HINTA: A Linearization Algorithm for Physical Clustering of Complex OLAP Hierachies

Roland Pieringer
TransActon Software GmbH
Thomas-Dehler-Str. 18,
81737 München, Germany
pieringer@transaction.de

Volker Markl
IBM Almaden Research Center K55/B1,
650 Harry Road,
San Jose, CA 95120-6099
marklv@us.ibm.com

Frank Ramsak
Bayerisches Forschungsinstitut für
wissensbasierte Systeme
Orleansstr. 34, 81667 München
frank.ramsak@forwiss.de

Rudolf Bayer
Institut für Informatik
Technische Universität München,
Orleansstr. 34, 81667 München
bayer@in.tum.de

Abstract

Hierarchies are an important means to categorize data stored in OLAP systems. OLAP queries follow the drill/slice/dice-paradigm and therefore exhibit navigation patterns that follow the hierarchy of a dimension. In real-world applications, hierarchies are often unbalanced and share levels, resulting in complex hierarchy structures. So far, encoding methods for simple structured hierarchies have been introduced to handle hierarchies efficiently for query processing. In this paper we propose the HINTA algorithm to compute the clustering order for complex hierarchies by linearization. The physical clustering of OLAP data computed by HINTA significantly improves the performance of OLAP queries. HINTA enables clustering of complex hierarchies that can share hierarchy levels in several classifications over one dimension.

1 Introduction

A data warehouse (DW) is a physical database with an integrated view onto arbitrary data. A multidimensional (MD) view enables complex interactive, explorative data analysis (OLAP, i.e. OnLine Analytical Processing).
Conceptually, the data of a DW is stored in data cubes. A data cube consists of a set of dimensions and a set of measures. Dimensions provide categorical (qualitative) data (e.g., products, customers, time), which determine the context of the measures (e.g., items sold, cost, turnover).

Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'2001)
Interlaken, Switzerland, June 4, 2001
(D. Theodoratos, J. Hammer, M. Jeusfeld, M. Staudt, eds.)
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-39/

The set of base values forming a dimension generally is classified according to a set of hierarchies. For instance, the time dimension may have a hierarchy *all-year-month-day* or *all-year-week-day*. In this paper we will discuss and further detail how the set of hierarchies can be represented and efficiently utilized for query processing.
Multidimensional clustering indexes (e.g., UB-Tree, R-Tree) handle multiple dimensions for multidimensional range queries ([Mar99]). Encoding methods prepare hierarchical classification for the use of clustering B-Trees for one hierarchy ([ZSL98], [MRB99]). This encoding, however, is only useful for a special case of hierarchies, i.e., hierarchy trees or simple hierarchies. In reality, hierarchies are more complex, e.g., hierarchies are unbalanced, have alternative paths and shared levels. To solve this severe problem and make encoding techniques useful for real world scenarios, we propose HINTA, an algorithm that transforms an instantiation of a complex hierarchy to a hierarchy tree. In combination with the above mentioned encoding schemes, the resulting hierarchy can be used for clustering.
In this paper, we present a formal hierarchy model, that is based on graph algorithms and is introduced by the instantiation of the hierarchies.
The rest of the paper is organized as follows. Section 2 lists related work. Section 3 gives a motivating example how to use hierarchy encoding and to make use of HINTA. In Section 4, we present the hierarchy model. Section 5 describes HINTA, a transformation algorithm of complex hierarchies to simple hierarchies. Section 6 summarizes this paper and gives an outlook to future work.

2 Related Work

In the DW community, some formal models of DW, dimensions, hierarchies etc. already have been worked out. Some approaches do not explicitly include hierarchical classification in their data model ([AGS97], [BPT97]). In [Sap01], [Leh98a] and [Alb01], the authors work out a hierarchical classification, defining hierarchy schemata

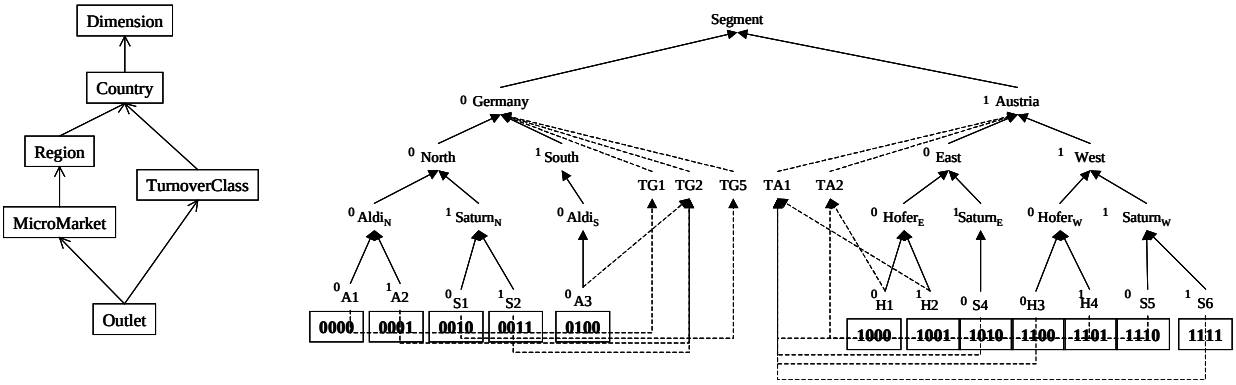

Figure 3-1: Hierarchy with Encoding

with classify-relationships. In [LW96], a MD model is discussed, based on relational elements.

Many publications propose first to establish the conceptual model and then to do the actual implementation ([WB97], [CT98], [GMR98]). [HLV00] show how to systematically derive a conceptual warehouse schema from a generalized multidimensional normal form.

[FS99] introduce a conceptual data model, that allows complex descriptions of the structure of aggregated entities and multiply hierarchically organized dimensions. [VS99] presents an overview of the understanding of commercial and scientific concepts of DW modeling.

For single hierarchies, [ZSL98] discusses the linearization and presents the physical representation within DBMS. [MRB99] extend the linearization to multiple dimensions and hierarchies and discuss query processing of hierarchically organized multidimensional data.

In this paper, we further present a linearization method for complex hierarchies by transforming complex hierarchies to simple hierarchies and using the linearization method already published in [MRB99].

[PJD99] discuss a transformation algorithm to achieve summarizability on unbalanced hierarchies.

3 Motivation

In a star schema ([Kim96]), dimension tables are connected to a large fact table via dimension attributes (join attributes). The dimension table usually contains the hierarchies of the dimension, where for every path through the hierarchy an artificial unique id (*dimID*) is used as join attribute. This dimID can be a computed number with respect to the encoding of the hierarchy for hierarchical clustering: dimID=surr(v_m, v_{m-1}, …, v_{leaf}). The function *surr* computes a surrogate id for the path of the dimension tuple. The schema of a dimension table usually includes the hierarchy attributes of all simple hierarchies.

Conventional approaches to process queries in DW schemata in relational DBMS are star join algorithms, where restrictions on the dimension tables result in a number of dimension values that are joined with the fact table. Queries that restrict dimensions, have predicates on hierarchy levels. These predicates usually are point or interval restrictions ([Sar97]) and result in large point sets on base granularity (i.e., the leaf level of the hierarchy). Such point sets can be replaced by a smaller set of interval restrictions depending on the predicate. The predicate "Germany" of the hierarchy in Figure 3-1 would result in the leaf members {"A1", "A2", "S1", "S2", "A3"}, and

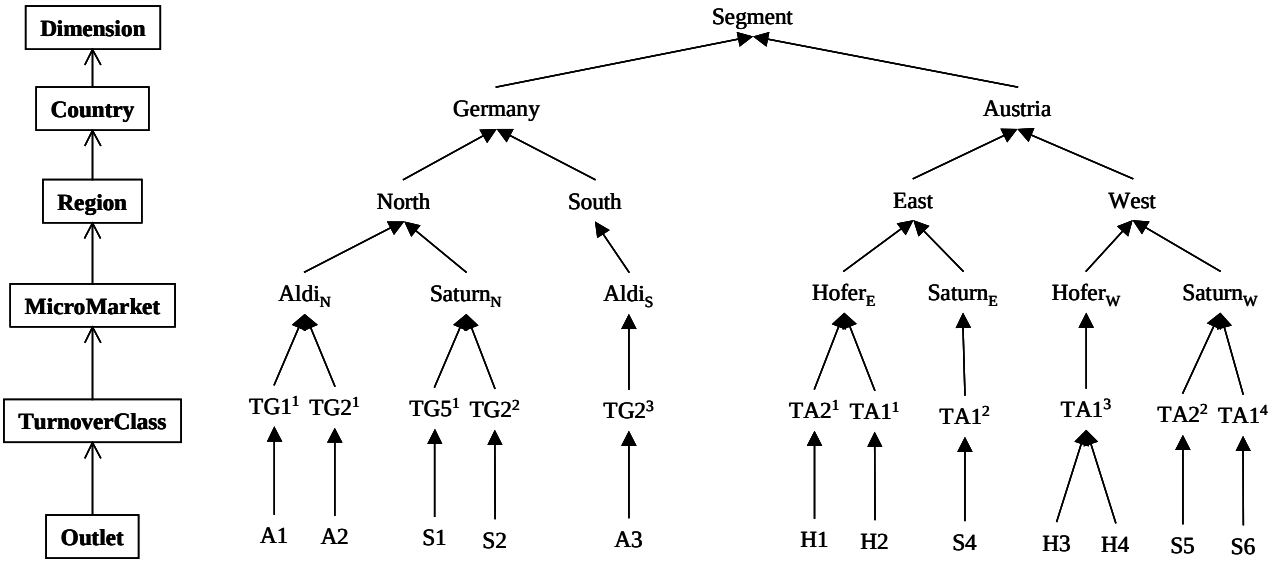

Figure 3-2: Transformed Hierarchy

every such member is a join predicate to the fact table.
Figure 3-1 shows a hierarchy schema (on the left) and one hierarchy instance (on the right). The hierarchy is a complex hierarchy with the paths Dimension-Country-Region-MicroMarket-Outlet (solid arrows) or alternatively Dimension-Country-TurnoverClass-Outlet (dashed arrows).

3.1 Hierarchy Encoding

The identifier of the paths must be unique. Thus, a number can be used to represent the corresponding path in the hierarchy. We establish an encoding schema on the hierarchy, that numbers (*surrogate* number) the children of every level. The resulting identifier, called *compound surrogate*, are the concatenated surrogates of the path, one for each level. It is shown in Figure 3-1 in the rectangles.
With this encoding ([ZSL98], [MRB99]), hierarchical point sets can be replaced by intervals. The predicate "Germany", is mapped to the interval [000; 0100]. This new interval predicate speeds up query execution on the fact table, when using corresponding clustering indexes (because a local interval predicate can be performed on the fact table instead of a join). Such an encoding is known for simple hierarchies. But predicates on a complex hierarchy often result in point restrictions on the leaf members. The predicate "TG2" specifies the leaf members {"A2", "S2", "A3"}, that cannot be expressed by an interval when encoding the hierarchy with respect to the previous case.
A solution to speed up queries for DW applications with complex hierarchies is to transform the complex hierarchy into a simple hierarchy while leaving hierarchical dependencies. With this transformation and the mentioned encoding, a predicate on the dimension hierarchy can be mapped to a relatively small number of intervals on the fact table. Thus, a query with a number of intervals on the fact table is performed instead of a complex join operation between dimension and fact table.
HINTA changes the hierarchy from a complex to a simple hierarchy, where alternative paths are concatenated by preserving hierarchical dependencies. Figure 3-2 shows, the result of HINTA for the complex hierarchy of Figure 3-1 (the detailed transformation algorithm is discussed in Section 5).

3.2 HINTA for Star Schemata

The advantage of using HINTA in combination with hierarchy encoding is, that the dimension is left unchanged for the members of the hierarchy. Only the artificial key has to be recomputed. A dimension table D for Figure 3-1 may have the schema D(country, region, micromarket, turnoverclass, outlet, dimID). For the geographical hierarchy, dimID=$surr_{geo}$(country, region, micromarket, outlet), for the transformed hierarchy, dimID=$surr_{geotc}$(country, region, micromarket, turnoverclass, outlet), where surr is a function that computes the encoding for the corresponding hierarchy path.
These *physical* properties do not affect the schema. If the optimizer is able to handle hierarchy encoding, another hierarchy schema and therefore encoding even is transparent to the SQL statements (i.e., the optimizer recognizes, that a predicate on the dimension table with a corresponding join to the fact table can be replaced by a number of local interval predicates on the fact table). In such a case, the generated operator tree avoids expensive join operations. However, for the so called residual join, i.e., the join for the result set of the fact table to the dimension table in order to perform grouping, sorting, feature evaluation, postfiltering etc., the join cannot be prevented. Compared to the first pass of query evaluation, this residual join will be performed on a relatively small number of tuples and thus usually will not be critical for query execution.

4 A Hierarchy Model

Graphs represent relationships between vertices. Members in hierarchies are classified by relationships (usually 1:n relationships), which we in the following call *hierarchical relationships*. These hierarchical relationships can be represented in a directed graph. A hierarchy instance is the actual instantiation of the hierarchical relationship. A special case of a hierarchy instance is a hierarchy tree. In this paper, we extend the simple structure of a hierarchy tree to a more complex hierarchy graph. We use equivalence classes defined on the graph to describe hierarchy instances.
In the first part of this section, we work out properties of directed acyclic graphs (DAG) as model to describe hierarchies. The second part introduces hierarchy instances and schemata. We define some special hierarchies and describe typical hierarchies of data warehouses.
Basically, a *hierarchy instance* H corresponds to a graph $G = (V, E)$ with vertices $v_i \in V$ and typed edges $e_j \in E$. V is a finite set and E is a subset of $V \times V \times N$: $e^t \in E = (v_1, v_2,)^t$, where $v_1, v_2 \in V$ and $t \in N$ is a *type determinator (type)* specifying the type of the edge. We define a function $T: V \times V \times N \rightarrow N$ that returns the type of an edge e: $T(e) = T((v_1, v_2)^t) = t$.

4.1 Typed Directed Acyclic Graphs

We concentrate on DAGs ([CLR90]) with *typed edges*, abbreviated by *tDAG*. In a DAG, a vertex *v* is *adjacent* to *u*, if $u \rightarrow v$ or $(u, v) \in E$.

Example 4-1 (Graph):
Figure 4-1 illustrates a sample graph. This graph is a tDAG (the direction of the edges is denoted by arrows, the type of the edges is denoted by the edge style, a solid arrow denotes type 1, a dashed arrow denotes type 2). The vertices v_i are { *Germany, Austria, North, South, East, West, …, S6* }, the edges are: $E=\{A1 \rightarrow Aldi_N, Aldi_N \rightarrow North, North \rightarrow Germany, \ldots, West \rightarrow Austria\}$ or equivalently a set of pairs $E=\{(A1, Aldi_N)^1, (Aldi_N, North)^1, \ldots, (TG2, Germany)^2, \ldots, (West, Austria)^1\}$.

Definition 4-1 (Path ϕ, Typed Path ϕ^t, Pathlength):
A *path* ϕ from *u* to *v* is a sequence of adjacent vertices $(v_1, v_2, \ldots, v_n)$, where $v_i \rightarrow v_{i+1}$, $i = 1, \ldots, n-1$ and $v_1 = u$ and v_n

$= v$. We say, v is *reachable* from u via ϕ: $u \xrightarrow{\Phi} v$. We say, ϕ *contains* the vertices $v_1, v_2, \ldots, v_n$.
A *typed path* ϕ^t is a path with a type t, the function $T: (E\times .. \times E) \rightarrow N$ returns the type:

$$T(\Phi)=\begin{cases} t & \text{if } \forall v_i, v_{i+1} \in \Phi: T((v_i, v_{i+1}))=t \quad (i=1,\ldots,n-1) \\ \perp & \text{otherwise} \end{cases}$$

Two paths $\phi_1 = (v^1_1, v^1_2, \ldots, v^1_n)$ and $\phi_2 = (v^2_1, v^2_2, \ldots, v^2_n)$ have the same type t, if the types of all edges of ϕ_1 and ϕ_2 are the same: $T(\phi_1) = T(\phi_2) = t$.
The *pathlength* is the number of edges in path ϕ. *pathlength*(ϕ) = *pathlength*(path(u, v)), if $\phi = u \xrightarrow{\Phi} v$ and *path*(u,v) is the path ϕ from u to v.
The type of a path is only defined, if all edges in the path have the same type.

Example 4-2 (Path, Pathlength):
We use Figure 4-1 as example graph. There are two paths from *"A1"* to *"Segment"* ϕ_1*=("A1", "Aldi$_N$", "North", "Germany", "Segment")* and ϕ_2*=("A1", "TG1", "Germany", "Segment")*. *pathlength*(ϕ_1) *= 4* and *pathlength*(ϕ_2) *= 3*, where $T(\phi_1)=1$ and $T(\phi_2) = 2$.

Definition 4-2 (Rooted tDAG):
A *rooted tDAG* is a tDAG that has one vertex r that is reachable from all vertices $v_i \in V \setminus \{r\}$. Thus, there is a path from all $v_i \in V$ to r, $v_i \neq r$. Vertex r is called *root vertex* (or *root*).
If the union of two tDAGs $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$ is not rooted (i.e., $G=G_1 \cup G_2=(V_1 \cup V_2, E_1 \cup E_2)$ is not a rooted tDAG), but G_1 and G_2 are rooted tDAGs, we can construct a rooted tDAG G of $G_1 \cup G_2$ by adding a new vertex r and two edges $e_1 = (r_{G1}, r)$ and $e_2 = (r_{G2}, r)$, where r_{G1} is root of G_1 and r_{G2} is root of G_2: $G=(V_1 \cup V_2 \cup r, E_1 \cup E_2 \cup (r_{G1}, r) \cup (r_{G1}, r))$.

Definition 4-3 (Outdegree, Indegree):
The *out-degree* of a vertex u (*outdegree(u)*) is the number of edges leaving u, *outdegree^t(u)* is the number of edges with type *t*, leaving *u*.
The *in-degree* of u (*indegree(u)*) is the number of edges entering *u*, *indegree^t(u)* is the number of edges with type *t* entering *u*, correspondingly.
The *degree* of *u* is the sum of indegree(u) and outdegree(u).
A rooted tDAG has a number of vertices v_i with indegree(v_i) = 0. These vertices are called *leaf vertices* v_{leaf} (or *leaves*). In the graph of Figure 4-1, the leaf vertices are *{A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6}*. A *root vertex (root)* r has an out-degree of 0.
We further consider graphs, where every leaf vertex has at least one typed path to the root. We additionally require, that for every vertex *v outdegree^t(v)=1*.

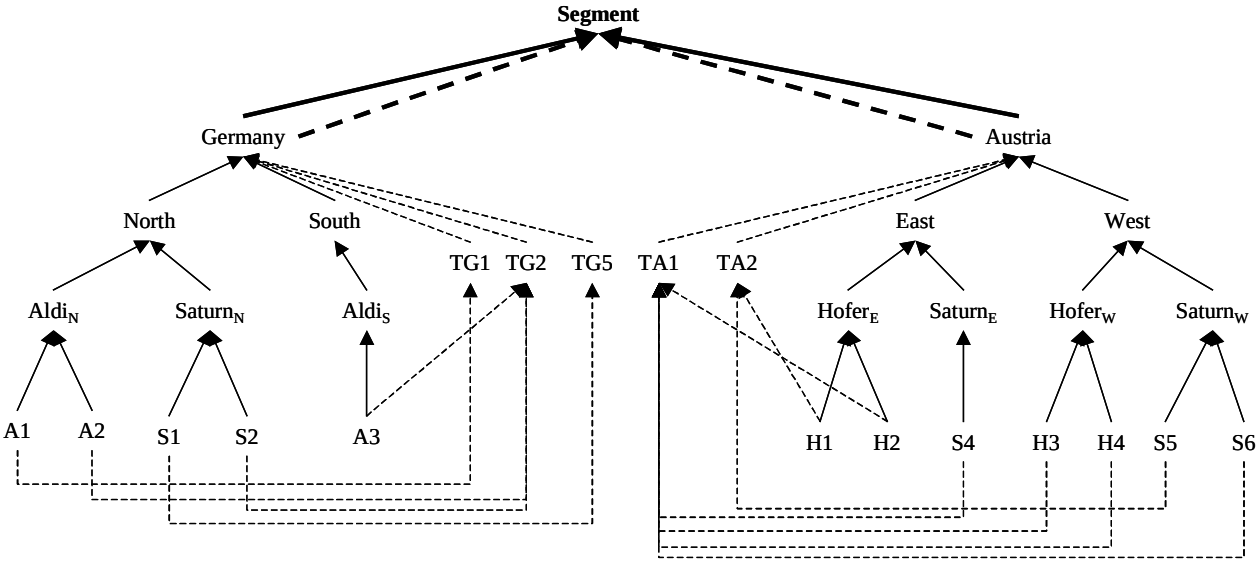

Figure 4-1: Rooted Directed Acyclic Graph

Example 4-3 (Indegree, Outdegree, Degree):
In Figure 4-1, the vertex Saturn$_N$ has the following degrees: indegree(Saturn$_N$) = indegree^1(Saturn$_N$) = 2, outdegree(Saturn$_N$)=1, degree(Saturn$_N$)=3.

Definition 4-4 (Subgraph):
A *subgraph G'* of graph *G=(V,E)* is a graph, whose vertices *V'* and edges *E'* are subsets of vertices *V* and edges *E* of *G*: *G'=(V', E'), V'⊆V, E'⊆E.*

Definition 4-5 (Simple tDAG):
A *simple tDAG (stDAG)* $T^s=(V^s, E^s)$ is a subgraph of *G* with edges of one type *t*. The vertices of T^s are the vertices contained in all paths ϕ^t_k from leaves of *G* to the root, and *pathlength*(ϕ^t_i)= *pathlength*(ϕ^t_j), i.e., all paths from leaves to root with same type and length.

Theorem 4-1:
A simple tDAG T^S is a balanced tree.

Proof:

1. A stDAG is a tree:
   According to the definition of trees ([Knu99])[1], a tree T has the following properties:
   $T=(V, E)$, where $v_i \in V$ are the vertices and $e_i \in E$ are directed edges, where $e_i = (root(T_j), root(T))$ and $1 \leq j \leq m$. T is a special case of a DAG, where $outdegree(v_i)=1$ for all $v_i \in V \setminus \{root(T)\}$. For every $v_i \in V \setminus \{root(T)\}$, there is a path from v_i to $r = root(T)$: $\forall v_i \in V \setminus \{root(T)\}\ \exists \phi: v \xrightarrow{\Phi} r$.
   A stDAG (V, E) is a rooted DAG with edges of t. $outdegree^t(v_i)=1 = outdegree(v_i)$ (see Definition 4-5) for $v_i \in V \setminus \{r\}$, where r is the root. For every vertex v_i of the stDAG, there is a path from v_i to *root*: $\forall v_i \in V \setminus \{r\}\ \exists \phi: v \xrightarrow{\Phi} r$.
   Thus, a stDAG is a tree.
2. A stDAG is a balanced tree:
   In a balanced tree, the height (i.e., the maximum path-length of the path from leaves to the *root*) of the sub-trees is equal or has a difference of at most 1.
   In a stDAG, the pathlength of all paths from the leaves to the root is equal.
   Thus, a stDAG is a balanced tree.

q.e.d.

Definition 4-6 (Equivalence Class):
An *equivalence class* is a set of vertices with the following properties: Two vertices u, v of a simple tDAG $T^S=(V^S, E^S)$, $u, v \in V^s$ are elements of equivalence class c, if $pathlength(path(u, root)) = pathlength(path(v, root))$, i.e., if the path length of the path from the vertices of c to the root is identical (same distance).

Example 4-4 (Simple tDAG, Equivalence Class):
In the graph of Figure 4-1, two simple tDAGs T^1 and T^2 are defined:
$T^1 = (V^1, E^1)$, where V^1 = {A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6, $Aldi_N$, $Saturn_N$, $Aldi_S$, $Hofer_E$, $Saturn_E$, $Hofer_W$, $Saturn_W$, North, South, East, West, Germany, Austria, Segment}
$T^2 = (V^2, E^2)$, where V^2 = {A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6, TG1, TG2, TG5, TA1, TA2, Germany, Austria, Segment}
Equivalence classes of T^1 are c_1^1={A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6 }, c_2^1={$Aldi_N$, $Saturn_N$, $Aldi_S$, $Hofer_E$, $Saturn_E$, $Hofer_W$, $Saturn_W$}, c_3^1={North, South, East, West}, c_4^1={Germany, Austria} and c_5^1={Segment}.
Equivalence classes of T^2 are c_1^2={A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6}, c_2^2={TG1, TG2, TG5, TA1, TA2}, c_3^2={Germany, Austria} and c_4^2={Segment}.

4.2 Hierarchies

With the definitions of graphs, we now can define hierarchies. We draw a parallel from the concepts of rooted tDAGs to hierarchies. This section describes hierarchies and their properties.

Definition 4-7 (Hierarchy Instance):
A hierarchy instance H is a rooted tDAG $H=(V, E)$ with members $m_i \in V$ and directed, typed edges $e_j \in E$. The edges are called hierarchical relationships. We call a member m_i *hierarchically dependent* on m_j, if $m_j \rightarrow m_i$ (or equivalently $(m_j, m_i) \in E$). We call a member m_i *indirect hierarchically dependent* on m_j, if m_i is reachable from m_j via a path ϕ: $m_j \xrightarrow{\Phi} m_i$, also denoted by $m_j \xrightarrow{*} m_i$.

We additionally define *sub-hierarchies*, called *simple hierarchies* $H^S=(V^S,E^S)$ that correspond to simple graphs. All simple hierarchies H_i^S of a hierarchy instance H are partitions of H. The union of the simple hierarchies is the hierarchy instance H: $\bigcup_i H_i^S = H$. This follows from the definition of simple graphs.

Definition 4-8 (Hierarchy Level):
A *hierarchy level or level* is an equivalence class of a simple hierarchy containing members with the same distance from the root. We call the level, consisting of leaves, *leaf level* and the level, consisting of the root, *root level*. A simple hierarchy is a *balanced hierarchy tree* with a depth equal to the pathlength of the path from the leaves to the root.

Example 4-5 (Hierarchy Instance, Simple Hierarchy, Hierarchy Level):
The graph illustrated in Figure 4-1, is a hierarchy instance with two simple hierarchies H_1 and H_2.
The levels of $H_1=(V_1, E_1)$ are V_1={h_1^1, h_2^1, h_3^1, h_4^1, h_5^1}, where h_1^1={A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6}, h_2^1={$Aldi_N$, $Saturn_N$, $Aldi_S$, $Hofer_E$, $Saturn_E$, $Hofer_W$, $Saturn_W$}, h_3^1={North, South, East, West}, h_4^1={Germany, Austria} and h_5^1={Segment}.

Definition 4-9 (Hierarchically Dependent Levels):
A level h_j is *hierarchically dependent* on h_i, if all members $m^j_k \in h_j$ are hierarchically dependent on members $m^i_h \in h_i$, i.e., $\forall m^j_k \in h_j\ \exists m^i_h \in h_i$: $(m^i_h, m^j_k) \in E$. The function $HD: (V, V \times V) \rightarrow (V \times V)$ computes the hierarchical relationships $\{(h_i, h_j)\}$ of the levels of a hierarchy instance $H=(V, E)$, where h_i, h_j are levels of H and h_i is hierarchically dependent on h_j.
A level h_j is *indirect hierarchically dependent* on h_i, if all members $m^j_k \in h_j$ are indirect hierarchically dependent on members $m^i_h \in h_i$.

The order of dependencies often intuitively is misunderstood. A simple example illustrates the correct hierarchical dependencies: In a geographic hierarchy with levels *country*, *state* and *town*, the level state is hierarchically dependent on *town*, because state is determined by the towns. The level *country* also is hierarchically dependent on *town*, however indirect (via level *state*).

[1] A tree is a finite set T of one or more vertices such that there is one specially designated node called the root of the tree, $root(T)$, and the remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$ and each of these sets in turn is a tree. The trees $T_1, \ldots, T_m$ are called the subtrees of the root.

Example 4-6 (Hierarchically Dependent Levels):
$HD(H_1)$ returns the following hierarchical dependencies: $HD(H_1)$ = {(A1, $Aldi_N$), (A2, $Aldi_N$), (S1, $Saturn_N$), (S2, $Saturn_N$), (A3, $Aldi_S$), …, (Germany, Segment), (Austria, Segment)}, i.e., all edges of the graph representing H_1.

Definition 4-10 (Shared Level):
Two levels $h_1=\{m_k^1\}$ and $h_2=\{m_j^2\}$ are *shared levels*, if the intersection of h_1 and h_2 is not empty. Otherwise the levels h_1 and h_2 are not shared. We call such levels h_1 and h_2 *disjoint levels*.

Definition 4-11 (Distinct Operator):
The *distinct* operator $L \rightarrow L$ returns a subset of levels $L'=\{h_k\}$ of a set of levels $L=\{h_i\}$: $distinct(L)=\{h_k\}=L'$, where $\forall h_k, h_h \in L': h_k \neq h_h$. There are no equal levels in L'.
If there are several paths from a member to the root (usually true for shared levels), we call these paths *alternative paths*.

Example 4-7 (Shared Level, Distinct Operator):
According to Example 4-5, shared levels are: $h_1^1=h_1^2$, $h_4^1=h_3^2$, $h_5^1=h_4^2$.
For the hierarchy instance $H = H_1 \cup H_2$, the distinct operator returns the following levels:
$distinct(H) = \{ h_1^1, h_2^1, h_3^1, h_4^1, h_5^1, h_2^2\}$.
The distinct operator generally is not deterministic. However, the members of the levels specified by the distinct operator, are deterministic (e.g., the members of h_1^1 and h_1^2 are the same).

Definition 4-12 (Balanced Hierarchy):
A *balanced hierarchy* is a hierarchy, whose leaf members are contained in one (shared) level h_l. Simple hierarchies are always balanced hierarchies, because they have only one leaf level (balanced tree).

Definition 4-13 (Hierachy Schema):
The *hierarchy schema HS* is a rooted tDAG specified by $HS=(L^S, E^S)$, where L is a set of levels h_i, and E^S is a set of hierarchical relationships $E^S=(h_i, h_j)$ between the levels, i.e., h_j is hierarchically dependent on h_i.

Definition 4-14 (Schema-Instance Conformity):
A hierarchy schema $HS=(L^S, E^S)$ *conforms* to a hierarchy instance $H=(V^H, E^H)$, if the number of levels of *HS* and *H* is equal, and the hierarchical dependencies of these levels are equal:
$\forall (h_i^S, h_j^S) \in E^S: \exists(h_i^H, h_j^H) \in HD(H) \wedge \forall(h_i^H, h_j^H) \in HD(H): \exists(h_i^S, h_j^S) \in E^S$

Example 4-8 (Hierarchy Schema and Instance):
On the left side of Figure 3-1, a hierarchy schema is illustrated: $HS=(L, E^S)$, where $L=\{Outlet, MicroMarket, Region, TurnoverClass, Country, Dimension\}$ and $E^S=\{(Outlet, MicroMarket), (MicroMarket, Region), (Region, Country), (Outlet, TurnoverClass), (TurnoverClass, Country), (Country, Dimension)\}$.
As hierarchy instance H, we use Example 4-5. $H = H_1 \cup H_2 = (V, E^H)$, where the levels are $L^H=\{h_1^1, h_2^1, h_3^1, h_4^1, h_5^1, h_1^2, h_2^2, h_3^2, h_4^2\}$. The distinct levels are $distinct(L^H) = \{h_1^1, h_2^1, h_3^1, h_4^1, h_5^1, h_2^2\}$ and the hierarchical dependencies are $HD(H) = \{(h_1^1, h_2^1), (h_2^1, h_3^1), (h_3^1, h_4^1), (h_4^1, h_5^1), (h_1^1, h_2^2), (h_2^2, h_4^1)\}$. If we map *Outlet* to h_1^1, *MicroMarket* to h_2^1, *Region* to h_3^1, *Country* to h_4^1, *Dimension* to h_5^1 and *TurnoverClass* to h_2^2, the hierarchy schema *HS* conforms to hierarchy instance *H* of Figure 4-1.

4.3 Hierarchies in Data Warehouses

Hierarchies are used to classify the dimensions of a DW. DW model complex business contexts. Additional attributes are used to provide additional classification information, e.g., the screen size of TV sets. For this reason, the members can be augmented by *classification features*. Therefore, a member v in a hierarchy graph is a pair $v=(id, \{f_i\})$, where *id* is a unique identifier of the vertex, called *member label* (or *label*), and $\{f_i\}$ is a set of additional attributes, called *feature attributes*. We call such a graph an *attributed tDAG*.
Feature attributes are assigned to hierarchy members. Generally, a member can have an arbitrary number of features. In many DW hierarchies, however, the hierarchy members of one hierarchy level have the same number of feature attributes[2]. In this case, features are assigned to hierarchy levels.
In a DW, hierarchies are assigned to dimensions. One dimension can contain several hierarchies. We combine all hierarchies of one dimension to one hierarchy instance corresponding to a rooted tDAG, where the root is the "All" level. Such a hierarchy instance is called *DW-hierarchy*. Usually, facts have a base granularity with respect to every dimension. This base granularity corresponds to one leaf level of the DW hierarchy. Thus, a DW-hierarchy only has one (shared) leaf level.
If facts are classified with respect to different granularities[3] (leaf hierarchy levels), new aggregation and grouping semantics have to be introduced ([Leh98a]). The hierarchy model, however, supports such degenerated hierarchies.

5 Transforming Hierarchy Instances to Simple Hierarchies

First, we discuss some algorithms, that are used by HINTA. Then we discuss HINTA in detail and show a complete example of HINTA for the hierarchy instance of Figure 3-1.

5.1 Primitive Hierarchy Instances (phi)

We use the term *primitive hierarchy instance, phi,* for hierarchy instances, that consist of two simple hierarchies with one shared leaf level - the remaining levels are disjoint. Such a phi can be transformed to one simple hierarchy instance. A phi is some kind of sub-hierarchy of a

[2] Hierarchy members of one hierarchy level usually categorize the same information, e.g., the level "country" may have feature attributes like number of inhabitants, gross national product, etc. for every country stored in the hierarchy.
[3] unbalanced hierarchies

conventional hierarchy instance consisting of two simple hierarchies.

A phi H consists of a number of hierarchically dependent disjoint levels and one shared leaf level. Figure 5-1 illustrates all possible hierarchy schemata of phi (phi_1, phi_2, phi_3).

phi_1 only contains one shared level, i.e., the leaf level of H. Such a phi can be constructed, if a hierarchy has several hierarchically dependent shared levels. This sequence of levels is split into phi's of type phi_1 for every level. Usually, edges of both simple hierarchies "leave" phi_1 (illustrated by dotted arrows). Thus, the original hierarchy has a level hierarchically dependent on h_i, if h_i is not the root level.

phi_2 is the general case for a phi. Two simple hierarchies H_1 and H_2 have one shared leaf level h_i and a number of hierarchically dependent levels h_k, …, h_h for H_1 and h_j, …, h_l for H_2. Usually, a level h_x (shared level) is hierarchically dependent on h_h and h_l. The dotted arrows denote these hierarchical relationships.

phi_3 is a special case of phi_2, where H_1 only consists of the shared leaf level h_i, and H_2 consists of additional hierarchically dependent levels h_j, …, h_l.

In Figure 5-1, a splitting of a hierarchy schema of hierarchy H with the two simple hierarchies H^S_1 and H^S_2 into phi's is illustrated. H^S_1 consists of the levels *{A, B, D, E, G, J}*, H^S_2 consists of the levels *{A, B, E, F, G, I, J}*. Shared hierarchy paths (levels *A* and *B*) are from type phi_1, the alternative paths for levels $G \rightarrow D \rightarrow C$ and $G \rightarrow F \rightarrow E$ are from type phi_2, and the alternative paths J and $J \rightarrow I$ are from type phi_3.

No other phi are possible for two simple hierarchies, because by concatenating phi's, all hierarchy instances for

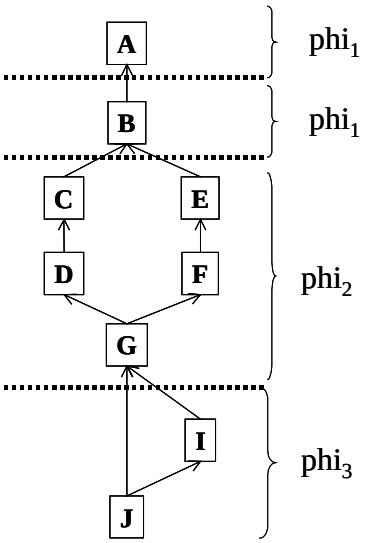

Figure 5-2: Example of phi's

two simple hierarchies can be constructed.

A phi of a hierarchy instance $H=H_1 \cup H_2$ formally is defined in the following way:

Definition 5-1 (Primitive Hierarchy Instance, phi):
The *primitive hierarchy instance (phi)* of a hierarchy instance H consisting of two simple hierarchies $H^S_1=(V^S_1, E^S_1)$ and $H^S_2=(V^S_2, E^S_2)$ is a hierarchy instance $H^{phi}=(V^{phi}, E^{phi})$:

$V^{phi} = \{h_1^m, h_1^{m-1}, \ldots, h_1^k, h_2^n, h_2^{n-1}, \ldots, h_2^h\}$, where h_1^m resp. h_2^n are root levels of H^S_1 resp. H^S_2 and h_1^k and h_2^h are shared levels and $\forall h_1^j$, $k<j\leq m$: $\forall h_2^i$, $h<i\leq n$: h_1^j, h_2^i are disjoint levels.

$E^{phi} = \{e_i\}$, where $e_i = (v_i, v_j) \in (E^S_1 \cup E^S_2)$: $v_i, v_j \in \{(V^{phi} \cup v_x)\}$, $v_k \rightarrow v_x$, $v_k \in V^{phi}$.

E^{phi} contains all original edges between the members of V^{phi} and the "leaving" edges.

Example 5-1 (Primitive Hierarchy Instance):
This example shows the phi's of the sample hierarchy instance H of Figure 4-1. H consists of three phi's: H^{p1}, H^{p2} and H^{p3}.

H^{p1} is of type phi_1. $V^{p1} = \{Segment\}$, $E^{p1} = \varnothing$, because the root does not have leaving edges.

H^{p2} is of type phi_1 again and consists of the members $V^{p2} = \{Germany, Austria\}$ and the edges $E^{p2} = \{(Germany, Segment)^1, (Austria, Segment)^1, (Germany, Segment)^2, (Austria, Segment)^2\}$

The edges are of type 1 and 2 (see Figure 4-1).

H^{p3} is of type phi_2 and consists of two alternative paths with shared leaf level *Outlet* (see Figure)

$V^{p3} = \{A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6, Aldi_N, Saturn_N, Aldi_S, Hofer_E, Saturn_E, Hofer_W, Saturn_W, TG1, TG2, TG5, TA1, TA2, North, South, East, West\}$

$E^{phi3} = \{(A1, Aldi_N), (A2, Aldi_N), (S1, Saturn_N), (S2, Saturn_N), (A3, Aldi_S), \ldots, (South, Germany), (East, Austria), (West, Austria)\}$

5.2 Transformation of Primitive Hierarchy Instances

A phi can be transformed to a simple hierarchy. In this section, members are denoted by v. If v is in level h_i, i.e., $v \in h_i$, we write v_i, if $v \in h_j$, we write v_j etc. We write v_x and v_y for members not within the hierarchies. An edge (v_h, v_x)

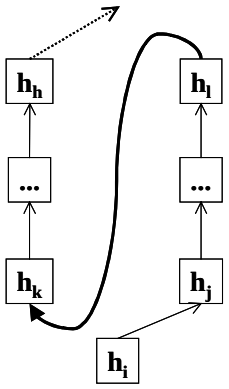

Figure 5-4: Transformation of a phi_2

is a leaving edge of v_h. Depending on the type of the phi, the hierarchy is transformed by deleting and adding special members and edges. A phi consists of two simple hierarchies. For the transformation, one hierarchy is *preferred*, i.e., the levels of the preferred hierarchy usually are more significant for the encoding than the levels of the other hierarchy (predicate *isPreferred*). The *isPreferred:*

E→Bool predicate (i.e., isPreferred(e) = TRUE | FALSE) returns TRUE, if edge e is the edge of the preferred hierarchy. Usually, a hierarchy is preferred, if it is used in more queries than the other hierarchy. There can be many preference criteria (e.g., numbers, importance or kind of queries etc.).

The transformation algorithm is specified in pseudo code:

TransformPhiToSimpleHierarchy:

```
if type(H^phi)=phi1, then
  forall edges (vi, vx)
    if not isPreferred(vi, vx) then
      delete edge(vi, vx)
    /* delete leaving edges of the non-
       preferred hierarchy, leaving edges
       of preferred hierarchy remain*/
if type(H^phi)=phi2, then
  forall edges (vl, vx)
    if not isPreferred(vl, vx) then
      delete edges (vl, vx)
      /* delete leaving edges of the non-
         preferred hierarchy */
  forall edges (vi, vk)
    if not pathexists(vi→vj'→…→vl'→vk)
      insertpath(vi→vj'→…→vl'→vk)
    delete edges (vi, vk)
  forall edges (vi,vj) delete edges (vi,vj)
  /* make vk indirect hierarchically
     dependent on vi (instead of direct
     hierarchically dependent) by duplica-
     ting vertices and edges */
if type(H^phi)=phi3, then
  for all edges (vi, vx)
    delete edges (vi, vx)
```

For phi's of type phi_1 and phi_3, only "leaving" edges must be removed. For a phi_1, all leaving edges of one of the two hierarchies must be removed (in this case, the non-preferred hierarchy). For a phi_3, we must remove the "leaving" edges of the "small" hierarchy, because the levels of the other hierarchy must remain for hierarchical classification.

For phi's of type phi_2, a kind of hierarchy interleaving is performed. The alternative paths are concatenated in the meaning, that the levels of the non-preferred hierarchy are made hierarchically dependent on the levels of the preferred hierarchy. Members of the shared leaf level h_i are not directly hierarchically dependent on members of h_k any more (see Figure 5-4). The operation `insert-path(path)` inserts members and edges of *path*. This is necessary, because a member $v_j \in h_j$ can be adjacent to several members $v_i \in h_i$, that do not correspond to an equal number of members $v_k \in h_k$. Thus, we have to duplicate the path to preserve hierarchical dependencies.

Instead of the discussed transformation algorithm, that concatenates two simple hierarchies, a *hierarchy level interleaving* also is possible. This interleaving corresponds to a topological sorting of the hierarchies. Further research is necessary to work out the advantages of the transformation methods.

5.3 Hierarchy Instance Transformation Algorithm (HINTA)

The **H**ierarchy **IN**stance **T**ransformation **A**lgorithm, *HINTA*, transforms a hierarchy instance *H=(V, E)*, represented by a rooted tDAG (e.g., a DW-hierarchy) into simple hierarchy $H^S = HINTA(H) = (V^S, E^S)$. The input of HINTA is a hierarchy instance that consists of an arbitrary number *n* of simple hierarchies H^S_k. We transform two simple hierarchies H^S_1 and H^S_2 to one simple hierarchy by splitting them into phi's and transforming each phi with

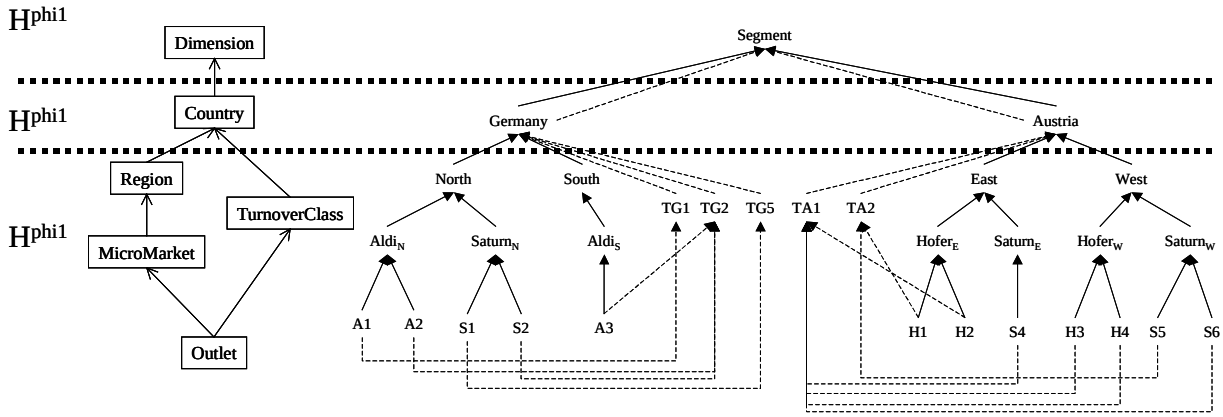

Figure 5-3: Schema and Instance of phi's

`TransformPhiToSimpleHierarchy` into a primitive simple hierarchy. The primitive simple hierarchies are merged to the resulting simple hierarchy H^S_{12}. We now transform H^S_{12} and the next simple hierarchy H^S_3 to H^S_{123} according to the previous described steps etc. Thus, at the end of HINTA, we get one simple hierarchy $H^S_{123..n}$.

In the following, we describe the process in a more formal manner:

The input hierarchy H is split into simple hierarchies H^S_i:

$H = \bigcup_i H_i^S$, where H^S_i is *preferred* to H^S_{i+1}.

HINTA: H^S = HINTA(H)

According to the informal description of HINTA above, we transform a pair of simple hierarchies into one simple hierarchy, starting with the first two simple hierarchies in preference order.

H_{12} = Transform($H^S_1 \cup H^S_2$)

The resulting simple hierarchy H_{12} and the next preferred simple hierarchy H^S_3 are transformed:

H_{123} = Transform($H_{12} \cup H^S_3$)

The resulting simple hierarchy H_{123} and the next preferred simple hierarchy H^S_4 are transformed etc. Thus, we have *n-1* calls of Transform for *n* simple hierarchies of *H*. The transformation calls also can be summed up in one expression:

H_{12} = Transform($H^S_1 \cup H^S_2$)
H_{123} = Transform($H_{12} \cup H^S_3$) = Transform(
Transform($H^S_1 \cup H^S_2$) $\cup$ H^S_3)
.....
$H_{123..n}$ = Transform($H_{12..n-1} \cup H^S_n$) =

Transform(Transform(.... Transform
($H^S_1 \cup H^S_2$) $\cup$ H^S_3) $\cup$ … $\cup$ H^S_{n-1}) $\cup$ H^S_n)

The function Transform splits a hierarchy instance *H* consisting of two simple hierarchies into phi's, transforms each phi into a simple hierarchy (TransformPhiToSimpleHierarchy) and concatenates the resulting simple hierarchies to one simple hierarchy:

Transform (H):
if H is phi then
Transform(H) = TransformPhiToSimpleHierarchy(H)
otherwise
Transform(H) = TransformPhiToSimpleHierarchy(phi(H)) $\cup$ Transform(H\phi(H))

Transform is a recursive function, that transforms the first phi of H into a simple hierarchy H^S and concatenates H^S with the rest of the transformed phi's by calling Transform again. It terminates, when *H* is already a phi, i.e., if the last phi of the original hierarchy instance is the input parameter.

The "$\cup$" operator means, that for $H_1 \cup H_2$ the hierarchies H_1 and H_2 are concatenated via the existing edges of members of H_1 and H_2.

The "\" operator is a splitting of the hierarchies H_1 and H_2, i.e. $H^* = H_1 \setminus H_2$ means, that H* is the hierarchy H_1 without the members and edges of H_2. Thus, $H \setminus phi(H)$ is hierarchy *H* without the first phi of *H*.

Transform is called n times, if H consists of n phi's. Thus, Transform terminates, because a hierarchy instance H consists of a finite number *n* of phi's.

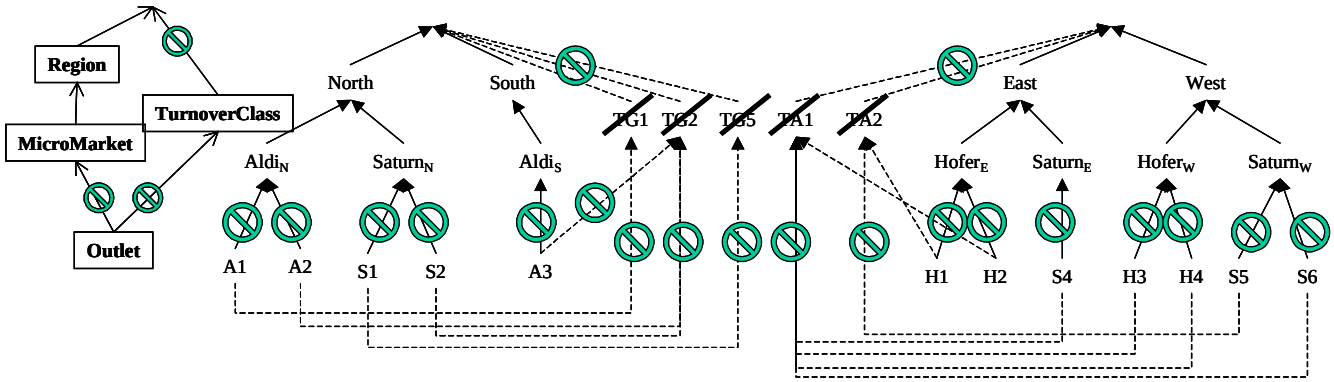

Figure 5-5: Deleting Members and Edges

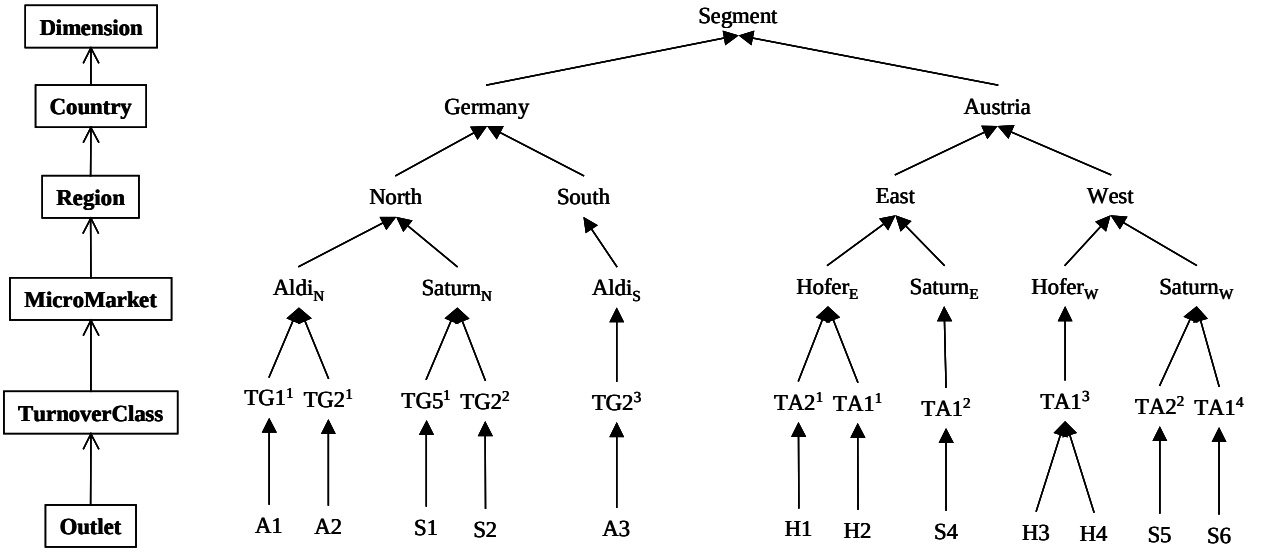

Figure 5-6: Final Simple Hierarchy Instance and Schema

5.4 Example of HINTA

To illustrate HINTA, we use the hierarchy instance $H=(V^H, E^H)$ of Figure 4-1, where *H* contains two simple hierarchies H_1^S and H_2^S (see Example 4-5). We assume, that H_1^S is preferred to H_2^S. The resulting simple hierarchy H^S is computed by:

H^S= HINTA(H)

The pair of simple hierarchies H_1 and H_2 is transformed by Transform($H_1 \cup H_2$), where $H_1 \cup H_2$ is not a phi. ($H_1 \cup H_2$) consists of three phi, i.e., H^{phi1}, H^{phi2} and H^{phi3}.

H^{p1} (of type phi_1) is the root level without edges, because the root level does not have leaving edges.

H^{p2} (of type phi_1) consists of the members {Germany, Austria} and the corresponding edges to the root (see Figure 5-3 and Example 5-1).

H^{p3} =(V, E) (of type phi_2) consists of two alternative paths with shared leaf level *Outlet* (see Figure 5-3 and Example 5-1).

Now we transform H^{p1}, H^{p2} and H^{p3} to simple hierarchies:

H^S_1=TransformPhiToSimpleHierarchy(H^{p1})

H^S_2=TransformPhiToSimpleHierarchy(H^{p2})

H^S_2=TransformPhiToSimpleHierarchy(H^{p3})

H^S_1 = ({Segment}, ∅}, i.e. the root without edges.

H^S_2 = ({Germany, Austria}, {(Germany, Segment), (Austria, Segment)}, because H^{p2} is a phi of type phi_1, the edges of type 2 are deleted.

H^S_3 = (V, E): H^{p3} is of type phi_2 and

we **delete the edges** {(A1, $Aldi_N$), (A2, $Aldi_N$), (S1, $Saturn_N$), (S2, $Saturn_N$), (A3, $Aldi_S$), (H1, $Hofer_E$), (H2, $Hofer_E$), (S4, $Saturn_E$), (H3, $Hofer_W$), (H4, $Hofer_W$), (S5, $Saturn_W$), (S6, $Saturn_W$)} and the edges {(TG1, Germany), (TG2, Germany), (TG5, Germany), (TA1, Austria), (TA2, Austria)} (edges between leafs and the preferred hierarchy are deleted, because now the members of the non preferred hierarchy are directly dependent on the leafs).

Figure 5-5 illustrates, which edges are deleted.

We **delete members** {TG1, TG2, TG5, TA1, TA2} and **insert new members** {$TA1^1$, $TG2^1$, $TG5^1$, $TG2^2$, $TG2^3$, $TA2^1$, $TA1^1$, $TA1^2$, $TA1^3$, $TA1^3$, $TA2^2$, $TA1^4$} and get the set of members:

V = { A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6, $Aldi_N$, $Saturn_N$, $Aldi_S$, $Hofer_E$, $Saturn_E$, $Hofer_W$, $Saturn_W$, North, South, East, West, $TA1^1$, $TG2^1$, $TG5^1$, $TG2^2$, $TG2^3$, $TA2^1$, $TA1^1$, $TA1^2$, $TA1^3$, $TA1^3$, $TA2^2$, $TA1^4$}

We **insert new edges** of level *TurnoverClass* to *Micromarket* preserving hierarchical dependencies: {(A1, $TG1^1$), ($TG1^1$, $Aldi_N$), (A2, $TG2^1$), ($TG2^1$, $Aldi_N$), (S1, $TG5^1$),

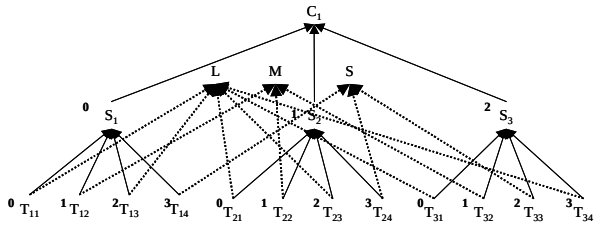

Figure 5-7: Complex Hierarchy

($TG5^1$, $Saturn_N$), (S2, $TG2^2$), ($TG2^2$, $Saturn_N$), (A3,$TG2^3$), …, ($TA1^4$, $Saturn_W$)}

After deleting and inserting, the edges are:

E = {(A1, $TG1^1$), (A2, $TG2^1$), (S1, $TG5^1$), (S2, $TG2^2$), (A3, $TG2^3$), (H1, $TA2^1$), (H2, $TA1^1$), (S4, $TA1^2$), (H3, $TA1^3$), …, (North, Germany), (South, Germany), (East, Austria), (West, Austria)}

The resulting simple hierarchy of HINTA is the union of H^S_1, H^S_2 and H^S_3, as illustrated in Figure 5-6.

5.5 Effects of HINTA

This section illustrates the benefit of HINTA in an example. Usually, the benefit of HINTA is as better as larger

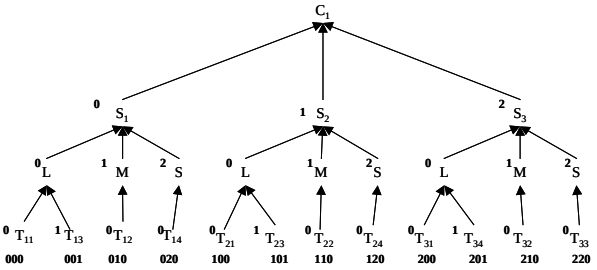

Figure 5-8: Transformed Hierarchy

the hierarchies are. Due to graphical illustrations, however, only small hierarchies can be drawn. In Figure 5-7 we show a complex hierarchy with the levels H_1=*Country-State-Town* and H_2=*Country-Category-Town*, where *Category* characterizes the size of the towns (*large*, *middle*, *small*). Hierarchy Encoding is applied to H_1, i.e., a predicate "*Category=L*" would include the leaves {T11, T13, T21, T23, T31, T34} or equivalently the dimIDs {00, 02, 10, 12, 20, 23}, i.e., 6 intervals include one single value. With a transformed hierarchy (as in Figure 5-8), we get three intervals [000, 001], [100, 101] and [200,201]. With these intervals, the query will be processed faster on the fact table.

5.6 Remarks

This section gives a short impact to the quality of HINTA. We consider a hierarchy such as in Figure 3-1, where one is the *preferred* hierarchy, the other is the *non-preferred* hierarchy. We use an encoding of the preferred hierarchy (EPH), an encoding of the non-preferred hierarchy (ENPH) and an encoding of the transformed hierarchy with HINTA (ETH).

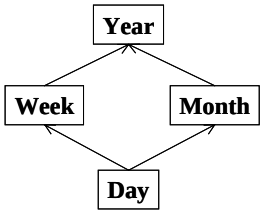

Figure 6-1: Time Hierarchy

In general, no perfect linearization is possible for all simple hierarchies. However, due to the transformation algorithm, the preferred hierarchy still is encoded perfectly, i.e., a predicate (point restriction) on EPH resulting in one interval also will result in one interval for ETH, because the affected leaves are the leaves of a complete sub-tree. This is due to the higher priority of the preferred hierarchy (the levels of the preferred hierarchy usually are higher levels than the levels of the non-preferred hierar-

chy). Thus, no disadvantages in query execution will result from HINTA for such predicates.
The clustering of ETH compared to ENPH, however, is not optimal, and a predicate on a level of ENPH might result in a number of intervals instead of one interval. This number highly depends on the correlation of the two simple hierarchies: If the hierarchies are correlated in a large extent (e.g., for the time hierarchy in Figure 6-1 with the preferred hierarchy year – month – day), most restrictions such as year = 1999 and week = 33 will result in an interval on dimID for the transformed hierarchy year – month – week – day.

6 Conclusions and Future Work

In this paper, we present a concept, how to represent hierarchies by directed acyclic typed graphs. We concentrate on complex hierarchies and describe HINTA, a method to linearize complex hierarchies by transforming them to simple hierarchies and thus allow hierarchical clustering on such hierarchies.
The quality of clustering depends on the correlation of the simple hierarchies. For query processing, predicates on hierarchy levels are mapped to intervals by an encoding that preserves hierarchical clustering.
Currently, in the EDITH project ([Edi01]), we are integrating encoding algorithms into the kernel of the relational DBMS TransBase ([Tra01]) and are investigating query processing algorithms and optimizer strategies to efficiently support these methods. Application partners will test the DBMS implementation. We are going to analyze HINTA by transforming complex real world hierarchies and compare clustering properties.
We thank Martin Zirkel and Robert Fenk for fruitful discussions for reading and correcting this paper.

7 References

[AGS97] R. Agrawal, A. Gupta, S. Sarawagi: *Modelling Multidimensional Databases*. In Proceedings of the 13 th International Conference on Data Engineering (ICDE 97), Birmingham, UK, pp. 232-243, IEEE Computer Society, 1997.

[Alb01] J. Albrecht: *Anfrageoptimierung in Data-Warehouse-Systemen auf Grundlage des multidimensionalen Datenmodells* (in German). Dissertation Thesis, Universität Erlangen-Nürnberg, 2001.

[BPT97] E. Baralis, S. Paraboschi, E. Teniente: *Materialized Views Selection in a Multidimensional Database*. In Proceedings of the 18 th International Conference on Very Large Data Bases (VLDB 97), Athens, Greece, pp. 156-165, Morgan Kaufmann, 1997.

[CLR90] T. H. Cormen, C.E. Leiserson, R.L. Rivest.: *Introduction to Algorithms*, MIT Press Cambridge, Massachusetts London, 1990

[CT98] L. Cabibbo, R. Torlone, *A logical approach to multidimensional databases*, Proc. 6th EDBT 1998, LNCS 1377, 183-197

[Edi01] EDITH: European Development on Indexing Techniques for Databases with Multidimensional Hierarchies, http://edith.in.tum.de/

[FS99] E. Franconi, U. Sattler: *A Data Warehouse Conceptual Data Model for Multidimensional Aggregation*, Proc. DMDW, 1999

[GMR98] M. Golfarelli, D. Maio, S. Rizzi: *Conceptual design of data warehouses from E/R schemes*, Proc. 32th HICSS 1998

[HLV00] B. Hüsemann, J. Lechtenbörger, G. Vossen: Conceptual Data Warehouse Design, Proc. DMDW, 2000

[Kim96] R. Kimball: *The Data Warehouse Toolkit*. John Wiley & Sons, New York. 1996.

[Knu99] D. E. Knuth: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Third Edition, Addison Wesley, Sixth printing, 1999

[Leh98a] W. Lehner: *Modeling Large Scale OLAP Scenarios*. In Proceedings of the 6 th International Conference on Extending Database Technology (EDBT' 98), Valencia, Spain, LNCS Vol. 1377, pp. 153-167, Springer Verlag, 1998.

[Leh98b] W. Lehner: *Adaptive Preaggregations-Strategien für Data Warehouses*.(in German) Dissertation Thesis, University of Erlangen-Nuremburg, 1998.

[LW96] C. Li, X. Sean Wang. *A Data Model for Supporting On-Line Analytical Processing*. CIKM 1996.

[Mar99] V. Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. Ph.D. Thesis, Technische Universität München, 1999.

[MRB99] V. Markl, F. Ramsak, and R. Bayer. *Improving OLAP Performance by Multidimensional Hierarchical Clustering*. Proc. of IDEAS'99, Montreal, Canada, 1999.

[PJD99] T. B. Pedersen, C. S. Jensen, C. E. Dyreson: Extending Practical Pre-Aggregation in On-Line Analytical Processing, Proc. 25th VLDB, 663-674, 1999

[Sap01] C. Sapia: *PROMISE: Modeling and Predicting User Behavior for Online Analytical Processing Applications*: Ph.D. Thesis submitted, Technische Universität München, 2001

[Sar97] S. Sarawagi. *Indexing OLAP data*. Data Engineering Bulletin 20 (1), 1997, pp. 36-43.

[Tra01] TransAction Software Gmbh. TransBase Documentation, 2001. www.transaction.de

[VS99] P. Vassiliadis, T. Sellis, *A Survey on Logical Models for OLAP Databases*: ACM SIGMOD Record 28(4) 1999, 64-69

[WB97] M.-C. Wu, A. P. Buchmann, *Research Issues in data warehousing*. Proc. Zth BTW 1997, 61-82

[ZSL98]C. Zou, B. Salzberg, and R. Ladin. *Back to the Future: Dynamic Hierarchical Clustering*. Proc. of the ICDE 1998: 578 – 587, 1998.