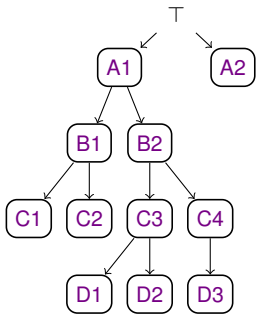
Supporting Hierarchical Data in SAP HANA

Robert Brunel* Jan Finis*
Gerald Franz† Norman May† Alfons Kemper* Thomas Neumann* Franz Faerber†
* Technische Universität München, Garching, Germany
firstname.lastname@cs.tum.edu
† SAP SE, Walldorf, Germany
firstname.lastname@sap.com

***Abstract*—Managing hierarchies is an ever-recurring challenge for relational database systems. Through investigations of customer scenarios at SAP we found that today's RDBMSs still leave a lot to be desired in order to meet the requirements of typical applications. Our research puts a new twist on handling hierarchies in SQL-based systems. We present an approach for modeling hierarchical data natively, and we extend the SQL language with expressive constructs for creating, manipulating, and querying a hierarchy. The constructs can be evaluated efficiently by leveraging existing indexing and query processing techniques. We demonstrate the feasibility of our concepts with initial measurements on a HANA-based prototype.**



BOM

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

Fig. 1. Example hierarchy and corresponding table

I. Introduction

Hierarchical relations appear in virtually any business application, be it for representing organizational structures such as reporting lines or geographical divisions, the structure of assemblies, taxonomies, marketing schemes, or tasks in a project plan. But as tree and graph structures do not naturally fit into the flat nature of the traditional relational model, handling such data today remains a clumsy and unnatural task for users of the SQL language. We have investigated a number of SAP applications dealing with hierarchies and collected their typical requirements (Sec. II). Within SAP's Enterprise Resource Planning system, hierarchies are used in the human resources domain to model reporting lines of employees, in asset management to keep track of production-relevant assets and their functional locations (e. g., plants, machines, machine parts, tools, equipment), and in materials planning to represent an assembly of components into an end product, a so-called *bill of materials* (BOM). Due to the limitations of the relational model and SQL, logic for hierarchy handling within these applications has mostly been written in ABAP and therefore runs within the application server. We have identified almost a dozen different implementations of more or less the same hierarchy-handling logic, which is unfortunate not only from an interoperability and maintainability point of view. In most cases, hierarchies are represented in the database schema using a simple relational encoding, and converted into a custom-tailored format within the application, if needed. The most widespread encoding is a self-referencing table resembling an adjacency list. It is well-known in the literature (e. g., [1]) under the term *Adjacency List model*. Fig. 1 shows an example instance: table `BOM` represents a bill of materials. Field `id` uniquely identifies each part. The hierarchical relationship is established by a self-reference `pid` associating each row with its respective parent row, the part containing it. The resulting hierarchy is displayed on the left.

The only standard hierarchy-handling tools that SQL-based DBMSs today offer are *Recursive Common Table Expressions* (RCTEs) and some means to define custom stored procedures for working on adjacency lists like `BOM`. But these tools suffer from certain usability and performance deficiencies. The conventional alternative is to abandon the Adjacency List model and implement a more suitable encoding scheme manually, either on the relational level [1] or within the application (i. e., ABAP), relinquishing any kind of engine support either way. We consider neither RCTEs nor alternative encoding schemes nor any other purported solution we investigated sufficient to meet the common requirements that we have identified (Sec. III). These requirements call for a solution that seamlessly integrates hierarchical and relational data by combining an expressive front end (new language constructs) with a powerful back end (indexing and query processing techniques), without departing farther than necessary from the philosophy of the relational model. Such a solution is missing to date.

That is not to say the research community has not given enough attention to the underlying technical challenges. In fact, the problem of storing and indexing hierarchical or recursively structured data has been studied deeply in the past decade, in the course of research efforts in the domains of XML and semi-structured databases. While recognizing that many challenges of indexing and query processing have been successfully tackled previously (e. g., by RCTEs, encoding schemes, and structural joins), we believe there is an open opportunity to reconcile the ideas and techniques from the mentioned areas into a general framework for working with hierarchies in a relational context.

The core concept of our framework is to encapsulate structural information into a table column of a new special-purpose data type `NODE` (Sec. IV). Our extensions to the SQL query language comprise a small yet essential set of built-in

functions operating on the NODE values (Sec. V), plus DDL and DML constructs for obtaining and manipulating a NODE column in the first place (Sec. VI). Altogether the language elements cover the typical requirements and blend seamlessly with the look and feel of SQL, as we illustrate on some advanced scenarios (Sec. VII). The back-end design we present is geared to the in-memory column store of SAP HANA [2]. From an early stage on, HANA has been envisioned as a multi-engine query processing environment offering abstractions for data of different degrees of structure, and the idea of hierarchical tables fits well in this spirit. That said, our framework is generic in such a way that it can be adapted for different RDBMSs, including classical row stores. Our HANA-based prototype provides insights into architectural and implementation aspects (Sec. VIII) as well as a proof of concept and initial performance characteristics (Sec. IX).

II. Requirements Review

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

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

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

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

#4 *Support explicit modeling of hierarchical data.* In a plain relational database, a schema designer initially has to decide on how to represent a hierarchy (e. g., as an adjacency list). Even though relational tree encodings are well understood nowadays [1], carefully choosing and implementing one still requires advanced knowledge. What's more, a tree encoding generally disguises the fact that the corresponding table contains a hierarchy. What is needed is a way to *explicitly* model hierarchies in the schema, using abstract, special-purpose data definition (DDL) constructs that hide the fiddly storage details.

#5 *Provide front- and back-end support for data manipulation*. Hierarchies in business scenarios are usually *dynamic*. In some cases manipulations involve only insertions or removals of individual leaf nodes, while in other cases more complex update operations are required, such as collective relocations of large subtrees. As an example, consider the enterprise assets (EA) hierarchy of an automotive company that contains a lot of machines and robots and relocates them whenever a new production line is established. Through a previous analysis of a customer's EA hierarchy, we found that as much as 31% of all recorded update operations were subtree relocations [3]. Consequently, the system must provide an interface (DML) and efficient back-end support for both leaf and subtree manipulations.

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

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

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

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

III. From Status Quo to Hierarchies in HANA

Do we need yet another solution? The problem at hand is almost ancient in terms of DBMS history. *Hierarchical Queries* [4], a proprietary SQL extension for traversing recursively structured data in the Adjacency List format, have been a part of Oracle Database for about 35 years. In 1999, Recursive Common Table Expressions (RCTEs) [5], [6] brought standard SQL a general and powerful facility for recursive

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

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

Fig. 2. Using SQL/XML to generate XML from a relational table (top, [7]) and to evaluate XQuery (bottom, [8])

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

XML. Storing and querying XML—an inherently hierarchical data format—and the idea of bridging the technology stacks of the relational and XML worlds have received a lot of attention in the research community. One research track pursues the idea of adapting RDBMSs for storing XML fragments and evaluating XPath and XQuery based on relational query processing techniques [9], [10]. Beyond that, the idea of joining the XML and relational data models in order to enable queries over both tables and XML documents has resulted in the SQL/XML standard [11], which integrates XML support into SQL. It has been implemented by prominent vendors [7], [12], [13]. Fig. 2 depicts two example SQL/XML queries. So-called publishing functions enable the user to generate XML from relational input data (top example). Conversely, XQuery can be used within SQL statements to extract data from an XML fragment and produce either XML or a relational view (bottom example). While SQL/XML is the tool of choice for working with XML within RDBMSs, it cannot hide the fact that the underlying data models were not designed for interoperability in the first place. "Casting" data from a relational to an XML format or vice versa induces a lot of syntactic overhead, as the many `XML...` clauses cluttering the top example of Fig. 2 attest. In addition, SQL/XML requires users to know both data models and the respective query languages, which is a challenge to SQL-only users. Since our requirements #1 and #2 mandate that the data model and query language should blend seamlessly with SQL, we do not consider SQL/XML a candidate for general hierarchy support in HANA. That said, we recognize that many techniques from the XML field, such as join operators and labeling schemes, can be leveraged for our purpose.

`hierarchyid` is a variable-length system data type introduced in Microsoft SQL Server 2008 [14], [15] whose values represent a position in an ordered tree using ORDPATH [16], a compact and update-friendly path-based encoding scheme. The feature is apparently a by-product of SQL Server's XML support and as such a good demonstration that XML technology can be leveraged for more general uses. The data type provides methods for working with nodes, such as `GetLevel` and `IsDescendantOf`. Nodes can be inserted and relocated by generating new values using, for example, methods `GetReparentedValue` and `GetDescendant`. From a syntax and data model perspective, this is the related work that is most similar to what we present in this paper. However, there are several major differences to our design: The `hierarchyid` field is provided as a simple tool for modeling a hierarchy; yet, a collection of rows with such a field does *not* necessarily represent a valid hierarchy. It is up to the user to generate and manage the values in a reasonable way. By design, the system does not enforce the structural integrity of the represented tree. For example, it does not guarantee the uniqueness of generated values, and it does not prevent accidentally "orphaning" a subtree by deleting its parent node. In contrast, we require the system to ensure structural integrity at any time (Req. #7), so that queries on hierarchies will not yield surprising results. Of course, this design choice comes at a price, as consistency has to be checked on each update. As another difference, we opt to provide flexibility regarding the underlying indexing scheme. Rather than hardwiring a particular scheme such as ORDPATH, our design allows a scheme to be chosen according to the application scenario at hand. ORDPATH has the inherent deficiency that relocating a subtree incurs changes to *all* `hierarchyid` values in that subtree (cf. Req. #5).

Hierarchical Queries in Oracle Database [4] extend the `SELECT` statement by the constructs `START WITH`, `CONNECT BY`, and `ORDER SIBLINGS BY`. The wording of these constructs and the related built-in functions and pseudo-columns such as `ROOT`, `IS_LEAF`, and `LEVEL` clearly hint at their intended use for traversing hierarchical data in the Adjacency List format. The underlying recursion mechanism is conceptually similar to RCTEs. Most functionality can in fact be expressed straightforwardly using RCTEs [17], so the discussion in the following paragraphs applies to Hierarchical Queries as well.

Recursive Common Table Expressions are a standard tool that can, among other things, be used for working with a table in the Adjacency List format. We refer to this particular combination as the "RCTE-based approach". As a generic mechanism, RCTEs do in fact have interesting uses far beyond traversing hierarchical data, and we by no means intend to render them obsolete. To convey an impression of how our design and the RCTE-based approach differ, we revisit the BOM example from Fig. 1. Consider the following query, which is derived from a customer scenario:

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

In the example BOM, the qualifying node triples would be $(\mathrm{B2}, \mathrm{D2}, \mathrm{C3})$, $(\mathrm{B2}, \mathrm{D2}, \mathrm{A1})$, and $(\mathrm{B1}, \mathrm{C2}, \mathrm{A1})$. This query is not entirely trivial in that it involves not only two, but three nodes that are tested for hierarchical relationships. Fig. 3 shows an RCTE-based solution. It selects the `id` and a `payload` of each node. We use two RCTEs: one starting from an engine e and navigating downwards from e to r, the other navigating upwards from r to c. Obviously, the statement is not particularly intelligible to readers, and somewhat tedious to write down in

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

Fig. 3. Example hierarchical query, expressed using two RCTEs.

the first place. What's more, it is not the only way to solve the problem. An alternative would be to fully materialize all ancestor/descendant combinations (u, v) using a single RCTE and then work (non-recursively) on the resulting table. This option is generally inferior due to the large intermediate result, though it might be feasible if only a small hierarchy is involved. The point is that it is up to the user to choose the most appropriate strategy for answering the query. The first choice to make is the basic approach to use: one RCTE materializing all pairs versus two RCTEs as in the example. The second choice is the join direction to proceed in: towards the root versus away from the root. The query optimizer is tightly constrained by the approach the user is prescribing. In fact, the query statement is *imperative* rather than declarative: A bad choice from the user's side can easily result in incorrect answers or severe performance penalties.

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

In addition, the RCTE-based approach bears some inherent inefficiencies. We give two examples: First, note that even though in Fig. 3 we are interested only in qualifying ancestor/descendant pairs (e, r) and (r, c), but *not* in any nodes in between, the RCTE necessarily touches all intermediate nodes anyway. Second, attributes of interest to the user (`payload` in the example) must often be materialized early and carried along throughout the recursion, which is costly.

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

Fig. 4. The example query from Fig. 3, expressed using the proposed SQL language extensions.

From RCTEs to our syntax. Fig. 4 shows how the example query is expressed in the syntax we introduce in sections IV and V. In a nutshell, the `pos` field of type `NODE` identifies a row's position in the depicted hierarchy, which makes `BOM` a *hierarchical table*. Without delving into details, we illustrate three major cornerstones of our design: First, with the RCTE-based approach, the task of "discovering" and navigating the hierarchy structure on the one hand, and the task of actually using the hierarchy to compute hierarchical properties of interest on the other hand, are inseparably intertwined. By contrast, the query statement in Fig. 4 makes use of an *available* hierarchical table exposing the `pos` field. The hierarchy structure is known ahead and persisted. The task of querying the hierarchy is cleanly separated from the task of specifying and building the hierarchy structure. Thus, any duplication of "discovery logic" in user code is avoided. Second, unlike the generic RCTE mechanism, our syntax is particularly tailored for working with hierarchies. This way we can provide increased intelligibility, user-friendliness, and expressiveness, and we can employ particularly tuned data structures and algorithms on the back-end side. Third, our syntax states the hierarchical relationships clearly and in a *declarative* way (e. g., `IS_DESCENDANT`). This allows the query optimizer to reason about the user's intent and pick an optimal evaluation (i. e., join) strategy and direction.

The example shows how we achieve requirements #1, #2, and #3: Our syntax blends with SQL (#1), as we stick to joins and built-in functions to provide all required query support (#3). As a corollary, the syntactic impact is minimal (#2). In fact, a hierarchy query does not need *any* extensions to the SQL grammar. Still, the syntax is highly expressive (#2): the query in Fig. 4 reads just like the English sentence defining it.

IV. Hierarchical Tables: Our Model

Basic Terms. We use the term *hierarchy* to denote an *ordered, rooted, labeled tree*. The tree in Fig. 1 is an example. *Labeled* means each vertex in the tree has a label, which represents the attached data. *Rooted* means a specific node is marked as root, and all edges are conceptually oriented away from the root. We require that every hierarchy contains by default a single, virtual root node, which we denote by $\top$ and call the *super-root*. As a virtual node, $\top$ is hidden from the user. The children of $\top$ are the actual roots in the user data. Through this mechanism we avoid certain technical complications in handling empty hierarchies as well as hierarchies with multiple roots, so-called *forests*. Furthermore, a hierarchy is *ordered*, that is, a total order is defined among the children of each node. That said, for many applications the relative order of siblings is actually not relevant. While we recognize this use case by

providing order-indifferent update operations, the system always maintains an internal order. This way order-based functionality such as preorder ranking is well-defined and deterministic.

Hierarchical Tables. In a database context, a hierarchy is not an isolated object but rather closely tied to an associated table. A hierarchy has exactly *one* associated table. (Of course, additional tables can be tied to a hierarchy by using joins; cf. Sec. VII.) Conversely, a table might have multiple associated hierarchies. In Fig. 1, for instance, table BOM has one associated hierarchy, which arranges its rows in a tree, thus adding a *hierarchical dimension* to BOM. We call a table with at least one associated hierarchy a *hierarchical table*. Let H be a hierarchy attached to a table T. Each row r of T is associated with *at most one* node v of H, so there may also be rows that do not appear in the hierarchy. Conversely, each node except for $\top$ is associated with *exactly one* row of T. The values in the fields of r can be regarded as labels attached to v or to the edge onto v. Besides the tree structure and a node–row association, H conceptually does not contain any data. A user never works with the hierarchy object H itself but only works with the associated table T. Consequently, a row-to-node handle is required to enable the user to refer to the nodes in H. Such a handle is provided by a column of type NODE in T.

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

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

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

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

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

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

Fig. 5. Projecting hierarchy properties of BOM.

V. Querying Hierarchies

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

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

- LEVEL(v) — The number of edges on the path from $\top$ to v.
- IS_LEAF(v) — Whether v is a leaf, i. e., has no children.
- IS_ROOT(v) — Whether v is a root, i. e., its parent is $\top$.
- PRE_RANK(v) — The preorder traversal rank of v.
- POST_RANK(v) — The postorder traversal rank of v.

Fig. 5 shows the result of projecting all these properties for BOM. The values of LEVEL, PRE_RANK, and POST_RANK are 1-based. There are certain more or less obvious equivalences. For example, IS_ROOT(v) is equivalent to LEVEL(v) = 1 and thus redundant, strictly speaking. However, for the sake of convenience and expressiveness we do *not* aim for a strictly orthogonal function set.

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

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

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

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

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

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

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

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

- IS_PARENT(u,v) — whether u is a parent of v.
- IS_CHILD(u,v) — whether u is a child of v.
- IS_SIBLING(u,v) — whether u is a sibling of v, i. e., has the same parent.
- IS_ANCESTOR(u,v) — whether u is an ancestor of v.
- IS_DESCENDANT(u,v) — whether u is a descendant of v.
- IS_PRECEDING(u,v) — true iff u precedes v in preorder and is not an ancestor of v.
- IS_FOLLOWING(u,v) — true iff u follows v in preorder and is not a descendant of v.

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

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

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

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

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

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

VI. Creating and Manipulating Hierarchies

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

A. Deriving a Hierarchy from an Adjacency List

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

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

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

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

2) Perform the join

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

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

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

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

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

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

The HIERARCHY syntax is intentionally close to an RCTE and even more so to Oracle Hierarchical Queries. (The self-join via *parent_name* is comparable to a CONNECT BY via PRIOR in a Hierarchical Query.) However, the semantics are quite different in that by design only a *single* self-join is performed on the input rather than a recursive join. As our experiments show, this allows for a very efficient evaluation algorithm compared to a recursive join. Furthermore, there is a major conceptual difference to the mentioned approaches: The HIERARCHY expression does nothing more than define a hierarchy. That hierarchy can be queried by wrapping the expression into a SELECT statement. In contrast, a RCTE both defines and queries a hierarchy in one convoluted statement. We believe that separating these two aspects greatly increases comprehensibility. As an example, consider again the BOM of Fig. 1. The following statement uses a CTE to derive the pos column from id and pid, then selects the id and level of all parts that appear within part C2:

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

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

B. Hierarchical Base Tables

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

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

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

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

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

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

C. Manipulating Hierarchies

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

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

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

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

For example, we can add a node B3 as new child of A2 into the hierarchy of Fig. 1 like this:

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

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

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

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

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

VII. Advanced Customer Scenarios

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

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

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

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

```
CREATE TABLE Part (
   id INTEGER PRIMARY KEY,
   kind VARCHAR(16),
   price INTEGER, ...   -- master per-part data
)
CREATE TABLE BOM (
   node_id INTEGER PRIMARY KEY,
   HIERARCHY pos,
   part_id INTEGER,   -- a node is a part (N:1)
   FOREIGN KEY (part_id) REFERENCES Part (id),
   ...   -- additional node attributes
)
```

Heterogeneous Hierarchies. Often, entities of different types are mixed in a single hierarchy. "Different types" means that the entities are characterized by different sets of attributes. Especially in XML documents, it is very common to have various node types (i. e., tags with corresponding attributes), and XPath expressions routinely interleave navigation with filtering by node type (so-called node tests). The SQL way of modeling multiple entity types is to define a separate table per entity type, each with an appropriate set of columns. Returning to our BOM, we further enhance the `Part`–`BOM` data model with master data specific to engines:

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

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

While working with a BOM, the user can use type-specific part attributes for filtering purposes simply by joining in the

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

Fig. 6. Querying a heterogeneous hierarchy

corresponding master data. As an example, suppose that fittings by manufacturer X have been reported to outwear too quickly when used in combination with engines more powerful than 700 watts, and we need to determine the compounds that contain this hazardous combination in order to issue a recall. Fig. 6 shows the solution. Note that the `BOM`–`Engine` join implies the test that node `e` is of kind `'engine'`.

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

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

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

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

Note the straightforward reading direction, which intuitively matches the direction of navigation in the hierarchy. This example and the one from Fig. 6 in particular show how our language extensions maintain the join "look and feel" of SQL, so even large queries look familiar to SQL programmers.

VIII. Architecture and Implementation Aspects

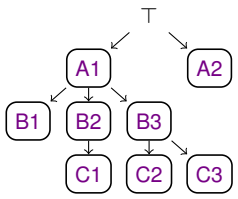
On the back-end side, the foundation for implementing the functionality described in sections V and VI is the *hierarchy indexing scheme* underlying each hierarchical dimension. As Requirement #8 anticipates, no single scheme can serve all application scenarios equally well; there is no "one size fits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

A. Hierarchy Indexing Schemes

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

Indexing schemes of varying complexity and sophistication are conceivable: Among the simplest indexing schemes are those based on *labeling schemes*; they are "simple" in that the labels can be stored directly in the `NODE` column (and possibly indexed using ordinary database indexes); no special-purpose data structures are required. Labeling schemes have been studied extensively in the XML context. Two prominent subcategories are *order-based schemes* as studied by Grust et al. [18], and *path-based schemes* such as ORDPATH [16]. In our prototype we have implemented a simple yet effective order-based variant: the pre/size/level scheme (PSL) [9], where we label each node with its preorder rank, subtree size, and level. We have also implemented a path-based scheme comparable to ORDPATH. Fig. 7 depicts the `NODE` column for an example hierarchy using these schemes.

Other, more sophisticated indexing schemes rely on auxiliary structures. Examples are BOXes [19] and our own DeltaNI [3]; our prototype incorporates the latter. Both represent the hierarchy information in special-purpose data structures, and the `NODE` column contains handles into those structures. The figure shows a possible `NODE` column for DeltaNI but omits the auxiliary delta structures.



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

Fig. 7. An example hierarchy and the contents of the NODE column using different indexing schemes

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

The indexing scheme is meant to be chosen by the DBMS per hierarchical dimension, transparently to the user. The user indirectly influences the choice through the UPDATES option (Sec. VI). Our prototype decides as follows: For derived hierarchies, which are by design static, and for immutable hierarchical tables (UPDATES = BULK), the obvious choice is PSL. If the user requires support for complex updates (SUBTREE), as well as for system-versioned tables, we choose DeltaNI. For ordinary, non-versioned tables, and if the user settles for simple updates (NODE), we resort to the path-based scheme.

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

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

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

B. Hierarchy-Aware Join Operators

Like functions, binary predicates such as IS_DESCENDANT can be translated into invocations of the underlying index. But this is not adequate if they are used as join conditions, since the query optimizer would have to resort to nested-loops-based join evaluation. Therefore, we enhance the optimizer such that joins involving a hierarchy predicate are translated into efficient hierarchy-aware physical join operators. Various hierarchy-aware join operators have been proposed in the literature, mostly for XPath processing [20], [21], [22]. Basically any of these operators can be used in our setting, with slight adaptions to account for SQL semantics. An XPath axis step, for example, is implicitly a semi-join and performs duplicate elimination. With SQL, we have to support general joins, and duplicate elimination is not necessary in the default case. For our prototype we have adapted the Staircase Join [21] to support all join axes and work with the mentioned indexing schemes.

C. Bulk-Building

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

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

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

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

(Step b.) β first materializes all edges into an array and then sorts this array by r_P. Because all row IDs r_P are in the range $0, \ldots, n-1$, where n is the number of rows in T, we can use

a "perfect" *bucket sort* algorithm with $n+1$ buckets, which is much faster than a general-purpose sort algorithm. Edges without a parent ($r_P = \texttt{NULL}$) are put into bucket n. By always pushing values to the back of a bucket, we achieve *stable* sorting, that is, the relative order among rows with identical r_P values (and thus the `SEARCH BY` *order*, if specified) is preserved.

(Step c.) After bucket sorting, β builds the hierarchy index during a depth-first traversal of the edges. The traversal is straightforward: We start with entries in bucket n, which correspond to roots in the hierarchy. Since a bucket B_i contains all rows with $r_P = i$, we can look up the children of a node r_P by inspecting its respective bucket.

For each node we visit during the traversal, we incorporate a corresponding entry into the indexing scheme and add a value to the `NODE` column. This is the only index-specific part of the algorithm. For example, with the PSL scheme we track the current *pre-rank* and *level* for each visited node during the traversal (the *pre-rank* and *level* values are inserted before visiting children, the *size* after visiting children) and encode them into the corresponding `NODE` field. With DeltaNI, we add an entry to the auxiliary structure and insert a handle to this entry into the `NODE` field for each visited node.

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

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

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

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

Fig. 8. Bulk-building performance

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

Fig. 9. Query performance

IX. Experiments

Although this paper focuses on concepts rather than on the performance characteristics of alternative implementations, we conducted an experiment to demonstrate that an efficient evaluation of our language constructs is possible. We derived a BOM hierarchy from the materials planning data of a large SAP customer with a few million nodes/rows. The original non-hierarchical table encodes the hierarchy structure in the Adjacency List format. It contains an `INTEGER` primary key and an `INTEGER` column referencing the superordinate part. For the equivalent hierarchical table, we employ the PSL indexing scheme. In order to assess performance on varying hierarchy sizes, we scale the data by removing or replicating nodes, covering data sizes that easily fit into cache (10^3) as well as sizes that by far exceed cache capacity (10^7). The benchmark is executed on an Intel Core i7-4770K CPU at 3.50 GHz, running Ubuntu 14.04.

Fig. 8 shows measurements for deriving a hierarchy from the adjacency list using our bulk-build algorithm: the times of the three steps of the algorithm on the top; the total time of all steps together below that; and lowermost, for purposes of comparison, the time of a recursive join over the super-part column. Such a recursive join could be used to implement the recursive variant of β, as outlined in the previous section. Note that the measured recursive join does not perform duplicate elimination (i. e., cycle elimination), which would make it considerably slower. The table unveils the most expensive step of the bulk-build process: the initial outer hash join building the edge list (a.). In contrast, all steps of the bulk-build operator β together (b. and c.) take only around one third of the time of the hash join. We therefore conclude that the proposed bulk-building mechanism is indeed very efficient. Furthermore, we see that executing an ordinary join is considerably faster than the recursive join, especially so for large hierarchies, so the recursive variant of β is in most cases inferior to the non-recursive variant.

We measured query performance by executing the query from Fig. 4 on the hierarchy and comparing it with the equivalent RCTE from Fig. 3 as baseline. We use a 16 byte payload column, so the size of a result row containing 3 `INTEGER` keys and 3 payload fields is 60 byte. Fig. 9 includes the runtimes of the two algorithms and the sizes of the result sets. The last two rows show the results for analogous measurements with the `INTEGER` key replaced by a `CHAR(16)` key. As we see in the table, the hierarchy-aware join (HAJoin) easily outperforms

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

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

X. Conclusion

Our work has been motivated by customer demand and findings from our investigation of typical requirements of SAP applications featuring hierarchical data. Our analysis leads us to conclude that the conventional approaches to handling such data—particularly recursive CTEs—are not fully satisfactory to meet the requirements. As a solution, we propose to enhance the relational model to incorporate hierarchies by means of a new data type `NODE`. This data type opaquely represents the hierarchy structure without mandating a specific encoding, in order to leave the system full flexibility in choosing the most appropriate indexing scheme. We introduce extensions to the SQL language that allow the user to specify queries over hierarchical data in a concise and expressive manner. The syntax extensions are minimal in that they rely mostly on built-in hierarchy functions and predicates operating on `NODE` values. Because of this, SQL programmers can adapt easily to the new syntax, and its integration into an existing RDBMS is straightforward. We propose efficient bulk-update operations for legacy applications, as well as fine grained update operations for greenfield applications. On the back-end side, we incorporate well-proven techniques from relational and XML database research.

The proof-of-concept HANA-based prototype we have implemented shows the feasibility and performance potential of the language. Once we have completed a customer-ready implementation, our plan is to conduct a user study among customers for a thorough evaluation of the expected benefits regarding usability and actual performance.

References

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

[2] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database—an architecture overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, 2012.

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

[4] Oracle Database SQL language reference 12c release 1 (12.1). E17209-15. Oracle Corp. [Online]. Available: http://docs.oracle.com/cd/E11882_01/server.112/e26088/toc.htm

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

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

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

[8] A. Eisenberg and J. Melton, "Advancements in SQL/XML," *SIGMOD Rec.*, vol. 33, no. 3, 2004.

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

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

[11] *Information technology — Database languages — SQL, Part 14: XML-Related Specifications (SQL/XML)*, ISO/IEC JTC 1/SC 32 Std. 9075-14, 2011.

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

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

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

[15] Books Online for SQL Server 2014 – Database Engine – Hierarchical Data. Microsoft Corp. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms130214.aspx

[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] K. Morton, K. Osborne, R. Sands, R. Shamsudeen, and J. Still, *Pro Oracle SQL*, 2nd ed. Apress, 2013.

[18] T. Grust, "Accelerating XPath location steps," in *SIGMOD*, 2002.

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

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

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

[22] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching," in *SIGMOD*. ACM, 2002.