Index-Assisted Hierarchical Computations in Main-Memory RDBMS

Robert Brunel* Norman May† Alfons Kemper*

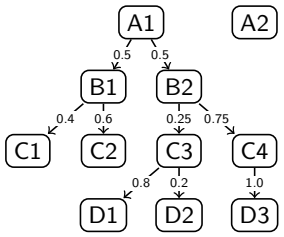
* Technische Universität München, Garching, Germany
† SAP SE, Walldorf, Germany
* *firstname.lastname*@cs.tum.edu † *firstname.lastname*@sap.com

ABSTRACT

We address the problem of expressing and evaluating computations on hierarchies represented as database tables. Engine support for such computations is very limited today, and so they are usually outsourced into stored procedures or client code. Recently, data model and SQL language extensions were proposed to conveniently represent and work with hierarchies. On that basis we introduce a concept of structural grouping to relational algebra, provide concise syntax to express a class of useful computations, and discuss algorithms 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. In the underlying RDBMS they are represented in *hierarchical tables* using relational tree encodings [4, 8]. Looking at typical queries especially in analytic applications, we see hierarchies serve mainly two purposes. The first is structural pattern matching, i. e., filtering and matching rows based on their positions in a hierarchy. The second is hierarchical computations: propagating measures and performing aggregation-like computations alongside the hierarchy structure. To address both purposes on RDBMS level, we need to solve two challenges: how can a user express a task at hand intuitively 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 on hierarchy axes, and techniques for appropriate indexes and join operators are well-studied [13, 26, 1, 3]. The same cannot be said of hierarchical computations. For the purpose of computations, a subset of the hierarchy nodes is dynamically associated with values to be propagated or aggregated, and possibly filtered. In analytic applications, this has always been a routine task: Dimension hierarchies are typically modeled by denormalized leveled tables such as City–State–Country–Continent. Certain computations can then be expressed using SQL's basic grouping mechanisms (in particular `ROLLUP` [12]). However, this is insufficient for computations beyond simple rollups, especially 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. 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? This quickly turns exceedingly difficult in SQL. One tool that comes to mind are recursive common table expressions (RCTEs). However, more intricate computations tend to result in convoluted, inherently inefficient statements. Lacking RDBMS support, today users resort to stored procedures or client code as workarounds. These are unsatisfactory not only concerning expressiveness, they also ignore the known hierarchy structure and are thus handicapped in terms of efficiency.



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* HT

We address the open issues of expressiveness and efficiency regarding complex computations on arbitrary irregular hierarchies by enhancing the RDBMS backend. Our foundation are the data model and SQL constructs from [2], which allow the user to conveniently define and query arbitrary hierarchies. This opens up new opportunities: the backend becomes aware of the hierarchy structure and can rely on powerful indexing schemes for query processing. We first introduce the basic concepts of hierarchical computations (Sec. 2), then proceed to corresponding SQL constructs (Sec. 3), which are translated into structural grouping operations in relational algebra (Sec. 4). The efficient evaluation of structural grouping requires index-assisted physical algebra operators (Sec. 5). We assess them against common alternative approaches (Sec. 6). Finally, we examine related work (Sec. 7) and wrap up the key properties of our solution (Sec. 8).

Proceedings of the VLDB Endowment, Vol. 9, No. 12

2. MOTIVATION

Our starting point is a representation of hierarchical data in a relational table. More specifically, we assume a table that encodes—using a suitable scheme—a forest of ordered, rooted, labeled trees, such that one table tuple (row) represents one hierarchy node. The *labels* of a node are the associated row's fields. For trees a 1 : 1 association between a node and its incoming edge can be made, so each field value can be interpreted as a label on either the node or edge. In our example table HT of Fig. 1, we view Weight as an edge label. The *ordered* property means that siblings have a defined order. It implies that every node has a well-defined rank in the pre- or post-order sequence of all nodes; e. g., B1 in the figure has pre rank 2 and post rank 3. While there are lots of options regarding the actual tree encoding to use, we intend to keep our discussion of hierarchical computations completely encoding-agnostic. The *hierarchical table model* [2] helps us with that: it conveniently hides the encoding details through an abstract data type `NODE`. The Node attribute of HT identifies a row's position and is backed by a hierarchy index H, which encapsulates the forest structure. We assume the index supports at the minimum two basic primitives, is-before-pre and is-before-post, in $\mathcal{O}(\log |\text{HT}|)$ or even $\mathcal{O}(1)$. Given a pair (ν_1, ν_2) of node values, they test whether ν_1 precedes ν_2 with respect to pre- and post-order traversal of the hierarchy. This allows us to test pairs of nodes against the main *hierarchy axes*:

preceding:	is-before-pre$(\nu_1, \nu_2) \wedge$ is-before-post(ν_1, ν_2)
descendant:	is-before-pre$(\nu_2, \nu_1) \wedge$ is-before-post(ν_1, ν_2)
following:	is-before-pre$(\nu_2, \nu_1) \wedge$ is-before-post(ν_2, ν_1)
ancestor:	is-before-pre$(\nu_1, \nu_2) \wedge$ is-before-post(ν_2, ν_1)
self:	$\nu_1 = \nu_2$

In HT, C3 follows B1/C1/C2 and precedes A2/C4/D3. The ancestor/descendant and preceding/following axes are symmetric. (Refer to [13] for a very visual discussion.) In pseudo code we denote e. g. "H.is-descendant(ν_1, ν_2)" for an axis check "ν_1 is a descendant of ν_2", and sometimes use "-or-self" variants with the obvious meaning. Specific index implementations will natively support these and other axes as well as further primitives (e. g. is-child, level), but our algorithms rely only on is-before-pre/post. An example implementation is the simple PPPL labeling scheme [8]. Here, Node is a 4-tuple storing the pre/post ranks, the parent's pre rank, and the level of the node. Additionally, the hierarchy table is indexed on the pre/post ranks using two simple lookup tables. With PPPL, the index primitives obviously boil down to very cheap $\mathcal{O}(1)$ arithmetics on Node, so this is as fast as a hierarchy index can get. If some degree of update support is needed, however, a more sophisticated indexing scheme must be chosen; see [8] for a recent overview. Note again that, while we rely on the `NODE` abstraction for ease of presentation, the concepts and algorithms of this paper could easily be adapted to any specific "hard-coded" encoding that affords the said primitives.

A *hierarchical computation* propagates or 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*, matching the natural direction of the edges). Unlike the "static" labels stored with the base table itself (e. g. ID and Weight in HT), the computation input is generally the result of an arbitrary subquery that associates some hierarchy nodes with input values, such as table Inp1 in Fig. 2a.

Inp1

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

Out1

A1
B1
C1

(a)

Inp2

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
A2	'A2'	NULL	NULL

(b)

Figure 2: Example tables — (a) input/output nodes for binary grouping; (b) combination of HT and Inp1 for unary grouping

In an analytic scenario, HT may be a so-called dimension hierarchy arranging products (leaves) into products groups (inner nodes), and a fact table Sales may associate each sale item with a specific product, i. e., a leaf of HT:

Sales : {[Sale, Item, Customer, Product, Date, Amount]}

Here, the computation input are the amounts from Sales, attached to some of the product leaves via join. A canonical task in such scenarios known as rollup is to sum up the revenue of certain products—say, "type A"—along the hierarchy bottom up and report these sums for certain product groups visible in the user interface—say, the three uppermost levels. The following SQL statement I-a computes the rollup, using the self-explanatory `IS_DESCENDANT_OR_SELF` and `LEVEL` constructs from [2]:

I-a

```
WITH Inp1 AS (
   SELECT p.Node, s.Amount AS Value
     FROM HT p JOIN Sales s ON p.Node = s.Product
    WHERE p.Type = 'type A' )
SELECT t.*, SUM(u.Amount) AS Total
  FROM HT t LEFT OUTER JOIN Inp1 u
         ON IS_DESCENDANT_OR_SELF(u.Node,t.Node)
 WHERE LEVEL(t.Node) <= 3
 GROUP BY t.*
```

This represents a class of hierarchical computations with two particular characteristics: First, only a subset of nodes carry an input value—often only the leaves, as in the example; we call these input nodes. Second, the set of input nodes is mostly disjoint from the output nodes that after the computation carry a result we are interested in. Input and output nodes are therefore determined by separate subqueries and the queries follow a join–group–aggregate pattern. We refer to this scheme as *binary structural grouping*. "Structural" here alludes to the role the hierarchy structure plays in forming groups of tuples. The query plans are typically variations of $\Gamma_{t.*;\, x\,:\, f}(e_1[t] \bowtie_{u\, <\, t} e_2[u])$, where $\bowtie$ denotes the left outer join, Γ denotes unary grouping (cf. [17]), and $<$ reflects the input/output relationship among tuples. Suppose we wanted to compute a rollup based on our example input Inp1, and we are interested in three output nodes given by Out1 in Fig. 2a. To do so, we use $e_1 =$ Out1, $e_2 =$ Inp1, and define the $<$ predicate as H.is-descendant-or-self$(u.\text{Node}, t.\text{Node})$ and $f(X)$ as $\sum_{u \in X} u.\text{Value}$. This yields the sums 6310, 310, and 100 for A1, B1, and C1, respectively.

Such query plans perform acceptably when f is cheap to compute and the set of output nodes is rather small. However, there is a major efficiency issue: for each e_1 tuple, the computation f bluntly sums up all matching input values from e_2, while ideally we would reuse results from previously processed e_1 tuples. In our example, to compute the sum for A1 we can save some arithmetic operations by reusing the

sum of (B1) and adding just the input values of (D1)/(D2)/(D3). With respect to $<$, we say that the output node (B1) is *covered* by the output node (A1) and thus carries a reusable result. To enable such reuse, the binary grouping algorithms we propose in this paper process the e_1 tuples in $<$ order and memorize any results that may be relevant for upcoming e_1 tuples. Thereby they overcome the mentioned inefficiencies.

From an expressiveness point of view, the widespread join–group–aggregate statements are fairly intuitive to most SQL users, yet not fully satisfactory: they lack conciseness, since conceptually a table of $<$ pairs 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. They become 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 Inp1 AS t LEFT OUTER JOIN Inp1 AS u
       ON IS_DESCENDANT_OR_SELF(u.Node, t.Node)
 GROUP BY t.*
```

Our proposed extensions to SQL's windowed table mechanism will allow us to equivalently write:

```
SELECT Node, SUM(Value) OVER (HIERARCHIZE BY Node)             II-b
  FROM Inp1
```

We refer to this scheme as *unary structural grouping*, since the computation now works on a single table. It inherently yields a result for every tuple, i. e., every node acts as both an input and output node. A binary grouping query can usually be rewritten to unary grouping by working on a merged "$e_1 \cup e_2$" table and filtering the output nodes a posteriori. For example, Inp2 in Fig. 2b shows a combination of HT and Inp1; here we assigned `NULL` as a neutral value to nodes which do not carry a meaningful value. Rewriting binary to unary computations will often result in more concise and intuitive statements. Especially when there is no clear distinction between input and output nodes, unary grouping is the most natural approach.

The unary structural grouping mechanism offers us another attractive language opportunity: support for structural recursion. Using a structurally recursive expression we can state the rollup in Stmt. II-a and II-b in yet another way:

```
SELECT Node, RECURSIVE INT (Value + SUM(x) OVER w) AS x        II-c
  FROM Inp1 WINDOW w AS (HIERARCHIZE BY Node)
```

This expression for x sums up the readily computed sums x of all tuples that are *covered* by the current tuple. Unlike binary grouping, unary grouping with structural recursion makes the reuse of previous results explicit and thus inherently translates into the efficient evaluation approach. Furthermore, it enables us to state significantly more complex computations with remarkable conciseness. For example, we can now straightforwardly take the edge weights from Inp2 into account in our rollup:

```
SELECT Node, RECURSIVE DOUBLE (                                III
         Value + SUM(Weight * x) OVER w ) AS x
  FROM Inp2 WINDOW w AS (HIERARCHIZE BY Node)
```

Rather than actually performing recursion, our operators evaluate unary grouping in a bottom-up fashion, leveraging a $<$-sorted input table like their binary counterparts.

3. EXPRESSING COMPUTATIONS IN SQL

Unlike binary grouping, unary structural grouping is a novel concept to SQL. Following our informal motivation in the previous section, we now cover the syntax and semantics of our extensions for unary grouping.

3.1 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 extend this mechanism by *hierarchical* windows. Let us first briefly review the standard terminology and behavior of windowed tables (refer to e. g. [25] for details). A standard *window specification* may comprise a *partition clause*, an *ordering clause*, and a *frame clause*. Consider how we may annotate our Sales table from Sec. 2 with per-customer sales totals running over time:

```
SELECT Customer, Date, SUM(Amount) OVER w
  FROM Sales WINDOW w AS (
         PARTITION BY Customer ORDER BY Date
         RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
         EXCLUDE NO OTHERS )
```

The frame clause "`RANGE...NO OTHERS`" is the implicit default and could be omitted. Briefly put, the query is conceptually 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. Note that tuples need 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`.

For unary structural grouping, our windowed table will be some collection of nodes (e. g. Inp1); that is, there is a `NODE` field whose values are drawn from a hierarchical base table (e. g. HT). 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 ν [BOTTOM UP|TOP DOWN]
```

The clause determines the `NODE` field ν, its underlying hierarchy index 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:

$$\begin{aligned} &\text{top-down:} && u < t \;:\Longleftrightarrow\; H.\text{is-descendant}(t.\nu, u.\nu) \\ &\text{bottom-up:} && u < t \;:\Longleftrightarrow\; H.\text{is-descendant}(u.\nu, t.\nu) \end{aligned}$$

We additionally need the notion of *covered* elements we used informally in Sec. 2. An element u is said to be *covered* by another element t if no third element lies between them:

$$u <: t \;:\Longleftrightarrow\; u < t \;\wedge\; \neg\exists\, u' : u < u' < t. \qquad \text{Eq. 1}$$

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 intermediate nodes are missing in the input.

Inp3		Window Frame						Result
Node	Value		0	1	2	3	4	5	x
C1	100	0	≙						100
C2	200	1		≙					200
D1	1000	2			≙				1000
D3	3000	3				≙			3000
B2	20	4			<:	<:	≙		4020
A1	1	5	<:	<:	<	<	<:	≙	4321

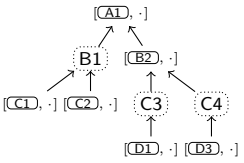


Figure 3: A bottom-up hierarchical window

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

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 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 itself. The handling of (c) tuples can be controlled independently via the `EXCLUDE` clause, but we omit these details for brevity.

Consider Fig. 3, where we apply a bottom-up hierarchical window to table Inp3 and compute $x =$ `SUM(Value)` like in Stmt. II-b from Sec. 2. The matrix indicates the relationships of the tuples. Since our window uses the default frame clause, the frames comprise exactly the $<$, $<$:, and tied $\hat{=}$ tuples. Summing over them yields the x values shown to the right. Note that although Inp3 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. In that, it behaves precisely like the alternative binary approach based on an `IS_DESCENDANT_OR_SELF` join (Stmt. II-a). For basic rollups, which are by far the most common type of hierarchical computation, the implicit window frame clause does just 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.

3.2 Recursive Expressions

Thus far, hierarchical windows are merely a shorthand; they can equivalently be expressed through 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 (same partitioning and `HIERARCHIZE` clause, i. e., `NODE` field and 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 circular dependency. It is conceivable to loosen the restrictions somewhat and give the user more control via a custom `RANGE` clause, but we do not consider that in this paper. Third, the field c may appear only *within* one of the window function expressions $expr_i$; say, 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 c is available. While this standard behavior is what is usually intended and convenient, SQL has a way to override the implicit frame row access. We can e. g. refer to the current tuple even within *AGG* using a so-called *nested window function*:

```
AGG(...VALUE_OF(c AT CURRENT_ROW)...) OVER w
```

We prohibit this for c, but allow it for any other field.

Returning to our Fig. 3, we can now equivalently apply the *recursive* rollup expression of Stmt. II-c, $x =$ `RECURSIVE INT (Value + SUM(x) OVER w)`, to Inp3. The window frames are now restricted to the covered $<$: tuples. Since Inp3 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.

3.3 Further Examples

Even with non-recursive expressions, hierarchical windows are already an attractive alternative to verbose join–group–aggregate statements. Consider our opening query I-a from Sec. 2. SQL allows aggregation to be restricted by a `FILTER`. This handy feature allows us to state this query as follows:

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

This saves us one join over Stmt. I-a. Note the outer join may yield tuples where Amount is `NULL`, but these are conveniently ignored by `SUM`. Altogether there are three points where we could 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). For the latter we must nest two selections, as SQL currently has no `HAVING` equivalent for windowed tables.

Fig. 4 shows further meaningful expressions, including non-recursive variants where possible, each based on either a bottom-up or a top-down hierarchical window on Inp2:

IV
```
SELECT Node, expr FROM Inp2
WINDOW td AS (HIERARCHIZE BY Node TOP DOWN),
       bu AS (HIERARCHIZE BY Node BOTTOM UP)
```

```
(1a) SUM(Value) OVER bu
(1b) RECURSIVE INT (Value + SUM(x) OVER bu) AS x

(2a) PRODUCT(Weight) OVER td   -- non-standard
(2b) RECURSIVE DOUBLE (
       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 4: SQL examples for unary computations

(1) is our familiar rollup. Besides `SUM`, the operation in (1a) could e. g. be `AVG`, `MIN`, `MAX`, `COUNT` (cf. Ex. 7), `EVERY`, `ANY`, or `ARRAY_AGG` to simply collect all values in an array. SQL's `DISTINCT` and `FILTER` constructs add further expressiveness. E. g., in a bill of materials we may count the distinct types of subparts from some manufacturer each part is built of:

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

(2) is a top-down counterpart to (1); it yields the effective weights by multiplying over all tuples on the root path. (2a) uses a hypothetical `PRODUCT` aggregation function, which is curiously missing from standard SQL; (2b) works around that via recursion, aptly taking advantage of `FIRST_VALUE`. 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 often the part appears in total in the assembly. **(3)** is a variant of (1) summing over only the covered tuples. In (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. As Inp2 happens to contain all HT nodes, the 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 weighted rollup. (4d) is mostly equivalent to (4b), but brings it into a form similar to (4c) using a nested window function to access the Weight of the current row. In general, such weighted rollups cannot be performed without (structural) recursion. That said, a non-recursive solution that sometimes works is to "multiply out" the expression according to the distributivity law and use *two* separate computations: First (2a), yielding absolute weights `w` for each tuple, then `SUM(w * Value)` bottom up. **(5)** constructs a path-based Dewey representation of the hierarchy using the same technique as (2): it builds a string from the ID values on the root path, e. g. `'/A1/B1/C1'` for C1. **(6–9)** compute properties of the data flow graph over the input table. As Inp2 contains all nodes of HT, they are equal to the node's (6) level, (7) subtree size, (8) subtree height, and (9) child count. In general (7) gives us the size of the window frame and (9) the number of covered tuples. Finally **(10)**, 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. This way arbitrary computations can be plugged in.

4. STRUCTURAL GROUPING

This section covers the relational algebra level. We propose two logical operators for evaluating hierarchical computation queries, one for unary and one for binary structural grouping.

4.1 Binary Grouping

Binary structural grouping queries typically feature an inner or left outer join on a hierarchy axis such as `IS_DESCENDANT`, and subsequent grouping of the outer side. They are initially translated into plans of the form $\Gamma(\cdot \bowtie_\theta \cdot)$ with a suitable hierarchy predicate θ. Due to the efficiency issues noted in Sec. 2, we want the query optimizer to rewrite this pattern into a single combined operator. This idea is not new to relational algebra but commonly known as *binary grouping* or *groupjoin*. It has been explored in depth in [17] mainly for the equi-join setting, together with relevant rewrite rules for query optimization. We repeat [17]'s definition of the binary grouping operator $\bowtie\!\!\!\!\Gamma$ with minor adaptions. It 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 and $\{\tau_i\}_b$ denotes a bag of τ_i tuples. 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 $\mathcal{N}$. Then $\bowtie\!\!\!\!\Gamma$ is defined as

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

where $e[\theta t] := \{u \mid u \in e \wedge \theta(u,t)\}_b$. It extends each $t \in e_1$ by an x attribute of type $\mathcal{N}$, whose value is obtained by applying function f to the bag $e[\theta t]$ containing the relevant input tuples for t. As an example, the plan $\Gamma_{t.*;\,x\,:\,f}(\mathrm{Out1}[t] \bowtie_{u<t} \mathrm{Inp1}[u])$ from Sec. 2 can be rewritten into $\mathrm{Out1} \bowtie\!\!\!\!\Gamma^{<}_{x\,:\,f} \mathrm{Inp1}$, using the same definitions of f and $<$. Beyond optimizing $\Gamma(\cdot \bowtie_\theta \cdot)$ plans, we also use $\bowtie\!\!\!\!\Gamma$ to evaluate hierarchical windows with non-`RECURSIVE` expressions. Those are translated into binary self-grouping $e \bowtie\!\!\!\!\Gamma^{\theta}_{x\,:\,f}\ e$, with $\theta =$ is-descendant-or-self in the bottom-up and $\theta =$ is-ancestor-or-self in the top-down case (modulo handling details of the frame clause and `EXCLUDE`). Further optimizations are possible from there. Consider Stmt. I-b from Sec. 3.3. It has a condition $\phi = (H.\mathrm{level}(\nu) \leq 3)$ on the output that does not depend on the computed sum x. Select operators σ_ϕ of this kind can typically be pushed down to the left $\bowtie\!\!\!\!\Gamma$ input. A `FILTER` ψ can be handled by f or pushed down to the right input. Such rewriting from $\sigma_\phi(e \bowtie\!\!\!\!\Gamma_{x\,:\,f^\psi}\ e)$ to $\sigma_\phi(e) \bowtie\!\!\!\!\Gamma_{x\,:\,f}\ \sigma_\psi(e)$ will always pay off, especially when the selections can be pushed down even further.

4.2 Unary Structural Grouping

To evaluate recursive expressions on a hierarchical window, we need a new operator: *unary structural grouping*. Since the concept as such may be useful beyond hierarchical windows, we define it in terms of an abstract $<$ comparison predicate on the tuples of its input relation, which drives the data flow. It is required to be a strict partial order: irreflexive, transitive, and asymmetric. The operator arranges its input in an acyclic directed graph whose edges are given by the notion of *covered* tuples $<:$ (Eq. 1 in Sec. 3.1). On that structure it evaluates a structural aggregation function f,

(1b)	total Value	↑	$t.\text{Value} + \sum_{u \in X} u.x$
(2b)	absolute Weight	↓	$t.\text{Weight} * \Pi_{u \in X} u.x$
(3b)	Value sum over $<$:	↑	$\sum_{u \in X} u.\text{Value}$
(4a)	weighted rollup	↑	$t.\text{Weight} * (t.\text{Value} + \sum_{u \in X} u.x)$
(4b)			$t.\text{Value} + t.\text{Weight} * (\sum_{u \in X} u.x)$
(4c)			$t.\text{Value} + \sum_{u \in X} u.\text{Weight} * u.x$
(4d)			$t.\text{Value} + \sum_{u \in X} t.\text{Weight} * u.x$
(5)	Dewey conversion	↓	$\langle t.\text{ID} \rangle$ if $X = \{\}_b$, $u.x \circ \langle t.\text{ID} \rangle$ if $X = \{u\}_b$
(6b)	level	↓	$1 + \sum_{u \in X} u.x$
(7b)	subtree size	↑	$1 + \sum_{u \in X} u.x$
(8)	subtree height	↑	1 if $X = \{\}_b$, else $1 + \max_{u \in X} u.x$
(9b)	degree	↑	$\lvert X \rvert$

Symbols: ↑ bottom up ↓ top down

Figure 5: Example definitions of $\hat{\Gamma}$'s $f(t, X)$

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 $<$.

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 a scalar type $\mathcal{N}$. The *unary structural grouping* operator $\hat{\Gamma}$ associated with $<$, x, and f is defined as

$$\hat{\Gamma}^{<}_{x\,:\,f}(e) := \{t \circ [x : \text{rec}^{<}_{x\,:\,f}(e, t)] \mid t \in e\}_b, \text{ where}$$
$$\text{rec}^{<}_{x\,:\,f}(e, t) := f(t, \{u \circ [x : \text{rec}^{<}_{x\,:\,f}(e, u)] \mid u \in e[_{<:}t]\}_b).$$

We reuse the symbol Γ of common 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 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 a *strict* partial order.

For hierarchical windows, we define $<$ as in Sec. 3.1 in terms of H.is-descendant, which is indeed irreflexive, transitive, and asymmetric. We can now translate our two statements from Sec. 2 into plans based on $\hat{\Gamma}$:

II-c $\hat{\Gamma}^{<}_{x\,:\,f}(\text{Inp1}),\ f(t, X) = t.\text{Value} + \sum_{u \in X} u.x$
III $\hat{\Gamma}^{<}_{x\,:\,f}(\text{Inp2}),\ f(t, X) = t.\text{Value} + \sum_{u \in X} u.\text{Weight} * u.x$

Fig. 5 shows definitions of f corresponding to the SQL expressions of Fig. 4. As the examples attest, `RECURSIVE` expressions translate almost literally into suitable $f(t, X)$ formulas.

4.3 Unary Versus Binary Grouping

Theoretically, there are little restrictions on the function f of $\hat{\Gamma}$ and $\ltimes\!\Gamma$; 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 $e \ltimes\!\Gamma e$. An aggregation function $\{\tau\}_b \to \mathcal{N}$ for use with $\ltimes\!\Gamma$ is *simple* if it is of the form $\text{acc}_{\oplus;\,g}(X) := \bigoplus_{u \in X} g(u)$, where function $g : \tau \to \mathcal{N}$ extracts or computes a value from each tuple, 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)` where `AGG` is a basic aggregate function such as `SUM`, `MIN`, `MAX`, `EVERY`, or `ANY` without `DISTINCT` set quantifier. (A further extension to arbitrary `FILTER(WHERE` ψ`)` conditions is possible.)

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

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

In Fig. 5, 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 digraph imposed by $<$ on e is a tree—i. e. 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 $u \leq t \ :\Longleftrightarrow\ u < t \vee u = t$. 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:

$$e \ltimes\!\Gamma^{\leq}_{x\,:\,\text{acc}_{\oplus;\,g}}\ e = \hat{\Gamma}^{<}_{x\,:\,\text{str-acc}_{x\,:\,\oplus;\,g}}(e).$$

Note that this equivalence will *not* hold if there are multiple chains $u <: \ldots <: t$ connecting two tuples $u < t$ in the input e. In this situation $\hat{\Gamma}$ indirectly counts u multiple times into t's result, while $\ltimes\!\Gamma$ does not. This is due to the particular semantics of structural recursion, which simply propagates x values along the $<:$ chains. When we apply $\hat{\Gamma}$ in our hierarchical window setting, the equivalence holds, as $<:$ is derived from the acyclic tree structure of H, if 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 Sec. 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, $\ltimes\!\Gamma$ is an alternative to $\hat{\Gamma}$. If a `WHERE` condition ϕ on the output or a `FILTER` condition ψ is applied, $\sigma_\phi(e) \ltimes\!\Gamma\, \sigma_\psi(e)$ will usually be superior to $\sigma_\phi(\hat{\Gamma}_{f^\psi}(e))$, as already noted in Sec. 4.1. Finally, our manual rewrite of Stmt. I-a to I-b, where we saved one join, demonstrates an advanced optimization from $e_1 \ltimes\!\Gamma\, e_2$ to Γ: By "merging" the two inputs into e_{12}, we could (without going into details) rewrite $e_1 \ltimes\!\Gamma\, e_2$ to $e_{12} \ltimes\!\Gamma\, e_{12}$ and then $\hat{\Gamma}(e_{12})$, which pays off if e_{12} can be further simplified, e. g. when e_1 and e_2 were very similar in the first place. Establishing relevant equivalences to enable such optimizations is part of future work.

5. PHYSICAL ALGEBRA OPERATORS

We now discuss efficient algorithms for $\ltimes\!\Gamma^{\theta}_{x\,:\,f}$ and $\hat{\Gamma}^{<}_{x\,:\,f}$.

5.1 Overview

[$\ltimes\!\Gamma$] A general approach for $\ltimes\!\Gamma$ is to treat θ as an opaque join predicate with partial order properties, and stick to a generic sort-based join–group–aggregate technique: sort both inputs e_1 and e_2 according to θ, then use a sort-based left

outer join $e_1[t] \bowtie_\theta e_2[u]$, and then sort-based unary grouping $\Gamma_{t.*;\,x:f}$ to compute the result. This requires a non-equi join operator that can deal with the fact that some tuples may be incomparable through θ, and retains the order of e_1. Since we make no assumptions on e_1 and e_2, we have to use a nested loops join, making the runtime complexity an unattractive $\Theta(|e_1| \cdot |e_2|)$. An *index-based* nested loops join could *not* be used since there generally is no index on the given inputs—only the hierarchical base table HT is indexed. We refer to this approach by "$\bowtie\!-\!\Gamma$". It is usually the only option when an encoding such as PPPL from Sec. 2 is hand-implemented in an RDBMS without further engine support.

[hierarchy-$\bowtie\!-\!\Gamma$] When $\ltimes$ and $\hat{\Gamma}$ are used for hierarchical computations and θ and $<$ operate on `NODE` fields, the underlying hierarchy index H can and should be leveraged. A big improvement over $\bowtie\!-\!\Gamma$ is to use a *hierarchy merge join*, a sort-based structural join operator with a time and space complexity of $\mathcal{O}(|e_1| + |e_2| + |e_1 \bowtie e_2|)$. Al-Khalifa et al. [1] describe two variants of the algorithm, which consume preordered inputs and join on the *descendant* axis: "stack-tree-desc" retains the order of e_2 in the output; the somewhat less efficient "stack-tree-anc" retains the e_1 order. Although originally applied to XML data, both can be adapted to our SQL setting. We refer to this approach by "hierarchy-$\bowtie\!-\!\Gamma$". It can be considered the state of the art and a natural baseline for our native $\hat{\Gamma}$ and $\ltimes$ algorithms. Note that even though more sophisticated join techniques have been studied in the XML world, most of them are not applicable to our setting since we are working on *arbitrary* inputs rather than the base table HT, as mentioned above—see also Sec. 7.

[hierarchy-$\hat{\Gamma}$, hierarchy-$\ltimes$] While the said approaches keep implementation efforts low by reusing existing operators, they cannot evaluate the structural recursion of $\hat{\Gamma}$, and they suffer from the efficiency issues noted in Sec. 2: all $<$ join pairs rather than just the $<:$ pairs are materialized and processed during query evaluation, and results from covered tuples are not reused. We therefore propose four specialized operators: hierarchy-$\hat{\Gamma}$ and hierarchy-$\ltimes$, 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. We proceed to discuss their pseudo code. For ease of presentation, we directly use concepts from relational algebra level: An abstract data type Aggregate represents a tuple bag X and supports self-explanatory operations clear(), add(u), and merge(X'). During execution of e_1 hierarchy-$\ltimes$ e_2 or hierarchy-$\hat{\Gamma}(e_1)$, we create one Aggregate instance X per tuple $t \in e_1$, assemble the appropriate input tuples in it and feed it to the aggregation function $f(X)$ or $f(t, X)$ to obtain $t.x$. In the *actual* query-specific implementation of an Aggregate and its operations, significant optimizations may be possible depending on f; Sec. 5.4 will discuss them.

5.2 Unary Hierarchical Grouping

Alg. 1 shows the two variants of hierarchy-$\hat{\Gamma}$. In a single pass through the input e, they effectively issue the following call sequence for each tuple t:

X.clear(); X.add(u) for each $u <: t$; **yield** $t \circ [x : f(t, X)]$

where "yield" outputs a result tuple. The stack S (line 1) manages previously processed tuples u and their computation states, i. e., $u.x$ and the corresponding aggregate X for potential reuse. For each $t \in e$ (l. 3) we first check whether

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

Input: $e : \{\tau\}_b$, where τ has a ν : `NODE`^{H} field;
e ordered by ν in post-/pre-order (bottom up/top down)
Output: $\{\tau'\}_b$, where $\tau' := \tau \circ [x : \mathcal{N}]$; same order

```
1  S : Stack⟨[ν : NODE^H, u : τ', X : Aggregate⟨τ'⟩]⟩
2  X : Aggregate⟨τ'⟩
3  for t ∈ e
4      if S ≠ ⟨⟩ ∧ S.top().ν = t.ν
5          skip                                  // reuse previous X
6      else
7          X.clear()
8          ⟨collect input⟩*
9      yield t' ← t ∘ [x : f(t, X)]
10     S.push([t.ν, t', X])
```

*⟨collect input⟩ — bottom up:

```
11 while S ≠ ⟨⟩ ∧ ¬H.is-before-pre(S.top().ν, t.ν)
12     [·, u, X_u] ← S.pop()
13     X.add(u)                                  // leverage X_u if possible!
```

*⟨collect input⟩ — top down:

```
14 while S ≠ ⟨⟩ ∧ H.is-before-post(S.top().ν, t.ν)
15     S.pop()
16 if S ≠ ⟨⟩
17     for [ν, u, X_u] ∈ upper part of S where ν = S.top().ν
18         X.add(u)                              // leverage X_u if possible!
```

$t.\nu$ equals the previous node; in this case, we reuse X as is. (This step can be omitted if ν is known to be duplicate-free.) Otherwise, the "collect input" block (l. 8) maintains S and collects the tuples X covered by t. We then compute $f(t, X)$, construct and yield an output tuple and put it on S together with X for later reuse. Regarding "collect input", consider first the bottom-up case (postorder input): Previous 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 e tuples are on the *ancestor* or *following* axes). The covered tuples X we need for t are thus conveniently placed on the upper part of S. The while loop (l. 11) collects and removes them, as they will no longer be needed. Any remaining S entries are *preceding* and irrelevant to t, but might be consumed later. In the top-down case (preorder input), 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. The while loop (l. 14) first dismisses the *preceding* tuples. If there is an entry left on top of S (l. 16), it is a covered ancestor $u <: t$, and the for loop (l. 17) 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, since they may still be needed for upcoming *following* tuples.

Note that the algorithm needs no checks for $<:$, as the covered tuples are identified implicitly. Note also that in l. 13 and 18, the full X_u state corresponding to $u.x$ is available to add(). This state is needed for non-trivial computations where $u.x$ alone does not provide enough information (cf. Sec. 5.4). In case it is *not* needed, we need not keep X (marked blue) on S at all. Likewise, we may include only the fields of u actually accessed by f to save memory.

5.3 Binary Hierarchical Grouping

Alg. 2 shows hierarchy-$\ltimes$. The bottom-up variant (postorder inputs) joins on is-descendant-or-self, the top-down variant

Algorithm 2: e_1 hierarchy-$\ltimes^{\nu_1;\nu_2}_{x:f}$ e_2

Input: $e_1 : \{\tau_1\}_b$ and $e_2 : \{\tau_2\}_b$, where τ_i has a $\nu_i : \texttt{NODE}^H$ field e_i ordered by ν_i in post-/pre-order (bottom up/top down)
Output: $\{\tau_1 \circ [x : \mathcal{N}]\}_b$, same order as e_1

```
1  p : int, initially p ← 0    // position in e2 (iterator)
2  S1 : Stack⟨[ν : NODE^H, X : Aggregate⟨τ2⟩, i : int]⟩
3  S2 : Stack⟨τ2⟩
4  X : Aggregate⟨τ2⟩
5  for t1 ∈ e1
6      if S1 ≠ ⟨⟩ ∧ S1.top().ν = t1.ν1
7          yield t1 ∘ [x : f(X)]          // reuse previous X
8          continue
9      X.clear()
10     ⟨collect input⟩*
11     yield t1 ∘ [x : f(X)]
12     S1.push([t1.ν1, X, |S2|])
```

*⟨collect input⟩ — bottom up:

```
13 while S1 ≠ ⟨⟩ ∧ ¬H.is-before-pre(S1.top().ν, t1.ν1)
14     [·, X′, ·] ← S1.pop()
15     X.merge(X′)
16 while S2 ≠ ⟨⟩
17     t2 ← S2.top()
18     if ¬(t1.ν1 = t2.ν2 ∨ H.is-before-pre(t1.ν1, t2.ν2))
19         break
20     S2.pop()
21     X.add(t2)
22 while p ≠ e2.size()
23     t2 ← e2[p]
24     if H.is-before-post(t1.ν1, t2.ν2)
25         break
26     if t1.ν1 = t2.ν2 ∨ H.is-before-pre(t1.ν1, t2.ν2)
27         X.add(t2)
28     else
29         S2.push(t2)
30     p ← p + 1
```

*⟨collect input⟩ — top down:

```
31 while S1 ≠ ⟨⟩ ∧ H.is-before-post(S1.top().ν, t1.ν1)
32     S1.pop()
33 i′ ← 0
34 if S1 ≠ ⟨⟩
35     [·, X′, i′] ← S1.top()
36     X.merge(X′)
37 while i′ ≠ S2.size() ∧ H.is-before-post(t1.ν1, S2[i′].ν2)
38     X.add(S2[i′])
39     i′ ← i′ + 1
40 pop S2[i′], ..., S2.top()
41 while p ≠ e2.size()
42     t2 ← e2[p]
43     if H.is-before-pre(t1.ν1, t2.ν2)
44         break
45     if t1.ν1 = t2.ν2 ∨ H.is-before-post(t1.ν1, t2.ν2)
46         X.add(t2)
47         S2.push(t2)
48     p ← p + 1
```

(preorder inputs) on is-ancestor-or-self, with left outer join semantics. Other axes (child/parent and the non-"self" variants) as well as *inner* joins could be handled with minor adaptions, which we omit for brevity. Both inputs are sequentially accessed: The outer loop (l. 5) passes through e_1, whereas e_2 is accessed via an iterator p. S_2 stashes processed e_2 tuples that may still become relevant as join partners. S_1 collects processed nodes ν_1 from e_1 with the corresponding aggregates X of θ-matched e_2 tuples for reuse. i refers to an S_2 position and is needed in the top-down case only.

For each $t_1 \in e_1$ (l. 5) we again either reuse X from a previous equal node (l. 6–8) or assemble X via "collect input", before producing an output tuple and memoizing X on S_1. In the bottom-up case (postorder inputs), "collect input" first (l. 13) removes all covered *descendant* entries from S_1 and merges their aggregates into X. This operation is the key to effectively reusing previous results as motivated in Sec. 2. The following loop (l. 16) moves relevant θ matches on the *descendant-or-self* axis from S_2 to X, and the final loop (l. 22) advances the right input e_2 up to the first postorder successor of ν_1. Any encountered t_2 is either a postorder predecessor or $\nu_2 = \nu_1$; if t_2 is also a preorder successor, it is a descendant. θ matches are added straight to X (l. 27), *preceding* tuples are stashed on S_2 (l. 29).

The top-down case (preorder inputs) is more involved: 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 θ 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. 31), we pop obsolete *preceding* entries from S_1. Second (l. 34), any remaining entry on S_1 is an *ancestor*, so we reuse its X'. Third (l. 37), we add to X any additional ancestors t_2 that were not already in X' (starting from position i'). Then, the remaining S_2 tuples from positions i' to top are *preceding* and therefore obsolete (l. 40). Finally (l. 41), we advance e_2 up to the first preorder successor of ν_1, adding *ancestor-or-self* tuples to X and S_2 but ignoring *preceding* tuples.

5.4 Further Discussion

Recall from Sec. 4 that we use hierarchy-$\hat{\Gamma}$ for `RECURSIVE` expressions on hierarchical windows and hierarchy-$\ltimes$ for *non-recursive* expressions (through self-grouping $e \ltimes e$) as well as certain classes of join–group–aggregate statements. Handling the details of hierarchical windows—i. e., different variants of frame and `EXCLUDE` clauses—requires further additions to Alg. 1 and 2; in particular, tuples with equal ν 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 $\ltimes$ and $\hat{\Gamma}$: While the pseudo code of Alg. 1 and 2 explicitly collects tuples into a bag X, we can often avoid this buffering altogether by evaluating f on the fly. To this end the query compiler has to generate specific code in place for the Aggregate operations:

① X.clear(), ② X.add(u), ③ X.merge(X'), ④ $f(t, X)$.

Consider Expr. 1b from Fig. 5: The actual state of X would be a partial sum $x : \mathcal{N}$, and the operations boil down to

① $x \leftarrow 0$, ② $x \leftarrow x + u.x$, ③ $x \leftarrow x + X'.x$, and ④ $x + t.x$.

This works with both $\hat{\Gamma}$ and $\ltimes$. As a structurally recursive example with $\hat{\Gamma}$, consider Expr. 4c: here the state remains the same but ② becomes $x \leftarrow x + u.\text{Weight} * u.x$.

Eliminating X like this works whenever either the scalar x value itself or some data of $\mathcal{O}(1)$-bounded size can adequately represent the required state of a subcomputation. This 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 in [12]. But then there are SQL expressions, such as ARRAY_AGG or DISTINCT aggregates, for which we *have* to actually maintain X or some state of size $\Theta(|X|)$. Consider COUNT(DISTINCT Weight): To evaluate this using either $\hat{\Gamma}$ or $\ltimes\Gamma$, the Aggregate has to maintain a set of distinct Weight values. Still, our mechanism for reusing subcomputations provides certain optimization opportunities; like using an efficient set union algorithm for operation ③.

Complexities. With this in mind, let us consider the runtime and space complexities. We can assume the is-before primitives to be in $\mathcal{O}(1)$ for most static indexes and in $\mathcal{O}(\log|\text{HT}|)$ for common dynamic indexes [8], |HT| being the hierarchy size; either way, they are not affected by the input sizes of $\hat{\Gamma}$ and $\ltimes\Gamma$. Furthermore, if the computation is done inline as discussed, the size and all operations on X are actually in $\mathcal{O}(1)$. Under this assumption, the time and space complexity is $\mathcal{O}(|e|)$ for hierarchy-$\hat{\Gamma}$ and $\mathcal{O}(|e_1|+|e_2|)$ for hierarchy-$\ltimes\Gamma$. If the computation can *not* be inlined, we fall back to actually collecting the respective input tuples in the X bags; this means our algorithms degenerate to plain hierarchy merge join algorithms and their time and space complexities become $\mathcal{O}(|e_1|+|e_2|+|e_1 \ltimes e_2|)$. To obtain these results, an amortized analysis is needed to argue that the inner loops of the algorithms do not contribute to the overall complexity: Regarding hierarchy-$\hat{\Gamma}$, 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-$\ltimes\Gamma$, the loop bodies of l. 22 and l. 41 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 within each iteration, a similar argument applies.

6. EVALUATION

We explore the performance of our operators using a stand-alone single-threaded execution engine written in C++. It allows us to hand-craft query plans based on a push-based physical algebra. Our algorithms of Sec. 5 by design fit into this execution model by simply leaving out the outer for loops. Through careful use of C++ templating, GCC 5.2.1 with -O3 is able to translate the algebra expressions into efficient machine code with no visible operator boundaries within pipelines; thus, there is minimal friction loss through the algebra, and we get effective pipelining. We found the resulting code to be comparable in quality to what modern engines such as HyPer [18] and HANA Vora [21] emit. 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.

For our hierarchy table HT we use the schema from Fig. 1, where each tuple has a unique CHAR(8) ID and a TINYINT Weight randomly drawn from the small domain $[1, 100]$. We vary the table size |HT| from 10^3 to 10^6 to also cover loads that by far exceed L3 cache capacity: at 10^6, HT and its index use $\approx$ 218 MB. For the hierarchy index we compare two alternatives: [static] refers to the simple PPPL labeling scheme from Sec. 2, which does not support updates but is extremely fast and thus attractive for read-mostly analytic scenarios. [dynamic] refers to the BO-tree indexing scheme proposed in [8], where each Node is linked to two entries in a dynamic B$^+$-tree structure. We use the suggested configuration with *mixed* block sizes and *gap* back-links. It is a good allround fit for dynamic OLTP scenarios, although the support for updates comes at a cost of computationally non-trivial $\mathcal{O}(\log|\text{HT}|)$ index primitives and increased memory traffic. Other dynamic indexing schemes will of course show different characteristics (as studied in [8]); still, comparing dynamic vs. static gives us a good hint of the overhead to expect from accessing an external, dynamic index structure. All experiments use a generated forest structure Regular$\langle k\rangle$ where each tree is given $m = 10^4$ nodes and each inner node exactly k children. This way increasing |HT| does not affect the total height h. To assess the influence of the hierarchy shape, we compare very deep ($k = 2$, $h \approx 13.2$) trees to very shallow ($k = 32$, $h \approx 3.6$) trees.

Hierarchical Windows. To assess the bare performance of hierarchical windows, we run Stmt. IV (Sec. 3.3) with various expressions from Fig. 4 on a pre-materialized table Inp. Queries Q1 and Q2 compute Expr. 1a bottom up and top down, respectively and represent non-recursive computations. Q3 computes Expr. 4c and represents a structurally recursive computation. Q4 computes COUNT(DISTINCT Weight) bottom up and features a comparatively expensive duplicate elimination. For each query we measure alternative plans. All plans work on the same input Inp, which is prepared a priori as follows: We select the contents of HT (thus, |Inp| = |HT|), add a randomly populated INT Value field, project the required fields and 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 materialization of the query result. We compare the following plans, where applicable: (a) the straight translation into hierarchy-$\hat{\Gamma}$(Inp); (b) the alternative hierarchy-$\ltimes\Gamma$ (Inp, Inp), to assess the overhead over hierarchy-$\hat{\Gamma}$; (c) the hierarchy-$\bowtie\!-\!\Gamma$ approach of Sec. 5.1 with a preorder-based hierarchy merge join; (d) the $\bowtie\!-\!\Gamma$ approach with a nested loops join. As explained in Sec. 5.1, (c) is a natural baseline, whereas (d) would be the only option with hand-implemented encodings. We furthermore consider two plans based on a semi-naive least-fixpoint operator, which mimic SQL's recursive CTEs: (e) iterative uses repeated IS_CHILD hierarchy merge joins to first compute all $<$ pairs bottom up (Q1) or top down (Q2) and then performs the actual computation using sort-based grouping. (f) iterative* additionally applies sort-based "early grouping" within each iteration, inspired by [20]. This gives us a hint of the performance to expect from an exceptionally well-optimized RCTE or from a hand-crafted iterative stored procedure. We commonly see such procedures in real-world applications that still rely on trivial parent/child tables (known as *adjacency list model*, cf. Sec. 7). However, (e) and (f) are no general solutions; they work in our setup only because *all* HT nodes are present in Inp. Note also that plans (b)–(f) work only for *non-recursive* computations.

Fig. 6 shows the results, normalized with respect to the processed elements |Inp|. The red line indicates the speed of tuple-by-tuple copying a precomputed result table as the physical upper bound ($\approx$ 37.6M/s). In Q1–3 with static, $\hat{\Gamma}$ is remarkably close to this bound ($\approx$ 25.4M/s, or 67%). That non-recursive computations (Q1) using $\hat{\Gamma}$ are not slower than

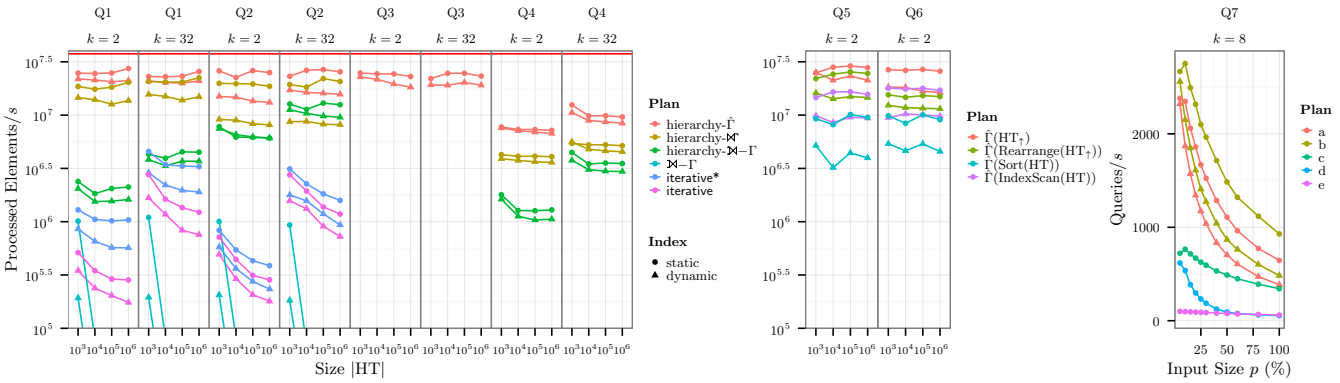


Figure 6: Experimental Results

recursive ones (Q3) comes at no surprise since the algorithm is identical. For both $\hat{\Gamma}$ and $\tilde{\bowtie}$, the top-down algorithms (Q2) are slightly slower than the bottom-up algorithms (Q1), as they cannot dismiss covered tuples as early and thus inherently issue more index calls. The duplicate elimination of Q4 is costly—both $\hat{\Gamma}$ and $\tilde{\bowtie}$ become roughly 3× to 4× slower over the trivial arithmetics of Q1–3. When comparing $e \tilde{\bowtie} e$ to $\hat{\Gamma}(e)$ over all queries Q1–4, 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. hierarchy-$\tilde{\bowtie}-\Gamma$ is significantly slower than $\tilde{\bowtie}$, mostly in bottom-up Q1 (e. g. ≈ 11× slower at $k = 2$) but also in top-down Q2 (≈ 3.5× at $k = 2$); the gap grows with the hierarchy height. This confirms the known "groupjoin advantage" also for the hierarchical case—in line with the reports on hash-based equi-groupjoins of [17]. hierarchy-$\tilde{\bowtie}-\Gamma$ is somewhat handicapped at Q1, as the hierarchy merge join algorithm of [1] we use is preorder-based; as preorder is more natural to top-down computations, it performs noticeably better at Q2. Interestingly, it is not slowed down as much at Q4 vs. Q1 as the others; the intermediate join apparently dominates the costs so that the subsequent processing-friendly sort-based grouping does not matter much. Thus, the overhead over hierarchy-$\tilde{\bowtie}$ is smaller at Q4, but still noticeable.

The iterative solutions are generally slow. Early aggregation helps much in the bottom-up case, where iterative* even approaches hierarchy-$\tilde{\bowtie}-\Gamma$ at $|\text{HT}| = 10^6$. In the top-down case, however, early aggregation does not help 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 versus static indexes, the more complex axis checks of the former are clearly noticeable; especially in top-down Q2, where inherently more axis checks are issued. Note our BO-tree is freshly bulkloaded; in practice the performance of most dynamic indexes tends to further degrade from incremental updates.

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 $\hat{\Gamma}$ and $\tilde{\bowtie}$ are practically indifferent. The intermediate join result of hierarchy-$\tilde{\bowtie}-\Gamma$ is somewhat proportional to h, so it is also affected to some extent (factor 2–3).

Increasing the hierarchy size $|\text{HT}|$ should slow down dynamic due to the $\mathcal{O}(\log |\text{HT}|)$ complexity of the index primitives. However, for the chosen block-based BO-tree index this apparently does not matter much in practice: the figures are practically indifferent to $|\text{HT}|$. One reason for this 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. hierarchy-$\tilde{\bowtie}-\Gamma$ and iterative are much more sensitive to $|\text{HT}|$ due to their growing intermediate results.

Note that the above experiments assess only $e_1 \tilde{\bowtie} e_2$ where $e_1 = e_2$, i. e., 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 completely in line with the linear time and space complexities of hierarchy-$\tilde{\bowtie}$, we omit them given the limited space.

Sorting. Being order-based, hierarchy-$\hat{\Gamma}$ and hierarchy-$\tilde{\bowtie}$ require pre- or post-ordered inputs. It is up to the cost-based optimizer to provide them by employing (a) explicit Sort operations via is-before; (b) ordered hierarchy index scans on the base table HT to establish the order in the first place; and (c) order-preserving operators such as hierarchy merge join to retain the order once established. (See [22] for the relevant techniques on maintaining *interesting orders*.) Even though this topic is orthogonal, we conducted some benchmarks. Queries Q5 and Q6 run Expr. 2b from Fig. 4 directly on HT. In the bottom-up case Q5, we compare $e_1 = \hat{\Gamma}(\text{HT}_{\text{post}})$ on an already post-ordered copy of HT, just like in Q1; $e_2 = \hat{\Gamma}(\text{Sort}_{\text{post}}(\text{HT}))$, a full sort; $e_3 = \hat{\Gamma}(\text{IndexScan}_{\text{post}}(\text{HT}))$, which accesses HT through a hierarchy index scan; and $e_4 = \hat{\Gamma}(\text{Rearrange}_{\text{post}}(\text{HT}_{\text{pre}}))$; mutatis mutandis in the top-down case Q6. The Rearrange operator consumes an already pre-ordered HT copy and employs a stack-based structural sorting algorithm similar to $\hat{\Gamma}$; its advantage is that it allows limited pipelining.

From the results in Fig. 6 we observe that full sorting is less expensive than one may expect (roughly 3× slower with static), considering that our algorithm is not multithreaded. Leveraging an index scan also helps much. But most interestingly, the "order-based sorting" of Rearrange is greatly superior to a full Sort, especially in the bottom-up static case: Rearrange closely approaches the "perfect" speed of e_0. This is again explained by pipelining effects and the favorable data locality in the already preordered inputs. Thus, our bottom-up algorithms are not restricted to postorder; they could be applied to preorder inputs as well at only moderate extra costs. To a slightly lesser extent this also applies to the preorder-based top-down algorithms.

Report Query. Having assessed hierarchical windows in isolation, we next look at a complete query, Q7. To emulate the setting of Stmt. I-a from Sec. 2, we use $|\mathrm{HT}| = 10^4$ and $k = 8$, and prepare a table Inp with only a *subset* of the hierarchy HT, namely p% of its 8751 leaf nodes, randomly chosen. At the heart, Q5 performs a bottom-up rollup as Q1, but additionally (a) needs a join/union with the relevant output nodes of HT, (b) computes the contribution in % of each node's x value to the parent's total, (c) carries 128 bytes of further payload through the computation, (d) outputs only the 3 upper levels (584 nodes), ordered in preorder, and visualizes the nodes' positions to the user by Dewey-style path strings. Such additional "stress factors" are commonly found in real-world queries. An example result line may be ['/A1/B1/C2', 125, 10%, *payload*], if the x value of '/A1/B1' is 1250. In SQL:

```
WITH T1 (Node, ID, Payload, x) AS (
  SELECT HT.Node, HT.ID, HT.Payload,
         SUM(Inp.Value) OVER (HIERARCHIZE BY HT.Node)
    FROM HT LEFT OUTER JOIN Inp ON HT.Node = Inp.Node ),
T2 (Node, ID, Payload, x, Contrib, Path) AS (
  SELECT Node, ID, Payload, x,
         RECURSIVE ( 100.0 * x / FIRST_VALUE(x) OVER w ),
         RECURSIVE VARCHAR (
          COALESCE(FIRST_VALUE(P) OVER w, '') || '/' || ID ) AS P
    FROM T1 WINDOW w AS (HIERARCHIZE BY Node TOP DOWN) )
SELECT Path, x, Contrib, Payload FROM T2
 WHERE LEVEL(Node) <= 3  -- φ
 ORDER BY PRE_RANK(Node)
```

We measure the following hand-optimized plans:

a. $\hat{\Gamma}(\mathrm{Rearrange}_{\mathrm{pre}}(\sigma_{\phi}(\hat{\Gamma}_{\mathrm{x}}(\mathrm{Sort}_{\mathrm{post}}(\mathrm{HT}_{\phi}) \cup \mathrm{Sort}_{\mathrm{post}}(\mathrm{Inp})))))$
b. $\hat{\Gamma}(\mathrm{Rearrange}_{\mathrm{pre}}(\mathrm{Sort}_{\mathrm{post}}(\mathrm{HT}_{\phi}) \lesssim_{\mathrm{x}} \mathrm{Sort}_{\mathrm{post}}(\mathrm{Inp})))$
c. $\mathrm{Map}(\bowtie(\bowtie(\Gamma_{\mathrm{x}}(\mathrm{Sort}_{\mathrm{pre}}(\mathrm{HT}_{\phi}) \bowtie \mathrm{Sort}_{\mathrm{pre}}(\mathrm{Inp})))))$
d. $\mathrm{Sort}(\mathrm{Map}(\bowtie(\bowtie(\Gamma_{\mathrm{x}}(\mathrm{HT}_{\phi} \bowtie \mathrm{Inp})))))$
e. $\mathrm{Iterative}_{\phi}(\mathrm{HT}, \mathrm{Inp})$

In all plans, σ_{ϕ} has been pushed down and is handled by an ordered index scan of HT. Plans a and b use our $\hat{\Gamma}$ and $\lesssim$ operators. The outer $\hat{\Gamma}$ handles both top-down computations and preserves the desired preorder. For Plan c we assume the hierarchical table model without our enhancements: It relies only on hierarchy merge joins $\bowtie$, i. e., the hierarchy-$\bowtie-\Gamma$ approach. Lacking our syntax extensions, a lot of manual "SQL labor" is involved: The upper 3 levels must be joined via two IS_PARENT joins and the path strings built by hand (the two outer $\bowtie$ and Map operators in c/d). For Plan d we assume a hand-implemented static PPPL-like labeling scheme. Lacking engine support, it can use only nested loops joins, i. e., the $\bowtie-\Gamma$ approach. 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 Q1–2. Although plans d–e are severely handicapped versus a–c, they are representative of the state of the art in real-world applications we encountered.

Fig. 6 shows the measured query throughput over varying p. The biggest pain point is the expensive sorting of Inp, which could be alleviated by parallel sorting. Nevertheless, we still see the merits of our proposed syntax and algorithms: Both $\hat{\Gamma}$ and $\lesssim$ reasonably handle the query, but the latter more naturally fits its binary nature. Their advantage over plain hierarchy-$\bowtie-\Gamma$ (c) is still visible, but less pronounced due to the damping effect of the sorting. It is not surprising that Plans c, d, and e—besides being unwieldy hand-crafted solutions—cannot hold up in terms of expressiveness and efficiency. Q7 is just one example query typically found in our application scenarios. We plan to study a wider range of application patterns in our future work.

7. RELATED WORK

Expressing Hierarchical Computations. While some query languages such as MDX [16] or XML/XQuery [10, 24] offer native support for hierarchical data and certain computations, our goal is to remain in the world of SQL [23]. Prior to the hierarchical tables of [2], a uniform data model and language for handling hierarchies in RDBMS was lacking. Earlier solutions [4] are therefore usually hard-wired to particular relational encodings, which largely dictate the computations that can be expressed: On the low end is the trivial *adjacency list model* [4] based on foreign key references to parent nodes, where recursion (see below) is required even for simple tasks. More sophisticated path- or containment-based encodings (e. g. [26, 13]) alleviate many tasks by allowing us to replace recursion by hierarchy joins, but computations are then limited to what join–group–aggregate statements can do. Another common "scheme" is the *leveled model*, where a denormalized table encodes a hierarchy with a fixed number of homogenous levels [19, 16]. Targeting this model in particular, SQL has a ROLLUP construct [12, 23] for simple sums, counts, and the like, but this is merely syntactic sugar for GROUPING SETS and again of limited expressiveness. The hierarchical table model relieves the user from dealing with the complexities and limitations of a hand-implemented encoding. Its abstract nature ensures that the provided constructs work with a multitude of indexing schemes on the query/update performance spectrum, as surveyed in [8]. Moreover, its main concept of a NODE field encapsulating the hierarchy provides attractive syntax opportunities (cf. Sec. 3).

Recursion in SQL. The only two common RDBMS-level mechanisms for working with recursively structured data are RCTEs [9, 23] and (iterative or recursive) stored procedures. [4] explores both. These mechanisms afford *generative* recursion and are thus more powerful than the *structural* recursion of our RECURSIVE expressions. But their power and generality also makes them difficult to handle and optimize. While the topic itself is old, recently Ordonez et al. [20] studied the optimization of linearly recursive CTEs with GROUP BY. They consider directed graphs, whereas our focus is specifically on tree structures. Unsurprisingly, our specialized algorithms easily outperform techniques for RCTEs (cf. Sec. 6). Also, the simple nature of structural recursion—where the recursion tree is predetermined—leaves more room for optimizations (as Sec. 4.3 outlines). Aside from performance one may ask whether RCTEs are at least "sufficient" in terms of expressiveness, i. e.: can RCTE-based recursion with GROUP BY emulate structural grouping? Alas, all our attempts to phrase such a computation in an iterative way—starting at the <-minimal tuples, then sweeping breadth-first over the input via <:—led us to very convoluted EXISTS subqueries. Also, GROUP BY is forbidden in an RCTE to enable the semi-naive fixpoint evaluation [9]. Even if GROUP BY could be used, it would not generally capture *all* relevant covered nodes in each iteration. Thus, for our use cases, the computational power of RCTEs is only of theoretical relevance.

Evaluating Aggregation Queries. Extensive literature exists on evaluating GROUP BY using either sort-based or hash-based methods [11]. Like sort-based grouping, our operators require ordered inputs and are order-preserving. *Groupjoin* [17, 15, 5] improves upon join–group–aggregate plans by fusing $\bowtie$ and Γ. While [17] discusses mainly hash-based equi-

groupjoins, [15, 7] consider the *non-equi* case, which is more comparable to our hierarchy-⋈ setting. Regarding `ROLLUP`, [12] discusses possible implementations. One approach uses a dedicated single-pass operator that reuses results of lower levels, which is similar in spirit to our approach. Regarding windowed tables [25], see [14] for a recent guide. Alas, techniques for standard windows cannot easily be adapted to our hierarchical windows due to their unique semantics.

Hierarchy-aware Operators. Since XML data is inherently hierarchical and often stored in relational tables, there is a significant body of work on querying native XML stores or XML-enhanced RDBMS. *Structural join* operators resembling self-merge-joins were studied in [26, 1, 13, 6]. Similar to our algorithms, they leverage an available (though hardwired) hierarchy encoding and maintain a stack of relevant intermediate results. Not all techniques from the XML world fit into our setting, however: Some of the more sophisticated join operators were designed to work directly on an indexed XML document. This enables advanced optimizations such as skipping [13]. In contrast, our operators are usually applied to *arbitrary* input tables with a `NODE` field (e. g. Inp1) rather than the hierarchical table (e. g. HT) itself. As indexing Inp1 on the fly seems infeasible, we rely only on HT's index, which renders many of the optimizations inapplicable. While we could e. g. adapt Staircase Join [13] for cases where the computation runs directly on HT, this would benefit only a limited number of queries. Beyond binary structural joins, powerful tree pattern matching operators (e. g. twig joins) were proposed in the XML context; but these are beyond the requirements for handling hierarchical data in RDBMS.

8. CONCLUSION

Expressing hierarchical computations in RDBMS has always been severely impeded by data model and language issues, and even when possible, convoluted RCTEs or procedure calls rendered an efficient evaluation very difficult. We resolve this situation by exploiting the opportunities of the hierarchical table model [2] regarding expressiveness and engine support. The `NODE` type and SQL's windowed tables turn out to be a natural fit. Together with structural recursion, a useful class of computations can be expressed concisely and intuitively. For their evaluation we propose order-based, index-assisted structural grouping operators. They rely entirely on pre- and post-order primitives and thus work with a multitude of indexing schemes. Our experiments confirm their merits over conventional approaches, which result from their robust linear space and time complexities and their computational power. As part of future work, we plan to investigate rewrite rules for optimizing structural grouping plans, adaptions of our algorithms to acyclic digraphs, and their interplay with temporal and multidimensional data. Based on experiences with applications at SAP, we will also consider refinements to our SQL extensions. Altogether this novel functionality promises to greatly simplify and speed up the many applications that deal with hierarchies, in business software and beyond, by allowing them to push even more logic down to the RDBMS layer.

9. REFERENCES

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, et al. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pp. 141–152, 2002.

[2] R. Brunel, J. Finis, G. Franz, N. May, A. Kemper, T. Neumann, and F. Faerber. Supporting hierarchical data in SAP HANA. In *ICDE*, pp. 1280–1291, 2015.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, pp. 310–321, 2002.

[4] J. Celko. *Trees and Hierarchies in SQL for Smarties, Second Edition*. Morgan Kaufmann, 2012.

[5] D. Chatziantoniou, T. Johnson, M. Akinde, and S. Kim. The MD-join: An operator for complex OLAP. In *ICDE*, pp. 524–533, 2001.

[6] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, et al. Twig^{2}Stack: Bottom-up processing of generalized tree pattern queries over XML documents. In *VLDB*, pp. 283–294, 2006.

[7] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *DBPL Workshop*, 1995.

[8] J. Finis, R. Brunel, A. Kemper, T. Neumann, N. May, and F. Faerber. Indexing highly dynamic hierarchical data. In *VLDB*, pp. 986–997, 2015.

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

[10] C. Gokhale, N. Gupta, P. Kumar, L. V. S. Lakshmanan, et al. Complex group-by queries for XML. In *ICDE*, pp. 646–655, 2007.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, et al. Data Cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[13] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a relational DBMS to watch its (axis) steps. In *VLDB*, pp. 524–535, 2003.

[14] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient processing of window functions in analytical SQL queries. In *VLDB*, pp. 1058–1069, 2015.

[15] N. May and G. Moerkotte. Main memory implementations for binary grouping. In *XSym*, pp. 162–176, 2005.

[16] Microsoft Corp., SQL Server 2012 Product Documentation – MDX Reference. msdn.microsoft.com/en-us/library/ms145506.aspx, July 2016.

[17] G. Moerkotte and T. Neumann. Accelerating queries with Group-By and Join by Groupjoin. *VLDB*, pp. 843–851, 2011.

[18] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, pp. 539–550, 2011.

[19] Oracle Corp., Oracle 9i OLAP User's Guide, Release 9.2, 2002. docs.oracle.com/cd/A97630_01/olap.920/a95295.pdf.

[20] C. Ordonez. Optimization of linear recursive queries in SQL. In *TKDE*, pp. 264–277, 2010.

[21] SAP SE. Solutions – SAP HANA Vora. go.sap.com/product/data-mgmt/hana-vora-hadoop.html, July 2016.

[22] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pp. 23–34, 1979.

[23] Information technology — database languages — SQL. ISO/IEC JTC 1/SC 32 9075, 2011.

[24] N. Wiwatwattana, H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. X^3: a cube operator for XML OLAP. In *ICDE*, pp. 916–925, 2007.

[25] F. Zemke, K. Kulkarni, A. Witkowski, and B. Lyle. Introduction to OLAP functions. *ANSI Document NCITS H2-99-154r2*, 1999.

[26] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pp. 425–436, 2001.