PUBLIC
SAP HANA Platform 2.0 SPS 06
Document Version: 1.0 – 2021-12-03

SAP HANA Hierarchy Developer Guide

THE BEST RUN SAP

Content

1 SAP HANA Hierarchy Developer Guide

This guide explains how to use the hierarchy functions that are an integral part of SAP HANA core functionality.

The information in the guide is organized as follows:

- Introduction
- Generating Hierarchies
- Hierarchy Attributes
- Navigating Hierarchies
- Hierarchy Metadata Views
- Comparison to Related SQL Database Concepts
- Typical Hierarchy Use Cases and How-To Recipes
- Migrating From Other RDBM Systems
- Performance Recommendations
- Troubleshooting

2 Introduction

Hierarchy functions are an integral part of SAP HANA core functionality.

They expand the SAP HANA platform with native support for hierarchy attribute calculation and navigation, and allow the execution of typical hierarchy operations directly on relational data, seamlessly integrated into the SQL query.

Hierarchy functions explicitly target the multitude of hierarchical data that already exist in most real-world database installations, but is not explicitly defined as hierarchical data. This includes, for example, company reporting or department structures, categorized document stores, hierarchy roles and permission systems, or even plain address data. Hierarchy functions enable users to efficiently query this data in a consistent and structured way, even though the layout of the source data can be very diverse.

This document provides a practical guide to developers how to use SAP HANA hierarchy functions to efficiently solve typical tasks when dealing with hierarchical relations.

2.1 Hierarchy Data Model

SAP HANA hierarchy functions are designed to work with a wide range of source data that is not structured into nodes and edges.

The most common types are parent-child hierarchies and leveled hierarchies. In parent-child hierarchies, each record references a parent record, defining the hierarchical structure. Common examples are company organization structures or HR reporting lines, where each employee record references another person a as direct report or manager. In leveled hierarchies, each record contains complete individual path information through the hierarchy. A common example is address data, where each record typically consists of country, state, city, street, and street number data items, which may also be interpreted as a geographical hierarchy. The attributes encoding the hierarchical structure can be of any scalar data type such as VARCHAR, INTEGER, or VARBINARY. Composite keys can easily be made consumable by hierarchy generator functions by concatenating them with the full toolset of HANA SQL scalar operators or functions (including the HIERARCHY_COMPOSITE_ID Scalar Function [page 36].

Specialized hierarchy generator functions translate the diverse relational source data into a generic and normalized tabular format that, for the sake of brevity, is just termed **hierarchy**. The structure of such a hierarchy is always the same, regardless of the original format of the source data, and consists of an ordered list of its nodes. The nodes themselves are represented by a minimal set of orthogonal hierarchy topology attributes plus a projection of the original source attributes.

Many basic properties of the hierarchy and its elements can be determined by directly querying this tabular hierarchy representation using the usual SQL filtering and expression facilities, as shown in more detail in the section Deriving Further Hierarchy Attributes [page 72]. More complex calculations on the hierarchy involving a start set of nodes and potentially multiple steps beyond their immediate neighborhood are termed **navigations**. For the most common use cases, SAP HANA provides built-in navigation functions that use optimized algorithms on the hierarchy attributes to efficiently compute ancestors, descendants, or siblings of nodes. Since the hierarchy is fully exposed to consumers as conventional numeric attributes, it is easy to leverage this structure to write optimized custom navigations on the hierarchy.

The two-step model consisting of hierarchy creation and navigation provides additional benefits for the users. It not only introduces an abstraction from the specific layout of the source data, it also allows users to fine-tune the performance and update characteristics of their queries based on each use case. The spectrum ranges from combined creation and navigation in a single ad hoc query without caching to hierarchy indices stored in trigger-updated (or even manually updated) database tables. This topic is further discussed in the section Separating Hierarchy Model From Navigation [page 40]. This conceptual separation also allows other source structure types to be added in the future, whereas the navigation operations remain the same.

Hierarchy functions are optimized to process source data as pure tree topologies, where any node has at most one parent node. However, real world data rarely completely adheres to this ideal case. To account for that, the hierarchy generator functions can handle typical deviations such as multiple parents, cycles, orphans, or isolated islands. They provide several options how to deal with deviating data and retain information about the original topology in additional node attributes. So, the result of a hierarchy generator function can always be processed as consisting of one or more strictly well-formed trees. Custom navigation functions can capitalize on this guarantee as well by using optimized simple algorithms. Examples are given in the section Custom Navigations [page 62]. However, if a source data topology rather resembles a generic graph or mesh with no dominant vertical structure, take some precautions during hierarchy generation to avoid very large result sets; these measures are discussed in detail in the section Handling Generic Graph Topologies [page 81].

2.2 Hierarchy Terminology

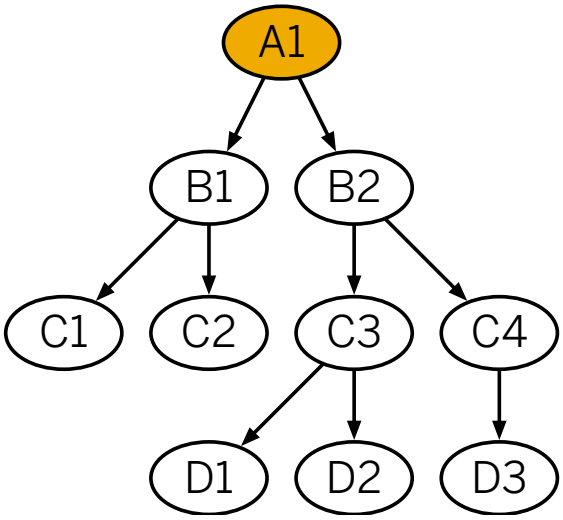
- **Hierarchy**
  A short term for any hierarchy generator function result (for example, using the HIERARCHY function). A generated hierarchy contains a set of standard attributes with fixed types. The topology of the generated hierarchy is strictly a tree or forest where each result row corresponds to a single graph node and edge. The computed index enables the navigation functions to traverse the hierarchy tree efficiently.
- **Node**
  The basic element of a hierarchy. Each node is defined as a row of a hierarchy generator function result. The primary identifier of a node is its preorder rank provided by the HIERARCHY_RANK attribute.
- **Edge**
  The basic relationship between two nodes in a hierarchy. In a hierarchy generator function result, all edges are directed. Each node has either 0 (root) or 1 incoming edge, represented by the value of the attribute pair HIERARCHY_PARENT_RANK/HIERARCHY_RANK. In addition, a node may have 0 (leaf) or more (branch) outgoing edges. Since a hierarchy is always a strict tree, edges are not independent entities like in generic graphs. All edge properties can be fully derived from the properties of their end-point node.
- **Parent**
  A node reached by an incoming edge is called a parent node of that node. A node has either one or no parent node. The preorder rank of the parent node is encoded in the HIERARCHY_PARENT_RANK attribute.
- **Child**
  A node reached by an outgoing edge is called a child node of that node. Children are not directly encoded in any attribute, but the interval given by [HIERARCHY_RANK, HIERARCHY_RANK + HIERARCHY_TREE_SIZE-1] includes all children and further descendants.
- **Root**
  Root nodes are all nodes that do not have a parent node. They all have a HIERARCHY_PARENT_RANK of 0, which is not a valid node rank.

- **Orphan**
  Orphaned nodes are all nodes of a hierarchy that are not reachable from a set of user-defined start nodes. In a recursive parent-child hierarchy, two sorts of orphans exist. On the one hand, there are records that do not match the START condition nor does their PARENT_ID value occur as a NODE_ID value. These nodes can be identified without computing the hierarchy result. On the other hand, there are orphaned islands made up of nodes, which build a cycle that is not connected to any root.

2.3 Topologies Used in Examples

For the sake of clarity, all examples in this guide are based on one set of trees defined in this section. In the figures, root nodes are denoted by ellipses with golden fill color, regular nodes are unfilled ellipses. Nodes that do not exist, but are referenced by other nodes, are shown as unfilled ellipses with a dashed outline.

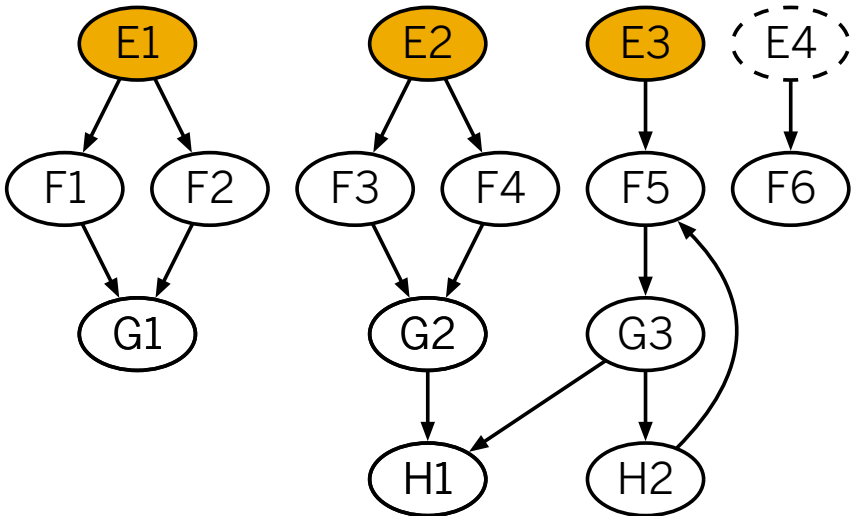
The table T_DEMO contains one well-formed tree:




```
CREATE COLUMN TABLE t_demo (
    parent_id VARCHAR(2), node_id VARCHAR(2), ord INTEGER, amount INTEGER );

INSERT INTO t_demo VALUES ( null, 'A1', 1, 1 );
INSERT INTO t_demo VALUES ( 'A1', 'B1', 1, 2 );
INSERT INTO t_demo VALUES ( 'A1', 'B2', 2, 4 );
INSERT INTO t_demo VALUES ( 'B1', 'C1', 1, 1 );
INSERT INTO t_demo VALUES ( 'B1', 'C2', 2, 3 );
INSERT INTO t_demo VALUES ( 'B2', 'C3', 3, 1 );
INSERT INTO t_demo VALUES ( 'B2', 'C4', 4, 2 );
INSERT INTO t_demo VALUES ( 'C3', 'D1', 1, 2 );
INSERT INTO t_demo VALUES ( 'C3', 'D2', 2, 3 );
INSERT INTO t_demo VALUES ( 'C4', 'D3', 3, 1 );
```

For other examples, trees with inconsistencies are required. The table T_DEMO_ERR contains some trees with topology errors:

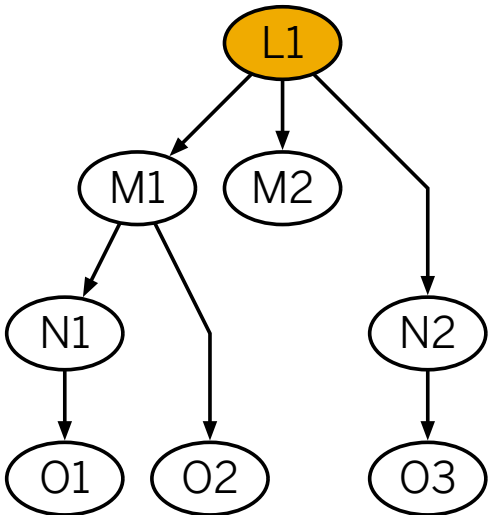


Tree 1, under node E1, contains a leaf node with multiple parents. Tree 2, under node E2, contains a regular node with multiple parents and tree 3, under node E3, contains a cycle and also a node from tree 2. There is also a node that does not belong to any tree. The parent of node F6, E4, does not exist, so F6 is an orphaned node.

```
CREATE COLUMN TABLE t_demo_err (
    parent_id VARCHAR(2), node_id VARCHAR(2), tree INTEGER );

INSERT INTO t_demo_err VALUES ( null, 'E1', 1 );
INSERT INTO t_demo_err VALUES ( 'E1', 'F1', 1 );
INSERT INTO t_demo_err VALUES ( 'E1', 'F2', 1 );
INSERT INTO t_demo_err VALUES ( 'F1', 'G1', 1 );
INSERT INTO t_demo_err VALUES ( 'F2', 'G1', 1 );
INSERT INTO t_demo_err VALUES ( null, 'E2', 2 );
INSERT INTO t_demo_err VALUES ( 'E2', 'F3', 2 );
INSERT INTO t_demo_err VALUES ( 'E2', 'F4', 2 );
INSERT INTO t_demo_err VALUES ( 'F3', 'G2', 2 );
INSERT INTO t_demo_err VALUES ( 'F4', 'G2', 2 );
INSERT INTO t_demo_err VALUES ( 'G2', 'H1', 2 );
INSERT INTO t_demo_err VALUES ( null, 'E3', 3 );
INSERT INTO t_demo_err VALUES ( 'E3', 'F5', 3 );
INSERT INTO t_demo_err VALUES ( 'F5', 'G3', 3 );
INSERT INTO t_demo_err VALUES ( 'G3', 'H1', 3 );
INSERT INTO t_demo_err VALUES ( 'G3', 'H2', 3 );
INSERT INTO t_demo_err VALUES ( 'H2', 'F5', 3 );
INSERT INTO t_demo_err VALUES ( 'E4', 'F6', null );
```

For a leveled hierarchy, we use the source data from table T_DEMO_LVL:

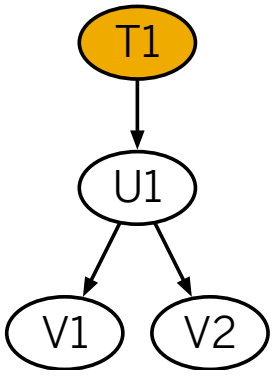


Since the leveled source format defines the level of each node, there can be gaps in the structure.

```
CREATE COLUMN TABLE t_demo_lvl (
    level_1 VARCHAR(2), level_2 NVARCHAR(2), attr_2 int,
    level_3 NVARCHAR(2), level_4 NVARCHAR(2), attr_4 int );

INSERT INTO t_demo_lvl VALUES ( 'L1', 'M1', 1, 'N1', 'O1', 10 );
INSERT INTO t_demo_lvl VALUES ( 'L1', 'M1', 2, NULL, 'O2', 20 );
INSERT INTO t_demo_lvl VALUES ( 'L1', 'M2', 3, NULL, NULL, NULL );
INSERT INTO t_demo_lvl VALUES ( 'L1', NULL, NULL, 'N2', 'O3', 30 );
```

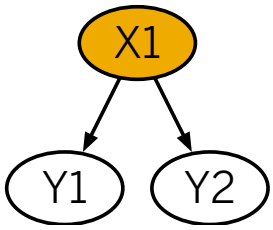
For time-dependent source data, we use the table T_DEMO_TIME:




```
CREATE TABLE t_demo_time (
    parent_id VARCHAR(32), node_id VARCHAR(32), valid_from DATE, valid_until
DATE );

INSERT INTO t_demo_time  VALUES ( NULL, 'T1', '2011-01-01', '2016-12-31' );
INSERT INTO t_demo_time  VALUES ( 'T1', 'U1', '2011-07-01', '2018-12-31' );
INSERT INTO t_demo_time  VALUES ( 'U1', 'V1', '2011-07-01', '2012-12-31' );
INSERT INTO t_demo_time  VALUES ( 'U1', 'V2', '2013-01-01', '2016-12-31' );
```

For source data with composite identifiers, we use the table T_DEMO_COMPOSITE:




```
CREATE COLUMN TABLE t_demo_composite (
    parent_1 VARCHAR(1), parent_2 INTEGER, id_1 VARCHAR(1),
    id_2 INTEGER, ord INTEGER );

INSERT INTO t_demo_composite VALUES ( NULL, NULL, 'X', 1, 1 );
INSERT INTO t_demo_composite VALUES ( 'X', 1, 'Y', 1, 1 );
INSERT INTO t_demo_composite VALUES ( 'X', 1, 'Y', 2, 2 );
```

3 Generating Hierarchies

Currently, SAP HANA offers four hierarchy generator functions.

Most prominently, the HIERARCHY Generator Function [page 11] covers the predominant case of generating a hierarchy from a recursive parent-child source data structure. The HIERARCHY_SPANTREE Generator Function [page 21] uses the same source data format as the HIERARCHY function, but generates only a minimal spanning tree, which is useful when the source data contains many multiple parent edges. The HIERARCHY_TEMPORAL Generator Function [page 26] operates on parent-child source data containing an additional validity interval. This generates a hierarchy for a specific time interval. The HIERARCHY_LEVELED Generator Function [page 32] generates an analogous output structure based on source data where individual source columns correspond to hierarchy levels and source rows define paths from a root to a leaf node. The HIERARCHY_COMPOSITE_ID Scalar Function [page 36] facilitates working with node identifiers consisting of multiple components.

3.1 HIERARCHY Generator Function

Generates a hierarchy based on recursive parent-child source data.

Syntax

```
HIERARCHY (
    <hierarchy_genfunc_source_spec>
    [<hierarchy_genfunc_start_cond>]
    <hierarchy_genfunc_order_spec>
    [<hierarchy_genfunc_depth_spec>]
    [<hierarchy_genfunc_multiparent_spec>]
    [<hierarchy_genfunc_orphan_spec>]
    [<hierarchy_genfunc_cycle_spec>]
    [<hierarchy_genfunc_cache_spec>]
    [<hierarchy_genfunc_load_spec>]
)
```

Syntax Elements

<hierarchy_genfunc_source_spec>

Specifies the source that the hierarchy is to be generated from.

```
<hierarchy_genfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <table_ref>
```

```
    | <subquery>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

Each row defines a parent-child relation and attributes of a node or edge that may become part of the output table, provided that they are reached during traversal. The source columns defining the recursive parent-child relation are identified by a naming convention or aliases NODE_ID (node) and PARENT_ID (parent). The data type of PARENT_ID and NODE_ID must be identical, and must belong to one of the following data types:

- Numeric types: TINYINT, SMALLINT, INTEGER, BIGINT, SMALLDECIMAL, DECIMAL
- Character string types: VARCHAR, NVARCHAR, ALPHANUM, SHORTTEXT
- Date/time types: DATE, TIME, SECONDDATE, TIMESTAMP
- Binary types: VARBINARY
- Spatial types: ST_GEOMETRY

<hierarchy_genfunc_start_cond>

Specifies the start condition to identify the root nodes.

```
<hierarchy_genfunc_start_cond> ::= START WHERE <condition>
```

See the *where_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The start condition is applied as an additional filter to the SOURCE specification. If a start condition is not specified, then root nodes are identified by the condition `WHERE parent_id IS NULL`.

> **i** Note
>
> The start condition must use the original unaliased source attribute names.

<hierarchy_genfunc_order_spec>

Defines the sort order of sibling nodes. This overrides any sort order the source might originally have.

```
<hierarchy_genfunc_order_spec> ::= SIBLING <order_by_clause>
```

See the *order_by_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

> **i** Note
>
> The order specification must use the source's result attribute names with aliases, if any. Also all attributes used in the order specification must be part of the source's projection list.

<hierarchy_genfunc_depth_spec>

Specifies the maximum search depth during hierarchy generation.

```
<hierarchy_genfunc_depth_spec> ::= DEPTH <integer>
```

A maximum traversal depth during hierarchy generation can improve the performance of ad hoc queries when the full depth of the hierarchy is not of interest. A predefined maximum depth can be particularly useful when

working with directed graph topologies. Start nodes and roots have a depth of 0. If the depth is set to less than 0, then the query returns an empty result set. The parameter is not required to avoid endless recursion due to cycles in the source data, because cycles are traversed at most once and are always broken up or terminated upon first re-entrance.

<hierarchy_genfunc_multiparent_spec>

Specifies the multiparent processing policy.

```
<hierarchy_genfunc_multiparent_spec> ::=
    MULTIPARENT
    | MULTIPARENT ERROR
    | MULTIPARENT LEAVES
```

Multiple parent nodes occur when two source rows have identical NODE_ID values, or when start nodes are contained in a cycle. This means that hierarchies with cycles always contain multiple parent nodes. Since duplicate rows are not automatically pruned, multiple parents already occur when the PARENT_ID value is also identical.

- MULTIPARENT
  Multiparent nodes are accepted and processed as any other node. This is the default behavior.
- MULTIPARENT ERROR
  If the hierarchy function result set contains non-distinct NODE_ID values, then an error is raised.
- MULTIPARENT LEAVES
  If the hierarchy function result set contains rows with non-distinct NODE_ID values whose HIERARCHY_TREE_SIZE is greater than 1, then an error is raised.

<hierarchy_genfunc_orphan_spec>

Specifies the orphan processing policy.

```
<hierarchy_genfunc_orphan_spec> ::=
    ORPHAN IGNORE
    | ORPHAN ERROR
    | ORPHAN ADOPT
    | ORPHAN ROOT
```

Orphans are source nodes that cannot be reached from any root node.
`<hierarchy_genfunc_orphan_spec>` and `<hierarchy_genfunc_depth_spec>` cannot be used at the same time. If a maximum search depth is specified, then orphaned nodes must be ignored.

- IGNORE
  Orphans are silently ignored. This is the default behavior.
- ERROR
  If the input data contains any orphans, an error is raised.
- ROOT
  Top-level orphans are treated as root nodes.
- ADOPT
  Top-level orphans are adopted as children of the last root node behind its regular descendants.

During orphan handling (that is, ORPHAN ROOT or ORPHAN ADOPT), only edges that have not been traversed when starting from a regular root node are considered. For these edges, the HIERARCHY_IS_ORPHAN value is set to 1. The original source attribute values are retained.

In the case of an orphaned cycle, an additional edge is introduced in order to define a root node or adopt the orphaned cycle by an existing root node. Since there is no corresponding input data for this edge, its

source attributes are set to NULL, except for NODE_ID, which reflects the value of the end point of the newly introduced edge.

<hierarchy_genfunc_cycle_spec>

Specifies the cycle processing policy.

```
<hierarchy_genfunc_cycle_spec> ::=
    CYCLE BREAKUP
    | CYCLE ERROR
```

- CYCLE BREAKUP
  Depth-first source traversal does not continue after cycle closure and setting the HIERARCHY_IS_CYCLE flag for the closing result row. This is the default behavior.
- CYCLE ERROR
  If the source data traversal passes a cycle (that means a node has the same node identifier as one of its ancestors), an error is raised.

<hierarchy_genfunc_cache_spec>

Specifies the caching policy for the generated hierarchy.

```
<hierarchy_genfunc_cache_spec> ::=
    CACHE
    | NO CACHE
    | CACHE FORCE
    | CACHE { 'on' | 'off' | 'force' }
```

- CACHE
  The generated hierarchy is cached if the system assesses the source to be reliably deterministic.
  This is the default behavior.
- NO CACHE
  The generated hierarchy is not cached.
- CACHE FORCE
  The generated hierarchy is cached even if the source cannot be assessed to be reliably deterministic.
- CACHE { 'on' | 'off' | 'force' }
  The generated hierarchy is cached according to the provided string literal value. All string literal values are processed in a case-insensitive manner.
  'on' is equivalent to CACHE
  'off' is equivalent to NO CACHE
  'force' is equivalent to CACHE FORCE

Caching may improve the performance for subsequent navigation of the same hierarchy.

<hierarchy_genfunc_load_spec>

Specifies the load policy for the generated hierarchy.

```
<hierarchy_genfunc_load_spec> ::=
    LOAD BULK
    | LOAD INCREMENTAL
    | LOAD { 'bulk' | 'incremental' }
```

- LOAD BULK
  The complete source table of the hierarchy is loaded. This is default behavior

- **LOAD INCREMENTAL**
  Only the rows of the source table that can be reached from the start nodes are loaded.
- **LOAD { 'bulk' | 'incremental' }**
  The hierarchy source data is loaded according to the provided string literal value. All string literal values are processed in a case-insensitive manner.
  'bulk' is equivalent to LOAD BULK
  'incremental' is equivalent to LOAD INCREMENTAL

Description

The HIERARCHY function calculates hierarchical attributes for each edge and node based on a tabular SOURCE containing an adjacency list (columns that form a recursive parent-child relation between rows of the source data). The source columns carrying the particular recursive semantics must be named or aliased as NODE_ID and PARENT_ID.

The result table returned by the HIERARCHY function provides the general hierarchical output attributes provided in the section Basic Attributes [page 38].

Examples

The examples are based on the data in tables T_DEMO and T_DEMO_ERR from the section Topologies Used in Examples [page 7].

Example

You calculate all hierarchy attributes of all edges and nodes starting from the default condition `WHERE parent_id IS NULL`:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    hierarchy_root_rank AS root_rank,
    hierarchy_is_cycle AS is_cycle,
    hierarchy_is_orphan AS is_orphan,
    parent_id,
    node_id
FROM HIERARCHY (
    SOURCE t_demo
    SIBLING ORDER BY ord )
ORDER BY hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	ROOT_RANK	IS_CYCLE	IS_ORPHAN	PARENT_ID	NODE_ID
1	10	0	1	1	0	0		A1
2	3	1	2	1	0	0	A1	B1

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	ROOT_RANK	IS_CYCLE	IS_OR-PHAN	PARENT_ID	NODE_ID
3	1	2	3	1	0	0	B1	C1
4	1	2	3	1	0	0	B1	C2
5	6	1	2	1	0	0	A1	B2
6	3	5	3	1	0	0	B2	C3
7	1	6	4	1	0	0	C3	D1
8	1	6	4	1	0	0	C3	D2
9	2	5	3	1	0	0	B2	C4
10	1	9	4	1	0	0	C4	D3

Example

You determine the set of descendants of node A1 with a depth horizon of 2 (that is, down to level C). Other nodes are automatically ignored:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    hierarchy_is_cycle AS is_cycle,
    hierarchy_is_orphan AS is_orphan,
    parent_id,
    node_id
FROM HIERARCHY (
    SOURCE t_demo
    SIBLING ORDER BY ord
    DEPTH 2 )
WHERE
    hierarchy_level > 1
ORDER BY
    node_id;
```

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	IS_CYCLE	IS_ORPHAN	PARENT_ID	NODE_ID
2	3	1	2	0	0	A1	B1
5	3	1	2	0	0	A1	B2
3	1	2	3	0	0	B1	C1
4	1	2	3	0	0	B1	C2
6	1	5	3	0	0	B2	C3
7	1	5	3	0	0	B2	C4

Example

You calculate all hierarchy attributes starting from B1 and B2, and top-level orphaned nodes become root nodes:

```
SELECT
```

```
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_root_rank AS root_rank,
    hierarchy_level AS level,
    hierarchy_is_cycle AS is_cycle,
    hierarchy_is_orphan AS is_orphan,
    parent_id,
    node_id
FROM HIERARCHY (
    SOURCE t_demo
    START WHERE node_id IN ('B1', 'B2')
    SIBLING ORDER BY ord
    ORPHAN ROOT )
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	ROOT_RANK	LEVEL	IS_CYCLE	IS_ORPHAN	PARENT_ID	NODE_ID
1	3	0	1	1	0	0	A1	B1
2	1	1	1	2	0	0	B1	C1
3	1	1	1	2	0	0	B1	C2
4	6	0	4	1	0	0	A1	B2
5	3	4	4	2	0	0	B2	C3
6	1	5	4	3	0	0	C3	D1
7	1	5	4	3	0	0	C3	D2
8	2	4	4	2	0	0	B2	C4
9	1	8	4	3	0	0	C4	D3
10	1	0	10	1	0	1		A1

Nodes that are orphaned, in this case node A1, retain their original PARENT_ID value (NULL in this case).

Example

You calculate all hierarchy attributes starting from nodes B1 and B2 nodes. Orphaned nodes are adopted by B2, which is the last root node in preorder traversal:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_root_rank AS root_rank,
    hierarchy_level AS level,
    hierarchy_is_cycle AS is_cycle,
    hierarchy_is_orphan AS is_orphan,
    parent_id,
    node_id
FROM HIERARCHY (
    SOURCE t_demo
    START WHERE node_id LIKE 'B%'
    SIBLING ORDER BY ord
    ORPHAN ADOPT )
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PA-RENT_RANK	ROOT_RANK	LEVEL	IS_CYCLE	IS_OR-PHAN	PARENT_ID	NODE_ID
1	3	0	1	1	0	0	A1	B1
2	1	1	1	2	0	0	B1	C1
3	1	1	1	2	0	0	B1	C2
4	7	0	4	1	0	0	A1	B2
5	3	4	4	2	0	0	B2	C3
6	1	5	4	3	0	0	C3	D1
7	1	5	4	3	0	0	C3	D2
8	2	4	4	2	0	0	B2	C4
9	1	8	4	3	0	0	C4	D3
10	1	4	4	2	0	1		A1

You can identify the parent that adopted the orphaned nodes by looking at the HIERARCHY_PARENT_RANK (or PARENT_RANK, in the results) value from the nodes at level 2, whose IS_ORPHAN flag is 1. For example, the orphaned node A1 shows that it has been adopted by HIERARCHY_PARENT_RANK 4, which corresponds to node B2.

Example

You determine the ancestors of node D1 up to the roots (note the reversal of the NODE_ID and PARENT_ID columns in the SOURCE specification):

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    hierarchy_is_cycle AS is_cycle,
    hierarchy_is_orphan AS is_orphan,
    node_id,
    parent_id
FROM HIERARCHY (
    SOURCE ( SELECT parent_id AS node_id, node_id AS parent_id, ord FROM t_demo )
    START WHERE node_id = 'D1'
    SIBLING ORDER BY ord
    ORPHAN IGNORE )
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	IS_CYCLE	IS_ORPHAN	NODE_ID	PARENT_ID
1	3	0	1	0	0	C3	D1
2	2	1	2	0	0	B2	C3
3	1	2	3	0	0	A1	B2

Example

You use the MULTIPARENT specification to enforce topology restrictions. Trees 1 and 2 from table T_DEMO_ERR contain nodes with multiple parents.

Multiple parents are not allowed:

```
SELECT
    hierarchy_rank, node_id
FROM HIERARCHY(
    SOURCE (SELECT * FROM t_demo_err WHERE tree IN (1,2))
    SIBLING ORDER BY node_id
    MULTIPARENT ERROR
)
ORDER BY
    hierarchy_rank;
```

```
Error 5088
Hierarchy error:
The hierarchy source data contains node G1 having multiple parents but the
hierarchy is specified to reject such nodes.: line 1 col 37 (at pos 36)
```

Multiple parents are allowed for leaf nodes, and only leaf nodes have multiple parents:

```
SELECT
    hierarchy_rank, node_id
FROM HIERARCHY(
    SOURCE (SELECT * FROM t_demo_err WHERE tree=1)
    SIBLING ORDER BY node_id
    MULTIPARENT LEAVES
)
ORDER BY
    hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
1	E1
2	F1
3	G1
4	F2
5	G1

The second tree in the source data has a regular node with multiple parents, but multiple parents are allowed for leaf nodes only:

```
SELECT
    hierarchy_rank, node_id
FROM HIERARCHY(
    SOURCE (SELECT * FROM t_demo_err WHERE tree=2)
    SIBLING ORDER BY node_id
    MULTIPARENT LEAVES
)
ORDER BY
    hierarchy_rank;
```

```
Error 5088
Hierarchy error:
The hierarchy source data contains branch node G2 having multiple parents but
the hierarchy is specified to reject such nodes.: line 1 col 37 (at pos 36)
```

All kinds of multiple parents are allowed:

```
SELECT
```

```
    hierarchy_rank, node_id
FROM HIERARCHY(
    SOURCE (SELECT * FROM t_demo_err WHERE tree IN (1,2))
    SIBLING ORDER BY node_id
    MULTIPARENT
)
ORDER BY
    hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
1	E1
2	F1
3	G1
4	F2
5	G1
6	E2
7	F3
8	G2
9	H1
10	F4
11	G2
12	H1

Example

You use the CYCLE specification to enforce topology restrictions. Tree 3 from table T_DEMO_ERR contains a cycle.

Cycles are not allowed:

```
SELECT
    hierarchy_rank, node_id
FROM HIERARCHY(
    SOURCE (SELECT * FROM t_demo_err WHERE tree=3)
    SIBLING ORDER BY node_id
    CYCLE ERROR
)
ORDER BY
    hierarchy_rank;
```

```
Error 5088
Hierarchy error:
The hierarchy source data contains an edge H2 -> F5 which closes a cycle but the
hierarchy is configured to reject cycles.: line 1 col 37 (at pos 36)
```

Cycles are allowed and are broken up:

```
SELECT
    hierarchy_rank, hierarchy_is_cycle, node_id
FROM HIERARCHY(
    SOURCE (SELECT * FROM t_demo_err WHERE tree=3)
    SIBLING ORDER BY node_id
    CYCLE BREAKUP
```

```
)
ORDER BY
    hierarchy_rank;
```

HIERARCHY_RANK	HIERARCHY_IS_CYCLE	NODE_ID
1	0	E3
2	0	F5
3	0	G3
4	0	H1
5	0	H2
6	1	F5

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

3.2 HIERARCHY_SPANTREE Generator Function

Generates a hierarchy for a recursive parent-child source containing only the first shortest path between each start and result node.

Syntax

```
HIERARCHY_SPANTREE (
    <hierarchy_genfunc_source_spec>
    [<hierarchy_genfunc_start_cond>]
    <hierarchy_genfunc_order_spec>
    [<hierarchy_genfunc_depth_spec>]
    [<hierarchy_genfunc_cache_spec>]
    [<hierarchy_genfunc_load_spec>])
```

Syntax Elements

The syntax elements correspond to the HIERARCHY generator function. The only difference is that HIERARCHY_SPANTREE has no multiparent, orphan, and cycle specifications. Orphans are always ignored and multiple parents and cycles cannot occur due to the shortest path policy.

<hierarchy_genfunc_source_spec>

Specifies the source that the hierarchy is to be generated from.

```
<hierarchy_genfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <table_ref>
    | <subquery>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

Each row defines a parent-child relation and attributes of a node or edge that may become part of the output table if they are reached during traversal. The source columns defining the recursive parent-child relation are identified by a naming convention or aliases NODE_ID (node) and PARENT_ID (parent). The data type of PARENT_ID and NODE_ID must be identical, and must belong to one of the following data types:

- Numeric types: TINYINT, SMALLINT, INTEGER, BIGINT, SMALLDECIMAL, DECIMAL
- Character string types: VARCHAR, NVARCHAR, ALPHANUM, SHORTTEXT
- Date/time types: DATE, TIME, SECONDDATE, TIMESTAMP
- Binary types: VARBINARY
- Spatial types: ST_GEOMETRY

<hierarchy_genfunc_start_cond>

Specifies the start condition to identify the root nodes.

```
<hierarchy_genfunc_start_cond> ::= START WHERE <condition>
```

See the *where_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The start condition is applied as an additional filter to the SOURCE specification. If a start condition is not specified, then root nodes are identified by the condition `WHERE parent_id IS NULL`.

> **i** Note
>
> The start condition must use the original unaliased source attribute names.

<hierarchy_genfunc_order_spec>

Defines the sort order of sibling nodes. This overrides any sort order the source might originally have.

```
<hierarchy_genfunc_order_spec> ::= SIBLING <order_by_clause>
```

See the *order_by_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

> **i** Note
>
> The order specification must use the source's result attribute names with aliases, if any. Also all attributes used in the order specification must be part of the source's projection list.

<hierarchy_genfunc_depth_spec>

Specifies the maximum search depth during hierarchy generation.

```
<hierarchy_genfunc_depth_spec> ::= DEPTH <integer>
```

A maximum traversal depth during hierarchy generation can improve the performance of ad hoc queries, when the full depth of the hierarchy is not of interest. A predefined maximum depth can be particularly useful when working with directed graph topologies. Start nodes and roots have a depth of 0. If the depth is set to less than 0, then the query returns an empty result set. The parameter is not required to avoid endless recursion due to cycles in the source data, because cycles are traversed at most once and are always broken up or terminated upon first re-entrance.

<hierarchy_genfunc_cache_spec>

Specifies the caching policy for the generated hierarchy.

```
<hierarchy_genfunc_cache_spec> ::=
    CACHE
    | NO CACHE
    | CACHE FORCE
    | CACHE { 'on' | 'off' | 'force' }
```

- CACHE
  The generated hierarchy is cached if the system assesses the source to be reliably deterministic.
  This is the default behavior.
- NO CACHE
  The generated hierarchy is not cached.
- CACHE FORCE
  The generated hierarchy is cached even if the source cannot be assessed to be reliably deterministic.
- CACHE { 'on' | 'off' | 'force' }
  The generated hierarchy is cached according to the provided string literal value. All string literal values are processed in a case-insensitive manner.
  'on' is equivalent to CACHE
  'off' is equivalent to NO CACHE
  'force' is equivalent to CACHE FORCE

Caching may improve the performance for subsequent navigation of the same hierarchy.

<hierarchy_genfunc_load_spec>

Specifies the load policy for the generated hierarchy.

```
<hierarchy_genfunc_load_spec> ::=
    LOAD BULK
    | LOAD INCREMENTAL
    | LOAD { 'bulk' | 'incremental' }
```

- LOAD BULK
  The complete source table of the hierarchy is loaded. This is default behavior.
- LOAD INCREMENTAL
  Only the rows of the source table that can be reached from the start node are loaded.
- LOAD { 'bulk' | 'incremental' }
  The hierarchy source data is loaded according to the provided string literal value. All string literal values are processed in a case-insensitive manner.

'bulk' is equivalent to LOAD BULK
'incremental' is equivalent to LOAD INCREMENTAL

Description

The HIERARCHY_SPANTREE function calculates hierarchical attributes for each edge and node based on a tabular SOURCE containing an adjacency list (columns that form a recursive parent-child relation between rows of the source data). The source columns carrying the particular recursive semantics must be named or aliased as NODE_ID and PARENT_ID.

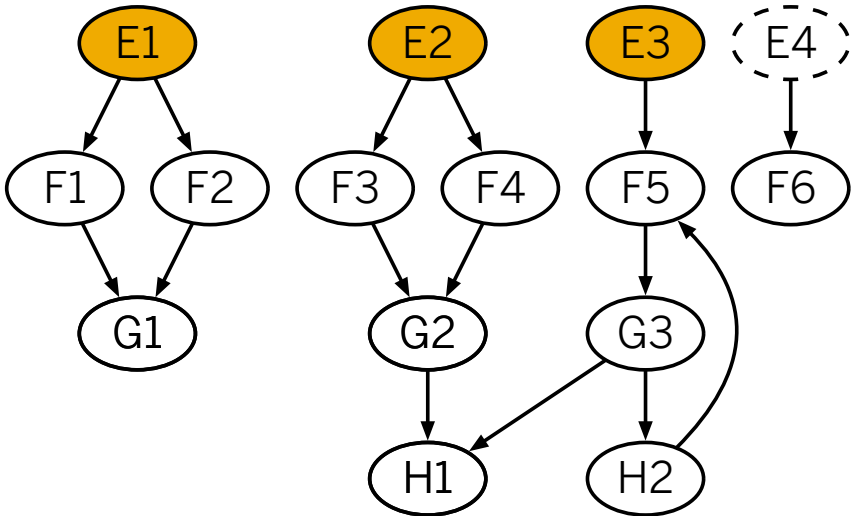
The result table returned by the HIERARCHY_SPANTREE function provides the general hierarchical output attributes provided in the section Basic Attributes [page 38].

If the recursive relations contained in the source data describe a well-formed tree, the output of HIERARCHY_SPANTREE is the same as the output of the HIERARCHY Generator Function [page 11], but at a higher computational cost.

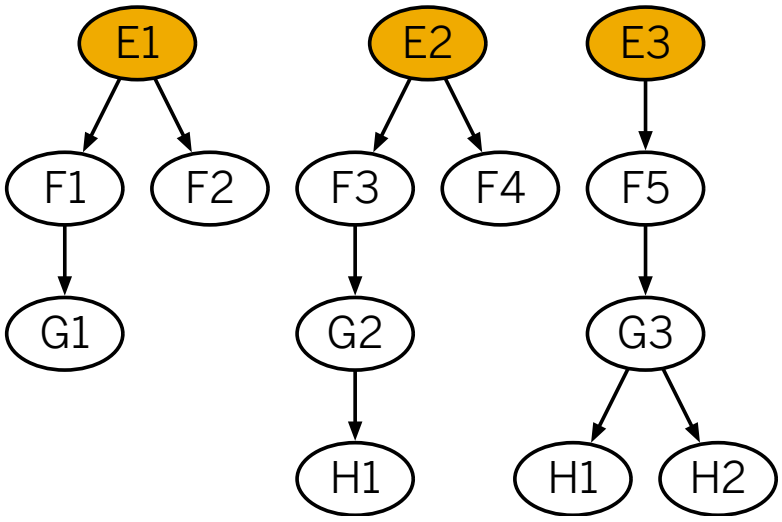
If the recursive relations describe a graph that is not a well-formed tree, a spanning tree/forest is computed, starting from the root nodes and traversing the source graph in breadth-first order. If multiple paths from the roots to a given node exist in the source graph, only the path with the shortest length and which appears first is selected for the spanning tree. Other paths are ignored.

For all start nodes, the spanning tree result represents the same connectivity information as the source graph. If nodes are directly or indirectly connected to a start node, they are also connected in the resulting spanning tree. This proposition holds true for start nodes, but not necessarily for originally connected branch or leaf nodes. In contrast, the more generic HIERARCHY Generator Function [page 11] correctly renders the connectivity of all result nodes, at the price of duplicating branches that are reachable by more than one path. This can easily lead to an exponentially growing number of hierarchy result nodes, whereas HIERARCHY_SPANTREE yields at most a result size of the source size multiplied by the number of start nodes.

The example hierarchy from table T_DEMO_ERR:



Looks like this when turned into a spanning tree:



Node H1 appears twice in the resulting spanning tree, once under the root E2 and once under the root E3.

The primary use case for HIERARCHY_SPANTREE is connectivity testing. In authorization checks, for example, connectivity between users and objects represents access rights, and nested roles and their relation of inclusion form a graph-like, intermediate structure between the users and objects. For an actual authorization check query, it is not relevant which and how many paths lead from a user to an object. It is sufficient to test whether or not there is a connection at all. This can be done much more efficiently with a spanning tree than with a fully expanded tree.

Examples

The examples are based on the data in table T_DEMO_ERR from the section Topologies Used in Examples [page 7].

Example

You create a minimal spanning tree of the source data:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_root_rank AS root_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    tree
FROM HIERARCHY_SPANTREE (
    SOURCE t_demo_err
    SIBLING ORDER BY node_id
)
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PA-RENT_RANK	ROOT_RANK	LEVEL	PARENT_ID	NODE_ID	TREE
1	4	0	1	1		E1	1
2	2	1	1	2	E1	F1	1
3	1	2	1	3	F1	G1	1
4	1	1	1	2	E1	F2	1
5	5	0	5	1		E2	2
6	3	5	5	2	E2	F3	2
7	2	6	5	3	F3	G2	2
8	1	7	5	4	G2	H1	2
9	1	5	5	2	E2	F4	2
10	5	0	10	1		E3	3
11	4	10	10	2	E3	F5	3
12	3	11	10	3	F5	G3	3
13	1	12	10	4	G3	H1	3
14	1	12	10	4	G3	H2	3

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

3.3 HIERARCHY_TEMPORAL Generator Function

Generates a time-dependent hierarchy for recursive parent-child source data whose edges are additionally qualified by validity intervals.

Syntax

```
HIERARCHY_TEMPORAL (
    <hierarchy_genfunc_source_spec>
    [ <hierarchy_genfunc_start_cond> ]
    <hierarchy_genfunc_order_spec>
    <hierarchy_genfunc_validity_spec>
    [ <hierarchy_genfunc_depth_spec> ]
    [<hierarchy_genfunc_multiparent_spec>]
    [<hierarchy_genfunc_cycle_spec>]
```

```
    [ <hierarchy_genfunc_cache_spec> ]
    [ <hierarchy_genfunc_load_spec> ]
)
```

Syntax Elements

The syntax elements widely correspond to the HIERARCHY generator function. The only differences are that HIERARCHY_TEMPORAL has a validity specification but no orphan specification. Orphans are always ignored.

<hierarchy_genfunc_source_spec>

Specifies the source that the hierarchy is to be generated from.

```
<hierarchy_genfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <table_ref>
    | <subquery>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

Each row defines a parent-child relation and attributes of a node or edge that may become part of the output table, provided that they are reached during traversal. The source columns defining the recursive parent-child relation are identified by a naming convention or aliases NODE_ID (node) and PARENT_ID (parent). The data type of PARENT_ID and NODE_ID must be identical, and must belong to one of the following data types:

- Numeric types: TINYINT, SMALLINT, INTEGER, BIGINT, SMALLDECIMAL, DECIMAL
- Character string types: VARCHAR, NVARCHAR, ALPHANUM, SHORTTEXT
- Date/time types: DATE, TIME, SECONDDATE, TIMESTAMP
- Binary types: VARBINARY
- Spatial types: ST_GEOMETRY

The source columns defining the validity interval are identified by naming conventions VALID_FROM and VALID_UNTIL. They must have the same type and must support the SQL BETWEEN expression.

<hierarchy_genfunc_start_cond>

Specifies the start condition to identify the root nodes.

```
<hierarchy_genfunc_start_cond> ::= START WHERE <condition>
```

See the *where_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The start condition is applied as an additional filter to the SOURCE specification. If a start condition is not specified, then root nodes are identified by the condition `WHERE parent_id IS NULL`.

> **Note**
>
> The start condition must use the original unaliased source attribute names.

<hierarchy_genfunc_order_spec>

Defines the sort order of sibling nodes. This overrides any sort order the source might originally have.

```
<hierarchy_genfunc_order_spec> ::= SIBLING <order_by_clause>
```

See the *order_by_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

> **i** Note
>
> The order specification must use the source's result attribute names with aliases, if any. Also all attributes used in the order specification must be part of the source's projection list.

<hierarchy_genfunc_validity_spec>

Specifies the query time of the current function call.

```
<hierarchy_genfunc_validity_spec> ::= VALID {
    FROM <expression> [ EXCLUDED ] UNTIL <expression> [ EXCLUDED ]
    | AT <expression> [ FROM EXCLUDED ] [ UNTIL EXCLUDED ] }
```

The first variant specifies the lower and upper bound of the query validity interval, the second one specifies a single point-in-time value. The data type of the inner expressions must be castable to the type of the VALID_FROM/VALID_UNTIL source columns.

The interval bounds are closed by default, which means that both boundary values are included. Optional EXCLUDED directives allow the specification of open interval bounds. For the interval-based validity specification variant, the EXCLUDED directives affect both the interpretation of source and query interval. For the single point-in-time variant, the EXCLUDED directives relate to the source only.

<hierarchy_genfunc_depth_spec>

Specifies the maximum search depth during hierarchy generation.

```
<hierarchy_genfunc_depth_spec> ::= DEPTH <integer>
```

A maximum traversal depth during hierarchy generation can improve the performance of ad hoc queries when the full depth of the hierarchy is not of interest. A predefined maximum depth can be particularly useful when working with directed graph topologies. Start nodes and roots have a depth of 0. If the depth is set to less than 0, then the query returns an empty result set. The parameter is not required to avoid endless recursion due to cycles in the source data, because cycles are traversed at most once and are always broken up or terminated upon first reentrance.

<hierarchy_genfunc_multiparent_spec>

Specifies the multiparent processing policy.

```
<hierarchy_genfunc_multiparent_spec> ::=
    MULTIPARENT
    | MULTIPARENT ERROR
    | MULTIPARENT LEAVES
```

Multiple parent nodes occur when two source rows have identical NODE_ID values, or when start nodes are contained in a cycle. This means that hierarchies with cycles always contain multiple parent nodes. Since duplicate rows are not automatically pruned, multiple parents already occur when the PARENT_ID value is also identical.

- MULTIPARENT
  Multiparent nodes are accepted and processed as any other node. This is the default behavior.
- MULTIPARENT ERROR
  If the hierarchy function result set contains non-distinct NODE_ID values, an error is raised.
- MULTIPARENT LEAVES
  If the hierarchy function result set contains rows with non-distinct NODE_ID values whose HIERARCHY_TREE_SIZE is greater than 1, an error is raised.

<hierarchy_genfunc_cycle_spec>

Specifies the cycle processing policy.

```
<hierarchy_genfunc_cycle_spec> ::=
    CYCLE BREAKUP
    | CYCLE ERROR
```

- CYCLE BREAKUP
  Depth-first source traversal does not continue after cycle closure and setting the HIERARCHY_IS_CYCLE flag for the closing result row. This is the default behavior.
- CYCLE ERROR
  If the source data traversal passes a cycle (that means a node has the same node identifier as one of its ancestors), an error is raised.

<hierarchy_genfunc_cache_spec>

Specifies the caching policy for the generated hierarchy.

```
<hierarchy_genfunc_cache_spec> ::=
    CACHE
    | NO CACHE
    | CACHE FORCE
    | CACHE { 'on' | 'off' | 'force' }
```

- CACHE
  The generated hierarchy is cached if the system assesses the source to be reliably deterministic. This is the default behavior.
- NO CACHE
  The generated hierarchy is not cached.
- CACHE FORCE
  The generated hierarchy is cached even if the source cannot be assessed to be reliably deterministic.
- CACHE { 'on' | 'off' | 'force' }
  The generated hierarchy is cached according to the provided string literal value. All string literal values are processed in a case-insensitive manner.
  'on' is equivalent to CACHE
  'off' is equivalent to NO CACHE
  'force' is equivalent to CACHE FORCE

Caching may improve the performance for subsequent navigation of the same hierarchy.

<hierarchy_genfunc_load_spec>

Specifies the load policy for the generated hierarchy.

```
<hierarchy_genfunc_load_spec> ::=
    LOAD BULK
```

```
    | LOAD INCREMENTAL
    | LOAD { 'bulk' | 'incremental' }
```

- **LOAD BULK**
  The complete source table of the hierarchy is loaded. This is the default behavior.
- **LOAD INCREMENTAL**
  Only the rows of the source table that can be reached from the start node are loaded.
- **LOAD { 'bulk' | 'incremental' }**
  The hierarchy source data is loaded according to the provided string literal value. All string literal values are processed in a case-insensitive manner.
  'bulk' is equivalent to LOAD BULK
  'incremental' is equivalent to LOAD INCREMENTAL

Description

The HIERARCHY_TEMPORAL function calculates hierarchical attributes for each edge and node based on the rows of a tabular SOURCE containing an adjacency list (columns that form a recursive parent-child relation between rows of the source data) and a validity interval. The source columns carrying the particular recursive semantics must be named or aliased as NODE_ID and PARENT_ID. The source columns carrying the validity interval information need to be named or aliased as VALID_FROM (lower boundary) and VALID_UNTIL (upper boundary).

The result table returned by the HIERARCHY function provides the general hierarchical output attributes provided in the section Basic Attributes [page 38].

Examples

The examples are based on the data in table T_DEMO_TIME in section Topologies Used in Examples [page 7].

Example

You build a hierarchy for the time interval from 2011-04-01 until 2013-12-31:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    valid_from,
    valid_until
FROM HIERARCHY_TEMPORAL (
    SOURCE t_demo_time
    SIBLING ORDER BY node_id, valid_from
    VALID FROM '2011-04-01' UNTIL '2013-12-31' EXCLUDED );
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	VALID_FROM	VALID_UNTIL
1	4	0	1		T1	2011-04-01	2013-12-31
2	3	1	2	T1	U1	2011-07-01	2013-12-31
3	1	2	3	U1	V1	2011-07-01	2012-12-31
4	1	2	3	U1	V1	2013-01-01	2013-12-31

Example

You build a hierarchy for the time interval from 2015-07-01 until 2019-12-31:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    valid_from,
    valid_until
FROM HIERARCHY_TEMPORAL (
    SOURCE t_demo_time
    SIBLING ORDER BY node_id, valid_from
    VALID FROM '2015-07-01' UNTIL '2019-12-31' );
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	VALID_FROM	VALID_UNTIL
1	3	0	1		T1	2015-07-01	2016-12-31
2	2	1	2	T1	U1	2015-07-01	2016-12-31
3	1	2	3	U1	V1	2015-07-01	2016-12-31

Example

You build a hierarchy for the single point-in-time at 2015-07-02:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    valid_from,
    valid_until
FROM HIERARCHY_TEMPORAL (
    SOURCE t_demo_time
    SIBLING ORDER BY node_id, valid_from
    VALID AT '2015-07-02' UNTIL EXCLUDED );
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	VALID_FROM	VALID_UNTIL
1	3	0	1		T1	2015-07-02	2015-07-02
2	2	1	2	T1	U1	2015-07-02	2015-07-02

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	PARENT_ID	NODE_ID	VALID_FRO M	VALID_UN-TIL
3	1	2	3	U1	V1	2015-07-02	2015-07-02

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

3.4 HIERARCHY_LEVELED Generator Function

Creates a hierarchy based on source data that has a leveled format.

Syntax

```
HIERARCHY_LEVELED (
    <hierarchy_genfunc_source_spec>
    [<hierarchy_genfunc_level_spec>]
    <hierarchy_genfunc_order_spec>
    [<hierarchy_genfunc_cache_spec>]
)
```

Syntax Elements

<hierarchy_genfunc_source_spec>

Specifies the source table for the hierarchy to be generated from.

```
<hierarchy_genfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <table_ref>
    | <subquery>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The source has the following characteristics:

- Each source column defines one level or level attribute of the generated hierarchy.

- Each row defines one path from a root node to a leaf node.
- The sequence of source columns determines their level.

The data types of the source columns must belong to one of the following data types:

- Numeric types: TINYINT, SMALLINT, INTEGER, BIGINT, SMALLDECIMAL, DECIMAL
- Character string types: VARCHAR, NVARCHAR, ALPHANUM, SHORTTEXT
- Date/time types: DATE, TIME, SECONDDATE, TIMESTAMP
- Binary types: VARBINARY
- Spatial types: ST_GEOMETRY

Level columns of type VARBINARY and ST_GEOMETRY are valid only if the types of all level columns are the same.

<hierarchy_genfunc_level_spec>

Specifies which source columns define a level.

```
<hierarchy_genfunc_level_spec> ::= LEVELS ( <column_name_list> )
<column_name_list> ::= <column_name> [ { , <column_name> } ... ]
<column_name> ::= <identifier>
```

If `<hierarchy_genfunc_level_spec>` is not specified, then all source columns are treated as defining a new level.

If `<column_name>` is not a member of the projection list of `<hierarchy_genfunc_source_spec>` , then an error is raised.

<hierarchy_genfunc_order_spec>

Defines the sort order of sibling nodes. This overrides any sort order the source might originally have.

```
<hierarchy_genfunc_order_spec> ::= SIBLING <order_by_clause>
```

See the *order_by_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

> **i** Note
>
> The order specification must use the source's result attribute names with aliases, if any. Also all attributes used in the order specification must be part of the source's projection list.

<hierarchy_genfunc_cache_spec>

Specifies the caching policy for the generated hierarchy.

```
<hierarchy_genfunc_cache_spec> ::=
    CACHE
    | NO CACHE
    | CACHE FORCE
    | CACHE { 'on' | 'off' | 'force' }
```

- **CACHE**
  The generated hierarchy is cached if the system assesses the source to be reliably deterministic. This is the default behavior.
- **NO CACHE**

The generated hierarchy is not cached.

- **CACHE FORCE**
  The generated hierarchy is cached even if the source cannot be assessed to be reliably deterministic.
- **CACHE { 'on' | 'off' | 'force' }**
  The generated hierarchy is cached according to the provided string literal value. All string literal values are processed in a case-insensitive manner.
  'on' is equivalent to CACHE
  'off' is equivalent to NO CACHE
  'force' is equivalent to CACHE FORCE

Caching may improve the performance for subsequent navigation of the same hierarchy.

Description

HIERARCHY_LEVELED creates a hierarchy based on source data having a leveled format. The optional LEVELS specification adds the possibility of distinguishing between source attributes that define a new level, and additional custom level attributes. The source table may contain NULL values. Only non-NULL values in source level columns generate result rows.

In addition to the general hierarchical output attributes listed in the section Basic Attributes [page 38], the HIERARCHY_LEVELED function generates the additional attributes listed in the section Attributes of Leveled Hierarchies [page 39].

Examples

The examples are based on the data in table T_DEMO_LVL from the section Topologies Used in Examples [page 7].

Example

The following example query calls the HIERARCHY_LEVELED function on data selected from the table T_DEMO_LVL. Since each column from the source table is treated as one level, also the columns ATTR_2 and ATTR_4 are included in the hierarchy, although they are not meant to be part of the hierarchy:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_level_name AS level_name,
    level_1 AS l1,
    level_2 AS l2,
    attr_2 AS a2,
    level_3 AS l3,
    level_4 AS l4,
    attr_4 AS a4
FROM HIERARCHY_LEVELED(
    SOURCE t_demo_lvl
    SIBLING ORDER BY level_1 NULLS LAST, level_2 NULLS LAST, level_3 NULLS LAST,
level_4 NULLS LAST )
```

```
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	LEVEL_NAME	L1	L2	A2	L3	L4	A4
1	14	0	1		L1	LEVEL_1	L1					
2	8	1	2	L1	M1	LEVEL_2	L1	M1				
3	4	2	3	M1	1	ATTR_2	L1	M1	1			
4	3	3	4	1	N1	LEVEL_3	L1	M1	1	N1		
5	2	4	5	N1	O1	LEVEL_4	L1	M1	1	N1	O1	
6	1	5	6	O1	10	ATTR_4	L1	M1	1	N1	O1	10
7	3	2	3	M1	2	ATTR_2	L1	M1	2			
8	2	7	5	2	O2	LEVEL_4	L1	M1	2		O2	
9	1	8	6	O2	20	ATTR_4	L1	M1	2		O2	20
10	2	1	2	L1	M2	LEVEL_2	L1	M2				
11	1	10	3	M2	3	ATTR_2	L1	M2	3			
12	3	1	4	L1	N2	LEVEL_3	L1			N2		
13	2	12	5	N2	O3	LEVEL_4	L1			N2	O3	
14	1	13	6	O3	30	ATTR_4	L1			N2	O3	30

Example

The following example shows the use of the LEVELS clause to distinguish between level-defining attributes (LEVEL_1, LEVEL_2, LEVEL_3, LEVEL_4) and additional purely descriptive custom level attributes (ATTR_2, ATTR_4):

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_level_name AS level_name,
    level_1 AS l1,
    level_2 AS l2,
    attr_2 AS a2,
    level_3 AS l3,
    level_4 AS l4,
    attr_4 AS a4
FROM HIERARCHY_LEVELED(
    SOURCE t_demo_lvl
```

```
    LEVELS ( level_1, level_2, level_3, level_4 )
    SIBLING ORDER BY level_1 NULLS LAST, level_2 NULLS LAST, level_3 NULLS LAST,
level_4 NULLS LAST )
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	LEVEL_NAME	L1	L2	A2	L3	L4	A4
1	8	0	1		L1	LEVEL_1	L1					
2	4	1	2	L1	M1	LEVEL_2	L1	M1	1			
3	2	2	3	M1	N1	LEVEL_3	L1	M1	1	N1		
4	1	3	4	N1	O1	LEVEL_4	L1	M1	1	N1	O1	10
5	1	2	4	M1	O2	LEVEL_4	L1	M1	2		O2	20
6	1	1	2	L1	M2	LEVEL_2	L1	M2	3			
7	2	1	3	L1	N2	LEVEL_3	L1			N2		
8	1	7	4	N2	O3	LEVEL_4	L1			N2	O3	30

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

3.5 HIERARCHY_COMPOSITE_ID Scalar Function

Concatenates multicolumn tuple-like node identifiers into single scalar values.

Syntax

```
HIERARCHY_COMPOSITE_ID ( <expression> [ { , <expression> } ... ] )
```

HIERARCHY_COMPOSITE_ID accepts one or more scalar expression parameters of any type that is castable to NVARCHAR.

Syntax Elements

<expression>

Specifies a scalar value that can be cast to NVARCHAR.

Description

Use the HIERARCHY_COMPOSITE_ID function to combine scalar expressions into a single value to use as a unique and stable node identifier value by hierarchy generator functions. The resulting values have additional characters encoding the lengths of the components to ensure uniqueness within the results.

Examples

```
SELECT HIERARCHY_COMPOSITE_ID('HUGO','EGON') AS node_id FROM dummy;
```

NODE_ID
4,HUGO;4,EGON

```
SELECT HIERARCHY_COMPOSITE_ID(123, NULL) AS node_id FROM dummy;
```

NODE_ID
3,123;

For more realistic examples, see Working With Composite Node Identifiers [page 86].

4 Hierarchy Attributes

4.1 Basic Attributes

The following default attributes are calculated by all hierarchy generator functions.

As a whole, they form the basis of an efficient hierarchy index structure capitalized by navigation functions:

Column name	Data type	Value range	Description
HIERARCHY_RANK	BIGINT NOT NULL	[1,N]	The preorder rank of the node in the tree representation of the result set. It also serves as the primary key of the hierarchy node.
HIERARCHY_TREE_SIZE	BIGINT NOT NULL	[1,N]	The number of descendant nodes +1.
HIERARCHY_PARENT_RANK	BIGINT NOT NULL	[0,N-1]	The preorder rank of the parent node, the value for root nodes is 0.
HIERARCHY_ROOT_RANK	BIGINT NOT NULL	[1,N]	The rank of the root node of the tree in which the node is located.
HIERARCHY_LEVEL	INTEGER NOT NULL	[1,N]	The level to which the node belongs. The value depends on the generation function and is strictly larger than the level of the parent node.
HIERARCHY_IS_CYCLE	TINYINT NOT NULL	[0,1]	A value indicating whether the current edge has closed a cycle (1) or not (0). Closed cycles are not further recursed.
HIERARCHY_IS_ORPHAN	TINYINT NOT NULL	[0,1]	A value indicating whether the current edge has been traversed during orphan handling (1) or not (0).
...	...	...	All columns of the source specification.

> **i Note**
>
> N denotes the tree cardinality, that is, the number of output nodes after treeification.

For most generation functions, the value of the HIERARCHY_LEVEL attribute is equal to the distance from the root node +1. The only exception is the HIERARCHY_LEVELED generation function, where it's derived directly

from the source data. In this case, it is possible that root nodes are not on level 1 and direct children may be more than one level below their parents.

In addition, the hierarchy generator functions also project all columns from their source object to the output, but any NOT NULL constraints are removed. Due to the naming conventions of the HIERARCHY function's SOURCE specification, the existence of the projected source attributes NODE_ID and PARENT_ID can be taken for granted as well. In the case of the HIERARCHY_TEMPORAL generation function, this also applies to the VALID_FROM and VALID_UNTIL columns.

4.2 Attributes of Leveled Hierarchies

The HIERARCHY_LEVELED function also generates the following columns:

Column name	Data type	Description
HIERARCHY_LEVEL_NAME	NVARCHAR(256) NOT NULL	The name of the level the hierarchy node belongs to.
NODE_ID	inferred from source level columns NOT NULL	The identifier of the current node.
PARENT_ID	inferred from source level columns	The identifier of the parent node.

The value of the HIERARCHY_LEVEL_NAME column is the name of the column corresponding to the highest level where the value is not NULL. The HIERARCHY_LEVEL_NAME column may be useful as part of the unique external key of a hierarchy node.

The value of the NODE_ID column is taken from the level column corresponding to the highest level where the value is not NULL. This value is not necessarily unique in the hierarchy. The value of the PARENT_ID column is the NODE_ID value of the parent node. The PARENT_ID value of root nodes is NULL.

The NODE_ID and PARENT_ID columns allow the generic use of the hierarchy result without knowledge of the generator function. The exact data type of the NODE_ID and PARENT_ID columns depends on the source level columns. If there is at least one VARCHAR/NVARCHAR source level column, the resulting data type is VARCHAR(5000)/NVARCHAR(5000). The base type of the two columns is guaranteed to be the same, except for an additional NOT NULL constraint on the NODE_ID column.

If a custom level attribute column corresponds to a level higher than the level of the current node, its value is set to NULL.

5 Navigating Hierarchies

This section briefly shows how typical hierarchy navigations can be done using SAP HANA hierarchy functions. Often, two alternative approaches are shown; one based on built-in navigation functions and the other directly on the hierarchy generator function result. If both options are available, the built-in navigation function-based approach is usually recommended due to its greater clarity and optimization potential. The purpose of the alternatives without navigation functions is to convey a basic understanding of how to use arithmetic on hierarchy attributes to write custom navigations.

The table H_DEMO_FACTS is used in the examples for HIERARCHY_DESCENDANTS_AGGREGATE Navigation Function [page 54]. It can be joined to a hierarchy generated from the table T_DEMO from the section Topologies Used in Examples [page 7].

```
CREATE COLUMN TABLE h_demo_facts (
    node VARCHAR(2), amount_dec_fact DECIMAL(10,2) );
INSERT INTO h_demo_facts VALUES ( 'A1', 5.3 );
INSERT INTO h_demo_facts VALUES ( 'B1', 1.9 );
INSERT INTO h_demo_facts VALUES ( 'B2', 1.8 );
INSERT INTO h_demo_facts VALUES ( 'C1', 6.3 );
INSERT INTO h_demo_facts VALUES ( 'C3', 4.4 );
INSERT INTO h_demo_facts VALUES ( 'C4', 8.2 );
INSERT INTO h_demo_facts VALUES ( 'D1', 0.5 );
INSERT INTO h_demo_facts VALUES ( 'D2', 2.7 );
INSERT INTO h_demo_facts VALUES ( 'D3', 3.9 );
INSERT INTO h_demo_facts VALUES ( 'D3', 5.1 );
INSERT INTO h_demo_facts VALUES ( 'X1', 8.0 );
INSERT INTO h_demo_facts VALUES ( 'X2', 9.9 );
INSERT INTO h_demo_facts VALUES ( null, 7.6 );
```

5.1 Separating Hierarchy Model From Navigation

Often multiple queries will be executed using the same hierarchy generator function result set. Therefore, it is recommended to either materialize the hierarchy generator function output into a temporary table or define a view over it. Materializing the hierarchy generator function output into a temporary table guarantees consistent and stable navigation results for the entire lifecycle of the temporary table and completely eliminates the hierarchy generation cost for subsequent queries:

```
CREATE LOCAL TEMPORARY COLUMN TABLE #h_demo AS (
SELECT *
FROM HIERARCHY (
    SOURCE t_demo
    SIBLING ORDER BY ord )
ORDER BY
    hierarchy_rank );
SELECT
    hierarchy_rank,
    parent_id,
    node_id
FROM #h_demo;
```

HIERARCHY_RANK	PARENT_ID	NODE_ID
1		A1
2	A1	B1
3	B1	C1
4	B1	C2
5	A1	B2
6	B2	C3
7	C3	D1
8	C3	D2
9	B2	C4
10	C4	D3

Materializing a hierarchy into a temporary or persistent table cuts all connections between the hierarchy and its sources. If a query should return the most current state of the data, it is recommended to create a view over a hierarchy generator function, which guarantees that hierarchy navigation results always correctly reflect the current transactional view of the source data. If a hierarchy source is fully deterministic (that means that multiple query executions over the same data set always return the same result), the hierarchy generator function output will be cached automatically. Caching avoids the overhead of recalculating the hierarchy if the sources do not change between subsequent navigations and at the same time guarantees a transactionally correct result:

```
CREATE VIEW h_demo AS
SELECT *
FROM HIERARCHY (
    SOURCE t_demo
    SIBLING ORDER BY ord )
ORDER BY
    hierarchy_rank;
SELECT
    hierarchy_rank,
    parent_id,
    node_id
FROM h_demo;
```

HIERARCHY_RANK	PARENT_ID	NODE_ID
1		A1
2	A1	B1
3	B1	C1
4	B1	C2
5	A1	B2
6	B2	C3
7	C3	D1
8	C3	D2
9	B2	C4

HIERARCHY_RANK	PARENT_ID	NODE_ID
10	C4	D3

Some of the hierarchy navigation functions can also work on partial subtrees of a hierarchy. To demonstrate their use, the view H_DEMO_B2 only contains the complete subtree under node B2.

```
CREATE VIEW subtree_B2 AS
SELECT
    hierarchy_rank,
    hierarchy_tree_size,
    hierarchy_parent_rank,
    hierarchy_level,
    hierarchy_is_cycle,
    hierarchy_is_orphan,
    parent_id,
    node_id,
    ord,
    amount
FROM
HIERARCHY_DESCENDANTS(
    SOURCE h_demo
    START WHERE node_id = 'B2');
SELECT
    hierarchy_rank,
    parent_id,
    node_id
FROM subtree_B2 ORDER BY hierarchy_rank;
```

HIERARCHY_RANK	PARENT_ID	NODE_ID
5	A1	B2
6	B2	C3
7	C3	D1
8	C3	D2
9	B2	C4
10	C4	D3

In an analog way, the view H_DEMO_ERR is defined as:

```
CREATE VIEW h_demo_err AS
SELECT *
FROM HIERARCHY (
    SOURCE t_demo_err
    SIBLING ORDER BY node_id )
ORDER BY
    hierarchy_rank;
SELECT
    hierarchy_rank,
    parent_id,
    node_id
FROM h_demo;
```

HIERARCHY_RANK	PARENT_ID	NODE_ID
1		A1

HIERARCHY_RANK	PARENT_ID	NODE_ID
2	A1	B1
3	B1	C1
4	B1	C2
5	A1	B2
6	B2	C3
7	C3	D1
8	C3	D2
9	B2	C4
10	C4	D3

⚠ Caution

Please be aware that hierarchy navigation functions consuming views over hierarchies as a source (or hierarchy generation function results directly) require fully deterministic sources for correct operation, including a stable sort order of sibling nodes. Otherwise, it cannot be guaranteed that hierarchy ranks calculated within the source specification semantically match the hierarchy ranks calculated within in the START / START WHERE specification, leading to non-deterministic and wrong results. This also applies to hierarchy self-joins and similar operations.

For compatibility reasons, a strict enforcement of the `<hierarchy_genfunc_order_spec>` is not possible. Therefore, a potential determinism issue is indicated by a warning:

```
Warning 654:
no row order on table set:
hierarchy generation function result is partially non-deterministic due to
missing SIBLING ORDER BY clause or missing ORDER BY clause in SOURCE
specification.
```

However, it cannot be detected if a SIBLING ORDER BY clause is present, but does not ensure a deterministic sibling sort order.

5.2 HIERARCHY_DESCENDANTS Navigation Function

Returns all descendants of a set of start nodes in a hierarchy.

Syntax

```
HIERARCHY_DESCENDANTS (
    <hierarchy_navfunc_source_spec>
    [<hierarchy_navfunc_start_spec>]
    [<hierarchy_navfunc_distance_spec>]
```

```
)
```

Syntax Elements

<hierarchy_navfunc_source_spec>

Specifies a hierarchy for the function to operate on.

```
<hierarchy_navfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <hierarchy_generation_function>
    | <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

`<hierarchy_generation_function>` specifies a hierarchy generation function directly (for example, the HIERARCHY function).

Generic database objects must be an unfiltered view or materialized result set, such as a table, containing all of the basic hierarchy attributes computed by a hierarchy generation function. A filtered result with all the basic hierarchy attributes containing the complete subtree of particular node of a hierarchy is also supported. If a hierarchy contains multiple root nodes, this can also be the complete tree of one particular root node in the original hierarchy.

<hierarchy_navfunc_start_spec>

Specifies the start nodes as an additional input table or as a filter condition on the source. If `<hierarchy_navfunc_start_spec>` is not specified, then the navigation starts from all nodes in the hierarchy.

```
<hierarchy_navfunc_start_spec> ::= START { <table_valued_expression> | WHERE
<condition> }
<table_valued_expression> ::=
    <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The `<table_valued_expression>` object must contain, at minimum, a column named or aliased as START_RANK that has a data type that can be cast to BIGINT. `<condition>` specifies a starting condition that is semantically equivalent to `START ( SELECT hierarchy_rank AS start_rank FROM <source> WHERE <condition> )`.

<hierarchy_navfunc_distance_spec>

Specifies a distance window filtering the function result.

```
<hierarchy_navfunc_distance_spec> ::= DISTANCE {
    | <from_to_expression>
    | <from_expression>
    | <to_expression>
```

```
    | <expression>
}
<from_to_expression> ::= <from_expression> <to_expression>
<from_expression>    ::= FROM <expression>
<to_expression>      ::= TO <expression>
```

- **<from_to_expression>**
  Specifies a minimum and a maximum HIERARCHY_DISTANCE. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance BETWEEN <from_expression> AND <to_expression>`.
- **<from_expression>**
  Specifies a scalar integer value as a minimum HIERARCHY_DISTANCE. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance >= <expression>`.
- **<to_expression>**
  Specifies a scalar integer value as a maximum HIERARCHY_DISTANCE. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance <= <expression>`.
- **<expression>**
  Specifies a scalar integer value. If only a single `<expression>` is specified, then the HIERARCHY_DISTANCE of all result rows exactly matches this value. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance = <expression>`.

If a distance filter is specified, then it is applied before result materialization and, therefore, may accelerate query execution. By default, no distance filter is applied.

Description

The HIERARCHY_DESCENDANTS function extracts partial trees from a given hierarchy starting from and including a known set of start nodes. The result can be prefiltered by a distance window.

Column-wise, the function projects all attributes of the source hierarchy plus a lateral projection of the corresponding START record. Additionally, a HIERARCHY_DISTANCE column is generated, which contains the level difference between a descendant node and its respective start node. The returned distances are always larger than or equal to 0. The data type of HIERARCHY_DISTANCE is INTEGER NOT NULL.

Due to its flexibility, the HIERARCHY_DESCENDANTS function provides an efficient means for several typical types of hierarchy navigation such as the determination of children, subordinate leaves, subtrees, or connectivity tests.

Examples

The examples are based on the data set introduced in the section Topologies Used in Examples [page 7]. The view H_DEMO is defined in section Separating Hierarchy Model From Navigation [page 40].

HIERARCHY_DESCENDANTS operating on an unfiltered hierarchy

You determine the set of descendants of node A1 with a depth window between 1 and 2 (that is, levels B and C):

```
SELECT
    hierarchy_rank,
    hierarchy_level,
```

```
    node_id,
    hierarchy_distance
FROM HIERARCHY_DESCENDANTS (
    SOURCE h_demo
    START WHERE node_id = 'A1'
    DISTANCE FROM 1 TO 2 )
ORDER BY
    hierarchy_rank;
```

HIERARCHY_RANK	HIERARCHY_LEVEL	NODE_ID	HIERARCHY_DISTANCE
2	2	B1	1
3	3	C1	2
4	3	C2	2
5	2	B2	1
6	3	C3	2
9	3	C4	2

You calculate the partial hierarchies with all attributes starting from B1 and B2:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_distance AS distance,
    start_rank,
    start_id
FROM HIERARCHY_DESCENDANTS (
    SOURCE h_demo
    START ( SELECT hierarchy_rank AS start_rank,  node_id AS start_id FROM
h_demo WHERE node_id IN ('B1',  'B2') ) )
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	DISTANCE	START_RANK	START_ID
2	3	1	2	A1	B1	0	2	B1
3	1	2	3	B1	C1	1	2	B1
4	1	2	3	B1	C2	1	2	B1
5	6	1	2	A1	B2	0	5	B2
6	3	5	3	B2	C3	1	5	B2
7	1	6	4	C3	D1	2	5	B2
8	1	6	4	C3	D2	2	5	B2
9	2	5	3	B2	C4	1	5	B2
10	1	9	4	C4	D3	2	5	B2

You count the number of subordinate leaves of all nodes:

```
SELECT DISTINCT
    start_rank AS hierarchy_rank,
    node_id,
    COUNT(*) OVER ( PARTITION BY start_rank ) AS num_leaves
FROM
    HIERARCHY_DESCENDANTS( SOURCE h_demo )
WHERE
    hierarchy_tree_size = 1
ORDER BY
    start_rank;
```

HIERARCHY_RANK	NODE_ID	NUM_LEAVES
1	C1	5
1	C2	5
1	D1	5
1	D2	5
1	D3	5
2	C1	2
2	C2	2
3	C1	1
4	C2	1
5	D3	3
5	D2	3
5	D1	3
6	D1	2
6	D2	2
7	D1	1
8	D2	1
9	D3	1
10	D3	1

HIERARCHY_DESCENDANTS operating on a complete subtree in a hierarchy

HIERARCHY_DESCENDANTS can also operate on a SOURCE, which is a complete subtree of a node in a given hierarchy. The view SUBTREE_B2 is defined in the section Navigating Hierarchies [page 40].

You calculate the descendants of node C4 within the subtree of node B2:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_distance AS distance,
    start_rank
```

```
FROM HIERARCHY_DESCENDANTS(
    SOURCE subtree_B2
    START WHERE node_id = 'C4')
ORDER BY hierarchy_rank;
```

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	PARENT_ID	NODE_ID	DISTANCE	START_RANK
9	2	5	3	B2	C4	0	9
10	1	9	4	C4	D3	1	9

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

5.3 HIERARCHY_ANCESTORS Navigation Function

Returns all ancestors of a set of start nodes in a hierarchy.

Syntax

```
HIERARCHY_ANCESTORS (
    <hierarchy_navfunc_source_spec>
    [<hierarchy_navfunc_start_spec>]
    [<hierarchy_navfunc_distance_spec>]
)
```

Syntax Elements

<hierarchy_navfunc_source_spec>

Specifies a hierarchy for the function to operate on.

```
<hierarchy_navfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <hierarchy_generation_function>
    | <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

`<hierarchy_generation_function>` specifies a hierarchy generation function directly (for example, the HIERARCHY function).

Generic database objects must be an unfiltered view or materialized result set, such as a table, containing all of the basic hierarchy attributes computed by a hierarchy generation function. A filtered result with all the basic hierarchy attributes containing the complete subtree of particular node of a hierarchy is also supported. If a hierarchy contains multiple root nodes, this can also be the complete tree of one particular root node in the original hierarchy.

<hierarchy_navfunc_start_spec>

Specifies the start nodes as an additional input table or as a filter condition on the source. If `<hierarchy_navfunc_start_spec>` is not specified, then the navigation starts from all nodes in the hierarchy.

```
<hierarchy_navfunc_start_spec> ::= START { <table_valued_expression> | WHERE
<condition> }
<table_valued_expression> ::=
    <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The `<table_valued_expression>` object must contain, at minimum, a column named or aliased as START_RANK that has a data type that can be cast to BIGINT. `<condition>` specifies a starting condition that is semantically equivalent to `START ( SELECT hierarchy_rank AS start_rank FROM <source> WHERE <condition> )`.

<hierarchy_navfunc_distance_spec>

Specifies a distance window filtering the function result.

```
<hierarchy_navfunc_distance_spec> ::= DISTANCE {
    | <from_to_expression>
    | <from_expression>
    | <to_expression>
    | <expression>
}
<from_to_expression> ::= <from_expression> <to_expression>
<from_expression>    ::= FROM <expression>
<to_expression>      ::= TO <expression>
```

- **<from_to_expression>**
  Specifies a minimum and a maximum HIERARCHY_DISTANCE. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance BETWEEN <from_expression> AND <to_expression>`.
- **<from_expression>**
  Specifies a scalar integer value as a minimum HIERARCHY_DISTANCE. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance >= <expression>`.
- **<to_expression>**
  Specifies a scalar integer value as a maximum HIERARCHY_DISTANCE. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance <= <expression>`.

- **<expression>**
  Specifies a scalar integer value. If only a single `<expression>` is specified, then the HIERARCHY_DISTANCE of all result rows exactly matches this value. Result-wise, it is equivalent to a post-filter condition `WHERE hierarchy_distance = <expression>`.

If a distance filter is specified, then it is applied before result materialization and, therefore, may accelerate query execution. By default, no distance filter is applied.

Description

The HIERARCHY_ANCESTORS function extracts ascending branches from a given hierarchy starting from and including a known set of start nodes. The result can be prefiltered by a distance window.

Column-wise, the function projects all attributes of the source hierarchy plus a lateral projection of the corresponding START record. Additionally, a HIERARCHY_DISTANCE column is generated, which contains the level difference between an ancestor node and its respective start node. The returned distances are always less than or equal to 0. The data type of HIERARCHY_DISTANCE is INTEGER NOT NULL.

Due to its flexibility, the HIERARCHY_ANCESTORS function provides an efficient means for several types of typical hierarchy navigation such as the determination of parents, paths to the root, or upward connectivity tests.

Examples

The examples are based on the data set introduced in the section Topologies Used in Examples [page 7]. The view H_DEMO is defined in section Separating Hierarchy Model From Navigation [page 40].

HIERARCHY_ANCESTORS operating on an unfiltered hierarchy

You determine the ancestors of node C4:

```
SELECT
    hierarchy_rank,
    hierarchy_level,
    node_id,
    hierarchy_distance
FROM HIERARCHY_ANCESTORS (
    SOURCE h_demo
    START WHERE node_id = 'C4' )
ORDER BY
    node_id;
```

HIERARCHY_RANK	HIERARCHY_LEVEL	NODE_ID	HIERARCHY_DISTANCE
1	1	A1	-2
5	2	B2	-1
9	3	C4	0

You calculate the grandparents of nodes C1 and D1:

```
SELECT
    hierarchy_rank,
    hierarchy_level,
    start_id,
    node_id AS grandparent_id
FROM HIERARCHY_ANCESTORS (
    SOURCE h_demo
    START ( SELECT hierarchy_rank AS start_rank, node_id AS start_id FROM h_demo
WHERE node_id IN ('C1', 'D1') )
    DISTANCE -2 )
ORDER BY
    start_rank ASC, hierarchy_rank DESC;
```

HIERARCHY_RANK	HIERARCHY_LEVEL	START_ID	GRANDPARENT_ID
1	1	C1	A1
5	2	D1	B2

HIERARCHY_ANCESTORS operating on a complete subtree in a hierarchy

HIERARCHY_ANCESTORS can also operate on a SOURCE, which is a complete subtree of a node in a given hierarchy. The view SUBTREE_B2 is defined in the section Navigating Hierarchies [page 40].

You calculate the ancestors of node C4 within the subtree of node B2:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_distance AS distance,
    start_rank
FROM HIERARCHY_ANCESTORS(
    SOURCE subtree_B2
    START where node_id = 'C4')
ORDER BY hierarchy_rank;
```

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	PARENT_ID	NODE_ID	DISTANCE	START_RANK
5	6	1	2	A1	B2	-1	9
9	2	5	3	B2	C4	0	9

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

5.4 HIERARCHY_SIBLINGS Navigation Function

Returns all siblings of a set of start nodes, including the start nodes.

Syntax

```
HIERARCHY_SIBLINGS (
    <hierarchy_navfunc_source_spec>
    [ <hierarchy_navfunc_start_spec> ]
)
```

Syntax Elements

<hierarchy_navfunc_source_spec>

Specifies a hierarchy for the function to operate on.

```
<hierarchy_navfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <hierarchy_generation_function>
    | <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

`<hierarchy_generation_function>` specifies a hierarchy generation function directly (for example, the HIERARCHY function).

Generic database objects must be an unfiltered view or materialized result set, such as a table, containing all of the basic hierarchy attributes computed by a hierarchy generation function. A filtered result with all the basic hierarchy attributes containing the complete subtree of particular node of a hierarchy is also supported. If a hierarchy contains multiple root nodes, this can also be the complete tree of one particular root node in the original hierarchy.

<hierarchy_navfunc_start_spec>

Specifies the start nodes as an additional input table or as a filter condition on the source. If `<hierarchy_navfunc_start_spec>` is not specified, then the navigation starts from all nodes in the hierarchy.

```
<hierarchy_navfunc_start_spec> ::= START { <table_valued_expression> | WHERE
<condition> }
<table_valued_expression> ::=
    <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The `<table_valued_expression>` object must contain, at minimum, a column named or aliased as START_RANK that has a data type that can be cast to BIGINT. `<condition>` specifies a starting condition that is semantically equivalent to `START ( SELECT hierarchy_rank AS start_rank FROM <source> WHERE <condition> )`.

Description

The HIERARCHY_SIBLINGS function returns siblings of a set of start nodes including the respective start nodes. The result can be prefiltered by a distance window.

Column-wise, the function projects all attributes of the source hierarchy plus a lateral projection of the corresponding START record. Additionally, a HIERARCHY_SIBLING_DISTANCE column is generated, which contains the difference between the HIERARCHY_RANK values from a result node and its corresponding start node. The data type of HIERARCHY_SIBLING_DISTANCE is BIGINT NOT NULL.

The main purpose of the HIERARCHY_SIBLING_DISTANCE column is the determination of siblings with particular relative locations, such as the first sibling (minimum distance), the next preceding sibling (highest distance less than 0), the next following sibling (lowest distance greater than 0), and the last sibling (maximum distance).

Examples

The examples are based on the data set introduced in the section Topologies Used in Examples [page 7]. The view H_DEMO is defined in section Separating Hierarchy Model From Navigation [page 40]. Further examples are given in section *First sibling, previous sibling, next sibling, last sibling* in Custom Navigations [page 62].

HIERARCHY_SIBLINGS operating on an unfiltered hierarchy

You determine the set of siblings of node C4:

```
SELECT DISTINCT
    hierarchy_rank,
    hierarchy_level,
    node_id,
    hierarchy_sibling_distance
FROM HIERARCHY_SIBLINGS (
    SOURCE h_demo
    START WHERE node_id = 'C4' )
ORDER BY
    node_id;
```

HIERARCHY_RANK	HIERARCHY_LEVEL	NODE_ID	HIERARCHY_SIBLING_DISTANCE
6	3	C3	-3
9	3	C4	0

HIERARCHY_SIBLINGS operating on a complete subtree in a hierarchy

HIERARCHY_SIBLINGS can also operate on a SOURCE, which is a complete subtree of a node in a given hierarchy. The view SUBTREE_B2 is defined in the section Navigating Hierarchies [page 40].

You calculate the set of siblings of node C4 within the subtree of node B2:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_sibling_distance AS sibling_distance
FROM HIERARCHY_SIBLINGS(
    SOURCE subtree_B2
    START WHERE node_id = 'C4')
ORDER BY hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	SIBLING_DISTANCE
6	3	5	3	B2	C3	-3
9	2	5	3	B2	C4	0

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

5.5 HIERARCHY_DESCENDANTS_AGGREGATE Navigation Function

Efficiently calculates aggregates along a hierarchy in a bottom-up direction.

Syntax

```
HIERARCHY_DESCENDANTS_AGGREGATE (
    <hierarchy_navfunc_source_spec>
    [<hierarchy_aggfunc_join_spec>]
    <hierarchy_aggfunc_measures_spec>
    [<hierarchy_aggfunc_subtotal_spec>]
    [<hierarchy_aggfunc_balance_spec>]
    [<hierarchy_aggfunc_not_matched_spec>]
    [<hierarchy_aggfunc_total_spec>]
```

```
)
```

Syntax Elements

<hierarchy_navfunc_source_spec>

Specifies a hierarchy for the function to operate on.

```
<hierarchy_navfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <hierarchy_generation_function>
    | <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

`<hierarchy_generation_function>` specifies a hierarchy generation function directly (for example, the HIERARCHY function).

Generic database objects must be an unfiltered view or materialized result set, such as a table, containing all of the basic hierarchy attributes computed by a hierarchy generation function.

<hierarchy_aggfunc_join_spec>

An additional join condition (as in the SELECT statement) to join a fact table to the source. By default, the join is a left outer join. If a not matched specification is present, it is a full outer join. Attributes to be aggregated can also originate directly from the source hierarchy.

```
<hierarchy_aggfunc_join_spec> ::= JOIN <table> ON <predicate>
```

See the *joined_table* section in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

<hierarchy_aggfunc_measures_spec>

Specifies the columns containing the values to aggregate, the functions used, the names of the result columns, and an optional condition (WHERE clause) defining the set of nodes for which aggregated values are returned. If no condition is defined, then aggregates are returned for all nodes.

```
<hierarchy_aggfunc_measures_spec> ::=
    MEASURES ( <hierarchy_aggfunc_measures_list> )  [ WHERE <condition> ]
<hierarchy_aggfunc_measures_list> ::=
    <hierarchy_aggfunc_measure_item> [ { ,
<hierarchy_aggfunc_measure_item> } ... ]
<hierarchy_aggfunc_measure_item> ::=
    <aggfunc_name> ( [ DISTINCT | ALL ] { <expression> | * } ) [ AS
<column_alias> ]
```

The following aggregate functions are supported within the measures specification. The functions usually take a single argument, which can be an arbitrary expression. Multiple column names in the same argument can only refer to either the SOURCE table or the JOIN table, not both.

- SUM

- PRODUCT
- COUNT
- COUNT DISTINCT
- AVG
- MIN
- MAX

If the source column of a measure is contained in the joined fact table, only the following aggregation functions are supported:

- SUM
- COUNT (excluding COUNT(*))
- MIN
- MAX

<hierarchy_aggfunc_subtotal_spec>

Specifies that an additional result row shall be calculated showing the total of all root nodes included in the traversal. If this specification is missing, then the row is not included in the result. The subtotal result row is identified by result column HIERARCHY_AGGREGATE_TYPE value 1. An optional scalar expression sets the NODE_ID value of this result row.

```
<hierarchy_aggfunc_subtotal_spec> ::= WITH SUBTOTAL [ <scalar_expression> ]
```

<hierarchy_aggfunc_balance_spec>

Specifies that an additional result row shall be calculated showing the total of all nodes and facts not included in the traversal. If this specification is missing, then the row is not included in the result. The balance result row is identified by result column HIERARCHY_AGGREGATE_TYPE value 2. An optional scalar expression sets the NODE_ID value of this result row.

```
<hierarchy_aggfunc_balance_spec> ::= WITH BALANCE [ <scalar_expression> ]
```

<hierarchy_aggfunc_not_matched_spec>

Specifies that an additional result row shall be calculated showing the total of all facts that do not join with any hierarchy node. This specification is only applicable if a join is also specified; if it is missing, then the row is not included in the result. The not matched result row is identified by result column HIERARCHY_AGGREGATE_TYPE value 3. An optional scalar expression sets the NODE_ID value of this result row.

```
<hierarchy_aggfunc_not_matched_spec> ::= WITH NOT MATCHED [ <scalar_expression> ]
```

<hierarchy_aggfunc_total_spec>

Specifies that an additional result row shall be calculated showing the total of all nodes and facts. For a sum aggregation, this value is equal to the sum of the values for subtotal, balance, and not matched (for joined measures). If this specification is missing, then the row is not included in the result. The total result row is identified by result column HIERARCHY_AGGREGATE_TYPE value 4. An optional scalar expression sets the NODE_ID value of this result row.

```
<hierarchy_aggfunc_total_spec> ::= WITH TOTAL [ <scalar_expression> ]
```

Description

The hierarchy descendants aggregation function provides optimized hierarchical aggregate capabilities. By reusing results of subordinate nodes, all aggregates can be calculated by one single linear index traversal. The result columns consist of a projection of the source columns, a HIERARCHY_AGGREGATE_TYPE column, and the calculated measures. The HIERARCHY_AGGREGATE_TYPE value qualifies a row either as a normal node (0), as a subtotal aggregate (1), as a balance aggregate (2), as a not matched aggregate (3), or as a total aggregate (4).

Examples

The examples are based on the data set introduced in the section Topologies Used in Examples [page 7]. The view H_DEMO is defined in section Separating Hierarchy Model From Navigation [page 40]. They are also based on the fact table H_DEMO_FACTS from the section Navigating Hierarchies [page 40].

Example

The following query joins the hierarchy to the fact table and calculates various aggregate measures, originating both from the hierarchy itself and the fact table. The result set contains all nodes with a level less than or equal to 2 plus additional subtotal, balance, and total result rows.

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_level AS level,
    hierarchy_aggregate_type AS type,
    node_id,
    amount AS amount_int,
    avg_amount_int,
    sum_amount_dec,
    sum_amount_cents,
    num_nodes,
    num_facts
FROM
    HIERARCHY_DESCENDANTS_AGGREGATE(
        SOURCE h_demo
        JOIN h_demo_facts ON node_id = node
        MEASURES (
            AVG(h_demo.amount) AS avg_amount_int,
            SUM(h_demo_facts.amount_dec_fact) AS sum_amount_dec,
            SUM(h_demo_facts.amount_dec_fact * 100) AS sum_amount_cents,
            COUNT(DISTINCT h_demo.hierarchy_rank) AS num_nodes,
            COUNT(h_demo_facts.amount_dec_fact) AS num_facts
        ) WHERE hierarchy_level <= 2
        WITH SUBTOTAL '(subtotal)'
        WITH BALANCE '(remainder)'
        WITH NOT MATCHED '(unassigned)'
        WITH TOTAL '(total)'
    )
ORDER BY
    type, rank;
```

RANK	LEVEL	TYPE	NODE_ID	AMOUNT_INT	AVG_AMOUNT_INT	SUM_AMOUNT_DEC	SUM_AMOUNT_CENTS	NUM_NODES	NUM_FACTS
1	1	0	A1	1	2	40.1	4010	10	10
2	2	0	B1	2	2	8.2	820	3	2
5	2	0	B2	4	2.166666	26.6	2660	6	7
		1	(subtotal)		2	40.1	4010	10	10
		2	(remainder)			0	0	0	0
		3	(unassigned)			25.5	2550		3
		4	(total)		2	65.6	6560	10	13

Just for comparison, an equivalent result (without subtotal, remainder and total) can be calculated less efficiently by using the conventional descendants navigation function and standard SQL aggregation as follows:

```
WITH
    aggr_h AS (
        SELECT
            start_rank,
            AVG(amount) AS avg_amount_int,
            COUNT(DISTINCT hierarchy_rank) AS num_nodes
        FROM
            HIERARCHY_DESCENDANTS(
                SOURCE h_demo
                START WHERE hierarchy_level <= 2
            )
        GROUP BY
            start_rank
    ),
    aggr_f AS (
        SELECT
            start_rank,
            SUM(amount_dec_fact) AS sum_amount_dec,
            SUM(amount_dec_fact * 100) AS sum_amount_cents,
            COUNT(*) AS num_facts
        FROM
            HIERARCHY_DESCENDANTS(
                SOURCE h_demo
                START WHERE hierarchy_level <= 2
            )
            RIGHT JOIN h_demo_facts ON node_id = node
        GROUP BY
            start_rank
    )
SELECT
    hierarchy_rank AS rank,
    hierarchy_level AS level,
    COALESCE(node_id, '(unassigned)') AS node_id,
    amount AS amount_int,
    avg_amount_int,
    sum_amount_dec,
    sum_amount_cents,
    num_nodes,
    COALESCE(num_facts, 0) as num_facts
FROM
    aggr_h
    FULL JOIN aggr_f ON aggr_h.start_rank = aggr_f.start_rank
    LEFT JOIN h_demo ON aggr_h.start_rank = h_demo.hierarchy_rank
```

```
ORDER BY
    rank NULLS LAST;
```

RANK	LEVEL	NODE_ID	AMOUNT_INT	AVG_AMOUNT_INT	SUM_AMOUNT_DEC	SUM_AMOUNT_CENTS	NUM_NODES	NUM_FACTS
1	1	A1	1	2	40.1	4010	10	10
2	2	B1	2	2	8.2	820	3	2
5	2	B2	4	2.166666	26.6	2660	6	7
		(unassigned)			25.5	2550		3

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

5.6 HIERARCHY_ANCESTORS_AGGREGATE Navigation Function

Efficiently calculates aggregates along a hierarchy in a top-down direction.

Syntax

```
HIERARCHY_ANCESTORS_AGGREGATE (
    <hierarchy_navfunc_source_spec>
    [<hierarchy_navfunc_start_spec>]
    <hierarchy_aggfunc_measures_spec>
)
```

Syntax elements

<hierarchy_navfunc_source_spec>

Specifies a hierarchy for the function to operate on.

```
<hierarchy_navfunc_source_spec> ::= SOURCE <table_valued_expression>
<table_valued_expression> ::=
    <hierarchy_generation_function>
```

```
    | <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

`<hierarchy_generation_function>` specifies a hierarchy generation function directly (for example, the HIERARCHY function).

Generic database objects must be an unfiltered view or materialized result set, such as a table, containing all of the basic hierarchy attributes computed by a hierarchy generation function.

<hierarchy_navfunc_start_spec>

Specifies the start nodes as an additional input table or as a filter condition on the source. If `<hierarchy_navfunc_start_spec>` is not specified, then the navigation starts at all roots. In that case, all regular nodes are reached exactly once.

```
<hierarchy_navfunc_start_spec> ::= START { <table_valued_expression> | WHERE
<condition> }
<table_valued_expression> ::=
    <table_ref>
    | <function_reference>
    | <table_variable>
```

See the *from_clause* in the SELECT Statement (Data Manipulation) in the SAP HANA SQL and System Views Reference.

The `<table_valued_expression>` object must contain, at minimum, a column named or aliased as START_RANK that has a data type that can be cast to BIGINT. `<condition>` specifies a starting condition that is semantically equivalent to `START ( SELECT hierarchy_rank AS start_rank FROM <source> WHERE <condition> )`.

<hierarchy_aggfunc_measures_spec>

Specifies the columns containing the values to aggregate, the functions used, the names of the result columns, and an optional condition (WHERE clause) defining the set of nodes for which aggregated values are returned. If no condition is defined, then aggregates are returned for all nodes.

```
<hierarchy_aggfunc_measures_spec> ::=
    MEASURES ( <hierarchy_aggfunc_measures_list> )  [ WHERE <condition> ]
<hierarchy_aggfunc_measures_list> ::=
    <hierarchy_aggfunc_measure_item> [ { ,
<hierarchy_aggfunc_measure_item> } ... ]
<hierarchy_aggfunc_measure_item> ::=
    <aggfunc_name> ( [ DISTINCT | ALL ] { <expression> | * } ) [ AS
<column_alias> ]
```

The following aggregate functions are supported within the measures specification. The functions usually take a single argument, which can be an arbitrary expression.

- SUM
- PRODUCT
- COUNT
- COUNT DISTINCT
- AVG

- MIN
- MAX
- STRING_AGG
  This function takes two arguments. The first is the usual expression, and the second one is a string used as a delimiter.

Description

The hierarchy ancestors aggregation function provides optimized hierarchical aggregate capabilities. By reusing results of subordinate nodes, all aggregates can be calculated by one single linear index traversal.

Example

The example is based on the data set introduced in the section Topologies Used in Examples [page 7]. The view H_DEMO is defined in section Separating Hierarchy Model From Navigation [page 40].

You calculate a product aggregate measure and a string aggregate for nodes C3 and D1 starting from B2:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    amount,
    prod_amount,
    path
FROM
    HIERARCHY_ANCESTORS_AGGREGATE(
        SOURCE h_demo
        START WHERE node_id = 'B2'
        MEASURES (
            PRODUCT(amount) AS prod_amount,
            STRING_AGG(node_id, '/') AS path
        ) WHERE node_id IN ( 'C3', 'D1' )
    );
```

RANK	LEVEL	PARENT_ID	NODE_ID	AMOUNT	PROD_AMOUNT	PATH
6	3	B2	C3	1	4	B2/C3
7	4	C3	D1	2	8	B2/C3/D1

Related Information

SAP HANA SQL Reference Guide for SAP HANA Platform
SELECT Statement (Data Manipulation)

5.7 Custom Navigations

Descendants, subtree

The local temporary table #H_DEMO is defined in the section Separating Hierarchy Model From Navigation [page 40]. Besides using the HIERARCHY_DESCENDANTS Navigation Function [page 43], subtree navigations can be executed by using hierarchy attribute arithmetics as well:

```
SELECT
    h2.hierarchy_rank AS rank,
    h2.hierarchy_tree_size AS tree_size,
    h2.hierarchy_parent_rank AS parent_rank,
    h2.hierarchy_level AS level,
    h2.parent_id,
    h2.node_id,
    h2.hierarchy_level - h1.hierarchy_level AS distance,
    h1.hierarchy_rank AS start_rank
FROM #h_demo AS h1, #h_demo AS h2
WHERE h1.node_id = 'B2'
      AND h2.hierarchy_rank
        BETWEEN h1.hierarchy_rank
        AND h1.hierarchy_rank + h1.hierarchy_tree_size - 1
ORDER BY h2.hierarchy_rank;
```

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	PARENT_ID	NODE_ID	DISTANCE	START_RANK
5	6	1	2	A1	B2	0	5
6	3	5	3	B2	C3	1	5
7	1	6	4	C3	D1	2	5
8	1	6	4	C3	D2	2	5
9	2	5	3	B2	C4	1	5
10	1	9	4	C4	D3	2	5

Children

Restrict the result to the direct children of B2.

Based on a navigation function

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_distance AS distance,
    start_rank
FROM HIERARCHY_DESCENDANTS(
        SOURCE #h_demo
```

```
        START WHERE node_id = 'B2'
        DISTANCE 1
    )
ORDER BY hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	DISTANCE	START_RANK
6	3	5	3	B2	C3	1	5
9	2	5	3	B2	C4	1	5

Based on hierarchy attributes

```
SELECT
    h2.hierarchy_rank AS rank,
    h2.hierarchy_tree_size AS tree_size,
    h2.hierarchy_parent_rank AS parent_rank,
    h2.hierarchy_level AS level,
    h2.parent_id,
    h2.node_id,
    h2.hierarchy_level - h1.hierarchy_level AS distance,
    h1.hierarchy_rank AS start_rank
FROM #h_demo AS h1, #h_demo AS h2
WHERE h1.node_id = 'B2'
    AND h2.hierarchy_parent_rank = h1.hierarchy_rank
ORDER BY h2.hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	DISTANCE	START_RANK
6	3	5	3	B2	C3	1	5
9	2	5	3	B2	C4	1	5

Leaves

Extract the leaves below a given set of reference nodes (B1, B2, B3).

Based on a navigation function

```
SELECT
    hierarchy_rank,
    start_id,
    node_id AS leaf_id
FROM HIERARCHY_DESCENDANTS(
     SOURCE #h_demo
     START ( SELECT hierarchy_rank AS start_rank, node_id AS start_id
               FROM #h_demo WHERE node_id IN ( 'B1', 'B2', 'B3') ) )
WHERE hierarchy_tree_size = 1
ORDER BY start_rank, hierarchy_rank;
```

HIERARCHY_RANK	START_ID	LEAF_ID
3	B1	C1

HIERARCHY_RANK	START_ID	LEAF_ID
4	B1	C2
7	B2	D1
8	B2	D2
10	B2	D3

Based on hierarchy attribute arithmetics

```
SELECT
    h2.hierarchy_rank,
    h1.node_id as start_id,
    h2.node_id AS leaf_id
FROM
    #h_demo AS h1, #h_demo AS h2
WHERE h1.node_id IN ( 'B1', 'B2', 'B3' )
      AND h2.hierarchy_rank BETWEEN h1.hierarchy_rank
                            AND h1.hierarchy_rank + h1.hierarchy_tree_size - 1
      AND h2.hierarchy_tree_size = 1
ORDER BY h1.hierarchy_rank, h2.hierarchy_rank;
```

HIERARCHY_RANK	START_ID	LEAF_ID
3	B1	C1
4	B1	C2
7	B2	D1
8	B2	D2
10	B2	D3

Members of a given level

The most straightforward way to express this navigation is by filtering a hierarchy generator function result on HIERARCHY_LEVEL.

```
SELECT
    hierarchy_rank,
    node_id
FROM #h_demo
WHERE hierarchy_level = 2
ORDER BY hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
2	B1
5	B2

Parents

Determine the parents of node D3.

Based on a navigation function

```
SELECT
    hierarchy_rank,
    node_id
FROM HIERARCHY_ANCESTORS( SOURCE #h_demo START WHERE node_id = 'D3' DISTANCE -1 )
ORDER BY hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
9	C4

Based on hierarchy attribute arithmetics

```
SELECT
    h2.hierarchy_rank,
    h2.node_id
FROM #h_demo AS h1, #h_demo AS h2
WHERE h1.node_id = 'D3'
      AND h1.hierarchy_rank BETWEEN h2.hierarchy_rank
                            AND h2.hierarchy_rank + h2.hierarchy_tree_size - 1
      AND h1.hierarchy_level = h2.hierarchy_level + 1
ORDER BY h2.hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
9	C4

Roots

A navigation to roots can be expressed by adding a filter on `HIERARCHY_PARENT_RANK = 0`. Note that `HIERARCHY_LEVEL = 1` is not an appropriate root filter condition, because it is not guaranteed that a root node is always on level 1. Yet the most efficient means to determine all roots is querying the siblings of the first hierarchy node:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    hierarchy_sibling_distance AS sibling_distance,
    start_rank
FROM HIERARCHY_SIBLINGS( SOURCE h_demo_err START WHERE hierarchy_rank = 1 )
ORDER BY hierarchy_rank ASC;
```

RANK	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID	SIBLING_DIS-TANCE	START_RANK
1	0	1		E1	0	1
6	0	1		E2	5	1
13	0	1		E3	12	1

First sibling, previous sibling, next sibling, last sibling

These navigations can be expressed as filters on the HIERARCHY_SIBLING_DISTANCE attribute of the HIERARCHY_SIBLINGS Navigation Function [page 52]:

First sibling of node C2

```
SELECT TOP 1
    hierarchy_rank,
    node_id
FROM HIERARCHY_SIBLINGS( SOURCE #h_demo START WHERE node_id = 'C2' )
ORDER BY hierarchy_sibling_distance ASC;
```

HIERARCHY_RANK	NODE_ID
3	C1

Previous sibling (the sibling to the immediate left) of node C2

```
SELECT TOP 1
    hierarchy_rank,
    node_id
FROM HIERARCHY_SIBLINGS( SOURCE #h_demo START WHERE node_id = 'C2' )
WHERE hierarchy_sibling_distance < 0
ORDER BY hierarchy_sibling_distance DESC;
```

HIERARCHY_RANK	NODE_ID
3	C1

Next following sibling (the sibling to the immediate right) of node B1

```
SELECT TOP 1
    hierarchy_rank,
    node_id
FROM HIERARCHY_SIBLINGS ( SOURCE #h_demo START WHERE node_id = 'B1' )
WHERE hierarchy_sibling_distance > 0
ORDER BY hierarchy_sibling_distance ASC;
```

HIERARCHY_RANK	NODE_ID
5	B2

Last sibling of node B1

```
SELECT TOP 1
```

```
    hierarchy_rank,
    node_id
FROM HIERARCHY_SIBLINGS ( SOURCE #h_demo START WHERE node_id = 'B1' )
ORDER BY hierarchy_sibling_distance DESC;
```

HIERARCHY_RANK	NODE_ID
5	B2

Following nodes

Following nodes are all nodes that have a higher preorder rank than a reference node and are not descendants. This definition is directly translatable to an expression on the hierarchy attributes:

```
SELECT
    h2.hierarchy_rank,
    h2.node_id
FROM #h_demo AS h1, #h_demo AS h2
WHERE h1.node_id = 'B1'
      AND h2.hierarchy_rank >= h1.hierarchy_rank + h1.hierarchy_tree_size
ORDER BY h2.hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
5	B2
6	C3
7	D1
8	D2
9	C4
10	D3

Preceding nodes

Preceding nodes are all nodes that have a lower preorder rank than a reference node and are not ancestors. This definition is directly translatable to an expression on the hierarchy attributes:

```
SELECT
    h2.hierarchy_rank,
    h2.node_id
FROM #h_demo AS h1, #h_demo AS h2
WHERE h1.node_id = 'B2'
      AND h2.hierarchy_rank + h2.hierarchy_tree_size <= h1.hierarchy_rank
ORDER BY h2.hierarchy_rank;
```

HIERARCHY_RANK	NODE_ID
2	B1

HIERARCHY_RANK	NODE_ID
3	C1
4	C2

6 Hierarchy Metadata Views

6.1 HIERARCHY_OBJECTS System View

The HIERARCHY_OBJECTS system view provides a list of all objects that can be used as source objects of hierarchy navigation functions. Objects are identified by their column signature. The exact type of object (table, view, table function, and so on) does not play a role.

This is a valid identification criterion, because any database object with the required hierarchy topology attributes works in navigation functions irrespective of its origin -if the actual attribute values are consistent.

Structure

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	The schema name for the hierarchy.
OBJECT_NAME	NVARCHAR(256)	The name of the hierarchy.
OBJECT_TYPE	VARCHAR(32)	The object type of the hierarchy. Possible values are TABLE, VIEW, or FUNCTION.
OBJECT_OID	BIGINT	The object ID of the hierarchy.

The system view contains only the minimal information to identify hierarchies. For further information, such as their creation time or definition, the HIERARCHY_OBJECTS system view can be joined with other system views. In the following example, the projected columns have not been chosen for utility, but for readability of the result.

```
SELECT
    h.object_name,
    h.object_type,
    COALESCE(v.is_column_view, t.is_column_table) AS is_column_engine,
    COALESCE(v.is_unicode, f.is_unicode) AS is_unicode
FROM
    sys.hierarchy_objects AS h
    LEFT JOIN sys.views AS v
        ON h.object_oid = view_oid
    LEFT JOIN sys.functions AS f
        ON h.object_oid = function_oid
    LEFT JOIN sys.tables AS t
        ON h.object_oid = table_oid
WHERE
    object_name = 'H_DEMO';
```

OBJECT_NAME	OBJECT_TYPE	IS_COLUMN_ENGINE	IS_UNICODE
H_DEMO	VIEW	FALSE	FALSE

7 Comparison to Related SQL Database Concepts

7.1 WITH RECURSIVE Common Table Expressions

Standard SQL:1999 provides WITH RECURSIVE common table expression as a means to express recursive relationships within data sets. While probably a large part of the real-world uses of WITH RECURSIVE can be covered with SAP HANA's hierarchy functions, there are certainly cases where the flexibility offered by common table expressions is required. Whereas common table expressions are a generic low-level tool for recursive computations, SAP HANA's hierarchy functions provide a ready-to-use and much more high-level toolset for typical hierarchical use cases - sacrificing some flexibility in favor of usability and abstraction. It is this higher level of abstraction that potentially allows for better optimization.

7.2 SAP HANA Graph

SAP HANA offers a comprehensive set of features to query and analyze graph structures. Hierarchies and graphs have many similarities. From a mathematical point of view, hierarchies are merely a subset of graphs having a well-formed tree topology. Thus, SAP HANA Graph and hierarchy do overlap, but there are also some notable differences, which will be briefly sketched below to facilitate whether to use SAP HANA hierarchy functions or SAP HANA Graph.

- SAP HANA hierarchy functions focus on tree-like or almost tree-like data topologies. Due to that limited scope, the offered hierarchy functions and algorithms can be optimized easier and can offer an increased performance over generic graph algorithms, but may not be ideal for generic mesh-like topologies.
- SAP HANA hierarchy functions are native and fully integrated SQL extensions. Therefore, all hierarchy input and output parameters can be directly combined with other database objects and can also be dynamically calculated by standard SQL means. As an additional benefit, hierarchies tend to be easier to use without requiring domain-specific graph knowledge.
- SAP HANA graph, on the other hand, provides domain specific languages for graph pattern matching and navigations, which may be more concise and more flexible for customer specific non-standard tasks.

Summing up, as the terms suggest, hierarchies are most suitable to process tree-like structures that are found in relational data, whereas SAP HANA Graphs excels at generic topologies and specific graph applications. Since both feature sets are integral parts of SAP HANA, however, there is no reason not to combine the two and benefit from the sum of the individual strengths.

Related Information

SAP HANA Graph Reference

8 Typical Hierarchy Use Cases and How-To Recipes

The use cases and how-to recipes presented here rely on data from the sections Topologies Used in Examples [page 7] and Separating Hierarchy Model From Navigation [page 40].

8.1 Deriving Further Hierarchy Attributes

Some scenarios require further hierarchical attributes that can be directly derived from the basic attributes.

IS_LEAF node flag

An IS_LEAF node flag can be generated with the following expression:

```
SELECT
    node_id,
    CASE hierarchy_tree_size WHEN 1 THEN 1 ELSE 0 END AS is_leaf
FROM
    HIERARCHY( SOURCE t_demo SIBLING ORDER BY ord )
ORDER BY
    hierarchy_rank;
```

NODE_ID	IS_LEAF
A1	0
B1	0
C1	1
C2	1
B2	0
C3	0
D1	1
D2	1
C4	0
D3	1

IS_ROOT node flag

An IS_ROOT node flag can be generated with the following expression:

```
SELECT
    node_id,
    CASE hierarchy_parent_rank WHEN 0 THEN 1 ELSE 0 END AS is_root
FROM
    HIERARCHY( SOURCE t_demo SIBLING ORDER BY ord )
ORDER BY
    hierarchy_rank;
```

NODE_ID	IS_ROOT
A1	1
B1	0
C1	0
C2	0
B2	0
C3	0
D1	0
D2	0
C4	0
D3	0

DRILL_STATE node attribute

The drill state of a node in a filtered result set (leaf = L, collapsed = C, expanded = E) can be determined by the following expression:

```
SELECT
    node_id,
    CASE
        WHEN hierarchy_tree_size = 1 THEN 'L'
        WHEN LEAD(hierarchy_rank, 1) OVER (ORDER BY hierarchy_rank)
            = hierarchy_rank + hierarchy_tree_size THEN 'C'
        ELSE 'E'
        END AS drill_state
FROM
    HIERARCHY( SOURCE t_demo SIBLING ORDER BY ord )
WHERE
    hierarchy_rank NOT IN (3, 4)
ORDER BY
    hierarchy_rank;
```

NODE_ID	DRILL_STATE
A1	E

NODE_ID	DRILL_STATE
B1	C
B2	E
C3	E
D1	L
D2	L
C4	E
D3	L

Postorder rank

The postorder rank of a node can be determined by the following expression:

```
SELECT
    node_id,
    row_number() OVER(ORDER BY hierarchy_rank + hierarchy_tree_size ASC,
hierarchy_rank DESC) AS postorder_rank
FROM
    HIERARCHY( SOURCE t_demo SIBLING ORDER BY ord )
ORDER BY
    hierarchy_rank;
```

NODE_ID	POSTORDER_RANK
A1	10
B1	3
C1	1
C2	2
B2	9
C3	6
D1	4
D2	5
C4	8
D3	7

Path

Paths between nodes can be calculated as a single concatenated string with the HIERARCHY_ANCESTORS_AGGREGATE Navigation Function [page 59]:

```
WITH h AS ( SELECT * FROM HIERARCHY(
```

```
    SOURCE t_demo
    SIBLING ORDER BY ord ) )
SELECT
    node_id,
    path
FROM HIERARCHY_ANCESTORS_AGGREGATE (
    SOURCE h
    MEASURES ( string_agg(node_id, '/') AS path ) )
ORDER BY
    node_id;
```

NODE_ID	PATH
A1	A1
B1	A1/B1
B2	A1/B2
C1	A1/B1/C1
C2	A1/B1/C2
C3	A1/B2/C3
C4	A1/B2/C4
D1	A1/B2/C3/D1
D2	A1/B2/C3/D2
D3	A1/B2/C4/D3

If ad hoc path calculation time turns out to be a limiting factor, you can cache the calculated paths by stacking two hierarchies on top of each other:

```
WITH h AS ( SELECT * FROM HIERARCHY(
    SOURCE t_demo
    SIBLING ORDER BY ord ) )
SELECT
    id AS node_id,
    path
FROM HIERARCHY (
    SOURCE (
        SELECT
            hierarchy_parent_rank AS parent_id,
            hierarchy_rank AS node_id,
            node_id AS id,
            path
        FROM HIERARCHY_ANCESTORS_AGGREGATE (
            SOURCE h
            MEASURES ( string_agg(node_id, '/') AS path ) ) )
    START WHERE hierarchy_parent_rank = 0
    SIBLING ORDER BY node_id )
ORDER BY
    node_id;
```

NODE_ID	PATH
A1	A1
B1	A1/B1
B2	A1/B2

NODE_ID	PATH
C1	A1/B1/C1
C2	A1/B1/C2
C3	A1/B2/C3
C4	A1/B2/C4
D1	A1/B2/C3/D1
D2	A1/B2/C3/D2
D3	A1/B2/C4/D3

Root

A self join over the HIERARCHY_ROOT_RANK column is the most efficient way to determine the identifiers and other properties of the roots for a given set of nodes. This example is based on the data from table T_DEMO_ERR, because it contains multiple trees:

```
WITH h_err AS ( SELECT * FROM HIERARCHY (
    SOURCE t_demo_err
    SIBLING ORDER BY node_id ) )
SELECT
    h.node_id,
    r.node_id AS root_id,
    r.hierarchy_tree_size AS root_size
FROM
    h_err AS h JOIN h_err AS r
    ON h.hierarchy_root_rank = r.hierarchy_rank
ORDER BY
    h.hierarchy_rank;
```

NODE_ID	ROOT_ID	ROOT_SIZE
E1	E1	5
F1	E1	5
G1	E1	5
F2	E1	5
G1	E1	5
E2	E2	7
F3	E2	7
G2	E2	7
H1	E2	7
F4	E2	7
G2	E2	7
H1	E2	7

NODE_ID	ROOT_ID	ROOT_SIZE
E3	E3	6
F5	E3	6
G3	E3	6
H1	E3	6
H2	E3	6
F5	E3	6

Following node

The identifier (or other attributes) of the following node can be selected using a LEAD window function:

```
SELECT
    node_id,
    LEAD(node_id, hierarchy_tree_size) OVER ( ORDER BY HIERARCHY_RANK ) AS
following_node_id
FROM
    HIERARCHY (
        SOURCE t_demo
        SIBLING ORDER BY ord )
ORDER BY
    hierarchy_rank;
```

NODE_ID	FOLLOWING_NODE_ID
A1	
B1	B2
C1	C2
C2	B2
B2	
C3	C4
D1	D2
D2	C4
C4	
D3	

8.2 Parametrized Hierarchies

Sometimes an application wants to predefine a family of related hierarchies that only differ with respect to one or a few parameters. For example, an application requires a set of hierarchy queries with a fixed start condition,

but a hierarchy revision counter or the current user name needs to be taken into account as a filter condition for the hierarchy source.

One obvious solution for this requirement would be to generate the whole hierarchy generation and navigation as an ad hoc SQL statement on the fly. However, since such large string-based SQL statement generators tend to be error-prone and difficult to debug in a running real-world application, the recommended best practice is to wrap such a family of hierarchies by a parametrized view or table-valued SQLScript function.

The trees from table T_DEMO_ERR are a good example:

```
CREATE VIEW h_demo_pv(
    IN tree INTEGER,
    IN start_id VARCHAR(2) DEFAULT 'E1' )
AS SELECT * FROM HIERARCHY(
    SOURCE (
        SELECT
            parent_id,
            node_id
        FROM
            t_demo_err
        WHERE
            tree = :tree
        ORDER BY
            node_id
    )
    START WHERE
        node_id = :start_id
);
```

```
SELECT
    hierarchy_rank,
    hierarchy_level,
    parent_id,
    node_id
FROM h_demo_pv( tree => 1 );
```

HIERARCHY_RANK	HIERARCHY_LEVEL	PARENT_ID	NODE_ID
1	1		E1
2	2	E1	F1
3	3	F1	G1
4	2	E1	F2
5	3	F2	G1

```
SELECT
    hierarchy_rank,
    hierarchy_level,
    parent_id,
    node_id
FROM h_demo_pv( tree => 2, start_id => 'F3' );
```

HIERARCHY_RANK	HIERARCHY_LEVEL	PARENT_ID	NODE_ID
1	1	E2	F3
2	2	F3	G2

HIERARCHY_RANK	HIERARCHY_LEVEL	PARENT_ID	NODE_ID
3	3	G2	H1

8.3 Working With Unclean Source Data

SAP HANA hierarchies are meant for real-world usage with real-world data. Real-world data is characterized by a certain amount of data records that deviate from the originally intended main structure. The reasons for such deviations vary. Sometimes they are caused by technical malfunctions or by erroneous user input, but sometimes they just reflect an irregular or more manifold reality than initially expected. In many cases, ignoring such deviant data records is not a viable option - it may, for example, lead to incorrect aggregation totals. Technically, such deviant data records may be reflected by certain topological exceptions:

- orphaned nodes or branches
- cycles
- duplications of branches or nodes due to multiple parent edges

Orphans

Orphaned nodes are merely data records of a source data set that are not touched by the initial traversal originating from the start nodes. They are usually not an issue and are expected to occur, for example, if a hierarchy generation starts from a subordinate branch. In this case, they usually can be ignored, which is the default behavior of the HIERARCHY generator function. If the occurrence of an orphan indicates an intolerable corruption of the source data, then the option ORPHAN ERROR is appropriate. However, if an application should be more fault-tolerant and data is aggregated along the hierarchy, ignoring orphans would result in incomplete totals. In other cases, it is required to specifically identify these data records, for example, in a data cleanup run. For these purposes the HIERARCHY generator function provides the orphan handling modes ROOT and ADOPT that alternatively turn them into additional root nodes (ORPHAN ROOT) or collect them under the last root (ORPHAN ADOPT). A typical pattern for the latter case is to provide an artificial orphan collector root, often called rest node, or remainder node that is added to the normal source data by a UNION ALL in the hierarchy source.

Example artificial orphan collector / rest node:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_level AS level,
    hierarchy_is_orphan AS is_orphan,
    parent_id,
    node_id
FROM HIERARCHY (
    SOURCE (
        SELECT node_id, parent_id FROM (
            SELECT 1 AS src_ord, node_id, parent_id, ord FROM t_demo
            UNION ALL
            SELECT 2 AS src_ord, 'SP' AS node_id, null AS parent_id, 1 AS ord
FROM dummy
        )
        ORDER BY
```

```
            src_ord, ord
    )
    START WHERE node_id IN ( 'B2', 'SP' )
    ORPHAN ADOPT )
ORDER BY
    hierarchy_rank;
```

RANK	LEVEL	IS_ORPHAN	PARENT_ID	NODE_ID
1	1	0	A1	B2
2	2	0	B2	C3
3	3	0	C3	D1
4	3	0	C3	D2
5	2	0	B2	C4
6	3	0	C4	D3
7	1	0		SP
8	2	1		A1
9	3	1	A1	B1
10	4	1	B1	C1
11	4	1	B1	C2

Cycles

Since an unrestricted traversal of cycles in hierarchy source data would quickly exhaust any system resources due to endless recursion, the HIERARCHY generator function always breaks them up after the first re-entrance. If a hierarchy-consuming application wants to reconstruct the original topology, it has to query the ancestors of all nodes having a HIERARCHY_IS_CYCLE flag until finding a matching NODE_ID.

Duplications of Branches or Nodes due to Multiple Parent Edges

This situation occurs when the source data contains multiple records with the same NODE_ID value. The PARENT_ID values are not required to be different. To transform such a topology into a well-formed tree, all descendant nodes of the nodes with multiple parents must be duplicated for each parent.

If nodes with multiple parents are defects and only a few exist, this is usually not a problem. However, if the data normally contains nodes with multiple parents and they appear frequently, this can cause major problems.

The multiplication of nodes leads to an increased memory consumption and an increased runtime. Certain topologies cause an exponential growth of the number of nodes, easily exhausting the memory of every possible system.

Hierarchy functions offer two ways to prevent an unchecked increase in the node number: The DISTANCE specification limits the maximum distance from the start node that can be reached. The

HIERARCHY_SPANTREE function builds a minimal spanning tree, meaning each NODE_ID value is only reached once during traversal and all further edges leading to that node are ignored.

8.4 Handling Generic Graph Topologies

If the source data topology substantially deviates from the ideal case of disjoint well-formed pure trees, hierarchies may nevertheless provide an option for effective query processing:

- If the source data set is small, a full-scale tree expansion with many duplicate branches may still be processed in an acceptable amount of time. In order to assess the effectiveness, you have to inspect the result size of the hierarchy generation. If the hierarchy generation results consist of no more than a few million records, it can usually be processed fast enough.
- A simple and effective means to reduce the overall hierarchy generator function result size is to limit the traversal depth by using the optional DEPTH horizon parameter (given that the application knows beforehand that a certain maximum search depth will be sufficient). In this case and also generally, if queries are just interested in the near neighborhood of a few known nodes, it is usually more efficient to avoid full complete hierarchy models and execute limited ad hoc queries instead.
- The HIERARCHY_SPANTREE Generator Function [page 21] has been specifically designed for connectivity tests and the determination of the shortest paths between nodes in generic graph topologies.

Example

You aggregate data based on a limited local ad hoc query instead of a full hierarchy:

```
SELECT
    SUM(amount)
FROM HIERARCHY (
    SOURCE ( SELECT node_id, parent_id, amount FROM t_demo )
    START WHERE node_id = 'B2'
    DEPTH 2
    ORPHAN IGNORE );
```

SUM(AMOUNT)
13

A less efficient aggregation based on a partial tree of a complete hierarchy:

```
WITH
   h AS ( SELECT * FROM HIERARCHY (
       SOURCE ( SELECT node_id, parent_id, amount FROM t_demo ) ) )
SELECT
    SUM(amount)
FROM
    HIERARCHY_DESCENDANTS( SOURCE h START WHERE node_id = 'B2' );
```

SUM(AMOUNT)
13

8.5 Handling Source Topology Quirks

Finding Topology Quirks in Hierarchies

SAP HANA 2.0 hierarchy generator functions allow easy identification of orphans and cycles due to the HIERARCHY_IS_ORPHAN and HIERARCHY_IS_CYCLE flags. If preferred, source data with orphans or cycles lead to a runtime error.

Having nodes with multiple parents is identical to the multiple occurrences of a NODE_ID value.

The following query lists all nodes that have been multiplied due to multiple parent nodes in the hierarchy H_DEMO_ERR:

```
SELECT
    hierarchy_rank,
    parent_id,
    h.node_id
FROM (
    SELECT node_id FROM h_demo_err GROUP BY node_id HAVING COUNT(node_id) > 1
    ) AS c JOIN h_demo_err AS h
ON c.node_id=h.node_id;
```

HIERARCHY_RANK	PARENT_ID	NODE_ID
3	F1	G1
5	F2	G1
8	F3	G2
9	G2	H1
11	F4	G2
12	G2	H1
14	E3	F5
16	G3	H1
18	H2	F5

In a generated hierarchy, this statement provides all nodes that get multiplied due to the treeification of nodes with multiple parents. Not just the nodes that actually have multiple parent nodes, but also their children. It also includes nodes that close cycles, but these can easily be removed by a filter on the HIERARCHY_IS_CYCLE attribute.

The best method to enforce a single parent policy is to define a unique constraint on the NODE_ID column of the corresponding hierarchy source table. If that is not an option, the MULTIPARENT specification can be used.

Finding Topology Quirks in the Source Data

In certain cases, it might be preferred to identify topology quirks in the source data. This is not as simple as identifying them in the final hierarchy, but since it may be impossible to load very large hierarchies that include many nodes with multiple parents, finding topology quirks in the source data can be extremely helpful.

Since duplicate edges or nodes are never removed from a hierarchy, the total number of edges is equivalent to the number of NODE_ID values that are not NULL:

```
SELECT COUNT(node_id) AS num_edges FROM t_demo_err;
```

NUM_EDGES
18

If root nodes are identified by the default condition `WHERE parent_id IS NULL` , the number of root nodes can be determined as follows:

```
SELECT COUNT(COALESCE(parent_id,'')) AS num_roots FROM t_demo_err WHERE
parent_id IS NULL;
```

NUM_ROOTS
3

Orphaned top level nodes can be determined by:

```
SELECT
    parent_id,
    node_id
FROM t_demo_err
    WHERE node_id iS NOT NULL AND parent_id IS NOT NULL
        AND parent_id NOT IN (SELECT node_id FROM t_demo_err);
```

PARENT_ID	NODE_ID
E4	F6

The adaption of both queries to arbitrary START WHERE clauses is straightforward.

The following statement calculates the number of leaf nodes:

```
SELECT COUNT(*) AS num_leafs FROM t_demo_err WHERE node_id NOT IN (SELECT
parent_id FROM t_demo_err WHERE parent_id IS NOT NULL);
```

NUM_LEAFS
5

The average number of children is given by the number of edges, which are not root nodes or top level orphans, divided by the number of non-leaf nodes:

```
<avgNumChildern> = (<numEdges>-<numRoots>-<numOrphans>)
                    / (<numEdges>-<numLeaves>)
```

Nodes with multiple parent nodes can be listed by the following statement, as for already generated hierarchies. However, if the statement is executed on the source data, only nodes that actually have multiple parent nodes are displayed, their children are not. It is also not possible to remove nodes closing a cycle.

```
SELECT
    parent_id,
    s.node_id
FROM (
    SELECT node_id FROM t_demo_err GROUP BY node_id HAVING COUNT(node_id) > 1
    ) AS c JOIN t_demo_err AS s
ON c.node_id=s.node_id;
```

PARENT_ID	NODE_ID
E3	F5
F1	G1
F2	G1
F3	G2
F4	G2
G2	H1
G3	H1
H2	F5

The following SQLScript procedure analyses hierarchy source data:

```
CREATE PROCEDURE analyse_hier_src(
    IN source TABLE(parent_id VARCHAR(5000), node_id VARCHAR(5000)), IN
rootNodeId VARCHAR(5000) DEFAULT NULL,
    OUT info TABLE(key VARCHAR(64), value DECIMAL, parent_id VARCHAR(5000),
node_id VARCHAR(5000)))
AS
BEGIN
    DECLARE general TABLE(key VARCHAR(64), value DECIMAL);
    DECLARE multiparents TABLE(parent_id VARCHAR(5000), node_id VARCHAR(5000));
    DECLARE orphans TABLE(parent_id VARCHAR(5000), node_id VARCHAR(5000));
    DECLARE numEdges DECIMAL;
    DECLARE avNumChil DECIMAL;
    DECLARE numRoots DECIMAL;
    DECLARE numLeaves DECIMAL;
    DECLARE numMultiparents DECIMAL;
    DECLARE numOrphans DECIMAL;
    IF rootNodeId IS NULL THEN
        orphans = SELECT parent_id, node_id FROM :source
            WHERE node_id IS NOT NULL AND parent_id IS NOT NULL
                AND parent_id NOT IN (SELECT node_id FROM :source);
        SELECT COUNT(COALESCE(parent_id,'')) INTO numRoots FROM :source WHERE
parent_id IS NULL;
    ELSE
        orphans = SELECT parent_id, node_id FROM :source
            WHERE node_id IS NOT NULL AND parent_id<>:rootNodeId
                AND parent_id NOT IN (SELECT node_id FROM :source);
        SELECT COUNT(parent_id) INTO numRoots FROM :source WHERE
parent_id=:rootNodeId;
    END IF;
    multiparents =
        SELECT parent_id, s.node_id
        FROM (
            SELECT node_id FROM :source GROUP BY node_id HAVING COUNT(node_id) >
1
```

```
            ) AS c JOIN :source AS s
        ON c.node_id=s.node_id;
    SELECT COUNT(node_id) INTO numEdges FROM :source;
    SELECT COUNT(*) INTO numMultiparents FROM :multiparents;
    SELECT COUNT(*) INTO numOrphans FROM :orphans;
    SELECT COUNT(*) INTO numLeaves FROM :source WHERE node_id NOT IN (SELECT
parent_id FROM :source);
    avNumChil = (numEdges-numRoots-numOrphans)/(numEdges-numLeaves);
    general.key[1]='number of unique edges';
    general.value[1]=numEdges;
    general.key[2]='number of root nodes';
    general.value[2]=numRoots;
    general.key[3]='average number of children';
    general.value[3]=round(avNumChil,3);
    general.key[4]='number of leaf nodes';
    general.value[4]=numLeaves;
    general.key[5]='number of multiparent edges';
    general.value[5]=numMultiparents;
    general.key[6]='number of orphaned top level nodes';
    general.value[6]=numOrphans;
    info =
        SELECT key, value, parent_id, node_id FROM (
            SELECT *, NULL AS parent_id, NULL AS node_id, 0 as ord FROM :general
            UNION ALL
            SELECT
                'multiparent edge' AS key,
                NULL AS value,
                parent_id,
                node_id,
                1 as ord
            FROM :multiparents
            UNION ALL
            SELECT
                'orphaned edge' AS key,
                NULL AS value,
                parent_id,
                node_id,
                2 as ord
            FROM :orphans
        )
        ORDER BY ord;
END;
```

```
CALL analyse_hier_src(t_demo_err, info=>?);
```

KEY	VALUE	PARENT_ID	NODE_ID
number of unique edges	18		
number of root nodes	3		
average number of children	0.778		
number of leaf nodes	0		
number of multiparent edges	8		
number of orphaned top-level nodes	1		
multiparent edge		F1	G1
multiparent edge		F2	G1
multiparent edge		F4	G2

KEY	VALUE	PARENT_ID	NODE_ID
multiparent edge		F3	G2
multiparent edge		G2	H1
multiparent edge		G3	H1
multiparent edge		E3	F5
multiparent edge		H2	F5
orphaned edge		E4	F6

8.6 Working With Composite Node Identifiers

The correct processing of composite node identifiers requires that the individual parent and node identifier components are projected in a concatenated form in the source specification and aliased to PARENT_ID / NODE_ID. Instead of dissecting the concatenated input in the generator function output, it is usually more convenient to project the individual components as well:

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_level AS level,
    parent_id,
    node_id,
    parent_1,
    parent_2,
    id_1,
    id_2
FROM HIERARCHY ( SOURCE (
    SELECT
        id_1||id_2 AS node_id,
        parent_1||parent_2 AS parent_id,
        id_1,
        id_2,
        parent_1,
        parent_2
    FROM
        t_demo_composite
    ORDER BY
        ord )
   START WHERE
        parent_1 = 'X' )
ORDER BY
    hierarchy_rank;
```

RANK	LEVEL	PARENT_ID	NODE_ID	PARENT_1	PARENT_2	ID_1	ID_2
1	1	X1	Y1	X	1	Y	1
2	1	X1	Y2	X	1	Y	2

To generate composite node identifiers that are guaranteed to be free of name clashes, you can use the auxiliary HIERARCHY_COMPOSITE_ID Scalar Function [page 36] instead of manual string concatenation:

```
SELECT
    hierarchy_rank AS rank,
```

```
    hierarchy_level AS level,
    parent_id,
    node_id,
    parent_1,
    parent_2,
    id_1,
    id_2
FROM HIERARCHY ( SOURCE (
    SELECT
        HIERARCHY_COMPOSITE_ID(id_1, id_2) AS node_id,
        HIERARCHY_COMPOSITE_ID(parent_1, parent_2) AS parent_id,
        id_1,
        id_2,
        parent_1,
        parent_2
    FROM
        t_demo_composite
    ORDER BY
        ord )
   START WHERE
        parent_1 = 'X' )
ORDER BY
    hierarchy_rank;
```

RANK	LEVEL	PARENT_ID	NODE_ID	PARENT_1	PARENT_2	ID_1	ID_2
1	1	1,X;1,1	1,Y;1,1	X	1	Y	1
2	1	1,X;1,1	1,Y;1,2	X	1	Y	2

8.7 Connectivity Tests

Connectivity testing between two sets of nodes, following edges in both directions, is best done directly on the hierarchy attributes without using navigation functions:

```
SELECT DISTINCT
    h1.node_id AS node_set_1,
    h2.node_id AS node_set_2
FROM
    #h_demo AS h1, #h_demo AS h2
WHERE
    h1.node_id IN ( SELECT DISTINCT node_id FROM t_demo WHERE node_id LIKE
'B%' ) -- node_set_1
    AND h2.node_id IN ( SELECT node_id FROM t_demo WHERE node_id LIKE 'D%' ) --
node_set_2
    AND ( h2.hierarchy_rank BETWEEN h1.hierarchy_rank AND h1.hierarchy_rank +
h1.hierarchy_tree_size - 1
        OR h1.hierarchy_rank BETWEEN h2.hierarchy_rank AND h2.hierarchy_rank +
h2.hierarchy_tree_size - 1 )
ORDER BY
    node_set_1, node_set_2;
```

NODE_SET_1	NODE_SET_2
B2	D1
B2	D2

NODE_SET_1	NODE_SET_2
B2	D3

If the general direction on how to navigate from node set 1 to node set 2 is known, it is syntactically more concise and also better from a performance point of view to use the navigation functions HIERARCHY_DESCENDANTS (downward) / HIERARCHY_ANCESTORS (upward) for connectivity testing:

```
SELECT DISTINCT
    start_id,
    node_id
FROM
    HIERARCHY_DESCENDANTS ( -- or HIERARCHY_ANCESTORS
        SOURCE #h_demo
        START ( SELECT hierarchy_rank AS start_rank, node_id AS start_id FROM
#h_demo WHERE node_id LIKE 'B%' ) -- node_set_1
    )
WHERE
    node_id LIKE 'D%' -- node_set_2
ORDER BY
    start_id, node_id;
```

START_ID	NODE_ID
B2	D1
B2	D2
B2	D3

8.8 Determining the Nearest Common Ancestor of Two Known Nodes

A node is a common ancestor of two other nodes if both other nodes are contained in its subtree. That is, the HIEARRCHY_RANK of the two other nodes is in the interval [HIERARCHY_RANK, HIERARCHY_RANK + HIERARCHY_TREE_SIZE-1] of the ancestor node.

Among all common ancestor nodes, the one with the highest HIERARCHY_RANK is the nearest common ancestor.

```
WITH
    h AS ( SELECT * FROM HIERARCHY_SPANTREE (
        SOURCE ( SELECT
            node_id,
            parent_id
        FROM
            t_demo
        ORDER BY
            ord )
        START WHERE node_id = 'A1' )
    ),
    vals AS ( SELECT hierarchy_rank AS nrank, node_id AS id FROM h WHERE node_id
IN ( 'D1', 'D3' ) )
SELECT TOP 1
    hierarchy_rank,
    hierarchy_level,
```

```
    node_id
FROM
    h
WHERE
    hierarchy_rank <= (SELECT MIN(nrank) FROM vals)
    AND hierarchy_rank + hierarchy_tree_size > (SELECT MAX(nrank) FROM vals)
ORDER BY
    hierarchy_rank DESC;
```

HIERARCHY_RANK	HIERARCHY_LEVEL	NODE_ID
5	2	B2

8.9 Creating Synthetic Hierarchical Test Data

The most convenient way to quickly generate a balanced hierarchy of arbitrary depth and size is using the SAP HANA SERIES_GENERATE function family as a data source. The following example creates a hierarchy consisting of 100 nodes where each branch node has 4 children:

```
SELECT TOP 10
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    hierarchy_is_cycle AS is_cycle,
    hierarchy_is_orphan AS is_orphan,
    node_id,
    parent_id
FROM
    HIERARCHY (
        SOURCE (
            SELECT
                TO_INT(element_number) AS node_id,
                CASE WHEN element_number <=4 THEN null ELSE
TO_INT((element_number-1)/4) END AS parent_id
            FROM
                SERIES_GENERATE_INTEGER(1,1,101)
        )
        SIBLING ORDER BY node_id
    )
ORDER BY
    hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	IS_CYCLE	IS_ORPHAN	NODE_ID	PARENT_ID
1	37	0	1	0	0	1	
2	21	1	2	0	0	5	1
3	5	2	3	0	0	21	5
4	1	3	4	0	0	85	21
5	1	3	4	0	0	86	21
6	1	3	4	0	0	87	21

RANK	TREE_SIZE	PA-RENT_RANK	LEVEL	IS_CYCLE	IS_ORPHAN	NODE_ID	PARENT_ID
7	1	3	4	0	0	88	21
8	5	2	3	0	0	22	5
9	1	8	4	0	0	89	22
10	1	8	4	0	0	90	22

8.10 Creating Date/Time Hierarchies

SAP HANA series functions serve as a powerful basis to create ad hoc date or time hierarchies of arbitrary size by just a single select statement. The HIERARCHY_LEVELED generator function provides an intuitive means to define the hierarchy's granularity:

```
SELECT TOP 10
    hierarchy_rank AS rank,
    hierarchy_level AS level,
    node_id,
    parent_id,
    HIERARCHY_LEVEL_NAME AS level_name,
    year,
    quarter,
    month,
    day
FROM
    HIERARCHY_LEVELED (
        SOURCE (
            SELECT
                YEAR(generated_period_start) AS year,
                SUBSTR_AFTER(QUARTER(generated_period_start),'-') AS quarter,
                MONTHNAME(generated_period_start) AS month,
                DAYOFMONTH(generated_period_start) AS day
            FROM
                SERIES_GENERATE_DATE('INTERVAL 1 DAY', '2016-01-01',
'2018-01-01')
        )
        SIBLING ORDER BY day
    )
ORDER BY
    hierarchy_rank;
```

RANK	LEVEL	NODE_ID	PARENT_ID	LEVEL_NAME	YEAR	QUARTER	MONTH	DAY
1	1	2016		YEAR	2016			
2	2	Q1	2016	QUARTER	2016	Q1		
3	3	FEBRUARY	Q1	MONTH	2016	Q1	FEBRUARY	
4	4	1	FEBRUARY	DAY	2016	Q1	FEBRUARY	1
5	4	2	FEBRUARY	DAY	2016	Q1	FEBRUARY	2
6	4	3	FEBRUARY	DAY	2016	Q1	FEBRUARY	3

RANK	LEVEL	NODE_ID	PARENT_ID	LEVEL_NAME	YEAR	QUARTER	MONTH	DAY
7	4	4	FEBRUARY	DAY	2016	Q1	FEBRUARY	4
8	4	5	FEBRUARY	DAY	2016	Q1	FEBRUARY	5
9	4	6	FEBRUARY	DAY	2016	Q1	FEBRUARY	6
10	4	7	FEBRUARY	DAY	2016	Q1	FEBRUARY	7

8.11 UI-driven Interactive Data Drill-Down

The backend functions for a simple, stateless, interactive UI with drill-down and drill-up operations can be implemented as follows. In addition to the hierarchy source data, the database must store the state of the displayed hierarchy. To achieve this in SAP HANA, global temporary tables can be used. This automatically isolates different users from each other, as long as they are not sharing the same database connection.

Initial setup

Indexed hierarchy

This table contains the view h_demo from section Separating Hierarchy Model From Navigation [page 40].

```
CREATE GLOBAL TEMPORARY COLUMN TABLE hier AS (
    SELECT * FROM h_demo
);
```

Currently displayed result

This table is initialized with all roots.

```
CREATE GLOBAL TEMPORARY COLUMN TABLE result AS (
    SELECT *
    FROM HIERARCHY_DESCENDANTS (
        SOURCE h_demo
        START WHERE hierarchy_parent_rank = 0
        DISTANCE 0
    )
);
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id
FROM result;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID
1	10	0	1		A1

Temporary for intermediate results

This table is initially empty.

```
CREATE GLOBAL TEMPORARY COLUMN TABLE temp_store LIKE result;
```

Interactive Drill-Down and Drill-Up

The user can then execute drill-down and drill-up operations on the displayed hierarchy. For a drill-down operation on a given node (identified by its hierarchy_rank, in this example 1), the result table has to be altered as follows:

```
INSERT INTO result
    SELECT *
    FROM HIERARCHY_DESCENDANTS(
        SOURCE h_demo
        START (
            SELECT hierarchy_rank AS start_rank
            FROM result
            WHERE hierarchy_rank = 1
        )
        DISTANCE 1
    );
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id
FROM result;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID
1	10	0	1		A1
2	3	1	2	A1	B1
5	6	1	2	A1	B2

Selecting the start nodes from the result instead of from the complete hierarchy makes the system a bit more robust, as only visible nodes can be drilled.

To drill-up from a given node, records have to be deleted from the result table. Since deleting is not possible for temporary tables in SAP HANA, the result has to be copied:

```
INSERT INTO temp_store
    SELECT * FROM result ORDER BY hierarchy_rank;
TRUNCATE TABLE result;
INSERT INTO result
    SELECT * FROM temp_store WHERE hierarchy_parent_rank <> 1;
TRUNCATE TABLE temp_store;
```

After each operation, the result can be retrieved.

```
SELECT
    hierarchy_rank AS rank,
    hierarchy_tree_size AS tree_size,
    hierarchy_parent_rank AS parent_rank,
    hierarchy_level AS level,
    parent_id,
    node_id
FROM result ORDER BY hierarchy_rank;
```

RANK	TREE_SIZE	PARENT_RANK	LEVEL	PARENT_ID	NODE_ID
1	10	0	1		A1

The use of the * symbol in the projection list is for brevity only. In the current example, the columns HIERARCHY_DISTANCE and START_RANK do not contain any information and should be removed.

8.12 Visualizing Hierarchies

Currently SAP HANA does not offer a graphical hierarchy viewer. To get a visual representation of a hierarchy, you need external tools, for example, the open source graphviz package.

The following query creates a graphviz source file from a hierarchy generation function call:

```
WITH
    h AS ( SELECT * FROM h_demo )
SELECT
    graphviz_code
FROM (
    SELECT
        'digraph G { node [shape=oval fontname="Arial"];' AS graphviz_code,
        0 AS ord
    FROM dummy
    UNION ALL
    SELECT
        CASE WHEN
            hierarchy_parent_rank = 0
        THEN
            hierarchy_rank ||' [label="'|| node_id || '"];'
        ELSE
            hierarchy_parent_rank ||' -> '|| hierarchy_rank ||'; ' ||
hierarchy_rank ||' [label="'|| node_id || '"];'
        END AS graphviz_code,
        CASE WHEN
            hierarchy_parent_rank = 0
        THEN 1
        ELSE 2 END AS ord
    FROM
        h
    UNION ALL
    SELECT
        '}' AS graphviz_code,
        3 AS ord
    FROM dummy )
ORDER BY ord, graphviz_code;
```

GRAPHVIZ_CODE
digraph G { node [shape=oval fontname="Arial"];
1 [label="A1"];
1 -> 2; 2 [label="B1"];
1 -> 5; 5 [label="B2"];
2 -> 3; 3 [label="C1"];
2 -> 4; 4 [label="C2"];
5 -> 6; 6 [label="C3"];
5 -> 9; 9 [label="C4"];
6 -> 7; 7 [label="D1"];
6 -> 8; 8 [label="D2"];
9 -> 10; 10 [label="D3"];
}

Related Information

graphviz is freely available at http://graphviz.org

9 Migrating From Other RDBM Systems

For information about migrating hierarchy structures to SAP HANA, see SAP Note 2446997 - Migrating Hierarchy Structures to SAP HANA.

Related Information

SAP Note 2446997

10 Performance Recommendations

- Prefer the HIERARCHY_SPANTREE Generator Function [page 21] over the HIERARCHY Generator Function [page 11] for pure connectivity tests or the determination of a shortest path between nodes, if the hierarchy is not a well-formed tree.
- Restrict the projection list of the hierarchy source object to the minimum required set of attributes. If for some cases additional attributes are needed, check whether a join of the hierarchy result with its source object is a viable alternative providing better overall performance.
- Use static preconditions (WHERE filters) on the generator function source data whenever applicable.
- If a composite key component is always the same value for both parent and node identifier, prefer a static precondition over a composite hierarchy node.
- Prefer hierarchy function parameters (preconditions) over WHERE conditions (post-filters) on the hierarchy function result.
- If caching does not result in a sufficient performance boost, consider whether an intermediate materialization of the hierarchy generation result into a temporary table could be viable. While caching avoids the overhead of calculating the hierarchy attributes as long as the source data stays unchanged, an intermediate materialization allows to trade memory consumption and data currentness for performance.
- To obtain stable results for hierarchy generation functions, the order of siblings in the source data must be stable. The easiest way to achieve this is usually to order all siblings by their node ID, i.e. SIBLING ORDER BY node_id. However, the performance of this sort operation depends heavily on the data type of the node_id column and on the actual values. If the actual sibling order is not important as long as it is stable, it may be beneficial to use a different column for sorting or even generate an integer type column just for that purpose.

11 Troubleshooting

- Generally, SAP HANA hierarchy functions have a very transparent step-by-step computation model. If the output of a hierarchy function call does not match your expectations, it is recommended to dissect the calculation into its constituting atomic operations: source query - determination of source start nodes - hierarchy generation - determination of navigation start nodes - navigation result and inspect them individually for their plausibility. It is therefore often helpful to use a radically restricted source data set and temporarily materialize the intermediate results into temporary tables, thereby reducing the complexity of the overall computation.
- If a hierarchy navigation function returns deterministic inconsistent nonsense results, check whether a hierarchy navigation gets an unfiltered hierarchy generation function result as a source input.
- If a hierarchy navigation function returns inconsistent and sometimes also non-deterministic results (in particular if the hierarchy is large), check if the underlying hierarchy generation function source specification has a deterministic stable source sort order - at least for sibling nodes. This is a prerequisite for the correct operation of navigation functions, which require that the hierarchy rank values specified in the START / START WHERE clause exactly match the hierarchy rank values defined in the SOURCE clause.

> **i Note**
>
> Note that the following warning only indicates the complete absence of a SIBLING ORDER BY clause:
>
> ```
> Warning 654:
> no row order on table set:
> hierarchy generation function result is partially non-deterministic due to
> missing SIBLING ORDER BY clause or missing ORDER BY clause in SOURCE
> specification.
> ```
>
> There is unfortunately no technical mechanism available that allows the generation function to detect an incomplete sibling sort order that may lead to unstable and non-deterministic navigation function results as well.

12 Important Disclaimer for Features in SAP HANA

For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon  : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon  : You are leaving the documentation for that particular SAP product or service and are entering an SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

THE BEST RUN SAP