# Index-Assisted Hierarchical Computations in Main-Memory RDBMS

Robert Brunel* Norman May† Alfons Kemper*

* Technische Universität München, Garching, Germany
† SAP SE, Walldorf, Germany
* *firstname.lastname*@cs.tum.edu † *firstname.lastname*@sap.com

## ABSTRACT

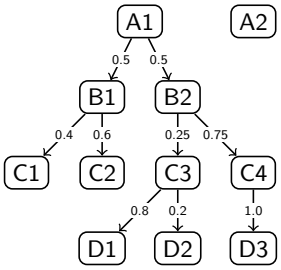
We address the problem of expressing and evaluating computations on hierarchies represented as tables within relational databases. Engine support for such computations is very limited today, and so they are usually outsourced into stored procedures or client code. A recent proposal adds data model and language means to conveniently represent and work with hierarchies. On that basis we introduce a concept of hierarchical grouping on relational algebra level, add concise syntax to SQL to express a class of useful computations, and discuss physical algebra operators to evaluate them efficiently by exploiting available indexing schemes. This extends the versatility of RDBMS towards a great many use cases dealing with hierarchical data.

## 1. INTRODUCTION

In business and scientific applications hierarchies appear in many scenarios: organizational or financial data, for example, is typically organized hierarchically, while the sciences routinely use hierarchies in taxonomies, say for animal species. In the underlying RDBMS they are stored in *hierarchical tables* and "indexed" using some relational tree encoding [1, 2]. If we look at typical queries, especially with an eye on challenging analytic applications, we see that hierarchies serve mainly two purposes. The first is structural *pattern matching*, that is, filtering and matching sets of rows (say, from a fact table) based on their positions in a hierarchy. The second is *hierarchical computations*: propagating measures and performing aggregation-like computations alongside the hierarchical dimension. To address both purposes on RDBMS level, we need to solve two essential challenges: how can a user express a task at hand adequately and concisely in SQL (*expressiveness*)? —and: how can the engine process these SQL queries efficiently (*efficiency*)? Regarding pattern matching queries, both can be considered adequately solved, as they boil down to straightforward filters and structural joins using hierarchy predicates such as "is-descendant" [3], and techniques for appropriate indexes and join operators are well-studied [4, 5, 6, 7]. The same cannot be said of hierarchical computations.



HT : {[ID, Node : NODE$^H$, Weight]}




| ID | Node | Weight |
|---|---|---|
| 'A1' | A1 | NULL |
| 'A2' | A2 | 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 |

**Figure 1: A *hierarchical table* according to [3].**

For the purpose of computations, some hierarchy nodes are dynamically associated with values to be propagated or aggregated, and possibly filtered. In analytic applications, computations like this have always been a routine task, albeit for a comparatively simple case: Dimension hierarchies are typically modeled using denormalized tables with dedicated columns per hierarchy level, such as Day–Month–Quarter–Year or City–State–Country–Continent. On such tables, certain common computations can be expressed and evaluated simply using SQL's grouping mechanisms (`GROUP BY`, in particular `ROLLUP` [8]) and the standard aggregation functions `SUM` et cetera. However, this is insufficient for more complex cases, in particular when the hierarchy is not organized into levels but exhibits an irregular structure—where nodes on a level may be of different types—and arbitrary depth, or when the computations are more involved than simple rollups. Consider the hierarchy in Fig. 1. Suppose we wanted to compute weighted sums of some values attached to the leaves—how could we state a rollup formula incorporating the edge weights? Expressing computations like this in SQL is often exceedingly difficult. One standard tool that comes to mind are recursive table expressions (RCTEs). However, while in theory they can handle arbitrary iterative computations and therefore general graph traversals [9], more complex aggregation-like computations tend to result in extremely intricate, inherently inefficient query statements (cf. the discussion in [3]). Lacking specific RDBMS support, today users necessarily resort to client code or stored procedures for such tasks. These workarounds are unsatisfactory not only from an expressiveness point of view, they also ignore the known hierarchy structure and are thus handicapped in terms of efficiency.

Our aim is to address the open issues of expressiveness and efficiency regarding complex computations on arbitrary irregular hierarchies, by enhancing the RDBMS backend following the spirit of [3, 2]. Our foundation are the data model and language constructs from [3], which allow the user to conveniently define arbitrary hierarchies via DDL statements and express queries on them. This opens up new opportunities; in particular, the backend becomes aware of the hierarchy structure and can rely on powerful indexing schemes [2] for query processing. We first examine the concept of *hierarchical grouping* on relational algebra level (Sec. 3). Then we proceed to propose SQL extensions that enable the user to express hierarchical computations in the first place (Sec. 4), based on the idea of enhancing SQL's windowed tables. Their efficient evaluation requires special-purpose physical algebra aggregation operators at the backend (Sec. 5). Besides these operators, we discuss alternative conventional approaches and include them in our analysis and performance evaluation (Sec. 6).

## 2. MOTIVATION

Our starting point is the **hierarchical table**, a representation of hierarchical data in a relational table. More specifically, we assume a table that encodes the structure of a hierarchy, such that one table tuple (row) represents one hierarchy node. There are lots of options regarding the actual encoding to use; see [2] for a recent overview, as well as [1]. We use the *hierarchical table* model [3], which conveniently hides the fiddly details of a specific encoding through an abstract data type `NODE`. The underlying topology is a forest of ordered, rooted, labeled trees. The label of a node are the associated row's fields. For trees a $1:1$ association between each node and its incoming edge can be made, so a field can be interpreted as either a node or an edge label.

Throughout the paper, we consider the simple example table HT of Fig. 1, where we view Weight as an edge label. The Node attribute encapsulates the hierarchy structure and is backed by a hierarchy index $H$. We assume the abstract index data structure supports a basic set of query primitives for axis checks such as "$u$ is a descendant of $v$", as described in [2]. Our pseudo-code notation for these binary predicates is "$H$.is-descendant$(u, v)$". The particular primitives we require are is-before-pre and is-before-post, is-descendant, and is-preceding. Note that, while we rely on the `NODE` abstraction for ease of presentation, it could be replaced by any hand-crafted labeling or indexing scheme that affords the said primitives (e. g., the one of [4]).

A **hierarchical computation** propagates and accumulates data—usually numeric values—alongside the hierarchy edges. Data flow can happen either in the direction towards the root (*bottom up*) or away from the root (*top down*, matching the natural direction of the edges). In an analytic scenario, HT may arrange products into a hierarchy of product groups (dimension hierarchy), and a sales table F (fact table) may associate each sale with a specific product, i. e., a leaf of HT. A canonical task in such scenarios known as *rollup* is to sum up the revenue along the hierarchy bottom up and report these sums for certain product group nodes currently visible in the user interface, say, the two uppermost hierarchy levels.

This illustrates three common characteristics: First, the computation input is generally dynamic, i. e., the result of an arbitrary subquery that associates some hierarchy nodes

Inp1

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

(a)

Out1

| Node |
|---|
| A1 |
| B1 |
| C1 |

(a)

InpOut1

| Node | Value |
|---|---|
| C1 | 100 |
| C2 | 200 |
| B1 | 10 |
| D1 | 1000 |
| D2 | 2000 |
| D3 | 3000 |
| A1 | NULL |

(b)

Result1

| Node | $f(t)$ |
|---|---|
| C1 | 100 |
| C2 | 200 |
| B1 | 310 |
| D1 | 1000 |
| D2 | 2000 |
| D3 | 3000 |
| A1 | 6310 |

(c)

InpOut2

| Node | ID | Weight | Value |
|---|---|---|---|
| C1 | 'C1' | 0.4 | 100 |
| C2 | 'C2' | 0.6 | 200 |
| B1 | 'B1' | 0.5 | NULL |
| 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 |
| A2 | 'A2' | NULL | NULL |

(d)

**Figure 2: Example input/output tables.** — (a) input and output nodes; (b) combined input/output nodes; (c) rollup result; (d) combined input/output with weights

with some values—unlike the static ID and Weight values stored with HT itself. In the analytic scenario, the input may be pre-aggregated measures from F, attached to the leaves via join:

$$\text{HT} \bowtie_{\text{HT.ID=F.HT\_ID}} (\Gamma_{\text{F.HT\_ID; Value}:\texttt{SUM(F.Measure)}}(\text{F}))$$

Second, often only a subset of nodes carry an input value. We call these *input nodes*. In the example only leaves are potential input nodes. Third, input nodes are not always also *output nodes* that after the computation carry a result value we are interested in.

We expect the set of output nodes to be mostly disjunct from the input nodes, and comparatively small. Therefore, the hierarchy table may contain a large number of *intermediate nodes*, which merely provide connectivity between input and output nodes, but are not actually relevant to the computation. This leads us to an important design goal: *The evaluation of a hierarchical computation should not be required to touch (iterate over, or consider in any way) any intermediate nodes.* These nodes should effectively be ignored while preserving connectivity properties between input/output nodes, and their number should be insignificant to overall performance.

Now, suppose we are given table Inp1 from Fig. 2a and want to sum up the values bottom-up along the hierarchy HT (neglecting the edge weights for now), and we are interested in three output nodes given by table Out1. A straightforward way to do this is to consider each tuple $t$ in Out1 and first associate it with all relevant Inp1 tuples $X = \{u \in \text{Inp1} \mid u < t\}$, where $<$ reflects the input/output relationship among tuples and in the bottom-up case means "$u$.Node is a descendant of $t$.Node"; then sum up all Values to obtain the result $f$:

$$f(t, X) = t.\text{Value} + \sum_{u \in X} u.\text{Value}. \tag{1}$$

It is obvious how this can be both expressed in relational algebra and evaluated using a join-group-aggregate technique:

$$\Gamma_{t.*;\ f:\texttt{SUM}(u.\text{Value})}(\text{Out1}[t] \bowtie_{H.\text{is-descendant}} \text{Inp1}[u])$$

We refer to this setting, where input and output nodes are given by separate tables, as *binary hierarchical grouping*. The join-group-aggregate approach is perfectly adequate when the set of output nodes is rather small and mostly disjunct from the input nodes. It also meets our first design goal. However, it bluntly repeats the computation of partial sums for each $t$, and thus fails to meet a second important goal: *The*

*computation for an output node should reuse any previously computed results as far as possible.*

In the bottom-up case of the example, candidates for reuse are the $f$ values of any output nodes that are descendants of the current output node. In the example both $f(\text{B1})$ and $f(\text{C1})$ are useful in computing $f(\text{A1})$. To enable this reuse, the Out1 tuples must be processed in a *topological* order according to $<$, which is the bottom-up case means any order that places a node before any of its parent or ancestor nodes. We say that B1 and C1 *precede* A1 with respect to $<$. The binary hierarchical grouping algorithms we discuss in this paper allow such reuse through topological sorting and achieve both mentioned design goals.

**Recursive Computations**. By reusing preceding computations, we blur the distinction between input and output nodes: in general, the computation of $f$ for a node in Out1 (say B1) consumes certain inputs from Inp1 (C2), reuses certain *immediately preceding* output values (C1), and produces an output value that in turn contributes to the immediately following computation (A1). These ideas lead us to an alternative way to perform a rollup using structural recursion, based on combining Inp1 and Out1 into a single table InpOut1 (Fig. 2b). We apply to each InpOut1 tuple a *recursive* formula for $f$ that is now allowed to make use of the readily computed sums $f$ of any immediately preceding nodes within that same table:

$$f(t, X_t) = t.\text{Value} + \sum_{u \in X_t} f(u, X_u), \tag{2}$$

where $X_t = \{u \in \text{InpOut1} \mid u <: t\}$ collects the tuples corresponding to *immediately preceding* input nodes. For example, $f(\text{B1})$ would be computed simply as $10 + f(\text{C1}) + f(\text{C2})$. This way we obtain table Result1 (Fig. 2c). But instead of performing actual recursion, we apply this approach in a bottom-up fashion. Fig. 2b shows one possible topological order for InpOut1 that allows us to do so. The overall evaluation strategy is: (1) collect both input and output nodes in a table $T$ and associate the input data with the input nodes; (2) topologically sort $T$ according to $<$; (3) iterate over $T$ in that order, applying formula $f$ on the go, feeding the appropriate preceding result values to $f$ and appending the result to each processed tuple; (4) retain only tuples of output nodes in the final result. We refer to this setting as *unary hierarchical grouping*. Recursive expressions thus not only allow us to specify computations in a general and effective way, they also afford a natural and inherently efficient evaluation technique that fully leverages previously computed results. Beyond that, recursive expressions also add expressive power. For example, we can straightforwardly take the edge weights into account in our rollup by including them in table $T$, yielding InpOut2 in Fig. 2d. This makes *every* HT node an input node. Our rollup formula becomes

$$f(t, X) = t.\text{Value} + \sum_{u \in X} u.\text{Weight} \cdot f(u, X_u).$$

For expressing more complex computations like this, a mechanism for structural recursion along the hierarchy is needed. In this paper we extend RDBMS by means to express both binary and (potentially recursive) unary hierarchical computations, and to evaluate them in the most natural and efficient way.

# 3. HIERARCHICAL GROUPING

To be able to express computations on hierarchies, we extend the relational algebra by two operators, one for unary hierarchical grouping and one for binary hierarchical grouping.

## 3.1 Preliminaries

We closely follow the relational algebra notation used in [10] and assume the reader to be familiar with that in the following. Our operators work on bags (denoted $\{\}_b$), though generalizations to other bulk types are conceivable. Both unary and binary grouping rely on a $<$ predicate on the tuples of their input table(s) that reflects the *preceding* relationships among tuples. This predicate is assumed to be a strict partial order: irreflexive, transitive, and asymmetric. As sketched in the example bottom-up rollup of the previous section, it will be defined in terms of the hierarchy underlying the corresponding `NODE` fields, which we discuss in Sec. 3.5. For now, think of $u < t \;:=\; H.\text{is-descendant}(u.\text{Node}, t.\text{Node})$, assuming the fields are named "Node" in both $u$ and $t$. Based on $<$ we derive the notion of *covered* elements: an element $a$ is said to be *covered* by another element $b$, written $a <: b$, if $a < b$ but not $a < c < b$ for any $c$; in other words, no third element fits between them. Given a relational expression $e$, we refer to tuples being covered by a given tuple $t$ with respect to $<$ as its *immediately preceding tuples*, and use the following notational shorthand:

$$e[_\theta t] := \{u \mid u \in e \;\wedge\; u\ \theta\ t\}_b.$$

$e[_< t]$ gives us all preceding tuples, $e[_{<:} t]$ the immediate ones.

Hierarchical computations involve an aggregation-like computation whenever a set of immediately preceding tuples is combined for a current tuple $t$. A *hierarchical aggregation function* $f$ is a function with the signature $\tau_1 \times \{\tau_2\}_b \to \mathcal{N}$ for some types $\tau_1$, $\tau_2$, and $\mathcal{N}$. The current tuple $t$ of type $\tau_1$ is passed as a dedicated first argument. The result type $\mathcal{N}$ will usually be scalar and numeric.

## 3.2 Binary Hierarchical Grouping

Binary hierarchical grouping can be dissected into a join using the $<$ predicate and subsequent unary grouping. Therefore, it is not a new operation to relational algebra but an instance of *binary grouping* $\bowtie\Gamma$ (also known as *groupjoin* [10]). We use a slightly customized variant. Binary hierarchical grouping has the signature $\bowtie\Gamma^{<}_{x:f} : \{\tau_1\}_b \times \{\tau_2\}_b \to \{\tau_1 \circ [x : \mathcal{N}]\}_b$. It consumes two input relations given by expressions $e_1 :: \{\tau_1\}_b$ and $e_2 :: \{\tau_2\}_b$, where $\tau_1$ and $\tau_2$ are tuple types. The strict partial order $<$ needs to be defined on $\tau_2 \times \tau_1$. $x$ is an attribute name, $f$ a hierarchical aggregation function $\tau_1 \times \{\tau_2\}_b \to \mathcal{N}$ for some type $\mathcal{N}$. The operation is defined as

$$e_1 \bowtie\Gamma^{<}_{x:f} e_2 \;:=\; \{t \circ [x : f(t, e_2[_< t])] \mid t \in e_1\}_b.$$

The main twist of hierarchical grouping over the common uses of groupjoin is that the join condition is an inequality predicate and implies a *partial* order. This suggests different, sort-based implementation techniques in physical algebra than the classic equality predicates [11].

## 3.3 Unary Hierarchical Grouping

For our second kind of grouping we introduce a new operator that essentially arranges its input tuples in a directed acyclic graph as determined by $<$, and then performs a structurally recursive computation on that structure.

Let $e :: \{\tau\}_b$ for some tuple type $\tau$; $<$ a comparator for $\tau$ elements that provides a strict partial ordering of the tuples in $e$; $x$ an attribute name; $f$ a hierarchical aggregation

function $\tau \times \{\tau \circ [x : \mathcal{N}]\}_b \to \mathcal{N}$, for some type $\mathcal{N}$. The *unary hierarchical grouping* operator $\hat{\Gamma}$ associated with $<$, $x$, and $f$ is defined as

$$\text{Signature:}\quad \hat{\Gamma}^{<}_{x:f} : \{\tau\}_b \to \{\tau \circ [x : \mathcal{N}]\}_b$$
$$\hat{\Gamma}^{<}_{x:f}(e) := \{t \circ [x : \text{agg}^{<}_{x:f}(e,t)] \mid t \in e\}_b, \text{ where}$$
$$\text{agg}^{<}_{x:f}(e,t) := f(t, \{u \circ [x : \text{agg}^{<}_{x:f}(e,u)] \mid u \in e \wedge u <: t\}_b)$$

The operator extends each given input tuple $t$ in $e$ by a new attribute $x$ and assigns it the result of applying $f$ to $t$ and the group (bag) of immediately preceding tuples. The aggregation function $f$ has a major twist: each grouped preceding tuple $u$ passed to $f$ already carries the $x$ value, and this value is computed by in turn applying $f$ to $u$, in a recursive fashion. Thus, while $f$ itself is not recursive, a recursive computation is encapsulated in $\hat{\Gamma}$'s "implementation". That computation is obviously guaranteed to terminate, since $<$ is a strict partial order.

We reuse the common symbol of standard unary grouping $\Gamma$ for $\hat{\Gamma}$. Both operators are similar in that they form groups of the input tuples, but unlike $\Gamma$, which "folds away" the tuples in each group, $\hat{\Gamma}$ retains every input tuple in the output. The group of any $t$ consists of all tuples $u$ that are covered by $t$. Thus, $<$ dictates the data flow of the operation in terms of which direction the computation proceeds in and the possible paths it takes. In essence, a *variable* recursive computation $f$ is performed on a predetermined recursion tree given by $<$.

## 3.4 Unary Versus Binary Grouping

Generally, there are no restrictions on the function $f$ performing the actual per-tuple computation. It is, however, useful to distinguish a special case that lets us establish a correspondence between $\hat{\Gamma}(e)$ and binary "self"-grouping $e \bowtie\!\!\!\!\!\Gamma\ e$, when $\nu$ is distinct in $e$. Let $t :: \tau$ and $X :: \{\tau \circ [x : \mathcal{N}]\}_b$. Function $f$ is a *simple* hierarchical aggregation function if it is of the form

$$f(t, X) = \text{acc}^{g}_{\oplus}(t, X) := g(t) \oplus \bigoplus_{u \in X} u.x,$$

where $g : \tau \to \mathcal{N}$ extracts or computes a value from a $\tau$ tuple, and a single commutative, associative operator $\oplus$ is used for combining $g(t)$ and all input $x$ values. For example, $g(t)$ may just access $t$.Value, and $\oplus$ could be $+$, $\cdot$, min, or max. If we assume that all $\nu$ values are distinct in $e$ and consider the resulting relation $R = \hat{\Gamma}^{<}_{x:f}(e)$, we find that the following holds for all $t \in R$:

$$t.x = g(t) \oplus \bigoplus_{u \in R[<:t]} u.x = g(t) \oplus \bigoplus_{u \in e[<:t]} g(u).$$

The properties of $f$ thus allows us to cover the recursion through $<$ and obtain a closed form of the expression for $t.x$ based on the tuples in $e$. We can state the following correspondence between $\hat{\Gamma}$ and binary grouping $\bowtie\!\!\!\!\!\Gamma$:

$$\hat{\Gamma}^{<}_{x:\text{acc}^{g}_{\oplus}}(e) = e \bowtie\!\!\!\!\!\Gamma_{<;x:f}\ e \quad \text{with} \quad f(t, X) := g(t) \oplus \bigoplus_{u \in X} g(u).$$

This correspondence means two things: First, if an intended computation $f$ can be algebraically transformed into a simple form, it can be expressed already today—namely as a combination of join and grouping—and evaluated using standard binary grouping as shown. Vice versa, what SQL allows us to express using joins and grouping with one of the common aggregation functions `COUNT`, `SUM`, et cetera roughly corresponds to these simple computations. Second, we may sometimes use $\hat{\Gamma}$-based plans to optimize $\bowtie\!\!\!\!\!\Gamma$-based plans following the above pattern. $\hat{\Gamma}(e)$ can significantly outperform $e \bowtie\!\!\!\!\!\Gamma\ e$, as $e$ need not be evaluated twice, and the $\hat{\Gamma}$ algorithms have simpler logic and allow for more effective pipelining.

At this point it would be interesting to mention basic equivalences that hold; refer to [10] if applicable.

## 3.5 Applying Hierarchical Grouping

We now discuss how to apply $\bowtie\!\!\!\!\!\Gamma$ and $\hat{\Gamma}$ for scenarios on hierarchical tables, as motivated in Sec. 2. First, let us recapitulate these scenarios in the light of the notations from the previous section. $\bowtie\!\!\!\!\!\Gamma$ is given a left and a right input $e_1$ and $e_2$, which intuitively comprise the output and input nodes, while $\hat{\Gamma}$ is given a combined input/output table. For example, we can compute a rollup using Out1 $\bowtie\!\!\!\!\!\Gamma$ Inp1 and $f$ defined as in Eq. 1 from Sec. 2; and we can obtain the same result using $\hat{\Gamma}$(InpOut1) with $f(t, X) := t.\text{Value} + \sum_{u \in X} u.x$ analogously to Eq. 2, then filtering the Out1 nodes.

Both $\bowtie\!\!\!\!\!\Gamma$ and $\hat{\Gamma}$ rely entirely on an abstract $<$ predicate to steer the data flow. These inputs $e_1/e_2$ or $e$ each contain a field of type $\texttt{NODE}^H$ whose values are arbitrarily drawn from an original hierarchical table HT with the associated hierarchy $H$. We assume all `NODE` fields are named $\nu$, which can be achieved through renaming if necessary. Depending on the direction of the intended data flow, we now define $<$ accordingly:

$$\text{top-down:}\quad u < t :\Longleftrightarrow H.\text{is-descendant}(t.\nu, u.\nu)$$
$$\text{bottom-up:}\quad u < t :\Longleftrightarrow H.\text{is-descendant}(u.\nu, t.\nu)$$

This essentially reverses the edges of $H$ for bottom-up computations. $<$ inherits the required partial order properties from is-descendant. We define $<:$ in terms of $<$ as follows:

$$u <: t :\Longleftrightarrow u < t \ \wedge\ \neg \exists\, u' \in e\ .\ u < u' < t. \qquad (3)$$

Fig. 3 illustrates a unary bottom-up computation based on $e = \text{InpOut3}$. The input is missing some intermediate nodes—there are no tuples with $\nu =$ (B1), (C3), or (C4)—and contains two (B2) tuples. We can visualize the resulting data flow by viewing the input relations as a directed graph $G$, the *data flow graph*, whose nodes $V(G)$ are all tuples produced by $e$ in the unary case, or $e_1$ and $e_2$ in the binary case (including unique tuple IDs, so we get a set rather than a bag), and whose edges $E(G)$ are given by $<:$. Can the InpOut3 example be merged with another one? Also, a binary data flow graph illustration would be nice. But too little space :( The $<$ and $<:$ predicates then correspond to reachability and adjacency in $G$.

If we compare $G$ with $H$, we observe that while $H$ is a forest, $G$ is generally an acyclic digraph containing a subset of the nodes in the underlying hierarchy, although some nodes may appear multiple times. Our definition of the edges $E(G)$ effectively retains all edges from $H$ between nodes that are *both* also contained in $V(G)$ (though they are reversed in the bottom-up case), and furthermore adds a minimal number of "interpolating" edges between components of $G$ in such way that the reachability characteristics from $H$ are retained. In our bottom-up example, our graph $G$ has six such edges, providing connections from the (C1)/(C2) tuples to the (A1) tuple through (B1), and from the (D1)/(D2) tuples to both (B2) tuples through (C3)/(C4). Thanks to this we meet our first design goal: the data flow adheres to the structure of $H$ even when intermediate nodes are missing.

The figure also shows the effect of multiply occurring inner nodes in a unary computation: Due to (B2) appearing twice in the input, there are two possible paths in $G$ to (A1) from the

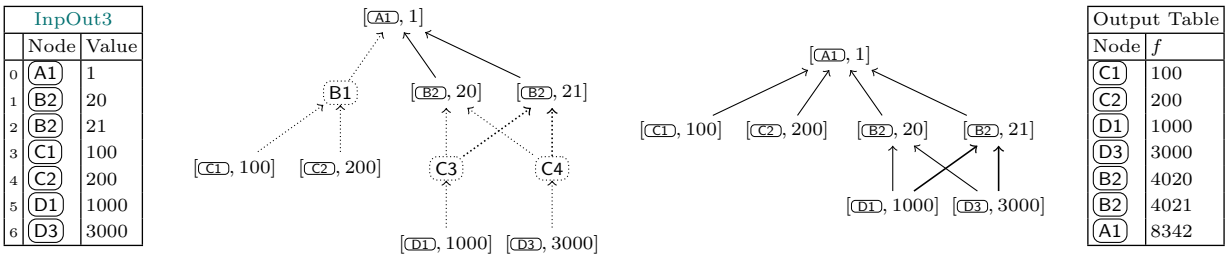



**Figure 3: A bottom-up hierarchical computation.** — (a) the input table, containing node B2 twice and "missing" other nodes; (b) how the data flow graph is inferred; (c) the actual data flow; (d) the output. shrink!

| | | | |
|---|---|---|---|
| *(1) | total Value | ↑ | $\text{acc}^g_+$ with $g(t) := t.\text{Value}$ |
| *(2) | absolute Weight | ↓ | $\text{acc}^g$ with $g(t) := t.\text{Weight}$ |
| (3) | total Value of children | ↑ | $\sum_{u \in X} u.\text{Value}$ |
| (4a) | weighted accumulation | ↑ | $t.\text{Weight} \cdot (t.\text{Value} + \sum_{u \in X} u.x)$ |
| (4b) | | | $t.\text{Value} + t.\text{Weight} \cdot (\sum_{u \in X} u.x)$ |
| (4c) | | | $t.\text{Value} + \sum_{u \in X} u.\text{Weight} \cdot u.x$ |
| *(6) | level | ↓ | $\text{acc}^g_+$ with $g(t) := 1$ |
| *(7) | subtree size | ↑ | $\text{acc}^g_+$ with $g(t) := 1$ |
| (8) | subtree height | ↑ | $1 + \max_{u \in X} u.x$ |
| (9) | degree | ↑ | $\lvert X \rvert$ |
| (10) | Dewey conversion | ↓ | $\langle t.\text{ID} \rangle$ if $X = \{\}_b$, $u.x \circ \langle t.\text{ID} \rangle$ if $X = \{u\}_b$ |
| (11) | Nested Sets conversion | ↑ | $(t.\text{ID}, \bigcup^b_{u \in X} u.x)$ |

Symbols: * simple ↑ bottom-up ↓ top-down

**Figure 4: Example hierarchical computations and corresponding definitions of $\hat{\Gamma}$'s $f(t, X)$.**

D1 and D3 tuples, respectively. Consequently, the D1/D3 tuples each count twice—once per connecting path—into $f(\text{A1})$, yielding 8341. This behavior is inherent to $\hat{\Gamma}(e)$ and intuitive, but it differs from what $e \bowtie\!\!\!\!\!\!\Gamma\, e$ would produce, which may be surprising. Other ways of handling duplicates are possible—in particular, our $\hat{\Gamma}$ algorithm can emulate the behavior of $\bowtie\!\!\!\!\!\!\Gamma$, and vice versa—, but we omit a further discussion due to limited space.

Note again that $G$ is only a conceptual device for visualizing the data flow. Since $<$ and $<:$ is ultimately defined in terms of $H$, this graph is not actually materialized during query evaluation.

**Unary Grouping Scenarios**. While only the $\nu$ field is relevant for $<$, besides $\nu$ the input $e$ to $\hat{\Gamma}$ may also contain further relevant fields for $f$, as well as additional payload fields, such as key attributes for identifying the tuples (e. g., ID in InpOut2). All these are simply carried along. Thus, the *output* table contains the same tuples as $e$, with an added $x$ attribute but otherwise untouched $\nu$ and payload values. From there, the interesting output tuples must be filtered using ordinary selection ($\sigma$), as this is not done by $\hat{\Gamma}$ itself. Note also that the input and output nodes must be appropriately assembled prior to $\hat{\Gamma}$ in the query plan; in particular, since only input nodes are associated with meaningful values for $f$, the corresponding fields of output nodes must be filled with neutral values with respect to the computation—for example, 0 in case of simple sum, 1 in case of product, or simply `NULL` like for InpOut2's Weight and Value fields in Fig. 2. This task is ultimately up to the user writing or the program generating the SQL statements using the constructs we introduce in Sec. 4.

Based on InpOut2 we now consider some meaningful example computations using $\hat{\Gamma}$. Fig. 4 shows the corresponding definitions of $f$; those marked "simple" can straightforwardly be used with $\bowtie\!\!\!\!\!\!\Gamma$ as well. **(1)** is our simple bottom-up rollup of Value. Besides $+$ (total), the $\oplus$ operation could also be $\cdot$, min, or max. Using a degenerate $g(t) := 1$ with $+$, we can simply count the number of preceding tuples in $G$ (cf. Example 7). $\oplus$ could even be $\cup^b$ to simply collect all values in a bag. Taking this even further, a prior duplicate elimination on the input bag could performed (analogously to SQL's `DISTINCT` modifier, as opposed to the default `ALL`); in a bill of materials, for example, we may want to count the number of distinct part types appearing within each part. **(2)** is a top-down counterpart to (1); it multiplies all Weight values of tuples on the root path, associating each tuple with the effective weight of the corresponding node. The bill of materials is another example, where rather than a weight we may have a multiplicity ("how many?") of each part within its respective super-part, and we may want to know, for each part, how often it appears in total within the complete assembly. **(3)** is a degenerate variant of (1) that sums up Value "non-transitively", including only the children of each tuple in the sum rather than all descendants. The same result could be achieved using a simple `IS_CHILD` join plus grouping. **(4a)** and **(4b)** are two variants of summing up Value while taking Weight into account, as motivated in Sec. 2. The (4b) variant does not immediately apply $t$.Weight onto $t$.Value, but defers it until when $t.x$ is incorporated into its parent's $x$ value. Therefore, in addition to $u.x$ the input tuples' $u$.Weight values must be accessed during the computation of $t.x$. While these aggregation functions are not simple, in some cases of weighted accumulation, there is an alternative solution based on *two* simple functions, which basically corresponds to just "multiplying out" the recursive formula according to the distributivity law. For example, the result of (4a) can alternatively be obtained by first applying $\text{acc}^{t.\text{Weight}}_*$ top down, producing absolute weights $t.w$, and then applying $\text{acc}^g_+$ with $g(t) = t.w * t.\text{Value}$ bottom up. **(6)**–**(9)** compute properties of the data flow graph $G$, similar to the standard hierarchy functions `LEVEL`, `SUBTREE_SIZE`, `SUBTREE_HEIGHT`, and `DEGREE`, respectively [3]. If these computations were run on the original hierarchical table ($HT$), they would yield values equivalent to $H.\text{level}(t.\nu)$ et cetera. However, since $\hat{\Gamma}$'s input does not generally contain all of $HT$'s nodes, we cannot use the index functions but have to compute them by hand. The degree computation (9) is a degenerate case: $f$ does not depend on the actual input values, only on the cardinality; like (3), it could be emulated using an `IS_CHILD` join. (10) and (11) compute two alternative hierarchy representations.

**(10)** constructs a path-based hierarchy encoding: it builds a string from the ID values on the root path. **(11)** constructs a nested sets encoding. On InpOut2 this for example associates (C3) with ('C3', {('D1'), ('D2')}).

# 4. EXPRESSING COMPUTATIONS IN SQL

In this section we explore the expressiveness of SQL with respect to hierarchical computations, and we propose concise and intuitive extensions to express *unary* hierarchical grouping and structurally recursive computations directly.

## 4.1 What is possible today?

What SQL today allows us to express—in a reasonably concise and readable statement, that is—corresponds roughly to what we characterized previously as binary computations and certain unary computations using simple aggregation functions. As outlined in Sec. 3.4, these can be stated and naively computed via join/group/aggregate, where the join explodes all involved $<$ pairs in a table. To illustrate a unary bottom-up computation we reuse InpOut2 from Fig. 2. The appropriate self-join predicate is `IS_DESCENDANT` [3]:

```
-- Query 1
SELECT t.Node, t.Value + SUM(u.Value) AS x   -- f(t, X)
  FROM InpOut2 t LEFT OUTER JOIN InpOut2 u
         ON IS_DESCENDANT(u.Node, t.Node)   -- u < t
 GROUP BY t.Node, t.Value
```

Such queries can straightforwardly be translated into plans using $\bowtie\!\!\!\!\!\!\Gamma$. Since the left and right inputs are identical, a rewrite to the more efficient $\hat{\Gamma}$ is possible from that. Filters on input or output nodes can be added as usual. In the following example, the output nodes we are interested in are on levels $\leq 3$, and the input nodes are determined through a join with a table F and an additional Weight filter:

```
-- Query 2
WITH
  HT1 AS (SELECT * FROM HT WHERE LEVEL(Node) <= 3),
  F1  AS (SELECT * FROM HT NATURAL JOIN F WHERE HT.Weight > 3)
SELECT HT1.ID, SUM(F1.Value) AS Result   -- ⊕(g(t))
FROM HT1 LEFT OUTER JOIN F1 ON IS_DESCENDANT(F1.Node, HT.Node)
GROUP BY HT1.*
```

Although this approach works, it lacks conciseness, since conceptually (in terms of language) a table of $<$ pairs must be materialized "manually" prior to grouping. Also, the fact that a bottom-up rollup is being done is somewhat disguised.

Structurally recursive unary hierarchical computations, on the other hand, cannot convincingly be expressed in pure SQL. The only obvious direct approach would be to combine SQL's mechanism for generative recursion, an RCTE, with `GROUP BY`. That would first of all require us to state the computation in an iterative way, starting at the minimal tuples with respect to $<$, then sweeping over $G$ in a breadth-first, set-oriented manner using $<$: joins. But how can we ensure that the `GROUP BY` captures all relevant input nodes within each iteration? We found this to be very challenging: In an irregular hierarchy, the nodes in the starting set are not necessarily all on the same level, to begin with. Moreover, it is non-obvious how the starting condition "$t$ is minimal wrt. $<$" and the join "$u <: t$" could be translated to SQL. The former corresponds to a check whether $t$ is a source node in $G$, the latter to an adjacency check. Since we are working on arbitrary inputs rather than the original hierarchy table HT, the hierarchy index and the `IS_LEAF` and `IS_CHILD` predicates are not helpful. Furthermore, even if these issues could be solved, using `GROUP BY` within an RCTE is forbidden in most existing RDBMS relying on the common semi-naive evaluation strategy (to guarantee stratification, cf. [12] Sec. 3.3.2 and 5.3). All in all, RCTE-based recursion does not seem an adequate approach for the hierarchical computations we target. The current options in standard SQL are to algebraically transform the computation at hand into two or more simple computations, if possible, as outlined in Sec. 3.5, and if that fails, to implement parts of the logic in a SQL scripting language or in the client code.

## 4.2 Windowed Tables and Hierarchies

We now explore SQL syntax extensions for stating unary hierarchical grouping directly. It turns out that the semantics of unary grouping fit very well with SQL's concept of *windowed tables*. First, $\hat{\Gamma}$ retains all tuples from the input $e$ in the output; it does not "fold away" any tuples. Second, the formed groups generally overlap; for example, $e[<u] \subseteq e[<t]$ holds in a bottom-up computation if $u$ is a descendant of $t$. Both these aspects are analogous to how windowed tables work. We therefore proceed to extend this mechanism by *hierarchical* windows.

Let us first review the standard behavior of windowed tables and introduce some terminology. We assume the reader to be somewhat familiar with windowed tables already (refer to e. g. [13]). A standard *window specification* may comprise a *window partition clause*, a *window ordering clause*, and a *window frame clause*. For example:

$w \equiv$ `PARTITION BY` $a$ `ORDER BY` $b$
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
`EXCLUDE NO OTHERS`

The above frame clause matches the implicit default and could be omitted. A simple query applying $w$ may be

```
SELECT SUM(c) OVER w FROM T WINDOW w AS (w)
```

Briefly put, it is conceptually evaluated as follows: (1) table T is partitioned according to the partition clause; (2) each partition is sorted according to the ordering clause; (3) within each sorted partition, each tuple $t$ is associated with a group of T tuples relative to $t$, as determined by the frame clause; (4) the computation (`SUM`) is evaluated for that group and its result appended to the current tuple $t$. The group of tuples relative to $t$ is referred to as its *window frame* $\mathscr{F}(t)$, a subsequence of the current ordered partition. The order imposed by the ordering clause is actually a partial order, since all tuples may not be mutually distinct with respect to the `ORDER BY` fields. Tuples that match in these fields are called *peers* or `TIES`.

We propose to enhance the standard window specification with a new `HIERARCHIZE BY` clause specifying a *hierarchical window*. This clause takes the place of the ordering specification behind the partitioning clause. In other words, hierarchizing replaces ordering; partitioning happens first as usual. Unlike window ordering, which turns each partition into a (partially) ordered sequence, hierarchizing turns each partition into a acyclic digraph—the data flow graph $G$ from Sec. 3.5. We begin our discussion with the simplest possible hierarchical window specification (and without any partitioning) on our example table InpOut2 from Fig. 3:

```
SELECT SUM(Value) OVER (HIERARCHIZE BY Node BOTTOM UP)
  FROM InpOut2
```

The clause determines the `NODE` field and with it the corresponding hierarchy index $H$, as well as the direction of the computation (`TOP DOWN` or `BOTTOM UP`), giving us all information we need to construct the data flow graph $G$. Since there is no explicit frame clause, the default from above applies.

| $t$ | $x$ | $\mathscr{F}(t)$ | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| [C1, 100] | 100 | × | | | | | | |
| [C2, 200] | 200 | | × | | | | | |
| [D1, 1000] | 1000 | | | × | | | | |
| [D3, 3000] | 3000 | | | | × | | | |
| [B2, 20] | 4020 | | | × | × | × | | |
| [B2, 21] | 4021 | | | × | × | | × | |
| [A1, 1] | ... | × | × | × | × | × | × | × |

Symbols: × included for (a)
× included for (a) and (b)

**Figure 5: Hierarchical Window Frames** — per tuple $t$ = [Node, Value], for a bottom-up hierarchical window `W` and expression (a) `SUM(Value)` and (b) `RECURSIVE (Value + SUM(x) OVER w) AS x` on InpOut3.

Unlike window ordering, which places all tuples for which neither $u < t$ nor $t < u$ hold into $t$'s peer group ($<$ being defined according to the ordering clause), hierarchizing uses a modified concept: $t$'s *peers* are all tuples with equal $\nu$ values. A tuple $u$ from the current partition can therefore be related in four distinct ways to the current tuple $t$:

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

Category (d) is always excluded from the window frame—it corresponds to tuples where $\nu$ is `NULL` or on the *preceding* or *following* hierarchy axis. We now have to reinterpret three concepts from windowed tables accordingly:

(a) `PRECEDING` tuples (b) `FOLLOWING` tuples (c) `TIES`

This reinterpretation allows us to reuse the window frame clause without any modifications. These terms are intuitive in the sense that $u$ precedes/follows $t$ with respect to a traversal of the data flow graph $G$. They are not to be mixed up with *preceding* and *following* hierarchy axes: in the bottom-up case, `PRECEDING` tuples correspond to descendants and `FOLLOWING` tuples to ancestors of $\nu$.

**Example**. Our above example query applies a minimal bottom-up hierarchical window specification and the default frame clause to the table of Fig. 3 and computes a simple rollup. Fig. 5 shows the resulting frames for each tuple. The tuples $t$ are rearranged in an appropriate bottom-up order. A × marks which tuples belong to $\mathscr{F}(t)$. Evaluating `SUM(Value)` for each of these frames produces a result equivalent to the Query 1 from Sec. 4.1. Thanks to the way $G$ is derived, our reinterpretation of `PRECEDING`/`FOLLOWING`, and the default window frame clause, simple rollups like this "just work" as expected, and this is arguably the most concise and straightforward way of expressing them.

SQL allows window aggregation to be restricted by filters (`FILTER`). This handy feature allows us to equivalently state Query 2 from Sec. 4.1 as follows:

```
SELECT ID, x
  FROM (
    SELECT ID, Node,
           SUM(Value) FILTER (WHERE Weight > 3) OVER w AS x
      FROM HT NATURAL LEFT OUTER JOIN F
    WINDOW w AS (HIERARCHIZE BY Node BOTTOM UP)
) WHERE LEVEL(Node) <= 3
```

Check. Equivalent only if F is distinct!? Show rewriting from $\bowtie^{\curlywedge}$ to $\hat{\Gamma}$ and vice versa; in this case, $\bowtie^{\curlywedge}$ may be superior due to pre-grouping of F optimization.

## 4.3 Recursive Expressions

Our extensions thus far allow us to express unary hierarchical grouping with *simple* aggregation functions. To support more complex computations, we need a way to specify structurally recursive expressions. We propose to reuse the SQL keyword `RECURSIVE` for this purpose. It can be wrapped around an expression containing a window function (`OVER`):

```
SELECT Node, RECURSIVE INT (Value + SUM(x) OVER w) AS x
  FROM InpOut2 WINDOW w AS (HIERARCHIZE BY Node BOTTOM UP)
```

This makes a field `x` of type `INT` accessible within any contained window function, in this case `SUM`, and thus provides a way to recursively access the computed value of the surrounding expression for any tuple in the window frame. The general form is "`RECURSIVE` *type* `(`*expr*`) AS` *c*". The following syntactic rules apply: First, *expr* must contain one or more window function expressions of the form "$expr_i$ `OVER` $w_i$". If there is more than one, all windows $w_i$ must be mutually compatible in the sense that the window partition and `HIERARCHIZE` clauses match (same `NODE` field and direction).

Second, the maximal window frame of each window $w_i$ is restricted: `EXCLUDE GROUP` is enforced, and only *immediately* preceding tuples can be included in the frame:

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

This is the implicit default for any window frame within a recursive expression. It in particular ensures that the window frame will never contain the `CURRENT ROW`, any `TIES`, or any `FOLLOWING` tuples. If any of those were contained in the frame, any appearance of the recursively computed field $c$ within *expr* would be a circular self-reference. It is conceivable to give the user more control over the window frame, but we do not consider that in this paper.

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

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

According to SQL's rules, mentioning $c$ outside the value expression $expr'$ implicitly accesses the current row, which is forbidden, while mentioning $c$ within a window function (within $expr'$) implicitly accesses the *frame row* (`FRAME_ROW`), which thanks to our restrictive window frame can only be a preceding tuple for which the $c$ value is already available. While this behavior is what is usually intended and quite convenient, it is possible to override the implicit frame row access using a *nested window function* to refer to the current row: *AGG*`(...VALUE_OF(`$c$ `AT CURRENT_ROW)...) OVER` $w$. This is forbidden for $c$, but allowed for any other field.

Note that the type of $c$ must be specified explicitly, since it cannot generally be inferred from the expression non-ambiguously. Automatic type deduction in certain cases is a possible future extension.

**Examples**. In Fig. 5, the more restrictive window frames for the recursive variant of the expression are indicated by ×. The second column shows the `x` values of the frame rows that are accessed within the expression.

Fig. 6 provides SQL translations for our example computations on InpOut2, based on two windows:

```
WINDOW td AS (HIERARCHIZE BY Node TOP DOWN),
       bu AS (HIERARCHIZE BY Node BOTTOM UP)
```

Simple computations like (1) can now be stated using either an ordinary or a `RECURSIVE` expression; both alternatives are given in Fig. 6. The first version of (2) uses a hypothetical `PRODUCT` aggregation function, which is curiously missing from standard SQL. The second version works around that via recursion. It aptly takes advantage of `FIRST_VALUE` yielding `NULL` for a root node in $G$ due to the window frame being empty. To understand the example, note that for a top-down recursive computation on a tree, the window frame can be

```
(1) SUM(Value) OVER bu
    RECURSIVE INT(Value + SUM(x) OVER bu) AS x
(2) PRODUCT(Weight) OVER td  -- non-standard
    RECURSIVE INT(Weight * NVL(FIRST_VALUE(x) OVER td, 1)) AS x
(3) RECURSIVE INT(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
(6) COUNT(*) OVER td
    RECURSIVE INT(NVL(FIRST_VALUE(x) OVER td, 0) + 1) AS x
(7) COUNT(*) OVER bu
    RECURSIVE INT(NVL(FIRST_VALUE(x) OVER td, 0) + 1) AS x
(8) RECURSIVE INT(1 + MAX(x) OVER bu) AS x
(9) RECURSIVE INT(COUNT(x) OVER bu) AS x
(10) RECURSIVE VARCHAR(
       NVL(FIRST_VALUE(x) OVER td, '') || '/' || ID) AS x
(11) RECURSIVE ARRAY(ROW(ID, ARRAY_AGG(x))) AS x
```

**Figure 6: SQL expressions corresponding to Fig. 4.**

either empty or contain one tuple, the parent (plus any peers in case of non-distinct Node values). To perform weighted rollup as in (4a–c), one may come up with a false attempt using simple computation: `SUM(Weight*Value) OVER bu`. This would apply the Weight only to the each tuple's Value, rather than the accumulated value, which is not what we intend; we need structural recursion. (4a–c) show the three recursive expressions for the weighted accumulations from Fig. 4. We could bring (4b) into a form similar to (4c) using a nested window function to access the appropriate Weight of the current row within the `SUM` expression:

```
RECURSIVE (Value +
   SUM(VALUE_OF(Weight AT CURRENT_ROW) * V) OVER w
) AS V
```

(10) uses the same technique as (2) to construct a path string. Although (11) could in principle be translated using advanced SQL:99 features—row value constructors and arrays—, the type of the expression depends on the structure of the hierarchy itself, which is not supported.

Overall, $f(t, X)$ formulas translate quite naturally into SQL. But what if we need to go beyond SQL's simple standard aggregate functions `SUM` et cetera? One option is to use `ARRAY_AGG` to collect data from the immediately preceding tuples in an array, and then pass that array to a user-defined function. This way arbitrary $f(t, X)$ functions can be plugged in. Another option is to implement a user-defined aggregate function, which is possible in many RDBMS using proprietary extension mechanisms.

Probably need to elaborate more on when unary vs. binary are needed, in which use cases which one is better, how the translation and rewriting of the queries works.

# 5. PHYSICAL ALGEBRA OPERATORS

## 5.1 Overview

In this section we discuss evaluation strategies for $e_1 \bowtie^{\Gamma} e_2$ and $\hat{\Gamma}(e)$ for given $<$, $x$, and $f$. Regardless of the definition of $<$, there are two *generic approaches* that can always be used. Binary grouping can be dissected into a left outer join with subsequent unary grouping:

$$e_1 \text{ generic-}\bowtie^{\Gamma}_{x:f}\ e_2 := e_1 \leftarrow \text{Sort}_<(e_1);\ e_2 \leftarrow \text{Sort}_<(e_2);\ \Gamma^{\text{sort-based}}_{t.*;\,x:f}(\rho_t(e_1)\ \bowtie^{\text{sort-based}}_{u<t}\ \rho_u(e_2))$$

As discussed in Sec. 3.4, the same approach works for unary grouping if $f$ is simple:

$$\text{generic-}\hat{\Gamma}^{<}_{x:f}(e) := e \text{ generic-}\bowtie^{\Gamma<}_{x:f}\ e \text{ if } f = \text{acc}^{\oplus}_g.$$

For non-equi-join conditions sort-based algorithms for join and subsequent grouping are a natural choice. (Refer to [14] for common techniques.) Note that the join algorithm must be able to deal correctly with $<$ being a *partial* order. The main virtue of these generic approaches is the minimal implementation effort, since they are based on standard operators. The resulting join-group-aggregate plans are well-studied and there are some optimization opportunities, like employing a groupjoin operator [10], eliding the sorting step if the needed order can already be established in the input, or taking advantage of the sort order being retained in the output. While [10] discusses mainly a hash-based equi-groupjoin implementation, a sort-based implementation is required in our case (see [11]). On the downside, the generic approaches can handle only simple computations. They also fail to satisfy our design goals from Sec. 2: a large intermediate result is conceptually produced by the join, and all $<$ join pairs, rather than just the $<:$ pairs, are touched during query evaluation (i. e., materialized at least in processor registers). Results of preceding tuples are not reused for their $<:$ successors.

We therefore focus on *hierarchy-aware approaches* when $\bowtie^{\Gamma}$ and $\hat{\Gamma}$ are used for hierarchical computations. In this case, $<$ operates on `NODE` fields $\nu_1$ and $\nu_2$ associated with a hierarchy index $H$, which can and should be leveraged in the evaluation. A straightforward improvement over generic-$\bowtie^{\Gamma}$ is to use a *hierarchy merge join*, a structural join similar to a merge join which requires the inputs $e_1$ and $e_2$ to be in sorted preorder and retains the order of either $e_1$ or $e_2$ in its output. By leveraging the sorted inputs, a runtime complexity of $\mathcal{O}(|e_1| + |e_2| + |e_1 \bowtie e_2|)$ is achieved. (The basic algorithm has been described by Al-Khalifa et al. [6] under the name *stack-tree join*.) We refer to this approach as generic-hierarchy-$\bowtie^{\Gamma}$.

Beyond the generic approaches, we propose four specialized physical operators: hierarchy-$\hat{\Gamma}$ for general (non-simple) unary grouping and hierarchy-$\bowtie^{\Gamma}$ for binary grouping, each in a bottom-up and a top-down variant. Their inputs are required to be sorted in pre-order (top-down case) or post-order (bottom-up case). They leverage the ordered input and the index $H$, and thus require only a single pass through the input tables. A particularly important observation is that *adjacent subtrees will appear next to each other in the ordered input.* Recall our definition of adjacency in $G$ in terms of reachability (Eq. 3 in Sec. 3.5). If we find that $u < v$ holds for two subsequent tuples $u$ and $v$ in the input, we know that $u <: v$. This means our order-based algorithms never have to actually check for adjacency; $<$ is sufficient.

Note that while sorting steps via $H$.is-before-pre$(\nu_1, \nu_2)$ (or -post) are not cheap and break the pipeline, they can often be elided if a *hierarchy index scan* can be employed in the query plan to enumerate the input in the required pre- or post-order in the first place, or if the order can be reused from a previous order-preserving operator. Once the data is sorted, this may also be beneficial to other operators such as the mentioned hierarchy merge join. [Todos] [Pre-Post Rearrange operator]

For ease of presentation, we describe the pseudo code of $\bowtie^{\Gamma}$ and $\hat{\Gamma}$ in terms of $f(t, X)$, reusing the concept from relational algebra level. That is, we collect, for each tuple $t$, the appropriate bag $X$ of preceding tuples and feed them

into $f(t, X)$ to obtain $t.x$. In an actual implementation, this buffering of tuples in $X$ explicitly should be avoided, as we discuss in Sec. 5.4. We furthermore assume a push-based pipelining query execution [15], where the operator performs some processing on each incoming tuple $t$ and then constructs a result tuple and yields it to its parent operator. The task of the $\bowtie\Gamma$ and $\hat{\Gamma}$ algorithms then is to effectively issue the following call sequence for each $t$:

$X \leftarrow \emptyset$
$X \leftarrow X \cup \{u\}_b$ for each appropriate "input" tuple $u$
**yield** $t \circ [x : f(t, X)]$

During execution, our algorithms use stacks to manage previously seen tuples and corresponding computation results (as represented by the bag $X$) to be reused, which allows them to achieve our design goals while minimizing the required memory for intermediate results.

Possibly rewrite is-descendant to is-before-pre/-post checks.

## 5.2 Unary Hierarchical Grouping

Fig. 1 shows both the top-down and the bottom-up variants of hierarchy-$\hat{\Gamma}$. They consume one sorted input and retain the exact order in the output. We initialize the stack $S$ with a sentinel tuple $\top$ whose Node is a pseudo node that is on neither the *preceding* nor the *descendant* axis relative to any other node. The outer for loop first pre-groups incoming tuples with equal $\nu$ values, which can be omitted if the $\nu$ is known to be distinct in the input. The "collect input" block maintains the stack and collects the relevant input tuples $X$ for the current group $G$. For each tuple $t$ in the group, the inner for loop then finalizes the computation, constructs and yields an output tuple and places it on the stack together with $X$ for potential reuse.

Regarding "collect input", consider first the bottom-up case: Since the input $e$ is in post-order, previously processed tuples on $S$, if any, will be on the *descendant* and *preceding* axes relative to $\nu$, in that order when viewed from the top of stack; upcoming tuples from $e$ will be on the *ancestor* or *following* axes. Therefore, at the beginning of each iteration the immediately preceding tuples $X = \{u_k, \ldots, u_1\}_b$ we need for $G$ are conveniently placed on the upper part of $S$. The "collect input" block simply collects them off the stack, since they will no longer be required in further iterations. In contrast, any tuples waiting below $u_1$ on $S$ are not relevant to the current group $G$, but may be consumed later on (by ancestors of $\nu$). Note that having placed $\top$ onto the bottom of $S$ saves us from explicitly checking $S$ for emptiness anywhere and thus reduces branching in the algorithm—since the pseudo node is never a descendant of any node, $S$ cannot become empty.

In the top-down case, at each iteration $S$ may contain obsolete *preceding* tuples (up to one sibling of $\nu$), then a group of relevant *ancestor* tuples, then further possibly-still-relevant ancestors, in that order when viewed from the top. We first pop the obsolete *preceding* tuples. Due to the tree structure we then know that there may be at most one group of immediately preceding ancestors of $\nu$ to incorporate into $X$ (or at most one tuple if $\nu$ is distinct in $e$). Unlike in the bottom-up case, however, we cannot pop those entries immediately, since they may still be needed for upcoming input tuples.

Regarding tie handling, the algorithm as shown has `EXCLUDE GROUP` behavior. For simple computations, the other variants (`EXCLUDE CURRENT ROW`, `NO OTHER`, or `TIES`) are applicable. Supporting them requires modifications to how a group $G$ is handled by the second for loop. As they are straightforward, we omit a detailed discussion.

**Algorithm 1:** hierarchy-$\hat{\Gamma}^{\nu}_{x\,:\,f}(e)$

**Input:** $e : \{\tau\}_b$, where $\tau \leq [\nu : \text{Node}^H]$; ordered by $\nu$
**Output:** $\{\tau''\}_b$, where $\tau'' := \tau \circ [x : \tau']$; same order

```
S : Stack ⟨[u : τ'', X : {τ''}_b]⟩, initially S ← ⟨[⊤, ∅]⟩
for each distinct ν range G in e
    X : {τ''}_b;  X ← ∅
    ⟨collect input⟩*
    for t ∈ G
        yield t' ← t ∘ [x : f(t, X)]
        S.push([t', X])
```

```
*⟨collect input⟩ — bottom up:
while H.is-descendant(S.top().u.ν, ν)      // always true for ⊤
    [u, X_u] ← S.pop()                     // u < t: consume u
    X ← X ∪ {u}_b                          // leverage X_u if possible!
```

```
*⟨collect input⟩ — top down:
while H.is-preceding(S.top().u.ν, ν)
    S.pop()
[u, X_u] ← S.top()
if H.is-descendant(ν, u.ν)                 // always false for ⊤
    for [u, X_u] ∈ S range for u.ν
        X ← X ∪ {u}_b                      // leverage X_u if possible!
```

## 5.3 Binary Hierarchical Grouping

Alg. 2 shows bottom-up and top-down hierarchy-$\bowtie\Gamma$, which is essentially a merge-based hierarchy groupjoin combining a left outer join on the descendant (bottom-up) or ancestor (top-down) axis with grouping logic. The left input is pipelined through, so its order is retained in the output. Due to the push-based query execution model we assume, we need to access the right input via an iterator interface, which requires buffering. Two stacks are used: $S_1$ collects previously processed nodes $\nu$ from the left input $e_1$ together with the corresponding collections $X$ of right input tuples, which can potentially be reused in future iterations. $S_2$ collects previously stashed right input tuples that may still become relevant as join partners. We again initialize $S_1$ with a sentinel (to do). At each iteration we first check whether $t_1.\nu$ matches the previous left input tuple; in this case, we can reuse its $X$ as is. Otherwise, we assemble $X$ for $t_1$, which again differs for the two cases. The final while loop collects further relevant tuples from the right input $e_2$ and either adds them straight to $X$ (descendants) or stashes them onto $S$ (post-order successors). Finally, an output tuple is constructed and yielded, and the assembled $X$ is pushed onto $S_1$ for potential reuse.

In the bottom-up case (postorder input), "collect input" consists of two steps: The first loop removes all relevant $S_1$ entries—the descendants of $\nu$—and merges the corresponding tuple collections into $X$. Note that if the is-before-post check in the loop condition is false, we know that uppermost tuple on $S_1$ is a descendant of $\nu$. The second loop removes relevant tuples from $S_2$ and incorporates them into $X$. The top-down case (preorder input) works analogously, but in addition we first have to pop entries on the *preceding* axis from $S_1$ that are no longer relevant.

## 5.4 Implementation Notes

As mentioned, an actual implementation of our operators would not actually materialize the $X$ bags as in the pseudo

**Algorithm 2:** $e_1$ hierarchy-$\ltimes^{\nu}_{x:f}$ $e_2$

**Input:** $e_1 : \{\tau_1\}_b$ and $e_2 : \{\tau_2\}_b$, where $\tau_1, \tau_2 \leq [\nu : \text{Node}^H]$
$e_1$ and $e_2$ ordered by $\nu$
**Output:** $\{\tau_1 \circ [x : \tau']\}_b$, same order as $e_1$

$S_1 : \text{Stack}\,\langle[\nu : \text{Node}^H, X : \{\tau_2\}_b]\rangle$
$S_2 : \text{Stack}\,\langle\tau_2\rangle$
$X : \{\tau_2\}_b$
**for** $t_1 \in e_1$ (let $\nu := t_1.\nu$)
  **if** $S_1 \neq \langle\rangle \wedge \nu = S_1.\text{top}().\nu$
    $X \leftarrow S_1.\text{top}().X$
    **yield** $t_1 \circ [x : f(t_1, X)]$
    **continue**
  $X \leftarrow \emptyset$
  $X \leftarrow \langle$collect input$\rangle$*
  **while** $t_2 \leftarrow$ current from $e_2$ (if any)
    **if** $H$.is-descendant$(t_2.\nu, \nu)$/$H$.is-descendant$(\nu, t_2.\nu)$
      $X \leftarrow X \cup \{t_2\}_b$
    **else if** $H$.is-before-post/pre$(t_2.\nu, \nu)$
      $S_2$.push$(t_2)$
    **else**
      **break**
    advance $e_2$
  **yield** $t_1 \circ [x : f(t_1, X)]$
  $S_1$.push$([\nu, X])$

*$\langle$collect input$\rangle$ — bottom up:
  **while** $S_1 \neq \langle\rangle \wedge \neg H$.is-before-post$(S_1.\text{top}().\nu, \nu)$
    $X \leftarrow X \cup S_1.\text{pop}().X$
  **while** $S_2 \neq \langle\rangle \wedge H$.is-descendant$(S_2.\text{top}().\nu, \nu)$
    $X \leftarrow X \cup \{S_2.\text{pop}()\}_b$

*$\langle$collect input$\rangle$ — top down:
  **while** $S_1 \neq \langle\rangle \wedge H$.is-before-pre$(S_1.\text{top}().\nu, \nu)$
    $S_1$.pop()
  **if** $S_1 \neq \langle\rangle \wedge H$.is-descendant$(\nu, S_1.\text{top}().\nu)$
    $X \leftarrow X \cup S_1.\text{top}().X$
  **while** $S_2 \neq \langle\rangle \wedge H$.is-descendant$(\nu, S_2.\text{top}().\nu)$
    $X \leftarrow X \cup \{S_2.\text{pop}()\}_b$

code. Rather, the query compiler would generate specific code to perform the computation $f$ of the given query on the fly as far as possible, without moving any tuples out of the processor registers. In the pseudo code, $X$ is involved in four operations:

| | |
|---|---|
| ① $X \leftarrow \emptyset$ | reset to "empty" |
| ② $X \leftarrow X \cup \{u\}_b$ | add an input tuple $u : \tau$ |
| ③ $X \leftarrow X \cup X'$ | merge |
| ④ $f(t, X)$ | compute the result for tuple $t$ |

Operation ③ is used by the binary operators when reusing a bag from a previous left input tuple (add algorithm line numbers!). $X$ and these three operations can be viewed as a stateful abstract data structure representing a (partial) computation of $f$. During execution of $e_1 \ltimes e_2$ or $\hat{\Gamma}(e_1)$, one $X$ instance per tuple $t \in e_1$ is created. In a modern, bytecode-emitting query compiler, all code could be compiled in place into the algorithms, so this is a zero-cost abstraction. Consider our Example (3) from Fig. 4, where $f = \text{acc}^g_+(t, X)$. Here, the state of $X$ would be a simple partial sum $x$, and the three operations map to

$$① \; x \leftarrow 0, \qquad ②/③ \; x \leftarrow x + u.x, \qquad \text{and } ④ \; x.$$

For the non-simple Example (4c), the state and operations remain the same except that ② becomes $x \leftarrow x + u.\text{Weight} \cdot u.x$. To support all SQL features, these operations must handle additional details, such as `NULL` values and potentially `FILTER` or `DISTINCT`. For example, for the expression `SUM(t.a*t.b)`, the code for $f(t, X)$ encapsulates extracting $t.a$ and $t.b$ from $t$, computing their product, checking whether this value is null, and if not, adding it to the sum. Furthermore, note that the way the two unary operators are designed, the full $X_u$ state is available for any $u$ in operation ②, which is important for some computations. For example, the expression `COUNT(DISTINCT a)` requires a duplicate elimination step, which can also be handled on the fly: the state of $X$ then is a handle to a hash-based set of `a` values, and operations ② and ③ leverage that to construct the combined set efficiently.

Finally, it is also conceivable to have users plug in custom code for the three operations. Similar extension mechanisms are provided for user-defined aggregate functions in many RDBMS [8].

Make an example referring to the data flow graph figure.
Discuss asymptotics.

# 6. EXPERIMENTS

## 6.1 Setup

To explore the performance of our proposed operators, we use a standalone, single-threaded execution engine based on a push-based physical algebra. Base data is stored column-wise, whereas operators pass tuples through the pipelines in row format. The framework allows us to hand-craft query plans, which through careful use of C++ template mechanisms are compiled into bytecode fragments where the operator boundaries between pipelines vanish; thus, friction loss through the algebra is eliminated, and register locality and low-level optimization opportunities are maximized. We found the resulting code to be comparable to what modern engines such as HyPer and HANA Vora could emit.

Compiler, Processor, Operating System

[Experiments Setup] Our **hierarchy table** is similar to HT from Fig. 1 with the schema

$$HT : \{[\underline{\text{Node}} : \texttt{NODE}, \text{ID} : \texttt{CHAR(8)}, \text{Weight} : \texttt{TINYINT}]\}$$

Cluster by Node or by ID? The table is clustered by its primary key Node in preorder. Each node is labeled with a unique ID. We give Weight a small domain and populate it with random integers drawn from $[0, 32[$. If HT were a bills of materials, Weight might for example be the part type (node attribute) or a part multiplicity (edge weight) and ID a foreign key to the part master data table. We vary the hierarchy table size $N$ from $10^3$ to $10^7$ to cover data sizes that easily fit into cache as well as sizes that by far exceed (L3) cache capacity.

Rationale/Goals: Separate the hierarchy table from the actual input table. Model a hierarchy table that could reasonably appear in an analytic scenario (i. e., as a dimension hierarchy), or in a bills of materials scenario.

For the **hierarchy index** underlying Node we use three setups: (a) As a *static index* (static), we use a simple yet effective labeling scheme called PPPL (cf. [2]). Node stores the pre- and post-order rank, the parent's pre-order rank, and the level of the node. In addition, the rows are indexed by the pre- and post-order ranks using two simple lookup tables. As all query primitives boil down to simple $\mathcal{O}(1)$ arithmetics directly on Node, this is as fast as an index can get. It may be a good fit to read-mostly analytic scenarios, where the hierarchical table is loaded once and rarely changed. (b) As a *dynamic index*, we use a path-based variable-length labeling scheme similar to Ordpath [16]. `Node` stores a binary-encoded ordpath, and there is an ordered B-tree index on `Node`. A dynamic scheme is needed for OLTP scenarios, and path-

based schemes are generally good allround fits. However, the support for updates comes at a cost of slowing down most index primitives to $\mathcal{O}(l)$. More details? One dynamic index should be enough. It would also be nice to use BO-tree with $B$ = mix and *gap* backlinks, if we get it working. However, Ordpath may be more interesting to many readers, since it is used in Microsoft SQL Server and PostgreSQL. (c) For some experiments we also emulate the trivial *adjacency list* scheme, where Node consists of a pair (ID, PID) of integers, and there is a (unique) hash index on ID and another (non-unique) hash index on PID.

We generate the **hierarchy structure** represented by Node according to the following parameters: [Regular$\langle k \rangle$] Each inner node is given exactly $k$ children. This results in a hierarchy height of $h = \log_k(N(k-1)+1)$. With $N = 10^7$, $k = 2$ gives us $h \approx 23$, while $k = 5$ gives us $h \approx 11$ and $k = 10$ gives us $h \approx 8$. This allows us to assess inhowfar the performance depends on the hierarchy height. [Realistic] Here the hierarchy structure is based on real-world materials planning data of an SAP application. We expand its size to $N$ by replicating subtrees in such a way that the structural properties, in particular the average height of 10.33, remain similar. Rationale/Goals: Be able to vary hierarchy sizes, depths, and "widths", while keeping a constant structure.

Our **input table** has the schema

$$T : \{[\underline{\text{Node}} : \texttt{NODE}, \text{Value} : \texttt{INT}, \text{Payload} : \texttt{BINARY(32)}]\}$$

It associates a subset of the nodes with some values as computation input, and additional opaque payload data to be carried along. While in a real query $T$ would be the result of a subquery (and pipelined through), for our experiments we simply materialize it with random data for Value and Payload.

## 6.2 Experiment #1: Basic Queries

We first measure some hierarchical computations with $\bowtie$ and $\hat{\Gamma}$ operators in isolation to assess their bare performance. The basic query is the same as in our Fig. 6 example:

```
SELECT Node, expr AS Result FROM T1
WINDOW bu AS (HIERARCHIZE BY Node BOTTOM UP), td AS (...)
```

Queries Q1 and Q2 use $expr$ = `SUM(Value)` over `bu` and `td`, respectively, representing a simple computation. Query Q3 is a non-simple weighted sum computation as in Fig. 6 (4c). Query Q4 uses `COUNT(DISTINCT Weight)` over `bu`, representing a complex aggregate state, as a hash table is needed for duplicate elimination. For each query we compare, where applicable,

hierarchy-$\hat{\Gamma}(T')$, $T'$ hierarchy-$\bowtie T'$, and $T'$ generic-$\bowtie T'$.

The input $T'$ is a projection of the required fields of T—i. e., Node and Value for Q1 and Q2, Node and Weight for Q3 and Q4—that is sorted and materialized in a row format a priori in the appropriate order for the respective query and operator—preorder for top-down and postorder for bottom-up computations—and then just scanned during the query. We also measure a plan based on a least fixpoint operator that uses iterative `IS_CHILD` hierarchy merge joins and sort-based grouping $\Gamma$ (except for the adjacency list model, where we use hash-based join and grouping operators). This plan is hand-crafted and would be difficult to even express in SQL, as we discussed in Sec. 4.1, but we include it to give a hint of the performance to be expected from an RCTE or iterative stored procedure. Recursive approach?

Fig. 7 shows the results. Observations here. Expected results: Q1 is very fast and top-down Q2 possibly even a bit faster. Non-simple computations Q3 are not significantly slower as simple computations Q1/Q2. $\hat{\Gamma}(e)$ should outperform $e \bowtie e$ as predicted in Sec. 3. The generic hierarchy-$\bowtie$ should be some factor slower than the native hierarchy groupjoin. With Q4 we expect a big advantage of reusing partial subaggregates. $\hat{\Gamma}$ and hierarchy-$\bowtie$ should be quite indifferent to the size $N$. The iterative plan should be by far the slowest.

## 6.3 Experiment #2: Complex Queries

Rationale/Goals: Show a more realistic, "holistic" query with some SQL clutter. Use FILTER –> subhierarchy with added interpolated edges. Show the potential advantage of the FILTER and HAVING optimizations. Maybe use especially complicated filters to increase the advantage.

Beyond the isolated operators, we now assess two complete query plans featuring a hierarchical computation. We extend Q1 with filters on both the input and output nodes. As discussed in Sec. 4, there are two ways to write this down, using either (a) an ordinary join-group-aggregate statement or (b) the `HIERARCHIZE BY` construct plus a `FILTER` and an a-posteriori `WHERE` condition. We compare the resulting queries Q5 using bottom-up hierarchy-$\bowtie$ on two separate input/output tables and Q6 using $\hat{\Gamma}$ on a combined input/output table. Both query plans also includes a join between `HT` and `T` and carry along the Payload. Fix the Payload field.

As discussed, we can push a combined filter $\phi(t) \vee \psi(t)$ below the left outer join, where $\phi(t) := t.\text{Weight} = 1$ and $\psi(t) := H.\text{level}(t.\text{Node}) \leq 3$.

Finally, we demonstrate a complex report query. Query Q7 does the same weighted rollup as Q5, but additionally relates the result value of each node to the result value of its parent node, in percent:

```
SELECT ID,
       RECURSIVE (Value + SUM(Weight * Result) OVER w) AS x,
       Result / VALUE_OF(Result) OVER (w RANGE 1 FOLLOWING)
-- todo!
  FROM HT NATURAL LEFT OUTER JOIN T
WINDOW w AS (HIERARCHIZE BY Node)
```

With $\hat{\Gamma}$ the result table can be computed in a single pass: Explain plan. Compare with (a) custom C++ code? (b) double-iterative workaround? Rationale/Goals: Show that some impressive-sounding queries can now be done pretty well and the overall performance is fine, since complex computations like this were not possible earlier. More ideas for a complex non-simple computation: Dewey conversion; Nested Sets conversion; deepest nesting of parts of a certain type (similar to subtree height); Springende Konten use case by Bastian; maybe a real big query involving many computations at once, mixing top-down and bottom-up, etc.; specific scenario: maybe profit center accounting

## 6.4 Experiment #3: Topological Sorting

In this experiment we investigate into the impact of topological sorting, which is required in any plan using our order-based operators.

Compare: (a) scan of clustered `HT`; (b) hierarchy index scan + table lookup; (c) scan of unclustered `HT` + sort.

Compare: (a) full pre- or postorder sort; (b) pre-to-post/post-to-pre rearrange. Goal: (b) is hopefully a few times faster than (a) – thus, at most one expensive sorting step is needed in most plans.

We also implemented a preorder-based hierarchy-$\bowtie$ bottom-up and a postorder-based top-down variant. Compare: (a) pre-based/post-based binary bottom-up hierarchical grouping; (b) pre-to-post/post-to-pre rearrange + post-based/pre-based bottom-up binary hierarchical grouping. If the performance is similar, we can get rid of the pre-based bottom-up variant, and don't need to measure it in Experiment #1.

## 6.5 Etc.

Try out all examples from Sec. 3/4 in the prototype in case there are bugs.

Assess robustness against degenerated hierarchies, for example: a hierarchy where one subtree is exceptionally deep.

# 7. RELATED WORK

**Representing Hierarchies** in a relational database schema is a deep subject [1]. As noted in Sec. 2, we hide the complexities through the abstract hierarchical table model of [3]. Our algorithms are thus largely generic and applicable to a multitude of indexing schemes on the query/update performance spectrum, in particular those surveyed in [2]. In the application scenarios we explored, two basic "schemes" are particularly widespread: the *adjacency list model* [1], which uses foreign key references to parent nodes, and the *leveled model* with a fixed number of named hierarchy levels (cf. Sec. 1 and [17, 18]). Both do not qualify for the hierarchical table model but are popular due to their simplicity.

**Expressing Hierarchical Computations**. While query languages such as MDX [18] or variations of grouping in XML/XQuery [19, 20] feature hierarchical navigation primitives and computations as first-class concepts and offer considerable expressiveness, our goal is to remain within SQL and leverage the language opportunities the `NODE` data type provides. Since a uniform SQL "interface" for handling hierarchies was lacking prior to [3], expressiveness largely depended on the chosen encoding scheme. On the low end, the adjacency list requires recursion for even simple structural pattern matching and computations. Stored procedures and RCTEs are two options to implement them; see [1] for an iterative solution and [21] for an exploration of RCTEs. Hand-crafted path- or containment-based encodings (e. g. [5]) alleviate many tasks by allowing users to avoid recursion through joins and explicit predicates such as $t_1.\text{pre} < t_2.\text{pre} \wedge t_1.\text{post} < t_2.\text{post}$ (see [4]), but computations are still limited to what is possible with join-group-aggregate (see Sec. 4) or inconvenient stored procedures and RCTEs. Targeting the leveled model in particular, SQL provides a `GROUP BY CUBE` and `ROLLUP` syntax [8, 22] that facilitates simple rollups (sums, counts, and the like) along named hierarchy levels. This is merely syntactic sugar for `GROUPING SETS` and thus of limited expressiveness. That said, combining `ROLL UP` with the `NODE` field and our algorithms would be an interesting language opportunity. Windowed tables are another flavor of grouping in SQL, which we use as a basis for expressing hierarchical computations due to certain semantic similarities (Sec. 4.2). This functionality in general has received surprisingly little attention in the research community; but see Leis et al. [13] for a recent guide.

**Evaluating Aggregation Queries**. Extensive literature is available on implementing general `GROUP BY` efficiently using either sort-based or hash-based methods [23]. Like classic sort-based grouping, our operators rely on sorted inputs (in this case, pre-/post-order) and are order-preserving. Techniques for maintaining *interesting orders* in cost-based optimizers [24] are therefore highly relevant. *Groupjoin* [10, 11, 25] improves join-group-aggregate plans by combining join and grouping, thus avoiding a materialization of the join result. As outlined in Sec. 3.4, $\bowtie$ and $\hat{\Gamma}$ can in certain cases be viewed as (self-)groupjoins with hierarchical join conditions. Non-equality groupjoins were considered in [11, 26]. Regarding `CUBE` and `ROLLUP`, [8] elaborates on possible implementations. A common approach is to reduce `CUBE` to a series of `ROLLUP`s and compute each using a dedicated single-pass operator that reuses results of lower levels. In this regard the operator is similar in spirit to ours. Regarding windowed tables, Leis et al. [13] provide an extensive treatment for modern multi-core architectures. However, due to the special semantics of hierarchical windows different operators have to be employed. Finally, regarding RCTEs, [21] recently reconsidered the optimization of SQL queries with linear recursion and `GROUP BY`. Their scope is more general as they consider arbitrary directed graphs, while our work focuses on hierarchies. Apart from that, the issues of RCTEs discussed in Sec. 4.1 apply.

**Hierarchy-aware Operators**. Since XML documents can be interpreted as hierarchies and represented in relational tables, there is a significant body of work on query processing in native XML stores or XML-enhanced RDBMS. Binary *structural join* operators resembling self-merge-joins were studied in [5, 6, 4]. Similar to our algorithms, they rely on an available (but hard-wired) hierarchy encoding and maintain a runtime stack of relevant intermediate results. Among the many subsequently proposed variants, [27] resembles ours in that they exploit a pre- and post-order encoding on the XML document. More powerful tree pattern matching operators were proposed in the XML context, but are beyond the requirements for handling hierarchical data in RDBMS.

# 8. CONCLUSION

[Summary] We propose to extend to RDBMS with native support for hierarchical computations by exploiting the opportunities the hierarchical table model [3] provides in terms of expressiveness and engine support. The `NODE` data type and SQL's windowed table mechanism turn out to be a very natural fit. Together with structurally recursive expressions, the user can state a useful class computations in an exceptionally intuitive and light-weight way—while avoiding the more powerful but cumbersome generative recursion mechanism of RCTEs. Our approach to evaluation is based on an available hierarchy indexing scheme and a hierarchy-aware query optimizer and execution engine featuring order-based hierarchical grouping operators. [Applicability] The operators work with a multitude of known indexing schemes given that pre-/post-order sorting and axis checks are supported. While our prototype uses a push-based query execution model, the techniques can be adapted to any relational DBMS. Evaluation summary. [Impact] Together, we expect this functionality to support and simplify many RDBMS-based applications dealing with hierarchies. Since the data model supports *arbitrary* hierarchies, we in particular expect new opportunities for applications that previously were restricted to leveled hierarchies due to expressiveness and performance issues. [Future Work] As part of future work, we investigate how the techniques can be extended to non-strict hierarchies or even general acyclic digraphs, given a topological ordering and efficient reachability checks; while the `NODE` model can straightforwardly accommodate DAGs, certain semantic issues due to ambiguous data flow need to be tackled. Based on experiences with our prototype in customer applications, we will also consider further opportunities for language extensions. [HANA integration teaser] ...

# 9. REFERENCES

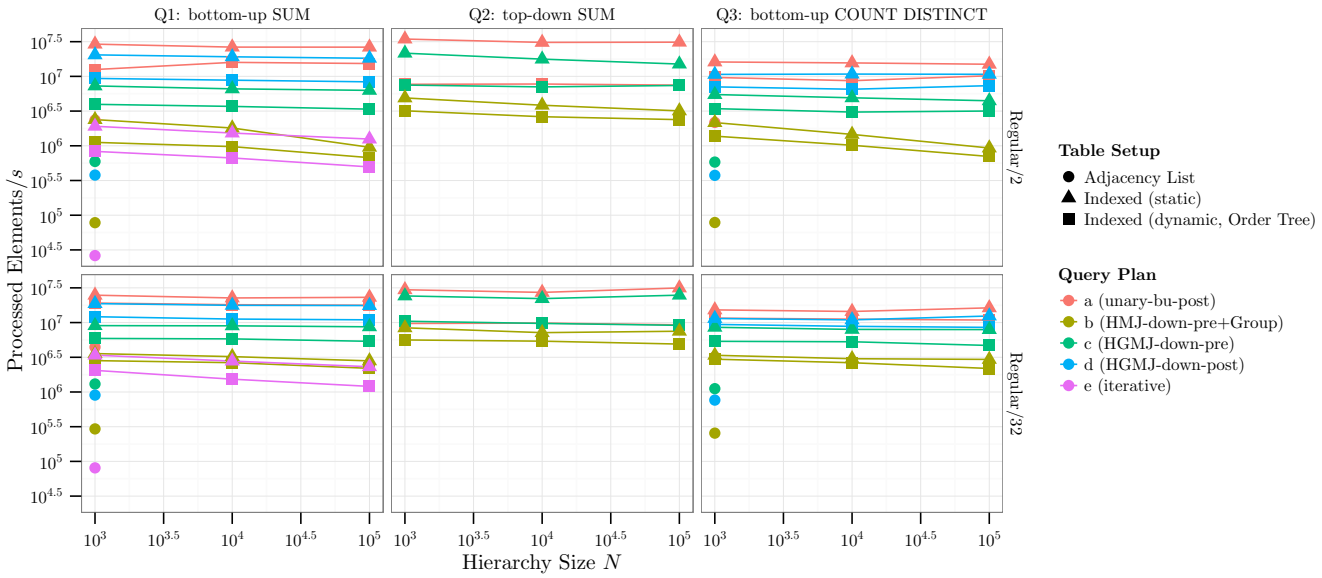



**Figure 7: Experiment #1**


[1] J. Celko, *Trees and Hierarchies in SQL for Smarties*, 2nd ed. Morgan Kaufmann, 2012.

[2] J. Finis, R. Brunel *et al.*, "Indexing highly dynamic hierarchical data," in *VLDB*, 2015.

[3] R. Brunel, J. Finis *et al.*, "Supporting hierarchical data in SAP HANA," in *ICDE*, 2015.

[4] T. Grust *et al.*, "Staircase Join: Teach a relational DBMS to watch its (axis) steps," in *VLDB*, 2003.

[5] C. Zhang *et al.*, "On supporting containment queries in relational database management systems," in *SIGMOD*, 2001.

[6] S. Al-Khalifa *et al.*, "Structural joins: A primitive for efficient XML query pattern matching," in *ICDE*, 2002.

[7] N. Bruno, "Holistic twig joins: Optimal XML pattern matching," in *SIGMOD*, 2002.

[8] J. Gray *et al.*, "Data Cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[9] C. Ordonez *et al.*, "Recursive query evaluation in a column DBMS to analyze large graphs," in *DOLAP*, 2014.

[10] G. Moerkotte and T. Neumann, "Accelerating queries with Group-By and Join by Groupjoin," *VLDB*, 2011.

[11] N. May and G. Moerkotte, "Main memory implementations for binary grouping," in *XSym*, 2005.

[12] S. Finkelstein *et al.*, "Expressing recursive queries in SQL," in *ANSI Document X3H2-96-075r1*, 1996.

[13] V. Leis *et al.*, "Efficient processing of window functions in analytical SQL queries," in *VLDB*, 2015.

[14] D. DeWitt, J. Naughton, and D. Schneider, "An evaluation of non-equijoin algorithms," in *VLDB*, 1991, pp. 443–452.

[15] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," in *VLDB*, 2011.

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

[17] "Oracle Corporation, Oracle 9i OLAP User's Guide, Release 2(9.2)," 2002, https://docs.oracle.com/cd/A97630_01/olap.920/a95295.pdf.

[18] "Multidimensional Expressions (MDX) Reference. SQL Server 2012 Product Documentation." http://msdn.microsoft.com/en-us/library/ms145506.aspx.

[19] C. Gokhale *et al.*, "Complex group-by queries for XML," in *ICDE*, 2007.

[20] N. Wiwatwattana, H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava, "X$\hat{3}$: a Cube operator for XML OLAP," in *ICDE*, 2007.

[21] C. Ordonez, "Optimization of linear recursive queries in SQL," in *TKDE Journal*, 2010.

[22] *Information technology — Database languages — SQL*, ISO/IEC JTC 1/SC 32 Std. 9075, 2011.

[23] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–169, 1993.

[24] P. G. Selinger *et al.*, "Access path selection in a relational database management system," in *Proceedings of ACM SIGMOD Conference*, 1979.

[25] D. Chatziantoniou *et al.*, "The MD-join: An operator for complex OLAP," in *ICDE*, 2001.

[26] S. Cluet and G. Moerkotte, "Efficient evaluation of aggregates on bulk types," in *Proc. Int. Workshop DBPL*, 1995.

[27] S. Chen *et al.*, "Twig$^2$Stack: Bottom-up processing of generalized-tree-pattern queries over XML documents," in *VLDB*, 2006.