a depth-first traversal of H'. During the traversal it performs a *pre* and a *post* visit of each row: When a node is encountered first, the associated row r is pre-visited, then its descendants are processed, and finally r is post-visited again. The indexing schemes only differ in what is done during the pre/post visits, and in what data needs to be tracked with the visited nodes during the traversal. We therefore encapsulate these actions behind four internal operations:

H.initialize(N) — Initiates bulk building. $N \geq 0$ is the number of nodes in H'.

H.visit-pre(r) — Pre-visits row r.
Precondition: r has not been pre- or post-visited.

$\ell \leftarrow H$.visit-post(r) — Post-visits row r.
Returns the label of the associated node.
Precondition: r has been pre-visited previously.

H.finalize$()$ — Finalizes bulk building.

The following protocol involving these operations must be strictly followed: initialize() must be called first (exactly once), then visit-pre() and visit-post() may be called repeatedly, then finalize() must be called (exactly once). No other index operations on H may be called in between, since it will be in a consistent state only *after* finalize() has been called. Furthermore, the visit() calls must happen in a correctly nested pre/post order, and their given arguments r must be row IDs in the range $[0, N[$. After the post visit of a row r, the `Label` object of the corresponding node for r becomes available and is returned by visit-post().

Figure 4.8a shows the algorithm for build(). Its input is a Hierarchy IR $H' = (C, R, M)$. Its overall output is a stream of [row ID, `Label`] tuples, which provides a `Label` value for each row ID that has actually been included in H. This information is subsequently used to populate a corresponding `NODE` column, as we describe in § 4.6.3. As mentioned, the algorithm essentially performs a depth-first traversal of H'. We maintain a stack S to track the current position in the hierarchy. It contains pairs of a row ID and a boolean mark. The sequence of row IDs on S represents the path from $\top$ to the current node. We use the mark to determine whether the current node is reachable from any start node and thus should be added to H. During the traversal, visit-pre() and visit-post() are called for marked nodes and any of their descendants; all other nodes are traversed but otherwise ignored.

In the initialization phase (l. 2–5) we first make a mutable copy C_p of C. We use C_p to keep track of the "visited" parts of C. We begin the traversal with the super root $\top$ by

pushing its pseudo row ID N onto the stack. We then continue the following process until the stack becomes empty, which means that H' has been traversed completely:

- Inspect the row ID r_P and the mark m_P ("P" stands for "parent") on the top of the stack (l. 7). Use C_p and C to determine the next child of node r_P to visit:

  If $C_p[r_P] = C[r_P + 1]$, then r_P either has no children at all, or we have already visited all its children (l. 8). This means we are done with r_P. If m_P was set, this means that r_P was previously pre-visited. In that case, post-visit r_P, obtain the corresponding `Label` value ℓ, and yield a tuple $[r_P, \ell]$ to the result (l. 9–11). Either way, pop r_P from the stack and continue (l. 12–13).

  If $C_p[r_P] < C[r_P + 1]$, then there is at least one more child of r_P left to be visited next. That child's row ID r is stored in $R[C_p[r_P]]$.

- To visit row r (l. 14–20):

  Determine whether r should be included in the index (l. 16–17). This is the case only if it either is a start node by itself ($M[i]$ is set) or if its parent r_P has already been pre-visited (m_P from the top of S is set). Invoke H.visit-pre(r) only if r should be included in the index (l. 18).

  Push r onto the stack (l. 19). This has the effect that r will subsequently be considered as a parent, and its children, if any, will be visited next (in a depth-first search manner). Finally, increase $C_p[r_P]$ by one (l. 20). Since the "current" child r of r_P has just been visited, this makes $C_p[r_P]$ point to the next child.

Figure 4.8b shows the traversal steps that are taken for the example input. Initially, S contains $\top$, but the super root itself is not pre-visited. The algorithm first pre-visits the nodes $\boxed{4}$, $\boxed{1}$, $\boxed{2}$, $\boxed{0}$ (arrows ①–④ in the figure) and pushes them onto the stack S (not shown). It then determines that $\boxed{0}$ is a leaf, as $C_p[0] = C[1] = 0$. It therefore post-visits $\boxed{0}$, pops it from S and checks $\boxed{2}$ again. However, as $\boxed{2}$ has only one child, now also $C_p[2] = C[3] = 3$, so $\boxed{2}$ is post-visited and popped as well. At this point, $C_p[1] = 1$, but there is another child of $\boxed{1}$ to visit ($C_p[1] \neq C[2]$), namely $R[C_p[1]] = 5$. The algorithm therefore pre-visits $\boxed{5}$ (arrow ⑤). After post-visiting and popping $\boxed{5}$, $\boxed{1}$, and $\boxed{4}$, the algorithm considers $\boxed{3}$ as $R[C_p[6]] = 3$ (arrow ⑥). However, as $\boxed{3}$ is not a start node ($M[3] = 0$), row ID `3` is pushed and subsequently popped without generating actual pre- or post-visits. Finally, $\top$ is popped but not post-visited. At this point the algorithm has pre- and post-visited all nodes in the following order:

$$[\boxed{\top}\ [\boxed{4}\ [\boxed{1}\ [\boxed{2}\ [\boxed{0}\ \boxed{0}]\ \boxed{2}]\ [\boxed{5}\ \boxed{5}]\ \boxed{2}]\ \boxed{1}]\ \boxed{4}]\ [\boxed{3}\ \boxed{3}]\ \boxed{\top}].$$

Constructing the Indexing Scheme. As mentioned before, most indexing schemes we investigated can be constructed during a *single* depth-first traversal of the intermediate representation H'. To adapt the generic build() algorithm for these indexing schemes, we only have to suitably define its four internal operations. Let us consider a few example schemes.

```
1 function H.initialize(N)
2     reset counter c ← 0
3     allocate 𝓛[0, ..., N[
4 function H.visit-pre(r)
5     𝓛[r].lower ← c++
6 function H.visit-post(r)
7     𝓛[r].upper ← c++
8     return 𝓛[r]
```

(a) Nested Intervals

```
1 function H.initialize(N)
2     reset pre, post ← 0
3     allocate 𝓛[0, ..., N[
4 function H.visit-pre(r)
5     𝓛[r].pre ← pre++
6 function H.visit-post(r)
7     𝓛[r].post ← post++
8     return 𝓛[r]
```

(b) Pre/Post

Figure 4.9: Defining the internal functions for bulk building.

```
1 function H.initialize(N)
2     pre ← 0
3     post ← 0
4     allocate pre-to-rowid[0, ..., N[
5     allocate post-to-rowid[0, ..., N[
```

```
1 function H.visit-pre(r)
2     pre-to-rowid[pre] ← r
3     ℓ_P ← 𝓛[r_P]
4     𝓛[r].pre ← pre++
5     𝓛[r].parent-pre ← ℓ_P.pre
6     𝓛[r].level ← ℓ_P.level + 1
7 function H.visit-post(r)
8     post-to-rowid[post] ← r
9     𝓛[r].post ← post++
10    return 𝓛[r]
```

Figure 4.10: Defining the internal functions for bulk building: PPPL.

The simple *nested intervals* and *pre/post* encodings can both be constructed straightforwardly by maintaining a global counter c that is incremented on each pre and post visit. Figure 4.9 shows the pseudo code. A table $\mathscr{L}$ associates each seen node with its (incomplete) label. On visit-pre the lower bound of the node is assigned from c to the label. On visit-post the upper bound is assigned, making the label complete.

For the construction of the *PPPL* scheme we also associate each seen node with its preliminary label. Figure 4.10 shows the pseudo code. We track the current pre-rank and post-rank in global counters. During visit-pre(), we assign the pre-rank to the label. Additionally, we determine the parent node and use it to fill in the parent-pre and level fields. The post-rank is available on visit-post(), making the label complete. The pre-to-rowid[] and post-to-rowid[] indexes are populated on the fly.

Order indexes can be constructed during a depth-first traversal by essentially building up the ordered index structure from left to right: H.initialize() pre-allocates memory and prepares the underlying ordered data structure. H.visit-pre(r) appends an entry for the lower bound to the order index, and enters a link to that entry into the preliminary `Label`. H.visit-post(r) appends an entry for the upper bound and again links to it from the `Label`. H.finalize() constructs the remainder of the data structure—such as

the inner levels in case of the BO-Tree. All three implementations fit into the generic bulk-building algorithm and meet its guaranteed $\mathcal{O}(N \log N)$ worst-case complexity: For the AO-Tree and the BO-Tree, rather than performing one-at-a-time inserts for the entries, we use straightforward adaptions of the common $\mathcal{O}(N)$ algorithms for constructing AVL trees and B$^+$-trees sequentially in a bottom-up manner. The B$^+$-tree algorithm first completes the leaf level, then during finalize() proceeds upwards level by level (see [68] Sec. 15.5.3). The O-List algorithm is simpler, as there are no inner blocks. However, it needs a second pass during finalize() to assign evenly-spaced keys to its blocks. This two-pass algorithm is still in $\mathcal{O}(N)$.

4.6.3 Implementing Derived Hierarchies

We now discuss how to efficiently evaluate the *derived hierarchy* construct, whose basic we defined in § 3.4.2 as follows:

```
HIERARCHY (
   USING source table AS source name
   derived hierarchy spec
   SET name
)
```

A central pre-processing step behind this construct is to transform the relational *source table* into a Hierarchy IR representation H' according to the specifications given by the *derived hierarchy spec*. This step is independent of the type of the target indexing scheme.

Note that our focus is on data that is already in a relational table. That said, it is well conceivable to also support deriving hierarchies from structured text such as XML and JSON. Reading such data in document order is effectively a depth-first traversal, and it is straightforward to generate corresponding visit-pre and visit-post events on the fly. Such a traversal is already all functionality we require from H'.

Regardless of the actual input format that is specified by the *derived hierarchy spec*, the general process to evaluate a *derived hierarchy* construct is as follows:

1. Evaluate *source table* and materialize the result into a temporary table T. If the source table is a subquery or a view, this involves translating the query into an optimized plan with a materialization operator to store the result. All columns the user chooses to select in that query or view will be included in the final hierarchical table.

2. Evaluate the *derived hierarchy spec*. (We discuss this for the adjacency list format in § 4.6.4.) The result of this operation is a Hierarchy IR object H' associated with T.

3. Create and populate the new hierarchy index H from H' using the H.build(H') algorithm of § 4.6.2. The output of this operation is a stream of [row ID, `Label`] tuples, associating rows of T with `Label` values corresponding to the created nodes.

4. Allocate the new `NODE` column and append it to T. Populate it from the output of the previous operation. For rows that are not associated to a `Label`, set the `NODE` value to `NULL`. — T is the final hierarchical table.

	T1		
	ID	PID	Ord
0	'A'	'C'	1
1	'B'	'E'	1
2	'C'	'B'	1
3	'D'	NULL	2
4	'E'	NULL	1
5	'F'	'B'	2

(a) Input table

```
HIERARCHY (
   USING T1 AS cc
   START WHERE PID IS NULL
   JOIN PARENT pp ON cc.PID = pp.ID
   SEARCH BY Ord
   SET Nod
)
```

(b) *derived hierarchy* syntax

Figure 4.11: Example derived hierarchy for the adjacency list input format.

The final step depends much on the architecture of the database engine at hand. However, no special logic should be needed: As `NODE` is a normal column, the usual updating mechanisms can be used.

4.6.4 Transforming Adjacency Lists

We now concentrate on how to efficiently implement Step 2 of the described process for the adjacency list format, which is the most common type of input. Let us revisit the syntax of § 3.4.3:

```
HIERARCHY (
   USING source table AS source name
  [START WHERE start condition]
   JOIN PRIOR parent name ON join condition
  [SEARCH BY order]
   SET node column name
)
```

Figure 4.11a shows an example source table named T1. Figure 4.11b shows a `HIERARCHY` clause that derives a hierarchy from T1.

The transformation consists of two steps: The first step is to extract an *edge list* from the source table according to the specifications. The second step is to efficiently transform this edge list into a Hierarchy IR. An important design goal is to reuse existing relational operators for as many aspects as possible in order to leverage their proven and optimized implementations.

The detailed process for Step 2 is as follows:

2.1 To determine the edges of the hierarchy, evaluate an outer self-join on T according to the *join condition*: 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. We use an *outer* join to not preclude nodes without a parent node. In the absence of a *start condition*, these nodes are by default the roots of the hierarchy.

```
T1 AS C LEFT OUTER JOIN T1 AS P ON C.PID = P.ID
```

r_C	C.ID	C.PID	C.Ord	r_P	P.ID	P.PID	P.Ord
0	'A'	'C'	1	2	'C'	'B'	1
1	'B'	'E'	1	4	'E'	NULL	1
2	'C'	'B'	1	1	'B'	'E'	1
3	'D'	NULL	2	NULL	NULL	NULL	NULL
4	'E'	NULL	1	NULL	NULL	NULL	NULL
5	'F'	'B'	2	1	'B'	'E'	1

(a)

r_C	r_P	m
2	1	0
4	NULL	1
1	4	0
0	2	0
3	NULL	1
5	1	0

(b)

Figure 4.12: Intermediate results during the evaluation of a *derived 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. Furthermore, the r_C are not necessarily unique: While we would expect an $N : 1$ join for an adjacency list that represents a strict hierarchy, this need not necessarily be the case with the given data.

In our example, this step results in the table of Figure 4.12a.

2.2 If *order* is specified, use a Sort operator to sort the join result accordingly. (If *order* is missing, use an implementation defined ordering operation to ensure a deterministic order.)

2.3 If a `START WHERE` clause is specified, use a Map operator to explicitly evaluate the *start condition*. This results in a boolean column m (for *start mark*), which marks all rows satisfying the condition. Note the start condition may reference the *parent* row. This is why we have to evaluate it after the join.

In the absence of a `START WHERE` clause, mark the rows that had no join partner in the outer join.

2.4 Remove all columns except for r_P, r_C, and m.

The result of this step is a stream of marked parent/child pairs (i. e., edges) in the desired sibling order. In our example, after adding column m, sorting the table by `C.ord`, and removing columns `C.*` and `P.*`, we obtain table Figure 4.12b.

2.5 Construct H' from the stream of marked edges. We encapsulate this process into a new relational operator, *Edge List Transformation*, which is described next.

With the exception of the last step (Edge List Transformation), we are only reusing existing query processing operators and facilities. In terms of relational algebra, the process of executing our example `HIERARCHY` clause could therefore look as follows:

$$T \rtimes T \longrightarrow \text{Sort} \longrightarrow \text{Map}\,[m] \longrightarrow \text{Project}\,[r_C,\ r_P,\ m]$$
$$\longrightarrow \text{Edge List Transformation} \longrightarrow H.\text{build}(H') \longrightarrow \text{Update}.$$

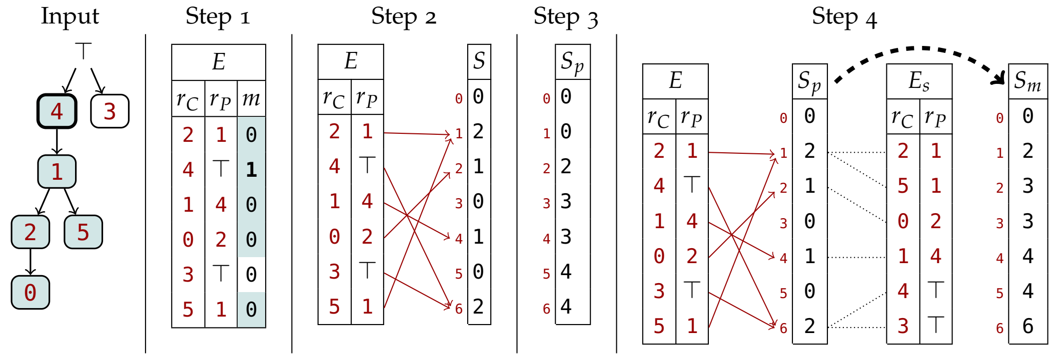


Figure 4.13: Executing the bulk-building operator on an example hierarchy.

4.6.5 Transforming Edge Lists to Hierarchy IR

The Edge List Transformation operator is the final step in transforming a table in the adjacency list format into a Hierarchy IR. It converts an unordered edge list (consisting of parent/child row ID pairs) into the ordered and indexed Hierarchy IR presentation. As such it has applications beyond the adjacency list transformation as well.

The operator consumes a stream of $[r_C, r_P, m]$ tuples. Their order is significant: it implicitly reflects the intended sibling order. The operator's output is the corresponding Hierarchy IR representation H'. In the process, it also identifies non-tree edges, to the system can detect early if the data does not represent a strict hierarchy. The operator is designed for efficiency: it runs in $\mathcal{O}(N)$ time, where N is the size of its input stream, that is, the number of edges in the hierarchy.

Figure 4.13 illustrates the steps of the algorithm. To the left, our example hierarchy is shown again. Figure 4.14 shows the complete pseudo code, which proceeds as follows:

2.5.1 Materialize all edges into an vector E. In the process, track the highest row ID $r_{\max}$ encountered in the input. Set $r_\top$ to $r_{\max} + 1$.

$r_\top$ is a "virtual" row ID we assign to the super root $\top$. In the figure, the maximum row ID is 5, so $r_\top = 6$.

The purpose of the following two steps is to collect information needed to sort E by r_P as efficiently as possible. Note that since $\top$ has been assigned the highest row ID, edges without a parent will be placed at the back.

2.5.2 Count the number of children each node has. In the process, identify multi-parent nodes to ensure there are no non-tree edges.

To this end, allocate a vector C of size $r_\top + 1$. Each entry $C[i]$ stores the child count of the node with row ID i. Nodes without a parent (r_P is `NULL`) are counted in the last slot $C[r_\top]$. Additionally, allocate a bitvector B of size $r_\top$, initially set to false, to mark encountered r_C values. Once an r_C value is encountered more than once (i.e., $B[r_C]$ is found to be already set when it is encountered), a non-tree edge $r_P \to r_C$ is identified. This case can be handled according to a policy

specified by the user: by simply omitting the edge, or by raising an error and aborting the entire process. After this step, B can be disposed again.

In the example, node (1) has two children, so $S[1] = 2$. There are two nodes, (3) and (4), without a parent, so $S[r_\top] = 2$. The figure omits the bitvector B for brevity reasons.

2.5.3 Compute the prefix sums over the array of counts C:

$$C_\Sigma[k] := \sum_{i=0,\,\ldots,\,k-1} C[i].$$

Note the sums "lag behind" by one element, so $C_\Sigma[1] = 0$. The count $C[1] = 2$ first contributes to $C_\Sigma[2]$.

As C is subsequently not needed anymore, the computation can happen in place to increase efficiency: Iterate over C while maintaining a running prefix sum of the counts, and store the prefix sums within C itself.

2.5.4 With the help of C (alias C_Σ), sort vector E by r_P, producing the two sorted vectors R (from $E.r_C$) and M (from $E.m$). (The information of $E.r_P$ is no longer needed.)

To do the sorting, first make a copy C' of C, since we need to modify C' in the process but still need the original C state for the Hierarchy IR. Then, iterate over E; for each tuple $[r_C, r_P, m]$, the target position j in R and M is given by $C[r_P]$. Increment $C[r_P]$ after moving each tuple.

In the example, node 2 is sorted into $R[0]$ as $C[1] = 0$. Then $C[1]$ is incremented to 1, which results in node 5 being subsequently sorted into $R[1]$.

After the algorithm, C_Σ and the sorted R and M vectors constitute the Hierarchy IR H' and can be passed to H.build(H') algorithm.

Note that the final step is a "perfect" bucket sort: Thanks to the preprocessing steps, the algorithm can directly determine the final target position of each tuple. Its asymptotic worst-case time complexity is $\mathcal{O}(N)$, whereas common comparison-based sort algorithms are in $\mathcal{O}(N \log N)$. Only five simple operations per tuple are executed, making this step extremely fast. Furthermore, the sort is *stable*: that is, the relative order among rows with identical r_P values is preserved. In the example, node 2 is guaranteed to remain before node 5. This is important because otherwise the desired sibling order (as indicated via `SEARCH BY`) would be destroyed.

Late Sorting. When a `SEARCH BY` term is specified, the transformation process as described performs a complete Sort before executing the bulk-build. This is one of the most expensive steps of the whole process.

An alternative approach is to defer sorting until *after* the bucket sort. To enable this, all fields appearing in the `SEARCH BY` term—rather than just r_C and r_P—must be included in the edge list E, which slows down the bucket sort due to larger tuples. The

```
   // Step 1
1  materialize input into E : {[r_C, r_P, m]}_b
2  r_⊤ ← 1 + max r_C or r_P in E
   // Step 2
3  C[0, ..., r_⊤] ← 0
4  B[0, ..., r_⊤] ← false
5  for [r_C, r_P, ·] ∈ E
6  |  i ← (r_P = NULL ? r_⊤ : r_P)
7  |  C[i] ← C[i] + 1
8  |  if B[r_C] ∨ (r_P = r_C)
9  |  |  non-tree edge r_P → r_C found
10 |  B[r_C] ← true

   // Step 3
1  Σ ← 0
2  for i ∈ ⟨0, ..., r_⊤⟩
3  |  c ← C[i]
4  |  C[i] ← Σ
5  |  Σ ← Σ + c
   // Step 4
6  C′ ← C
7  allocate R[0, ..., r_⊤[
8  allocate M[0, ..., r_⊤[
9  for [r_C, r_P, m] ∈ E
10 |  i ← (r_P = NULL ? r_⊤ : r_P)
11 |  j ← C′[i]++
12 |  R[j] ← r_C
13 |  M[j] ← m
14 return C, R, M
```

Figure 4.14: Pseudo code for the *Edge List Transformation* operator.

advantage is that not *all* rows but only rows within each bucket then have to be sorted by the `SEARCH BY` columns, which should speed up the sorting step considerably.

On the other hand, the original approach is appealing in that the implementation of the Edge List Transformation operation remains simple and compact. Furthermore, since the available Sort operator of the system is reused for the `SEARCH BY` sorting step, we expect it to be highly optimized and potentially parallelized. We therefore recommend to enable the late sorting option only if the Sort step appears to be the bottleneck of the transformation process, and stick to the original approach otherwise.

Iterative Approach. The approach we discussed so far includes the *entire* hierarchy in the Hierarchy IR and performs a *complete* traversal of H', regardless of whether a potentially selective `START WHERE` clause is present or not. This way the full work would be done even if—in an extreme case—only a few leaf nodes qualify as start nodes.

An different approach to the adjacency list transformation is to iteratively descend from the qualifying start nodes, thus "discovering," processing and traversing *only* the nodes that are reachable by the start nodes:

1. Evaluate and materialize T (as is), and determine the start rows R_0.

2. Iteratively use the join condition to select the next level of rows from T, initially starting from rows in R_0, in order to eventually enumerate all (and *only*) the reachable nodes. Output corresponding edges during each iteration.

3. Convert the edge list into a Hierarchy IR using the Edge List Transformation operation as is.

However, as our experiments indicate (§ 6), a recursive join can be very expensive in contrast to an ordinary join, so the recursive variant should only be chosen if it is very certain the final hierarchy H (spanned by the start rows R_0) is very small in comparison to the full intermediate hierarchy H' we would otherwise construct. Unfortunately, the final size of H is not easy to predict, since the size of H is not related to the number of the start nodes $|R_0|$: Suppose, for example, R_0 contains only a single row r_0, so a naïve query optimizer might choose the recursive algorithm. If r_0, however, happens to be the *only* root of H, then $|H| = |H'|$ and the optimizer's choice would be very bad. We therefore do not suggest using the recursive algorithm, unless derived hierarchies with exceptionally selective `START WHERE` clauses become a bottleneck.

4.7 Handling SQL Data Manipulation Statements

Our extensions for data manipulation (§ 3.5) realize updates to the hierarchy structure via `INSERT`, `UPDATE`, and `DELETE` statements that provide appropriate node anchors as values for the `NODE` field. The anchors may be `ABOVE(`ℓ`)`, `BELOW(`ℓ`)`, `BEFORE(`ℓ`)`, or `BEHIND(`ℓ`)`. The special value `NULL` excludes a row from the hierarchy. The special value `DEFAULT` is automatically translated to a specific anchor (e. g., "`BEHIND` the last root").

The following table summarizes how the different DML statements affect the hierarchical table T on the one hand, and the associated indexing scheme H on the other hand. The `NODE` field is assumed to be named Node.

- **`INSERT` row**. Insert a new row r into T, with the Node field initially set to `NULL`. If the insert sets Node to `NULL`, the hierarchy H is not affected. If the anchor is `ABOVE`, insert a node using H.insert-inner$(r, [\nu', \nu'])$, where ν' is the target node of the anchor. Otherwise, insert a leaf using H.insert-leaf(r, p) and a corresponding position p. (These primitives populate the label stored in Node accordingly.)

- **`UPDATE` row r.** Update the fields of r as usual. Obtain a `Node` object ν for the label stored in the Node field of row r. Check whether ν is a leaf. If the update sets Node to `NULL`, invoke H.remove-leaf(ν) if ν is a leaf, otherwise H.remove-inner(ν). If the update sets Node to some *anchor*, translate it into a suitable target position p and invoke H.relocate-leaf(ν, p) or H.relocate-subtree(ν, p), depending on whether ν is a leaf.

- **`DELETE` row r.** Obtain the `Node` object ν for the label stored in the Node field of row r. Invoke H.remove-inner(ν) or H.remove-leaf(ν) depending on H.is-leaf(ν), then delete row r.

On an `INSERT`, the row needs to be physically created first, whereas on a `DELETE`, the corresponding index call needs to happen *before* physically removing the row, since the row's label may need to be accessed to execute the remove operation.

In addition to these constraints, the system needs to make sure to keep the node–row associations up to date at any time by carefully issuing all pending relink() calls *before* invoking any index primitives, as discussed in § 4.1.3. (Recall that update primitives may access arbitrary labels, so there must not be any stale row ID references at any time.) The circumstances in which relink() calls are needed are system-dependent. Some systems guarantee stable row IDs, in which case no relinking is needed at all.

Invalidation issues are another intricacy the system has to take into account. Calling an index primitive may not only change the `Label` values in the (original) `NODE` column, it may also invalidate any existing temporary `Node` or `Cursor` objects. Only the `Label` objects in the original `NODE` column are guaranteed to be kept up to date. Therefore, the `Node` arguments needed for the update primitives should be obtained through the row ID *immediately before* the respective primitives are called. The fact that `Label` values get invalidated by updates also means that if any such values are present in scratch variables of running SQL scripts, or in the system cache (e. g., materialized views containing a `NODE` field), those items have to be invalidated.

Outlook: Handling Concurrent Transactions. As indexing schemes in general store non-redundant information and are closely tied to tables via direct row–node associations, some care is needed to realize SQL transactions in an ACID-compliant way. Essentially, when one transaction modifies the hierarchy, other transactions must be prevented from seeing these updates before the writing transaction commits. At the same time it must be possible to revert to the original hierarchy state in case of an abort. With the exception of DeltaNI, the indexing schemes we discussed capture only a single, namely the most recent, state of the hierarchy. This makes it hard to allow for concurrent access with strong isolation guarantees. Furthermore, the protocols to handle such concurrency and isolation issues differ highly from system to system. For every database engine, a custom solution essentially has to be crafted. We can thus only give a brief outlook of conceivable approaches:

- A trivial approach is to abort any transaction that accesses, or has accessed, a hierarchy index that is being modified concurrently by a writing transaction.
- Another option is to use a second, static indexing scheme to create a read-only snapshot of the original hierarchy state as soon as a transaction attempts to modify the hierarchy. This allows older read-only transactions to access the original state, and in case the modifying transaction gets aborted, the changes can be undone by bulk-loading the original (dynamic) index from the snapshot.
- Another viable approach is to use indexes with multi-versioning capabilities (such as DeltaNI) to capture the relevant recent history of the hierarchy structure. Transactions can then use time-traveling to access the version they are supposed to see. This also integrates well with systems based on multi-version concurrency control (MVCC, [3]), where updates create new versions of the involved rows and retain the old versions in their original location. MVCC essentially adds a temporal dimension to the table, and a temporal hierarchy index can natively

capture that. The downside is that multi-versioning complicates the index logic significantly and incurs considerable performance penalties. Then again, one gets support for system-time temporal hierarchical tables at no extra cost.

- An interesting approach inspired by the XQuery Update Facility (XUF, [112]) is to *defer* the application of structural updates to the hierarchy until commit time. Rather than executing the updates to the `NODE` field and the index immediately, they are collected in an *edit script* ("pending update lists" in XUF). When the transaction commits, the complete edit script is checked for conflicts—such as the same node being inserted, relocated, or removed more than once—and atomically applied to the index. This is also a convenient time to normalize and group the operations: For example, individual node operations can be rewritten to more expressive subtree and range operations in order to reduce the number of operations in total and to leverage the update capabilities of more sophisticated indexing schemes. Unfortunately, this approach significantly impacts the programming style, as structural updates remain *invisible* even to the updating transaction prior to the commit, so the user cannot build on prior changes when making a series of changes. However, the key advantage of this approach is that the hierarchy index remains unchanged during the whole transaction, which elegantly circumvents many concurrency issues.

A further discussion of these approaches falls out of the scope of this thesis, especially since they are highly system-dependent. However, we do consider them an interesting topic for future research.

5

Query Processing on Hierarchical Data

This chapter covers the process of translating the query language extensions introduced in § 3 into executable query plans. We first discuss which operators are needed on the relational algebra level (§ 5.1). Most of our SQL extensions take the form of functions operating on the abstract `NODE` data type and as such do not actually extend the syntax and semantics of SQL. Thus, it comes at no surprise that we can rely mostly on existing relational algebra operators in their translation. On the physical algebra level (§ 5.2), the key to efficient query evaluation is a set of special-purpose hierarchy operators. Unlike many existing works (§ 5.3), our algorithms are not tied to specific types of labeling schemes. They rely entirely on the abstract `Label`, `Node`, and `Cursor` concepts and the query primitives of the interface presented in § 4.1.2. This way, a lot of complexity is hidden, and query processing can use the same algorithms irrespective of the underlying indexing scheme.

5.1 Logical Algebra Operators

We first investigate the translation of queries against hierarchical tables on the relational algebra level. Part of the design goals for our language extensions was to be able to evaluate most constructs without having to introduce many new relational operators. Thanks to this, creating (logical) query plans is fairly straightforward and can rely on well-known techniques. Most of our query language constructs result in basic Map (§ 5.1.3) and Join (§ 5.1.4) operations. For evaluating hierarchical windows we reuse the concept of Groupjoin (§ 5.1.5). Only for hierarchical windows with recursive expressions, a new operator is needed: Structural Grouping (§ 5.1.6). For each operator, we introduce our notation and discuss its uses by the example of simple SQL queries.

5.1.1 Preliminaries

Although we assume the reader to be fairly familiar with the general concepts of relational algebra, we first clarify some of the less common notations. Generally, we follow the style used by Moerkotte et al. (e. g., [74]).

A relation is a bag (multiset) of tuples. We denote a set by $\{\}$ and a bag by $\{\}_b$. For an expression e that evaluates to a tuple or a relation, $\mathcal{A}(e)$ gives us the set of attributes

---

The material in this chapter on processing structural grouping queries has previously been published in [10]. High-level discussions of query processing on hierarchical tables have also appeared in [9] and [33]. This chapter extends these discussions and also includes additional, novel material.

provided by the tuple or relation, and $\mathcal{F}(e)$ gives us the set of free (unbound) attributes in the expression. We use $\perp_\tau$ as a short-hand for a tuple of type τ that consists only of NULL values.

The $\circ$ operator is overloaded in different contexts: For tuples $t_1 : \tau_1$ and $t_2 : \tau_2$, the notation $t_1 \circ t_2$ means tuple concatenation. When applied to tuple *types* rather than concrete tuples (e. g., $\tau_1 \circ \tau_2$), the operation $\circ$ yields the type of the concatenated tuple. When applied to sequences, it yields the concatenated sequence.

Given a bag-valued expression e (i. e., the expression produces a bag of elements) and a variable name t, the notation $e[t]$ binds the elements to the variable; formally, $e[t] := \{[t : x] \mid x \in e\}$.

We use a number of well-known relational algebra operators: selection σ, projection Π, duplicate-eliminating projection Π^{D}, map χ, the common join operations ⋈, ⟗, ⟕, ⟖, ⋉, ⋊, ▷, ◁, as well as the dependent join operators ⧑, ⧒, ⋉, ⋊, ▶, ◀.

The following conventions apply for the remainder of this chapter: We assume T is a hierarchical table containing a label attribute named ℓ with the underlying hierarchy index H; formally, T is of type $\{\tau_T\}_b$, where τ_T is some type that has a field ℓ : $\texttt{Label}^H$. By convention, the symbols e, e_1, e_2, et cetera denote arbitrary input expressions producing some relations. Symbols ν, ν_1, ν_2, et cetera denote names of $\texttt{Node}^H$ attributes. Symbols τ, τ_i, et cetera denote tuple types.

5.1.2 Expression-level Issues and Normalization

On the SQL level, our set of hierarchy functions and predicates is intentionally somewhat redundant. The indexing scheme interface, however, exposes a more minimal set of primitives. While the SQL-level functions and predicates operate on NODE values of data type $\texttt{Label}^H$, the index primitives operate only on $\texttt{Node}^H$ and $\texttt{Cursor}^H$ objects. This distinction between Label and Node/Cursor is invisible to the user. To account for the differences, some transformation and normalization has to take place as a first step during query compilation.

Canonicalization of Hierarchy Functions and Predicates. A Label ℓ can occur in the SQL statement either in a hierarchy function, a hierarchy predicate, or a hierarchical window specification. For each distinct Label field ℓ used, we have to make the corresponding Node ν available. The easiest way to achieve this is to generate explicit calls to an internal TO_NODE conversion function whenever a label is accessed:

```
FUNC(ℓ)             ⇝  FUNC(TO_NODE(ℓ))
PRED(ℓ1, ℓ2)        ⇝  PRED(TO_NODE(ℓ1), TO_NODE(ℓ2))
ℓ1 = ℓ2             ⇝  TO_NODE(ℓ1) = TO_NODE(ℓ2)
HIERARCHIZE BY ℓ    ⇝  HIERARCHIZE BY TO_NODE(ℓ)
```

The TO_NODE function maps directly into a node() index call. This first rewriting step can be done already during parsing with minimal implementation efforts. Alternatively, the same could be achieved during the subsequent semantic analysis step, at the time the ℓ_i references are resolved.

Let us next consider the translation of SQL hierarchy functions `FUNC(ℓ)` or `FUNC(ν)` to corresponding index invocations. This is straightforward:

`TO_NODE(ℓ)`	$\rightsquigarrow$	H.node(ℓ)
`IS_LEAF(ν)`	$\rightsquigarrow$	H.is-leaf(ν)
`IS_ROOT(ν)`	$\rightsquigarrow$	H.is-root(ν)
`DEPTH(ν)`	$\rightsquigarrow$	H.depth(ν)
`SIZE(ν)`	$\rightsquigarrow$	H.size(ν)
`DEGREE(ν)`	$\rightsquigarrow$	H.degree(ν)
`HEIGHT(ν)`	$\rightsquigarrow$	H.height(ν)
`PRE_RANK(ν)`	$\rightsquigarrow$	H.pre-rank(ν)
`POST_RANK(ν)`	$\rightsquigarrow$	H.post-rank(ν)

`NULL` values simply propagate through the functions; for example, `TO_NODE(NULL)` is `NULL`, and `DEPTH(NULL)` is still `NULL`. However, the index primitives internally expect valid `Label`, `Node`, or `Cursor` argument objects, which must not be `NULL`. In situations where they could possibly be null, the query compiler has to make sure this is handled by generating appropriate checks. In terms of SQL, it conceptually has to rewrite the function invocation to "`CASE WHEN ν IS NULL THEN NULL ELSE FUNC(ν) END`."

Let us now consider the binary hierarchy predicates `PRED(ν1, ν2)`. To handle them effectively during query optimization, we eliminate "redundant" predicates by rewriting all predicates to uniform axis checks:

`IS_ANCESTOR(ν1, ν2)`	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{ancestor}\}$
`IS_ANCESTOR_OR_SELF(ν1, ν2)`	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{ancestor, self}\}$
`IS_DESCENDANT(ν1, ν2)`	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{descendant}\}$
`IS_DESCENDANT_OR_SELF(ν1, ν2)`	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{self, descendant}\}$
`IS_PRECEDING(ν1, ν2)`	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{preceding}\}$
`IS_FOLLOWING(ν1, ν2)`	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{following}\}$
$\nu_1 = \nu_2$	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \in \{\text{self}\}$
ν_1 <> ν_2	$\rightsquigarrow$	$H.\text{axis}(\nu_1, \nu_2) \notin \{\text{self}\}$

Recall from § 4.1.2 that two given nodes can be on exactly one of five axes

$$\text{Axes} := \{\text{preceding}, \text{ancestor}, \text{self}, \text{descendant}, \text{following}\}.$$

The parent, child, and sibling axes are not part of the basic five axes. Nodes on the parent and child axes are subsets of the nodes on the ancestor and descendant axes, respectively. Nodes on the sibling axes are a subset of the nodes on the {preceding, self, following} axes. These three axes need special treatment:

`IS_PARENT(ν1, ν2)`	$\rightsquigarrow$	$H.\text{is-parent}(\nu_1, \nu_2)$
`IS_CHILD(ν1, ν2)`	$\rightsquigarrow$	$H.\text{is-parent}(\nu_2, \nu_1)$
`IS_SIBLING(ν1, ν2)`	$\rightsquigarrow$	$H.\text{is-sibling}(\nu_1, \nu_2)$

Again, when one of the arguments can potentially be `NULL`, the query compiler has to add `NULL` checks. In terms of SQL, this precaution conceptually looks as follows:

```
CASE WHEN ν1 IS NULL OR ν2 IS NULL THEN UNKNOWN ELSE PRED(ν1, ν2) END
```

The representation of SQL-level predicates as axis() calls based on *sets* of axes is useful to reason about complex compound predicates. Two operations on axis sets are helpful: First, given a axis set $A \subseteq \text{Axes}$, we can obtain the inverse set $\bar{A} := \text{Axes} \setminus A$. Second, we can "reflect" a set of axes: $\text{reflect}(A) := \{\text{reflect}(a) \mid a \in A\}$, where $\text{reflect}(a)$ maps preceding $\mapsto$ following, ancestor $\mapsto$ descendant, self $\mapsto$ self, descendant $\mapsto$ ancestor, and following $\mapsto$ preceding. With these helpers we can state a set of rules for transforming a complex boolean expression of axis checks:

$$
\begin{array}{lcl}
H.\text{axis}(\nu_1, \nu_2) \in A_1 \wedge H.\text{axis}(\nu_1, \nu_2) \in A_2 & \rightsquigarrow & H.\text{axis}(\nu_1, \nu_2) \in A_1 \cap A_2 \\
H.\text{axis}(\nu_1, \nu_2) \in A_1 \vee H.\text{axis}(\nu_1, \nu_2) \in A_2 & \rightsquigarrow & H.\text{axis}(\nu_1, \nu_2) \in A_1 \cup A_2 \\
H.\text{axis}(\nu_1, \nu_2) \notin A & \rightsquigarrow & H.\text{axis}(\nu_1, \nu_2) \in \bar{A} \\
H.\text{axis}(\nu_1, \nu_2) \in A & \rightsquigarrow & H.\text{axis}(\nu_2, \nu_1) \in \text{reflect}(A) \\
H.\text{axis}(\nu_1, \nu_2) \in \text{Axis} & \rightsquigarrow & \text{true} \\
H.\text{axis}(\nu_1, \nu_2) \in \{\} & \rightsquigarrow & \text{false}
\end{array}
$$

The goal of the transformations is to simplify a complex expression, potentially consisting of multiple conjunctive or disjunctive predicates, into a single non-negated term "$H.\text{axis}(\nu_1, \nu_2) \in A$," where A is as small as possible. For example, from the SQL expression "`IS_ANCESTOR(`ν_1`, `ν_2`) OR (`$\nu_1 = \nu_2$`)`" the rules allow us to obtain

$$H.\text{axis}(\nu_1, \nu_2) \in \{\text{ancestor}, \text{self}\} \quad \text{or} \quad H.\text{axis}(\nu_2, \nu_1) = \{\text{self}, \text{descendant}\}.$$

Either one may be the most convenient choice for the evaluation of the overall query.

Altogether, the outlined transformations allow us to rewrite an arbitrarily complex boolean expression of SQL hierarchy predicates into a potentially simpler expression consisting only of axis(), is-parent(), or is-sibling() checks.

Optimizations on Expression Level. Since invocations of index functions and predicates have a non-negligible cost for most hierarchy index types, a goal of the query optimizer is to minimize the overall number of such calls.

For node() and the unary functions and predicates, ideally one call at most should be issued for every distinct `Label` field being actually accessed. To achieve this, a conventional analysis and elimination of common subexpressions first helps us to eliminate any redundant expressions (such as multiple `TO_NODE()` calls) in a query block. After translating the query into an internal algebra-based representation, there will be a Map operator (see § 5.1.3) that performs the calls. During query optimization, the query optimizer may reorder this operator in two steps to potentially eliminate it: The first step is to push it down as far as possible. In the process, another Map operator for the same call may be found and eliminated. When a Hierarchy Index Scan or a Hierarchy Index Join is encountered at a leaf of the plan, it may be able to handle the call implicitly and thus eliminate the Map. In particular, node() can always be eliminated since HIS and HIJ can produce the `Node` objects directly. In case the call cannot be eliminated, a second step is to reduce the number of calls by pulling the operator up again (across selections), towards the operator that actually consumes the results.

In § 4.1.4 we discussed several relevant properties and equivalences that apply for the hierarchy functions and predicates. These are useful for query optimization in two ways: First, to eliminate redundant terms in logical expressions and thus simplify their evaluation. Second, to derive and introduce further predicates from existing ones, which are subsequently moved around (pushed down or pulled up) in the query plan in order to reduce result sizes early. For example, the equivalences for the unary predicates is-root() and is-leaf() could be applied in both directions: For a set of is-leaf()-filtered nodes, there is no need to explicitly compute degree(), size(), or height(). Vice versa, if we can infer that the depth of a node variable is always 1 or always > 1, we can omit an explicit is-root() check. As another example, if a join algorithm does not support an is-parent() join directly, the equivalence to an *ancestor* join with a depth() filter gives us an alternative way to evaluate it. These optimization techniques are used in advanced SQL query compilers and are also known under the term *predicate inference and subsumption*. As this is a non-trivial, orthogonal topic, we point the reader to [14, 59, 77] and the overview in [85].

5.1.3 Map

In § 5.1.2 we examined the translation of SQL hierarchy functions and predicates into internal calls of index primitives. After this step, the index invocations can in principle be executed *explicitly* in the query plans using the Map operator χ. This basic operator produces new attributes a_i by applying given functions $f_i : \tau \to \tau_i$ to each tuple in the input e, for $a_i \notin \mathcal{A}(e)$. It is commonly defined as

$$\chi_{a_1 : f_1, \ldots, a_n : f_n}(e) := \{t \circ [a_1 : f_1(t), \ldots, a_n : f_n(t) \mid t \in e]\}_b.$$

Example I. The basic query "`SELECT *, DEPTH(Node) FROM T`" is initially translated to

$$e_1 \leftarrow \chi_{\nu : H.\text{node}(\text{Node})}(T); \quad e_2 \leftarrow \chi_{\text{depth} : H.\text{depth}(\nu)}(e_1).$$

Note the first invocation of node(): As described in § 5.1.2, whenever a function or predicate is (explicitly) evaluated, the plan will also contain a node() call to first obtain `Node` objects from the labels stored in the `NODE` field.

Example II. As the index interface supports explicit axis() checks, we can even handle hierarchy joins naïvely via Map. The query

```
SELECT * FROM (e1) u JOIN (e2) v ON IS_DESCENDANT(v.Node, u.Node)
```

can be translated into $\sigma_{a = \text{descendant}}(\chi_{a : H.\text{axis}(v.\nu_2, u.\nu_1)}(e_1[u] \times e_2[v]))$. This assumes the ν_1 and ν_2 fields have already been obtained via node() calls. It first creates the cross-product of the inputs, then uses χ to perform an axis check on each pair, and σ to filter pairs on the *descendant* axis.

Example III. The query "SELECT * FROM T ORDER BY PRE_RANK(Node)" is translated to

$$e_1 \leftarrow \chi_{\nu\,:\,H.\text{node(Node)}}(T); \quad e_2 \leftarrow \chi_{r\,:\,H.\text{pre-rank}(\nu)}(e_1); \quad e_3 \leftarrow \text{Sort}\,[r]\,(e_2).$$

Remarks. The examples attest that Map alone would in theory be sufficient to naïvely translate almost all of our language extensions, with the exception of HIERARCHIZE BY with recursive expressions. However, in the generated code Map results in explicit per-tuple index calls. Plans relying on Map alone would therefore rarely deliver an acceptable performance. Whenever possible, we want to avoid Map or optimize it "away" in favor of the following operators:

- Hierarchy Index Scan (§ 5.2.1). When the plan works directly on the hierarchical table, as in Examples I and III above, this operator can be used (on the physical algebra level) instead of an ordinary table scan and Map. It can sometimes obviate the need to explicitly evaluate certain functions of predicates by enumerating the desired nodes directly.

- Join (§ 5.1.4). When a binary predicate is evaluated for the purpose of a join, as in Example II above, an appropriate join operator should be chosen.

- Sort (§ 5.2.11). When PRE_RANK or POST_RANK appear only in the ORDER BY clause—which is their main use case—then there is no actual need for the expensive computation of the absolute ranks. For sorting purposes, a pairwise comparison via is-before-pre() and is-before-post() is sufficient. In the case of Example III:

  $$e_1 \leftarrow \chi_{\nu\,:\,H.\text{node(Node)}}(T); \quad e_2 \leftarrow \text{Sort}\,[\nu;\ \text{is-before-pre}]\,(e_1).$$

  Even better, the mentioned Hierarchy Index Scan can sometimes be used to enumerate the tuples in preorder or postorder directly.

Cases where Map is still needed are when the result values are explicitly requested by the query, when none of the mentioned alternatives are applicable, or when the plans that use the alternative hierarchy operators are not globally optimal.

5.1.4 Join

The join operation $e_1 \bowtie_\theta e_2$ is simply defined as $\sigma_\theta(e_1 \times e_2)$. It comes in several well-known variants $⟗$, $⟕$, $⟖$, $\ltimes$, $\rtimes$, $\triangleright$, and $\triangleleft$, but we do not repeat their definitions for brevity reasons. In our context we are only interested in joins where θ is a *hierarchy join predicate*. A hierarchy join predicate is a boolean expression that relates two tuples of different relations and features one or more invocations of hierarchy index primitives. Following the normalization step of § 5.1.2, all remaining hierarchy join predicates are of the form

$$H.\text{axis}(\nu_1, \nu_2) \in A, \text{ for } A \subseteq \text{Axes}, \quad \text{or} \quad H.\text{is-parent}(\nu_1, \nu_2), \quad \text{or} \quad H.\text{is-sibling}(\nu_1, \nu_2).$$

Such predicates can appear as join conditions or filter conditions. A conventional query optimizer that does not understand hierarchy join predicates would produce plans based on $\times$, χ, and σ, as shown in Example I of 5.1.3. These plans result in nested loops with an explicit evaluation of the join condition, which is infeasible for all but the smallest input sizes. Instead, we want these instances to be translated into standard joins on logical algebra level, just like ordinary equi joins. This way, the physical hierarchy join operators we discuss in § 5.2 can be considered during algorithm selection. These operators generally do not have to enumerate the complete cross product, and handle the join condition *implicitly* rather than computing it explicitly per tuple pair, which is critical to the query performance of our framework.

The case "$H.\text{axis}(\nu_1, \nu_2) \in \{\text{self}\}$" is special in that it corresponds to an ordinary equi join. Hierarchy predicates using other sets of axes (e. g., $\{\text{descendant}\}$) are more comparable to $\leq$ joins rather than equi joins. For these cases special join operators are needed in physical algebra. Generally, each such operator can handle only a limited set of axis combinations A. For uncommon cases (e. g., $A = \{\text{preceding}, \text{self}, \text{descendant}\}$) the only options may therefore be to construct the union of two supported cases, or to fall back to nested loops. We will discuss the particular cases our physical operators can handle in the respective sections under § 5.2.

Example I. Consider the join–group–aggregate pattern for hierarchical computations:

```
SELECT u.ID, SUM(v.Value)
  FROM T u JOIN T v ON IS_DESCENDANT_OR_SELF(v.Node, u.Node)
 GROUP BY u.*
```

The plan features an inner join $\bowtie_\theta$ over $\theta := (H.\text{axis}(v.\nu_2, u.\nu_1) \in \{\text{self}, \text{descendant}\})$. This particular pattern of a descendant-or-self self-join is very common.

Example II. In real-world queries the types of hierarchy join predicates listed above may not necessarily appear in isolation. Sometimes, queries feature more complex *compound* join predicates. The predicate could be a conjunction or disjunction of multiple hierarchy predicates, and it could contain further terms that are not necessarily hierarchy predicates. For example, we may be interested in descendants *of the same type*:

$$\theta := (H.\text{axis}(t_1.\nu_1, t_2.\nu_2) = \text{descendant}) \wedge (t_1.\text{Type} = t_2.\text{Type}).$$

Or we may wish to query only the *following* siblings of some nodes:

$$\theta := H.\text{is-sibling}(t_1.\nu_1, t_2.\nu_2) \wedge (H.\text{axis}(t_1.\nu_1, t_2.\nu_2) = \text{following}).$$

Also, nothing hinders the user from creating highly uncommon axis checks:

$$\theta := (H.\text{axis}(t_1.\nu_1, t_2.\nu_2) \in \{\text{preceding}, \text{following}\}).$$

None of our algorithms can handle such complex predicates directly. For this reason, in such queries the optimizer has to determine which of the terms of the compound

predicates could potentially be handled by a built-in join operator. The most selective predicate term can then be handled by a join, and the remainder of the predicate by a subsequent selection σ. As this is an orthogonal standard task of algorithm selection, we do not discuss it further.

Example III. Suppose we wanted to select the "lowermost" nodes in an arbitrary subset U of nodes from T. Since U does not necessarily contain *all* nodes from T, we cannot use `IS_LEAF` for this. Rather, we have to use an `EXISTS` subquery: lowermost nodes are those for which no descendants exist in the input.

```
SELECT * FROM U v
 WHERE [NOT] EXISTS ( SELECT * FROM U w WHERE IS_DESCENDANT(w.Node, v.Node) )
```

This is a common use of *semi* or *anti* hierarchy joins. The initial translation of the SQL query is U $\ltimes$ (σ(U)) for the `EXISTS` version and U $\blacktriangleright$ (σ(U)) for the `NOT EXISTS` version. The d-joins can immediately be fused with the σ operators and replaced by proper semi and anti joins, yielding the "perfect" plans U $\ltimes$ U and U $\triangleright$ U. Note that in the case that U contains *all* nodes from T, the same result can be achieved using a simple `IS_LEAF(Node)` filter.

Example IV. The following query performs XPath-style path pattern matching:

```
SELECT DISTINCT w.*
  FROM T u, T v, T w
 WHERE IS_ROOT(u.Node)
   AND IS_DESCENDANT(v.Node, u.Node)  -- θ1
   AND IS_DESCENDANT(w.Node, v.Node)  -- θ2
   AND φu(u) AND φv(v) AND φw(w)
```

This query is comparable to the XPath expression "`/*[`ϕ_u`]//*[`ϕ_v`]//*[`ϕ_w`]`." The initial plan after pushing down selections and introducing join operators is

$$\Pi^{\text{distinct}}_{w.*}(\sigma_{\phi_u}(T[u]) \bowtie_{\theta_1} \sigma_{\phi_v}(T[v]) \bowtie_{\theta_2} \sigma_{\phi_w}(T[w])).$$

By pushing down the projection Π, the inner joins can be replaced by semi-joins:

$$\sigma_{\phi_u}(T[u]) \rtimes_{\theta_1} \sigma_{\phi_v}(T[v]) \rtimes_{\theta_2} \sigma_{\phi_w}(T[w]).$$

Examples III and IV show that self- and anti-joins deserve special care. Significant optimizations are possible: Of the non-retained join side, only the fields accessed by the join condition (i. e., the Node field) need to be projected, and duplicate Node values can be eliminated if `DISTINCT` semantics are desired. Going further, we can apply a "staircase filter" to eliminate Node values which are dominated by others and thus do not contribute to the join result. This technique is discussed in § 5.2.6.

5.1.5 Binary Structural Grouping

The *binary structural grouping* operator we introduce in this section is used for two purposes: First, to optimize a join–group–aggregate query such as Example I of § 5.1.4,

and second, to evaluate a query that specifies a hierarchical window (§ 3.3.3) but does *not* contain a recursive expression.

Definition. Binary structural grouping consumes two input relations $\{\tau_1\}_b$ and $\{\tau_2\}_b$ given by expressions e_1 and e_2, where τ_1 and τ_2 are tuple types. Let θ be a join predicate, x a new attribute name, and f a scalar *aggregation function* $\{\tau_2\}_b \to \mathcal{N}$ for some type $\mathcal{N}$. The signature of the operation is

$$\bowtie\!\!\Gamma^{\theta}_{x:f} : \{\tau_1\}_b \times \{\tau_2\}_b \to \{\tau_1 \circ [x : \mathcal{N}]\}_b.$$

Its definition is

$$e_1 \bowtie\!\!\Gamma^{\theta}_{x:f}\; e_2 := \{t \circ [x : f(e_2[\![_\theta t]\!])] \mid t \in e_1\}_b, \text{ where } e[\![_\theta t]\!] := \{u \mid u \in e \wedge \theta(u,t)\}_b.$$

It extends each tuple $t \in e_1$ by an x attribute of type $\mathcal{N}$, whose value is obtained by applying function f to the bag $e[\![_\theta t]\!]$, which contains the relevant input tuples for t.

Example I. The initial query plans resulting from join–group–aggregate statements are typically variations of

$$\Gamma_{t.*;\, x:f}(e_1[t] \bowtie_{u<t} e_2[u]),$$

where Γ denotes unary grouping and $<$ is a predicate—usually an axis check—that reflects the input/output relationship among tuples. Using $\bowtie\!\!\Gamma$ they can be rewritten to

$$e_1 \bowtie\!\!\Gamma^{<}_{x:f}\; e_2$$

with the same definitions of f and $<$.

In § 2.2.4 we discussed a non-weighted rollup based on an input table Input1 and a table of output nodes Output1 (see Figure 2.4, p. 18). Assuming χ operators that perform the necessary node() calls, a plan to evaluate this example rollup would be

$$\chi(\text{Output1}) \bowtie\!\!\Gamma^{<}_{x:f}\; \chi(\text{Input1}), \text{ with } u < t := (H.\text{axis}(u.\nu,\, t.\nu) = \{\text{self}, \text{descendant}\}),$$
$$f(X) := \sum_{u \in X} u.\text{Value}.$$

Example II. Besides for optimizing $\Gamma(\cdot \bowtie_\theta \cdot)$ plans, we also use $\bowtie\!\!\Gamma$ to evaluate hierarchical windows with non-RECURSIVE expressions. They are translated into binary self-grouping $e[t] \bowtie\!\!\Gamma^{\theta}\; e[u]$, with $\theta(t,u)$ defined as appropriate. Consider our example query from § 3.1:

```
SELECT Node, SUM(Value) OVER (HIERARCHIZE BY Node) FROM Input1
```
II-b

The resulting query plan is

$$e[t] \bowtie\!\!\Gamma^{<}_{x:f} e[u], \text{ with } e := \chi_{\nu : H.\text{node}(\text{Node})}(\text{Input1}), \text{ and } u < t \text{ and } f(X) \text{ as in Example I.}$$

Remarks. Having the query optimizer rewrite the $\Gamma(\cdot \bowtie_\theta \cdot)$ pattern into a binary grouping operator allows us to employ corresponding physical operators and thus overcome the efficiency issues noted in § 2.2.4. This idea is not new to relational algebra but commonly known as *binary grouping* or *groupjoin*. It has been explored in [74] mainly for the equi join setting, together with relevant rewrite rules for query optimization. Our definition of $\bowtie\!\Gamma$ is a minor adaption of [74]'s definition of the binary grouping operator. Its semantics are those of left outer join; it can however be adapted to inner join semantics as well.

For hierarchical windows, the join condition θ is defined by the system in terms of an axis() check. The basic join axis is $A_0 = \{\text{descendant}\}$ for a `BOTTOM UP` window and $A_0 = \{\text{ancestor}\}$ for a `TOP DOWN` window. Details of the window frame exclusion clause (`EXCLUDE`, § 3.3.3) can be handled on logical algebra level by adjusting the join condition:

`NO OTHERS`	$\theta(u,t) := (H.\text{axis}(\nu_1, \nu_2) \in A_0 \cup \{\text{self}\})$
`CURRENT ROW`	$\theta(u,t) := (H.\text{axis}(\nu_1, \nu_2) \in A_0 \cup \{\text{self}\}) \wedge (u.\text{rowid} \neq t.\text{rowid})$
`GROUP`	$\theta(u,t) := (H.\text{axis}(\nu_1, \nu_2) \in A_0)$
`TIES`	$\theta(u,t) := (H.\text{axis}(\nu_1, \nu_2) \in A_0) \vee (u.\text{rowid} = t.\text{rowid})$

Example III. Starting from a binary self-grouping plan as shown in Example II, more advanced rewrites are possible. Consider again Statement I-b from § 3.3.5:

```
SELECT * FROM (                                                        I-b
   SELECT h.*,
          SUM(Amount) FILTER (WHERE Type = 'type A') OVER w
     FROM Hierarchy1 h LEFT OUTER JOIN Sale s ON Node = s.Product
   WINDOW w AS (HIERARCHIZE BY Node)
) WHERE DEPTH(Node) <= 3
```

The basic shape of the plan is $\sigma_\phi(e \bowtie\!\Gamma^{\theta}_{x:f_\psi} e)$, with $e := (\text{Hierarchy1} \bowtie \text{Sale})$. There is a condition $\phi(t) := (H.\text{depth}(t.\nu) \leq 3)$ on the output that does not depend on the computed sum x. Select operators of this kind can typically be pushed down to the left input of $\bowtie\!\Gamma$. The `FILTER` $\psi(t) := (t.\text{Type} = \ldots)$ can be handled by f or pushed down to the right input. The shape of the rewritten plan would be

$$\sigma_\phi(e \bowtie\!\Gamma^{\theta}_{x:f_\psi} e) \quad \rightsquigarrow \quad \sigma_\phi(e) \bowtie\!\Gamma^{\theta}_{x:f} \sigma_\psi(e).$$

Such rewriting always pays off, especially when the filters can be pushed down further.

5.1.6 Unary Structural Grouping

Translating a hierarchical window that features a *recursive* expression requires a new operator, *unary structural grouping* $\hat{\Gamma}(e)$.

Definition. Let expression e produce a relation $\{\tau\}_b$ for some tuple type τ; let $<$ be a comparator for τ elements providing a strict partial ordering of e's tuples, x a new attribute name, and f a *structural aggregation function* $\tau \times \{\tau \circ [x : \mathcal{N}]\}_b \to \mathcal{N}$, for some

scalar type $\mathcal{N}$. The *unary structural grouping* operator $\hat{\Gamma}$ associated with $<$, x, and f has the signature

$$\hat{\Gamma}^{<}_{x:f} : \{\tau\}_b \to \{\tau \circ [x : \mathcal{N}]\}_b$$

and is defined as

$$\hat{\Gamma}^{<}_{x:f}(e) := \{t \circ [x : \mathrm{rec}^{<}_{x:f}(e,t)] \mid t \in e\}_b, \text{ where}$$
$$\mathrm{rec}^{<}_{x:f}(e,t) := f(t, \{u \circ [x : \mathrm{rec}^{<}_{x:f}(e,u)] \mid u \in e[\![<:t]\!]\}_b).$$

Remarks. This new operator is necessary because the quasi-recursive computation cannot be expressed in terms of conventional relational algebra operators. Since the concept as such may be useful beyond hierarchical windows, we define it based on an abstract $<$ comparison predicate on the tuples of the input relation, which drives the data flow of the operation (i. e., the way the recursive grouping happens). It is required to be a strict partial order relation: irreflexive, transitive, and asymmetric. The operator arranges its input in an acyclic directed graph whose edges are given by the notion of *covered* tuples $<:$ we defined in § 3.3.3. On that structure it evaluates a structural aggregation function f, which performs an aggregation-like computation given a current tuple t and the corresponding bag of covered tuples. In other words, a variable, pseudo-recursive expression f is evaluated on a recursion tree predetermined by $<$.

We reuse the symbol Γ of standard unary grouping for $\hat{\Gamma}$. Both are similar in that they form groups of the input tuples, but $\hat{\Gamma}$ does not "fold away" the tuples. Instead, it behaves like SQL-style window grouping: it extends each tuple t in e by a new attribute x and assigns it the result of "rec," which applies f to t and the bag of its covered tuples u. The twist is that each tuple u in the bag already carries the x value, which has in turn been computed by applying rec to u, in a recursive fashion. Thus, while f itself is not recursive, a structurally recursive computation is encapsulated in $\hat{\Gamma}$'s definition. The recursion is guaranteed to terminate, since $<$ is *strict*.

Examples. For hierarchical windows, we define $<$ as in § 5.1.5 in terms of axis(). As we discussed in § 3.3.4, the default frame clause of a window function within a recursive expression is

```
RANGE BETWEEN 1 PRECEDING AND CURRENT ROW EXCLUDE GROUP
```

Using the rules of § 5.1.6 we thus obtain

$$(u < t) := (H.\mathrm{axis}(u.\nu_1, t.\nu_2) \in A)$$

where $A = \{\text{descendant}\}$ for a `BOTTOM UP` window and $A = \{\text{ancestor}\}$ for a `TOP DOWN` window. This definition makes $<$ indeed irreflexive, transitive, and asymmetric.

We can now translate our two example statements from § 3.1 into plans based on $\hat{\Gamma}$:

(1b)	total Value	$\uparrow$	$t.\text{Value} + \sum_{u \in X} u.x$
(2b)	absolute Weight	$\downarrow$	$t.\text{Weight} * \Pi_{u \in X} u.x$
(3b)	Value sum over "<:"	$\uparrow$	$\sum_{u \in X} u.\text{Value}$
(4a)	weighted rollup	$\uparrow$	$t.\text{Weight} * (t.\text{Value} + \sum_{u \in X} u.x)$
(4b)			$t.\text{Value} + t.\text{Weight} * (\sum_{u \in X} u.x)$
(4c)			$t.\text{Value} + \sum_{u \in X} u.\text{Weight} * u.x$
(5)	Dewey conversion	$\downarrow$	$\langle t.\text{ID} \rangle$ if $X = \{\}_b$, $u.x \circ \langle t.\text{ID} \rangle$ if $X = \{u\}_b$
(6b)	depth	$\downarrow$	$1 + \sum_{u \in X} u.x$
(7b)	subtree size	$\uparrow$	$1 + \sum_{u \in X} u.x$
(8)	subtree height	$\uparrow$	1 if $X = \{\}_b$, else $1 + \max_{u \in X} u.x$
(9b)	degree	$\uparrow$	$\lvert X \rvert$

Symbols: $\uparrow$ bottom up $\downarrow$ top down

Figure 5.1: Example definitions of $\hat{\Gamma}$'s structural aggregation function $f(t, X)$.

II-c
```
RECURSIVE (Value + SUM(x) OVER (HIERARCHIZE BY Node)) AS x
```
$\rightsquigarrow \quad \hat{\Gamma}^{<}_{x\,:\,f}(\text{Input1}), \quad f(t, X) = t.\text{Value} + \sum_{u \in X} u.x$

III
```
RECURSIVE (Value + SUM(Weight * x) OVER (HIERARCHIZE BY Node)) AS x
```
$\rightsquigarrow \quad \hat{\Gamma}^{<}_{x\,:\,f}(\text{Input2}), \quad f(t, X) = t.\text{Value} + \sum_{u \in X} u.\text{Weight} * u.x$

Figure 5.1 shows further example definitions of f. They correspond to the SQL expressions of Figure 3.7 (p. 48). As the examples attest, RECURSIVE expressions can be translated almost literally into corresponding $f(t, X)$ formulas.

5.1.7 Unary Versus Binary Grouping

Theoretically, there are few restrictions on the function f of $\hat{\Gamma}$ and $⋈\!\!\Gamma$, which performs the actual per-tuple computation. The practical limit is what SQL's expression language allows us to write. It is, however, useful to distinguish a class of common "simple" functions that let us establish a correspondence between $\hat{\Gamma}(e)$ and binary self-grouping $e ⋈\!\!\Gamma\, e$.

An aggregation function $\{\tau\}_b \to \mathcal{N}$ for use with $⋈\!\!\Gamma$ is *simple* if it is of the form

$$\text{acc}_{\oplus;\, g;\, \phi}(X) := \bigoplus_{u\,:\,u \in X \wedge \phi(u)} g(u),$$

where function $g : \tau \to \mathcal{N}$ extracts or computes a value from each tuple, ϕ is a predicate to filter the input, and $\oplus$ is a commutative, associative operator to combine the $\mathcal{N}$ values. This largely corresponds to what SQL allows us to express in the form

```
AGG(expr) FILTER(WHERE φ),
```

where *AGG* is a basic aggregate function such as SUM, MIN, MAX, EVERY, or ANY without DISTINCT set quantifier.

We can define a structural counterpart as follows: A structural aggregation function $\tau \times \{\tau \circ [x : \mathcal{N}]\}_b \to \mathcal{N}$ for use with $\hat{\Gamma}$ is *simple* if it is of the form

$$\text{str-acc}_{x\,:\,\oplus;\,g;\,\phi}(t, X) := \begin{cases} g(t) \oplus \bigoplus_{u \in X} u.x & \text{if } \phi(t), \\ \bigoplus_{u \in X} u.x & \text{otherwise.} \end{cases}$$

In Figure 5.1, functions 1b, 2b, 6b, and 7b are in fact simple.

To obtain our correspondence, consider $R := \hat{\Gamma}^{<}_{x\,:\,\text{str-acc}}(e)$. If the acyclic directed graph imposed by $<$ on e is a tree—that is, there are no undirected cycles—the following holds for all $t \in R$:

$$t.x = g(t) \oplus \bigoplus_{u \in R[\![<:t]\!]} u.x = g(t) \oplus \bigoplus_{u \in e[\![<t]\!]} g(u) = \bigoplus_{u \in e[\![\leq t]\!]} g(u)$$

where we define $u \leq t \;:\Longleftrightarrow\; u < t \vee u = t$, and assume $\phi = \text{true}$ for simplicity. The simple form of the aggregation function allows us to "hide" the recursion through the $<$ predicate and obtain a closed form of the expression for $t.x$ based on the original input e. We can thus state the following correspondence:

$$e \ltimes\!\!\!\Gamma^{\leq}_{x\,:\,\text{acc}_{\oplus;\,g;\,\phi}}\; e = \hat{\Gamma}^{<}_{x\,:\,\text{str-acc}_{x\,:\,\oplus;\,g;\,\phi}}(e).$$

Note that this equivalence will *not* hold if there are multiple paths $u <: \ldots <: t$ connecting two tuples $u < t$ in the input e. In this situation, $\hat{\Gamma}$ would indirectly count u multiple times into t's result, while $\ltimes\!\!\!\Gamma$ would not. This is due to the particular semantics of structural recursion, which simply propagates x values along the $<:$ paths. When we apply $\hat{\Gamma}$ in our hierarchical window setting, the equivalence holds, as $<:$ is derived from the acyclic tree structure of H, provided that we additionally make sure there are no duplicate ν values in the current window partition.

The correspondence is then useful in both directions and enables significant optimizations: As many typical non-recursive hierarchical window computations (and sometimes even join–group-aggregate queries) fit the form of acc, we can rewrite their initial translation $e \ltimes\!\!\!\Gamma\; e$ into $\hat{\Gamma}(e)$. As we assess in § 6, even when e is just a table scan, our $\hat{\Gamma}$ algorithms outperform $\ltimes\!\!\!\Gamma$ due to their simpler logic (e need not be evaluated twice) and effective pipelining.

Vice versa, if we can algebraically transform a given RECURSIVE expression into the form of str-acc, then $\ltimes\!\!\!\Gamma$ becomes an alternative to $\hat{\Gamma}$. If a WHERE condition ϕ on the output or a FILTER condition ψ is applied to the aggregate function, $\sigma_\phi(e) \ltimes\!\!\!\Gamma\; \sigma_\psi(e)$ will usually be superior to $\sigma_\phi(\hat{\Gamma}_{f_\psi}(e))$, as already noted in Example III of § 5.1.5.

Finally, consider again our manual rewrite of Statement I-a to I-b from Figure 3.6 (p. 47). Here we saved one join by using a hierarchical window. This demonstrates an advanced optimization from $e_1 \ltimes\!\!\!\Gamma\; e_2$ into Γ: By "merging" the two inputs into a combined input e_{12}, we can potentially (without going into details) rewrite $e_1 \ltimes\!\!\!\Gamma\; e_2$

Operator [Parameters] (Inputs) Short Name; Reference	Order (in; out) Output Size	Time Complexity Space Requirements
Hierarchy Index Scan $[H; \nu; \varsigma; \mathcal{L}; \phi]$ () HIS; § 5.2.1	—; ς $\lvert\sigma_\phi(T)\rvert \leq \lvert H\rvert$	$\mathcal{O}(\lvert H\rvert)$ $\mathcal{O}(1)$
Nested Loop Join $[\theta; \iota]$ (e_1, e_2) NLJ; § 5.2.3	arbitrary; ‡ $\lvert e_1 \bowtie e_2\rvert$	$\mathcal{O}(\lvert e_1\rvert \cdot \lvert e_2\rvert)$ $\mathcal{O}(1)$, mat. e_2
Hierarchy Index Join $[H; \nu_2; \theta; \iota; \varsigma]$ (e_1) HIJ; § 5.2.4	arbitrary; ‡ $\lvert e_1 \bowtie T\rvert$	$\mathcal{O}(\lvert e_1\rvert \cdot \lvert H\rvert)$ $\mathcal{O}(1)$
Hierarchy Merge Join $[H; \nu_1; \nu_2; \varsigma; A; \iota]$ (e_1, e_2) HMJ; § 5.2.5	†; ‡ $\lvert e_1 \bowtie e_2\rvert$	$\mathcal{O}(\lvert e_1\rvert + \lvert e_2\rvert + \lvert e_1 \bowtie e_2\rvert)$ —"—, mat. e_2
Hierarchy Staircase Filter $[H; \nu; \varsigma; a]$ (e) HSF; § 5.2.6	†; ‡ $\leq \lvert e\rvert$	$\mathcal{O}(\lvert e\rvert)$ $\mathcal{O}(1)$
Hierarchy Merge Groupjoin $[H; \nu_1; \nu_2; \varsigma; A; x : f]$ (e_1, e_2) HMGJ; § 5.2.8	†; ‡ $\lvert e_1\rvert$	$\mathcal{O}(\lvert e_1\rvert + \lvert e_2\rvert)$ $\mathcal{O}(\lvert e_2\rvert)$, mat. e_2
Hierarchical Grouping $[H; \nu; \varsigma; A; x : f]$ (e) HG; § 5.2.9	†; ‡ $\lvert e\rvert$	$\mathcal{O}(\lvert e\rvert)$ $\mathcal{O}(\lvert e\rvert)$
Sort $[<]$ (e) Sort; § 5.2.11	arbitrary; $<$ $\lvert e\rvert$	$\mathcal{O}(\lvert e\rvert \cdot \log \lvert e\rvert)$ $\mathcal{O}(\lvert e\rvert)$, mat. e
Hierarchy Rearrange $[H; \nu; \varsigma]$ (e) HR; § 5.2.11	†; ς $\lvert e\rvert$	$\mathcal{O}(\lvert e\rvert)$ $\mathcal{O}(\lvert e\rvert)$

H hierarchy index; ν, ν_i `Node` attribute names; $\varsigma \in \{\text{pre}, \text{post}\}$ sort order; $\mathcal{L}$ list of name/expression pairs; ϕ filter predicate; θ join predicate; ι join type; a axis; A set of axes;
† requires suitably ordered inputs; ‡ order of input(s) is retained in the output; *mat.* materialization required

Figure 5.2: An overview of the hierarchy operators in physical algebra.

to $e_{12} \bowtie^{\Gamma} e_{12}$ and then $\hat{\Gamma}(e_{12})$. This would pay off if e_{12} can be further simplified, for example, when e_1 and e_2 were very similar in the first place. Establishing relevant equivalences to enable such advanced optimizations is part of future work (see § 5.3).

5.2 Physical Algebra Operators

On the level of physical algebra, several new algorithms are needed in order to efficiently evaluate the logical operators we discussed in § 5.1. Figure 5.2 shows an overview of the operators we cover in the following: Hierarchy Index Scan (HIS) accesses the rows of the hierarchical table through a hierarchy traversal in either preorder or postorder. Nested Loop Join (NLJ), Hierarchy Index Join (HIJ), and Hierarchy Merge Join (HMJ) evaluate different types of hierarchy joins. Hierarchy Merge Groupjoin (HMGJ) and Hierarchical Grouping (HG) evaluate unary and binary structural grouping queries (that is, join–group–aggregate statements and hierarchical windows); in particular, the latter is needed whenever a `RECURSIVE` expression is involved. Hierarchy Rearrange (HR) reorders inputs that are already sorted in either preorder or postorder;

where applicable, it can outperform a full Sort.

For each operator, we specify a notation and syntactic rules for when it is applicable, define its exact functionality, discuss its practical uses and restrictions, and provide the pseudo code and a detailed analysis of the underlying algorithms. Thanks to our common index interface, the operators work for *every* type of hierarchy indexing scheme that is implemented by the system. In the discussion of runtime complexities, we assume the used index primitives to be in $\mathcal{O}(1)$ for static indexes and in $\mathcal{O}(\log |H|)$ for dynamic indexes, $|H|$ being the hierarchy size (cf. Figure 4.3, p. 74). Either way, the costs of the primitives are not affected by the input sizes and can therefore be considered constant-time for that matter.

We assume a push-based bottom-up model of query execution (see [76], § 5.3). For binary operators, we further stick to the convention that the *left* input e_1 is the one that is conceptually pushed towards the operator, thus forming a pipeline. The *right* input e_2 must be evaluated and materialized prior to the execution of the left pipeline, and can then be accessed through iterators (potentially in a random-access manner). In other words, the pipeline of the right input is always broken. In our pseudo code this is recognizable by a simple outer "for" loop that iterates over e_1 in a strictly forward manner, whereas e_2 is accessed like an array ("$e_2[i]$"). We use the keyword "yield" to indicate where a result tuple is pushed to the respective parent operator in the plan.

5.2.1 Hierarchy Index Scan

A hierarchy index provides an alternative access path to the hierarchical table T, much like other common database indexes such as B+-trees or hash tables. Instead of scanning the table, we can scan an attached hierarchy index H and obtain its nodes and the corresponding row IDs of T in the scan order ς. Additionally, a set of expressions $\mathcal{L}$ based on the hierarchy nodes can be evaluated on the fly. Finally, the nodes can be filtered by a predicate ϕ.

Syntax. Hierarchy Index Scan $[H; \nu; \varsigma; \mathcal{L}; \phi]\,()$, where H is a hierarchy index, ν is the name to be attached to the generated `Node` attribute, and $\varsigma \in \{\text{pre}, \text{post}\}$ is the scan order, either preorder or postorder. $\mathcal{L} = \{f_1 : e_1, \ldots, f_n : e_m\}$ is a list of names and expressions of types $\tau_1, \ldots, \tau_m$. The expressions may involve (only) the node attribute, that is, $\mathcal{F}(e_i) \subseteq \{\nu\}$. They are evaluated for each scanned node and their result is included in the output. ϕ is a boolean expression involving the ν attribute or any of the computed attributes, that is, $\mathcal{F}(\phi) \subseteq \{\nu, f_1, \ldots, f_n\}$.

The output is of type $[\text{rowid} : \texttt{ROWID}, \nu : \texttt{Node}^H, f_1 : \tau_1, \ldots, f_m : \tau_m]$ and sorted by ν in ς-order.

Remarks. Hierarchy Index Scan itself produces only a (sorted) stream e of row IDs and `Node` objects. To actually access some attributes $a_1, \ldots, a_n \in \mathcal{A}(T)$ of the hierarchical table T, a Map operator must be used to dereference the row IDs:

$$\text{Map}\,[t : T[\text{rowid}]; t.a_1, \ldots, t.a_n]\,(e)$$

It is possible to emulate Hierarchy Index Scan in terms of basic operators, namely by combining Scan, Map, Select and Sort. The following produces an equivalent result:

1 $e_1 \leftarrow \text{Scan}\,[T; \text{rowid}, \ell, a_1, \ldots, a_n]\,()$
2 $e_2 \leftarrow \text{Map}\,[\nu : H.\text{node}(\ell), f_1 : e_1, \ldots, f_m : e_n]\,(e_1)$
3 $e_3 \leftarrow \text{Select}\,[\phi]\,(e_2)$
4 $e_4 \leftarrow \text{Sort}\,[<]\,(e_3)$ where $(t_1 < t_2) := H.\text{is-before-}\varsigma(t_1.\nu, t_2.\nu)$

Compared to this emulation, HIS has a few home-field advantages: First, no explicit sorting is necessary; the sorting happens implicitly by traversing the index in the desired order. Second, calling node() to obtain `Node` objects from the labels ℓ is not necessary, as the index scan produces the `Node` objects directly. Third, filtering by ϕ happens *prior* to dereferencing the row IDs. This increases efficiency if the dereferencing has a non-negligible cost—as is commonly the case, especially with column-oriented table storage. Fourth, there are favorable cache locality effects: Since the predicate ϕ and the functions $\mathcal{L}$ are evaluated "just in time" during the scan, the parts of the index data structure that are needed for the evaluation are likely to be in cache. This enables HIS to significantly outperform the conventional Scan–Map–Select–Sort alternative.

Example. HIS is very useful whenever hierarchical filtering or ordering is performed directly on a hierarchical table:

```
SELECT Node, DEPTH(Node) FROM T WHERE DEPTH(Node) < 5 ORDER BY PRE_RANK(Node)
```

A conventional plan would be:

$$\text{Sort}\,[<_*]\,(\text{Select}\,[d < 5]\,(\text{Map}\,[\nu : H.\text{node}(\ell), d : H.\text{depth}(\nu)]\,(\text{Scan}\,[T]\,())))$$

where $t_1 <_* t_2 := H.\text{is-before-pre}(t_1.\nu, t_2.\nu)$. When the optimizer considers HIS as an alternative way to access T, it can detect that the Map, Select, and Sort instances can in fact be pushed down to the scan and eliminated. The result is an "index-only" plan:

$$\text{Hierarchy Index Scan}\,[H; \nu; \text{pre};\ d : H.\text{depth}(\nu);\ d < 5]\,()$$

Algorithm. The algorithm is a straightforward loop over the hierarchy, using the traversal primitives offered by our index interface. Its time complexity is in $\Theta(|H|)$. The space requirements are in $\mathcal{O}(1)$.

1 **operator** Hierarchy Index Scan $[H; \nu; \varsigma; \mathcal{L}; \phi]\,()$
2 $(c_1, c_2, \phi') \leftarrow (H.\varsigma\text{-begin}(), H.\varsigma\text{-end}(), \phi)$
3 $c \leftarrow c_1$ // c, c_1, c_2 are of type $\texttt{Cursor}^H$
4 **while** $c \neq c_2$
5     $x_i \leftarrow e_i(c)$ for $(f_i : e_i) \in \mathcal{L}$
6     $t \leftarrow [\text{rowid} : H.\text{rowid}(c), \nu : c, f_1 : x_1, \ldots, f_n : x_n]$
7   **if** $\phi'(t)$
8   **yield** t
9     $c \leftarrow H.\varsigma\text{-next}(c)$

Extensions. Evaluating the given $\mathcal{L}$ and ϕ expressions on the fly already gives HIS certain "just in time" advantages over a conventional a-posteriori Select$_\phi$. However, since $\mathcal{L}$ and ϕ are known at query compile time, we could improve the performance even further by generating tailored variants of the scan algorithm. A big potential gain is when a predicate ϕ can be handled *implicitly* by restricting the scan range. In this case the evaluation of ϕ does not add to the runtime costs at all—it even reduces them. To incorporate this idea into the above algorithm, we need to replace line 2 by

$$(c_1, c_2, \phi') \leftarrow \text{scan-range}\,[H; \nu; \varsigma; \phi]\,().$$

This helper function determines a range $[c_1, c_2[$ that is suitable for the given predicate ϕ and as small as possible. The range is guaranteed to not preclude any nodes for which ϕ may hold. However, it may still include nodes for which ϕ does *not* hold. Therefore, a "residual" predicate ϕ' is returned that remains to be checked for each node in $[c_1, c_2[$.

For example, a predicate ϕ of the form "H.axis(ν, ν_0) = descendant," where ν_0 is a constant, can be fully handled by scanning from ν_0 to its first *following* node. Here is a simple version of scan-range() that can detect this case:

```
1  function scan-range [H; ν; ς; φ] ()
2  |  if (ς = pre) ∧ (φ = [H.axis(ν, ν0) = {descendant}] for some ν0)
3  |  |  c0 ← H.cursor(ν0)
4  |  |  return (H.pre-next(c0), H.next-following(c0), true)
5  |  else
   |  |  // other cases ...
6  |  else                                              // fallback: full scan
7  |  |  return (H.ς-begin(), H.ς-end(), φ)
```

In summary, for the scan order ς = pre the four axes can be handled as follows:

Axis	*Restricted range* $[c_1, c_2[$	*Residual predicate* ϕ'
descendant	$[H$.pre-next(c_0), H.next-following$(c_0)[$	true
ancestor	$[H$.pre-begin$()$, $c_0[$	$\neg H$.is-before-post(ν, ν_0)
preceding	$[H$.pre-begin$()$, $c_0[$	H.is-before-post(ν, ν_0)
following	$[H$.next-following(c_0), H.pre-end$()[$	true

Of course, these optimizations are only beneficial for indexes where is-before-ς() is faster than axis(), and next-following() is not implemented in terms of a next-pre() loop. This is for example the case with our order indexes, as shown in § 4.5.5. Extending the outlined ideas further is a topic for future work (see also § 6).

5.2.2 Hierarchy Join: Overview

In the following sections we discuss algorithms to evaluate hierarchy joins $e_1 \bowtie_\theta e_2$, which we characterized in § 5.1.4. The primary algorithms are Hierarchy Index Join (HIJ, § 5.2.4) and Hierarchy Merge Join (HMJ, § 5.2.5). Nested Loop Join (NLJ, § 5.2.3) is mainly relevant as a fallback method. Each algorithm can handle only a limited set of join cases and subsets of axis sets A. The relevant cases are:

- Equi Joins — The case where $\theta(t_1, t_2)$ is "H.axis$(t_1.\nu_1, t_2.\nu_2) \in \{\text{self}\}$" corresponds to a plain equi join. `Node` objects are both equality-comparable and hashable, so this case can and should be handled with a standard equi join algorithm.

- HIJ — Joins of the form "$e_1 \bowtie_\theta \sigma(T)$," where e_1 is arbitrarily ordered and the right input is the (potentially restricted) hierarchical table itself, can be answered by Hierarchy Index Join. Instead of independently evaluating the right join side, it directly enumerates the joined tuples for each left input tuple in e_1. Thus the order of e_1 is retained. However, HIJ only supports the join types $\bowtie$, $⟕$, $\ltimes$, and $\triangleright$.

- HIS + HIJ — Self joins of the form $\sigma(T) \bowtie_\theta \sigma(T)$ on the (potentially restricted) hierarchical table can be answered using an "index only" approach that combines HIS (to enumerate the left side) and HIJ. The result will be ordered by ν_1–ν_2 in either preorder or postorder.

- HMJ — If both e_1 are e_2 are intermediate result tables (rather than the hierarchical table T itself), Hierarchy Merge Join can be used. It requires both inputs to be sorted by the `Node` fields in either preorder or postorder. The order is retained, that is, the resulting rows will be in preorder or postorder as well. HMJ supports any join type. As HMJ does not restrict the input expression e_2, it can be used to construct *bushy* query plans (like NLJ, but unlike HIJ).

- NLJ — Only the basic Nested Loop Join can work with arbitrary input tables without requiring them to be ordered as HMJ. Furthermore, it can handle *all* possible join predicates and join types.

- HSF — For semi and anti joins on ordered inputs, the Hierarchy Staircase Filter (§ 5.2.6) can be applied in order to reduce the size of the non-retained join side and thus improve the performance.

The naïve and inefficient NLJ should be considered only as a fallback method, and avoided by the optimizer in favor of the more efficient and robust HMJ and HIJ algorithms where possible. Introducing sort operators for both inputs to enable HMJ may well pay off. Compared to NLJ, HMJ is a major improvement in that it requires only a single pass through the two inputs and therefore has a linear worst-case time complexity. Even with HIJ, ordering the left input to enable HMJ can be worthwhile, especially if there are overlaps in the sets of join partners. That said, HMJ also requires more space, since intermediate results need to be buffered.

Aside from the general usage patterns listed above, further restrictions are imposed by the individual algorithms. For example, HMJ comes in two variants—bottom up and top down—with significantly different algorithms; the former handles axis $A = \{\text{descendant}\}$, the latter axis $\{\text{ancestor}\}$. During algorithm selection, the optimizer may need to apply the transformations discussed in § 5.1.2—in particular, inverting the axis set A or swapping the arguments using reflect()—in order to determine the applicability of a join algorithm in the light of such restrictions.

5.2.3 Nested Loop Join

Syntax. Nested Loop Join $[\theta; \iota]\ (e_1, e_2)$, where θ is a predicate with $\mathcal{F}(\theta) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, ι is the join type, $\iota \in \{⋈, ⟕, ⟖, ⟗, ⋉, ⋊, ▷, ◁\}$. If the inputs are of type $e_1 : \{\tau_1\}_b$ and $e_2 : \{\tau_2\}_b$ for some tuple types τ_1 and τ_2, then the output is of type $\{\tau_1 \circ \tau_2\}_b$ for $\iota \in \{⋈, ⟕, ⟖, ⟗\}$, of type $\{\tau_1\}_b$ for $\iota \in \{⋉, ▷\}$, and of type $\{\tau_2\}_b$ for $\iota \in \{⋊, ◁\}$.

Remarks. The standard algorithm naïvely checks the predicate θ for each pair in the cross product $e_1 \times e_2$ using nested loops. The only difference in our setting is that θ is a hierarchy predicate, as characterized in § 5.1.4. Although NLJ is often prohibitively inefficient, it is useful as a fallback: Its implementation is trivial, it is fully flexible regarding θ and ι, and it does not impose any requirements on the inputs regarding sortedness, the uniqueness of the `Node` values, or the shape of the subplans e_1 and e_2. Also, any available order in the inputs is retained (in e_1–e_2 order).

The basic algorithm can be adapted and slightly optimized for left/right outer/semi/anti join semantics. Furthermore, if the right input happens to be sorted by `Node`, the "$\leq$" nature of hierarchy predicates allows the algorithm to skip parts of e_2 that cannot match a given e_1 tuple. But regardless of such optimizations, the worst-case time complexity remains in $\mathcal{O}(|e_1| \cdot |e_2|)$. As the cross product is not actually materialized, no space is required besides for materializing the right input.

5.2.4 Hierarchy Index Join

Syntax. Hierarchy Index Join $[H; \nu_2; \theta; \iota; \varsigma]\ (e_1)$, where H is a hierarchy index, ν_2 is the name of the generated (right-hand) `Node` attribute, θ is a hierarchy join predicate, ι is the join type, $\iota \in \{⋈, ⟕, ⋉, ▷\}$, $\varsigma \in \{\text{pre}, \text{post}\}$ is the desired sort order of the join partners, and the input is of type $e_1 : \{\tau_1\}_b$ for some tuple type τ_1.

Let $\tau_2 := [\text{rowid} : \texttt{ROWID}, \nu_2 : \texttt{Node}^H]$. The output is of type $\{\tau_1 \circ \tau_2\}_b$ for $\iota \in \{⋈, ⟕\}$, and of type $\{\tau_1\}_b$ for $\iota \in \{⋉, ▷\}$.

Algorithm. The algorithm simply runs a Hierarchy Index Scan for each $t_1 \in e_1$ by partially binding the predicate θ to t_1 and giving it as a parameter to the scan. Then all tuples produced by the scan are combined with t_1 and yielded. This simple "for" loop can be expressed in terms of a dependent join ⧑:

$$\begin{aligned}&\text{Hierarchy Index Join}\,[H; \nu_2; \theta; ⋈; \varsigma]\ (e_1)\\ \equiv\ & e_1\ ⧑\ \text{Hierarchy Index Scan}\,[H; \nu_2; \varsigma; \{\}; \theta]\ ().\end{aligned}$$

The other join types ι translate into different variants of d-join (⧑, ⧔, and ▶).

As $|e_1|$ index scans are run in total, the time complexity is in $\mathcal{O}(|e_1| \cdot |H|)$. The space complexity is in $\mathcal{O}(1)$. For certain predicates θ, such as a join on the *descendant* axis, the range of each scan can be restricted to exactly the join partners via the mechanism discussed in § 5.2.1; the time complexity is then even in $\mathcal{O}(|e_1| + |e_1 ⋈ e_2|)$.

Remarks. HIJ is fairly efficient, as it pipelines e_1 and directly enumerates the T tuples that potentially match. In the *descendant* case, the joining happens completely *implicitly*: unlike with NLJ or HMJ, tuple pairs that do not actually match are never even considered. Beyond that, HIJ has some useful properties: First, it supports multiple join types—albeit only the "left" joins $\bowtie$, ⟕, $\ltimes$, or $\triangleright$ due to the dependent right side. Second, it does not require e_1 to be sorted and retains the available order. Third, it can enumerate the join partners in either preorder or postorder. Fourth, θ can include further restrictions besides the join condition, which are handled by HIS on the fly.

Example. The following query combines each root with the IDs of its descendants:

```
SELECT u.Node, v.ID FROM T u, T v
 WHERE IS_ROOT(u.Node) AND IS_DESCENDANT(v.Node, u.Node)
```

A plan based on Hierarchy Index Join would be:

1 $e_0 \leftarrow$ Hierarchy Index Scan $[H; \nu_1; \text{pre}; \{\phi : H.\text{is-root}(\nu_1)\}; \phi]$ ()
2 $\theta(t_1, t_2) := (H.\text{axis}(t_2.\nu_2, t_1.\nu_1) = \text{descendant})$
3 $e_1 \leftarrow$ Hierarchy Index Join $[H; \nu_2; \theta; \bowtie; \text{pre}]$ (e_0)
4 $e_2 \leftarrow$ Map $[t_2 : T[\text{rowid}]; t_2.\text{ID}]$ (e_1)

This is a reasonable and common use case. Note that each root node has a *distinct* set of (potential) join partners. Its simple logic makes HIJ preferable to HMJ in this case.

When e_1 contains duplicate `Node` values, the basic version of HIJ does redundant work. This also happens in cases where the sets of join partners overlap, such as when one left input node is a parent of another and thus shares all its descendants. For example, if we change the condition "`IS_ROOT(u.Node)`" to "`DEPTH(u.Node) < 3`," certain ν_2 values will be enumerated multiple times and thus repeatedly processed by the subsequent Map operator. In these cases, a HIJ-based plan may be inferior to a plan that first sorts e_1 and then employs Hierarchy Merge Join.

5.2.5 Hierarchy Merge Join

Syntax. Hierarchy Merge Join $[H; \nu_1; \nu_2; \varsigma; A; \iota]$ (e_1, e_2), where H is a hierarchy index, ν_1 and ν_2 are `Node` attribute names, $\varsigma \in \{\text{pre}, \text{post}\}$ is the order of the two inputs (either both preorder or both postorder), A is the set of hierarchy axes to join on, and ι is the join type, $\iota \in \{\bowtie, ⟕, ⟖, ⟗, \ltimes, \rtimes, \triangleright, \triangleleft\}$.

Both inputs e_1 and e_2 must have a `Node` field named ν_1 and ν_2, respectively; formally, $e_i : \{\tau_i\}_b$ for some tuple type τ_i with a field ν_i : `Node`^{H}. The inputs are required to be sorted by ν_i in ς-order.

For $\iota \in \{\ltimes, \triangleright\}$ the output is of type $\{\tau_1\}_b$ and ordered by ν_1 in ς-order. For $\iota \in \{\rtimes, \triangleleft\}$ the output is of type $\{\tau_2\}_b$ and ordered by ν_2 in ς-order. For $\iota \in \{\bowtie, ⟕, ⟖, ⟗\}$ the output is of type $\{\tau_1 \circ \tau_2\}_b$. It will be ordered either by ν_1–ν_2 or by ν_2–ν_1 in ς-order, as detailed below. ("Ordered by ν_1–ν_2" means that the joined tuples are ordered by first $e_1.\nu_1$ in ς-order, then $e_2.\nu_2$ in ς-order.)

Functionality. Hierarchy Merge Join is a family of four distinct algorithms. All four algorithms have in common that they require *both* inputs to be sorted in ς-order, and they retain one of the two input orders as primary order in the output. Furthermore, they generally pipeline their *left* input (and process it in a strictly forward manner) but materialize their *right* input (and access it through an iterator).

The two dimensions they differ in are, first, whether they process preorder or postorder inputs (ς = pre versus ς = post), and second, whether their output is ordered by ν_1–ν_2 (algorithm "A") or by ν_2–ν_1 (algorithm "B"). Each of the four algorithms supports (only) a particular join "direction": upwards or downwards. *Upwards* means that the supported join predicate θ is "H.axis$(\nu_2, \nu_1) = \{\text{ancestor}, \text{self}\}$." *Downwards* means that θ has to be "H.axis$(\nu_2, \nu_1) = \{\text{self}, \text{descendant}\}$." Altogether, this gives us the following choices depending on the available order ς of the inputs and the intended join direction (i. e., the axis A on which we wish to join e_2 against e_1):

	ς	*Direction*	*Suitable Algorithm*	*Output Order*
1.	pre	upwards	Hierarchy Merge Join A/pre [...] (e_1, e_2)	ν_1–ν_2 pre
2.	post	downwards	Hierarchy Merge Join A/post [...] (e_1, e_2)	ν_1–ν_2 post
3.	pre	upwards	Hierarchy Merge Join B/pre [...] (e_1, e_2)	ν_2–ν_1 pre
4.	post	downwards	Hierarchy Merge Join B/post [...] (e_1, e_2)	ν_2–ν_1 post
5.	pre	downwards	Hierarchy Merge Join A/pre [...] (e_2, e_1)	ν_2–ν_1 pre
6.	post	upwards	Hierarchy Merge Join A/post [...] (e_2, e_1)	ν_2–ν_1 post
7.	pre	downwards	Hierarchy Merge Join B/pre [...] (e_2, e_1)	ν_1–ν_2 pre
8.	post	upwards	Hierarchy Merge Join B/post [...] (e_2, e_1)	ν_1–ν_2 post

The first algorithm, Hierarchy Merge Join A, comes in two variants, based on either preorder or postorder inputs. The former joins e_1 and e_2 upwards (case 1), the latter joins e_1 and e_2 downwards (case 2). The second algorithm, Hierarchy Merge Join B, also comes in preorder and postorder variants and basically supports the same ς/direction combinations as HMJ A (cases 3 and 4), but produces the respective "swapped" output orders. That is, the output of HMJ B is ordered by ν_2–ν_1 when the output of HMJ A would be ordered by ν_1–ν_2 (cases 1 versus 3 and 2 versus 4).

By simply swapping the e_1 and e_2 arguments of HMJ A, we can use HMJ A to effectively join e_1 and e_2 in the respective *inverse* directions: pre/downwards (case 5) and post/upwards (case 6). Of course the resulting output order will then be ν_2–ν_1 accordingly. Analogously, by swapping the e_1 and e_2 arguments of HMJ B, we can, for a given ς, use HMJ B to join in the respective inverse directions in comparison to HMJ A but still achieve ν_1–ν_2 order (cases 7 and 8). Viewed from a different angle, HMJ B effectively joins downwards when HMJ A would join upwards, and vice versa. For example, we can use HMJ A for the pre/upwards case (1) and HMJ B for the pre/downwards case (7) and get output order ν_1–ν_2 in both cases.

That said, swapping the arguments e_1 and e_2 also swaps the pipelined/materialized roles of the inputs as a side effect, as HMJ A and HMJ B always pipeline their first argument and materialize their second argument. This may be relevant to the performance of the overall plan and should therefore also be considered during algorithm selection by the query optimizer. It may, for instance, be preferable to materialize the

smaller of the two inputs even if that would mandate using a (less efficient) HMJ B variant instead of HMJ A.

Remarks. The logic and properties of HMJ A and HMJ B are reminiscent of the classic Merge Join operator (which handles equi join "=" predicates). Taking advantage of their ordered inputs allows them to disregard ranges of the inputs that cannot possibly match. HMJ A and HMJ B thus are able to achieve the ideal linear time complexity of $\mathcal{O}(|e_1| + |e_2| + |e_1 \bowtie e_2|)$.

The two HMJ A variants work with the "natural" input orders for the respective cases, that is, orders that lend themselves well to merge-join-style forward processing. For upwards joins the natural input order is preorder, whereas for downwards joins the natural order is postorder. The HMJ A algorithms are therefore very efficient. HMJ B, by contrast, rearranges its output in order to produce the "swapped" output order (ν_2–ν_1 instead of ν_1–ν_2). As the above table shows, these rearranging joins are essentially equivalent to the "unnatural" join variants pre/downwards and post/upwards. The rearranging requires the HMJ B algorithms to maintain buffers of stashed intermediate results. Therefore, they are generally less efficient than the HMJ A algorithms.

Concerning the semi or anti join cases, the "left" variants $\ltimes$ and $\triangleright$ are generally preferable to the "right" variants $\rtimes$ and $\triangleleft$, when there is a choice (regardless of HMJ A versus HMJ B). This can be understood by inspecting the pseudo code of the $\rtimes$ and $\triangleleft$ variants, which use additional memory to mark the matched input tuples.

In the situation that a combination of input/output sort orders is desired that does not match any of the cases 1–8 listed above (e. g., the input e_1 is in preorder but e_2 is in postorder), Hierarchy Rearrange (§ 5.2.11) brings further flexibility. HR can be applied to one or both inputs and/or the output to convert them from preorder to postorder, and vice versa. Even though HR comes at a non-negligible runtime cost, this multiplies the combinations of input/output orders that can be achieved, without requiring us to implement even more variants of the Hierarchy Merge Join.

Our Hierarchy Merge Join operators are inspired by the work of Al-Khalifa et al. on so-called structural join operators for XML data. In [53] the authors present two algorithms "stack-tree-desc" and "stack-tree-anc," which consume lists of XML nodes in document order (i. e., preorder) and perform an axis step (i. e., join) on the *descendant* axis. HMJ can be seen as a generalization of these concepts based on general hierarchical tables and a generic hierarchy index interface. We also contribute join algorithms based on *postorder* inputs, which to our knowledge has not received much attention in the XML field (since XML data is naturally in document order). In our classification of Hierarchy Merge Join algorithms, stack-tree-desc is comparable to pre/upwards HMJ A, whereas stack-tree-anc is comparable to pre/upwards HMJ B.

Example. Consider a variant of the example we used for HIJ (§ 5.2.4):

```
SELECT u.Node, v.ID FROM T u, T v
 WHERE IS_DESCENDANT(v.Node, u.Node)
   AND DEPTH(u.Node) <= 2 AND DEPTH(v.Node) >= 5
```

The following HMJ-based plan generally outperforms an equivalent HIJ-based plan:

```
1  e0 ← Hierarchy Index Scan [H; ν1; pre; {d : H.depth(ν1)}; d ≤ 2] ()
2  e1 ← Hierarchy Index Scan [H; ν2; pre; {d : H.depth(ν2)}; d ≥ 5] ()
3  e2 ← Map [t : T[rowid]; t.ID] (e1)
4  e3 ← Hierarchy Merge Join [H; ν2; ν1; pre; {ancestor}; ⟗] (e2, e0)
```

With HMJ A/pre the output would be in ν_2–ν_1 order; with HMJ B/pre it would be in ν_1–ν_2 order. Note there is some overlap between the different sets of join partners of the left input tuples. Thanks to HMJ, we can push down Map in order to obtain the ID values *before* performing the join, so this work is done only once per distinct v node. This ability to create bushy plans allows HMJ-based plans to outperform the left-deep HIJ-based plans in many situations.

Algorithm for HMJ A/pre. The following pseudo code shows the basic outline of HMJ A, assuming the input order ς = pre, the join axes $A = \{\text{ancestor}, \text{self}\}$, and the join type $\iota = ⟗$.

```
1  operator Hierarchy Merge Join A [H; ν1; ν2; pre; {ancestor, self}; ⟗] (e1, e2)
2  S2 : Stack ⟨τ2⟩                                  // stack of matched e2 tuples
3  p : int, p ← 0                                    // position in e2 (iterator)
4  for t1 ∈ e1
5  │ ⟨maintain S2⟩
6  │ if S2 = ⟨⟩
7  │ │ yield t1 ∘ ⊥τ2                                // "left outer" e1 tuple
8  │ for t2 ∈ S2                                     // "inner" join matches
9  │ │ yield t1 ∘ t2
10 for t2 ∈ e2[p..|e2|[                              // remaining "right outer" e2 tuples
11 │ yield ⊥τ1 ∘ t2
```

Both inputs are accessed strictly sequentially: The outer loop (l. 4) passes through e_1, whereas e_2 is accessed via an iterator p. The stack S_2 keeps tuples from e_2 that still may have join partners. For each incoming left input tuple t_1, the $\langle$maintain $S_2\rangle$ block—which we consider below—maintains the stack S_2 in such a way that it contains exactly the join partners with respect to $t_1.\nu_1$ (l. 5). In the process it also adds further tuples from e_2 onto S_2, if necessary. If S_2 is empty, a "left outer" tuple is yielded for t_1 (l. 6–7). Otherwise, a simple "for" loop combines t_1 with the tuples on S_2 to produce the "inner" join tuples (l. 8–9). Finally, after all e_1 tuples have been processed, there might be further unprocessed tuples remaining in e_2, which are yielded as "right outer" join tuples (l. 10–11).

The $\langle$maintain $S_2\rangle$ block has to identify e_2 tuples that are on the *ancestor* or *self* axes with respect to $t_1.\nu_1$:

```
12 ⟨maintain S2⟩:
13 while S2 ≠ ⟨⟩ ∧ H.is-before-post(S2.top().ν2, t1.ν1)     // preceding axis
14 │ S2.pop()
```

15 **while** $p \neq |e_2|$
16 $\quad t_2 \leftarrow e_2[p]$
17 $\quad$ **if** $t_2.\nu_2 = t_1.\nu_1$ // *self* axis
18 $\quad\quad S_2.\text{push}(t_2)$
19 $\quad$ **else if** $\neg H.\text{is-before-pre}(t_2.\nu_2, t_1.\nu_1)$ // *descendant* or *following* axis
20 $\quad\quad$ **break**
21 $\quad$ **else if** $H.\text{is-before-post}(t_2.\nu_2, t_1.\nu_1)$ // *preceding* axis
22 $\quad\quad$ **yield** $\bot_{\tau_1} \circ t_2$ // "right outer" e_2 tuple
23 $\quad$ **else**
24 $\quad\quad S_2.\text{push}(t_2)$ // *ancestor* axis
25 $\quad p$++

The first loop pops *preceding* tuples from S_2 that are no longer relevant. The second loop examines further tuples from e_2 up to the first *descendant* or *following* tuple. It pushes them onto S_2 if they are on the *ancestor* or *self* axes, and dismisses them—after yielding a corresponding "right outer" tuple—if they are on the *preceding* axis. The seemingly redundant check in l. 17 is an optimization for the *self* case. An equality check is generally very cheap compared to the is-before() primitives.

To modify the algorithm to exclude the *self* axis, that is, to join on $A = \{\text{ancestor}\}$ only, it is sufficient to replace l. 18 by "**break**."

As noted, the shown algorithm performs a full outer join. With minor modifications, it can support any other join type ι as well:

- ⟕ Based on the $\iota =$ ⟗ case, remove l. 10–11 and replace l. 22 by a no-op.
- ⟖ Based on the $\iota =$ ⟗ case, remove l. 6–7.
- ⋈ Apply the modifications for ⟕ and ⟖.
- ⋉ Based on the $\iota =$ ⋈ case, replace the loop in l. 8–9 by the following:

  if $S_2 \neq \langle\rangle$
  $\quad$ **yield** t_1

- ⋊ Based on the $\iota =$ ⋈ case, remove the loop in l. 8–9. Replace the two occurrences of "$S_2.\text{push}(t_2)$" by "**yield** t_2."
- ▷ Based on the $\iota =$ ⟕ case, remove l. 8–9. Replace "**yield** $t_1 \circ \bot_{\tau_2}$" by "**yield** t_1."
- ◁ Based on the $\iota =$ ⟖ case, remove l. 8–9. Replace "**yield** $\bot_{\tau_1} \circ t_2$" by "**yield** t_2."

In the semi- and anti-join cases, a Staircase Filter can be applied to the non-retained join side; see § 5.2.6.

Algorithm for HMJ A/post. This algorithm handles *postorder* inputs (ς = post) and joins *downwards* with $A = \{\text{self}, \text{descendant}\}$. Its basic outline is:

```
1  operator Hierarchy Merge Join A [H; ν1; ν2; post; {self, descendant}; ⟗] (e1, e2)
2  S1 : Stack ⟨[ν1 : Node^H, i1 : int, i2 : int]⟩
3  i1, i2 : int,  i1 ← 0,  i2 ← 0
4  for t1 ∈ e1
5  │  ⟨determine [i1, i2[⟩
6  │  if i1 = i2
7  │  │  yield t1 ∘ ⊥τ2                                   // "left outer" e1 tuple
8  │  for t2 ∈ e2[i1, i2[                                 // "inner" join matches
9  │  │  yield t1 ∘ t2
10 for i ∈ [0, |e2|[ not covered by any [i1, i2[ on S1    // "right outer" e2 tuples
11 │  yield ⊥τ1 ∘ e2[i]
```

The left input is again pipelined, but unlike the pre/upwards case of HMJ A, the right input is not accessed in a strictly forward manner. (In return, no extra buffer S_2 for e_2 tuples is needed.) The stack S_1 stores, for each seen node ν_1 in the left input e_1, a corresponding range $[i_1, i_2[$ of e_2 that indicates the join partners with respect to $t_1.\nu_1$. Thus, once the range has been determined for ν_1 and placed on the stack, this information can be reused for upcoming nodes ν_1' who are ancestors of ν_1. Whenever such an ancestor is placed on S_1, it "sucks up" any descendant entries already on S_1 and their join ranges. The $\langle$determine $[i_1, i_2[\rangle$ block (l. 5) does the critical work of determining the appropriate range of join partners, and maintains S_1 in the process. Once this is done, an empty range indicates an unmatched "left outer" tuple (l. 6–7); otherwise, the join pairs are straightforwardly enumerated (l. 8–9). In the end, unmatched "right outer" tuples can be identified by the fact that they are not covered by any $[i_1, i_2[$ range on S_1, and enumerated in a simultaneous loop over S_1 and e_2 (l. 10–11).

The $\langle$determine $[i_1, i_2[\rangle$ block has to identify e_2 tuples that are on the *self* or *descendant* axes with respect to ν_1:

```
12 ⟨determine [i1, i2[⟩:
13 if S1 ≠ ⟨⟩ ∧ S1.top().ν1 = t1.ν1                        // same as previous
14 │  goto l. 6
15 while i2 ≠ |e2| ∧ (e2[i2].ν2 = t1.ν1 ∨ H.is-before-post(e2[i2].ν2, t1.ν1))
16 │  i2++
17 i1 ← i2
18 while S1 ≠ ⟨⟩ ∧ ¬H.is-before-pre(S1.top().ν1, t1.ν1)      // descendants of ν1 on S1
19 │  i1 ← S1.pop().i1
20 while i1 > 0 ∧ ¬H.is-before-pre(e2[i1 − 1].ν2, t1.ν1)     // further descendants in e2
21 │  i1−−
22 S1.push([ν1, i1, i2])
```

If the previous node on S_1 matches the current node, the $[i_1, i_2[$ range can immediately be reused (l. 13–14). Otherwise, the first "while" loop (l. 15–16) increases the upper bound i_2 up to the first node in the right input that is on the *following* or *ancestor* axis of ν_1. (Note the i_2 value is initially taken over from the previous iteration.) The

lower bound i_1 is initially set to i_2 ("no join partners"); however, any of the previous nodes on S_1 (when viewed from the top) might be descendants of ν_1. In this case, these entries are removed and their join ranges are subsumed (l. 18–19). This is correct because any range of join partners of a descendant on S_1, as well as any e_2 nodes *in between* two such ranges, are also valid join partners for ν_1. After that, the interval $[i_1, i_2[$ may need to be further extended on the lower end, as there might be tuples on the *self* or *descendant* axes in the range $e_2[S_1.\text{top}().i_2, i_1[$. This is done by the third "while" loop (l. 20–21). Note that at this point $i_1 = i_2$ is possible if all considered nodes in e_2 and on S_1 happened to be on the *preceding* axis. Finally, the determined information is memorized on S_1 for later reuse (l. 22).

To modify the algorithm so that the *self* axis is not included in the join, it is sufficient to adjust the condition in l. 15 accordingly.

Again, the algorithm can be modified to handle any other join type ι, just like we did for the pre/upwards variant of HMJ A.

Algorithm for HMJ B/pre. The following pseudo code shows the basic outline of HMJ B, assuming the input order $\varsigma = \text{pre}$, the join axes $A = \{\text{ancestor}, \text{self}\}$, and the join type $\iota = \bowtie$.

```
1  operator Hierarchy Merge Join B [H; ν1; ν2; pre; {ancestor, self}; ⋈] (e1, e2)
2  S2 : Stack ⟨[t2 : τ2, joined : List ⟨τ1 ∘ τ2⟩, inherited : List ⟨τ1 ∘ τ2⟩]⟩
3  unmatched-t1 : List ⟨τ1⟩
4  p : int,  p ← 0
5  for t1 ∈ e1
6  |  ⟨maintain S2⟩
7  |  if S2 = ⟨⟩
8  |  |  unmatched-t1.add(t1)
9  |  for s ∈ S2                                        // "inner" join matches
10 |  |  s.joined.add(t1 ∘ s.t2)
11 while S2 ≠ ⟨⟩
12 |  ⟨pop S2⟩
13 for t1 ∈ unmatched-t1                                // "left outer" e1 tuples
14 |  yield t1 ∘ ⊥τ2
```

To produce the desired ν_2–ν_1 output order, this algorithm has to stash joined tuples in tuple lists until they are ready to be yielded. The stack S_2 stores, for each encountered right input tuple t_2, a list of tuples "joined" thus far with t_2, as well as a list of joined tuples that have been "inherited" from descendants of $t_2.\nu_2$ that were removed from the stack.

The ⟨maintain S_2⟩ block maintains S_2 in such way that it contains exactly the join partners of t_1 (l. 6). These tuples are then joined with t_1. However, instead of yielding the joined tuples immediately, they are added to the appropriate "joined" lists (l. 7–10). If there is no join partner for t_1, it is stashed in the unmatched-t_1 list. The lists of joined tuples are rearranged and possibly yielded at the point when the corresponding tuples

are *removed* from S_2; this is done by the $\langle$pop $S_2\rangle$ block. After e_1 has been completely processed, stack S_2 is unwinded to ensure all stashed joined tuples are yielded (l. 11–12). Finally, "left outer" tuples for the unmatched-t_1 entries are yielded (l. 13–14). The output shall be ordered by v_2–v_1, so these tuples with $v_2 =$ `NULL` are yielded last.

The $\langle$maintain $S_2\rangle$ block has to identify e_2 tuples that are on the *ancestor* or *self* axes with respect to $t_1.v_1$:

```
15 ⟨maintain S2⟩:
16 while S2 ≠ ⟨⟩ ∧ H.is-before-post(S2.top().t2.v2, t1.v1)        // preceding axis
17 |  ⟨pop S2⟩
18 while p ≠ |e2|
19 |  t2 ← e2[p]
20 |  if ¬(t2.v2 = t1.v1 ∨ H.is-before-pre(t2.v2, t1.v1))
21 |  |  break
22 |  S2.push([t2, ⟨⟩, ⟨⟩])
23 |  if H.is-before-post(t2.v2, t1.v1)                          // preceding axis
24 |  |  ⟨pop S2⟩
25 |  p++
```

The first loop (l. 16–17) unwinds the stack to remove all tuples that are now on the *preceding* axis. After that, only *ancestor* tuples are remaining on S_2. The second loop (l. 18–25) adds further tuples from e_2 that are on the *ancestor*, *preceding*, or *self* axes to S_2. However, tuples on the *preceding* axis are immediately popped again (l. 23–24). This conveniently results in a "right outer" tuple being produced by $\langle$pop $S_2\rangle$. The net result is that only the relevant join partners for t_1 remain on S_2. Finally, the pseudo code for the $\langle$pop $S_2\rangle$ block is as follows:

```
26 ⟨pop S2⟩:
27 [t2, joined, inherited] ← S2.pop()
28 if joined = ⟨⟩
29 |  joined.add(⊥_τ1 ∘ t2)                          // "right outer" e2 tuple
30 if S2 ≠ ⟨⟩
31 |  S2.top().inherited.append(joined ∘ inherited)
32 else
33 |  for t ∈ joined ∘ inherited
34 |  |  yield t
```

If the popped t_2 tuple didn't have a join partner, a "right outer" tuple is produced (l. 28–29). If there is another entry remaining on S_2, that entry inherits the "joined" and "inherited" lists of t_2 (l. 30–31). Otherwise—when the last entry has been removed from S_2—the collected joined tuples are finally yielded in the order "joined"–"inherited" (l. 32–34).

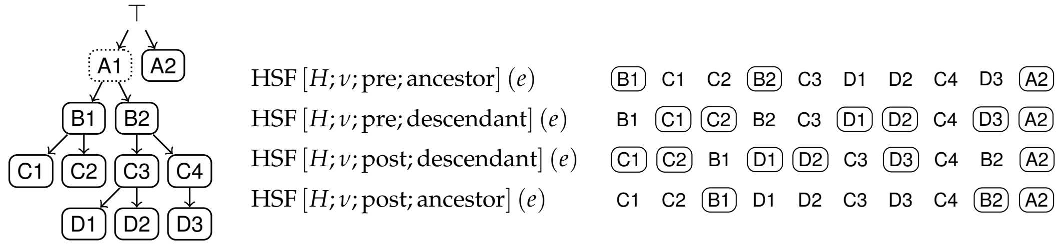


Figure 5.3: Applying a Hierarchy Staircase Filter to an example input.

5.2.6 Hierarchy Staircase Filter

Syntax. Hierarchy Staircase Filter $[H; \nu; \varsigma, a]\ (e)$, where H is a hierarchy index, ν is a `Node` attribute name, $\varsigma \in \{\text{pre}, \text{post}\}$, and $a \in \{\text{ancestor}, \text{descendant}\}$. The input is of type $e : \{\tau\}_b$ for some tuple type τ with a field $\nu : \texttt{Node}^H$. It is expected to be ordered by ν in ς-order. The output is of the same type as the input.

Functionality. The filter produces a subset $e' \subseteq e$ in the same order as the input. It removes any tuple t that is covered by another tuple t' whose node is either equal to $t.\nu$ or on the axis a, that is, $H.\text{axis}(t'.\nu, t.\nu) \in \{a, \text{self}\}$. The effect is that the nodes in the output form a “staircase” in the pre/post plane: both the preorder ranks and the postorder ranks of the nodes form a monotonically increasing sequence.

Example. Figure 5.3 shows a version of our example hierarchy with a second root A2. We apply HSF to a selection of the nodes that excludes A1. For each of the four cases of HSF, the figure shows the nodes sorted in the appropriate order (preorder or postorder), and marks the nodes that pass the respective filter.

As we already mentioned in § 5.1.4, the Staircase Filter is an important optimization for semi and anti joins. To our knowledge, the ideas have originally been examined systematically for the Staircase Join [41], a work in the context of XML/XPath processing. Consider the following example:

```
SELECT u.Node FROM T u
 WHERE EXISTS ( SELECT * FROM T v WHERE IS_DESCENDANT(v.Node, u.Node) AND φ(v.Node) )
```

A plan leveraging HSF would be:

1 $e_0 \leftarrow$ Hierarchy Index Scan $[H; \nu_1; \text{pre}; \{\}; \text{true}]\ ()$
2 $e_1 \leftarrow$ Hierarchy Index Scan $[H; \nu_2; \text{pre}; \{\}; \phi]\ ()$
3 $e_2 \leftarrow$ Hierarchy Staircase Filter $[H; \nu_2; \text{pre}; \text{descendant}]\ (e_1)$
4 $e_3 \leftarrow$ Hierarchy Merge Join $[H; \nu_2; \nu_1; \text{pre}; \{\text{ancestor}\}; \rtimes]\ (e_2, e_0)$

The presence of HSF does not affect the result, but it improves performance by removing tuples that are “superfluous” for the join from the input as early as possible.

Algorithm. The algorithms for the four different cases are largely straightforward. In the pseudo code, $|e| > 0$ is assumed for simplicity. If the input is empty, the output is trivially empty as well.

In the pre/ancestor case, a tuple t is excluded from the output if there is another $t' \in e$ with $H.\text{axis}(t'.\nu, t.\nu) \in \{\text{ancestor}, \text{self}\}$.

```
1 operator Hierarchy Staircase Filter [H; ν; pre; ancestor] (e)
2 yield e[0]
3 t′ : τ, t′ ← e[0]                                  // previous output tuple
4 for t ∈ e[1..[
5 │ if H.is-before-post(t′.ν, t.ν)                   // preceding axis
6 │ │ yield t
7 │ │ t′ ← t
```

In the pre/descendant case, a tuple t is excluded from the output if there is another $t' \in e$ with $H.\text{axis}(t'.\nu, t.\nu) \in \{\text{self}, \text{descendant}\}$.

```
1 operator Hierarchy Staircase Filter [H; ν; pre; descendant] (e)
2 t′ : τ, t′ ← ⊥τ                                    // previous input tuple
3 for t ∈ e
4 │ if H.is-before-post(t′.ν, t.ν)                   // preceding axis
5 │ │ yield t′
6 │ t′ ← t
7 if t.ν ≠ t′.ν
8 │ yield t
```

The post/descendant case is structurally similar to the pre/ancestor case:

```
1 operator Hierarchy Staircase Filter [H; ν; post; descendant] (e)
2 yield e[0]                                         // previous output tuple
3 t′ : τ, t′ ← e[0]
4 for t ∈ e[1..[
5 │ if H.is-before-pre(t′.ν, t.ν)                    // preceding axis
6 │ │ yield t
7 │ │ t′ ← t
```

Due to the nature of postorder, post/ancestor is the most complicated case. Because we cannot be sure whether the currently inspected tuple is already the "uppermost" node or whether an even higher ancestor is going to appear (such as the root itself), we need to keep the seen tuples on a stack and delay yielding them until we have processed *all* of the input. We can, however, safely remove any descendants from the stack at the time we push a new tuple.

```
1 operator Hierarchy Staircase Filter [H; ν; post; ancestor] (e)
2 S : Stack ⟨τ⟩
```

```
3 for t ∈ e
4 |  while S ≠ ⟨⟩ ∧ ¬H.is-before-pre(S.top().ν, t.ν)        // descendant or self axis
5 |  |  S.pop()
6 |  S.push(t)
7 for t ∈ S
8 |  yield t
```

Note how these algorithms also filter out duplicate ν values, as tuples on the *self* axis will not pass the "is-before" checks.

The time complexity of these algorithms is in $\mathcal{O}(|e|)$. Their space complexity is in $\mathcal{O}(1)$, except for the post/ancestor case, which is in $\mathcal{O}(|e|)$.

5.2.7 Structural Grouping: Overview

In this overview we discuss various approaches for the unary structural grouping $\hat{\Gamma}^{<}_{x:f}$ and binary structural grouping $\ltimes\!\Gamma^{\theta}_{x:f}$ operations we defined in § 5.1.5 and § 5.1.6.

Join-Group. A generic approach for $\ltimes\!\Gamma$ is to treat θ as an opaque join predicate with partial order properties, and stick to a sort-based join–group–aggregate plan: sort both inputs e_1 and e_2 in either preorder or postorder according to θ, then perform a sort-based left outer join $e_1[t] ⟕_\theta e_2[u]$, and finally use sort-based unary grouping to compute the result.

```
1 Join-Group [θ; x : f](e1, e2) :=
2     e1 ← Sort [θ] (e1)
3     e2 ← Sort [θ] (e2)
4     e3 ← Join [θ] (e1[t1], e2[t2])   // sort-based
5     Group [t1.∗; x : f] (e3)   // sort-based
```

This requires a non-equi join operator that retains the order of e_1 and deals appropriately with the fact that some tuples may not be comparable using θ. Unfortunately, the standard sort-based merge join supports only *equi* joins. If we make no further assumptions about e_1, e_2, and θ, we can only use some variant of Nested Loop Join, making the time complexity an unattractive $\mathcal{O}(|e_1| \cdot |e_2|)$. We refer to this approach by "Join-Group."

HMJ-Group. When $\ltimes\!\Gamma$ and $\hat{\Gamma}$ are used for hierarchical computations, where θ and $<$ operate on `NODE` fields, the underlying hierarchy index H can and should be leveraged. A significant improvement over the generic Join-Group approach is to use a sort-based Hierarchy Merge Join. Alternatively, a Hierarchy Index Join can be used if e_2 is compatible, that is, if e_2 is the hierarchical base table T itself and can be enumerated through a Hierarchy Index Scan. We refer to the variant of Join-Group based on Hierarchy Merge Join by "HMJ-Group."

HG and HMGJ. The two mentioned approaches keep implementation efforts low by reusing existing operators. However, they cannot evaluate the structural recursion of $\hat{\Gamma}$, and they suffer from the efficiency issues we already noted in § 2.2.4: They inherently need to materialize and process *all* $<$ join pairs rather than just the $<:$ pairs during query evaluation, and they are unable to reuse results from covered tuples.

We therefore propose four new special-purpose operators: Hierarchy Merge Groupjoin for binary structural grouping $\bowtie\!\!\Gamma$, and Hierarchical Grouping for unary structural grouping $\hat{\Gamma}$, each in a top-down and a bottom-up variant. The top-down variants require the inputs to be sorted in preorder, the bottom-up variants in postorder; this order is retained in the output.

The relational algebra definitions of binary and unary structural grouping apply the given aggregation function f to different bags of input tuples. In the pseudo code for HMGJ and HG, we use an abstract data type Aggregate to represent such a bag X. It supports the self-explanatory operations add(u), merge(X'), and clear$()$. While processing their inputs, the HMGJ and HG algorithms create one Aggregate instance X per tuple $t \in e_1$, assemble the appropriate input tuples in it, and pass it to the given aggregation function $f(X)$ or $f(t, X)$ in order to obtain $t.x$. In the *actual* query-specific implementation of an Aggregate and its operations, significant optimizations may be possible depending on the given f function, as we discuss in § 5.2.10.

5.2.8 Hierarchy Merge Groupjoin

Syntax. Hierarchy Merge Groupjoin $[H; \nu_1; \nu_2; \varsigma; A; x : f]\,(e_1, e_2)$, where H is a given hierarchy index, ν_1 and ν_2 are `Node` attribute names, $\varsigma \in \{\text{pre}, \text{post}\}$ is the input order, A is a set of hierarchy axes, and $x : f$ is a name/expression pair.

f must be a scalar aggregation function of type $\{\tau_2\}_b \to \mathcal{N}$ for some type $\mathcal{N}$. The inputs e_1 and e_2 must have a `Node` field named ν_1 and ν_2, respectively. Formally, $e_i : \{\tau_i\}_b$ for a type τ_i that has a field $\nu_i : \texttt{Node}^H$. The inputs are required to be sorted by ν_i in ς-order. The output is of type $\{\tau_1 \circ [x : \mathcal{N}]\}_b$ and has the same order as e_1.

Algorithm. The HMGJ algorithms evaluate structural grouping $\bowtie\!\!\Gamma$ (§ 5.1.5). They come in a bottom-up variant that handles $\varsigma = \text{post}$ and $A = \{\text{self}, \text{descendant}\}$, and a top-down variant that handles $\varsigma = \text{pre}$ and $A = \{\text{ancestor}, \text{self}\}$. The following pseudo code shows the basic framework for both variants. Overall, the logic is reminiscent of a left outer Hierarchy Merge Join A (§ 5.2.5).

```
1 operator Hierarchy Merge Groupjoin [H; ν1; ν2; ς; A; x : f] (e1, e2)
2 S1 : Stack ⟨[ν1 : Node^H, X : Aggregate⟨τ2⟩, i : int]⟩
3 S2 : Stack ⟨τ2⟩
4 p : int, p ← 0
5 X : Aggregate⟨τ2⟩
```

```
6  for t1 ∈ e1
7  │ if S1 ≠ ⟨⟩ ∧ S1.top().ν1 = t1.ν1
8  │ │ [·, X, ·] ← S1.top()
9  │ │ yield t1 ∘ [x : f(X)]
10 │ │ continue
11 │ X.clear()
12 │ ⟨collect input⟩
13 │ yield t1 ∘ [x : f(X)]
14 │ S1.push([t1.ν1, X, |S2|])
```

Two stacks are used: Analogously to Hierarchy Merge Join A, stack S_2 stashes processed e_2 tuples that may become relevant as join partners. Stack S_1 collects processed nodes ν_1 from e_1 with the corresponding aggregates X of matched e_2 tuples for reuse. Variable i refers to a position on S_2 and is needed in the top-down case.

For each $t_1 \in e_1$ (l. 6) we either reuse X from a previous equal node (l. 7–10) or assemble X via the ⟨collect input⟩ block, which differs for the top-down and bottom-up cases. Finally an output tuple is produced and X is memorized on S_1.

The bottom-up variant (postorder inputs) essentially performs a join on the *descendant* or *self* axes with left outer join semantics.

```
15 ⟨collect input⟩ — bottom up:
16 while S1 ≠ ⟨⟩ ∧ ¬H.is-before-pre(S1.top().ν1, t1.ν1)
17 │ [·, X', ·] ← S1.pop()
18 │ X.merge(X')
19 while S2 ≠ ⟨⟩ ∧ ¬H.is-before-pre(S2.top().ν2, t1.ν1)
20 │ X.add(S2.pop())
21 while p ≠ |e2|
22 │ t2 ← e2[p]
23 │ if t2.ν2 = t1.ν1
24 │ │ X.add(t2)
25 │ if ¬H.is-before-post(t2.ν2, t1.ν1)
26 │ │ break
27 │ if ¬H.is-before-pre(t2.ν2, t1.ν1)
28 │ │ X.add(t2)
29 │ else
30 │ │ S2.push(t2)
31 │ p++
```

The algorithm consists of three steps: The first loop (l. 16) removes all covered *descendant* entries from S_1 and merges their aggregates into X using the merge() operation. This operation is the key to effectively reusing partial results as motivated in § 2.2.4. The second loop (l. 19) adds relevant matches on the *descendant* or *self* axes from S_2 to X. The third loop (l. 21) advances the right input e_2 up to the first postorder successor of ν_1. Any encountered t_2 either is a postorder predecessor or has $t_2.\nu_2 = \nu_1$; if t_2 is also a preorder successor, it is a descendant. *Descendant* or *self* matches are added

straight to X (l. 28), whereas *preceding* tuples are stashed on S_2 (l. 30).

The top-down variant (preorder inputs) joins on the *ancestor* or *self* axes:

```
32 ⟨collect input⟩ — top down:
33 while S1 ≠ ⟨⟩ ∧ H.is-before-post(S1.top().ν, t1.ν1)
34 |   S1.pop()
35 j : int, j ← 0
36 if S1 ≠ ⟨⟩
37 |   [·, X′, j] ← S1.top()
38 |   X.merge(X′)
39 while j ≠ |S2| ∧ H.is-before-post(t1.ν1, S2[j].ν2)
40 |   X.add(S2[j])
41 |   j++
42 pop S2[j], . . . , S2.top()
43 while p ≠ |e2|
44 |   t2 ← e2[p]
45 |   if t2.ν2 = t1.ν1
46 |   |   X.add(t2)
47 |   |   S2.push(t2)
48 |   if ¬H.is-before-pre(t2.ν2, t1.ν1)
49 |   |   break
50 |   if ¬H.is-before-post(t2.ν2, t1.ν1)
51 |   |   X.add(t2)
52 |   |   S2.push(t2)
53 |   p++
```

A peculiarity of the top-down case is that S_1 and S_2 entries may be consumed multiple times and therefore cannot be immediately popped from the stacks. S_1 and S_2 are maintained in such way that they comprise the full chain of *ancestor* tuples from e_1 and e_2 relative to ν_1. Field i on S_1 establishes the relationship to S_2: For an S_1 entry $[\nu, X, i]$, the bag X incorporates all matches for ν, corresponding to the S_2 range $[0, i[$ (i. e., from the bottom to position i, exclusively). If there is another S_1 entry $[\nu', X', i']$ below, then ν' is the covered *ancestor* of ν, and X consists exactly of X' plus the S_2 tuples at positions $[i', i[$.

Maintaining these invariants requires four steps: First (l. 33), obsolete *preceding* entries are popped from S_1. Second (l. 36), any remaining entry on S_1 is an *ancestor*, so its aggregate X' is reused. Third (l. 39), any additional ancestors t_2 that were not already in X' (starting from position j) are added to X. Then, the remaining S_2 tuples from positions j to top are *preceding* and therefore obsolete (l. 42). Finally (l. 43), e_2 is advanced up to the first preorder successor of ν_1; in the process, tuples on the *ancestor* or *self* axes are added to X and S_2, whereas *preceding* tuples are ignored.

The two algorithms we discussed have *left outer* join semantics and include the *self* axis. They can be adapted for other axes (child/parent and the non-*self* variants) as well as *inner* joins analogously as described for Hierarchy Merge Join in § 5.2.5.

5.2.9 Hierarchical Grouping

Syntax. Hierarchical Grouping $[H; \nu; \varsigma; A; x : f]\ (e)$, where H is a hierarchy index, ν is a `Node` attribute name, $\varsigma \in \{\text{pre}, \text{post}\}$ is the input order, A is a set of hierarchy axes, and $x : f$ is a name/expression pair.

The input is of type $e : \{\tau\}_b$, where τ is a tuple type with a $\nu : \mathtt{Node}^H$ field, and e is ordered by ν in postorder (bottom up) or preorder (top down).

The output is of type $\{\tau'\}_b$, where $\tau' := \tau \circ [x : \mathcal{N}]$. It has the same order as e.

Algorithm. This HG algorithms handle *unary* structural grouping with structural recursion. In a single pass through the input e, HG effectively issues the following call sequence for each tuple t:

X.clear()
X.add(u) for each $u <: t$
yield $t \circ [x : f(t, X)]$

The following pseudo code shows the basic framework for both variants of HG:

```
1  operator Hierarchical Grouping [H; ν; ς; A; x : f] (e)
2  S : Stack ⟨[ν : Node^H, u : τ', X : Aggregate⟨τ'⟩]⟩
3  X : Aggregate⟨τ'⟩
4  for t ∈ e
5  |  if S ≠ ⟨⟩ ∧ S.top().ν = t.ν
6  |  |  skip   // reuse previous X
7  |  else
8  |  |  X.clear()
9  |  |  ⟨collect input⟩
10 |  yield t' ← t ∘ [x : f(t, X)]
11 |  S.push([t.ν, t', X])
```

The stack S (line 2) manages previously processed tuples u and their computation states, that is, $u.x$ and the corresponding aggregate X for potential reuse. For each input tuple t (l. 4) the algorithm first checks whether $t.\nu$ matches the previous node; in this case, it reuses X as is. (This step can be omitted if ν is known to be duplicate-free.) Otherwise, the ⟨collect input⟩ block (l. 9) maintains the stack and collects the tuples X covered by t. Then the algorithm computes $f(t, X)$, constructs and yields an output tuple and puts it onto the stack together with X for later reuse.

Regarding "collect input" let us first consider the bottom-up case (postorder input):

```
12 ⟨collect input⟩ — bottom up:
13 while S ≠ ⟨⟩ ∧ ¬H.is-before-pre(S.top().ν, t.ν)
14 |  [·, u, X_u] ← S.pop()
15 |  X.add(u)   // leverage X_u if possible!
```

Since the input e is in postorder, previously processed tuples on S, if any, are postorder predecessors and as such on the *descendant* and *preceding* axes relative to $t.\nu$,

in that order when viewed from the top of stack (whereas upcoming tuples from e are postorder successors and thus will be on the *ancestor* or *following* axes). Therefore, the covered tuples X we need for t are conveniently placed on the upper part of S. The "while" loop (l. 13) collects and removes them, as they will no longer be needed. Any remaining S entries are *preceding* and irrelevant to t—as preorder and postorder predecessors—but might be consumed in a later iteration.

Let us now consider the top-down case (preorder input):

```
16 ⟨collect input⟩ — top down:
17 while S ≠ ⟨⟩ ∧ H.is-before-post(S.top().ν, t.ν)
18 │ S.pop()
19 if S ≠ ⟨⟩
20 │ for [ν, u, X_u] ∈ upper part of S where ν = S.top().ν
21 │ │ X.add(u)   // leverage X_u if possible!
```

S may, when viewed from the top, contain obsolete *preceding* tuples, then relevant covered *ancestor* tuples to add to X, then further non-immediate ancestors which may still be needed in a future iteration. The "while" loop (l. 17) first dismisses the *preceding* tuples. If there is an entry left on top of S (l. 19), it is a covered ancestor $u <: t$, and the "for" loop (l. 20) collects it and further tuples below with equal ν (if not distinct in e). Due to the tree-structured data flow, there cannot be any further covered tuples. Unlike in the bottom-up case, we cannot pop the covered entries after adding them to X, since they may still be needed for upcoming *following* tuples (e. g., a sibling of ν).

Note that we do not need any explicit checks for $<:$ in this algorithm—the covered tuples are identified implicitly. Note also that in l. 15 and 21, the full X_u state corresponding to $u.x$ is available to the add() operation. This state may be needed for non-trivial computations where $u.x$ alone does not provide enough information, as we discuss in § 5.2.10. In case it is *not* needed, we do not need to keep the X objects (marked in teal in the code) on the stack at all. Likewise, we may include only the fields of u that are *actually* accessed by f to minimize memory consumption.

5.2.10 Structural Grouping: Further Discussion

Recall from § 5.1.5 and § 5.1.6 that the Hierarchical Grouping operator is primarily used for evaluating `RECURSIVE` expressions on hierarchical windows, and the Hierarchy Merge Groupjoin operator is used for *non-recursive* expressions (via self-grouping $e \bowtie\Gamma e$) as well as certain classes of join–group–aggregate statements. Handling further details of hierarchical windows—such as different variants of window frame and `EXCLUDE` clauses—would require further additions to the algorithms; in particular, tuples with equal `Node` values must be identified and handled as a group. As these adaptions are straightforward, we omit their discussion.

Inline Computations. The following optimization is crucial to the practical performance of HMGJ and HG: While their pseudo code literally collects the input tuples to the aggregation function f into a bag X, we can often avoid such buffering altogether

by evaluating the function *on the fly*. To this end the query compiler has to generate specific code in-place for the four different Aggregate operations:

$$①\ X.\text{clear}() \quad ②\ X.\text{add}(u) \quad ③\ X.\text{merge}(X') \quad ④\ f(t, X).$$

Consider the rollup Expression 1b from Figure 5.1 (p. 118): Rather than a list of tuples, the actual state of X would be a simple partial sum $x : \mathcal{N}$, and the Aggregate operations would boil down to

$$①\ x \leftarrow 0 \quad ②\ x \leftarrow x + u.x \quad ③\ x \leftarrow x + X'.x \quad ④\ x + t.x.$$

These definitions work with both HG and HMGJ. For a structurally recursive computation with HG, consider Expression 4c: Here the state remains the same as for Expression 1b, but operation ② becomes $x \leftarrow x + u.\text{Weight} * u.x$.

Eliminating the bag of tuples X like this is possible whenever either the scalar x value itself or some other data of $\mathcal{O}(1)$-bounded size can adequately represent the required result information of a sub-computation. This characterization roughly corresponds to the classes of *distributive* (e.g. `COUNT`, `MIN`, `MAX`, and `SUM`) and *algebraic* aggregation functions (e.g. `AVG`, standard deviation, and "k largest/smallest") identified by Gray et al. [38].

There are of course also SQL expressions, such as `ARRAY_AGG` or `DISTINCT` aggregates, for which we *have* to actually maintain the bag X literally, or some state of size $\Theta(|X|)$. Consider for example `COUNT(DISTINCT Weight)`: To evaluate this using either HG or HMGJ, the Aggregate has to maintain a set of distinct Weight values. But even then, our mechanism for reusing the results of sub-computation can provide certain optimization opportunities. Maintaining the distinct Weight values could for example be speeded up by employing an efficient set union algorithm for the merge() operation.

Complexities. With the optimization opportunities discussed in the previous paragraph in mind, let us consider the time and space complexities of HMGJ and HG.

If the computation is indeed done inline as discussed, the size of the Aggregate objects and the times of their operations are actually in $\mathcal{O}(1)$. Under this assumption, the time and space complexity is $\mathcal{O}(|e|)$ for HG, and $\mathcal{O}(|e_1| + |e_2|)$ for HMGJ.

However, if the computation cannot be inlined, we fall back to literally collecting the respective input tuples in the Aggregate objects. Our algorithms then essentially degenerate to merge joins, and their time/space complexities become $\mathcal{O}(|e_1| + |e_2| + |e_1 \bowtie e_2|)$.

To establish these asymptotic complexities, an amortized analysis is needed in order to argue that the inner loops of the HMGJ and HG algorithms do not contribute to the overall complexity. Regarding Hierarchical Grouping (algorithm on p. 140), observe that the outer "for" loop pushes each e tuple once onto S (so $|S| \leq |e|$), whereas the inner "while" loops remove one S entry per iteration; their bodies can thus be amortized to the respective pushes. Regarding Hierarchy Merge Groupjoin (see p. 138), the loop bodies of l. 21 and l. 43 are executed $|e_2|$ times in total, regardless of the outer

loop; at most $|e_1|$ and $|e_2|$ tuples are pushed onto S_1 and S_2, respectively; and since the other loops pop either an S_1 or S_2 entry per iteration, an analogous argument applies.

5.2.11 Hierarchy Rearrange

Syntax. Hierarchy Rearrange $[H; \nu; \varsigma \to \varsigma'] \, (e)$, where H is a hierarchy index, ν is a node attribute name, and $\varsigma \to \varsigma'$ is either "pre $\to$ post" or "post $\to$ pre."

The input must have a `Node` field ν. Formally, $e : \{\tau\}_b$ for some tuple type τ with a field $\nu : \texttt{Node}^H$. The input is required to be sorted by ν in ς-order.

The output is of type $\{\tau\}_b$ and sorted by ν in ς'-order.

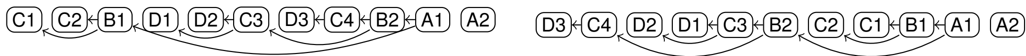
Functionality. To sort a table of nodes in preorder or postorder, we can always use an ordinary Sort operator with is-before-ς() as comparison predicates:

$$\text{Sort}\,[<]\,(e), \quad \text{where} \quad t_1 < t_2 \;:\Longleftrightarrow\; H.\text{is-before-pre}(t_1.\nu, t_2.\nu).$$

However, if the input happens to be already sorted in either postorder or preorder, the Hierarchy Rearrange operators can exploit that order to perform the same sorting tasks more efficiently.

Remarks. The main use case of HR is as a "glue" operator to combine preorder-based and postorder-based operators such as HMJ, HMGJ, and HG in a plan.

To understand how the algorithms work in principle, note that the *postorder* sequence of the nodes in a hierarchy is somewhat comparable to the *reverse preorder* sequence in that a node appears before any of its ancestors. The difference is that in postorder the children of each node (and with them their subtrees) are arranged in their original order, whereas in reverse preorder they are reversed. Compare, for instance, the postorder (left) and reverse preorder (right) sequences of our familiar example hierarchy:



To convert a general input e from preorder to postorder, we therefore have to delay yielding a tuple t until all its descendants have been processed. This reverses the "vertical" order of the input. At the same time, we need to make sure that all immediate descendants of the tuple t remain in their original order. This way we keep the "horizontal" order intact. The algorithms we discuss below achieve this by stashing their input tuples on a stack and carefully unwinding this stack at the point when the respective descendants (in the pre $\to$ post case) or ancestors (in the post $\to$ pre case) have been processed.

Algorithm for "pre $\to$ post." This case is simple and efficient:

1 **operator** Hierarchy Rearrange $[H; \nu; \text{pre} \to \text{post}] \, (e)$
2 S : Stack $\langle \tau \rangle$

```
3  for t ∈ e
4  |  while S ≠ ⟨⟩ ∧ H.is-before-post(S.top().ν, t.ν)
5  |  |  yield S.pop()
6  |  S.push(t)
7  while S ≠ ⟨⟩
8  |  yield S.pop()
```

The algorithm unconditionally stashes each input tuple t on the stack S. But first, the inner loop (l. 4–5) pops and yields any tuples whose node ν is on the *preceding* axis with respect to $t.\nu$. The effect of performing this loop at each iteration is that runs of tuples in *preceding–following* relationships (such as a run of siblings) will be yielded in their originally encountered order, whereas runs of tuples in *ancestor–descendant* relationships remain on the stack until all their descendants have been processed, and are then popped in reverse order. In other words, the "horizontal" order is maintained but the "vertical" order is reversed, which results in the desired postorder.

However, the pseudo code as shown also reverses the relative order of runs of tuples with equal ν values (i. e., in *self* relationships). If this is not desired, the inner loop needs to be refined so as to first identify those runs and yield them in their original order.

The runtime complexity of this operator is in $\mathcal{O}(|e|)$. An attractive property is that e is pipelined at least partially. The worst-case space complexity is in $\mathcal{O}(|e|)$ as well. This worst case happens, for example, when all ν values are equal. However, if we assume that all nodes are distinct, then the stack size $|S|$ is bounded by the height of the hierarchy H, since S then forms a run of tuples in *ancestor–descendant* relationships at any point.

Algorithm "post → pre." This case is comparatively involved. Due to the nature of postorder, pipelining is very limited; for instance, the root node of a hierarchy appears *last* in postorder but *first* in preorder. Since the algorithm cannot know whether it has already seen the "highest" ancestor in a run of tuples, it must buffer and analyze its whole input first before it can yield the first output tuple.

```
1  operator Hierarchy Rearrange [H; ν; post → pre] (e)
2  S : Vector ⟨[t : τ, cov : int]⟩
3  for t ∈ e
4  |  if S ≠ ⟨⟩ ∧ S.top().t.ν = t.ν
5  |  |  c ← S.top().cov
6  |  |  S.top().cov ← |S| − 1
7  |  |  S.push([t, c])
8  |  |  continue
9  |  cov ← |S|
10 |  while cov ≠ 0 ∧ ¬H.is-before-pre(S[cov − 1].t.ν, t.ν)
11 |  |  cov ← S[cov − 1].cov
12 |  S.push([t, cov])
13 ⟨enumerate tuples on S⟩
```

The algorithm stashes each input tuple on the vector S, which is built up like a stack but also accessed in a random manner. For each tuple t it identifies the range of previously seen tuples that are "covered" by t, that is, on the *self* or *descendant* axes with respect to $t.v$. The covered ranges are represented by the "cov" fields on the vector: cov is the index of the first (lowest) covered entry on S; that is, for each i, entry $S[i]$ covers entries $S[S[i].\text{cov}, i[$. In other words, the tuples $S[\text{cov}, i[$ are on the *self* or *descendant* axes, whereas the tuples $S[0, \text{cov}[$ are on the *preceding* axis with respect to $S[i].v$. The block l. 9–11 determines cov for t by hopping over S backwards (skipping any transitively covered entries). If, however, $t.v$ equals the previously processed node, special care has to be taken (l. 4–8). The entry for t then steals the cov value from the previous tuple u, whose cov value is adjusted to "empty range." This helps us identify runs of tuples with equal v values later on: In each such run $S[i_1, i_2]$, the uppermost tuple $S[i_2]$ covers the other tuples $S[i_1, i_2[$ in addition to the descendants, whereas each tuple in $S[i_1, i_2[$ covers only itself.

With the collected coverage information the ⟨enumerate tuples on S⟩ block can finally enumerate the output in the desired order. It effectively performs a preorder traversal of the constructed vector:

```
14 ⟨enumerate tuples on S⟩:
15 Q : Stack ⟨int⟩
16 for c ← |S|; c ≠ 0; c ← S[c].cov
17 |  c--
18 |  Q.push(c)
19 while Q ≠ ⟨⟩
20 |  c ← Q.pop()                                  // current subtree: c
21 |  cov ← S[c].cov                                // limit for traversal
22 |  c′ ← c
23 |  if c′ ≠ cov
24 |  |  while c′ ≠ 0 ∧ (S[c′ − 1].t.v = S[c′].t.v)
25 |  |  |  c′--
26 |  for j ∈ [c′, c]
27 |  |  yield S[j].t
28 |  for c′; c′ ≠ cov; c′ ← S[c′].cov
29 |  |  c′--
30 |  |  Q.push(c′)
```

The stack Q stores the indexes of remaining S entries to visit. The first loop (l. 16–18) initializes Q with the "top-level" tuples that are not covered by any other tuple. The main loop (l. 19) repeatedly picks the top-most tuple c on Q to visit. The tuples belonging to the subtree of $S[c].v$ can be found in the range $S[S[c].\text{cov}, c]$. The loop l. 22–25 first identifies the range $[c', c]$ of equal nodes. $S[c]$ and its equal nodes are then yielded in the original order (l. 26–27). The loop l. 28–30 then enumerates c's directly covered entries backwards and pushes them onto Q (similar to l. 16–18). This has the

effect that these entries will be visited subsequently in their correct (original) order.

In an actual implementation of this algorithm, it may be beneficial to not actually copy the input tuples onto S. Instead, the input can be materialized, so that S only needs to store the node ν and a pointer to the actual tuple. This approach improves cache efficiency if the tuples are large.

As another optimization, we can flush the stack early whenever the previous node was a root. To this end, insert an is-root() check after l. 8:

if $S \neq \langle\rangle \wedge H.\text{is-root}(S.\text{top}().t.\nu)$
  ⟨enumerate tuples on S⟩
  $S \leftarrow \langle\rangle$

5.3 Further Related Work and Outlook

This section points to relevant related work with regard to query processing on hierarchical data. We also outline several open problems in the context of our framework that would be interesting for future work.

Query Execution and Pipelining. Throughout this chapter we assumed a bottom-up push-based model of query execution, as described in [76]. This model allows algorithms which process their input in a single pass—such as our HIJ, HMJ, HMGJ, and HG operators—to "pipeline" the tuples in the sense that the relevant attribute values (in particular, the `Node` values) can potentially remain in processor registers during execution instead of being materialized in memory. With engines that compile query plans to native machine code, these algorithms can deliver excellent performance. However, binary operators can inherently pipeline only *one* of their inputs and always have to materialize the other input. (Our operators by convention materialize their *right* input.)

In the pull-based *iterator model* [37], both inputs would be accessed through an iterator interface with methods open(), next(), close(), et cetera. We can adapt our algorithms to this model by simply rewriting the outer loops over e_1 and the accesses to e_2 in terms of iterators. Although the iterator abstraction limits performance due to function call overhead, it is conceptually elegant in that both inputs of binary operators are handled uniformly. When they are accessed strictly sequentially, they can also be "pipelined" in the sense that all tuples are considered just in time without any upfront materialization.

In theory, the performance gains through pipelining can be maximized (and the overall memory usage reduced) by choosing the bigger input as the pipelined input and materializing the smaller input. However, some inherent asymmetries in our join algorithms (HIJ, HMJ, HMGJ) complicate matters: swapping the inputs and inverting the join condition does more than just swap the pipelined/materialized roles: it usually requires using a different variant of the algorithm with different properties that may impact the rest of the query plan. One therefore wants the query optimizer to be aware of the plan-relevant characteristics of the different variants of each algorithm, and to

also reflect their performance differences in the cost model. Studying these issues and how they affect algorithm selection is a possible direction of future work.

Operators on Tree-Structured Data. Since XML documents are inherently hierarchical and commonly stored in XML-enhanced relational databases in the form of tables (see § 2.3), algorithms for the efficient processing of queries on tree-structured data were also studied in that context. These are sometimes referred to as *structural* or *tree-aware* join operators. Examples are MPMGJN [118], tree-merge and stack-tree [53], Staircase Join [41], and Twig^{2}Stack [19]. Similar to our algorithms, they in general work on ordered inputs, use logic reminiscent of merge self-joins, and leverage an available (though usually hard-wired) tree encoding. [19] notably also works with bottom-up processing on postorder inputs, whereas other operators (e. g. [53]) usually require their inputs to be in *document* order (i. e., preorder).

Beyond binary structural joins, powerful path and tree pattern matching operators were proposed in the XML context; for example, TwigStack [11] and variations [51] for handling so-called twig joins. Translated to the SQL world, these operations could be characterized as n-way joins based on a tree-shaped pattern. While such complex patterns are comparatively easy to express in XPath, they are beyond our requirements for handling hierarchical data in RDBMS.

Although they are a fruitful source of inspiration, not all techniques from the XML world fit into our setting. Most of the more sophisticated join operators were designed to work directly on appropriately pre-processed and indexed XML documents. In contrast, our operators are not restricted to the hierarchical base table itself; they can be applied to *arbitrary* input tables containing a `NODE` field. As indexing the inputs on the fly seems infeasible, we rely only on the hierarchy index of the base table. Many of the proposed techniques are not applicable in this setting. As an example, we consider Staircase Join [41]. This operator performs an XPath axis step given a set of context nodes. It achieves high performance using sophisticated techniques to skip over irrelevant nodes. However, these techniques require direct access to the indexed XML document table, which needs to be encoded using the pre/post labeling scheme and physically sorted in preorder. In terms of SQL and our algorithms, an axis step is comparable to a right semi-join between a table of context nodes and the hierarchical base table, with duplicate elimination on the result (`DISTINCT`). For queries of this type, Staircase Join could be roughly emulated by combining a Hierarchy Index Join with a Staircase Filter on the left input. However, we expect the share of queries that will benefit from this optimization to be rather limited in the SQL context—as opposed to XPath, where they are routine.

Further Applications. We noted earlier that many algorithms from the literature were originally hard-coded against specific labeling schemes. As our algorithms are formulated in terms of a small set of index primitives, they are *generic* in the sense that we can plug in arbitrary indexing schemes in the backend. This ensures a clean separation between the physical operators and the layer of index implementations. Thus,

an interesting direction of future work would be to redesign some of the more sophisticated algorithms from the literature in terms of our index interface. As an example, the following pseudo code shows an adaption of the mentioned Staircase Join for the descendant axis:

```
1 function staircasejoin_desc(doc, context)
2 |  for each successive pair (c1, c2) in context
3 |  |  scanpartition_desc(c1, c2)
4 |  c ← last node in context
5 |  n ← end of doc
6 |  scanpartition_desc(c, n)
```

```
1 function scanpartition_desc(c1, c2)
2 |  for (c ← H.pre-next(c1); c ≠ c2; c ← H.pre-next(c))
3 |  |  if H.is-before-post(c, c1)
4 |  |  |  yield c
5 |  |  else
6 |  |  |  break // skip
```

As Staircase Join relies only on pre/post traversal and axis checks, converting the algorithm from the original definition (algorithms 2 and 3 in [41]) is particularly easy. However, such an adaption should be possible for other existing algorithms as well. This way our framework of indexing schemes and query processing algorithms could potentially be leveraged in a much broader range of database systems and applications, XPath processing being just one example.

Techniques for Grouping and Aggregation. Our algorithms HMGJ and HG perform a particular kind of grouping on relational tables. Therefore, relevant related work comes from the extensive literature on efficiently evaluating `GROUP BY` in SQL. Commonly, either sort-based or hash-based methods are used [37]. Similar to ordinary sort-based grouping, our operators rely on ordered inputs and are order-preserving.

Groupjoin aka. *binary grouping* [15, 65, 74] improves join–group–aggregate plans by fusing $\bowtie$ and Γ, which allows the engine to avoid materializing the intermediate join result. While [74] discusses mainly hash-based equi-groupjoins, [65] and [20] also consider the *non-equi* case. This case is more comparable to our setting, although identifying the groups is more involved with structural grouping. [20] discusses a method based on so-called *θ-tables*, which enable reusing the results from "covered" groups and are roughly comparable to the stacks our operators are based on.

`GROUP BY CUBE` and `ROLLUP` can be viewed as constructs for multi-dimensional hierarchical grouping, although they work only on a specific form of "denormalized" tables (see § 2.3.5). [38] discusses approaches to implement these constructs. The naïve approach is to execute 2^k separate aggregation operations and collect their results using `UNION`. A more sophisticated approach uses a dedicated single-pass operator that is able to reuse results of lower levels. This idea is similar in spirit to our approach. MD-Join [15] is another kind of groupjoin for evaluating `CUBE`, which is also able to reuse results from finer-grained groupings for coarser-grained groupings.

Windowed Tables [116] are another powerful way of grouping in SQL that resembles a "self-groupjoin." Their efficient evaluation has received surprisingly little attention from the research community. [58] is a recent guide with a focus on modern multi-core architectures. Alas, techniques for standard windowed tables cannot easily be adapted to our hierarchical windows due to their unique semantics.

Estimation and Statistics for Hierarchical Data. Being able to precisely estimate the output sizes of individual plan operations is a highly important basis for cost-based query optimization. In connection with queries on hierarchical data, cardinality estimation comes into play when a hierarchy predicate such as "$H.\text{depth}(\nu) \leq 3$" is used to filter tuples; when filtering hierarchy operators are used, such as Staircase Filter, or Index Scan with additional predicates; and most importantly, for the output of a hierarchy join, taking into account the axis and the type of join.

Cardinality estimation is already difficult for ordinary predicates and equi joins. For hierarchy joins the estimation errors tend to be even less predictable. Simple statistics for estimating equi join results—such as the number of distinct key values in the two inputs—are less useful for hierarchy joins. For example, consider a join on the *descendant* axis where one distinct `NODE` value is joined with a table of leaves. If the left input node happens to be the root, the join can easily degenerate into a cross product, but it could as well be empty if it happens to be a leaf itself. Furthermore, any specific formula for estimation would have to incorporate knowledge or assumptions on the hierarchy structure. A hierarchy of N nodes might contain only roots and thus be extremely flat, but it might also be unusually deep (e. g., more than 8 levels) or even a single linear chain of child nodes ($H.\text{height}(\top) = N$). A pragmatic model for cardinality estimation would be to assume a basic regular structure where each node has a fixed number of k children, and to set k to the average number of children in the actual data. Based on such a model, formulas could be stated to estimate, for instance, the total number of descendants for a (known) set of nodes. While such hand-crafted heuristics might work well in sane cases, more elaborate models for cardinality estimation are desirable, such as path query synopses [119]. Techniques such as sampling and re-optimization [107] would be useful in the context of hierarchical data as well. This is an interesting line of future work.

A related topic are *statistics* on hierarchical tables. To support cost estimation, one wants to be able to efficiently gage compact but useful summarized data about the structure of a hierarchy at bulk-build time, and update the information at regular intervals. For example, if we know only the minimum, average, and maximum number of children of the (non-leaf) nodes—three integers that can be collected during a single hierarchy traversal—we can already compute a reasonable estimate of the cardinality of an is-parent() join. The hierarchy index interface itself already provides access to interesting measures: For example, the higher the average depth() or size() of a set of nodes, the less join partners they will presumably have in a join on the *descendant* axis.

Yet another challenge is *cost estimation*, where the query optimizer estimates the CPU and memory access costs of physical operators in order to assess alternative plans.

With hierarchy operators the true costs do not only depend on the chosen algorithm but also on the type of hierarchy index. While our index interface offers certain asymptotic complexity guarantees, the observable runtime costs of the primitives may differ heavily between different indexing scheme. For example, PPPL can answer most primitives straight from the label and makes node() a no-op, whereas other schemes deal with variable-length labels and complex auxiliary data structures. To account for these differences, one could enhance the cost model by (statically) assigning an appropriate CPU and memory access cost to each index primitive. The costs of a particular algorithm would then be computed in terms of the input cardinalities and the index primitives it uses. This is another worthwhile topic for future work.

Plan Generation and Rewriting. In § 4.1.4 and § 5.1.2 we already explored most of the relevant equivalences and properties of our hierarchy functions and predicates. In future work, it will be interesting to examine further *rewrite rules on plan level* regarding the different physical operators, like the rewrites between binary and unary structural grouping we discussed in § 5.1.7 as an example. Such rewrites are not trivial to perform, but they allow the engine to leverage the available physical operators as effectively as possible and thus can enable high performance gains.

Another challenge is to deal with hierarchical sorting effectively. Queries on hierarchies often request the output to be sorted. Most of our operators also require sorted inputs in preorder or postorder, and some of them retain certain orders in the output. Once the data has been sorted in the plan, subsequent operators may therefore benefit as well. The query optimizer should leverage these properties by employing explicit Sort operations using the is-before-pre() and is-before-post() primitives; where possible, Hierarchy Index Scan operators on the base table to establish the desired order in the first place; chains of order-preserving operators to retain the order, once established; and in particular, Hierarchy Rearrange when a conversion from preorder to postorder, or vice versa, is necessary. Thus, techniques such as maintaining *interesting orders* for query plans [94] become particularly relevant for queries on hierarchies.

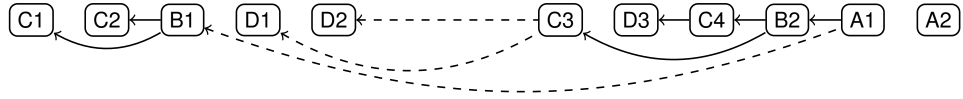
Enhanced Index Scans. Hierarchy Index Scan (§ 5.2.1) is a critical operator: it provides the access path to the hierarchical table and also serves as the basis of Hierarchy Index Join. It therefore seems worthwhile to further enhance its functionality. In § 5.2.1 we already outlined an approach to handle certain ϕ filters *implicitly* by delimiting the scan range upfront. This idea can be extended to further types of predicates, in particular axis checks where $\phi = (H.\text{axis}(\nu, \nu_0) \in A)$ for a constant ν_0 and a set of axes A, and arbitrary disjunctions thereof. However, many common predicates such as $d_1 \leq H.\text{depth}(\nu) \leq d_2$ cannot be handled by simply adjusting the overall scan range. For such predicates, significant speedups might still be achieved via fine-granular skipping over index ranges that *cannot* possibly fulfill the predicate, in the spirit of Staircase Join [41]. Further potential could be unlocked by integrating HIS more deeply with the underlying indexing scheme. For example, a hypothetical index maintaining lookup tables of the roots and leaves could handle predicates such as $H.\text{is-leaf}(\nu)$ and $H.\text{is-root}(\nu)$ by simply traversing over the available tables.

A related topic is about evaluating certain expressions of $\mathcal{L}$ *incrementally* during the scan in order to avoid explicit per-node index calls (as the basic Map operator would do). The idea is to call the index primitive e_i only once to obtain its initial value $e_i(c_1)$ at the begin of the scan range, then derive subsequent values from the previous value on each pre or post traversal step performed by the scan. Functions that can in principle be computed incrementally are pre-rank(), post-rank(), depth(), is-root(), height(), and—by peeking at the node ahead—is-leaf(). This technique would make the evaluation of such functions a matter of a few simple arithmetics per scanned node—even for dynamic indexes where they would otherwise involve costly algorithms.

Parallel Algorithms. While our operators are generally efficient in terms of both asymptotic complexities and "constant factors," they might be confronted with huge data sets especially in analytic applications. Parallelizing the algorithms is therefore an interesting line of future work. Consider the hierarchy join operation as an example. A helpful property of (inner) joins is that one input side can in principle be chunked and freely processed by parallel workers. A simple approach for a parallel Hierarchy Merge Join (and similarly, HIJ) would thus be to materialize the right input e_2 upfront as usual, then partition the left input e_1 and run the single-threaded algorithm $\mathrm{HMJ}(e_1', e_2)$ on each sorted partition e_1' against e_2. If the output order is irrelevant, HMJ (unlike HMGJ) does not strictly need e_1 to be *globally* sorted; each worker can sort its partition by ν_1 individually. If a consolidated output table must be produced, the individual outputs can simply be concatenated.

Note that for each e_1' partition the whole right input e_2 is potentially relevant. Scanning e_2 multiple times will significantly increase the total amount of work performed (e. g., in terms of is-before() calls) over the single-threaded case. The major challenge therefore is to optimally divide the work in order to maximize the gains from concurrent processing. Splitting the (left) input into equal-sized chunks is not sufficient, as the *actual* amount of work on the chunks may differ greatly due to a kind of "skew" that is inherent to many hierarchy operations. Consider a basic join $e_1 \bowtie e_2$ on the descendant axis: Input tuples from e_1 whose associated nodes are close to the root will match with many tuples from e_2, whereas leaf nodes may not find any join partners at all. When we divide e_1 into partitions e_1^i, a desirable goal is to balance the actual amounts of work $\mathcal{O}(|e_1^i| + |e_1^i \bowtie e_2|)$ as far as possible. To achieve a reasonable distribution, it may well be worthwhile to perform a first quick pass through the data to assess the skewedness for the purpose of partitioning. Alternatively, work stealing schemes such as the morsel-driven approach of [57] can elegantly circumvent many of the issues.

Besides the challenging problem of partitioning, the situation is even more complicated for HSF, HMGJ, HG, and HR, where each result tuple can have data dependencies to the previously processed nodes. A worker on partition j may depend on results from partition $j-1$ to be able to complete its work. Consider the bottom-up variant of Hierarchical Grouping, which is designed to reuse partial results from previous tuples. Suppose we are running HG against our familiar example hierarchy, assuming it has already been (globally) sorted in postorder and split into two halves as follows:



Two workers could work on the two chunks in parallel to a certain degree. For example, worker 1 can fold nodes C1 and C2 into B1, and worker 2 can fold D3 into C4. However, due to the data dependencies indicated by the dashed edges, both are unable to complete their work: Worker 1 cannot output B1, D1, or D2, because they are covered by upcoming nodes, and worker 2 cannot process C3 before D1 and D2 are ready. The workers can, however, easily *detect* these data dependencies: In the bottom-up HG case, worker j simply needs to check whether the last node of partition $j-1$ (D2 in our example) is a descendant of its currently processed node. A promising approach for stack-based algorithms like HG would thus be to let each worker run a modified HG algorithm on its partition, which processes all nodes that can be folded as usual, but detects and marks nodes that are on either end of a data dependency. Each worker would then end up with a stack of "incomplete" nodes that need further processing. For our example this would look as follows:

$$S^{(1)} = \langle \{\text{C1}, \text{C2}, \text{B1}\}, \{\text{D1}\}, \{\text{D2}\} \rangle \qquad S^{(2)} = \langle \text{C3}, \{\text{D3}, \text{C4}\}, \text{B2}, \text{A1}, \{\text{A2}\} \rangle$$

Here, sets of nodes indicate partially folded subtrees. These results cannot be yielded to the parent operator yet, as the dotted nodes from the second partition depend on them. Worker 2 just marks these dotted nodes on the stack without processing them. After both workers finish, a subsequent single-threaded merge step needs to pick up their unfinished stacks, finish the processing, and yield the actual output tuples.

In sane cases, the workers can be expected to complete a decent amount of work in parallel before the non-parallel merge step takes over: Assuming a regular hierarchy structure, perfectly equal-sized partitions, and only distinct nodes in the input, the size of the stacks is proportional to the hierarchy height h. Thus, the number of elements to merge is bounded by the number of partitions times h. Unfortunately, one can as well construct insane hierarchy structures for which parallelization is inherently impossible. A practical solution would have to detect such cases and fall back to non-parallel processing. These ideas could be applied to binary operators such as HMJ as well. Further research is needed to develop robust algorithms, study their properties, and assess their benefits against the sequential algorithms.

6 Experimental Evaluation

In this chapter we demonstrate the feasibility of our proposed framework for hierarchical data by means of an experimental evaluation, based on a prototypical execution engine that implements its key components. We conduct a series of experiments with synthetic and real-world data, covering both small and extremely large datasets in order to gain insights into caching effects and scalability. Our experiments in particular comprise the query primitives of the hierarchy index interface described in § 4.1.2, the bulk-loading algorithms of § 4.6, and the physical algebra operators of § 5.2. With respect to indexing, we concentrate on those hierarchy indexing schemes we recommend as a default choice in § 4, which includes a member of our own family of indexing schemes, the BO-tree. Where feasible, we compare our framework to state-of-the-art alternative approaches as well as common ad-hoc solutions, such as recursive common table expressions operating on adjacency lists.

6.1 Evaluation Platform

The platform for our experiments is a standalone single-threaded execution engine written in C++. It allows us to hand-craft query plans based on a push-based physical algebra as described in the previous chapter. All hierarchy operators we discussed in § 5.2 by design fit into this execution model. In addition to these operators, we implemented all SQL data types, as well as simple but reasonably efficient text-book algorithms of the essential operators of relational algebra. The base data is stored column-wise without further compression (nor dictionary encoding).

Through careful use of C++ templates, no virtual function calls are necessary in the hot paths and heavy inlining and low-level optimizations can be performed by the compiler. GCC 5.2.1 with `-O3` is able to translate query plans constructed from the physical algebra into efficient machine code, where no operator boundaries are apparent within the code fragments representing the different pipelines. Thus, the friction losses due to the physical algebra abstractions are minimal, and the resulting code is comparable in

Results from experiments based on our research prototype were previously published in [8], [31], [9], [32], [33], and [10]. In particular, experiments § 6.3.1, § 6.3.4, and § 6.3.7 were partially covered in [9]; experiments § 6.3.2 and § 6.3.3 were also covered in [32] and [33]; and experiments § 6.3.5, § 6.3.6, and § 6.3.8 were also covered in [10]. While this chapter does not include *all* of the previously published results, it provides an extended discussion of those experiments covered, and updates the reported numbers to reflect the most recent state of our prototype. Furthermore, all measurements are obtained from a single machine, which differs in some of our previous publications. However, these differences do not affect any of the conclusions we draw from our evaluation.

quality to what hand-written code would emit. According to a few micro-benchmarks we conducted it is also in the same order of magnitude as the single-threaded performance of modern engines such as HANA Vora [91]. This renders our execution engine a simple yet flexible and meaningful experimentation platform for evaluating our framework and in particular our proposed physical algebra operators.

Our test machine runs Ubuntu 15.10 and has two Intel Xeon X5650 CPUs at 2.67 GHz (6 cores, 2 hyperthreads each), 12 MB L3 cache, and 24 GB RAM. Our experiments are generally single-threaded and ensure that all data fits into RAM.

6.2 Test Data

For several of the experiments we use a hierarchical table HT with a schema similar to the one in Figure 2.3 (p. 17):

```
CREATE TABLE HT (
   Node NODE PRIMARY KEY,      -- size is index-dependent
   ID CHAR(8) UNIQUE,          -- 8 bytes
   PID CHAR(8),                -- 8 bytes
   Weight TINYINT              -- 1 byte
   Payload BINARY(p)           -- p bytes
)
```

Each tuple has a primary key Node, a unique ID and a Weight randomly drawn from the small domain $[1, 100]$. The PID column stores, for each node, the ID of the parent row. We use it only in query plans based on RCTEs. The table is stored column-wise and clustered by Node in preorder. There is a unique hash index on ID.

In the experiments we assess the performance on varying *hierarchy sizes* by varying the table size $|\mathrm{HT}|$ exponentially from 10^3 to 10^7 to also cover loads that by far exceed processor cache capacity. Already at $|\mathrm{HT}| = 10^6$ all indexes exceed L3 cache, yielding "worst case" uncached results. Regarding the *hierarchy structure*, we use the following test hierarchies:

- BOM $\langle N \rangle$: The structure of this hierarchy models a real-world bill of materials table. It is derived from a materials planning dataset of a large SAP customer. The original size is in the range of 10^5 nodes, and the average height of the individual trees is 10.3. The overall shape of the hierarchy is fairly regular and has no particularly challenging distortions. To expand the size to up to $N = 10^7$ nodes, we remove or replicate trees in such a way that the essential structural properties of the original hierarchy, especially the overall height, are preserved.

- BOM $\langle N, s \rangle$: For experiments involving subtrees of a specific size, we derive a family of hierarchies BOM $\langle N, s \rangle$ containing N nodes like BOM $\langle N \rangle$, but with a different shape: All children of the super-root $\top$ are trees of size s. We obtain each tree by choosing a random subtree of size at least s from BOM $\langle N \rangle$ and then removing random nodes to downsize it to s. Using BOM $\langle N, s \rangle$, we are able to easily pick random subtrees of size s among the children of $\top$.

- Regular $\langle N, k, s \rangle$: To be able to control and vary the size, depth, and "width" of the overall hierarchy while maintaining a homogeneous structure, we use an artificially generated hierarchy structure in some experiments. Regular $\langle N, k, s \rangle$ is a forest of N nodes in total, where each inner node is assigned exactly k children. Each tree in the forest is limited to s nodes in total. This way increasing N results in further trees of size s being added, but does not affect the total height of the forest. The height is controlled via k.

Regarding the *hierarchy index*, we cover the main indexing schemes we suggest in § 4.3 as well as a few additional alternatives of interest:

- PPPL: The simple Pre/Post/Parent/Level labeling scheme, which we fully describe in § 4.4. PPPL does not support any of the update primitives and thus is feasible only for read-mostly analytic scenarios, where the hierarchy is loaded once and changed rarely, if ever. As all query primitives boil down to very cheap $\mathcal{O}(1)$ arithmetics directly on the cache-resident `Label` objects, this is as fast as a hierarchy index can get. Measurements with PPPL thus indicate the upper performance bounds.

- BO-Tree: The BO-Tree indexing scheme outlined in § 4.5. We use a configuration with *mixed* block sizes and *gap* back-links, which we recommend as a good trade-off following our study in [33]. The BO-Tree indexing scheme is highly dynamic and at the same time robust against unfavorable "skewed" update patterns. This makes it a good all-round fit for dynamic OLTP scenarios. However, its merits come at a cost: Due to its elaborate structure—where each `Label` has links to two entries in the B^+-tree structure—the query primitives become computationally non-trivial $\mathcal{O}(\log |HT|)$ operations, and accessing the index blocks incurs additional cache misses. Comparing the measures of BO-Tree to those of PPPL thus gives us a good hint of the overhead to expect from a sophisticated dynamic index structure.

- Ordpath: We also implemented and measured the Ordpath labeling scheme as described in [79] (see also § 4.2.4). Here a `Label` stores a binary-encoded root path. The labels (`NODE` column) are indexed by a standard B-tree. Path-based schemes such as Ordpath generally support dynamic OLTP scenarios well, although their query primitives are slowed down by the variable-size labels, and, in comparison to BO-Tree, they are somewhat less robust and less flexible regarding complex updates (cf. [33]). We nevertheless include the measurements due to the popularity of path-based labeling schemes in commercial systems.

- Adjacency is an indexing scheme implementation which emulates the trivial adjacency list scheme. The `Label` object stores a pair (ID, PID) of integers. There is a (unique) hash index on ID and another (non-unique) hash index on PID. This scheme is very widespread in practice due to its simplicity. However, it does not efficiently support the query primitives nor the complex update primitives of

a full-blown hierarchy indexing scheme. Nevertheless, we implemented the index interface as far as possible, and include the numbers as an indication of the performance of RDBMS which do not adopt indexed hierarchical tables.

- DeltaNI: We also include our DeltaNI indexing scheme [31] as another example of a sophisticated index-based scheme, although it has not been a focus of this thesis. As a versioned index that is able to handle time-travel queries, it unsurprisingly comes with significant overhead in comparison to the other contenders. However, as mentioned in §4.2.5, DeltaNI can be useful in scenarios featuring temporal hierarchies.

This selection includes two proper "dynamic" indexing schemes. As §4.2 shows, there is actually a much wider range of dynamic schemes to choose from. Each of them will of course exhibit more or less different characteristics. For example, by replacing the BO-Tree by the O-List and tweaking the back-link representation and the block sizes, query performance could be further boosted. An O-List with a sufficiently large block size outperforms BO-Tree in queries by roughly 50%, although it also becomes less robust in dealing with skewed insertions and relocations of large subtrees and ranges. That said, a systematic comparison of dynamic indexing schemes is not our focus in this chapter. For more detailed insights we refer to our study in [32].

6.3 Experiments

In the following subsections we present the individual experiments we conducted. Each of these experiment focuses on a particular aspect of our framework.

6.3.1 Hierarchy Derivation

As a starting point, we evaluate the performance of building a hierarchy from scratch based on a table in the adjacency list format. This demonstrates the performance of the bulk-building process we described in §4.6.

Setup. To obtain the source data for bulk-building, we create a table Source in the adjacency list format. We assess the performance on varying input table sizes by scaling Source from $N = 10^3$ to 10^7. The table contains an ID primary key column and a PID column referencing the superordinate part. Both are of type `INTEGER`. The ID column is populated with generated IDs, and the PID column is populated so as to reflect the structure of our BOM $\langle N \rangle$ hierarchy. Additionally there is a 16 byte Payload field, which models data attached to the nodes that is to be included in the hierarchical table. Deriving a hierarchy from Source is expressed as follows using our SQL extensions:

```
SELECT Node, ID, Payload
  FROM HIERARCHY (USING Source JOIN PARENT p ON PID = p.ID SEARCH BY ID SET Node)
```

We assume table Source to be already materialized in memory, so that Step 1 of the process described in §4.6.3 (see p. 97) is skipped. For the initial outer self-join (Step 2.1

in §4.6.4, p. 98) we use a hash-based algorithm. Step 2.2 involves sorting by ID, for which we use a (single-threaded) standard library algorithm. In this case there is no explicit `START WHERE` clause that would need to be handled (Step 2.3).

For purposes of comparison, we measure the time of a recursive common table expression over Source using semi-naïve evaluation. This RCTE "discovers" the hierarchy structure in the source table through iterative joins over the ID–PID association:

```
WITH RECURSIVE RCTE (ID, Payload) AS (
   SELECT ID, Payload FROM Source WHERE PID IS NULL
    UNION ALL
   SELECT v.ID, v.Payload
     FROM Source v JOIN RCTE u ON v.PID = u.ID
)
SELECT * FROM RCTE
```

Such a recursive join could be used to implement the alternative *iterative* approach to the adjacency list transformation, as outlined in §4.6.5. Note that the measured RCTE does not perform duplicate elimination and thus cannot detect and eliminate cycles. Adding corresponding logic would make it considerably slower. Note also that ID and PID are processing-friendly `INTEGER` columns rather than, for example, `VARCHAR` columns which one often encounters in practice; this also benefits the iterative approach. The join algorithm we use for the RCTE is hash-based, just as for the `HIERARCHY()` statement.

Observations. Figure 6.1 displays the measurements. The shown times comprise all steps 1–4 of evaluating the `HIERARCHY()` statement. The right-most bar shows the time of the RCTE as an approximation of the mentioned iterative bulk-building approach.

The differences between the indexing schemes are not surprising. The static PPPL indexing scheme can be constructed with extremely low overhead (see the algorithm in §4.6.2). The overhead for BO-Tree is also quite moderate due to the efficient bulk construction algorithm of the underlying B^+-tree. DeltaNI takes around twice as long for constructing its non-trivial index structure based on binary trees, which are not particularly memory-efficient. Ordpath involves a comparatively costly manipulations of its variable-length strings during B^+-tree construction.

Most importantly, we see that—regardless of the chosen indexing scheme—bulk-loading a hierarchy index is considerably faster than the RCTE, especially so for large and deep hierarchies. The main reason for this is that the RCTE performs repeated (iterative) joins, whereas bulk-loading involves only a *single* self-join on the (complete) source table. The runtime of the RCTE is therefore proportional to the height of the loaded hierarchy (which in the case of the BOM $\langle N \rangle$ hierarchy is 10.3). Therefore, in this case—where no `START WHERE` clause was given—the iterative approach can be considered inferior to the non-recursive approach of evaluating a `HIERARCHY()` statement.

In Figure 6.2 the individual parts of the process are broken down further for the PPPL indexing scheme. Note that only steps 3 and 4 are index-specific (and as such the reason for the performance differences in Figure 6.1). Due to the minimal overhead of PPPL, the measurements of Step 3 essentially indicate the best achievable performance of the generic build() algorithm. The breakdown unveils the most expensive steps

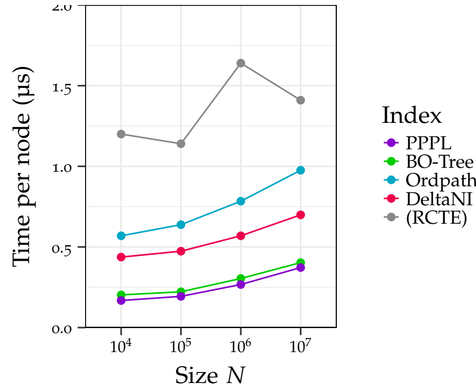


Figure 6.1: Experimental results for § 6.3.1.

Step		$N = 10^4$	10^5	10^6	10^7
1–2.1	Join	624 µs	5.6 ms	73.2 ms	1129 ms
2.2–2.4	Sort	654 µs	9.1 ms	119.2 ms	1234 ms
2.5	Edge List Transformation	136 µs	2.6 ms	69.5 ms	962 ms
3	build()	347 µs	3.9 ms	46.7 ms	390 ms
4	Populate `NODE` column	172 µs	0.3 ms	1 ms	10 ms
1–4	`HIERARCHY()` Total	1933 µs	21.5 ms	310 ms	3725 ms

Figure 6.2: Experimental results for § 6.3.1 — Breakdown of `HIERARCHY()` steps.

of the process: the initial hash-based self join (Step 2.1) and the subsequent sorting (Step 2.2). The actual Edge List Transformation and the build() algorithm are both cheaper than the join. Note, however, that in commercial RDBMS the join and sort operations can be expected to be further optimized and even parallelized. But even without further optimizations, we can safely conclude from the absolute numbers that the bulk-building process we propose in § 4.6.3 can deliver satisfying performance for even very large input sizes.

Memory Utilization. The memory utilization of the different indexing schemes is another consideration to make when choosing a suitable scheme. The following table compares the memory utilization for the case $|\text{HT}| = 10^7$ in bytes per hierarchy node:

	PPPL	BO-Tree	Ordpath	DeltaNI	Adjacency
clean	48.1	50.6	27.0	90.6	81.4
dirty	—	66.1	33.9	211.2	81.4

The first row “clean” shows the memory consumption immediately after bulk building. These sizes include the `NODE` column and the auxiliary data structure, but not the other columns of HT. The space required by our configuration of BO-Tree is comparatively high due to the employed *gap* back-links, which add 8 bytes per node: 2 bytes per key in the index entry and in the label, times two because each node has a lower and an upper

bound. The size could be reduced by modifying the configuration; for example, 1-byte gap keys could be used, or *pos* back-links, which require only 2 extra bytes. The BO-Tree block size B is also a factor: smaller blocks incur a certain (small) overhead over larger ones. Ordpath is remarkably compact, as it 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. The sizes for DeltaNI are constantly large, since its auxiliary data structure is based on space-costly binary trees with parent pointers. The high memory requirements of Adjacency are mainly due to its two hash indexes.

Another issue concerning dynamic indexes is that the memory utilization, and consequently the performance, degrades slightly when performing update operations. The "dirty" row in the table shows what the memory utilization would be if the identical hierarchy structure had been constructed via N individual inserts at random positions. (Note this is not applicable to PPPL.) Especially DeltaNI is affected by this issue. Particularly unfavorable "skewed" insertions can blow up the sizes of certain susceptible indexing schemes (in particular, containment-based variable-length labeling schemes) even more dramatically. Refer to [33] for more details on these effects.

6.3.2 Index Primitives for Queries

The goal of this experiment is to gain insights into the performance of different query primitives for the indexing schemes under consideration. These primitives are the building blocks for the algorithms we evaluate in the remainder, and as such influence their performance significantly. Since our focus is on query performance, we preclude update primitives. For an in-depth study that focuses on updates and compares various dynamic indexing schemes, we again refer the reader to our publication [33].

Setup. For this experiment our table HT is populated again with the BOM $\langle N \rangle$ hierarchy of sizes 10^4 to 10^7. We assess the query primitives node(), is-before-pre(), is-before-post(), axis(), is-parent(), depth(), and is-leaf(). They are invoked repeatedly with randomly selected `Node` objects of the hierarchy—`Label` objects in case of node(). We assume that the `Node` objects are already available, so the measured times of the primitives other than node() do *not* include the time of obtaining them first via node().

The measurements of the node() primitive, in particular, indicate the time it takes to retrieve a `Node` handle of the index structure given a `Label` object from the `NODE` column. For BO-Tree, node() essentially executes entry(l) using the location strategy. For PPPL, it involves a lookup in the pre-to-rowid[] index to make the corresponding row ID available for fast access. For Ordpath, it essentially performs a B-tree search for the label. For DeltaNI, node() involves a non-trivial process to "warp" the nested intervals bounds of the node to the most recent point in time using the auxiliary data structure, which consists of a pair of binary trees (see [31]).

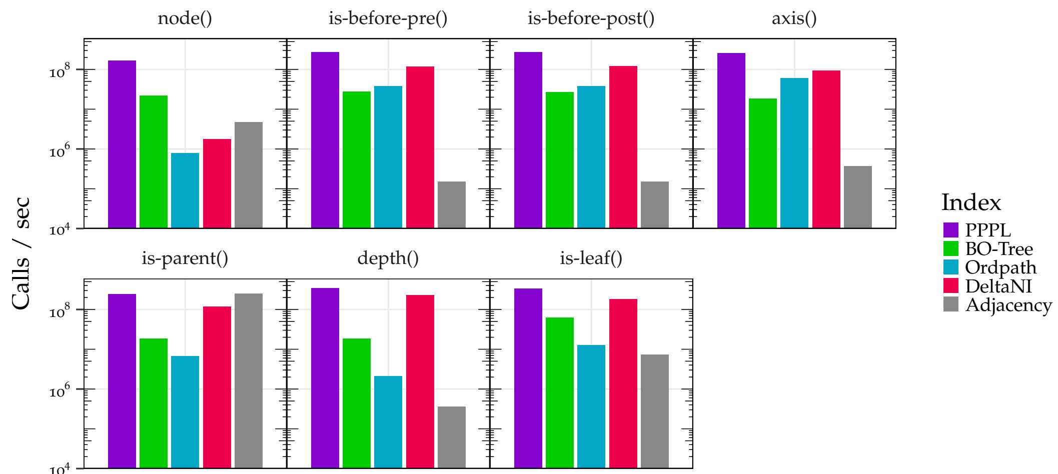


Figure 6.3: Experimental results for § 6.3.2.

Observations. Figure 6.3 displays the measurements for $|\mathrm{HT}| = 10^7$. To further demonstrate the accompanying caching effects, Figure 6.4 shows the average number of cache misses per operation (but omits some of the primitives for brevity).

We first observe that the naïve Adjacency indexing scheme fails to offer robust query performance, as is to be expected. While some queries such as is-parent() are fast, other important queries such as axis() and depth() are unacceptably slow. Also unsurprisingly, the static PPPL indexing scheme is hard to beat in terms of query performance. Ordpath also performs quite well, although it is outperformed by BO-Tree in some cases. The heavy-weight DeltaNI appears to perform astonishingly well on most queries, but this is only the case because all the complexity is hidden in node(), which DeltaNI must perform every time before it executes a query primitive.

Considering the node() primitive in particular, BO-Tree and order indexes in general benefit a lot from their fast back-link mechanism. This makes node() an $\mathcal{O}(1)$ operation and in practice involves around 2 cache misses. In contrast, labeling schemes such as Ordpath (but not PPPL) in general have to perform more costly $\mathcal{O}(\log N)$ B-tree key searches, which involve over 10 cache misses per lookup.

PPPL and Ordpath (and labeling schemes in general) can answer certain query primitives by touching only their `Label` objects. As the `Label` objects are readily loaded in cache, they show less than 0.5 cache misses for these primitives. Of course, the actual spent CPU cycles are much higher for Ordpath, which performs non-trivial operations on the binary-encoded ordpaths. This is most strongly visible with is-parent() and depth(), where it always compares the complete path strings.

In contrast to PPPL and Ordpath, as a block-based order index, BO-Tree touches the auxiliary data structure to evaluate any of the index primitives, and thus suffers at least 1 extra cache miss per invocation (around 1–3 cache misses according to Figure 6.4). This effect is particularly pronounced in this experiment. As we select the argument nodes randomly and thus also access the auxiliary data structure at random

	node()	is-before-pre()	axis()	is-parent()	depth()	is-leaf()
PPPL	1.07	0.24	0.21	0.23	0.15	0.15
BO-Tree	1.80	1.74	2.31	2.37	1.91	0.86
Ordpath	9.75	0.30	0.23	0.28	0.18	2.93
DeltaNI	13.77	1.35	0.62	1.11	0.66	0.22
Adjacency	6.74	70.52	33.59	0.60	33.67	2.47

Figure 6.4: Experimental results for § 6.3.2 — Average number of cache misses per call.

positions, the relevant index blocks will rarely be in cache. In this regard, this experiment is a worst-case scenario for BO-Tree in comparison to the labeling schemes. For query primitives where all indexing schemes inherently need to access their auxiliary index structure, labeling schemes show more cache misses and BO-Tree becomes more competitive. The is-leaf() primitive, for instance, requires a B-tree index access for labeling schemes (except for PPPL). Another example are traversal primitives, which we examine in the next experiment.

To assess the scalability of the indexing schemes, we also conduct the measurements on smaller hierarchies of size 10^6, 10^5, and 10^4. Figure 6.5 shows the results. (The query primitives we omit in this figure show a comparable behavior.) Regarding is-before-pre() and depth(), indexing schemes accessing the auxiliary index structure suffer between 10^5 and 10^6, where L3 cache size is reached. This is due to the mentioned extra cache misses, as these primitives are invoked with randomly chosen node arguments. The drop for node() is comparable for all indexing schemes, as they all need to access the auxiliary index structure. However, BO-Tree slows down only moderately between 10^6 and 10^7 nodes due to its $\mathcal{O}(1)$ node() algorithm, while other indexes drop further.

Apart from these details, all considered indexing schemes scale quite well on almost all query primitives, and therefore can be used for even very large hierarchies.

6.3.3 Hierarchy Traversal

Our previous experiment does not exercise the traversal primitives pre-next() and post-next(), as these primitives are never called in isolation in a query. Their main use is through Hierarchy Index Scan (HIS) or Hierarchy Index Join (HIJ) in order to navigate over the hierarchy.

Setup. In this experiment we use the same setup as in § 6.3.2. We execute the following query Q3.1:

```
SELECT ROWID() FROM HT ORDER BY PRE_RANK(Node)
```

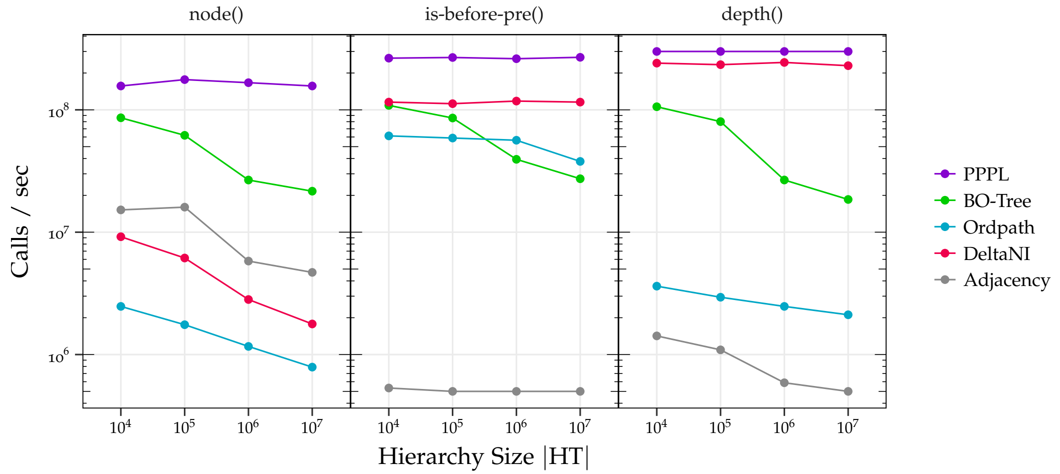


Figure 6.5: Experimental results for § 6.3.2 — Scalability.

The query plan consists of a single full Hierarchy Index Scan, which uses pre-next() to move over the hierarchy in preorder. It enumerates the row IDs of the nodes, but does *not* materialize them. No column of the table HT is touched except for (potentially) the NODE column. We do not report separate measurements for postorder, since traversal via post-next() works fully analogously for all indexing schemes under consideration.

Besides the performance of HIS, we measure another example query Q3.2, which uses a Hierarchy Index Join to self-join the hierarchical table HT on the descendant axis and additionally evaluates the depth() of each descendant:

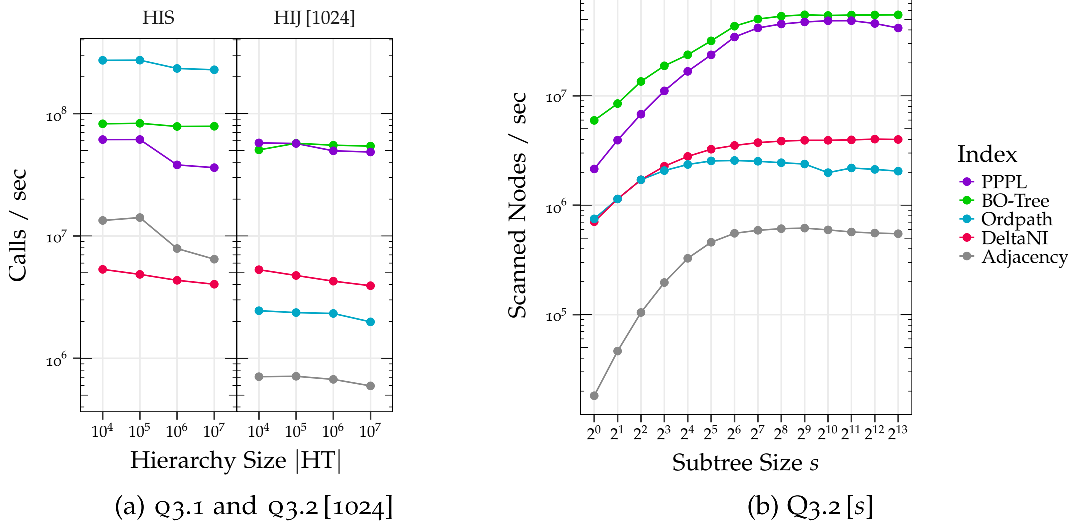
```
SELECT u.Node, v.Node, DEPTH(v.Node)
  FROM HT AS u JOIN HT AS v ON IS_DESCENDANT(v.Node, u.Node)
 WHERE IS_ROOT(u.Node)
 ORDER BY PRE_RANK(u.Node), PRE_RANK(v.Node)
```

The plan using Hierarchy Index Join is:

$e_0 \leftarrow$ Hierarchy Index Scan $[H; \nu_1; \text{pre}; \{\}; H.\text{is-root}(\nu_1)]\ ()$
$\theta(t_1, t_2) := (H.\text{axis}(t_2.\nu_2, t_1.\nu_1) = \text{descendant})$
$e_1 \leftarrow$ Hierarchy Index Join $[H; \nu_2; \theta; \ltimes; \text{pre}]\ (e_0)$
$e_2 \leftarrow$ Map $[d : H.\text{depth}(\nu_2)]\ (e_1)$

Generally, the performance of HIJ varies with the sizes of the subtrees that are scanned for each tuple from its (left) input. We therefore make these sizes variable: For s ranging from 2^0 to 2^{13}, we generate a BOM $\langle N, s\rangle$ hierarchy structure. Then we use the children of the super root $\top$ for the join input e_0. The first HIS in the plan therefore does not contribute to the measured execution times. As HIJ essentially executes a partial index scan for each input tuple to enumerate the s entries in the respective tree, we refer to these measurements by "Q3.2 [s]."

This instance of HIJ represents an important query pattern and thus can give us a hint of the performance we can expect of queries involving index scans in their plans. Note the depth() primitive is exercised in a different way than in § 6.3.2, where we measured the query primitives in isolation and called them on random individual nodes.



(a) Q3.1 and Q3.2 [1024]

(b) Q3.2 [s]

Figure 6.6: Experimental results for § 6.3.3.

Here, memory locality effects regarding the auxiliary data structure come into play due to the systematic traversal. This benefits in particular block-based indexes such as BO-Tree, since the relevant blocks will often be in cache already. depth() therefore becomes a comparatively cheap operation.

Observations. Figure 6.6 shows scalability measurements of Q3.1 and Q3.2 [1024] with varying hierarchy size $|HT|$, as well as measurements of Q3.2 [s] at $|HT| = 10^7$ with varying scan size s. The plotted measure is the number of scanned tuples—i. e., calls to pre-next()—per second.

Scans generally access the hierarchy index in a predictable pattern. Therefore, we intuitively expect the initial cursor() call of an index scan to be the most expensive individual operation, as it always incurs a cache miss. The subsequent traversal via repeated pre-next() calls is very prefetching-friendly and will thus often incur few further cache misses. This explains why all indexing schemes benefit from larger scans s in the graph to the right. We can further confirm this intuition by looking at the respective average number of cache misses per pre-next() call:

	Q3.1
PPPL	0.13
BO-Tree	0.04
Ordpath	0.09
DeltaNI	0.39
Adjacency	0.75

For PPPL, BO-Tree, and Ordpath there are virtually no cache misses during the scan.

When we look at the results for the pure scan Q3.1, Ordpath is fastest. This is not surprising, as it simply performs a scan over the B-tree which already arranges the

labels in preorder. BO-Tree performs a scan through its underlying B^+-tree-based data structure, which also benefits a lot from the mentioned locality effects. PPPL is slower than Ordpath and BO-Tree in this experiment, as it has to simultaneously access the Node column and its pre-to-rowid[] index. That said, its absolute performance is still very good. Adjacency is very slow, as its hash indexes have to be repeatedly accessed to determine the child nodes to be subsequently scanned. DeltaNI is even slower; it pays the price of a full-blown versioned indexing scheme.

When we look at the more complex query Q3.2 featuring the additional depth() call, things change noticeably. Ordpath suffers hard from its variable-length labels and the more expensive depth() primitive that involves counting the elements in the ordpaths. depth() becomes even more expensive for Adjacency, which has to perform traversals through the auxiliary data structure to determine the number of nodes on the respective root paths.

Regarding the scalability with respect to $|HT|$, there is a slight penalty when the hierarchy cannot fit into L3 cache any more. But in general, the performance drops less significantly in comparison to the analogous measurements of § 6.3.2. This is again due to the predictable patterns the data structures are accessed in, which particularly benefits the indexing schemes that are based on a block-based auxiliary data structure: BO-Tree as well as Ordpath with its B-tree of labels.

We conclude that both recommended indexing schemes PPPL and BO-Tree offer robust and high performance at raw index scans and at queries featuring scans as building blocks. Since HIS and HIJ are core operators of common query plans, outstanding query performance can be anticipated.

6.3.4 Hierarchy Joins

In this experiment we assess the performance of our join algorithms Hierarchy Merge Join (HMJ) and Hierarchy Index Join (HIJ).

Setup. For this experiment we again use the hierarchical table HT as described in § 6.2 and vary its size $|HT|$ from 10^3 to 10^6. Due to the large output sizes of the test queries we preclude $|HT| = 10^7$. The hierarchy structure is the generated forest structure Regular $\langle |HT|, k, s \rangle$, where each tree is given $s = 10^4$ nodes and each inner node exactly k children. To assess the influence of the hierarchy shape, we compare very deep trees ($k = 2$) trees to very shallow trees ($k = 32$). With $k = 2$ we get a total hierarchy height h of approximately 13.2, whereas for $k = 32$ we get $h \approx 3.6$. Since in a Regular hierarchy the majority of nodes is on the lowest levels, the average level of the nodes is 11.4 and 2.89, respectively.

We assess four queries. Query Q4.1 performs an inner self join on the *descendant* or *self* axes over the *complete* hierarchy and lists the ID pairs of the matching nodes:

```
SELECT t.ID, u.ID FROM HT t JOIN HT u ON IS_DESCENDANT_OR_SELF(u.Node, t.Node)
```

We assume the input table to be readily available in memory. To mask out the effects of accessing and sorting the required columns of HT, which is in column-oriented format,

we pre-process the input table as follows: First, we pre-materialize a row-oriented input table Inp with the columns ID and Node. Second, we use Map to convert the Node column (which contains `Label` objects) to a column ν of `Node` objects. This way we avoid having to insert node() calls into some of the assessed query plans, which would put them at a disadvantage against plans that don't require `Node` objects. Third, we order the table by Node in preorder as required by the respective query plans.

We use different alternative plans to compare HMJ, HIJ, Nested Loop Join (NLJ), and an RCTE-based solution. Each plan projects only the required t.ID and u.ID fields, and the time for result materialization is included in the measurements. The plans are:

(HMJ A) A plan based on HMJ A:

$$e_1 \leftarrow \text{Hierarchy Merge Join A}\,[H; t.\nu; u.\nu; \text{pre}; \{\text{self}, \text{descendant}\}; \ltimes]\,(\text{Inp}[t], \text{Inp}[u])$$

$$e_2 \leftarrow \text{Map}\,[t.\text{ID}, u.\text{ID}]\,(e_1)$$

(HMJ B) A plan based on HMJ B:

$$e_1 \leftarrow \text{Hierarchy Merge Join B}\,[H; u.\nu; t.\nu; \text{pre}; \{\text{ancestor}, \text{self}\}; \ltimes]\,(\text{Inp}[u], \text{Inp}[t])$$

$$e_2 \leftarrow \text{Map}\,[t.\text{ID}, u.\text{ID}]\,(e_1)$$

Note the HMJ-based plans are only possible because the table Inp is already sorted accordingly.

(HIJ-1) A plan based on HIJ:

$$e_1 \leftarrow \text{Hierarchy Index Join}\,[H; \nu_u; H.\text{axis}(\nu_u, t.\nu) \in \{\text{self}, \text{descendant}\}; \ltimes; \text{pre}]\,(\text{Inp}[t]);$$

$$e_2 \leftarrow \text{Map}\,[u : \text{HT}[H.\text{rowid}(\nu_u)]; t.\text{ID}, u.\text{ID}]\,(e_1)$$

(HIJ-2) An alternative plan based on HIJ with an inverted join direction:

$$e_1 \leftarrow \text{Hierarchy Index Join}\,[H; \nu_t; H.\text{axis}(\nu_t, u.\nu) \in \{\text{ancestor}, \text{self}\}; \ltimes; \text{pre}]\,(\text{Inp}[u]);$$

$$e_2 \leftarrow \text{Map}\,[t : \text{HT}[H.\text{rowid}(\nu_t)]; t.\text{ID}, u.\text{ID}]\,(e_1)$$

(NLJ) A plan based on NLJ:

$$e_1 \leftarrow \text{Nested Loop Join}\,[H.\text{axis}(u.\nu, t.\nu) \in \{\text{self}, \text{descendant}\}; \ltimes]\,(\text{Inp}[t], \text{Inp}[u]);$$

$$e_2 \leftarrow \text{Map}\,[t.\text{ID}, u.\text{ID}]\,(e_1)$$

(RCTE) An equivalent plan based on a semi-naïve least fixpoint operator. It works on the ID and PID columns of HT and mimics a solution based on SQL's recursive CTEs, under the assumption that efficient `IS_PARENT` joins are possible. The SQL definition of the RCTE is as follows:

```
WITH RECURSIVE RCTE (t_ID, u_ID) AS (
   SELECT ID, ID FROM HT
    UNION ALL
   SELECT a.t_ID, b.ID FROM RCTE t JOIN HT u ON u.PID = t.u_ID
) SELECT * FROM RCTE
```

The plan uses a hash-based algorithm to evaluate the `JOIN`.

Query Q4.1* assesses the overhead of large ID keys. The plans are the same as for Q4.1, but the ID and PID fields are of size 32 bytes instead of 8 bytes. This increases the sizes of the input table Inp and of the join result, but also the intermediate results—especially with the RCTE-based plan.

Query Q4.2 performs the same type of join as Q4.1, but with smaller, filtered left and right inputs. To this end we prepare two disjunct subsets LeftInp and RightInp of HT by filtering it by Weight in such way that a random 20% of the nodes is retained. We again pre-materialize these two tables in row format and order them by Node in preorder. The SQL definition of Q4.2 is:

```
SELECT t.ID, u.ID
  FROM LeftInp t JOIN RightInp u ON IS_DESCENDANT_OR_SELF(u.Node, t.Node)
```

The HMJ-based and NLJ-based plans are analogous to the plans of Q4.1, except that LeftInp and RightInp are used as inputs rather than two times Inp. The modified HIJ-based plans are as follows:

(HIJ-1) $e_1 \leftarrow \text{Hierarchy Index Join}\,[H; \nu_u; H.\text{axis}(\nu_u, t.\nu) \in \{\text{self}, \text{descendant}\}; \bowtie; \text{pre}]\,(\text{LeftInp}[t]);$

$e_2 \leftarrow \text{Map}\,[u : \text{HT}[H.\text{rowid}(\nu_u)]; t.\text{ID}, u.\text{ID}]\,(e_1)$

$e_3 \leftarrow \text{Join}\,[r.\text{ID} = u.\text{ID}; \rtimes]\,(\text{RightInp}[r], e_2)$

(HIJ-2) $e_1 \leftarrow \text{Hierarchy Index Join}\,[H; \nu_t; H.\text{axis}(\nu_t, u.\nu) \in \{\text{ancestor}, \text{self}\}; \bowtie; \text{pre}]\,(\text{RightInp}[u]);$

$e_2 \leftarrow \text{Map}\,[t : \text{HT}[H.\text{rowid}(\nu_t)]; t.\text{ID}, u.\text{ID}]\,(e_1)$

$e_3 \leftarrow \text{Join}\,[l.\text{ID} = t.\text{ID}; \rtimes]\,(\text{LeftInp}[l], e_2)$

The limitations of HIJ force us to add additional semi joins to filter for RightInp or LeftInp *after* performing the actual hierarchy joins. For these Join instances we use a hash-based algorithm. An analogous filter has to be added for the RCTE-based solution. The modified SQL definition of the RCTE for Q4.2 is:

```
WITH RECURSIVE RCTE (t_ID, u_ID) AS (
   SELECT ID, ID FROM LeftInp
    UNION ALL
   SELECT a.t_ID, b.ID FROM RCTE t JOIN HT u ON u.PID = t.u_ID
)
SELECT * FROM RCTE WHERE u_ID IN (SELECT ID FROM RightInp)
```

Finally, query Q4.2/semi is a variant of Q4.2 which features a left semi join:

```
SELECT t.ID FROM LeftInp t
 WHERE EXISTS ( SELECT * FROM RightInp u
                WHERE IS_DESCENDANT_OR_SELF(u.Node, t.Node) )
```

The HMJ-based and NLJ-based plans are analogous to the plans of Q4.2 except that $\iota = \ltimes$ and only t.ID is projected. The HIJ-based plans can *not* take advantage of the semi variants of HIJ, as the filtering for IDs in RightInp and LeftInp needs to happen *after* the actual join. Therefore, the only difference to the HIJ-based plans for Q4.2 is in the final projection after the semi join, which has to eliminate duplicates (`DISTINCT` semantics) using an order-based algorithm.

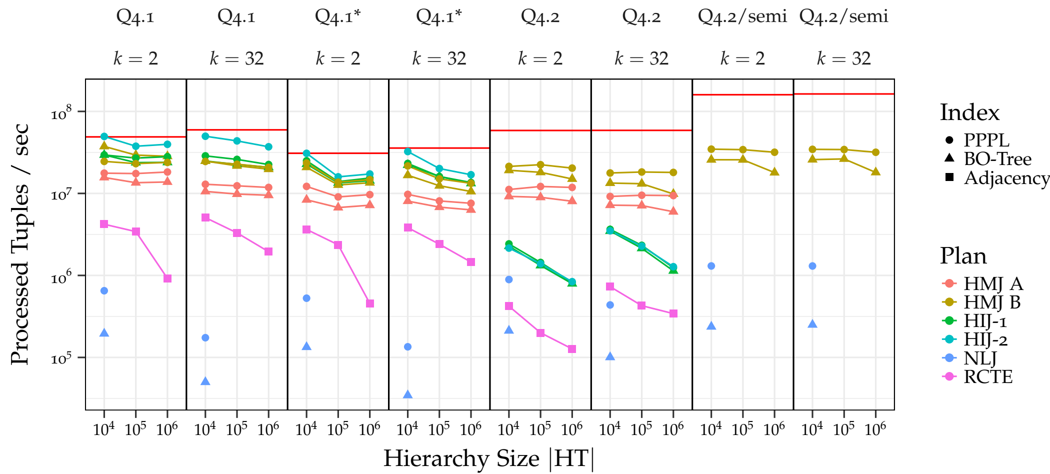


Figure 6.7: Experimental results for § 6.3.4.

Observations. Figure 6.7 shows the measured results. The y axis displays the execution time of the queries divided by the size of the output table; in other words, the number of joined tuples per second. We measured NLJ only for $|\mathrm{HT}| = 10^4$ due to its quadratically increasing runtimes. The output sizes are:

		Result Size		
		$\lvert\mathrm{HT}\rvert = 10^4$	$\lvert\mathrm{HT}\rvert = 10^5$	$\lvert\mathrm{HT}\rvert = 10^6$
Q4.1, Q4.1*	$k = 2$	113.644	1.136.440	11.364.400
	$k = 32$	28.912	289.120	2.891.200
Q4.2	$k = 2$	5.035	50.578	554.640
	$k = 32$	2.453	24.335	255.451
Q4.2/semi	$k \in \{2, 32\}$	1.921	18.860	190.114

The red lines indicate the speed of copying a precomputed result table tuple by tuple. This is a physical upper bound for the achievable performance.

Considering Q4.1 and Q4.1*, HIJ is generally fastest. It is followed closely by HMJ B (roughly 2–3 times slower), which in turn noticeably outperforms HMJ A (roughly 2 times slower). The RCTE-based solution is an order of magnitude below HIJ, and the NLJ-based plan is another order of magnitude below the RCTE.

When comparing the dynamic BO-Tree to the static PPPL index, we see almost no differences for the HIJ-based plans, as both indexes are very efficient at scans. With the HMJ-based plans we clearly see the effects of the more complex is-before-ς() implementations of a dynamic index. While they are almost for free with PPPL, the non-trivial implementations of BO-Tree incur noticeable CPU costs. The effect is strongest with the NLJ-based plans, as NLJ performs the most is-before checks. Note our BO-Tree in the experiments is freshly loaded. In practice, the internal structure of most dynamic indexes tends to degrade somewhat from incremental updates, as we saw in § 6.3.1 when comparing the index sizes. This will also have a slight effect on performance in practice. That said, the slow-down of BO-Tree is quite tolerable overall. Interestingly,

the $\mathcal{O}(\log |HT|)$ complexity of the index primitives is barely visible in the measurements. Theoretically, increasing the hierarchy size $|HT|$ should slow down BO-Tree further against PPPL; however, the figures are practically indifferent to $|HT|$. For the chosen block-based BO-Tree index, the logarithmic complexity does not matter much in practice. One reason for this is that the height of the BO-Tree grows extremely slowly, so the complexity can in practical terms be considered constant. Another reason is the favorable data locality in the ordered inputs: the nodes involved in is-before checks are usually close in terms of pre/post distance. Therefore, the relevant BO-Tree blocks will be in cache, and most checks can even be answered by considering only one leaf block hosting both nodes.

As the input sizes are equal to $|HT|$ (or 20% of $|HT|$ in case of Q4.2 and Q4.2*) and the output sizes are also largely proportional to $|HT|$, our chosen y axis allows us to get an indication of the scalability of the different query plans with respect to the hierarchy size. The figures confirm the linear asymptotic complexities of HMJ and HIJ, and the quadratic complexity of NLJ makes this plan infeasible for $|HT| = 10^5$ and higher. Still, HMJ and HIJ are slowed down slightly for higher hierarchy sizes, which can be attributed to the increased memory traffic. That slowdown is even higher for the RCTE, as that plan produces significantly bigger intermediate results. At $|HT| = 10^6$ and $k = 2$ there is a particularly pronounced slowdown, where the L3 cache seems to become ineffective. This also explains the differences we see when comparing Q4.1* to Q4.1. The inputs and outputs are identical in these queries; the only difference is the larger keys and thus the increased memory traffic. All the effects of increasing $|HT|$ are thus more pronounced in Q4.1*.

When comparing $k = 32$ against $k = 2$, one has to take into account that decreasing the hierarchy height also decreases the average number of join partners in our setting—and thus the size of the total join result. The smaller result is the reason for the slight speedup of $k = 32$ over $k = 2$. It also means the negative caching effects noted with $k = 2$ and higher hierarchy sizes are somewhat damped for $k = 32$. The RCTE-based plan gains a lot from the flatter hierarchy, as fewer iterations are needed.

Looking at Q4.2, we first notice that HIJ and RCTE are at a big disadvantage now, as they have to perform late filtering using an additional hash join. NLJ is speeded up significantly, as it has to deal with only 20% of the original input sizes and thus suffers less from its quadratic asymptotic complexity. The performance of HMJ remains (roughly) comparable. Note that both the input sizes and the join result sizes change over Q4.1; therefore, a direct comparison of the performance to Q4.1 is difficult.

Looking at Q4.2/semi, we see that both HMJ and NLJ become more effective in comparison to Q4.2 due to their specialized semi join variants. (We did not implement HIJ and RCTE against Q4.2/semi. In general, HIJ benefits in the left semi join case, but is inherently unable to handle right semi joins. The RCTE-based plan requires an additional a-posteriori semi join or selection for the filter, and thus would be slowed down even further. This is another inherent disadvantage of RCTEs.)

Overall, we can conclude that both HIJ and HMJ have their respective ideal scenarios where they deliver the best possible performance. Both easily outplay the suboptimal RCTE-based and NLJ-based alternatives by up to two orders of magnitude.

6.3.5 Hierarchical Windows

In this experiment we assess the bare performance of hierarchical windows.

Setup. The basic setup is the same as for § 6.3.4. In addition to HT we pre-materialize an input table Inp as described below. Then we run Statement IV from § 3.3.5 with various expressions from Figure 3.7 on Inp:

```
SELECT Node, expr FROM Inp
WINDOW td AS (HIERARCHIZE BY Node TOP DOWN),
       bu AS (HIERARCHIZE BY Node BOTTOM UP)
```

Queries Q5.1 to Q5.4 differ in which expression *expr* they evaluate:

- Queries Q5.1 and Q5.2 compute Expression 1a bottom up and top down, respectively, and represent non-recursive computations.

  `expr ≡ SUM(Value) OVER bu` (Q5.1)

  `expr ≡ SUM(Value) OVER td` (Q5.2)

- Query Q5.3 computes Expression 4c and represents a structurally recursive computation.

  `expr ≡ RECURSIVE DOUBLE (Value + SUM(Weight * x) OVER bu) AS x`

- Query Q5.4 computes a count/distinct aggregate bottom up and features a comparatively expensive duplicate elimination.

  `expr ≡ COUNT(DISTINCT Weight) OVER bu`

The aggregate state X of Q5.4 requires a data structure for duplicate elimination. We maintain the distinct Weight values in a local array that spills to a hash table on the heap once its fixed capacity of 128 values is exceeded.

For each of these queries we measure alternative plans. All plans work on the same input Inp, which associates all nodes of HT with values as the computation input. Inp is prepared a priori as follows: We select the contents of HT (thus, $|\text{Inp}| = |\text{HT}|$), add a randomly populated field Value of type `INT`, and run Map to obtain a column ν of `Node` objects from the `Label` objects of the Node field. Then we project (in row format) the required fields for the respective query: $[\nu, \text{Value}]$ for Q5.1/Q5.2, $[\nu, \text{Value}, \text{Weight}]$ for Q5.3, and $[\nu, \text{Weight}]$ for Q5.4. Finally we sort the data in either preorder or postorder as needed by the respective plan. The measurements thus show the bare performance of the respective operators without any pre- or post-processing—in particular, without sorting—but including the materialization of the query result. We compare the following alternative plans, where applicable:

(a) the straight translation into

Hierarchical Grouping $[H; \nu; \text{post}; \{\text{descendant}\}; x : \textit{expr}]$ (Inp); — for window `bu`

Hierarchical Grouping $[H; \nu; \text{pre}; \{\text{ancestor}\}; x : \textit{expr}]$ (Inp); — for window `td`

(b) the alternative

Hierarchy Merge Groupjoin $[H; t_1.\nu_1; t_2.\nu_2; \text{post}; \{\text{descendant}\}; x : \mathit{expr}]$ $(\text{Inp}[t_1], \text{Inp}[t_2])$

Hierarchy Merge Groupjoin $[H; t_1.\nu_1; t_2.\nu_2; \text{pre}; \{\text{ancestor}\}; x : \mathit{expr}]$ $(\text{Inp}[t_1], \text{Inp}[t_2])$;

(c) the equivalent HMJ-Group approach of § 5.2.7 with a preorder-based Hierarchy Merge Join (variant HMJ A);

(d) the equivalent Join-Group approach of § 5.2.7 with a Nested Loop Join.

Comparing Plan b to Plan a shows us the overhead of using $e \bowtie^{\Gamma} e$ instead of $\hat{\Gamma}(e)$ for a non-recursive computation. Plan c can be considered state of the art and a natural baseline for the comparison against our native $\hat{\Gamma}$ and $\bowtie^{\Gamma}$ algorithms, as explained in § 5.2.7. Plan d, the Join-Group approach, will often be the only option when an encoding such as PPPL is hand-implemented in an RDBMS without further engine support. Like in § 6.3.4, we furthermore consider two plans based on a semi-naïve least-fixpoint operator to mimic a RCTE-based solution using iterative IS_PARENT joins:

(e) Iterative uses iterative hierarchy merge joins on the *parent* axis to first compute all $<$ pairs bottom up (Q5.1) or top down (Q5.2), analogously to § 6.3.4. Then it performs the actual computation using sort-based grouping.

(f) Iterative* additionally applies "early grouping" within each iteration, an optimization inspired by [82]. This early grouping can also be sort-based, as the hierarchy merge join conveniently retains the input order.

This gives us a hint of the performance that can be expected from an exceptionally well-optimized RCTE or from a hand-crafted iterative stored procedure. We commonly see such procedures in real-world applications that are based on trivial adjacency list tables. However, plans e and f are no general solutions; they work in our setup only because *all* nodes from the hierarchical table HT are also present in Inp.

Note also that plans b to f work only for *non-recursive* computations. Thus, we can evaluate Query Q5.3 only with Plan a.

Observations. Figure 6.8 shows the execution times, normalized with respect to the number of processed elements |Inp|. As in § 6.3.4, the red line indicates the speed of copying a precomputed result table, which is approximately 37.6M per second.

In Q5.1–3 with PPPL, the Hierarchical Grouping operator is remarkably close to this bound (around 25.4M per second, or 67%). That non-recursive computations using HG (Q5.1a) are not slower than recursive ones (Q5.3a) comes at no surprise, since the algorithm is identical. For both HG and HMGJ, the top-down algorithms (Q5.2ab) are slightly slower than the bottom-up algorithms (Q5.1ab), as they cannot dismiss covered tuples as early and thus inherently issue more index calls. The duplicate elimination of Q5.4 is costly—both HG and HMGJ become roughly 3 to 4 times slower over the trivial arithmetics of Q5.1–3. When comparing $e \bowtie^{\Gamma} e$ to $\hat{\Gamma}(e)$ over all queries (Q5.1–4ab), we see the latter is on average around 32% faster. The overhead of binary grouping stems

from evaluating e twice (which in this case is a table scan) and from the extra index calls needed to associate e_1 and e_2 tuples.

The HMJ-Group approach (Plan c) is significantly slower than HMGJ, mostly in bottom-up Q5.1 (for instance, around 11 times slower at $k = 2$) but also in top-down Q5.2 (around 3.5 times at $k = 2$); the gap grows with the hierarchy height. HMJ-Group is somewhat handicapped at Q5.1, as the hierarchy merge join algorithm we use in the plan is preorder-based. As preorder is more natural to top-down computations, HMJ-Group performs noticeably better at Q5.2. Interestingly, the HMJ-Group-based Plan c is not slowed down as much as the others at Q5.4 versus Q5.1. Apparently, the intermediate join dominates the costs so that the subsequent processing-friendly sort-based grouping does not matter much. Correspondingly, the overhead over HMGJ is smaller at Q5.4, though still noticeable.

The iterative solutions are generally slow. Early aggregation helps much in the bottom-up case, where Iterative* even approaches HMJ-Group at $|\mathrm{HT}| = 10^6$. In the top-down case, however, early aggregation does not help to reduce the intermediate result sizes, as IS_PARENT is an $N : 1$ join; here, the (minor) savings over Iterative come from saved arithmetic operations by reusing results of previous iterations.

Regarding dynamic (BO-Tree) versus static (PPPL) indexes, the more complex is-before-ς() implementations of the former are again clearly noticeable, as in § 6.3.4. The slowdown is most visible with the top-down Q5.2, where more is-before checks are issued inherently.

As we also saw in § 6.3.4, increasing the hierarchy size |HT| does not significantly slow down BO-Tree in spite of the $\mathcal{O}(\log |\mathrm{HT}|)$ complexity of the index primitives. HMJ-Group and Iterative are much more sensitive to |HT| due to their growing intermediate results.

If we consider the hierarchy shape—deep $k = 2$ versus flat $k = 32$—we see that Iterative and Iterative* are very sensitive—unsurprisingly, as their time complexity is proportional to h—whereas HG and HMGJ are practically indifferent. The intermediate join result of HMJ-Group is somewhat proportional to h, so it is also affected to some extent (factor 2–3).

Note that the above experiments assess only $e_1 \ltimes\!\!\!\Gamma\, e_2$ where $e_1 = e_2$, that is, a unary hierarchical window setup. We also conducted measurements where $e_1 \neq e_2$ with varying $|e_1|$ and $|e_2|$ sizes. However, as we found the results to be fully in line with the complexity $\mathcal{O}(|e_1| + |e_2| + |e_1 \ltimes\!\!\!\Gamma\, e_2|)$ of HMGJ, we do not separately report them.

Overall, the results exhibit the robust linear space and time complexities of our operators for hierarchical windows, and again highlight their merits over conventional approaches. Furthermore, they confirm the known "groupjoin advantage" also for the hierarchical case—in line with the reports on hash-based equi-groupjoins of [74].

6.3.6 Hierarchical Sorting

In this experiment we compare plans based on Sort to plans based on Hierarchy Rearrange (HR). Recall that the HR operator consumes an input that is already sorted in preorder or postorder and employs a stack-based algorithm to convert it to the respec-

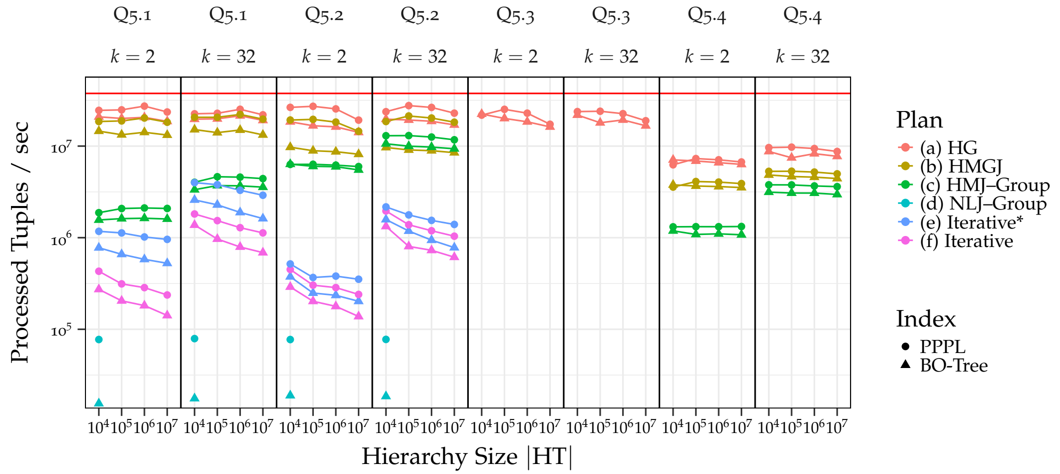


Figure 6.8: Experimental results for § 6.3.5.

tive other order. Its advantage over Sort is that it allows for pipelining to some degree. Here we investigate the potential of these "order-based" sorting algorithms.

Setup. Queries Q6.1 and Q6.2 work directly on the hierarchical base table and access only the Node field. They simply select all nodes and sort them in either postorder (Q6.1) or preorder (Q6.2):

```
SELECT Node, ROWID FROM HT ORDER BY [POST_RANK(Node)|PRE_RANK(Node)]
```

For each query we compare four different ways to do the sorting. Like in § 6.3.4 and § 6.3.5, we first do some preprocessing work on HT that is not included in the time measurements: We use Map to convert the Node column of HT to a column ν of corresponding `Node` objects. We project only the row ID and ν fields and materialize the result in a table Inp. For some of the plans we need a preorder-sorted copy Inp-pre of Inp and a postorder-sorted copy Inp-post, which we also prepare and materialize ahead. In the bottom-up case Q6.1, the four plans are:

(a) A simple $e_1 \leftarrow \text{Scan}(\text{Inp-post})$.

(b) Using Hierarchy Rearrange on the preorder-sorted copy of Inp:

$e_1 \leftarrow \text{Hierarchy Rearrange}\,[H; \nu; \text{pre} \rightarrow \text{post}]\,(\text{Inp-pre})$

(c) Performing a full sort of Inp:

$e_1 \leftarrow \text{Sort}\,[<]\,(\text{Inp})$ where $t_1 < t_2 := H.\text{is-before-post}(t_1.\nu, t_2.\nu)$

(d) Enumerating the HT tuples in the desired order using an index scan:

$e_1 \leftarrow \text{Hierarchy Index Scan}\,[H; \nu; \text{post}; \{\}; \text{true}]\,(\text{HT})$

In the top-down case Q6.2, the plans are identical, except that the preorder and postorder roles have to be swapped. Note that Plan d works only for these particular queries, as we are working directly on HT instead of an arbitrary input table.

Beyond Q6.1 and Q6.2, which allow us to assess the peak performance of the different sort methods, it is also interesting to see how these operators interact with other operators in a more "useful" query. To this end, queries Q6.3 and Q6.4 perform a hierarchical computation based on a bottom-up and a top-down window, respectively.

```
SELECT Node, SUM(Value) OVER w
  FROM Inp WINDOW w AS (HIERARCHIZE BY Node [BOTTOM DOWN|TOP DOWN])
```

These queries are closely comparable to Q5.1 and Q5.2 of § 6.3.5, and thus give us an indication of the additional costs when the input is *not* already in the required order. The Value field is generated upfront as described in § 6.3.5. The plans we compare are just as for Q6.1 and Q6.2, except that an additional Hierarchical Grouping operator is applied to the sorted output e_1, and in case of Plan d an additional Map operator is needed to bring in the Value field. The HG operator is identical in all plans for Q6.3:

$$\text{Hierarchical Grouping}\,[H; \nu; \text{post}; \{\text{descendant}\}; x : \text{sum(Value)}]\,(e_1)$$

In the top-down case Q6.4, the descendant axis becomes the ancestor axis, accordingly.

Observations. From the results in Figure 6.9, we first see that dynamic indexes suffer from their more expensive is-before() calls, which are heavily exercised for sorting. Unsurprisingly, this affects full sorting (Plan c) in particular. Apart from that, we observe that full sorting is less expensive than one may expect—roughly 3 times slower than Plan a with PPPL—considering that our algorithm is not multithreaded. Leveraging an index scan (Plan d) also helps a lot, but is of course possible only when working directly on the hierarchical table. Most interestingly, the "order-based sorting" of Hierarchy Rearrange (Plan b) is greatly superior to a full Sort, especially in the bottom-up PPPL case: HR closely approaches the "perfect" speed of Plan a, where no sorting is performed at all. This can be explained by the favorable data locality in the already preorder-sorted inputs.

The pre-sorted scenario (Plan a) of Q6.3 and Q6.4 is closely comparable to our setup for Q5.1 and Q5.2. When comparing the numbers for the HR- and HIS-based plans to Plan a, we notice only a moderate slowdown. The push-based query execution model allows our HR and HIS operators to efficiently pipeline their results into the subsequent Hierarchical Grouping operator, which effectively hides parts of its costs. As a full pipeline breaker, Sort (Plan c) is again at a disadvantage in that regard.

On a higher level, these results also mean that our hierarchy operators that consume postorder inputs are not actually restricted to *only* postorder; by adding Hierarchy Rearrange operators, they can be applied to preorder inputs as well at only moderate extra costs. This also applies to the preorder-based (top-down) algorithms, although HR is somewhat less effective in these cases due to the more complicated logic and additional buffering that is necessary for post-to-pre conversion.

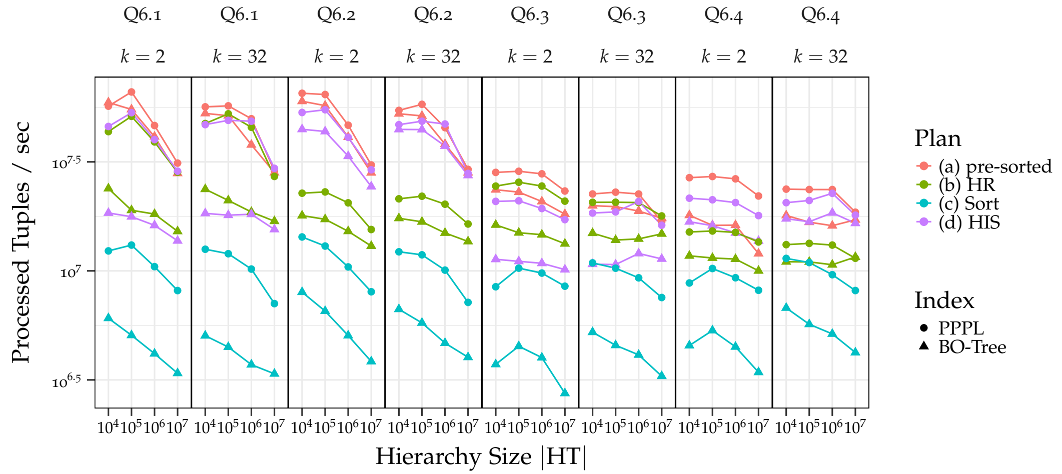


Figure 6.9: Experimental results for § 6.3.6.

6.3.7 Pattern Matching Query

In this experiment we examine the performance to expect from basic structural pattern matching queries. Our test query is similar to the SQL statement of Figure 3.1 (p. 36):

```
SELECT a.ID, a.Payload, b.ID, b.Payload, c.ID, c.Payload
  FROM HT a
  JOIN HT b ON IS_DESCENDANT(b.Node, a.Node)
  JOIN HT c ON IS_DESCENDANT(c.Node, b.Node)
 WHERE a.Weight IN (...) AND b.Weight IN (...) AND c.Weight IN (...)
```

The three `WHERE` conditions are chosen to be quite selective; each selects a different, random 5% of the tuples.

Setup. We use the hierarchical table HT with the Regular $\langle N, k, 10^4 \rangle$ hierarchy structure, and set the size of the Payload field to 16 bytes. Thus, the size of a result row containing three `CHAR(8)` keys and three payload fields is 72 bytes.

We compare three plans. In the plans we use three select conditions $\phi_a(t)$, $\phi_b(t)$, and $\phi_c(t)$ according to the query above. **Plan a** is based on a Hierarchy Index Join:

$a_1 \leftarrow \text{Scan}\,[\text{Node}_a, \text{ID}_a, \text{Payload}_a, \text{Weight}_a]\,(\text{HT})$
$a_2 \leftarrow \text{Select}\,[\phi_a]\,(a_1)$
$a_3 \leftarrow \text{Map}\,[\nu_a : H.\text{node}(\text{Node}_a)]\,(a_2)$
$ab_1 \leftarrow \text{Hierarchy Index Join}\,[H; \nu_b; H.\text{axis}(\nu_b, \nu_a) = \text{descendant}; \ltimes; \text{pre}]\,(a_3)$
$ab_2 \leftarrow \text{Map}\,[b : \text{HT}[\text{rowid}(\nu_b)]; \text{ID}_b : b.\text{ID}, \text{Payload}_b : b.\text{Payload}, \text{Weight}_b : b.\text{Weight}]\,(ab_1)$
$ab_3 \leftarrow \text{Select}\,[\phi_b]\,(ab_2)$
$abc_1 \leftarrow \text{Hierarchy Index Join}\,[H; \nu_c; H.\text{axis}(\nu_c, \nu_b) = \text{descendant}; \ltimes; \text{pre}]\,(ab_3)$
$abc_2 \leftarrow \text{Map}\,[\text{ID}_c, \text{Payload}_c, \text{Weight}_c]\,(abc_1)$ — analogous to ab_2
$abc_3 \leftarrow \text{Select}\,[\phi_c]\,(abc_2)$

While the plan omits it for ease of presentation, our actual implementation of the three plans takes care to actually access the attributes ID, Payload, and Node at the latest

possible occasion, that is, only *after* performing the filtering by Weight. For example, the ab_2 operation is actually split into two Map operators, one before and one behind the Select operator ab_3. Also note that in a real-world schema there might conceivably be an index on the filter attribute (Weight), which could be used as an alternative access path to HT and further speed up this query.

Plan b is based on a Hierarchy Merge Join:

a_1 to a_3 like in Plan a
$b_1 \leftarrow$ Scan $[\text{Node}_b, \text{ID}_b, \text{Payload}_b, \text{Weight}_b]$ (HT)
$b_2 \leftarrow$ Select $[\phi_b]$ (b_1)
$b_3 \leftarrow$ Map $[\nu_b : H.\text{node}(\text{Node}_b)]$ (b_2)
c_1 to c_3 analogous to a_1 to a_3 and b_1 to b_3
$ab \leftarrow$ Hierarchy Merge Join $[H; \nu_b; \nu_a; \text{pre}; \{\text{ancestor}\}; \bowtie]$ (b_3, a_3)
$abc \leftarrow$ Hierarchy Merge Join $[H; \nu_c; \nu_b; \text{pre}; \{\text{ancestor}\}; \bowtie]$ (c_3, ab)

We applied a number of optimizations to this plan: First, the selections have been pushed further downwards than in the HIJ-based Plan a, making Plan b bushy. Second, Plan b takes advantage of HT being clustered and enumerated by the table scans in preorder. (If HT were not clustered, additional Sort operators would be needed prior to the HMJ operators, or alternatively, HT could be accessed via HIS.) Third, the plan demonstrates how two HMJ operators can be chained together by leveraging their sorted output. No additional sorting in between the HMJ operators is needed. Fourth, we changed the join axis to *ancestor* and swapped the join arguments in order to enable the more efficient HMJ A/pre algorithm.

For comparison, **Plan c** evaluates an RCTE that produces the equivalent result. The SQL statement of the RCTE is largely similar to the one shown in Figure 2.5. We omit the lengthy plan, as it is essentially a direct translation of the SQL query statement. It features *two* semi-naïve evaluation steps: First it applies filter ϕ_a, then it joins downwards via the ID–PID association to obtain all descendants, then applies filter ϕ_b, then joins downwards again, and finally applies filter ϕ_c. The used algorithm for the joins within the RCTEs is again a hash join.

Observations. Figure 6.10 shows the runtimes of the plans with different hierarchy indexes, sizes |HT| from 10^3 to 10^7, and shapes $k \in \{2, 32\}$. The times are normalized with respect to |HT|, that is, they show the runtime per 1000 input nodes, which exhibits the scalability properties of the different plans. Plan c is index-independent and measured on the plain table (ID and PID instead of Node). With our test data the sizes of the result sets are as follows:

Hierarchy Size |HT|	10^4	10^5	10^6	10^7
Result Size ($k = 2$)	62	424	3556	48988
Result Size ($k = 32$)	4	8	63	873

From the figure we see that the HIJ-based Plan a is slightly superior to the HMJ-based Plan b for all indexes except DeltaNI. This difference is amplified with flat hierarchies ($k = 32$), where the HMJ-based Plan b is up to three times slower. This may be

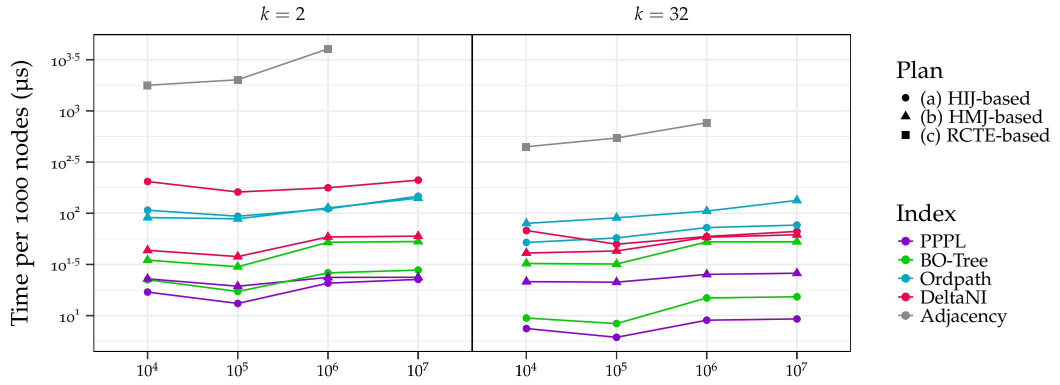


Figure 6.10: Experimental results for § 6.3.7.

surprising, as the HIJ-based plan cannot reduce its input sizes as early as the bushy HMJ-based plan. However, for flat hierarchies the intermediate join results will not be as large anyway, so the early reduction of intermediate result sizes in the bushy HMJ-based plan cannot play out its advantages in this particular query.

At both the HIJ- and the HMJ-based plans, Ordpath and DeltaNI perform painfully worse than the other indexes; this is explained by the is-before-ς() checks of HMJ, which are particularly costly for Ordpath due to its variable-length labels. HIJ relies on the cursor() and pre-next() primitives, which in turn are expensive for DeltaNI.

Regardless of the index type, both Plans a and b easily outperform the RCTE-based Plan c—for a hierarchy as large as $|\mathrm{HT}| = 10^6$ it is by a factor of over 30 slower than the HIJ-based Plan a. Furthermore, Plan c scales poorly with respect to $|\mathrm{HT}|$, which renders it infeasible for size $|\mathrm{HT}| = 10^7$. This difference stems from a fundamental problem of RCTEs: Regardless of the selectivity of the filters, the recursive join inherently iterates over the *complete* hierarchy and yields large intermediate results, only to find that most of these results are irrelevant as the nodes do not match the pattern. By contrast, HIJ and HMJ do not need to enumerate whole subtrees to find matching nodes. With HMJ in particular, the predicate can be pushed down in the plan and only nodes that already meet the filter conditions participate in the join in the first place.

We conducted analogous measurements where we compared `CHAR(8)` keys in the ID and PID columns with `CHAR(32)` keys. The following table shows the runtimes of the various plans with `CHAR(32)` keys, and the respective slowdown over `CHAR(8)`. As the effects are more visible with deep hierarchies, we report only the numbers for $k = 2$:

$\|\mathrm{HT}\|$	Index	Time of Plan a		Time of Plan b	
10^4	BO-Tree	0.3 ms	+25.1%	0.3 ms	+5.6%
	PPPL	0.3 ms	+23.4%	0.2 ms	−18.4%
10^5	BO-Tree	2.7 ms	+20.5%	2.8 ms	+4.0%
	PPPL	2.3 ms	+36.3%	2.1 ms	−8.2%
10^6	BO-Tree	38.3 ms	+18.7%	50.4 ms	+3.4%
	PPPL	33.1 ms	+36.8%	27.3 ms	+2.2%
10^7	BO-Tree	421.0 ms	+22.4%	516.4 ms	+4.5%
	PPPL	396.0 ms	+47.2%	277.0 ms	+1.9%

$\|\mathrm{HT}\|$	Time of Plan d	
10^4	23.3 ms	+94.5%
10^5	258.4 ms	+90.0%
10^6	4923.6 ms	+66.4%

The advantages of the bushy HMJ-based Plan b are now amplified; it it less affected by the larger key type than the HIJ-based Plan a, where the slowdown is roughly between 20% and 35%. These numbers reveal one more general advantage of hierarchical tables over adjacency lists with RCTEs: Hierarchy operators such as HIJ and HMJ work on the system-managed `NODE` column and auxiliary data structure, whereas RCTEs on an adjacency list table necessarily work on the user-prescribed ID/PID columns. A processing-unfriendly key type hurts the performance of RCTEs. This issue disappears with hierarchical tables. With the additional handicap for Plan c in this scenario, Plans a and b now outperform the RCTE by up to two orders of magnitude.

6.3.8 Complex Report Query

Having assessed hierarchical windows in isolation, we next look at a complete query featuring hierarchical windows, namely the report query we studied in § 3.6.5 (p. 60). At the heart, it performs a bottom-up rollup as Q1 in § 6.3.5, but with additional "stress factors" such as additional non-hierarchy joins, multiple hierarchical windows (bottom-up and top-down), carrying along payload, sorting the result in preorder, and computing Dewey-style paths.

Setup. We again use our hierarchical table HT. We choose $|\mathrm{HT}| = 10^4$ and the hierarchy structure Regular $\langle 10^4, k, s\rangle$, which contains a single tree ($s = 10^4$) with a moderate height ($k = 8$). We emulate a setting where the input values are attached to only a *subset* of the hierarchy HT, namely the leaf nodes. To this end we prepare a table Inp containing p% of the 8751 leaf nodes of HT, which are randomly chosen. Inp already has a field ν of the corresponding `Node` objects.

We measure different hand-optimized plans. In all plans, the depth filter on the output nodes, $\phi := (H.\mathrm{depth}(\nu) \leq 3)$, is fully pushed down and handled directly by an ordered index scan of HT. The following operation is common to all plans:

$$\mathrm{HT}_\phi \leftarrow \mathrm{Map}\,[\nu : H.\mathrm{node}(\mathrm{Node})]\,(\mathrm{Scan}\,[\phi]\,(\mathrm{HT})).$$

The following common operations appear in multiple plans:

$$\mathrm{HT}_\phi^{\mathrm{post}} \leftarrow \mathrm{Sort}\,[<_{\text{is-before-post}}]\,(\mathrm{HT}_\phi) \qquad \mathrm{Inp}^{\mathrm{pre}} \leftarrow \mathrm{Sort}\,[<_{\text{is-before-pre}}]\,(\mathrm{Inp})$$
$$\mathrm{HT}_\phi^{\mathrm{pre}} \leftarrow \mathrm{Sort}\,[<_{\text{is-before-pre}}]\,(\mathrm{HT}_\phi) \qquad \mathrm{Inp}^{\mathrm{post}} \leftarrow \mathrm{Sort}\,[<_{\text{is-before-post}}]\,(\mathrm{Inp})$$

Plans a and b use our $\hat{\Gamma}$ and $\bowtie\!\Gamma$ operators. **Plan a** unions together the input table Inp and the hierarchical table HT and runs $\hat{\Gamma}$ on the union table:

$e_1 \leftarrow \mathrm{Merge\ Union}\,[<_{\text{is-before-post}}]\,(\mathrm{HT}_\phi^{\mathrm{post}}, \mathrm{Inp}^{\mathrm{post}})$
$e_2 \leftarrow \mathrm{Hierarchical\ Grouping}\,[H; \nu; \mathrm{post}; \{\mathrm{descendant}\}; x : \mathrm{sum}(\mathrm{Value})]\,(e_1)$
$e_3 \leftarrow \mathrm{Select}\,[\phi]\,(e_2)$
$e_4 \leftarrow \mathrm{Hierarchy\ Rearrange}\,[H; \nu; \mathrm{post} \to \mathrm{pre}]\,(e_3)$
$e_5 \leftarrow \mathrm{Hierarchical\ Grouping}\,[H; \nu; \mathrm{pre}; \{\mathrm{ancestor}\}; (p, \mathrm{Path}) : \ldots]\,(e_4)$

The Merge Union is an efficient sort-based algorithm. The Select for ϕ "undoes" the union and retains only the output tuples we are interested in. The final HG handles both top-down computations: the contribution percentage p and the Dewey string Path. It also preserves the order of its input, which yields the result in the desired preorder.

Plan b uses a Hierarchy Merge Groupjoin between HT and Inp and thus does not need a Merge Union:

$e_1 \leftarrow$ Hierarchy Merge Groupjoin $[H; t.\nu; u.\nu; \text{post}; \{\text{self}, \text{desc.}\}; \ldots]$ $(\text{HT}_\phi^{\text{post}}[t], \text{Inp}^{\text{post}}[u])$
$e_2 \leftarrow$ Hierarchy Rearrange $[H; t.\nu; \text{post} \to \text{pre}]$ (e_1)
$e_3 \leftarrow$ Hierarchical Grouping $[H; t.\nu; \text{pre}, \{\text{ancestor}\}; (p, \text{Path}) : \ldots]$ (e_2)

For **Plan c** we assume the hierarchical table model without our syntax extensions for hierarchical windows and without our operators HG and HMGJ for structural grouping. It relies on Hierarchy Merge Join, that is, the HMJ-Group approach.

$e_1 \leftarrow$ Hierarchy Merge Join $[H; t.\nu; u.\nu; \text{pre}, \{\text{descendant}\}; ⟕]$ $(\text{HT}_\phi^{\text{pre}}[t], \text{Inp}^{\text{pre}}[u])$
$e_2 \leftarrow$ Group $[t.\nu; x : \text{sum}(u.\text{Value})]$ (e_1)
$e_3 \leftarrow$ Materialize(e_2)
$e_4 \leftarrow$ Hierarchy Merge Join $[H; t.\nu; p_1.\nu; \text{pre}; \{\text{parent}\}; ⟕]$ $(e_3[t], e_3[p_1])$
$e_5 \leftarrow$ Hierarchy Merge Join $[H; p_1.\nu; p_2.\nu; \text{pre}; \{\text{parent}\}; ⟕]$ $(e_4, e_3[p_2])$
$e_6 \leftarrow$ Map $[(p, \text{Path}) : \ldots]$ (e_4)

The grouping is sort-based. Lacking our syntax extensions, a lot of manual "SQL labor" is involved: The upper three levels are assembled via two left outer is-parent() joins, yielding the first parent p_1 and the second parent p_2. Then the final Map operator constructs the path strings and computes the contribution percentages by hand from p_2, p_1, and t.

With **Plan d** we emulate a hand-implemented static PPPL-like labeling scheme. Lacking engine support, it has to rely on nested loop joins, that is, the Join-Group approach.

$e_1 \leftarrow$ Nested Loop Join $[H.\text{axis}(u.\nu, t.\nu) \in \{\text{descendant, self}\}; ⟕]$ $(\text{HT}_\phi[t], \text{Inp}[u])$
$e_2 \leftarrow$ Group $[t.\nu; x : \text{sum}(u.\text{Value}]$ (e_1)
$e_3 \leftarrow$ Materialize(e_2)
$e_4 \leftarrow$ Nested Loop Join $[H.\text{is-parent}(p_1.\nu, t.\nu); ⟕]$ $(e_3[t], e_3[p_1])$
$e_5 \leftarrow$ Nested Loop Join $[H.\text{is-parent}(p_2.\nu, p_1.\nu); ⟕]$ $(e_4, e_3[p_2])$
$e_6 \leftarrow$ Map $[(p, \text{Path}) : \ldots]$ (e_5)
$e_7 \leftarrow$ Sort $[<_{\text{is-before-pre}}]$ (e_6)

For **Plan e**, we assume again the adjacency list model and a hand-written stored procedure which does an iterative fixpoint computation (like Iterative in § 6.3.5). After the bottom-up computation it uses a `PID IS NULL` filter and three hash joins to determine the upper three levels for the result and to fill in the p and Path fields.

Although Plans d–e appear to be severely handicapped versus a–c, they are representative of the state of the art in real-world applications we encountered. In particular, an approach based on the adjacency list model (Plan e) is very common in practice.

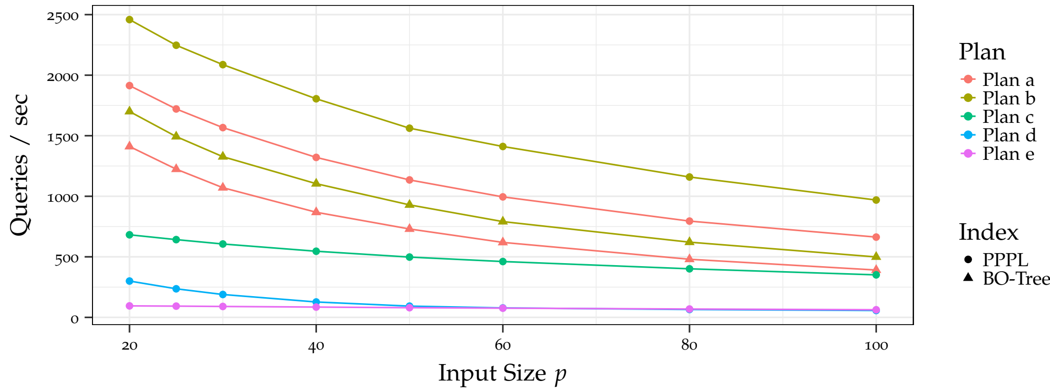


Figure 6.11: Experimental results for § 6.3.8.

Observations. Figure 6.11 shows the measured query throughput over varying p. The big performance gaps we have seen in some of the previous experiments are now less pronounced due to a damping effect of the other involved relational operators. In particular, a big pain point in this query is the expensive sorting of Inp, which turns more expensive the bigger p gets. This performance issue could be alleviated by employing a parallel sorting algorithm. Nevertheless, we still see the merits of our proposed syntax and algorithms for hierarchical computations: Both $\hat{\Gamma}$ (Plan a) and $\bowtie\!\Gamma$ (Plan b) handle the query quite reasonably. As the Hierarchy Merge Groupjoin more naturally fits the binary nature of this query (and obviates the Merge Union needed to prepare the input for $\hat{\Gamma}$ in Plan a), Plan b significantly outperforms Plan a in this case. The advantage of having dedicated operators HMGJ and HG over plain HMJ-Group (Plan c) is again clearly visible, although less pronounced than in § 6.3.5 due to the damping effect of the sorting. Plan d is painfully slowed down by the employed nested loop joins, though still faster than Plan e, which always has to perform an iterative join through the *complete* hierarchy. This renders it constantly slow, regardless of the actual input size p.

The numbers show a clear benefit of having dedicated operators for hierarchical computations (Plans a and b versus Plan c). They also show that Plans d and e are not just unwieldy hand-crafted solutions and unsatisfactory in terms of expressiveness; they also cannot hold up with our framework in terms off efficiency.

6.4 Summary

Altogether, our experiments touch on all major components of our proposed framework, and thus give a good indication of the performance one can expect in practice.

Our assessment of index primitives for queries (§ 6.3.2) shows that the interface for hierarchy indexes can be implemented highly efficiently even with dynamic schemes. The proposed standard schemes—PPPL for static scenarios and BO-Tree for dynamic scenarios—perform well throughout all disciplines. Beyond its query performance, BO-

Tree with mixed block sizes is also an excellent all-round index structure with robustness for all update operations; it therefore should be the first choice when the update pattern is unknown. Its performance can be optimized further by choosing the simpler O-List index structure, a different back-link representation, or a different block size, as we discuss in our paper [33].

Besides the index primitives, our experiments also assess the performance characteristics of our essential operators HIS, HIJ, HMJ, HMGJ, HG, and HR, both in isolation as well as in complex plans for typical pattern matching and hierarchical computation queries. They show that our framework clearly outperforms any solution based on adjacency list tables and recursive stored procedures or RCTEs. In fact, the query plans are often so fast that they would outperform an RCTE-based approach *even* if we perform a full bulk-build prior to executing them. For example, with 10^7 nodes, if we execute Plan a from §6.3.7 together with the bulk build (assessed in §6.3.1), the speedup over the RCTE is still more than a factor of 5. Of course, the ultimate goal of every legacy application should be to migrate from derived hierarchies to hierarchical base tables, so that bulk-building is never necessary. Still, our results show that even applications that conceptually derive a hierarchy for *every* query will be able to achieve considerable performance gains.

7 Conclusions and Outlook

Working with hierarchies in relational databases has always been complicated. None of the conventional solutions we surveyed were able to fully meet the typical requirements that we identified in the applications of SAP and beyond. These requirements call for a holistic approach that seamlessly integrates relational and hierarchical data on the levels of the data model, the query language, and the backend of the database system, while retaining the philosophy of the relational data model.

Our solution is to represent hierarchies by means of an abstract `NODE` data type and to provide the necessary language extensions to define, manipulate, and query hierarchies in a user-friendly way. Many formerly complicated tasks are thus greatly simplified: Developers are no longer burdened with physical design considerations, as a simple abstract data type replaces any hand-crafted tree encoding schemes. Expressive language constructs turn convoluted blocks of SQL text into concise and intuitive statements. The task of choosing a suitable indexing scheme that balances storage utilization, support for updates, and query performance is handed off to the system, which also makes it easy to adapt an evolving application to changing workloads throughout its lifetime. Altogether, these simplifications promise a boost in developer productivity and maintainability. In addition, the sophisticated indexing and query processing techniques we employ at the backend make sure that performance and scalability are not compromised along the way. These features do not only benefit green-field applications; they are also made available to existing applications through facilities for deriving hierarchies from existing data and for bulk-loading hierarchy indexes efficiently. This allows legacy applications to gradually push hierarchy-handling logic that formerly lived in the application layer down to the database layer, and thus benefit as well from the productivity and performance gains that our framework brings along.

Our research prototype already shows promising evidence of the mentioned merits over conventional approaches. As part of our cooperation with SAP, an "industrial-grade" proof-of-concept prototype was also developed in collaboration with members of the HANA team. It is especially pleasant to see that at the time of writing, the technology has already been made publicly available to users of both the HANA Database and the SAP Vora engine (see "`HIERARCHY` function" in [90], and [91, 92]). The HANA platform has from an early stage on been envisioned as a flexible query processing environment to accommodate data of different degrees of structure, and our concept of hierarchical tables fits well in this spirit. We hope that this will open the doors for conducting broader user studies and gaining first-hand customer feedback, which can eventually inspire further research and development around our framework.

Outlook. In this thesis we already exploit many of the opportunities our hierarchical table model provides both from a language perspective (`NODE` as a "syntactic handle") and from a backend perspective (indexing the `NODE` column and leveraging it during query evaluation). But further enhancements are always possible. Two especially rich areas of open problems we already covered earlier are about *concurrency control* (see § 4.7) and *query optimization and processing* on hierarchical data (see § 5.3). We conclude this thesis by outlining a few further promising ideas for future work.

Benchmarking. The aforementioned customer-available version of our framework will ideally enable us to collect experiences with a wider range of real-world hierarchical data sets and application patterns, and thus unveil areas for further improvement. It would be a promising endeavor to condense the most typical patterns into a compact benchmark suite in the spirit of the well-known TPC benchmarks [103]. Such a suite can be very helpful in motivating and driving future research in the area.

Other database systems. While our prototype was implemented with an eye towards modern engines based on memory-resident tables and a push-based query execution model, the techniques could in principle be adapted to other RDBMS architectures. With the exception of our SQL-based language extensions, we would in fact expect most components of our framework—such as the indexing schemes and the algorithms for query processing—to be applicable as well to a broader range of non-relational systems, such as XML or graph databases. For example, it should be possible to implement an XPath engine [110] entirely in terms of our low-level index interface. Such extended application areas could possibly be explored in the future.

Features. Many extensions to our feature set are conceivable, but they should ideally be motivated by realistic use cases. A few ideas are to add further syntax for customizing how window frames are formed in hierarchical computations (e. g., to take into account the sibling order); to allow users to add functionality by defining custom operations on the `NODE` type; or to generalize SQL's `ROLLUP` feature to work with hierarchical tables. The latter would benefit the tightly constrained applications that use `ROLLUP` or `MDX` today, as discussed in § 2.3.10 and § 2.3.5. A further challenge are *multi-dimensional* computations: While we considered only the single-dimensional case, generalizing hierarchical windows and rollups to two or more `NODE` fields should be possible.

Hierarchical and temporal data. To keep historic states of data available for analytics or auditing purposes, business applications routinely use *temporal* tables, where one or more system time or application time dimensions are modeled using SQL's `PERIOD` construct [49, 55]. When a hierarchical dimension is defined on a temporal table, the result is a *temporal hierarchy*. The hierarchy may arrange multiple instances of the same logical entity at different time periods, as if each edge was labeled with a certain validity period. Such hierarchies are strict trees only as long as they are restricted to a single point in time; when viewed over *all* periods they are general graphs. Advanced queries on temporal hierarchies may involve both dimensions simultaneously and even span over *intervals* of time; for example: "Perform a hierarchical sales rollup and determine the

point in time with the maximum sales total." These queries are very hard to evaluate efficiently, as indexes and algorithms are conventionally designed for either hierarchical *or* temporal data, but not both (but see [31, 69, 88, 120]). Although this topic was beyond our scope, our design takes care that a temporal table is conceptually and syntactically not hindered from having a hierarchical dimension as well. It can therefore be integrated with the functionality of `PERIOD` in the future.

Non-strict hierarchies. We handle non-strict hierarchies in a minimalistic but conceptually clean way by transforming them into trees at hierarchy derivation time. That said, our `NODE` model could in principle accommodate general acyclic directed graphs "natively": one could use graph indexes to store them and generalized algorithms to query them. There are some major challenges, though. Syntactically, it is not obvious how the edges (which become distinct entities) could be accessed in a concise and effective way without violating the look and feel of SQL. Semantically, issues would arise with tree-based hierarchy functions and predicates and the ambiguous data flow in hierarchical computations. Moreover, it is uncertain whether such support would have any selling point against dedicated graph stores and their more purpose-built query interfaces, which is why we did not pursue these ideas further (see also § 2.2.2).

Adaptivity. In § 4.3 we proposed a simple static selection method for indexing schemes. A more sophisticated system could conceivably *adapt* during the lifetime of the hierarchical table. It might, for example, take the current workload into account and decide to switch between indexes on demand. For slowly changing hierarchical tables, a static indexing scheme could be used when the table is first loaded and at times where no updates happen, but it might be converted to a query-optimized dynamic index type as soon as some updates are performed. The conversion costs could be kept moderate by leveraging the highly efficient build() operation. Even more sophisticated forms of adaptivity are conceivable: An adaptive index might *replicate* certain table columns that are accessed frequently in conjunction with the nodes. This would enable more index-only queries and thus further boost performance.

Recursive Common Table Expressions. There are certain situations where the features for RCTEs and hierarchical tables could interact. On the one hand, an RCTE operating on a hierarchical table could be accelerated if the system could deduce the kind of traversal that is being performed and then leverage the available hierarchy indexes accordingly. On the other hand, an RCTE could produce a hierarchical table from its "recursion tree" as a by-product. To do so, the semi-naïve algorithm would have to keep track of the parent rows that spawned each of the generated rows at the time the join with the recursive table is evaluated. This feature would enable applications to initially use an RCTE to explore an unknown graph, but subsequently switch to the more comfortable tools for hierarchies to analyze and work on the resulting tree.

Bibliography

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. "The Lorel query language for semistructured data." *Int. Journal on Digital Libraries (JODL)* vol. 1, no. 1 (1997), pp. 68–88.

[2] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. "QRS: A robust numbering scheme for XML documents." *Proc. 19th Int. Conf. Data Engineering (ICDE)*. IEEE, 2003, pp. 705–707.

[3] Philip A. Bernstein and Nathan Goodman. "Concurrency control in distributed database systems." *ACM Computing Surveys (CSUR)* vol. 13, no. 2 (1981), pp. 185–221.

[4] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert van der Linden, Brian Vickery, and Chun Zhang. "System RX: One part relational, one part XML." *Proc. 2005 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2005, pp. 347–358.

[5] K. R. Blackman. "Technical Note: IMS celebrates thirty years as an IBM product." *IBM Systems Journal* vol. 37, no. 4 (1998), pp. 596–603.

[6] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. "MonetDB/XQuery: A fast XQuery processor powered by a relational engine." *Proc. 2006 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2006, pp. 479–490.

[7] Peter Boncz, Stefan Manegold, and Jan Rittinger. "Updating the pre/post plane in MonetDB/XQuery." *Informal Proc. 2nd Int. Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*. 2005.

[8] Robert Brunel and Jan Finis. "Eine effiziente Indexstruktur für dynamische hierarchische Daten." *Datenbanksysteme für Business, Technologie und Web (BTW) 2013. Workshopband P-216* (Mar. 2013), pp. 267–276.

[9] Robert Brunel, Jan Finis, Gerald Franz, Norman May, Alfons Kemper, Thomas Neumann, and Franz Färber. "Supporting hierarchical data in SAP HANA." *Proc. 31st Int. Conf. Data Engineering (ICDE)*. IEEE, Apr. 2015, pp. 1280–1291.

[10] Robert Brunel, Norman May, and Alfons Kemper. "Index-assisted hierarchical computations in main-memory RDBMS." *Proc. VLDB Endowment* vol. 9, no. 12 (Aug. 2016), pp. 1065–1076.

[11] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. "Holistic Twig Joins: Optimal XML pattern matching." *Proc. 2002 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2002, pp. 310–321.

[12] Jing Cai and Chung Keung Poon. "OrdPathX: Supporting two dimensions of node insertion in XML data." *Proc. 20th Int. Conf. Database and Expert Systems Applications (DEXA)*. Springer-Verlag Berlin Heidelberg, 2009, pp. 332–339.

[13] Joe Celko. *Trees and Hierarchies in SQL for Smarties*. 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, Jan. 2012.

[14] Upen S. Chakravarthy, John Grant, and Jack Minker. "Logic-based approach to semantic query optimization." *ACM Transactions on Database Systems (TODS)* vol. 15, no. 2 (1990), pp. 162–207.

[15] Damianos Chatziantoniou, Theodore Johnson, Michael Akinde, and Samuel Kim. "The MD-join: An operator for complex OLAP." *Proc. 17th Int. Conf. Data Engineering (ICDE)*. IEEE, 2001, pp. 524–533.

[16] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. "An overview of Business Intelligence technology." *Communications of the ACM* vol. 54, no. 8 (2011), pp. 88–98.

[17] Sudarshan S. Chawathe and Hector Garcia-Molina. "Meaningful change detection in structured data." *Proc. 1997 ACM SIGMOD Int. Conf. Management of Data*. ACM, 1997, pp. 26–37.

[18] Peter Pin-Shan Chen. "The Entity-Relationship Model—Toward a unified view of data." *ACM Transactions on Database Systems (TODS)* vol. 1, no. 1 (1976), pp. 9–36.

[19] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. "Twig^{2}Stack: Bottom-up processing of generalized tree pattern queries over XML documents." *Proc. 32nd Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2006, pp. 283–294.

[20] Sophie Cluet and Guido Moerkotte. "Efficient evaluation of aggregates on bulk types." *Proc. 5th Int. Workshop on Database Programming Languages (DBPL)*. Springer-Verlag Berlin Heidelberg, 1995.

[21] Edgar F. Codd. "A relational model of data for large shared data banks." *Communications of the ACM* vol. 13, no. 6 (1970), pp. 377–387.

[22] Edith Cohen, Haim Kaplan, and Tova Milo. "Labeling dynamic XML trees." *SIAM Journal on Computing (SICOMP)* vol. 39, no. 5 (2010), pp. 2048–2074.

[23] *Common Warehouse Metamodel (CWM) Specification*. OMG Document formal/03-03-02. Version 1.1. Object Management Group. Mar. 2003. www.omg.org/spec/CWM/1.1/PDF.

[24] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. "A fast index for semistructured data." *Proc. 27th Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2001, pp. 341–350.

[25] Hugh Darwen and Christopher J. Date. "An overview and analysis of proposals based on the TSQL2 approach." *Date on Database: Writings 2000–2006*. New York City, USA: Apress, 2007.

[26] Reinhard Diestel. *Graph Theory*. 4th ed. Graduate Texts in Mathematics. Volume 173. Springer-Verlag Berlin Heidelberg, 2010.

[27] *Technische Produktdokumentation – CAD-Modelle, Zeichnungen und Stücklisten. Teil 1: Begriffe*. DIN 199-1:2002–03. Deutsches Institut für Normung, 2002.

[28] Andrew Eisenberg and Jim Melton. "Advancements in SQL/XML." *ACM SIGMOD Record* vol. 33, no. 3 (2004), pp. 79–86.

[29] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database—An architecture overview." *IEEE Data Engineering Bulletin* vol. 35, no. 1 (2012), pp. 28–33.

[30] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs. An Introduction to Programming and Computing*. Cambridge, MA, USA: The MIT Press, 2001.

[31] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Franz Färber, and Norman May. "DeltaNI: An efficient labeling scheme for versioned hierarchical data." *Proc. 2013 ACM SIGMOD Int. Conf. Management of Data*. ACM, June 2013, pp. 905–916.

[32] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. "Indexing highly dynamic hierarchical data." *Proc. VLDB Endowment* vol. 8, no. 10 (Aug. 2015), pp. 986–997.

[33] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. "Order Indexes: Supporting highly dynamic hierarchical data in relational main-memory database systems." *The VLDB Journal* vol. 26, no. 1 (Feb. 2017), pp. 55–80.

[34] Jan Finis, Martin Raiber, Nikolaus Augsten, Robert Brunel, Alfons Kemper, and Franz Färber. "RWS-Diff: Flexible and efficient change detection in hierarchical data." *Proc. 22nd ACM Int. Conf. Information and Knowledge Management (CIKM)*. ACM, Oct. 2013, pp. 339–348.

[35] Shel J. Finkelstein, Nelson Mattos, Inderpal Mumick, and Hamid Pirahesh. *Expressing recursive queries in SQL*. ISO/IEC JTC 1/SC 21 WG 3 Document X3H2-96-075r1. Joint Technical Committee ISO/IEC JTC 1, Mar. 6, 1996.

[36] Daniela Florescu and Donald Kossmann. "A performance evaluation of alternative mapping schemes for storing XML data in a relational database." Research Report RR-3680. Institut national de recherche en informatique et en automatique (INRIA), May 1999.

[37] Goetz Graefe. "Query evaluation techniques for large databases." *ACM Computing Surveys (CSUR)* vol. 25, no. 2 (1993), pp. 73–169.

[38] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. "Data Cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Totals." *Data Mining and Knowledge Discovery* vol. 1, no. 1 (1997), pp. 29–53.

[39] Torsten Grust. "Accelerating XPath location steps." *Proc. 2002 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2002, pp. 109–120.

[40] Torsten Grust, Maurice van Keulen, and Jens Teubner. "Accelerating XPath evaluation in any RDBMS." *ACM Transactions on Database Systems (TODS)* vol. 29, no. 1 (2004), pp. 91–131.

[41] Torsten Grust, Maurice van Keulen, and Jens Teubner. "Staircase Join: Teach a relational DBMS to watch its (axis) steps." *Proc. 29th Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2003, pp. 524–535.

[42] Torsten Grust, Jan Rittinger, and Jens Teubner. "Why off-the-shelf RDBMSs are better at XPath than you might expect." *Proc. 2007 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2007, pp. 949–958.

[43] Torsten Grust, Sherif Sakr, and Jens Teubner. "XQuery on SQL hosts." *Proc. 30th Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2004, pp. 252–263.

[44] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis D. Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. "Mixed mode XML query processing." *Proc. 29th Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2003, pp. 225–236.

[45] Birgitta Hauser. *Hierarchical queries with DB2 Connect By. A new method for recursively processing data relationships*. IBM Corp. Aug. 22, 2011. www.ibm.com/developerworks/ibmi/library/i-db2connectby.

[46] Michael Haustein, Theo Härder, Christian Mathis, and Markus Wagner. "DeweyIDs—The key to fine-grained management of XML documents." *Proc. 20th Brazilian Symposium on Databases (SBBD)*. Sociedade Brasileira de Computação, 2005, pp. 85–99.

[47] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. "Generalized search trees for database systems." *Proc. 21th Int. Conf. Very Large Databases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1995, pp. 562–573.

[48] Hosagrahar V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. "Timber: A native XML database." *VLDB Journal* vol. 11, no. 4 (2002), pp. 274–291.

[49] Christian S. Jensen, James Clifford, Ramez Elmasri, Shashi K. Gadia, Pat Hayes, and Sushil Jajodia. "A consensus glossary of temporal database concepts." *ACM SIGMOD Record* vol. 23, no. 1 (1994), pp. 52–64.

[50] Christian S. Jensen, Torben Bach Pederesen, and Christian Thomsen. "Multidimensional databases and data warehousing." *Synthesis Lectures on Data Management*. Ed. by Meral Özsoyoğlu. Vol. 2. 1. Morgan & Claypool Publishers, 2010, pp. 1–111.

[51] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. "Holistig Twig Joins on indexed XML documents." *Proc. 29th Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2003, pp. 273–284.

[52] *RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format*. Proposed Standard. Internet Engineering Task Force (IETF), Mar. 2014. tools.ietf.org/html/rfc7159.

[53] Shurug Al-Khalifa, Hosagrahar V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. "Structural Joins: A primitive for efficient XML query pattern matching." *Proc. 18th Int. Conf. Data Engineering (ICDE)*. IEEE, 2002, pp. 141–152.

[54] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit. The Complete Guide to Dimensional Modeling*. 2nd ed. John Wiley & Sons, 2002.

[55] Krishna Kulkarni and Jan-Eike Michels. "Temporal Features in SQL:2011." *ACM SIGMOD Record* vol. 41, no. 3 (2012), pp. 34–43.

[56] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. "Index structures for structured documents." *Proc. 1st ACM Int. Conf. Digital Libraries (DL)*. ACM, 1996, pp. 91–99.

[57] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." *Proc. 2014 ACM SIGMOD Int. Conf. Management of Data (SIGMOD)*. ACM. 2014, pp. 743–754.

[58] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. "Efficient processing of window functions in analytical SQL queries." *Proc. VLDB Endowment* vol. 8, no. 10 (2015), pp. 1058–1069.

[59] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. "Query optimization by predicate move-around." *Proc. 20th Int. Conf. Very Large Databases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1994, pp. 96–107.

[60] Changqing Li and Tok Wang Ling. "QED: A novel quaternary encoding to completely avoid re-labeling in XML updates." *Proc. 14th ACM Int. Conf. Information and Knowledge Management (CIKM)*. ACM, 2005, pp. 501–508.

[61] Changqing Li, Tok Wang Ling, and Min Hu. "Efficient processing of updates in dynamic XML data." *Proc. 22nd Int. Conf. Data Engineering (ICDE)*. IEEE, 2006.

[62] Changqing Li, Tok Wang Ling, and Min Hu. "Efficient updates in dynamic XML data: from binary string to quaternary string." *VLDB Journal* vol. 17, no. 3 (2008), pp. 573–601.

[63] Quanzhong Li and Bongki Moon. "Indexing and querying XML data for regular path expressions." *Proc. 27th Int. Conf. Very Large Databases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2001, pp. 361–370.

[64] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. "Native XQuery processing in Oracle XMLDB." *Proc. 2005 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2005, pp. 828–833.

[65] Norman May and Guido Moerkotte. "Main memory implementations for binary grouping." *Proc. 2005 Int. XML Database Symposium (XSym)*. Springer-Verlag Berlin Heidelberg, 2005, pp. 162–176.

[66] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. "Lore: a database management system for semistructured data." *ACM SIGMOD Record* vol. 26, no. 3 (1997), pp. 54–66.

[67] Jason McHugh, Jennifer Widom, Serge Abiteboul, Qingshan Luo, and Anand Rajaraman. *Indexing semistructured data*. Technical Report. Stanford University, 1998.

[68] Dinesh P. Mehta and Sartaj Sahni. *Handbook of Data Structures and Applications*. Boca Raton, Floria, USA: CRC Press, Oct. 2004.

[69] Alberto O. Mendelzon, Flavio Rizzolo, and Alejandro Vaisman. "Indexing temporal XML documents." *Proc. 30th Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2004, pp. 216–227.

[70] *Microsoft SQL Documentation – Relational databases – Hierarchical Data*. Microsoft Corp. Mar. 14, 2017. docs.microsoft.com/en-us/sql/relational-databases/hierarchical-data-sql-server.

[71] *Microsoft SQL Documentation – Multidimensional Expressions (MDX) Reference*. Microsoft Corp. Mar. 2, 2016. docs.microsoft.com/en-us/sql/mdx/multidimensional-expressions-mdx-reference.

[72] Tova Milo and Dan Suciu. "Index structures for path expressions." *Proc. 7th Int. Conf. Database Theory (ICDT)*. Springer-Verlag Berlin Heidelberg, 1999, pp. 277–295.

[73] Jun-Ki Min, Jihyun Lee, and Chin-Wan Chung. "An efficient XML encoding and labeling method for query processing and updating on dynamic XML data." *Journal of Systems and Software (JSS)* vol. 82, no. 3 (2009), pp. 503–515.

[74] Guido Moerkotte and Thomas Neumann. "Accelerating queries with Group-By and Join by Groupjoin." *Proc. VLDB Endowment* vol. 4, no. 11 (2011), pp. 843–851.

[75] Karen Morton, Kerry Osborne, Robyn Sands, Riyaj Shamsudeen, and Jared Still. *Pro Oracle SQL*. 2nd ed. New York City, USA: Apress, 2013.

[76] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware." *Proc. VLDB Endowment*. Vol. 4. 9. VLDB Endowment, 2011, pp. 539–550.

[77] Thomas Neumann, Sven Helmer, and Guido Moerkotte. "On the optimal ordering of maps and selections under factorization." *Proc. 21st Int. Conf. Data Engineering (ICDE)*. IEEE, 2005, pp. 490–501.

[78] Paul Nielsen, Mike White, and Uttam Parui. *Microsoft SQL Server 2008 Bible*. John Wiley & Sons, Sept. 2009.

[79] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. "ORDPATHs: Insert-friendly XML node labels." *Proc. 2004 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2004, pp. 903–908.

[80] *Oracle Database 12c Release 2 (12.2). SQL Language Reference*. Oracle Corp. Apr. 2017. docs.oracle.com/database/122/SQLRF/toc.htm.

[81] *Oracle 9i OLAP Release 2 (9.2). User's Guide*. Oracle Corp. Mar. 2002. docs.oracle.com/cd/A97630_01/olap.920/a95295.pdf.

[82] Carlos Ordonez. "Optimization of linear recursive queries in SQL." *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. Vol. 22. 2. IEEE, 2010, pp. 264–277.

[83] Carlos Ordonez, Achyuth Gurram, and Nirmala Rai. "Recursive query evaluation in a column DBMS to analyze large graphs." *Proc. 17th Int. Workshop Data Warehousing and OLAP (DOLAP)*. ACM, 2014, pp. 71–80.

[84] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. "XQuery implementation in a relational database system." *Proc. 31st Int. Conf. Very Large Databases (VLDB)*. VLDB Endowment, 2005, pp. 1175–1186.

[85] Glenn N. Paulley. "Exploiting functional dependence in query optimization." Ph.D. Thesis. University of Waterloo, Apr. 2000.

[86] Torben Bach Pedersen and Christian S. Jensen. "Multidimensional database technology." *IEEE Computer* vol. 34, no. 12 (2001), pp. 40–46.

[87] *PostgreSQL 9.6.3 Documentation – Appendix F: Additional Supplied Modules – ltree*. The PostgreSQL Global Development Group. 2017. www.postgresql.org/docs/9.6/static/ltree.html.

[88] Flavio Rizzolo and Alejandro A. Vaisman. "Temporal XML: modeling, indexing, and query processing." *VLDB Journal* vol. 17, no. 5 (2008), pp. 1179–1212.

[89] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., June 2015.

[90] *SAP HANA Platform 2.0 SPS 01. SAP HANA SQL and System Views Reference*. Document version 1.0. SAP SE. Apr. 12, 2017. help.sap.com/viewer/product/SAP_HANA_PLATFORM/2.0.01/en-US.

[91] *SAP Software Solutions – Products – SAP Vora*. SAP SE. 2017. go.sap.com/germany/product/data-mgmt/hana-vora-hadoop.html.

[92] *SAP Vora Developer Guide – Using Hierarchies*. SAP SE. 2017. help.sap.com/viewer/product/SAP_VORA/1.4/en-US.

[93] Sunita Sarawagi. "Indexing OLAP data." *IEEE Data Engineering Bulletin*. Vol. 20. 1. IEEE, 1997, pp. 36–43.

[94] Patricia Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. "Access path selection in a relational database management system." *Proc. 1979 ACM SIGMOD Int. Conf. Management of Data*. ACM, 1979, pp. 22–34.

[95] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. "Relational databases for querying XML documents: limitations and opportunities." *Proc. 25th Int. Conf. Very Large Databases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1999, pp. 302–314.

[96] Adam Silberstein, Hao He, Ke Yi, and Jun Yang. "BOXes: Efficient maintenance of order-based labeling for dynamic XML data." *Proc. 21st Int. Conf. Data Engineering (ICDE)*. IEEE, 2005, pp. 285–296.

[97] *ISO/IEC Standard 9075: Information technology – Database languages – SQL*. Joint Technical Committee ISO/IEC JTC 1, Dec. 2011.

[98] *ISO/IEC Standard 9075-14: Information technology – Database languages – SQL, Part 14: XML-Related Specifications (SQL/XML)*. Joint Technical Committee ISO/IEC JTC 1, Dec. 2011.

[99] Igor Tatarinov, Stratis Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. "Storing and querying ordered XML using a relational database system." *Proc. 2002 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2002, pp. 204–215.

[100] Dimitri Theodoratos. "Exploiting hierarchical clustering in evaluating multidimensional aggregation queries." *Proc. 6th ACM Int. Workshop Data Warehousing and OLAP (DOLAP)*. ACM, 2003, pp. 63–70.

[101] Erik Thomsen. *OLAP Solutions. Building Multidimensional Information Systems*. 2nd ed. John Wiley & Sons, Apr. 2002.

[102] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. "The design and performance evaluation of alternative XML storage strategies." *ACM SIGMOD Record* vol. 31, no. 1 (2002), pp. 5–10.

[103] The Transaction Processing Performance Council (TPC). www.tpc.org.

[104] Vadim Tropashko. "Nested intervals tree encoding in SQL." *ACM SIGMOD Record* vol. 34, no. 2 (2005), pp. 47–52.

[105] Vadim Tropashko. "Nested intervals tree encoding with continued fractions." *Computing Research Repository* (2004). arxiv.org/abs/cs.DB/0402051.

[106] Peter T. Wood. "Query languages for graph databases." *ACM SIGMOD Record* vol. 41, no. 1 (2012), pp. 50–60.

[107] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. "Sampling-based query re-optimization." *Proc. 2016 ACM Int. Conf. Management of Data (SIGMOD)*. ACM. 2016, pp. 1721–1736.

[108] *Extensible Markup Language (XML), Version 1.0 (5th ed.)* W3C Recommendation. The World Wide Web Consortium (W3C), Nov. 26, 2008. www.w3.org/TR/xml.

[109] *XML for Analysis Specification, Version 1.0*. Microsoft Corp. and Hyperion Solutions Corp., Apr. 24, 2001. msdn.microsoft.com/en-us/library/ms977626.aspx.

[110] *XML Path Language (XPath), Version 2.0 (2nd ed.)* W3C Recommendation. The World Wide Web Consortium (W3C), Dec. 14, 2010. www.w3.org/TR/xpath20.

[111] *XQuery: An XML Query Language, Version 1.0 (2nd ed.)* W3C Recommendation. The World Wide Web Consortium (W3C), Dec. 14, 2010. www.w3.org/TR/xquery.

[112] *XQuery Update Facility, Version 1.0*. W3C Recommendation. The World Wide Web Consortium (W3C), Mar. 17, 2011. www.w3.org/TR/xquery-update-10.

[113] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. "DDE: From Dewey to a fully dynamic XML labeling scheme." *Proc. 2009 ACM SIGMOD Int. Conf. Management of Data*. ACM, 2009, pp. 719–730.

[114] Jeffrey Xu Yu, Daofeng Luo, Xiaofeng Meng, and Hongjun Lu. "Dynamically updating XML data: Numbering scheme revisited." *World Wide Web Journal* vol. 8, no. 1 (2005), pp. 5–26.

[115] Jung-Hee Yun and Chin-Wan Chung. "Dynamic interval-based labeling scheme for efficient XML query and update processing." *Journal of Systems and Software (JSS)* vol. 81, no. 1 (2008), pp. 56–70.

[116] Fred Zemke, Krishna Kulkarni, Andy Witkowski, and Bob Lyle. *Introduction to OLAP functions*. ISO/IEC JTC 1/SC 32 WG 3 Document YGJ-068= ANSI NCITS H2-99-154r2. Joint Technical Committee ISO/IEC JTC 1, May 1999.

[117] Pavel Zezula, Federica Mandreoli, and Riccardo Martoglia. "Tree signatures and unordered XML pattern matching." *Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Springer-Verlag Berlin Heidelberg, 2004, pp. 122–139.

[118] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. "On supporting containment queries in relational database management systems." *ACM SIGMOD Record* vol. 30, no. 2 (2001), pp. 425–436.

[119] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. "XSEED: Accurate and fast cardinality estimation for XPath queries." *Proc. 22nd Int. Conf. Data Engineering (ICDE)*. IEEE. 2006, p. 61.

[120] Yuping Zhang, Xinjun Wang, and Ying Zhang. "A labeling scheme for temporal XML." *Proc. 2009 Int. Conf. Web Information Systems and Mining (WISM)*. IEEE, 2009, pp. 277–279.