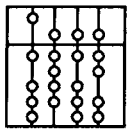
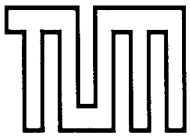


FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN



Modeling and implementing multidimensional hierarchically structured Data for Data Warehouses in relational Database Management Systems and the Implementation into Transbase®

Roland Pieringer

Institut für Informatik
der Technischen Universität München

Modeling and implementing multidimensional hierarchically structured Data for Data Warehouses in relational Database Management Systems and the Implementation into Transbase®

Roland Pieringer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Univ-Prof. Dr. F. Matthes

Prüfer der Dissertation:

1. Univ.-Prof. R. Bayer, Ph.D.
2. Univ.-Prof. Dr. B. Freitag
   Universität Passau

Die Dissertation wurde am 29.01.2003 bei der Technischen Universität München

eingereicht und durch die Fakultät für Informatik am 17.06.2003 angenommen.

In memory of my father, Georg Pieringer.

Acknowledgements

First of all, I want to thank my advisor Prof. Rudolf Bayer, Ph.D., for his support and fruitful discussions. Even for me who did an external Ph.D. thesis he found time to discuss and talk about problems. He spent his vacation reading and correcting my thesis!

Then, of course, I thank Transaction Software that supported me and gave me the time to work for the thesis. I also was allowed to participate in interesting conferences. I especially want to thank Klaus Elhardt who was a steady discussion partner and influenced the work a lot due to his experience in the field of database systems (theoretically and practically). I also thank Chrstian Roth, the head of Transaction Software, who enriched the work by many commercial ideas. Of course, I am very grateful to all other colleges, too, that are (in alphabetical order): Ralph Acker, Adolf Alt, Gerhard Dünzinger, Martha Jelinek, Dieter Killar, Susanne Krall, Erwin Loibl, Wolfgang Schwarz and Christof Seibt for all the moral support and other help.

I esspecially thank the people at FORWISS and TU-München, in particular, Volker Markl, Frank Ramsak, Robert Fenk and Martin Zirkel. The team work was and is still great. We had days of discussions that resulted in various publications and finally in this thesis. I also want to thank all other people at FORWISS that I had contact with.

Further, I thank all project partners of the EDITH project for their input and discussions and also very nice time spending in different cities all over Europe.

I thank my family, i.e., my mother and my sister. They did not urge me too much to finish the thesis.

Last but not least I thank all my friends, especially all Corpsbrüder of Corps Alemannia who always were willing to have a great free time after working periods, and all girls that distracted me from the work and prevented me from getting frustrated.

Abstract:

Efficient star query processing is crucial for a performant data warehouse (DW) implementation and much work is available on physical optimization (e.g., indexing and schema design) and logical optimization (e.g., pre-aggregated materialized views with query rewriting). Organizing fact tables with clustering multidimensional access methods (like the UB-Tree) are a promising approach to speed up star queries. However, the implementation into commercial products has not been done so far, since in addition to the clustering index organization, many parts of a database management system must be extended. For example, the query optimizer with corresponding cost model modifications must support the new organization and various optimization topics.

In this thesis, we present EHC, the Encoding for Hierarchical Clustering in combination with UB-Trees. EHC enables the use of clustering index structures also for hierarchical data. EHC is extended to MHC, the multidimensional hierarchical clustering by combining multiple dimensions. Based on the concept of MHC, we develop a number of query optimization algorithms, in order to support hierarchical clustering with query processing. For this purpose, we present a complete abstract processing plan that captures all necessary steps in evaluating star queries in these environments. One important step in the query processing phase is, however, still a bottleneck: the residual join of results from the fact table with the dimension tables in combination with grouping and aggregation. This phase typically consumes between 50% and 80% of the overall processing time. In typical data warehouse scenarios pre-grouping methods only have a limited effect as the grouping is usually specified on the hierarchy levels of the dimension tables and not on the fact table itself. Therefore, we suggest a combination of hierarchical clustering and pre-grouping. Exploiting hierarchy semantics for the pre-grouping of fact table result tuples is several times faster than conventional query processing. The reason for this is that hierarchical pre-grouping reduces the number of join operations significantly. With this method even queries covering a large part of the fact table can be executed within a time span acceptable for interactive query processing.

All these concepts have been implemented during this thesis into the commercial database management system Transbase® Hypercube and already run productive at a couple of customers of Transaction Software GmbH.

During the implementation further problems occurred, like complex aggregate expressions, multiple query boxes, non-clustering dimensions, complex schemata, multi-fact-table-joins etc. For these problems, solutions are described and have been implemented.

We further address some theoretical aspects of multiple hierarchies and dynamic changes of surrogates and a complete hierarchy model.

Finally, we present measurement results of a complex real-world sales transaction data warehouse of an electronic retailer and of the APB standard benchmark for OLAP. These measurements show the benefit of the implemented methods compared to conventional state of the art techniques and database management systems.

Table of Contents

1 Introduction

In the last decades, hardware has been improved, in order to solve current problems. The arguments of large database management system (DBMS) manufacturers are to buy better hardware, faster CPU's etc. when conventional tuning is not sufficient. This leads to additional investment costs and makes IT infrastructure more expensive. Not many algorithmic improvements have been implemented into commercial DBMS, although research is still going on. Implementing new technologies into existing DBMS is very expensive, has a long introduction and user acceptance time.

In addition, the large DBMS vendors modified their license policies and models recently. As consequence, the customers have to pay much more for the same benefit of their DBMS environment.

On the other side it is very difficult for a small DBMS company to compete with the "three large" in the market. You must have good arguments in order to convince customers to buy an unknown DBMS!

The challenge is to sell a product pasing the very high market barriers and being successful. Arguments for buying a new DBMS are for example:

- leading-edge technology
- cheap prices
- excellent support
- stability of the DBMS vendor

Thus, combining research and industry, i.e., implementing new technologies into commercial products serves at least one of the arguments.

1.1 Objective

Transaction Software decided to compete in the data warehouse market. For this purpose, the UB-Tree has been implemented as multidimensional index technique. However, in addition to a multidimensional index we need hierarchical encoding methods, in order to support hierarchical organized dimensions for data warehouses. This so called MHC, i.e., multidimensional hierarchical clustering, combines the strengths of a multidimensional clustering index with the concept of hierarchically structured dimensions for example in data warehouse applications.

The integration of MHC into Transbase® Hypercube, the relational DBMS of Transaction Software with the integrated UB-Tree, requires modifications of various layers in the DBMS. New query processing concepts and optimizing strategies must be added and transparency and applicability for the usage must be provided.

In this thesis, we describe

- a general hierarchy model,
- the concepts of an encoding for hierarchical clustering,
- multidimensional hierarchical clustering,
- the integration of MHC into the Transbase® kernel,
- query processing with MHC organized DW schemata,
- problems and solutions originating of these extensions (e.g., handling of large number of query boxes),
- optimizer extensions (considering MHC, special grouping techniques, complex aggregates, complex schema design etc.),
- extensions for the user of the DBMS, in order to design and maintain MHC organized schemata,
- dynamic aspects in combination with MHC

We cover the complete spectrum of implementing MHC into the Transbase® DBMS kernel, the problems and solutions.

1.2 Structure of the Thesis

Now we give an overview of the thesis in combination with hints for the reader.

The thesis consists of two main parts. Part I describes basic concepts about hierarchies (Section 2.2), dimensions, the design and modeling of dimensions and data warehouse schemata (Sections 2.3 and 2.4), the UB-Tree (Section 2.5), and some aspects of Transbase® Hypercube (Section 3).

The second part is the main part and contains the description of the encoding methods (Section 5), the encoding in combination with DW (6) and the integration into Transbase® Hypercube (Section 7). Section 8 contains all aspects how to compute and maintain the hierarchy encoding and use them in the context of DW. We especially address all practical issues to load and update DW applications.

The most important sections are Sections 9 and 10. These sections contain the complete query processing and optimizing concepts. In particular, Section 9 describes a framework of processing star queries with MHC with an abstract execution plan (Section 9.3). We describe the novel concept of hierarchical pre-grouping (Section 9.4) and usage of aggregation. Section 9.5 contains further aspects w.r.t. complex schemata, i.e., non-clustering dimensions. In Section 10, we describe the complete concept of optimizing star queries regarding MHC organized schemata. Section 10.1 contains the recognition of such schemata and how to handle them. Section 10.2 addresses very practical problems with aggregation, especially complex aggregations in star queries. In Section 10.3 we describe solutions for performance problems originating from a large number of multidimensional query boxes. Section 10.4 investigates the space partitioning of MHC organized data and explanations about the applicability of MHC. Section 10.5 covers also practical issues when dealing with DW schemata where not all dimensions can be used for clustering the fact table. Similar to this, we give a solution for schemata with multiple hierarchies on single dimensions in Section 10.6. Section 10.7 contains solutions how to handle schemata with several fact tables.

The remaining sections describe further concepts. Section 11 is a more theoretical chapter how to deal with complex hierarchies and Section 12 describes dynamic aspects and how to solve problems arising in this context.

Finally, we summarize the work and point out what to do in the future, since there is still work to do for a complete product suite of Transbase® Hypercube with MHC.

Part I – Preliminary Considerations

2 Terminology and Basic Concepts

2.1 Data Warehouse

Data warehouses (DW) got very popular in the last years and are expected to have great influence on companies in the future. The definition of DW, however, has not yet been standardized. Many different views on DW and data warehouse systems (DWS) are discussed in the DW community.

2.1.1 Classic Definition

One of the first definitions of a DW has been published by Inmon ([Inm96]): "A data warehouse is a subject oriented, integrated, non-volatile, and time variant collection of data in support of management's decisions."

This definition, however, is incomplete for today's DWs. According to Inmon, four basic properties are used for decisions:

- Subject Orientation: A DW must serve a special application goal, not fulfill a special task. The application goal usually is more complex and general than a task (e.g. human resources).
- Integration: The data stored in a DW usually comes from several databases. They are integrated into one large DW database.
- Non-volatile Data: Data stored in a DW will not be deleted or updated.
- Historic Data: DW applications analyze data with respect to the time. This time usually is historic, i.e. a rather long time span.

2.1.2 Modern Definition

Inmon's definition is too restrictive for the requirements of companies. Thus, we use a more general view of DW ([GB+01]).

A data warehouse is a physical database with an integrated view onto arbitrary data stored in the DW. The main applications on DWs are analyses of the data. For this purpose, a multidimensional view enables complex analysis functionality and allows interactive, explorative data analysis (OLAP, i.e., OnLine Analytical Processing). The main users of a DW are controllers and managers that have to deal with statistics, trends etc. Data stored in a DW usually are historic data, which means, that data is not updated or deleted from the DW. Often aggregated and consolidated data is extracted from operative systems (OLTP, i.e., OnLine Transaction Processing). Such data is extracted by ETL tools (i.e., Extraction, Transformation, Loading).

A system that integrates ETL tools, the DW and report and analysis functionality is called a data warehouse system (DWS). Usually a DW is combined with such tools.

Data often is loaded periodically into the DW, e.g. every day, every weekend. Thus it is often necessary to provide efficient mass loading functionality to load a large amount of data within a time window into the DW. Sometimes loading is required without a time window. Then, data consistency must be ensured.

2.1.3 Dimensions and Measures

The data stored in a DW, is organized multidimensionally. We distinguish between dimension and measure data.

Dimension attributes provide categorical (qualitative) data (e.g., products, customers, time), which determine the context of the measures. In most cases hierarchies are defined on dimensions (see Section 2.2). The time dimension often consists of the hierarchy *all-year-month-day* or *all-year-quarter-week-day*, where *all* represents all dimension elements. In most cases, dimensions are declared by a context (e.g., by the business context, organization structure). Usually, dimensions are almost static (e.g. a change on a geographic hierarchy, where Germany belongs to North America will seldom occur). However, in some applications frequent changes are possible. Special semantics are necessary to handle changes and data concerned by the changes([Kim96]). Sometimes even the schema of dimensions change (so called schema evolution, [Bla00]).

Measure attributes are numeric (quantitative) data (e.g., sales, cost, turnover) that are organized by multiple dimensions. In real multidimensional systems, dimension and measure attributes often are not distinguished.

2.2 Hierarchies

In data warehouses, the modeled business context can be very complex. The dimensions often represent very complicated and flexible relationships. We therefore need a hierarchy model that fulfills all possible hierarchical relationships.

Graphs represent relationships between vertices. Members in hierarchies are classified by relationships (usually *1:n* relationships), which in the following we call *hierarchical relationships*. These hierarchical relationships can be represented as directed acyclic graphs. A hierarchy instance is the actual instantiation of the hierarchical relationship. A special case of a hierarchy instance is a hierarchy tree. In this thesis, 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 DW community, some formal models of DW, dimensions, hierarchies etc. have already been worked out. Some approaches do not explicitly include hierarchical classification in their data model ([AGS97], [BPT97]). In [Sap01], [Leh98a] and [Alb01] the authors provide a hierarchical classification, defining hierarchy schemata with classify-relationships. In [LW96], a MD model based on relational elements is discussed.

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.

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

In this thesis, we describe a hierarchy model that is specified by the instance (in contrast to e.g., [Sap01]). We further present a linearization method for complex hierarchies by transforming complex hierarchies to simple hierarchies (Section 11).

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 $TE: V \times V \times N \rightarrow N$ that returns the type of an edge *e*: $TE(e) = TE((v_1, v_2)^t) = t$.

2.2.1 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 $\exists t\ (u, v)^t \in E$. Each vertex *u* has at most one out edge with type *t*, i.e., there are no two vertices v_1 and v_2 adjacent to *u* that have the same type *t*.

Example 2-1 (Graph):

Figure 2-1 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 $E=\{(A1, Aldi_N), (Aldi_N, North), \ldots, (West, Austria)\}$. □

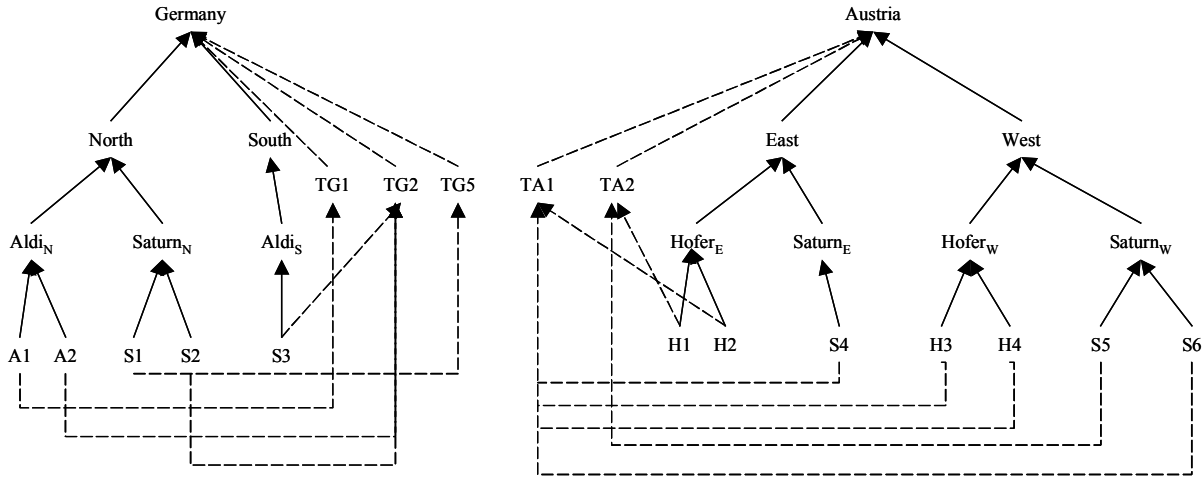


Figure 2-1: Directed Acyclic Graph

Definition 2-1 (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 $T: (E \times .. \times E)^t \rightarrow N$ returns the type:

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

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: $T(\phi_1) = T(\phi_2) = t$ and $t \neq \perp$.
The *pathlength* is the number of edges in path ϕ. $pathlength(\phi^t) = pathlength(path(u, v)^t)$, if $\phi^t = u \xrightarrow{\Phi} v$ and $path(u,v)^t$ is the path ϕ from *u* to *v* with type *t*. □

Note that a path always specifies a sequence of vertices, i.e., $\phi = (v_1, v_2, \ldots, v_n)$. If there is more than one path from u to v, each of the paths has a path length.

Note that the type of a path is only defined, if all edges in the path have the same type.

Example 2-2 (Path, pathlength):

We use Figure 2-2 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 $T(\phi_2) = 2$.

The path ϕ_3 *("S1", "Saturn$_N$", "North", "Germany", "Segment")* has the same type as ϕ_1, because all edges of ϕ_1 and ϕ_3 have the same type, but ϕ_3 has a different type compared to ϕ_2. □

Definition 2-2 (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 $r \notin V_1 \cup V_2$ 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: $G=(V_1 \cup V_2 \cup r, E_1 \cup E_2 \cup (r_{G1}, r) \cup (r_{G1}, r))$. Note that the direction G_1 and G_2 must be the same such that the union is a rooted tDAG again.

The graph of Figure 2-1 consists of two rooted tDAGs G_1 and G_2 with the roots *Germany* for G_1 and *Austria* for G_2. G_1 and G_2 are connected via an additional vertex *Segment* and the edges (*Germany, Segment*) and (*Austria, Segment*). $G=G_1 \cup G_2$ now is a rooted tDAG (Figure 2-2). In Figure 2-2, we add edges of type 1 and type 2.

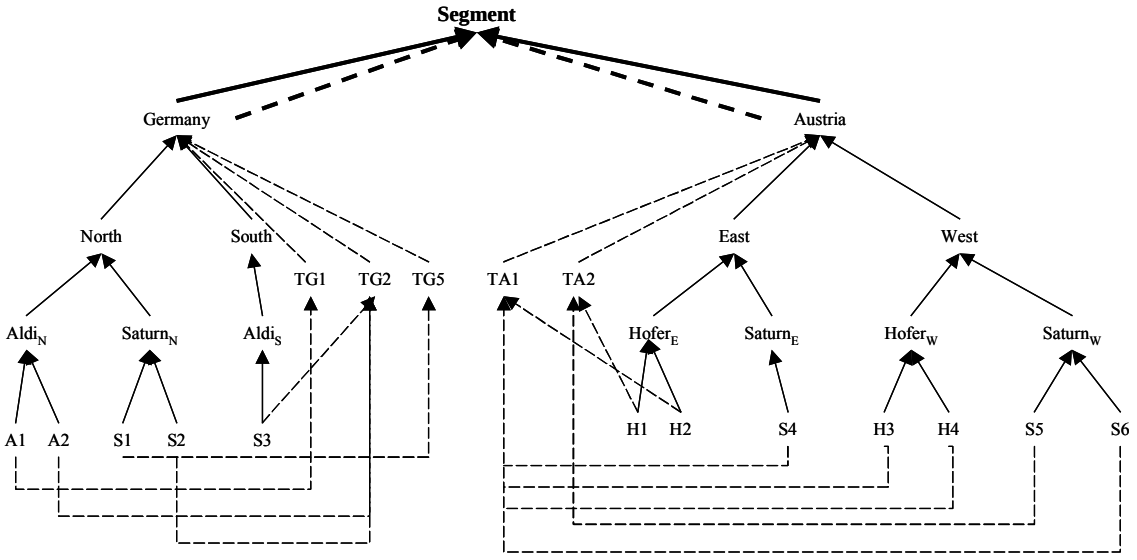


Figure 2-2: Rooted Directed Acyclic Graph

Definition 2-3 (Outdegree, Indegree, Degree):

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 2-2, 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 $outdegree^t(v)=1$. This requirement means that we consider trees as explained later.

Example 2-3 (Indegree, Outdegree, Degree):

In Figure 2-1, the vertex $Saturn_N$ has the following degrees:
$indegree(Saturn_N) = indegree^1(Saturn_N) = 2$, $outdegree(Saturn_N)=1$, $degree(Saturn_N)=3$.

The vertex *Germany* has the following degrees:
$indegree(Germany)=5$, $indegree^1(Germany)=2$, $indegree^2(Germany)=3$, $outdegree(Germany)=0$, $degree(Germany)=5$. *Germany* is a root vertex.

The vertex *S1* has the following degrees:
$indegree(S1)=0$, $outdegree(S1)=2$, $outdegree^1(S1)=1$, $outdegree^2(S1)=1$, $degree(S_1)=2$. *S1* is a leaf with two outgoing typed edges (type 1 and 2). □

In the following, we discuss rooted tDAGs, where every path from the leaves to the root has a defined type, i.e. every edge (v_i, v_j) of the path has the same type t. Every vertex v_i and v_j has at most one outgoing edge of type t, i.e., $outdegree^{tm}(v_i) = outdegree^{tn}(v_j) = 1$, $t_m \neq t_n$.

Definition 2-4 (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*: $G'=(V', E'), V' \subseteq V, E' \subseteq E$. □

Definition 2-5 (Simple tDAG):

A *simple tDAG* (abbreviated as *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 $pathlength(\phi^t_i) = pathlength(\phi^t_j)$, i.e., all paths from leaves to root with same type and length.

□

In real applications, we often have more complex graphs, e.g., hierarchies in data warehouses. See Sections 2.2.4 and 11 for more details.

Theorem 2-1:

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 $e_i = (root(T_j), root(T))$, $1 \leq j \leq m$ and T_j are the roots of the sub-trees of *T*. *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 type *t*. $outdegree^t(v_i)=1 = outdegree(v_i)$ (see Definition 2-5) 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.

[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.

2. A stDAG is a balanced tree:
   In a balanced tree, the height (i.e., the maximum pathlength of the path from all leaves to the *root*) of the subtrees 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.

For example, in Figure 2-2 the tDAG with type 1 is a simple tDAG. The length of every path from the leaves to root is the same.

Definition 2-6 (Equivalence Class):

An *equivalence class* is a set of vertices with the following properties: Two vertices *u, v* of a stDAG $T^S=(V^S, E^S)$, $u, v \in V^S$ 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). □

Note that the complete stDAG is one equivalence relation.

Example 2-4 (Simple tDAG, Equivalence Class):

In the graph of Figure 2-2, 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}.*

2.2.2 Hierarchies

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

Definition 2-7 (Hierarchy Instance):

A hierarchy instance *H* is a rooted tDAG *H=(V, E)* with vertices $m_i \in V$ and directed, typed edges $e_j \in E$. We call the vertices *members*. 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. The union of the simple hierarchies is the hierarchy instance *H:* $\bigcup_i H_i^S = H$. This follows from the definition of simple hierarchies. The simple hierarchies must have the same direction as already mentioned for Definition 2-2.

Definition 2-8 (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 2-5 (Hierarchy Instance, Simple Hierarchy, Hierarchy Level):

The graph illustrated in Figure 2-2, is a hierarchy instance with two simple hierarchies H_1 and H_2.

The levels of $H_1=(V_1, E_1)$ are $V_1= h_1^1 \cup h_2^1 \cup h_3^1 \cup h_4^1 \cup h_5^1$, 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}.*

The levels of $H_2=(V_2, E_2)$ are $V_2=\{h_1^2 \cup h_2^2 \cup h_3^2 \cup h_4^2)$, where h_1^2*={A1, A2, S1, S2, A3, H1, H2, S4, H3, H4, S5, S6},* h_2^2*={TG1, TG2, TG5, TA1, TA2},* h_3^2*={Germany, Austria} and* h_4^2*={Segment}.*

□

Definition 2-9 (Order of Levels):

The *order O* of a level h_i is the path length of one representant (member) m^i_j of h_i to the root *r*:
$O(h_i) = pathlength(path(m^i_j, r))$.
A level h_i is smaller (greater) than h_j, if $O(h_i) < O(h_j)$ $(O(h_i)>O(h_j))$.
The order of the root level is 0. □

Definition 2-10 (Hierarchically Dependent Members):

The member m_i is *hierarchically dependent* on m_j, if there is an edge $e = (m_j, m_i) \in E$. □

Definition 2-11 (Hierarchically Dependent Levels):

A level h_i is *hierarchically dependent* on h_j, if all members $m^i_k \in h_i$ are hierarchically dependent on members $m^j_h \in h_j$, i.e., $\forall m^i_k \in h_i \; \exists m^j_h \in h_j$: $(m^j_h, m^i_k) \in E$. The function *HD* 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 directly 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 $m^i_h \in h_i$. □

We define the order of hierarchy levels from bottom to top. 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 (by level *state*).

Example 2-6 (Order of Levels, hierarchically dependent Levels):

According to Example 2-5, the order of levels $O(h_1^1)=4$, $O(h_2^1)=3$, $O(h_3^1)=2$, $O(h_4^1)=1$, $O(h_5^1)=0$.

$HD(H_1)$ returns the following hierarchical dependencies: $HD(H_1) = \{(h_1^1, h_2^1), (h_2^1, h_3^1), (h_3^1, h_4^1), (h_4^1, h_5^1)\}$. □

Definition 2-12 (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*.
We distinguish several qualities of shared levels:

1. $h_1 \cap h_2 \neq \varnothing$
2. $h_1 \subset h_2$
3. $h_1 = h_2$

The first case requires, that h_1 and h_2 have at least one common member. Thus, if h_1 and h_2 are shared and h_2 and h_3 are shared, h_1 and h_3 may be disjoint levels (i.e., no transitivity)
In the second case, h_2 contains h_1. Thus, if $h_1 \subset h_2$ and $h_2 \subset h_3$, then $h_1 \subset h_3$, and h_1 and h_3 are also shared levels.
The third case is the strongest quality: h_1 and h_2 are *equal levels*, i.e., the set of members of h_1 is equal to the set of members of h_2. If $h_1 = h_2$ and $h_2 = h_3$, then h_1 and h_3 are also shared levels, even equal levels. □

Definition 2-13 (Distinct):

The operator *distinct:* $L \rightarrow L$ 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 not 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 2-7 (Shared Level, Distinct Operator):

According to Example 2-5, 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 operator *distinct* 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). In order to implement the operator, we check compare the members of the hierarchy levels. If all members of two hierarchy levels are the same, the hierarchy, one hierarchy level is returned.

Definition 2-14 (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 a simple hierarchy is a balanced tree having only one leaf level. □

Definition 2-15 (Hierarchy Schema):

The *hierarchy schema HS* is a rooted tDAG specified by $HS=(L^S, E^S)$ where L^S 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.
A hierarchy schema is represented by a rooted DAG. □

Definition 2-16 (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 2-8 (Hierarchy Schema and Instance):

In Figure 2-3, 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 2-5. $H = H_1 \cup H_2 = (V, E^H)$, 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 2-2. □

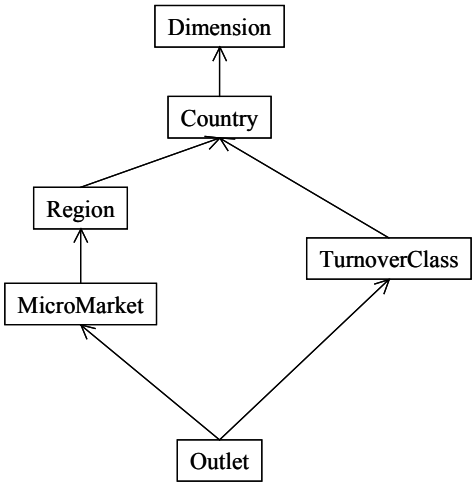


Figure 2-3: Hierarchy Schema of a Complex Hierarchy

2.2.3 Conclusion

With the introduced concept of defining hierarchies from the instance of the hierarchy graph, we are free to represent any kind of hierarchy (unbalanced, complex, overlapped etc.). The described operators provide a complete toolset, in order to model any hierarchical relationships of existing hierarchical dimensions.

For our investigation and implementation of multidimensional hierarchical clustering, however, we have to normalize degenerated hierarchies (see Section 6).

2.2.4 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. These additional attributes are called *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 set 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 granularity[3] (leaf hierarchy levels), new aggregation and grouping semantics have to be introduced ([Leh98a]). Our hierarchy model, however, supports such degenerated hierarchies. The same holds for unbalanced hierarchies. The paths (with different lengths) can be modeled, but for a reasonable representation in a relational DBMS, the hierarchy instances must be normalized. This normalization usually consists of introducing artificial hierarchy members,

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

in order to get hierarchy paths of equal lengths. Especially for the EHC and MHC modeling, this normalization is essential (see Section 6 for a description of the logical and physical modeling).

2.3 Star Schema

In relational DBMS, the schema of a data warehouse usually is represented by a star schema or snowflake schema (see Section 2.4). This section contains the description of the concept of star schemata.

A star schema combines the concepts of a relational DBMS with the multidimensional view to multidimensional data stored in a relational DBMS. This method is widely used for *relational OLAP* systems (*ROLAP*). Instead of storing all data (dimension attributes and measure attributes) in one single table, the *fact table FT*, dimension data is normalized out of the fact table into dimension tables. In contrast to the snowflake schema, dimension data is stored in one single dimension table for every dimension. Figure 2-4 illustrates a star schema with n dimensions, expressed by the dimension attributes $d_1, \ldots, d_n$ of FT and the corresponding dimension tables $D_1, \ldots, D_n$.

The center of the "star" is the fact table. The attributes d_i are the dimension attributes, the attributes m_j are measure attributes. Every d_i corresponds to $D_i.h_1$, i.e. the leaf level of the hierarchies in D_i. The (shared) leaf level of the hierarchy is the key of the dimension table. We require this key constraint due to two reasons: First, the composite key of the complete hierarchy path $(h_1, h_2, \ldots, h_m)$ would be too long to store also in the fact table for every dimension (because of the foreign key relationship of the dimension attributes in the fact table). The leaf level often is an artificial (short) key due to space saving reasons. Second, if we have more than one hierarchy, the shared leaf level h_1 for all hierarchies serves a common key.

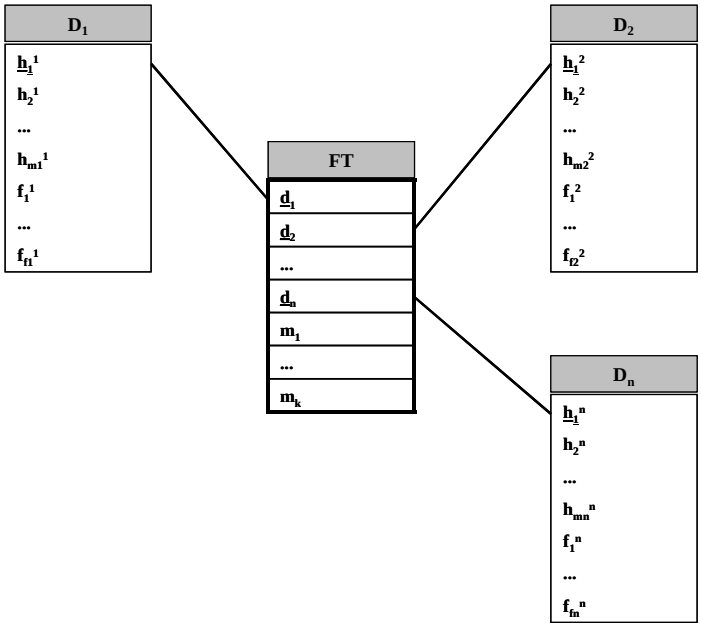


Figure 2-4: Star Schema

In addition to the hierarchy levels in D_i, feature attributes describe the dimension attributes and hierarchy attributes in more detail. Feature attributes can be assigned to each level in the hierarchy and thus may be stored redundantly in the dimension tables. Often the hierarchy levels are artificial keys used for hierarchical dependencies and contain an additional description field (modeled as feature).

In general more than one hierarchy is allowed per dimension. Storing data in a star schema will always guarantee correct hierarchies, because every tuple in the dimension table contains the full path of all hierarchies on that dimension. This leads to more space than in a snowflake schema, but will not

require a separate join for every hierarchy level. In this thesis, we call dimension tables of a star schema *flat dimensions*, where the complete dimension with hierarchies and features is stored within one dimension table.

Definition 2-17 (Flat Dimension):

A *flat dimension* table contains all dimension attributes, i.e., the hierarchy level attributes of all hierarchies and their feature attributes. Thus the hierarchical relationship is modeled within this table by storing the complete path within the tuple. □

In Figure 2-5 we show an example for a star schema with one fact table *FACT* and three dimension tables *Product*, *Segment*, and *Time*. The dimension key attributes *Item*, *Outlet*, and *Day* are the primary key of the fact table. The leaf levels of the hierarchies, i.e., *Item* for *Product* dimension, *Outlet* for *Segment* dimension, and *Day* for *Time* dimension are the primary key of the corresponding dimension tables.

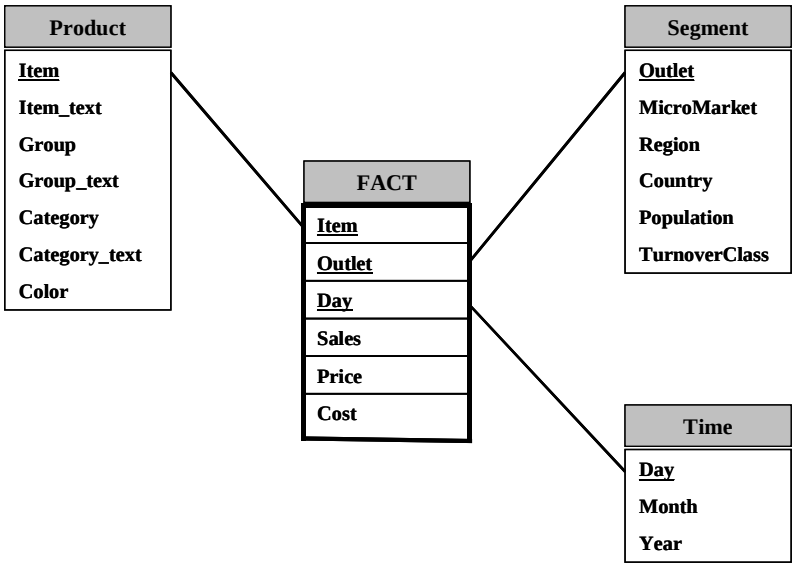


Figure 2-5: Sample Star Schema

2.4 Snowflake Schema

A *snowflake schema* is the standard and more general schema used in real data warehouses. Designing applications with a snowflake schema provides complex hierarchical relationships, common use of hierarchies for many applications, flexibility for queries and less requirements for data space. However, in general, the performance of queries will suffer from the more complex schema, especially the number of joins is higher than for a star schema. The DBMS must provide special methods, in order to efficiently support snowflake schemata. In this thesis, we design and implement our algorithms for both, star and snowflake schemata.

The snowflake schema is an "extended" star schema in the meaning, that there are several hierarchy tables for every dimension – a kind of normalization (see Figure 2-6). There are many different types of snowflake schemata depending on the normalization. We describe *completely normalized* and *partial normalized* snowflake schemata.

As in the star schema, the fact table *FT* is the center of the "snowflake" with the dimensions surrounding *FT*. Every dimension attribute d_i of *FT* has a foreign key relation to the primary key of the dimension table $D_i^1.h_1^i$. In Figure 2-6, we assume one hierarchy per dimension (in general, an arbitrary number of hierarchies per dimension is possible), where the key of the hierarchy is the leaf

level h_1. Thus, the following relationship is required: $FT.d_i = D_i^1.h_1$. We call the dimension table D_i^1 the *leaf dimension table LDT*.

The higher dimension tables, i.e., the normalized dimension tables, contain one or more hierarchy levels. Basically, each dimension table contains an arbitrary number of hierarchy levels (depending on the kind of normalization). There must be a foreign key relationship between a lower dimension table D_i^k and a higher dimension table D_i^{k+1}:

$(D_i^k.h_j^i,\ D_i^k.h_{j+1}^i,\ \ldots,\ D_i^k.h_{j+m}^i) \rightarrow (D_i^{k+1}.h_j^i,\ D_i^{k+1}.h_{j+1}^i,\ \ldots,\ D_i^{k+1}.h_{j+m}^i)$

This foreign key relationship requires that the attributes $(D_i^{k+1}.h_j^i,\ D_i^{k+1}.h_{j+1}^i,\ \ldots,\ D_i^{k+1}.h_{j+m}^i)$ are primary key of dimension table D_i^{k+1}.

Each dimension table contains an arbitrary number of feature attributes, usually related to the hierarchy levels $(h_j^i,\ h_{j+1}^i,\ \ldots,\ h_{j+m}^i)$.

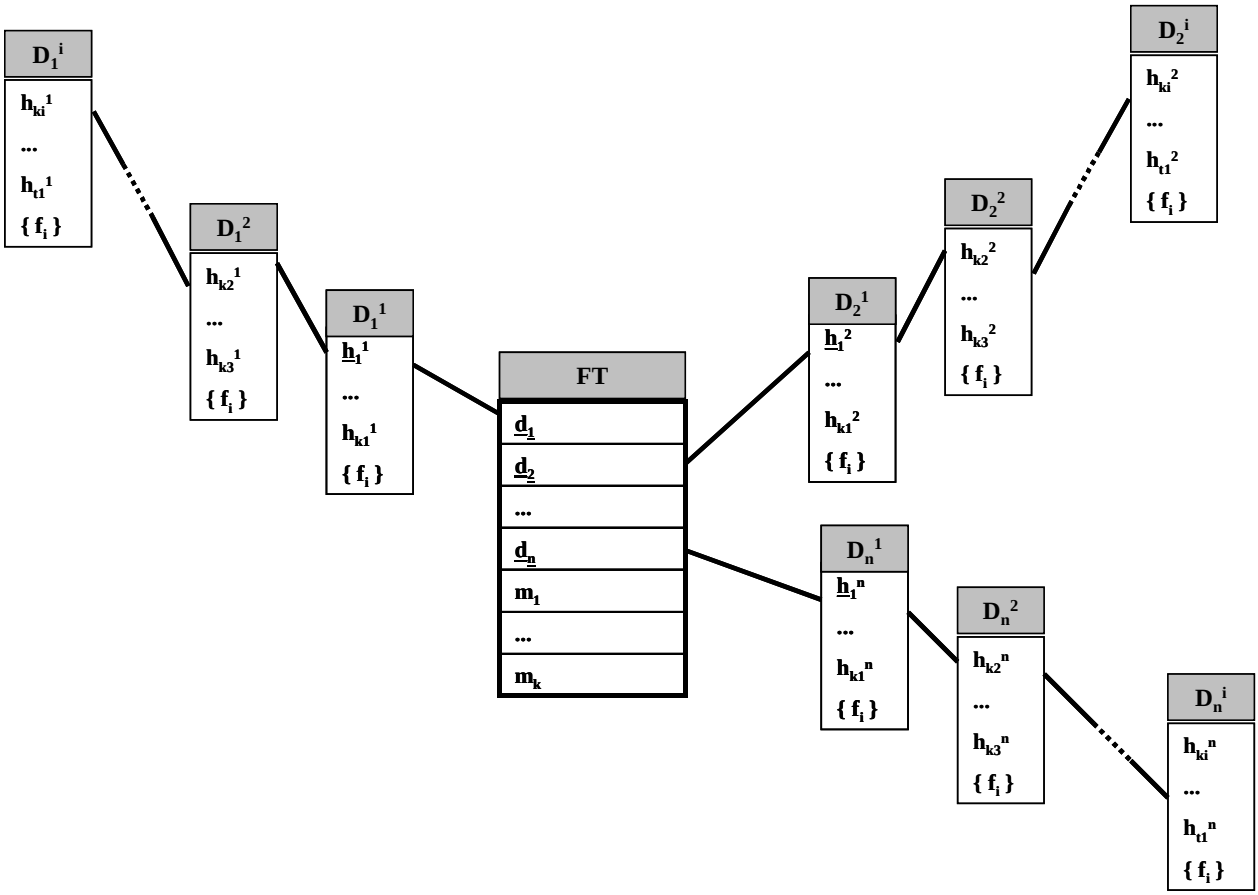


Figure 2-6: Snowflake Schema

Definition 2-18 (Leaf Dimension Table, LDT):

A *leaf dimension table* LDT of a dimension contains the key of the dimension, i.e., the dimension key attribute of the corresponding dimension in the fact table. This attribute is the leaf level for every hierarchy on the dimension. A LDT contains one or more foreign keys to *hierarchy level tables* and feature attributes. □

Definition 2-19 (Hierarchy Level Table):

A *hierarchy level table* contains a number of hierarchy levels of one hierarchy and a foreign key to the next (higher) hierarchy level. Additionally, feature attributes are stored in the hierarchy level table. □

In Figure 2-7 we show an example for a snowflake schema with a fact table *FACT* and three dimensions *Product*, *Segment* and *Time*. The dimensions *Product* and *Segment* are normalized into snowflake dimensions w.r.t. the feature attributes of the hierarchy levels. In the *Segment* hierarchy, the *Region* level has no feature attribute and no separate table is used for this hierarchy level. The *Time* dimension is modelled as a flat dimension.

For the sample snowflake schema, the foreign key relationships of Table 2-1are necessary:

Dimension Product	Dimension Segment	Dimension Time
`FACT(Item) → Product(Item)`	`FACT(Outlet) → Segment(Outlet)`	`FACT(Day) → Time(Day)`
`Product(Group) → Product_Group(Group)`	`Segment(MicroMarket → Segment_MM(Micromarket)`	
`Product_Group(Category) → Product_Cat (Category)`	`Segment_MM(Country) → Segment_Country(Country)`	

Table 2-1: Foreign Key Relationships for the Sample Snowflake Schema

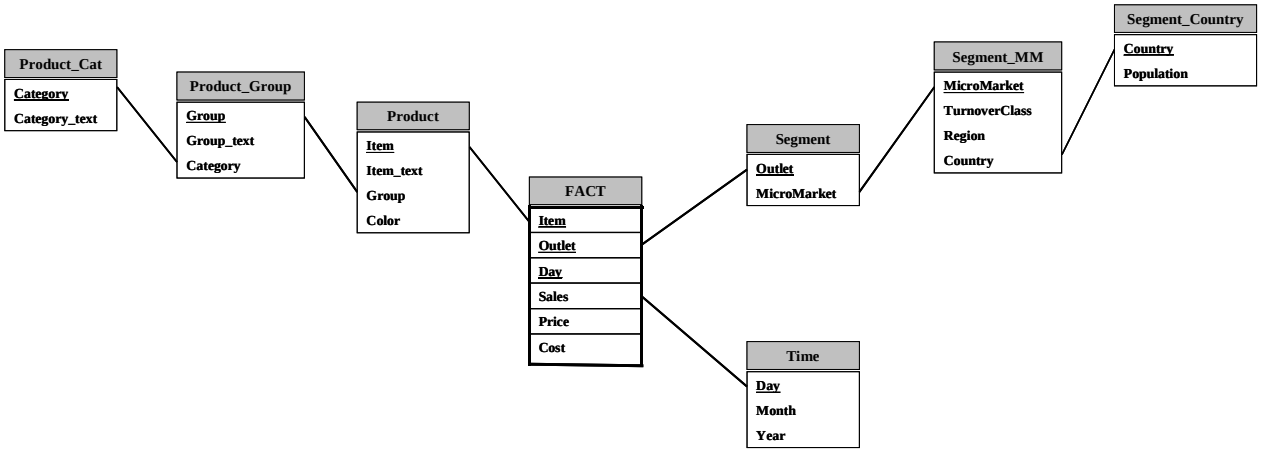


Figure 2-7: Sample Snowflake Schema

2.4.1 Normalization of Dimensions

Normalization of dimensions is not equivalent to classic normalization of relations. We talk about normalization, if a star schema is extended to a snowflake schema in the meaning, that hierarchy levels are stored in separate tables. On the one hand, normalization will save disk space, if feature attributes of higher levels are stored in a "higher" dimension level table and not redundant in the leaf dimension table. On the other hand, the schema can express hierarchical relationships that enable optimizing query processing, such as using properties of multidimensional hierarchical clustering. Additionally, maintenance of dimensions will be easier and more intuitive.

Many DBMS prefer star schemata over snowflake schemata, because query processing is easier and more performant. In a star schema, the number of joins is reduced to the dimension tables with the fact table. In a snowflake schema, the number of joins is increased, because each dimension can consist of several tables that must be joined for query processing. We prefer snowflake schemata, because more hierarchical relationships can be expressed in snowflake schemata. Consider a hierarchy with feature attributes. In a star schema, the feature attributes are stored in the dimension table and it is not clear to

which hierarchy level the feature attributes belong[4]. A predicate on a feature attribute is not mapped to a corresponding restriction on the hierarchy attribute and therefore cannot be optimized w.r.t. query processing with MHC (see Section 9).

For efficient query processing, we require some prerequisites for normalized dimensions, i.e., a *well-formed snowflake* schema. A well-formed snowflake schema can increase the query performance, because in snowflake schemata, hierarchical relationships between feature and hierarchy attributes can be expressed. A well-formed snowflake schema is a special type of snowflake schema which is described in the following.

Definition 2-20 (well-formed Snowflake Schema):

A well-formed snowflake schema consists of a fact table with the dimension key attributes $(d_1, d_2, \ldots, d_n)$ as primary key and a leaf dimension table D_i^1 for every dimension. The leaf dimension tables contain all hierarchy levels $h_1^i, h_2^i, \ldots, h_t^i$ for all hierarchies of the dimension. The higher dimension tables are connected with the leaf dimension table via foreign key relationships. Between every higher dimension table D_i^k, a foreign key relationship with D_i^{k+1} exists. □

We require the leaf dimension table with all hierarchy levels for an efficient computation of compound surrogates (see Section 5.3). Conceptually, the leaf dimension table is a view over a general snowflake dimension with the dimension tables $D_1, D_2, \ldots, D_k$:

```
CREATE VIEW D^leaf (h^1, h^2, …, h^t) AS
SELECT D^1.h^1, D^1.h^2, … D^1.h^j1, D^2.h^j1+1, D^2.h^j1+2, …, D^2.h^j2, …, D^k.h^jk-1+1,
…, D^k.h^t
FROM D^1, D^2, …, D^k
WHERE D^1.h^j1+1 = D^2.h^j1+1 AND D^1.h^j1+2 = D^2.h^j1+2 AND … AND D^1.h^j1+m = D^2.h^j1+m
AND … AND D^k-1.h^jk = D^k.h^jk AND … AND D^k-1.h^jk+o = D^k.h^jk+o
```

Thus, the hierarchy is de-normalized into the leaf dimension table defining the join over the complete hierarchy.

For the well-formed snowflake schema of Figure 2-7, the leaf dimension view for the *Product* dimension is the following:

```
CREATE VIEW DimProduct_leaf (Item, Group, Category) AS
SELECT
  DP_item.Item, DP_group.Group, DP_category.Category
FROM
  Product DP_item, Product_Group DP_group, Prodcut_Cat DP_cat
WHERE
  DP_item.Group = DP_group.Group AND DP_group.Category = DP_category.Category
```

An example for a well-formed snowflake schema are the *field normalized* and *path normalized* schemata described in the following sections. Different types of normalization can be combined within one snowflake schema, since normalization is a property of the dimension. Thus, one dimension can be organized as field normalized dimension, another as path normalized, and a third dimension can be organized as a different normalization.

For a field normalized and path normalized dimension we assume an LDT that contains all hierarchy level attributes of all hierarchies in the dimension. Thus the complete path of the hierarchies is stored in that LDT. In addition to the LDT, a number of hierarchy level tables define hierarchical dependencies. Without loss of generality, we assume one hierarchy for the dimension in the following sections.

[4] It is not possible to express functional dependencies between attributes within one table by standard SQL.

2.4.2 Field Normalized Dimensions

In a *field normalized dimension (FND)*, the leaf dimension table contains all hierarchy elements $h_1^i, \ldots, h_t^i$ of the corresponding hierarchy. The key of the LDT is h_1^i. For some or all h_j^i $(2 \le j \le t)$, a dimension level table D_i^j exists that contains the hierarchy level attribute h_j^i and a number of feature attributes $\{ f_k \}$. The following foreign key relationships of the hierarchy attributes in the LDT D_i^1 to the corresponding attribute in the dimension level table exist: $D_i^1 (h_2^i) \rightarrow D_i^2 (h_2^i)$, $D_i^1 (h_3^i) \rightarrow D_i^3 (h_3^i)$, …, $D_i^1 (h_t^i) \rightarrow D_i^t (h_t^i)$ (see Figure 2-8). The key of the satellite dimension table D_i^j is the hierarchical attribute: $D_i^j.h_j^i$.

A separate hierarchy level table D_i^j makes sense, if a feature attribute is assigned to the hierarchy level j, otherwise no table D_i^j is necessary for the hierarchy attribute $D_i^j.h_j$.

To use a field normalized schema, the hierarchy attributes must be unique (in contrast to the path normalized schema). This means, that the value of a hierarchy attribute must be unique in that level. Otherwise a combination of levels is necessary to get the key for D_i^j!

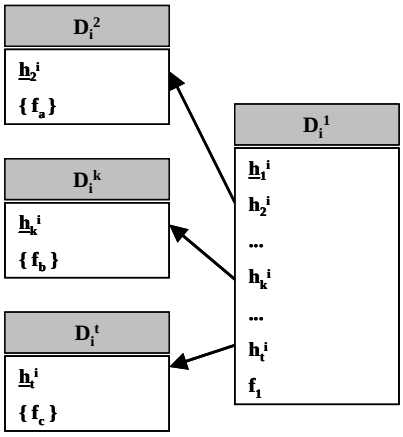


Figure 2-8: Field Normalized Dimension

2.4.3 Path Normalized Dimensions

In a *path normalized dimension (PND)*, the leaf dimension table D_i^1 contains all hierarchy attributes $h_1^i, \ldots, h_t^i$ of the corresponding hierarchy. Key is h_1^i.We additionally assume one hierarchy level table D_i^k for every hierarchy level k $(2 \le k \le t)$ that contains the path from the top of the hierarchy to the level k $(h_t^i, h_{t-1}^i, \ldots, h_k^i)$. Key of D_i^k can be the smallest hierarchy level h_k^i or a combination of hierarchy levels. Additionally a number of feature attributes $\{ f_k \}$ is stored in every D_i^k. (see Figure 2-9).

A PND additionally requires foreign key relationships for every hierarchy level table D_i^k to ensure a correct hierarchy: $D_i^k(h_t^i, h_{t-1}^i, \ldots, h_{k+1}^i) \rightarrow D_i^{k+1}(h_t^i, h_{t-1}^i, \ldots, h_{k+1}^i)$ for $2 \le k \le t-1$. Thus, the complete prefix path of levels $k+1$ to t must also exist in the next hierarchy level table. In order to define the foreign key relationship, we have to introduce corresponding unique indexes or define the primary key of the dimension tables correspondingly.

A PND can express hierarchical relationships between feature attributes and hierarchy levles, if the members in the hierarchy level are not identified uniquely.

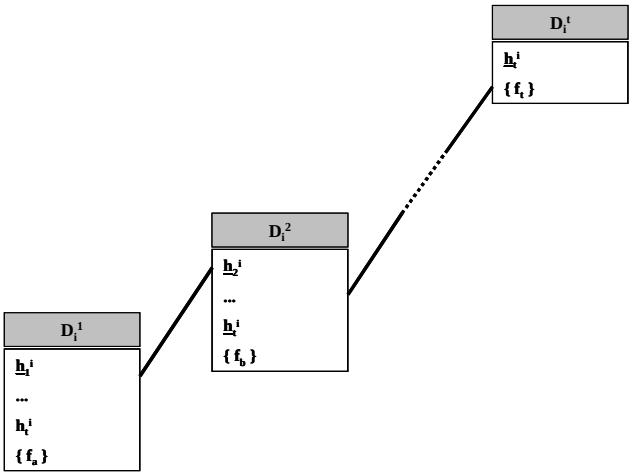


Figure 2-9: Path Normalized Dimension

In Figure 2-10, we show a path normalized dimension for the *Product* dimension of the sample schema. The leaf dimension table *Product* contains all hierarchy levels *Item*, *Group,* and *Category* and further feature attributes. The higher dimension table *Product_Group* contains the two hierarchy levels *Group* and *Category* and the highest dimension table *Product_Cat* contains the *Category* hierarchy level. We define the following foreign key relationships:

```
FACT(Item) → Product(Item)
Product(Group, Category) → Product_Group(Group, Category)
Product_Group(Category) → Product_Cat(Category)
```

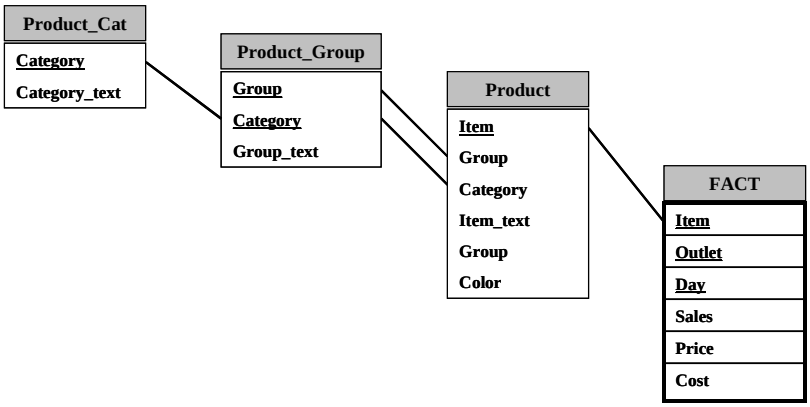
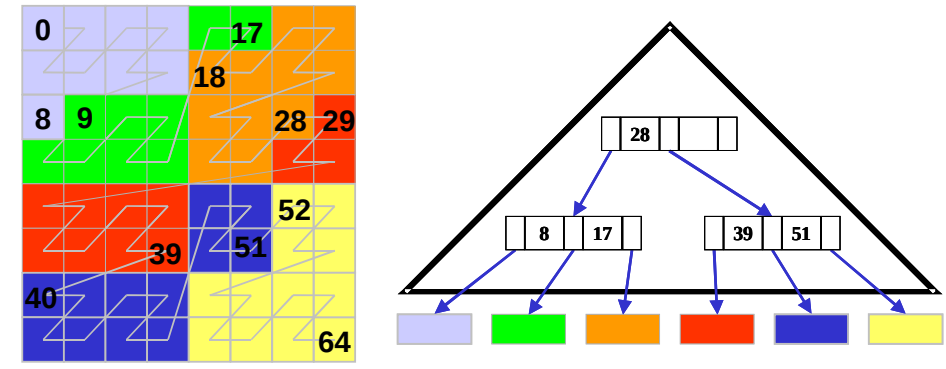


Figure 2-10: Sample Path Normalized Dimension

2.5 The UB-Tree

We just give a short introduction to UB-Trees here, details can be found in ([Bay97], [RMF+00], [Mar99]). The basic idea of the UB-Tree is to use a space-filling curve to map a multidimensional universe to one-dimensional space. Using the Z-curve for preserving multidimensional clustering it is a variant of the zkd-B-Tree [OM84]. A Z-address $\alpha = z(x)$ is the ordinal number of the key attributes of a tuple x on the z-curve, which can be efficiently computed by bit-interleaving. A standard B-Tree is used to index the tuples taking the Z-addresses of the tuples as keys. The pagination of the B-Tree creates a disjunctive partitioning of the multidimensional space into so-called z-regions. This allows for very efficient processing of multidimensional range queries.



Z-region partitioning of 2-D space

B^+-Tree with Z-values

Figure 2-11 UB-Tree: Z-region partitioning and underlying B-Tree

Figure 2-11 shows a z-region partitioning for a two-dimensional universe and the corresponding B-Tree. The interval limits of the z-regions are also depicted.

The processing of basic operations, i.e., insertion, deletion, update, and point query, of the UB-Tree are analogous to the basic operations of the B-Tree. For each tuple the corresponding z-address is computed, and with the resulting value the underlying B-Tree is accessed. Thus, all basic operations require only cost proportional to the height of the tree. The only recommendable modification to the standard B-Tree algorithms is an adaptation of the split algorithm to achieve a “good” (as rectangular as possible) z-region partitioning.

A UB-Tree is especially good in processing multidimensional range queries, as it only retrieves all z-regions that properly intersect the query box (see also Section 10.3.1). Consequently, it usually shows the nice property that the response time of the range query processing is proportional to the result set size.

3 Transbase® – a Relational Database Management System

Since most of the concepts are implemented in Transbase®, we now describe some basic concepts of the DBMS Transbase® Hypercube. Transbase® is a relational DBMS implementing the SQL-92 standard and supporting the ACID concept for all operations (DML and DDL). It further allows server client environments, multi-user access, and logging and recovery. The optimizer is a rule based optimizer supporting a large amount of heuristics. Queries are optimized statically, i.e., at compilation time.

The basic storage technique are B*-Trees ([Com79]). All tables are stored in B*-Trees w.r.t. the primary key of the table. Thus, Transbase® provides physical clustering of all tables. Additionally, secondary B*-Tree indexes can be created pointing to the primary B*-Tree (holding the table) via an indirection (IK-Tree, i.e., internal key tree).

On top of the primary B*-Trees, the UB-Tree was implemented with the z-value for the key of the B*-Tree ([RMF+00]). This extension resulted in the product Transbase® Hypercube. During this thesis, the MHC technique was implemented as described later.

For more information about Transbase® refer to [Tra01] and to later sections.

Part II – EHC Kernel Integration

In other commercial DBMS, multidimensional hierarchical clustering has not been implemented so far. We provide a description of an implementation of this subject into the relational DBMS Transbase® Hypercube ([Tra01]) using the UB-Tree as clustering multidimensional index.

4 Motivation for EHC

Clustering is a very important technique to increase performance of a DBMS. It reduces the number of disc accesses by placing tuples that are likely to be processed together close to each other. Thus, for query processing, the likelihood to access tuples already read from disk and cached in main memory is increased, because one disk access returns a number of tuples belonging to the result set of the query. The performance increase depends on some parameters such as clustering quality, query predicates, and page size. In the case of non-clustered data, one page only contains a small number of such tuples or even only one tuple contributing to the result set of a query.

Hierarchical clustering is a concept that is useful to store hierarchically organized data ([ZSL98]), e.g., hierarchies, as well as to store data characterized by hierarchical dimensions clustered with respect to the hierarchies ([MRB99], [KS01]).

In [ZSL98] the goal is to access subtrees of the hierarchy efficiently by organizing them with respect to hierarchical neighborhood. [MRB99] enables efficient access to data in fact tables with hierarchical organization in multiple dimensions for queries with hierarchical predicates on hierarchical dimensions. We use an encoding of the hierarchy paths to process such queries and use a space saving representation for hierarchical relationships. This *Encoding for Hierarchical Clustering (EHC)* is organized dynamically and in a space saving way.

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 an artificial unique id (*dimID*) is used as join attribute. This dimID can be a computed number with respect to EHC: $dimID=surr(v_m, v_{m-1}, \ldots, v_{leaf})$. The function *surr* computes a surrogate id for the path of the dimension tuple, $v_m, v_{m-1}, \ldots, v_{leaf}$ are the hierarchy members of the levels $h_m, h_{m-1}, \ldots, h_{leaf}$.

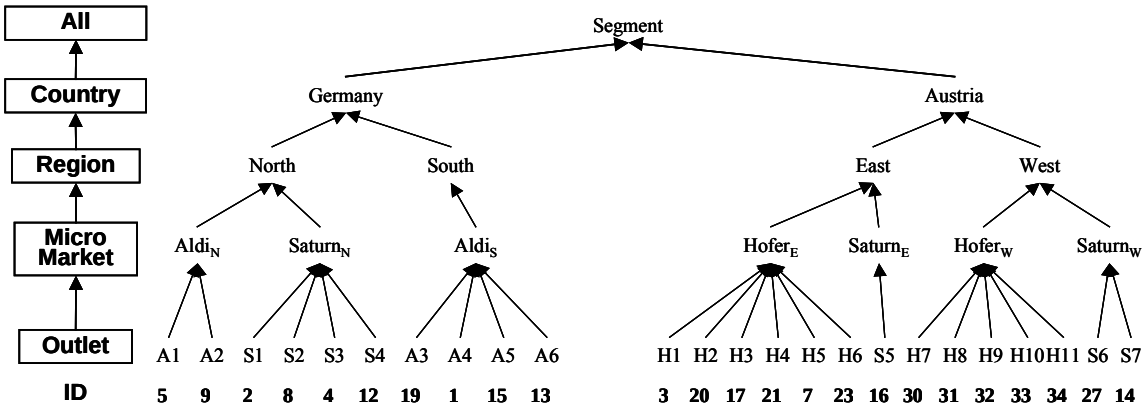


Figure 4-1: Hierarchy with artifical Surrogates without EHC

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). The predicate "Germany" of the hierarchy in Figure 4-1 would result in the leaf members *{"A1", "A2", "S1", "S2", "S3", "S4", "A3", "A4", "A5", "A6"}* or equivalently in a set of IDs *DID={5, 9, 2, 8, 5, 12, 19, 1, 15, 13}*, and every such member is a join predicate to the fact

table. If the surrogate numbers (dimID) are chosen w.r. to the hierarchical neighborhood, one interval can replace a set of dimID's for the join with the fact table (e.g., the predicate *"$Hofer_W$"* results in the dimID's *DID={30, 31, 32, 33, 34}*, equal to the interval [30 : 34]).

An equi-join strategy between fact and dimension table requires a large number of disk accesses, because for every element *did* $\in$ *DID* all join partners in the fact table have to be evaluated. With the use of secondary indexes, first a number of tuple identifiers (TID) is retrieved (sometimes via intersection methods, if multiple dimensions are involved), and for every TID, the base tuple must be read resulting in an additional number of disk accesses (often one per tuple, because the tuples are stored in insertion order, not in DID order). We call this disk access *materialization* of the result tuples. When using a primary index on the fact table on DID, a sort merge join speeds up query execution. No materialization is necessary in contrast to secondary indexes. However, a large number of pages may be accessed that contain a small number of tuples belonging to the join condition. A primary index clusters the fact table w.r. to a fixed ordered number of index attributes (*composite* or *compound index*). Thus, a restriction on the most significant attribute is important for efficient query processing. Typical DW queries, however, restrict any combination of dimensions, i.e., index attributes.

4.1 Sample Schema

In the following we often refer to the sample schema, in order to illustrate basic concepts and design alternatives. The sample schema is a DW schema with one fact table *fact* and the three dimension tables *dim_segment*, *dim_product* and *dim_time* with the following hierarchies:

- dim_segment: Country – Region – Micro Market – Outlet
- dim_product: Sector - Category – Product Group – Item
- dim_time: Year – Month - Day

The conceptual schema is shown in Figure 4-2.

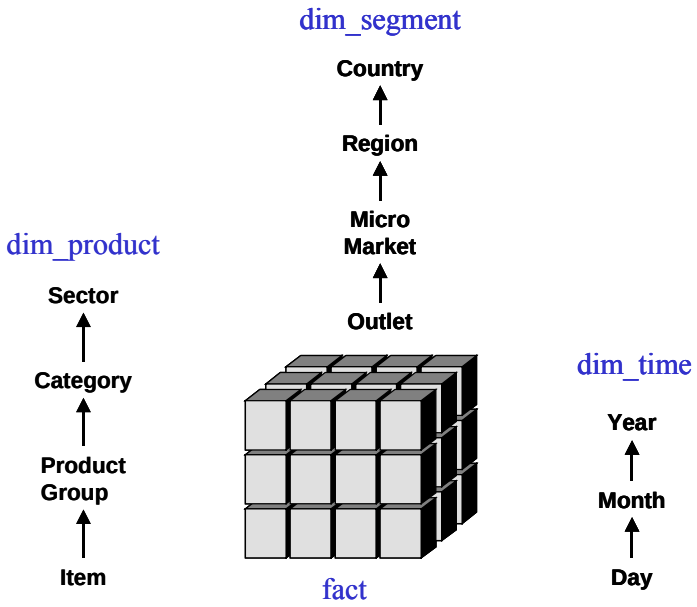


Figure 4-2: Conceptual Sample Schema

4.2 EHC – Encoding for Hierarchical Clustering

We transform hierarchical restrictions on the dimension table into range restrictions on the fact table. This method replaces an equi-join with the fact table via a set of dimID's by a semi-join (range restriction as "local" restriction on the fact table w. r. to the physical organization).

For each hierarchy level we assign surrogates to the children in the hierarchy, where the leftmost child starts with 0, the next child is assigned 1 etc. The concatenation of all surrogates (from top to leaves) is called *compound surrogate* and represents the whole hierarchy path (see Figure 4-3). The components of the compound surrogates, i.e., the surrogates of the corresponding hierarchy levels, are delimited by dots.

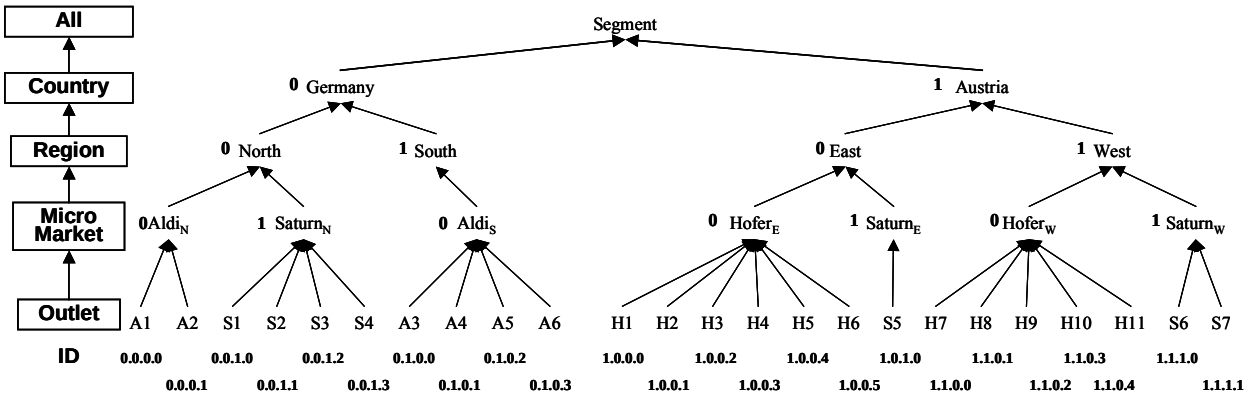


Figure 4-3: Hierarchy Encoding with EHC

With this encoding ([ZSL98], [MRB99]), point sets of subtrees of the hierarchy can be replaced by intervals. The predicate "Germany", is mapped to the interval [0.0.0.0 : 0.1.0.3]. This interval is used as predicate on the fact table. With the use of a clustering index, the tuples are read clustered and require a relatively small number of disk accesses. Section 6 describes EHC in more detail.

In order to use intervals instead of point sets, we transform the equi-join query into a semi-join:

```
SELECT SUM(F.turnover) FROM fact F, dim_segment D WHERE F.d_segment =
D.outletID AND D.Region = 'North'

SELECT SUM(F.turnover) FROM fact F WHERE F.cs_segment BETWEEN
MIN (SELECT ID FROM dim_segment WHERE D.Region = 'North') AND
MAX(SELECT ID FROM dim_segment WHERE D.Region = 'North')
```

The operator *MIN* (*MAX*) in the statement above returns the smallest (highest) number of the hierarchy for the corresponding predicate. For Figure 4-3, *MIN* returns 0.0.0.0 and *MAX* returns 0.0.1.3. A clustering index on *F.cs_segment* efficiently processes the range query. Otherwise a nested loop join (or sort merge join, if the fact table is sorted w.r.t. $d_{segment}$)[5] is used to process the query. A nested loop join performs a direct index access for each result of the dimension table predicate. The semi-join optimization is implemented into the optimizer of the DBMS and thus is transparent to the user (see Sections 7 and 10).

4.3 MHC – Multidimensional Hierarchical Clustering

The multidimensional nature of typical DW queries makes the use of *multidimensional* indexes attractive. With a *multidimensional clustering* index, query processing can be sped up significantly, if

[5] Also other join methods might be useful, e.g., hash joins, or the use of hash indexes.

data is multidimensionally clustered with respect to the queries. Because queries in DW contain hierarchical restrictions in multiple dimensions, we discuss a method to cluster data hierarchically *and* multidimensionally. If every dimension is encoded by a compound surrogate for its hierarchy, the records in the fact table are clustered with respect to these hierarchies. We call this the *Multidimensional Hierarchical Clustering, MHC*. The performance of query execution benefits from hierarchical predicates on several dimensions. The hierarchical predicates on the dimension tables are transformed to one local predicate (multidimensional range query) on the fact table (see Section 9).

4.4 Basic Design Decisions for the Implementation of EHC and MHC

EHC and the necessary surrogates are supposed to be transparent to the user. Instead of implementing high level language constructs for the definition of hierarchies and hierarchical dependencies ([Ora01]), we extend the `CREATE TABLE` statement by hierarchy specifications. We require, that all levels of the hierarchies of one dimension are stored within one basic dimension table (*leaf dimension table*) to efficiently compute the hierarchy encoding. This approach similar to a star schema may be extended by normalization to a snowflake schema (see Section 4.4.3).

4.4.1 Physical Design

The compound surrogates computed w.r.t. the hierarchy are stored as additional attributes in the leaf dimension tables. Indexes with hierarchical attributes to efficiently compute the encoding are necessary. These physical properties require that the user has to specify the EHC construct when he creates the physical data model, i.e., the schema in the DBMS.

The compound surrogate is a physical construct in the dimension table, i.e., the properties of the hierarchy such as the fanout (see Section 5.3) and an identifier to reference the compound surrogate are specified (see Section 7.6). As alternative, a "high level" statement would be possible that specifies the hierarchy with respect to one (or more) existing tables. In this case the existing dimension table is extended by physical constructs, or a new relation replaces the dimension table. In this thesis, we will not enlarge on this. Other DBMS like Oracle have such high level constructs (see [Ora01] and Section 7.6).

The advantage of the physical approach is, that high level constructs can be introduced that generate DDL statements with the corresponding EHC extensions. Thus, at a later step of the implementation, a create *hierarchy statement* may define compound surrogates transparently to the user. Another approach is that a design tool specifies the physical EHC constructs transparently to the user.

Two kinds of surrogates are necessary for EHC. The *compound surrogates* specify the hierarchy and are stored in the dimension table. Since the compound surrogates are used for hierarchical clustering in the fact table, they are also stored in the fact table. We call them *reference surrogates*. The reference surrogates in the fact table reference the corresponding compound surrogates in the dimension tables (foreign key relationship). They are often used as index key attributes, in order to cluster the tuples in the fact table w.r.t. the hierarchies.

The fact table contains the dimension keys and the reference surrogates. As alternative – especially to save space – the dimension keys could be omitted and replaced by the reference surrogates, because the reference surrogates are a bijective mapping of the dimension keys. The dimension keys themselves are stored in the dimension table. For most queries (star queries) the performance will not suffer from this optimization, but some basic queries will require additional joins to the dimension tables instead of being executed locally on the fact table, e.g., `select * from fact` or a dump of the fact table to a file requires additional lookups in all dimensions for all tuples.

An example for a fact table with dimension keys is shown in SQL Statement 1. This fact table contains the dimension key attributes *dseg*, *dprod,* and *dtime* that reference the keys of the corresponding dimension tables. Each dimension key is also represented by a reference surrogate. In SQL Statement

2, the dimension keys are removed. The reference surrogates have a reference to the keys of the dimension tables. A detailed description about the extended DDL syntax is given in Section 7.6.

```
CREATE TABLE fact (
      dseg  INTEGER REFERENCES dim_segment(outlet_id),
      dprod INTEGER REFERENCES dim_product(item_id),
      dtime INTEGER REFERENCES dim_time(day)
      turnover NUMERIC(10,2),
      SURROGATE cs_seg FOR dseg
      SURROGATE cs_prod FOR dprod,
      SURROGATE cs_time FOR dtime
) HCKEY is cs_seg, cs_prod, cs_time;
```

SQL Statement 1: DDL Fact Table with Dimension Key Attributes

```
CREATE TABLE fact (
      turnover NUMERIC(10,2),
      SURROGATE cs_seg REFERENCES dim_segment(cs)
      SURROGATE cs_prod REFERENCES dim_product(cs)
      SURROGATE cs_time REFERENCES dim_time(cs)
) HCKEY is cs_seg, cs_prod, cs_time;
```

SQL Statement 2: DDL Fact Table without Dimension Key Attributes

The schema design affects the join formulation of SQL statements, especially star queries. For a fact table with dimension keys, the star join is

```
SELECT sum(turnover)
FROM fact f, dim_segment s, dim_product p, dim_time t
WHERE f.dseg = s.outlet_id AND f.dprod = p.item_id AND
      f.dtime = t.day AND …
```

SQL Statement 3:Star Join on Fact Table with Dimension Key Attributes

For a fact table without dimension keys, the join condition is based on the reference surrogates:

```
SELECT sum(turnover)
FROM fact f, dim_segment s, dim_product p, dim_time t
WHERE f.cs_seg = s.cs AND f.cs_prod = p.cs AND f.cs_time = t.cs AND…
```

SQL Statement 4: Star Join on Fact Table without Dimension Key Attributes

As mentioned before, the surrogates should be considered as physical property of the fact table and dimension tables. Thus, the user should not see the surrogates in the tables at all and should not specify join conditions based on surrogates. We can define a view *user_fact* on the actual fact table that hides the reference surrogates and projects the dimension keys into the fact table:

```
CREATE VIEW user_fact (dseg, dprod, dtime, turnover) AS
SELECT s.outlet_id, p.item_id, t.day, f.turnover
FROM fact f, dim_segment s, dim_product p, dim_time t
WHERE f.cs_seg = s.cs AND f.cs_prod = p.cs AND f.cs_time = t.cs
```

SQL Statement 5: DDL View Definition to project Dimension Key Attributes into the Fact Table

The user now can formulate star joins in the following way:

```
SELECT sum(turnover)
FROM fact_user f, dim_segment s, dim_product p, dim_time t
WHERE f.dseg = s.outlet_id AND f.dprod = p.item_id AND
      f.dtime = t.day AND …
```

SQL Statement 6: Star Join with View

The use of the view fact_user instead of the actual fact table leads to query rewriting, because a view usually is rewritten very simply by replacing the view name by the view definition, e.g., for SQL Statement 6:

```
SELECT sum(turnover) FROM
  (SELECT s.outlet_id, p.item_id, t.day, f.turnover
     FROM fact f, dim_segment s, dim_product p, dim_time t
     WHERE f.cs_seg = s.cs AND f.cs_prod = p.cs AND
           f.cs_time = t.cs
  ), fact_user f, dim_segment s, dim_product p, dim_time t
  WHERE
     f.dseg = s.outlet_id AND f.dprod = p.item_id AND
     f.dtime = t.day AND …
```

Enhanced re-writing methods are necessary to reduce this complicated query – subquery combination to the same query as in SQL Statement 4. We will not enlarge on this in the thesis, because in our schema, we use the fact table with dimension key and surrogate attributes.

4.4.2 Denormalized Leaf Dimension Table

For the computation of EHC, all levels need to be stored within one relation. An index representing the hierarchical structure enables efficient computation of the compound surrogates. The user has to create a table that contains all levels of the hierarchy of the dimension. We call this table *leaf dimension table (LDT)*. This star schema approach can be extended and optimized by normalization (see Section 4.4.3).

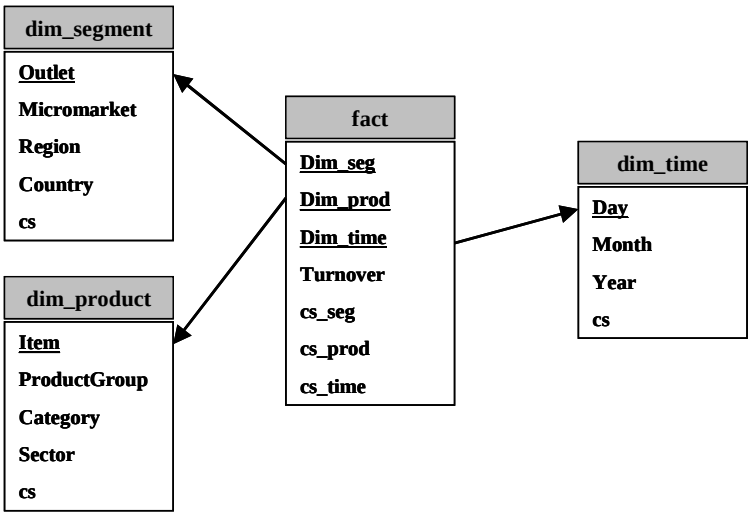


Figure 4-4: De-normalized Schema for Sample schema

For example, the segment dimension of the sample schema contains the hierarchy levels *country – region – micromarket – outlet*. In the leaf dimension table, we store all hierarchy levels and the compound surrogate *cs*: `dim_segment(outlet, micromarket, region, country,`

cs). Note that *outlet* is primary key of the segment dimension, i.e., *outlet* is a unique identifier for the hierarchy paths. The schema is shown in Figure 4-4.

Alternatively, the levels can be referenced from foreign tables and stored implicitly and invisible in the LDT. As consequence, the attributes of the hierarchy levels that are not located in LDT have to be added to LDT and maintained accordingly. The hierarchy index can be created on the hierarchy attributes and used for efficient computation of compound surrogates. The implementation would provide a view on such a dimension table without the additional attributes.

For example, the segment dimension could be modeled in a snowflake approach (see Figure 4-5):

```
segment(outlet, micromarket, …), seg_micromarket(micromarket,
region, …), seg_region(region, country, …), segm_country(country, …)
```

where each table represents one level and contains the hierarchical relationship via foreign key references. The dots "…" represent further feature attributes of the hierarchy levels. The handling of the compound surrrogate cs has been explained in Section 4.4.1. The physical schema where the leaf dimension table *segment_leaf* replaces the *dim_segment* table looks like the following:

```
segment_leaf(outlet, micromarket, region, country, cs, …).
```

The user has the view on the dimension:

```
CREATE VIEW dim_segment (outlet, micromarket, …) AS SELECT outlet,
micromarket, … FROM segment_leaf.
```

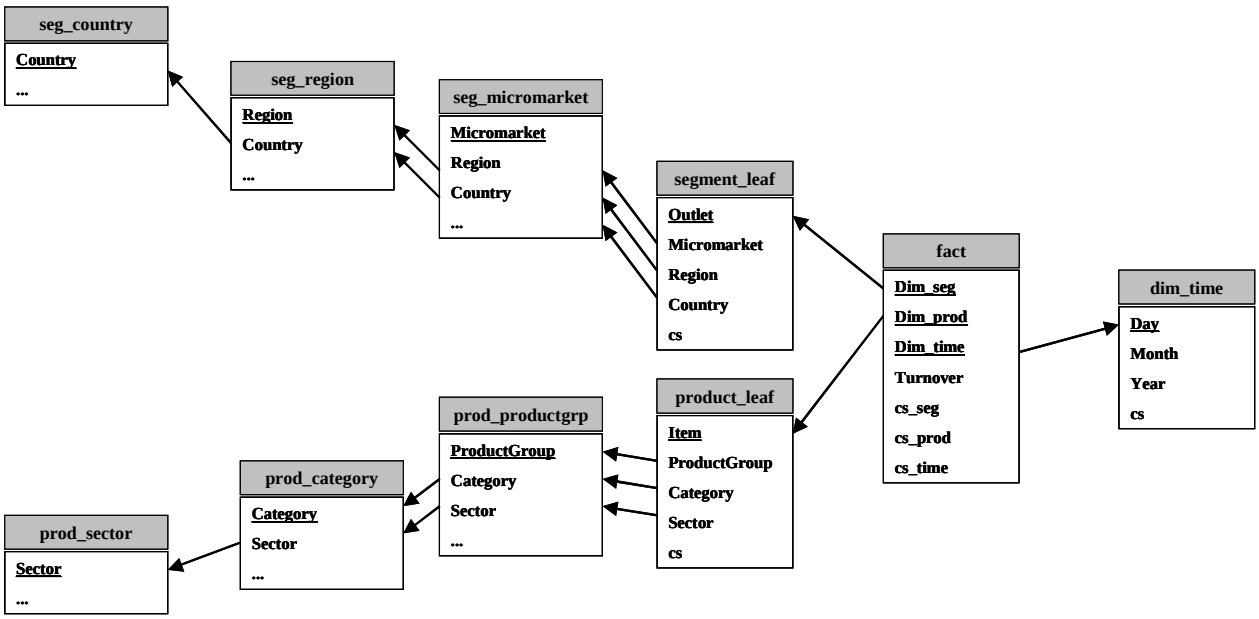


Figure 4-5: Path normalized Snowflake Schema for the Sample Schema

A second alternative is not to store the levels of foreign tables within the LDT but to use an index containing the hierarchy attributes of different tables (join index). Join indexes are not implemented in Transbase®, so we use the first case with the LDT forcing the user to explicitly store all levels of the hierarchy in the LDT (see Section 7.6).

4.4.3 Normalizing Hierarchies (Snowflake Schema)

The dimensions of DW applications may be very complex (complex structure of hierarchies, feature attributes etc.). For maintenance reasons, a normalization of the dimension tables may be useful. For

optimizing reasons, a normalization can make hierarchical queries recognizable for the optimizer and result in more efficient query processing (see Section 9). We require that the leaf dimension table again contains all hierarchy levels. Further normalization, however, is not restricted by the optimizer and is supported by the algorithms to process star queries (see Sections 2.4 and 10.1.5.1).

4.4.4 Indexes

As mentioned earlier, some indexes for the LDT are necessary for a fast computation of compound surrogates and efficient query processing. Basically, the index key of the physical clustering primary index of the LDT is the leaf level, i.e., the *dimension key*. This dimension key is the primary key of the dimension table.

The *hierarchy index*, called *DXh*, is a secondary index on LDT with the hierarchical attributes, feature attributes and a compound surrogate: $LDT(h_1, h_2, \ldots, h_t, f_1, \ldots, f_k, cs)$. *DXh* contains the hierarchy levels and the compound surrogate *cs*: $DXh = (h^t, h^{t-1}, \ldots, h^1, cs)$, where h^t is the top level and h^1 the leaf level and dimension key. *DXh* is useful to compute a new surrogate. A sophisticated lookup method is implemented that efficiently gets the next free surrogate (see Section 8.2). *DXh* is used to compute the intervals in the query processing phase for the semi-join (see Section 10.1.8).

We further need the secondary index *DXcs* that contains the compound surrogate: *DXcs = (cs)*. It is used to maintain compound surrogates and for query processing.

In the example for the product dimension, the following two indexes are created: $DXh_{product}$ = *(category, productgroup, item, cs)* and $DXcs_{product}$ = *(cs)*.

These indexes are “system indexes”, i.e., they are created automatically triggered by the surrogate clause in the create table statement. They are invisible to the user and cannot be dropped to ensure availability of the indexes.

Note that a LDT that contains *n* hierarchies has *n* DXh indexes and *n* DXcs indexes (one for each hierarchy).

5 Surrogates

It is a common method to encode information by numbers and use these so called *surrogates* as representatives. This indirection is robust w. r. to changes, because the surrogates can be used also for a changed string (e.g., if 'Jan.' is changed to 'January', we do not have to change the surrogate and the tables where these surrogates are used).

5.1 Concept of Surrogates

Surrogates are useful for many applications. Basically, a surrogate provides a mapping of an element with an arbitrary data type to a number. Such a mapping is necessary for methods that only support numeric values (e.g. queries on UB-Trees), for space saving reasons, anonymizing data, artificial orders etc.

Definition 5-1 (Surrogate):

A *surrogate* function $s: V^N \rightarrow N_0$ returns a number n, depending on the input parameter v, where v is a set $v = \{ v_i \}$: $s(v) = n$. The function is injective, the invers function $s^{-1}: N_o \rightarrow V^N$ returns the original value v: $s^{-1}(n) = v$. We call $s(v)$ the surrogate of v. Thus a surrogate replaces an element or a set of elements (e.g., a sequence of hierarchy levels) with arbitrary data type by a number. This number can either include a semantics or not. □

Surrogates are useful due to several reasons:

- The replacement can save space (e.g., long strings that are stored repeatedly are replaced by short numbers).
- For DBMS query optimizing, numbers with semantic can achieve advantages (e.g., properties of physical structures such as generating ranges to make use of UB-Trees in data warehouses).

We distinguish two different types of surrogates:

- *Non-Semantic* Surrogates
- *Semantic* Surrogates

Definition 5-2 (Non-Semantic Surrogates):

A *non-semantic surrogate* function $s_{ns}: V \rightarrow N_o$ is a surrogate function that returns a number n without any semantic information: $s_{ns}(v) = n$, e.g., there is no semantical ordering on the generated numbers. □

Example 5-1 (Non-Semantic Surrogate):

An example for a non-semantic surrogate is the *enumeration surrogate* that maps long character strings to numbers using a mapping table. Usually this mapping is an explicit table, e.g., $enum_{sns}$ containing the mapping. The result of the function s_{ns} depends on insertion order of the strings into the mapping table (i.e. no lexicographic order). The operations to insert and delete strings are straight forward:

- `insert(sns, v)` will insert a new tuple $(v, max(n)+1)$ in the mapping table $enum_{sns}$.
- `delete(sns, v)` will remove the tuple (v, n_v) from the mapping table $enum_{sns}$.

s_{ns} is the mapping function and $enum_{sns}(\underline{v}, n)$ is the enumeration table with the attributes v for the strings to map and n for the mapping value.

A DBMS can evaluate enumeration surrogates efficiently with indexes, e.g., the secondary index $enum_{sns}\_sec(\underline{n}, v)$. Enumeration surrogates can be used to replace long character strings.

Such a mapping method is useful for low cardinality domains (compared to the number of tuples in a table). □

Definition 5-3 (Semantic Surrogates):

A *semantic surrogate* function $s_{sem}: V \rightarrow N_0$ is a surrogate function $s_{sem}(v)=n$ that returns a number n including a semantic with respect to v. □

Example 5-2 (Semantic Surrogate):

A compound surrogate is an example for semantic surrogates. Compound surrogates are used for EHC to replace a hierarchy path by a number. Thus, a compound surrogate depends on the path of the tuple (see Section 5.3).

Another example are intensional surrogates that are also computed by a function and can be used for many applications (see Section 5.2). □

The following sections describe the computation of semantic surrogates for intensional surrogates and compound surrogates.

5.2 Intensional Surrogates

Intensional surrogates are semantic surrogates that are computed by the value of their arguments (i.e., no additional lookup table is used). In this Section, we discuss the *string surrogates* in more details.

5.2.1 Overview

An example for an intensional surrogate is the time surrogate. It can be useful for many applications. It implies a hierarchy depending on the specification of the ranges for the time attribute, but is encoded within one value without storing the hierarchy explicitly. For example, a time attribute spanning year – month – day is a hierarchy with the three levels year, month and day. The time values of the tuples are on day granularity. A predicate may restrict the year level (e.g., *time.year=2001*), which is transformed to an interval (e.g., *time between 366 and 731*, if time starts with the year 2000).

Strings can be stored in conventional B*-Trees, but are not integrated into the UB-Tree in Transbase® Hypercube[6], because the domain of strings is too large to be used in z-addresses. However, some applications require indexed attributes of string data type. The conventional way of encoding strings assigns each string a unique numeric id. This method is unfeasible for range queries, because the assignment of the id usually does not reflect the lexicographic order of strings if we assume random insert order. An order preserving mapping is necessary to support range queries for strings in UB-Trees. Thus, we introduce an order preserving mapping of strings to numbers: the *intensional string surrogates*.

5.2.2 Strings

It is inefficient to encode strings with the UB-Tree in the straight forward way, because a large number of bits is necessary to represent each string value by a bit string ([Mar99]). For example to encode a ten character string with the characters being coded with 8 bits (standard code page for the widely used LATIN character set), we need 80 bits. Usually, the domain is very sparse, because many characters will not occur in the strings. Depending on the data, the most frequently used characters are letters and digits. Often, strings do not have a fix length (e.g., *char(*)* as attribute type). Other encoding like UNICODE even require more bits for one character.

Applications, however, require to index strings multidimensionally in combination with other attributes (e.g., telephone books with last name, first name, post code, age etc.). The user does not want or even is not able to create order preserving surrogate numbers for the strings to store in the UB-Tree. Even, if he is, he has to formulate the queries according to this user-mapping.

[6] Only numeric data types are feasible, such as integer, numeric, bit strings etc.

For example, the strings of Table 5-1 are stored in a UB-Tree with the attribute *surr* as index key, the query `SELECT * FROM table WHERE name BETWEEN 'Bush' AND 'Clinton'` has to be rewritten into `SELECT * FROM table WHERE surr BETWEEN 201 and 327` to process it with the use of the UB-Tree. If the surrogates are not computable from the value of the string attribute, the rewriting is not possible without additional information or effort (e.g., looking for the bounds of the interval in a secondary index).

surr	Name
201	Bush
327	Clinton
438	Demirel
…	
1722	Stoiber

Table 5-1: Sample Mapping

Thus, we discuss a DBMS internal mapping between strings and bit strings that is efficient and not space wasting in order to not disturb the properties of UB-Trees. Assuming that most strings mainly contain letters, and digits, we encode a subset of all possible character values: digits, lower and upper letters and some special letters (e.g., German umlauts like 'ä', 'ö' etc.). The number of possible character values grows, if we support larger character sets like UNICODE.

We additionally limit the number of encoded characters (length of the encoded string) to a "reasonable" small number. For conventional strings, five or six characters should be enough, because most natural words (independent on the language) seldom have longer common prefixes. For special applications with composite words or equal prefixes, e.g., for SQL statements, this assumption is wrong. Also for generated strings, this assumption might be wrong leading to a small selectivity and therefore only a small reduction of the result set for multidimensional range queries. We will show some examples where this encoding method is not very useful. In such cases, more characters must be used for the encoding.

The limitation of the encoding to a prefix (such as the mapping of a number of seldom used characters to one single surrogate value) violates the injectivity property of the surrogate function. This function is not injective any more and the original argument, i.e., the string, must be stored separately in order to get the original string from a surrogate. The surrogate resulting from the string encoding is seen as a representative of the equivalence class of all strings with the same encoding.

The mapping can be applied to the SQL data types *char(n), char(*), varchar(n), binchar(n),* and *binchar(*)*. The mapping of the *binchar* data types differs from the mapping above, because in an attribute of type *binchar*, binary information is stored. We use an 8-bit coding, i.e., no equivalence classes are necessary for single characters, because every character of the *binchar* string has a unique surrogate (see Section 5.2.2.2).

Note that we store the original string attribute in the table and use the encoding as index attribute of the UB-Tree. A restriction via predicates such as `BETWEEN` or `LIKE` is mapped to a restriction on the corresponding surrogate index attribute. The encoding nature creates a bounding box that contains all strings with the same encoding, i.e., usually a superset of the tuples qualified by the predicate. Post-filtering is necessary to get the final result (see Section 5.2.2.3).

5.2.2.1 Encoding Characters

As mentioned above, strings often do not contain all possible characters of a specified character set (domain). Natural strings contain digits, lower and upper letters, and some special characters. For the

remaining characters we define equivalence classes, where a set of characters is represented by a single value. The mapping function therefore is not injective and a re-transformation is not possible. The goal is to use a minimum number of bits for the encoding.

One approach uses six bits per character, where 64 equivalence classes can be encoded: the digits (10), lower letters (26), upper letters (26), special characters (1) = 10 + 26 + 26 + 1 = 63, where the special characters are in one equivalence class (see Table 5-2). The equivalence classes of this mapping have the same order as the lexicographic order (apart from characters within one equivalence class). In the mapping tables, we denote a character with the LATIN code *1* in C syntax by *'\1'* etc., because the character cannot be represented graphically. For example, the equivalence class with code=0 contains the characters with LATIN code 0, 1, …, 48, i.e., the first 32 not printable characters and the printable characters from blank ' ' to '0'.

Character	Code	Character	Code	Character	Code	Character	Code
'\0' '\1' … '/' '0'	0	'G'	16	'W'	32	'm'	48
'1'	1	'H'	17	'X'	33	'n'	49
'2'	2	'I'	18	'Y'	34	'o'	50
'3'	3	'J'	19	'Z' '[' … '`'	35	'p'	51
'4'	4	'K'	20	'a'	36	'q'	52
'5'	5	'L'	21	'b'	37	'r'	53
'6'	6	'M'	22	'c'	38	's'	54
'7'	7	'N'	23	'd'	39	't'	55
'8'	8	'O'	24	'e'	40	'u'	56
'9' ':' … '@'	9	'P'	25	'f'	41	'v'	57
'A'	10	'Q'	26	'g'	42	'w'	58
'B'	11	'R'	27	'h'	43	'x'	59
'C'	12	'S'	28	'i'	44	'y'	60
'D'	13	'T'	29	'j'	45	'z'	61
'E'	14	'U'	30	'k'	46	'{' … '\255'	62
'F'	15	'V'	31	'l'	47		

Table 5-2: 6-Bit Encoding

Character	Code	Character	Code	Character	Code	Character	Code
'\0' '\1' … '/' '0'	0	'G' 'g'	8	'O' 'o'	16	'W' 'w'	24
'1' '2' '3' '4' '5' '6' '7' '8' '9' ':' … '@'	1	'H' 'h'	9	'P' 'p'	17	'X' 'x'	25
'A' 'a'	2	'I' 'i'	10	'Q' 'q'	18	'Y' 'y'	26
'B' 'b'	3	'J' ''j'	11	'R' 'r'	19	'Z' 'z', '[' … '`'	27
'C' 'c'	4	'K' 'k'	12	'S' 's'	20	'{' … '\255'	28
'D' 'd'	5	'L' 'l'	13	'T' 't'	21		
'E' 'e'	6	'M' 'm'	14	'U' 'u'	22		
'F' 'f'	7	'N' 'n'	15	'V' 'v'	23		

Table 5-3: 5-Bit Encoding (Case Insensitive Semantic)

If the number of bits should be reduced to five, we have to half the equivalence classes to at most 32. In this case, the lower and upper letters are in one equivalence class (e.g., ‘a’ and ‘A’) and all digits are in one equivalence class (see Table 5-3)**.** Another method is to store some “seldom” character values within one equivalence class and allow 10 equivalence classes for the digits (see Table 5-4).

Character	Code	Character	Code	Character	Code	Character	Code
‘\0’ ‘\1’ … ‘/’ ‘0’	0	‘8’	8	‘G’ ’g’	16	‘P’ ‘p’ ‘Q’ ‘q’	24
‘1’	1	‘9’ ‘:’ … ’@’	9	‘H’ ‘h’	17	‘R’ ‘r’	25
‘2’	2	‘A’ ‘a’	10	‘I’ ‘i’	18	‘S’ ‘s’	26
‘3’	3	‘B’ ‘b’	11	‘J’ ‘j’ ‘K’ ‘k’	19	‘T’ ‘t’	27
‘4’	4	‘C’ ‘c’	12	‘L’ ‘l’	20	‘U’ ‘u’	28
‘5’	5	‘D’ ‘d’	13	‘M’ ‘m’	21	‘V’ ‘v’ ‘W’ ‘w’	29
‘6’	6	‘E’ ‘e’	14	‘N’ ‘n’	22	‘X’ ‘x’ ‘Y’ ‘y’	30
‘7’	7	‘F’ ‘f’	15	‘O’ ‘o’	23	‘Z’ ‘z’ ‘[‘ … ‘’’ ‘{‘ … ‘\255’	31

Table 5-4: 5-Bit Encoding (seldom used Characters)

Other mappings are also possible. However, the mapping influences query processing, especially the `BETWEEN` predicates (see Section 5.2.2.3).

5.2.2.2 Computing Surrogates for Strings

The computation of the surrogates is necessary for two operations: inserting a tuple into the UB-Tree and computing the query predicates (especially the ranges for the multidimensional range query). In the following we assume, that the string attribute is one of the index attributes of a UB-Tree.

Inserting a tuple into the UB-Tree is a two step procedure. First, we compute the z-value of the tuple to insert. Second, the insertion into the B-Tree requires the computation of a number of z-values of the tuples already stored on the leaf page, because the z-values of the tuples on the leaf pages are not stored. In the Transbase® implementation of the UB-Tree the tuples on the leaf pages are stored in z-value order. Therefore, for every UB-Tree operation, a transformation of the UB-Tree index attributes to one z-value is necessary, especially for the insert on the leaf page (binary search). However, the number of transformations is low. A binary search with 128 tuples on the leaf page requires at most seven transformations to find the correct position to insert the tuple.

The algorithm for the computation of the surrogate and transfer into the z-value bit string is trivial (see Algorithm 5-1). In this algorithm, the constant *NR_CHAR_ENCODE* contains the number of characters that are used for the encoding, *BITS_PER_CHARACTER* is the number of bits used to encode one character (e.g., 6 for standard strings, 8 for binary strings). The string to encode is stored in *sourceStr*, the length of this string is *strlen*. The lookup table containing the encoding for each character is stored in *LOOKUP[0..255]*. *setBit(b, pos)* sets the bit on position *pos* of bit string *b* to 1. *surrvalue* is a bit string and contains the computed surrogate for the string. It is initialized with 00..00. *surrvalue* is used later for the computation of the z-value of the UB-Tree.

Algorithm 5-1 (Computation of String Surrogates):

```
if (strlen > NR_CHAR_ENCODE) NrChar = NR_CHAR_ENCODE
     else NrChar = strlen      // for shorter strings than length 6
for i = 0 to NrChar-1
     strsurr = LOOKUP(sourceStr[i])
     for j = 0 to BITS_PER_CHARACTER-1
          if strsurr MOD 2 == 1
```

```
                setBit(surrvalue, pos)
            strsurr = strsurr DIV 2
```

In the algorithm we compute the surrogate *strsurr* for the character by a lookup in the encoding table (e.g., Programming Example 1). The bits of *strsurr* are transferred to the with 0-bits initialized z-value *zvalue* by shifting the bits to right and setting only the 1-bits in *zvalue*.

The lookup table for the 6-Bit encoding (Table 5-2) looks like the following (in C syntax):

```
static unsigned char ub_code_table[256] = {
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  1, /* .............. 0 1 */
 2,  3,  4,  5,  6,  7,  8,  9,  9,  9, /* 2 3 4 5 6 7 8 9 ... */
 9,  9,  9,  9,  9, 10, 11, 12, 13, 14, /* ..........A B C D E */
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, /* F G H I J K L M N O */
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, /* P Q R S T U V W X Y */
35, 35, 35, 35, 35, 35, 35, 36, 37, 38, /* ..............a b c */
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, /* d e f g h i j k l m */
49, 50, 51, 52, 53, 54, 55, 56, 57, 58, /* n o p q r s t u v w */
59, 60, 61, 62, 62, 62, 62, 62, 62, 62, /* x y z ............. */
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
62, 62, 62, 62, 62, 62
};
```

Programming Example 1: Lookup Table for 6-Bit Encoding

Example 5-3 (String Encoding):

The encoding for the string ‘Roland’ with 6 bit encoding is done in the following way:

```
strlen(‘Roland’) = 6
NR_CHAR_ENCODE = 5
```

We compute the bits for each character of the prefix string ‘Rolan’ (the first 5 characters). The lookup in the 6 bit encoding LOOKUP table returns the following encoding values:

```
‘R’ » 27 = 011011
‘o’ » 50 = 110010
‘l’ » 47 = 101111
‘a’ » 36 = 100100
‘n’ » 49 = 110001
```

The resulting bit string is `surrvalue = 01101111001010111110010011000 1`. Note that for all strings with the prefix ‘Rolan’ the surrogate is the same.

5.2.2.3 Query Processing

For strings in UB-Trees we need the mapping to transform string predicates to index key predicates (surrogates). Especially, “range restrictions” on strings are interesting. Range restrictions are the predicates `BETWEEN` and `LIKE`, where `BETWEEN` specifies an exact range: `attr BETWEEN val1 AND val2` is the same as `attr >= val1 AND attr <= val2`. The `LIKE` predicate in combination with wildcards is an implicit range: `attr LIKE ‘pat%’` is true for strings with the prefix *‘pat’* and any following suffix. This predicate is equivalent to `attr > 'pas\255' AND attr < ‘pau’`.

Excursion: Trailing Blank Semantic in SQL-92

> It is not obvious, why the predicate `attr LIKE ‘pat%’` is mapped to `attr > 'pas\255' AND attr < ‘pau’`. Especially, the lower bound intuitively could be `attr>=‘pat’` instead of `attr>'pas\255'`. According to the SQL-92 standard ([DD93]), a string ‘pat’ is equal to a string with the prefix ‘pat’ and an arbitrary number of following blanks, e.g., ‘pat^^^^‘, where ‘^’ represents the blank character. However, the ASCII code for blanks (usually, 32) is higher than the ASCII code for other characters, like ‘\t’ (<tabulator>: eight). The lexicographic order w.r.t. ASCII of string ‘pat\t’ however is smaller than the string ‘pat^^’, where ‘^’ here represents a blank. Because the B-Tree uses ASCII order and does not interpret the arguments of range queries, the lower bound of such queries must be changed to the next smaller character combination. In the case of the LIKE predicate, this combination is `'pas\255'`, where 255 is the highest possible code, and *‘s’* is the previous character to *‘t’* in ASCII order.

A point restriction, such as `attr = val`, is equivalent to a range restriction with the same lower and upper bound: `attr BETWEEN val AND val`. In both cases, we use the mapping method to compute the multidimensional interval, i.e., the z-value for the multidimensional range query on the UB-Tree. In the operator tree, post-filtering with the original restriction checks all tuples returned by the MD range query to get the correct tuples. Some tuples of the equivalence classes defined by the mapping might not correspond with the range predicate, or additional restrictions can be specified in the query.

Post-filtering is used for all queries on UB-Trees in the Transbase® implementation, because the tuples on the B-Tree leaf pages do not contain the z-values. Instead of transforming the UB-Tree index attributes into a z-value, the predicate tree of the query generated by the optimizer is used to post-filter all tuples on the leaf page. Thus, there is no additional effort for the string mapping implementation in the query processing phase.

If a UB-Tree index attribute of character data type is not restricted, the minimum and maximum bit string values are used for the range query: all bits are set to 0 or 1 accordingly.

Note that for the 5-bit encoding we have a special case for range predicates. The mapping function must distinguish between a range on upper letters, lower letters or mixed (from a upper letter to a lower letter), because lexicographically, ‘b’ has a higher order than ‘Z’ (‘b’ > ‘Z’). Therefore a restriction `attr BETWEEN ‘C’ AND ‘k’` contains the letters ‘C’, ‘D’, ‘E’, …, ‘Z’, …, ‘a’, ‘b’, …, ‘k’. With the 5-bit encoding, however, the letters ‘C’, ‘c’ are in the same equivalence class such as ‘K’, ‘k’. A simple transformation to a range `f(attr) BETWEEN f(‘C’) AND f(‘k’)`, where *f* is the mapping function, qualifies a too small number of characters (only the characters ‘C’, ‘D’, … ‘K’, ‘c’, ‘d’, …, ‘k’).

Therefore we modify the transformation of the query to `f(attr) BETWEEN f('A') AND f('Z')`, i.e., the smallest and largest letter. Post-filtering with the original restriction will ensure that the correct tuples are returned. Note that the characters in ASCII between 'Z' and 'a' are in the same equivalence class as 'Z' and 'z'. Thus they are "covered" by the transformation (see Table 5-3).

An optimization is possible, if a range like `attr BETWEEN 'K' AND 'b'` is specified (i.e., the upper bound of the interval is alphabetically smaller than the lower bound). In this case, not the transformation to the complete range `f(attr) BETWEEN f('A') AND f('Z')` is necessary, because the equivalence classes of {'C', 'c'}, {'D, 'd'}, …, {'J', 'j'} are not covered by the restriction. In this case we can transform the restriction to two intervals `f(attr) BETWEEN f('K') AND f('Z') OR f(attr) BETWEEN f('a') AND f('b')`.

An alternative is to use a different 5 bit mapping:

`{ A, B } → 1, { C, D } → 2, …, { Y, Z } → 13,`

`{ a, b } → 14, { c, d } → 15, …, { y, z } → 26.`

With this mapping, the transformed ranges are more exact.

The LIKE predicate does not cause problems with the 5-bit encoding, because the optimizer transformation of the restriction `attr LIKE 'B%'` is transformed to `attr BETWEEN 'A\255' and 'C'` that is transformed to `f(attr) BETWEEN f('A\255') and f('C')` and qualifies the corresponding range for the encoding. Post-filtering removes the (usually small number of) tuples that do not fulfil the restriction.

5.2.2.4 Defining a Character Set

In some cases, the number of distinct characters used for the strings is limited to a couple of characters. Therefore, many bits are "wasted". The encoding could use much less bits and thus improve the multidimensional clustering in the UB-Tree. For example, if only post codes of Austria and Germany are stored, it is enough to use the characters 'A', 'D', '0', '1', …, '9' for strings like 'D80686' or 'A5122'. In this case, we need 12 distinct encodings, i.e., four bits per character are sufficient.

The domain of a string attribute in the CREATE TABLE statement can be a CHARSET[7]. The validation of the strings is done by the DBMS refusing all strings that contain other characters. For this purpose, we extend the CREATE TABLE statement.

```
CREATE TABLE tab (
     attr char(*) CHARSET ('A', 'D', '0', '1', '2',
     '3', '4', '5', '6', '7', '8', '9'),
…)
```

The valid characters are listed. When inserting a string, the interpreter checks that only characters of this character set occur.

The order of the characters is the lexicographic order (depending on the overall character set such as LATIN). The lookup table is stored persistently in an additional system catalog table. The number *b* of bits necessary for the mapping of a character depends on the cardinality *C* of the character set: $b = \lceil \log_2 C \rceil$. The number of characters used for the mapping is $\lfloor MAXBITS / b \rfloor$, where *MAXBITS* is the maximum length of the bit string for the character attribute.

[7] There is no SQL construct to define a character set on attribute level (see [DD93]). The CREATE CHARACTER SET statement is on the top of the SQL interface and is valid for a complete session. Such a character set must be a superset of all characters of SQL and the characters stored in the DBMS.

5.2.2.5 Problems

The mapping of string prefixes to surrogate bit strings is not bijective. Thus, different strings can have the same surrogate value. This surrogate value, however, is used to calculate the UB-Tree z-value and therefore the key value of the underlying B-Tree. It is not possible to use a unique z-value for the UB-Tree. Therefore, an additional unique index must be created, in order to define the key of the table.

Non-unique B-Trees cause some performance overhead compared to unique B-Trees (see Section 10.5).

5.3 Compound Surrogates

Compound surrogates are an example for extensional surrogates. They map hierarchy paths to numbers that encode the path. Adjacent hierarchy members (i.e., members of a subtree in the hierarchy) are mapped to adjacent numbers (i.e., an interval). We call this mapping a *hierarchical clustering*. A hierarchy subtree, i.e., all children (and their successors), can be specified by an interval instead of specifying all paths separately.

The following sections give an overview of the basic concepts of compound surrogates, discuss the mapping schema and present an efficient computation for the compound surrogates.

5.3.1 Basic Concept

DW queries often have hierarchical predicates with point restrictions on a hierarchy level. Intervals on a hierarchy level are seldom, because in most cases, no order is defined on hierarchy levels[8]. In this thesis, we describe a mapping that clusters hierarchical data according to the partial order that is defined by hierarchical relationships in the hierarchy.

A hierarchy provides a disjoint partitioning of the leaf level elements depending on the members of the hierarchy levels. A partitioning with respect to a higher level (i.e., all leaf levels that belong to a higher level) will reduce the number of partitions and increase the number of elements per partition, whereas a partition according to a lower hierarchy level will produce a high number of small partitions. The finest partitioning is on leaf level, where each partition contains one element and the number of partitions is equal to the number of leaf elements.

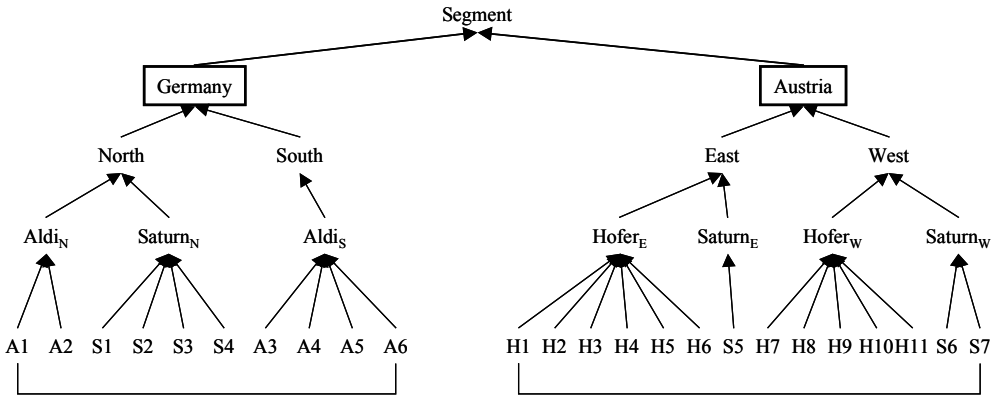


Figure 5-1: Partitioning of leaf levels according to a higher level

[8] Hierarchies with an order on levels are the time dimension or spatial dimensions. The time dimension has an order on years, months, days etc. A spatial dimension uses coordinates to establish an order.

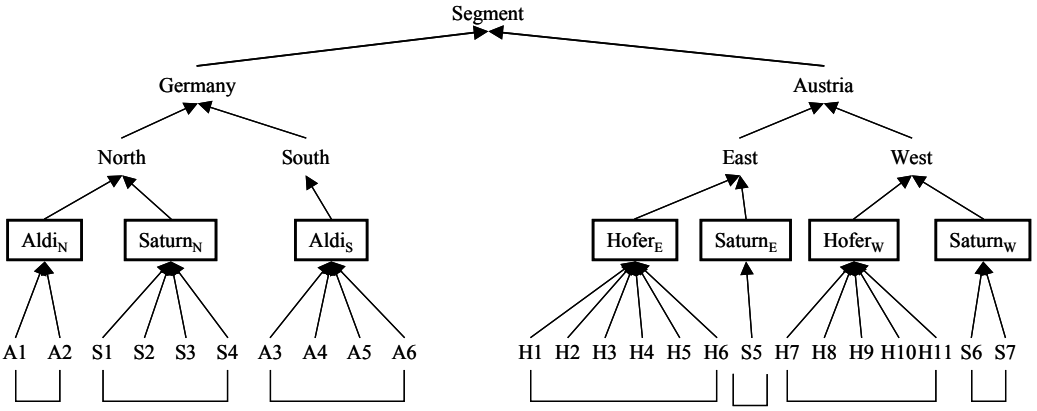


Figure 5-2: Partitioning of leaf levels according to a lower level

In Figure 5-1, a partitioning according to a high level is illustrated (two partitions), whereas the leaf members of Figure 5-2 are partitioned according to a lower level (seven partitions).

In the following, we use the notation m^i_j for hierarchy members to assign the members to a specific hierarchy level. The member m^i_j is the j^{th} member of level h^i of the hierarchy. The member of the top level (hierarchy level h^t) is denoted by m^t_1 (always one single member), the members of the leaf level (leaf members) are denoted by m^1_k.

Surrogates require simple hierarchies. We usually specify the path from a hierarchy member m^i_j to the top member in the order $(m^t_1, m^{t-1}_k, \ldots, m^i_j)$, i.e., the path from the root to the corresponding member. Note that there is only one unique path from m^t_1 to m^i_j due to the simple hierarchy properties.

Definition 5-4 (Compound Surrogate, cs):

A *compound surrogate cs* of a member m^i_j is the concatenation of surrogates of the members of the hierarchy path from the top member m^t_1 to member m^i_j. $cs(m^i_j) = cs(\Phi)$ denotes the compound surrogate of member m^i_j, where Φ is the path from m^t_1 to m^i_j. □

Compound surrogates provide hierarchical clustering, because the most significant cs component (surrogate) corresponds to the highest hierarchy levels, and the least significant cs component is the surrogate of the leaf hierarchy member. Members with the same father member have the same cs prefix (i.e., they lie within an interval). Usually, compound surrogates represent complete paths from top to leaf.

Definition 5-5 (Compound Surrogate Component, cs Component):

A *compound surrogate component*, denoted by cs^i, is the part of the compound surrogate that corresponds to the surrogate of the member m^i_j of level h^i. □

Definition 5-6 (Fanout):

The *fanout:* $h_i \rightarrow N_0$ of a hierarchy level h_i is the maximum number of children at level h_{i+1}: $fanout(h_i) = max\ (|\ children\ (m^{i+1}_k)\ |)$, with $1 \le k \le$ number of $|\ h_{i+1}\ |$. □

The function *children:* $h_i \rightarrow 2^{h_{i-1}}$ returns the child members of member m^i_j, $|\ children\ (m^{i+1}_k)\ |$ denotes the cardinality of the children, i.e., the number of children of m^{i+1}_k. The fanout function is necessary to get a maximum cardinality for every cs component.

5.3.2 Establishing an Enumeration Schema

A hierarchy level h^i consists of a number of members m^i_j: $h^i = \{ m^i_j \}$. Every member of level h^t (top level) to level h^2, where t is the depth of the hierarchy, has a set of children members m^{i-1}_k. We define a bijective function *ord* to enumerate the members of the hierarchy with respect to hierarchical clustering.

Definition 5-7 (ord):

The surrogate function *ord:* $h^i \rightarrow N_0$, $ord(m^i_j) = n$, returns a number n with respect to the number of the member in the child list of the father member, where $0 \leq n \leq | children(m^{i+1}_k)\text{-}1 | \}$ and m^i_j is a child of m^{i+1}_k. □

Each child is assigned a surrogate number between 0 and the maximum number of siblings ([Mar99]). Figure 5-3 illustrates this enumeration schema for the example hierarchy. The numbers in front of the hierarchy members contribute to the numbers returned by the *ord* function. The top member (m^t_1) usually is not enumerated, because it has only one member m^t_1: $ord(m^t_1) = 0$. The leaf members are assigned compound surrogates (as concatenation of the surrogates of the upper levels and the surrogates of the leaf level).

Example 5-4 (ord):

For the example hierarchy, $ord(\text{“}Saturn_E\text{”}) = 1$, because *“$Saturn_E$”* is the second child of *“East”*. Usually the ord function returns a non semantic surrogate, because no order will be established on the children of a member. □

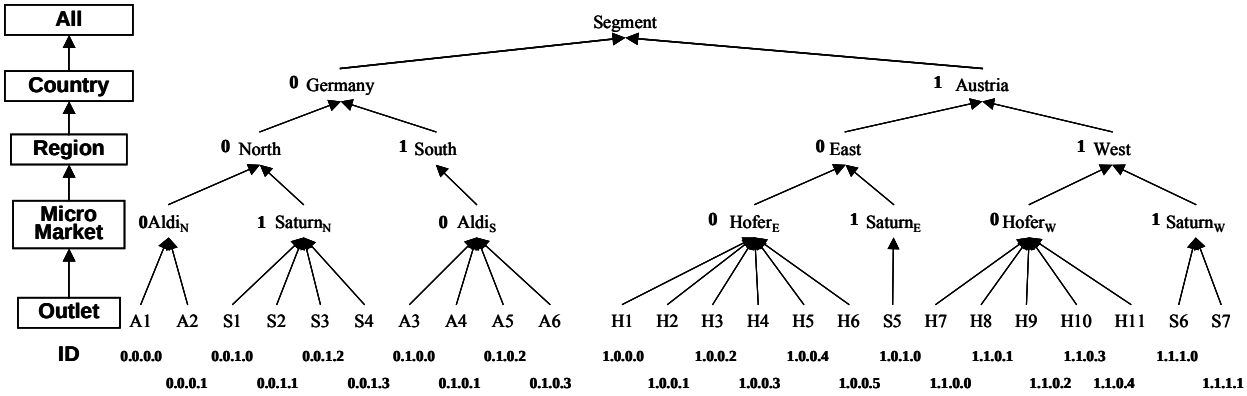


Figure 5-3: Sample Hierarchy with Enumeration Schema

5.3.3 Computation of Compound Surrogates

According to the definition of compound surrogates (see Definition 5-4), we define a recursive calculation formula for the compound surrogate of member m^i_j ([Mar99]):

$$cs(m^i_j) = \begin{cases} ord(m^i_j) & , \text{if } i = \text{t - 1} \\ cs\left(\text{father}(m^i_j)\right) \circ ord(m^i_j), & \text{otherwise} \end{cases}$$

For simple hierarchies, the function *father:* $h^i \rightarrow h^{i+1}$, $father(m^i_j) = m^{i+1}_k$, returns the predecessor (father) node in the hierarchy tree for levels $h^1, \ldots, h^{t-1}$, where h^t is the top level. This father member is a unique member, because every member has a unique father member. t is the depth of the hierarchy. The top hierarchy level h^t always contains one member (i.e., the “all” level) that is not used for the compound surrogate. Thus the recursion ends with the successor of the top level (i.e., the level h^{t-1}).

Each member is identified uniquely by a compound surrogate. A compound surrogate is calculated by concatenating the compound surrogate of the father member with the surrogate (*ord* function) of the current member.

The compound surrogate for a path Φ with $\Phi = \{ m^t_1, m^{t-1}_{km-1}, \ldots, m^i_{ki} \}$[9] is computed by

$$cs(\Phi) = cs(m^i_{ki}) = ord(m^{t-1}_{km-1}) \circ ord(m^{t-2}_{km-2}) \circ \ldots \circ ord(m^i_{ki})$$

A lexicographic order on compound surrogates is defined by ordering *cs* with respect to the components beginning with the most significant *cs* component (highest level).

Example 5-5 (Compound Surrogate):

The compound surrogate for "Hofer$_W$" of Figure 5-3 is the cs for the path "Austria" – "West" – "Hofer$_W$" and is computed by $cs(\text{"}Hofer_W\text{"}) = 1 \circ 1 \circ 0$.

The compound surrogate of the leaf member "A6" is calculated by the path "Germany" – "South" – "Aldi$_S$" – "A6" and has the value $cs(\text{"A6"}) = 0 \circ 1 \circ 0 \circ 3$. □

5.3.4 Bit Representation of Compound Surrogates

Instead of computing the compound surrogates via the recursive formula of Section 5.3.3, we describe a compact bit representation and an efficient calculation formula.

For every level, we reserve a number of bits contributing to the fanout of the level (see Definition 5-6). The number of bits reserved for cs component cs^i for level h_i is

$$l_{cs^i} = \lceil \log_2 \ fanout(h_i) \rceil.$$

We use l_i instead of l_{csi}. The complete compound surrogate requires l_{cs} bits:

$$l_{cs} = \sum_{i=1}^{t-1} l_i$$

A compound surrogate of path $\Phi = \{ m^t_1, m^{t-1}_{kt-1}, \ldots, m^i_{ki} \}$ is computed by

$$cs(\Phi) = cs(m^i_{ki}) = ord(m^{t-1}_{k_{t-1}}) \cdot 2^{l_{t-1}+l_{t-2}+\ldots+l_i} + ord(m^{t-2}_{k_{t-2}}) \cdot 2^{l_{t-2}+l_{t-3}+\ldots+l_i} + \ldots + ord(m^i_{k_i}) \cdot 2^{l_i}$$

The computation starts with level h^{t-1}, because the top level always contains one member that has the surrogate $ord(m^t_1) = 0$. The compound surrogate for a leaf member for a path of members $\Phi = \{ m_t, m_{t-1}, \ldots, m_1 \}$ is

$$cs(\Phi) = \sum_{i=1}^{t-1} ord(m_i) \cdot 2^{\sum_{j=1}^{i} l_j}$$

The compound surrogate has a fixed length, because all components have a fixed length.

[9] We use a path from the top member to member m^i_{ki}, because the order of cs components corresponds to this path. The arrows of the edges of the graph denote the opposite direction to show the hierarchical dependencies.

Example 5-6 (Computation of Compound Surrogate):

We assume the level fanouts of Table 5-5 for the hierarchy as illustrated in Figure 5-3. The compound surrogate of "A6" is

$cs(\text{"A6"}) = 0000.0001.0000000.00000000000011_b = 2097155_d.$

All *outlets* (i.e., the leaf level) of the segment hierarchy, are identified by 4+5+3+12 = 24 bits. □

Hierarchy Level	Fanout	Bits
Country	16	4
Region	19	5
Micromarket	6	3
Outlet	2202	12

Table 5-5: Fanout of Sample Hierarchy

Note that the size (length) of the compound surrogate components are fixed for each level. In reality, hierarchy instances may change (paths are inserted, updated or deleted). Even the schema of hierarchies may change. See Section 12 for a discussion about dynamic hierarchies.

5.3.5 Operations on Compound Surrogates

In this section we introduce the operations min_{cs}, max_{cs}, and $ival_{cs}$ on compound surrogates. They are useful to map a set of compound surrogates to intervals. These intervals are used to speed up query processing.

Definition 5-8 (min_{cs}):

The function min_{cs}: $h_i \rightarrow N_0$, returns the minimum compound surrogate cs_{min} of a member m^i_j: $cs_{min} = min_{cs}(m^i_j)$. The minimum compound surrogate is the lowest complete surrogate that contains the prefix *cs* from the top member to m^i_j. The remaining (least significant) bits are set to 0. Thus $min_{cs}(m^i_j) = cs(m^i_j) \circ 00..00$, where all bits of levels $h^{i-1}, h^{i-2}, \ldots, h^1$ are set to 0. □

Definition 5-9 (max_{cs}):

The function max_{cs}: $h^i \rightarrow N_0$ returns the maximum compound sorrogate cs_{max} of a member m^i_j : $cs_{max} = max_{cs}(m^i_j)$. Analogously to min_{cs}, the maximum compound surrogate of a member m^i_j is the highest compound surrogate containing m^i_j. The remaining bits are set to one: $max_{cs}(m^i_j) = cs(m^i_j) \circ 11..11$, where all bits of levels $h^{i-1}, h^{i-2}, \ldots, h^1$ are set to one. □

Definition 5-10 ($ival_{cs}$):

The function $ival_{cs}$: $h^i \rightarrow [N_0 ; N_0]$ returns an interval of the minimum and maximum compound surrogate of a member m^i_j: $ival_{cs}(m^i_j) = [(min_{cs}(m^i_j) ; max_{cs}(m^i_j)]$.

The interval resulting from $ival_{cs}$ is a closed interval. □

Example 5-7 (min_{cs}, max_{cs}, $ival_{cs}$):

The functions applied to the sample hierarchy return the following results:

$min_{cs}(\text{“Aldi}_S\text{”}) = 0000.0001.0000000.00000000000000_b = 2097152_d.$

$max_{cs}(\text{“Aldi}_S\text{”}) = 0000.0001.0000000.11111111111111_b = 2113535_d.$

$ival_{cs}(\text{“Aldi}_S\text{”}) = [2097152_d, 2113535_d]$ □

5.3.6 Remarks about Compound Surrogates

The number of bits to encode compound surrogates is quite small compared to the number of elements that such a compound surrogate can address. For example, a hierarchy tree with eight levels and eight branches per node for every level can store 8^8 = 16.777.216 elements in 24 bits ($\log_2$16.777.216).

However, the number of bits necessary to calculate the compound surrogates for a hierarchy depends on the instance of the hierarchy. If the hierarchy members are distributed regularly (i.e., the number of children members is about the same for every member of a level), a relative small number of bits is enough for the mapping, because the fanout of the hierarchy levels is quite small (see Figure 5-4). In Figure 5-4, 2 bits for level 3, 3 bits for level 2 and 1 bit for level 1 are enough to store 42 hierarchy elements (i.e., 6 bits for the complete hierarchy). In the case of an irregular distributed hierarchy as shown in Figure 5-5, 9 bits (2 for level 3, 4 for level 2 and 3 for level 1) are necessary to store the 42 hierarchy elements.

Often, the structure and the number of hierarchy members are not known in advance. In this case, some bits must be reserved to avoid overflows. Reserving one additional bit increases the fanout by a factor of two, two additional bits increase the fanout by a factor of four etc. Thus, a small number of additional reserved bits will reduce the chance of an overflow.

In data warehouses, some hierarchies are static (e.g. the time hierarchy or a geographical hierarchy). In such cases the compound surrogate specification can be optimal for the hierarchy. Minor changes on the hierarchy will seldom run into overflow problems.

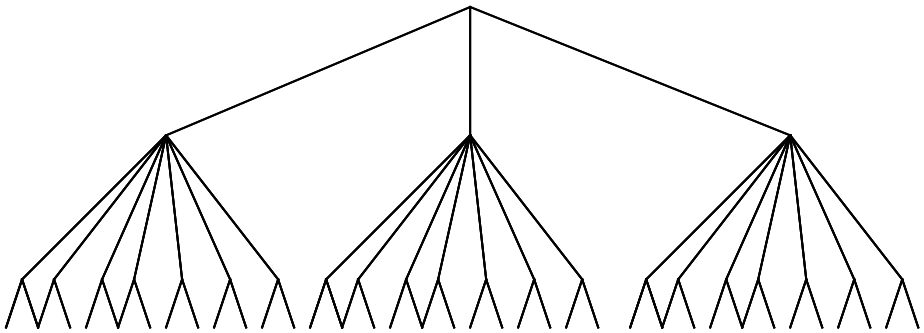


Figure 5-4: Hierarchy with regular Distribution

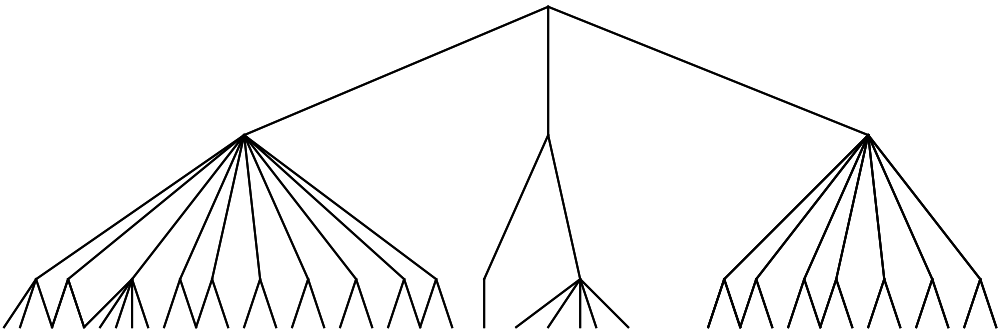


Figure 5-5: Hierarchy with irregular Distribution

6 EHC and Data Warehouses

This section contains a short introduction into EHC. Section 6.1 shortly describes the concept of EHC, in Section 6.2 we discuss the physical organization of the tables that use EHC.

6.1 Hierarchy and Encoding

The encoding for hierarchical clustering, EHC, establishes an enumeration schema for hierarchy elements. Physical access structures that are optimized for dealing with intervals (e.g., clustering indexes, such as B*-Tree or UB-Tree) gain advantage of EHC. EHC computes an artificial numeric key (compound surrogate) for every leaf dimension member with respect to the hierarchy path of this element.

Basically, EHC is useful for applications dealing with hierarchical data, e.g., classification hierarchies, XML etc. Especially, in data warehouse applications the dimensions are structured w.r.t. hierarchies. The multidimensional nature of a DW requires access structures to efficiently access data with predicates on several dimensions (attributes in the fact table). Multidimensional clustering access structures provide efficient multidimensional range query access. EHC is used to prepare hierarchical data for hierarchical "non-range" predicates, in order to efficiently deal with clustering multidimensional access structures (like UB-Trees).

For this purpose, we encode the hierarchy paths via compound surrogates (see Section 5.3). The resulting compound surrogates are used as a surrogate for the dimension key in the fact table and therefore as index key of the UB-Tree.

6.2 Physical Organization of Dimension and Fact Tables

EHC is a physical extension of the schema for dimension tables (with compound surrogates) and of the fact table (with reference surrogates). Both types of surrogate are stored in a separate attribute of bit string data type (see Section 7.3). The value of the attribute is dependent on one (reference surrogate) or more (compound surrogate) attributes of the table. The surrogate attributes are calculated attributes depending on other attributes and can be computed according to a function $surr_{ref}(a)$ and $surr_{comp}(a_t, \ldots, a_1)$.

6.2.1 Physical Organization of Dimension Tables

A compound surrogate *cs* is an additional attribute of the dimension table with special properties. It is dependent on a number of attributes of the dimension table: $cs = surr_{comp}(a_t, \ldots, a_1)$. In the Transbase® implementation, a special DDL construct specifies the components (and the order of the levels) of *cs*. a_t corresponds to the top level of the hierarchy and a_1 is the leaf level. a_1 additionally is primary key of the dimension table. Thus, there is a one-to-one relationship between a_1 and *cs*, both are candidate keys of the dimension table. In the following we choose a_1 as key.

The components (i.e., the attributes for the surrogates of cs) must be attributes of the dimension table. This means, that the dimension table includes all levels of the hierarchy of the dimension (a star schema w.r.t the hierarchy levels). Special algorithms are introduced in Section 9 to handle snowflake schemata. An alternative to forcing the user to store and maintain the hierarchy levels in the dimension table is to specify the components of *cs* from different tables and to physically store the corresponding components in *hidden* attributes (see Section 4.4) in the dimension table (or in a special join index).

Storing all hierarchy levels in one access structure is crucial to efficiently compute and maintain compound surrogates (see Section 8.2).

In the Transbase® implementation, the primary key (and clustering primary index attribute) is a_1, i.e., the leaf level of the hierarchy. With such an organization, the lookup to get the corresponding *cs* for a_1, is a simple B-Tree search. Two secondary indexes are necessary for computation and query processing. The index *DXh* is a secondary index on $(a_m, a_{m-1}, \ldots, a_1, cs)$ containing the hierarchy and the compound surrogate to efficiently compute a compound surrogate for a new hierarchy path (see Section 8.2). The index *DXcs* is a unique secondary index on *cs* to support query processing. These indexes are mandatory (and automatically created) and cannot be dropped by the user. For more information about the secondary indexes refer to Section 4.4.4.

Note that a_1 must be the key of the dimension table, because in the case of several hierarchies on the dimension, we have several paths as keys or several compound surrogates. Both of them (the hierarchy path and the compound surrogate) are also key candidates of dimension tables.

6.2.2 Physical Organization of Fact Table

The fact table FACT contains reference surrogates cs_{ref}, one for every dimension organized by compound surrogates. The reference surrogates usually are index attributes for the UB-Tree[10]. In the physical schema of the fact table, we replace the dimension key attributes by the corresponding reference surrogate attributes. The dimension key values are obtained from the corresponding dimension table via the surrogate reference (see Figure 6-1).

The fact table has the following attributes:

$$\texttt{fact\_physical}\ (cs_1, cs_2, \ldots, cs_n, \{m_i\}),$$

where cs_k are the reference surrogates (cs_1 references the compound surrogate of D_1, etc.) and m_i are the measure attributes of the fact table. Note that we call this fact table *fact_physical*, because it is the physical representation of the fact table.

Because surrogates are physical constructs and should not be visible for the user, we define a view on the fact table to hide the surrogates and make the dimension keys accessible (see also Section 4.4.1):

```
CREATE VIEW fact (d1, d2, …, dn, {mi}) AS
SELECT D1.h1, D2.h1, …, Dn.h1, {F.mi}
FROM fact_physical F, D1, D2, …, Dn
WHERE F.cs1 = D1.cs AND F.cs2 = D2.cs AND … AND F.csn = Dn.cs
```

If there are multiple hierarchies on one dimension that are represented by compound surrogates, we store one reference surrogate for each hierarchy in the fact table fact_physical. Thus, we adapt the view definition accordingly.

$$\texttt{fact\_physical}\ (cs_1^1, cs_1^2, \ldots, cs_1^{k1}, cs_2^1, cs_2^2, \ldots, cs_2^{k2}, \ldots, cs_n^{kn}, \{m_i\}),$$

```
CREATE VIEW fact (d1, d2, …, dn, {mi}) AS
  SELECT D1.h1, D2.h1, …, Dn.h1, {F.mi}
  FROM fact_physical F, D1, D2, …, Dn
  WHERE F.cs1^1 = D1.cs1 AND F.cs2^1 = D2.cs1 AND … AND F.csn^1 = Dn.cs1
```

[10] It is not required that the reference surrogates are keys of the UB-Tree (there could be too many dimensions for an appropriate multidimensional clustering), the optimizer also can make use of EHC organization of dimensions without being part of the clustering. See Section 110.5 for detailed discussion.

Note that all compound surrogates of one dimension are determined by h^1 (shared leaf level of all hierarchies and primary key of the dimension table). Thus it is sufficient to specify the dimension key of the view *fact* by a join from one reference surrogate to the corresponding compound surrogate in the dimension table.

Hierarchy changes may lead to changes on the compound surrogates and thus will also change reference surrogates of the fact table (see Section 8.2.3). The following section describes the relationships and necessary physical constructs to achieve correct surrogate resolution.

6.2.3 Surrogates and Reference Constraints

For correct surrogate handling (i.e., computation, maintenance and query processing) some dependencies are necessary. For a dimension with one hierarchy (and thus one compound surrogate), we have to specify the foreign key dependency of Figure 6-1.

Within D_i, the compound surrogate *cs* depends on the hierarchy path $\Phi(h^t, h^{t-1}, \ldots, h^1)$, i.e., *cs* is functional dependent on Φ: $(h^t, h^{t-1}, \ldots, h^1) \rightarrow cs$, where h^t is the top level of the hierarchy. However, *cs* determines the path $\Phi(h^t, h^{t-1}, \ldots, h^1)$, i.e., Φ is functional dependent on *cs*: $cs \rightarrow \Phi$. Note that these dependencies are expressed in DDL statements (Section 7.6).

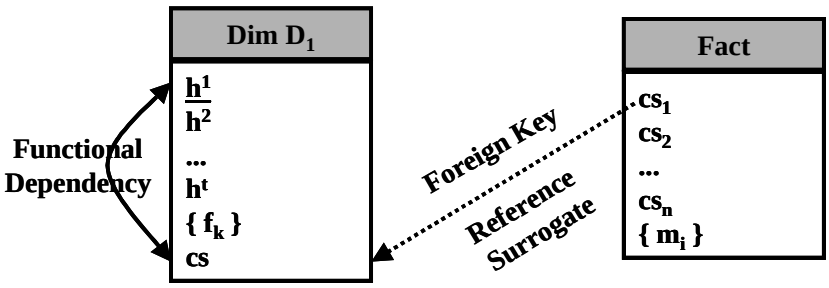


Figure 6-1: Dependencies of Dimension and Fact Table

For dimensions with several hierarchies (and several compound surrogates) these dependencies also must be fulfilled. In this case some DDL extensions are mandatory for an exact specification of the relationships (see Section 7.6.4).

In Figure 6-2 dimension D_1 has two hierarchies $h^3 - h^2 - h^1$ and $h^5 - h^4 - h^1$. The leaf level must be the same for both hierarchies, because it is the dimension key and therefore is unique for the dimension and for all hierarchies of this dimension. In this example the compound surrogates of both hierarchies are used as index attributes in *Fact* (two reference surrogates in *Fact*).

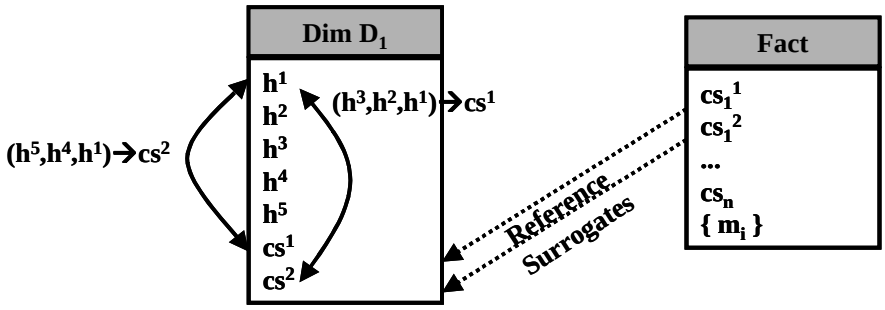


Figure 6-2: Dependencies of Dimension and Fact Table with multiple Hierarchies

In the case of several hierarchies as shown in Figure 6-2, it is not necessary to store all compound surrogates in the fact table and use them for hierarchical clustering. However, for grouping optimization (see Section 9.4), it is beneficial to store all compound surrogates in the fact table.

7 Integration of EHC and MHC into a DBMS

This section discusses basic implementation steps of the Transbase® implementation, such as the data type for surrogates, changes of the data dictionary, physical schema extensions, and the extension of the DDL. An example with DDL statements closes this section.

In contrast to Section 6.2.2 we use a modified physical representation of the fact table. For the sake of easier maintenance we store the dimension keys also in the fact table $fact_{physical}$ (replication from the dimension tables)[11]:

$fact_{physical}$ (d_1, d_2, …, d_n, {mi}, cs_1, cs_2, …, cs_n)

This is due to the fact that tables are standalone objects that must be maintained (created, dropped, spooled into flat files etc.) themselves. The reference surrogates cs_k are hidden from the user (they only occur in the DDL statement, see Section 7.6), the tuples of the fact table are identified by the dimension keys (at least for mass loading and inserting). Thus, we define a view *fact* on the fact table $fact_{physical}$ that is available to the user and occurs in the star join queries:

```
CREATE VIEW fact (d1, d2, …, dn, {mi}) AS
SELECT d1, d2, …, dn, {mi} FROM fact_physical
```

Note that in Transbase®, inserting tuples into views generally is allowed as far as views only concern one single table (i.e., no join is involved). Thus a physical representation as in Section 6.2.2 is not possible. In the following sections we assume the physical schema of *fact_physical* as described above.

7.1 Basic Requirements

For the integration of MHC into the Transbase® DBMS, it is postulated, that all parts of the DBMS support MHC. Thus, the user interface, such as programming interfaces, ODBC and JDBC drivers, the data definition and data manipulation language, mass loading and spooling, archiving and schema development tools must be adapted and integrated. Additionally, many internal structures and processes are extended to support MHC, such as the system catalog, internal structures to maintain MHC information, the query processor and optimizer. In addition to these MHC specific concepts, some general DBMS concepts had to be implemented that were not yet available in Transbase®, e.g., the SQL-92 full level reference constraint support (ON UPDATE CASCADE and ON DELETE CASCADE) and a general approach for hash tables for query processing (hash group, hash join etc.).

All extensions are implemented on the general EHC/MHC functionality, especially multiple hierarchies lead to some additional effort for the design and implementation phases.

7.2 General Transbase® Architecture

Transbase® is a modularly designed DBMS with modules and interfaces easy to extend and adapt to new requirements. Some layers are considerably more affected by the EHC/MHC integration, some have minor changes and some layers are not modified at all. Figure 7-1 gives an overview which layers are modified in which extent.

[11] For dimensions with multiple hierarchies the schema for the fact table is extended by the additional reference surrogates (see Section 16.2.2), but only one dimension key per dimension is added.

The lowest layers that handle the object administration, locking, logging and recovery are not changed at all. The *Catalog Manager* has minor changes (system catalog). The *SQL and DDL Compiler* are affected in a larger extent, because the extensions of the DDL require some basic extensions. Most changes occur in the *Query Processor* and *Query Optimizer* with sophisticated MHC algorithms to efficiently optimize and process star join queries, e.g., surrogate specific computations for maintenance queries, checks for surrogate handling etc..

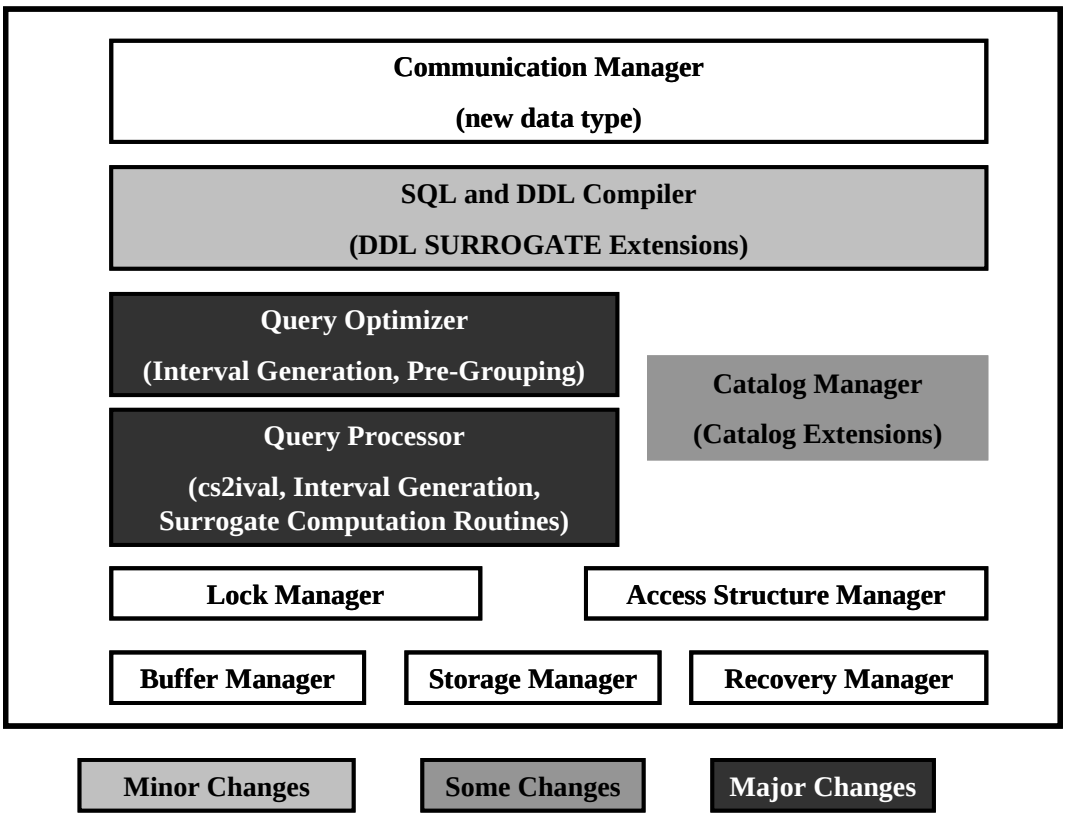


Figure 7-1: Basic Transbase® Architecture

The following sections describe the extensions in more detail, especially query processing and query optimizing are discussed in a large extent since these are crucial for a successful EHC/MHC implementation.

In addition to Transbase® kernel extensions, also tools to administrate databases are affected. Due to changes on the system catalog, a database created with a prior Transbase® version requires a migration to Transbase®/MHC, because additional system tables must be created. This also includes tools to archive and recover databases.

Additionally, the tools to check the correctness of databases must contain a consistency check of compound and reference surrogates.

Major extensions are necessary for the interactive database access tools that contain methods to get the physical schema of tables etc. In this case, some mechanisms to hide information (e.g., system indexes) are implemented to prohibit the user to remove necessary physical structures.

7.3 Introducing a new data type

Compound surrogates are represented by numbers that may get very large and thus would not fit into a four byte integer value. On the other side, for small hierarchies a number of bits is necessary that may be less than 32, such that space is wasted when using integer data type.

The most efficient way to represent compound surrogates is to use only the necessary number of bits (so called bit strings). The already existing *BITS* data type is declared with a fixed number of bits *(BITS(n))* or with a variable number of bits *(BITS(*))*. This data type consists of two components, the first component contains the number of bits stored in the bit string. The second component is the bit string. Because a byte is the finest granularity to store information, the space needed by the bit string is a four byte integer (for the length) and the number of bytes required to store *n* bits $(\lceil n/8 \rceil)$.

For conventional hierarchies, the number of bits required to represent the compound surrogate usually is in the range of 16 to 64 bits. E.g., the compound surrogate for the product hierarchy of the SALES DW needs 22 bits (3 byte for the bit combinations). Thus the four byte integer for the first BITS component is dimensioned too large for compound surrogates (a four byte integer can represent 2^{32}-1=4.294.967.295). A length field of a two byte integer (can store up to 65.535 bits) usually is enough to represent realistic hierarchies[12]. Using this new length description type, we reduce the space significantly to store compound surrogates (e.g., from 7 to 5 byte for the product hierarchy, i.e., more than 25 percent). We call this new data type *BITS2*[13].

Note that in an MHC organized fact table, these compound surrogates and therefore the bit strings are used as index attributes. The multidimensional UB-Tree index benefits from the space savings, because more index tuples can be stored in the leaf pages of the underlying B-Tree which reduces access and maintenance costs and increases the number of tuples fitting on one page. The size of the fact table is reduced, because every dimension organized by EHC requires at least one compound surrogate.

7.4 Extending the System Catalog

Each DBMS has a *data dictionary* or a *system catalog* to persistently store meta information about tables, views, indexes, constraints, users, privileges etc. Tables with compound or reference surrogates require persistent information, such as the fanout of the levels for a compound surrogate or the relationship between compound and reference surrogates etc.

In Transbase®, we extend some existing system tables and introduce a new system table especially for compound surrogates. The system table *syscolumn* holding information about the attributes of a table (e.g., name, data type, default value, null specification etc.) is extended by the attributes *surrid* and *surref*. surrid contains an identifier for the surrogate (also used as foreign key to the new system table *syssurrogates*) and is zero, if the attribute is no surrogate. The same holds for *surref*: *surref* is not zero, if the attribute is a reference surrogate (i.e., a reference surrogate of the fact table). In this case, *surref* contains the attribute position of the corresponding dimension key attribute of the fact table.

The system table *syssurrogate* specifies the compound surrogates in more detail. *syssurrogate* contains one tuple for each component of the compound surrogates (i.e., hierarchy level). Each compound surrogate has a unique identifier *surrid* used also in *syscolumn*. As redundant information the segment number of the table with the compound surrogate is stored in order to get this information efficiently for catalog queries. For each level we store the attribute position of the corresponding hierarchy level attribute of the dimension table. The attribute *siblings* in *syssurrogate* contains the fanout of the level represented as the number of necessary bits to hold the siblings.

7.5 Indexing and Access Paths

Secondary indexes are necessary for efficient computation and maintenance of compound surrogates, for query processing etc.. The two secondary indexes *DXh* and *DXcs* are created automatically for

[12] Such a hierarchy can store up to $2^{65535} \approx 10^{19728}$ hierarchy members.

[13] This data type is also used to represent z-values of the UB-Tree (in the index part of the B*-Trees and for internal representation).

each compound surrogate. These *system indexes* cannot be dropped as long as the dimension table exists.

DXh is a unique index containing the hierarchy for the compound surrogate (see Section 4.4.4): *DXh* = $(h^t, h^{t-1}, \ldots, h^1, cs)$, where h^t is the top level of the hierarchy. The hierarchy levels are attributes of the leaf dimension table, but could also be distributed over several dimension tables. In this case, *DXh* is a join index across many dimension tables. In the Transbase® implementation, the LDT must contain all hierarchy levels. This index is used for the computation of the compound surrogates (see Section 8.1), surrogate maintenance (decision whether a compound surrogate must be recomputed) and for query processing (computation of cs prefixes to get the intervals for the restriction on the UB-Tree, see Section 9).

DXcs is a unique index holding the compound surrogate of the LDT. This index is necessary for the computation of compound surrogates, especially for hole searches, and for some steps in the query processing.

7.6 Extending DDL

The SQL standards, SQL-92 and SQL-99, do not explicitly support the concept of hierarchies. Usually hierarchies are modeled by specifying *reference constraints* (foreign key references) representing hierarchical relationships. No explicit language constructs are provided in order to semantically create hierarchies.

Some DBMS, however, have such language extensions. For example, Oracle provides the declaration of dimensions and hierarchies. This logical concept is on top of the physical schema providing additional information for the optimizer to recognize the data warehouse schema and generate query execution plans accordingly.

Because of the physical nature of EHC and MHC, we introduce the concept of dimensions and hierarchies as a physical property of the DW schema by extending the DDL for the `CREATE TABLE` statement.

In order to compare the concept used in the Transbase® implementation to the concept of Oracle, we first describe the concept of hierarchies in Oracle and then discuss the specification of compound and reference surrogates in Transbase® with simple and complex hierarchies.

7.6.1 Dimensions and Hierarchies in Oracle

In Oracle, the concept of dimensions and classification structures on dimensions like hierarchies differs from the physical concept as discussed in this thesis. Oracle implements a logical warehouse concept on top of the physical schema. Dimensions and hierarchies are created on existing tables.

7.6.1.1 General Concept

Dimensions can be denormalized (star schema) or normalized (snowflake schema). Hierarchies may consist of levels (attributes) from different tables.

Each dimension can contain one or more hierarchies. Each dimension consists of a set of so called *levels*, i.e., usually hierarchy levels. The hierarchy levels are a collection of levels of several hierarchies for the dimension and can reside in different tables. There is no hierarchical dependency in the dimension. The hierarchical relationships are defined in a separate `HIERARCHY` clause. For every hierarchy, the level *ALL* is created implicitly. A hierarchy level can have one or more feature attributes. A feature attribute is functionally dependent on the hierarchy levels and is specified via the `DETERMINES` clause (see Section 7.6.1.2).

Because of the logical concept of dimensions and hierarchies, the physical organization of the fact table is not affected by the hierarchies. Thus, changes on the hierarchies, dimensions etc. are possible. Only meta information has to be changed.

For the hierarchies, a *1:n* relationship between parent and child levels and a *1:1* relationship between hierarchy levels and feature attributes is required. For normalized dimensions, i.e., when parent and child levels are in different relations, they must be in a *1:n* join relationship (ensured by corresponding reference constraints).

7.6.1.2 DDL Statements

Now we show the DDL constructs to create a new dimension and the corresponding hierarchies. In this example, the table *segment* is denormalized. In general, the dimension table could be normalized in any way. The `CREATE DIMENSION` statement contains one or more `HIERARCHY` clauses, each representing one hierarchy. Every hierarchy has a number of levels and functional dependent feature attributes. For the sample DW, the segment dimension is specified as follows:

```
CREATE TABLE segment (
  country_id          INTEGER,
  country_txt         CHAR(*),
  region_id           INTEGER,
  region_txt          CHAR(*),
  micromarket_id      INTEGER,
  micromarket_txt     CHAR(*),
  turnoverclass_id    INTEGER,
  turnoverclass_txt   CHAR(*),
  outlet_id           INTEGER,
  outlet_txt          CHAR(*)    );
```

Note, that the physical representation of the dimension table can be any star schema. In this case, the primary key of the dimension table segment is *outlet_id* (as dimension key of the dimension segment).

```
CREATE DIMENSION segment_dim
  LEVEL country       IS segment.country_id
  LEVEL region        IS segment.region_id
  LEVEL micromarket   IS segment.micromarket_id
  LEVEL turnoverlcass IS segment.turnoverclass_id
  LEVEL outlet        IS segment.outlet_id
HIERARCHY segment_geo (
  outlet        CHILD OF
  micromarket   CHILD OF
  region        CHILD OF
  country )
ATTRIBUTE country     DETERMINES segment.country_txt
ATTRIBUTE region      DETERMINES segment.region_txt
ATTRIBUTE micromarket DETERMINES segment.micromarket_txt
ATTRIBUTE outlet      DETERMINES segment.outlet_txt
HIERARCHY segment_class (
  outlet        CHILD OF
  turnoverclass CHILD OF
  country )
ATTRIBUTE country          DETERMINES segment.country_txt
ATTRIBUTE turnoverclass    DETERMINES segment.turnoverclass_txt
ATTRIBUTE outlet           DETERMINES segment.outlet_txt
;
```

The above SQL block is one SQL statement (finished by the semicolon). In this example the dimension is assigned to the segment dimension table (specified by `LEVEL` *`lev`* `IS table.attribute`, e.g., `LEVEL country IS segment.country_id`).

Two hierarchies, *segment_geo* and *segment_class* are defined on the dimension *segment_dim*. There can be shared levels between the hierarchies (in the example: *outlet_id*).

The `ALTER DIMENSION … DROP` statement removes, the `ALTER DIMENSION … ADD` statement adds dimension attributes. Via this statement, dimension attributes, hierarchies and levels of hierarchies are maintained.

7.6.2 Compound Surrogates

In contrast to the logical approach of dimensions and hierarchies in Oracle, compound surrogates in Transbase® are an additional physical attribute of the leaf dimension table with the data type *BITS2*. The `CREATE TABLE` statement is extended by a `SURROGATE .. COMPOUND` clause. For the compound surrogate, the levels of the hierarchy are specified via a sequence of hierarchy levels and their fanout (`SIBLINGS`).

```
CREATE TABLE dim_segment (
     country_id       INTEGER NOT NULL,
     country_txt      CHAR(*),
     region_id        INTEGER NOT NULL,
     region_txt       CHAR(*),
     micromarket_id   INTEGER(*) NOT NULL,
     micromarket_txt  CHAR(*),
     outlet_id        INTEGER NOT NULL
     outlet_txt       CHAR(*),
     SURROGATE cs_segment COMPOUND (country_id SIBLINGS 16,
          region_id SIBLINGS 19, micromarket_id SIBLINGS 6,
          outlet_id SIBLINGS 2202)
) KEY IS outlet_id;
```

The SQL statement above shows the specification of the geographic hierarchy of the segment dimension of the sample DW. The `SURROGATE` clause is emphasized via bold style and includes the name of the compound surrogate (*cs_segment*) and the hierarchy levels (*country_id, region_id, micromarket_id,* and *outlet_id*) in the order from top level to leaf level. These levels are stored in the system catalog in *syssurrogates* (see Section 7.4). The hierarchy levels must have the `NOT NULL` constraint, thus all hierarchy paths are fully specified. It is also required that every hierarchy level occurs only once in the hierarchy (no circles in the hierarchy) and the leaf level of the hierarchy must be the primary key in the dimension table.

The specification of `SIBLINGS`[14] determines the computation of compound surrogates as they denote the maximal fanout of the levels (see Section 5.3.3). In the compound surrogate *cs_segment* we define 4 bits for the first (SIBLINGS=16), 5 bits for the seconds (SIBLINGS=19), 3 bits for the third (SIBLINGS=6) and 12 bits for the fourth cs component (SIBLINGS=2202). With these information, the computation formula of compound surrogates is defined exactly. Note that the *ord* function of Section 5.3.2 is used for the computation of compound surrogates implicitly by index lookups and usually depends on insertion order (see Section 8.1 for more details).

[14] We used the notion `SIBLINGS` in order to specify the fanout, because there might be a misunderstanding what fanout means (the number of successors of the level).

Note further, that in the example the attributes with suffix *_id* are unique hierarchy level identifiers, whereas the attributes with suffix *_txt* are descriptive attributes, i.e., feature attributes (there might be duplicates of descriptive attributes for the different hierarchy level identifiers).

A formal specification of the DDL extension is found in Section 7.6.5.

7.6.3 Reference Surrogates

Reference surrogates are part of the fact table. Each reference surrogate is stored in an attribute of type *BITS2* (such as the corresponding compound surrogates in the leaf dimension table). As described in Section 6.2, the reference surrogate depends on the corresponding dimension key attribute in the fact table. The value of the reference surrogate is determined by the corresponding compound surrogate in the dimension table. These two dependencies are expressed in the `CREATE TABLE` statement for the fact table.

```
CREATE TABLE fact (
      dseg  INTEGER REFERENCES dim_segment(outlet_id)
            ON UPDATE CASCADE,
      dprod INTEGER REFERENCES dim_product(item_id)
            ON UPDATE CASCADE,
      dtime INTEGER REFERENCES dim_time(month2period_id)
            ON UPDATE CASCADE,
      turnover NUMERIC(10,2)
      …
      SURROGATE cs_seg REFERENCES dim_segment(cs_segment),
      SURROGATE cs_prod FOR dprod,
      SURROGATE cs_time FOR dtime
) HCKEY is cs_seg, cs_prod, cs_time;
```

The dimension attribute of the fact table must reference the corresponding dimension key of the LDT in a reference constraint with the `ON UPDATE CASCADE` extension.

Basically, two different kinds of `SURROGATE` clauses are possible. If the dimension table contains only one compound surrogate (one hierarchy in the dimension), the `SURROGATE … FOR` clause is sufficient expressing the functional dependency of the reference surrogate (via the reference constraint of the corresponding compound surrogate of the dimension table).

The `SURROGATE … REFERENCES` clause can be used alternatively, if only one reference constraint is defined between the fact table and the corresponding dimension table. The `REFERENCES` clause **must** be used, if multiple hierarchies are defined on the dimension, i.e., more than one compound surrogates are specified in the LDT.

In the case of several reference constraints between fact and dimension table and multiple hierarchies on the dimension, the combination of `FOR` and `REFERENCES` clauses is required to exactly specify the relationships and functional dependencies of the reference surrogates.

The reference surrogates can be used as index attributes of the UB-Tree (`HCKEY IS cs_seg, …`).

7.6.4 Multiple Hierarchies

In real world scenarios, dimensions often have multiple hierarchies. In such a case, it is difficult to decide which hierarchy to prefer and use for physical clustering. Thus, the implementation of EHC/MHC provides the possibility to define several hierarchies and therefore compound surrogates on one LDT (see Section 6.2) and use them as clustering index attributes.

All hierarchies of one dimension must have a common leaf level, because this level is the dimension key and thus the foreign key of the fact table. If more than one hierarchy is used for physical clustering in the fact table we define a reference surrogate for each hierarchy. The following DDL statements show the definition of two alternative hierarchies for the segment dimension of the sample DW.

```
CREATE TABLE dim_segment (
      country_id            INTEGER NOT NULL,
      country_txt           CHAR(*),
      region_id             INTEGER NOT NULL,
      region_txt            CHAR(*),
      micromarket_id        INTEGER NOT NULL,
      micromarket_txt       CHAR(*),
      turnoverclass_id      INTEGER NOT NULL,
      turnoverclass_txt     CHAR(*),
      outlet_id             INTEGER NOT NULL,
      outlet_txt            CHAR(*),
      SURROGATE cs_geo COMPOUND (country_id SIBLINGS 16,
           region_id SIBLINGS 19, micromarket_id SIBLINGS 6,
           outlet_id SIBLINGS 2202)
      SURROGATE cs_class COMPOUND (country_id SIBLINGS 16,
           turnoverclass_id SIBLINGS 50, outlet_id SIBLINGS 15000)
) KEY IS outlet_id
```

The dimension table segment contains two hierarchies: *country – region – micromarket – outlet* and *country – turnoverclass – outlet*. Every compound surrogate must have a unique identifier within the dimension table. The dimension table is created with two additional attributes, one for every compound surrogate. The common leaf level of the hierarchies (i.e., the least significant component of the compound surrogates) is the primary key of the table: *outlet_id*.

The fact table contains both compound surrogates as reference surrogates:

```
CREATE TABLE fact (
      dseg  INTEGER REFERENCES dim_segment(outlet_id)
           ON UPDATE CASCADE,
      dprod INTEGER REFERENCES dimp_roduct(item_id)
           ON UPDATE CASCADE,
      dtime INTEGER REFERENCES dim_time(month2period_id)
           ON UPDATE CASCADE,
      turnover NUMERIC(10,2)
      …
      SURROGATE cs_seg_geo REFERENCES dim_segment(cs_geo),
      SURROGATE cs_seg_class REFERENCES dim_segment(cs_class),
      SURROGATE cs_prod FOR dprod,
      SURROGATE cs_time FOR dtim
) HCKEY is cs_seg_geo, cs_seg_class, cs_prod, cs_time;
```

It is required to use the `REFERENCES` clause for the reference surrogate specification of the dimension segment, otherwise it is not decidable which compound surrogate of *dim_segment* is referenced. The reference surrogates *cs_geo* and *cs_class* are assigned to the attribute *dseg*, because there is only one foreign key reference from *fact* to *dim_segment*. If there are more foreign key references, the combination of `FOR` and `REFERENCES` clause is necessary:

```
CREATE TABLE fact (
     dseg  INTEGER REFERENCES dim_segment(outlet_id)
           ON UPDATE CASCADE,
     …
     SURROGATE cs_seg_geo FOR dseg REFERENCES dim_segment(cs_geo),
     … )
```

7.6.5 Formal DDL Specification

The previous sections already showed some example DDL statements. In this section we present the formal DDL specification for compound and reference surrogates embedded in the `CREATE TABLE` statement. Words written in upper case are key words, the remaining expressions are syntactic variables that are combined to get the complete statement. Note that there are some Transbase® specific extensions of the standard DDL ([Tra01]).

```
CreateTableStatement ::=
    CREATE TABLE TableName [ IkSpec ]
    ( TableElem [ , TableElem ] ... )
    [ KeySpec ]
```

The create table statement basically is similar to the DDL statements of many other relational DBMS with a table name and a sequence of table elements that are attributes (fields) or constraints. *IkSpec* is an extension that specifies whether to use the *internal key* (*rowid*), i.e., whether secondary indexes can be created on this table or not.

The constraint definitions `TableConstraintDefinition` and `FieldConstraint-Definition` correspond to the SQL-92 standard including `FOREIGN KEY`, `NOT NULL`, `DEFAULT` and `CHECK` constraints.

```
IkSpec ::=
    { WITH | WITHOUT } IKACCESS

TableElem ::=
    FieldDefinition
  | TableConstraintDefinition

FieldDefinition ::=
    StandardFieldDefinition
  | SurrogateDefinition
```

The surrogate definition belongs to the field definition, because an additional attribute is created and maintained for each surrogate.

```
StandardFieldDefinition ::=
    FieldName  DataTypeSpec   [ DefaultClause ]
    [ FieldConstraintDefinition  ] ...

SurrogateDefinition ::=
    SURROGATE SurrName CompoundDefinition
  | SURROGATE SurrName ReferenceDefinition

CompoundDefinition ::=
    COMPOUND CompoundList

CompoundList ::=
     (FieldName SIBLINGS Number [,FieldName SIBLINGS Number] … )
```

```
ReferenceDefinition ::=
    ForClause ReferencesClause
  | ForClause
  | ReferencesClause

ReferencesClause ::= REFERENCES TableName(SurName)

ForClause ::= FOR FieldName
```

7.6.6 Sample Schema

This example shows the DDL of the complete sample schema.

```
CREATE TABLE dim_product (
      sector_id        INTEGER NOT NULL,
      sector_txt       CHAR(*),
      category_id      INTEGER NOT NULL,
      category_txt     CHAR(*),
      productgroup_id  INTEGER NOT NULL,
      productgroup_txt CHAR(*),
      item_id          INTEGER NOT NULL,
      item_txt         CHAR(*),
      SURROGATE cs_prod COMPOUND (sector_id SIBLINGS 14,
            category_id SIBLINGS 9, productgoup_id SIBLINGS 83,
            item_id SIBLINGS 15601)
) KEY is item_id;

CREATE TABLE dim_segment (
      country_id            INTEGER NOT NULL,
      country_txt           CHAR(*),
      region_id             INTEGER NOT NULL,
      region_txt            CHAR(*),
      micromarket_id        INTEGER NOT NULL,
      micromarket_txt       CHAR(*),
      turnoverclass_id      INTEGER NOT NULL,
      turnoverclass_txt     CHAR(*),
      outlet_id             INTEGER NOT NULL,
      outlet_txt            CHAR(*),
      SURROGATE cs_geo COMPOUND (country_id SIBLINGS 16, region_id
            SIBLINGS 19, micromarket_id SIBLINGS 6,
            outlet_id SIBLINGS 2202)
      SURROGATE cs_class COMPOUND (country_id SIBLINGS 16,
            turnoverclass_id SIBLINGS 50, outlet SIBLINGS 15000)
) KEY IS outlet_id;

CREATE TABLE dim_time (
      year       INTEGER NOT NULL,
      month      INTEGER NOT NULL,
      day        INTEGER NOT NULL,
      SURROGATE cs_time COMPOUND(year SIBLINGS 10, month SIBLINGS 12,
            day SIBLINGS 31)
) KEY is day;
```

```
CREATE TABLE fact (
      dseg  INTEGER REFERENCES dim_segment(outlet_id)
           ON UPDATE CASCADE,
      dprod INTEGER REFERENCES dim_product(item_id)
           ON UPDATE CASCADE,
      dtime INTEGER REFERENCES dim_time(day) ON UPDATE CASCADE,
      turnover NUMERIC(10,2),
      …
      SURROGATE cs_seg REFERENCES dim_segment(cs_geo),
      SURROGATE cs_prod FOR dprod,
      SURROGATE cs_time FOR dtime
) HCKEY is cs_seg, cs_prod, cs_time;
```

Note that the reference surrogate for the alternative hierarchy cs_class for the dimension dim_segment can be incleded into the multidimensional index of the fact table. This example is shown in Section 7.6.4.

8 EHC Processing

EHC influences the maintenance of tables and query processing, because the compound surrogates replace predicates on hierarchies and they have to be maintained, if hierarchy members are changed. This section contains a description of the maintenance operations, e.g., if hierarchy paths are inserted, modified or deleted, or if data is inserted into fact tables. Section 9 describes the query processing within an MHC/EHC organized schema.

8.1 Computation of Compound Surrogates

The basic operation for EHC is the computation of the compound surrogates according to the hierarchy paths. This section describes the algorithms to compute compound surrogates.

Informally, an insert (or update) of a tuple in the LDT, i.e., the insert of a new hierarchy path, triggers the computation of the corresponding surrogate cs depending on the existing paths of the dimension. For this purpose, we first compute the common prefix of the path as fixed (matching) component of cs. Then the distinguishing component is computed, usually the maximum cs component (with the same prefix path) incremented by one. The remaining components to the leaf level are filled with 0.

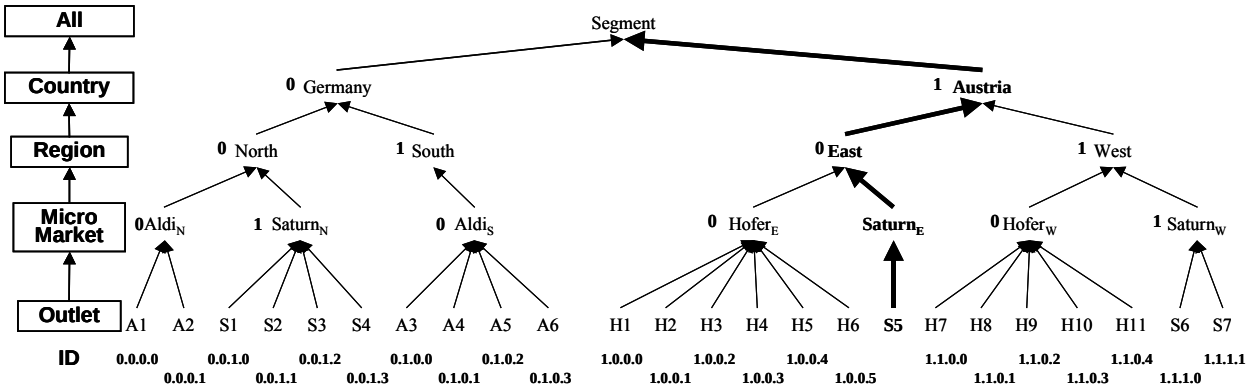


Figure 8-1: Insert of new Hierarchy Path

Figure 8-1 shows the insert operation of a new hierarchy path Φ=*("Segment", "Austria", "East", "Saturn$_E$", "S5")*, where the common prefix with existing paths is ("Segment", "Austria", "East"). The matching level is 3 (hierarchy level *Region*, leaf level *Outlet* is 1, etc.). Thus the cs components cs_4 and cs_3 (of the common prefix path) are $cs_4=1$ ("Austria") and $cs_3=0$ ("East"). The following level *MicroMarket* is assigned a new surrogate *s*, i.e., the largest surrogate of the children of the matching level incremented by one. In this case there is only one child "Hofer$_E$" that has the surrogate 0. Thus, the new surrogate *s* for "Saturn$_E$" is $s = 0 + 1 = 1$. The remaining level, i.e., level *Outlet*, is set to 0. Thus, the compound surrogate for Φ is *1.0.1.0*.

For an efficient computation of cs, we use two indexes *DXh* and *DXcs* (see Section 7.5), where *DXh* contains the levels of the hierarchy and cs, $DXh = (h^t, h^{t-1}, \ldots, h^1, cs)$, and DXcs contains *cs*.

A compound surrogate *cs* consists of components $cs_1, cs_2, \ldots, cs_t$ for *t* hierarchy levels: $cs = (cs_t, cs_{t-1}, \ldots, cs_1)$, each representing the enumeration of the corresponding level $h^t, h^{t-1}, \ldots, h^1$.

In the following, we call the new compound surrogate for path Φ: cs_{new}.

The following sections describe the steps in more detail that are necessary to compute the new compound surrogate.

8.1.1 Matching Level

Definition 8-1 (Matching Level, Prefix Path):

The *matching level* is the lowest level of the longest common path from top level to leaf level of the new tuple (inserted path) compared to the existing tuples. We call the path *prefix path*. □

The matching level is necessary to compute the final compound surrogate cs_{new} consisting of fixed cs components (due to the prefix path) and of new computed components (from the leaf level to the child of the matching level, see Figure 8-1).

For an efficient computation of the matching level, we use the clustering property of B-trees. The secondary index *DXh* stores the hierarchy paths and compound surrogates in the lexicographical order of the hierarchy levels[15] (from top level to leaf level). A `search` in DXh with search argument $\Phi=(m^t, m^{t-1}, \ldots, m^1)$, i.e., the new path, positions **between** the (lexicographically) "next smallest" and "next largest" tuple[16], if the tuple does not exist in the B-tree[17]. Figure 8-2 shows this positioning. For tuple t_{new}, where $t_5 < t_{new} < t_6$, the B-tree scan points between t_5 and t_6. The `prevread` operation returns t_5 and the `nextread` operation returns t_6.

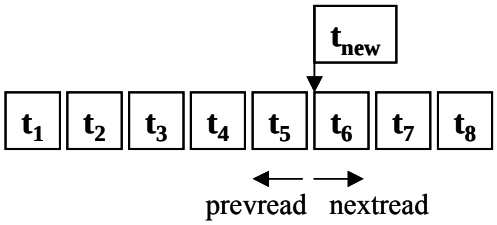


Figure 8-2: Positioning in Transbase® B-Tree

We call the next smaller tuple (i.e., the left tuple) t_{left} and the next larger tuple t_{right}. One of the tuples t_{left} and t_{right} (or both) contains the longest matching prefix paths of Φ.

We call t_m the tuple (either t_{left} or t_{right}) with the longest prefix path of Φ and *ml* is the **m**atching **l**evel. We use a new compound surrogate cs_{DX} for the computation of cs_{new}. cs_{DX} has the components cs_t, $cs_{t-1}, \ldots, cs_{ml}$. For further processing, we set the remaining cs components $cs_{ml-1}, \ldots, cs_1$ to high, i.e., all bits of the components are set to one. These components are changed in a later step, in order to calculate the final compound surrogate.

Example 8-1 (Matching Level):

For example, we compute the compound surrogate for the hierarchy path $\Phi = t_{new} = (\text{"Segment"}, \text{"Austria"}, \text{"East"}, \text{"Saturn}_E\text{"}, \text{"S5"})$ into our sample hierarchy (see Figure 8-1). A `search(Φ)` in DXh positions between the paths $t_{left} = (\text{"Segment"}, \text{"Austria"}, \text{"East"}, \text{"Hofer}_E\text{"}, \text{"H4"})$ and $t_{right} = (\text{"Segment"}, \text{"Austria"}, \text{"West"}, \text{"Hofer}_W\text{"}, \text{"H8"})$. The matching level is 3, i.e., level *Region* and $t_m = t_{\text{left}}$. Thus, the cs components of cs_{new} are $cs_4 = 1$, $cs_3 = 0$, $cs_2 = 111\ldots1$, $cs_1 = 111\ldots1$. If we assume a maximum fanout of 63 for the levels *MicroMarket* and *Outlet*, cs_{new} = 1.0.63.63. □

[15] Usually we use character data types for hierarchy levels. If other data types are used, the paths are stored in the "natural" order of the data types (e.g., "<" for numbers).
[16] Note that this is implementation specific for Transbase® B*-Trees.
[17] Note that the tuple must not exist in the B*-Tree, because the paths (and therefore tuples in the LDT) are unique.

The surrogate computed so far contains the prefix (matching level). The remaining components are computed in the following way.

8.1.2 Increment CS Component

The B-tree search with the new path as search argument positions to the lexicographically correct position within *DXh*. The surrogates are numbered within the levels according insertion order. The order of compound surrogates does not correspond to the order of the B-Tree attributes. Thus, the position resulting from the `search` operation in the B-Tree generally is not next to the largest compound surrogate with the matching prefix, but can be next to any paths with the matching prefix. The child level of the matching level is the maximum of the cs component in the hierarchy (w.r.t. the prefix path) incremented by one.

As described in Section 8.1.1, the previous step returns a compound surrogate cs_{DX} with the components cs_t, cs_{t-1}, …, cs_{ml} and the bits of the remaining components set to one, i.e., the maximum compound surrogate with the prefix path. cs_{DX} is used as search argument in a B-tree `search` in index *DXcs* to position "behind" the largest compound surrogate cs_l with the same prefix, if no such surrogate exists[18]. The component cs_{ml-1} of cs_l is the maximum component of level *ml-1* w.r.t the prefix path. We increment cs_{ml-1} by one. The remaining cs components cs_{ml-2}, cs_{ml-3}, …, cs_1 are set to zero.

Thus, the compound surrogate cs_{new} for the new path is computed: $cs_{new} = cs_t \circ cs_{t-1} \circ \ldots \circ cs_{ml} \circ cs_{ml-1} \circ 000\ldots0 \circ \ldots \circ 000\ldots0$.

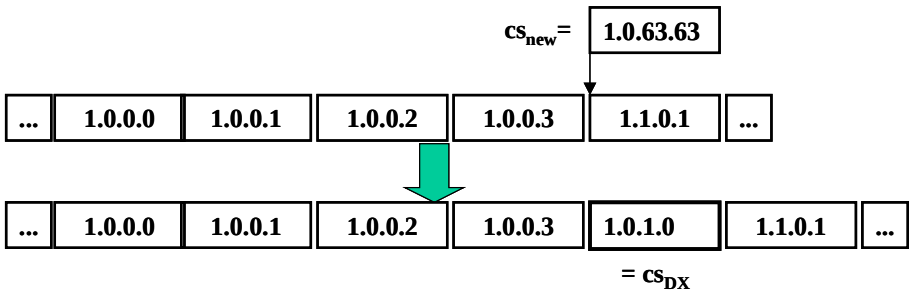


Figure 8-3: Positioning in DXcs

Figure 8-3 shows the positioning of `search(`cs_{DX}`)` with matching level *ml=3* for Example 8-1, cs_{new} = 1.0.63.63. The `search` operation positions behind the largest compound surrogate with the prefix path 1.0. The maximum surrogate for the matching level *MicroMarket* is 0, thus the incremented new cs_{ml-1} is 1, the remaining cs component are set to zero: cs_{DX} = 1.0.1.0. cs_{DX} is the final compound surrogate for the new path Φ = *("Segment", "Austria", "East", "Saturn$_E$", "S5")*.

If all bits of cs_{ml-1} of the previous cs are already 1, then we cannot compute the cs component by incrementing, i.e., an overflow occurs. However, deletion or update of paths may leave "holes" within cs components that could be re-used. Section 8.1.3 describes this *hole search*.

8.1.3 Hole Search

The hole search is necessary, if an overflow of bits within one level has occurred. Due to updates and deletions of hierarchy paths, some bit combinations may be unused and can be reused for the new compound surrogate. The bits of the upper cs components, i.e., the common prefix path, are fixed according to the prefix path of the inserted tuple.

[18] Without loss of generality we assume at the moment that no such surrogate exists. If such a surrogate already exists, we describe further proceeding in Section 18.1.3.

Informally, all bit combinations of the cs component successor level cs_{ml-1} of the matching level from 0..0 to 1..1 are tested to find an unused bit combination. This unused bit combination depends on the existing paths in the hierarchy and is most efficiently found in the secondary index *DXcs*. For every such bit combination, a `search` operation on *DXcs* `search(`cs_{pref} `o` cs_{var} `o 0..0)` is performed, where cs_{pref} is the concatenation of the prefix path components, cs_{var} is the cs component with the new bit combination (level cs_{ml-1}), the remaining cs components cs_{ml-2}, cs_{ml-3}, …, cs_1 are set to zero.

The `search` operation positions on cs_{pref} *o* cs_{var} *o 0..0*, if such a compound surrogate exists in DXcs, otherwise before the lexicographically next compound surrogate cs. If no cs with cs_{pref} *o* cs_{var} *o 0..0* exists, a hole has been found and we choose this surrogate combination.

Otherwise the next tuple cs_{next} in *DXcs* is read (`readnext`), i.e., the lexicographically smallest cs that is larger than cs_{pref} *o* cs_{var} *o 0..0*. If cs_{next} consists of cs_{pref} *o* cs_{var} *o* cs_{rest}, where cs_{rest} can be any bit combination, the bit combination cs_{var} exists and we try the next bit combination for cs_{var}. If cs_{next} does not have a prefix cs_{pref} *o* cs_{var}, then the bit combination of cs_{var} is not used and is the new bit combination for the compound surrogate.

The algorithm is shown in Algorithm 8-1. The variable *max* is decimal value of the highest bit combination that cs_{var} can reach ($2^{|csvar|}$*-1* sets all bits to 1). `search` positions in the corresponding B-tree (here *DXcs*) on the tuple of the search argument (here cs_{pref} *o* cs_{var} *o* cs_{rest}). If the tuple is not found, the B-tree scan is positioned before the smallest tuple with the lexicographical higher order. `readnext` returns this "larger" tuple (a `readprev` would return the tuple in front of the B-tree scan). The predicate `ISPREFIX (cs`^{1}`, cs`^{2}`, k)` tests whether cs^1 and cs^2 have the same cs components until a level *k*, i.e., `ISPREFIX` returns `TRUE`, if cs^1_m=cs^2_m, cs^1_{m-1}=cs^2_{m-1}, …, cs^1_k=cs^2_k.

Algorithm 8-1 (Hole Search):

```
max = 2|csvar|-1
csvar = 0..0
csrest = 0..0
found = FALSE
while not found and csvar < max
     if not TUPFOUND(search(DXcs, (cspref o csvar o csrest)))
          csn = readnext(DXcs)
          if not ISPREFIX(csn, (cspref, csvar, csrest), ml-1)
               found = TRUE
               break
     csvar = csvar + 1
```

For a search on the leaf level, an optimization of the hole search is obvious. Instead of direct search in the B-tree for every bit combination for the cs component one direct positioning with the smallest bit combination and consecutive `readnext` operations are possible. `readnext` is much faster, because usually the next tuple is on the same physical disk page and therefore already in the main memory. If the cs component cs_1 (leaf level) of the tuple returned by `readnext` is larger than cs_1*+1* of the tuple read before, a hole is found. This special case is much faster than consecutive direct positioning (`search`) operations.

8.1.4 Alternative Computation Method

The computation of compound surrogates cs is quite expensive, because for every cs direct search lookups in *DXh* and *DXcs* indexes are necessary that may lead to one or more disk accesses.

For initial bulk loading of a dimension table an alternative method for the compound surrogate computation is possible. Instead of inserting each dimension tuple (hierarchy path), maintaining the secondary indexes and computing the compound surrogate, we sort the tuples according to the lexicographical order of the hierarchy levels, i.e., w.r.t. h_t, h_{t-1}, …, h_1. Now cs can be computed by numbering the tuples w.r.t. the hierarchy path (see Algorithm 8-2). The new tuple is then inserted into the dimension table.

Algorithm 8-2 (Alternative CS Computation):

```
if EOD(sortedTups)
     return
tcs_old = readFirst(sortedTups)
tcs_old.cs = 0…0
writeTup(newTups, tcs_old)
while not EOD(sortedTups)
     tcs_new = readNext(sortedTups)
     ml = MATCHINGLEVEL(tcs_old, tcs_new)
     tcs_new.cs_t = tcs_old.cs_t, tcs_new.cs_t-1 = tcs_old.cs_t-1, …,
     tcs_new.cs_ml = tcs_old.cs_ml,
     if (tcs_old.cs_ml-1+1 < 2^|csml-1|)
          tcs_new.cs_ml-1 = tcs_old.cs_ml-1+1
     else
          ERROR = OVERFLOW
          return
     tcs_new.cs_ml-2 = 0…0, tcs_new.cs_ml-3 = 0…0, … tcs_new.cs_1 = 0…0
     writeTup(tcs_new)
     tcs_old=tcs_new
```

Algorithm 8-2 shows the alternative computation method. *sortedTups* is the tuple stream ordered according to the hierarchy levels h^t, h^{t-1}, …, h^1. *newTups* is the new tuple stream containing the dimension tuples with the computed compound surrogates. The tuples of *newTups* are used for insert into the dimension table and the secondary indexes. $tcs_{\{old,\ new\}}$ are tuples, the attributes are denoted by $tcs_{\{old,\ new\}}.h^t$ for the root hierarchy level or $tcs_{\{old,\ new\}}.cs$ for the compound surrogate of $tcs_{\{old,\ new\}}$. The predicate `MATCHINGLEVEL` returns the level h^l, $1 < l \leq t+1$, of the two tuples (in this case of the old and new tuple). The matching level must be at least two, i.e., the leaf level cannot be a matching level, because h^1 is unique. If the matching level is $t+1$, there is a new value for root level h^t and the bit combination of the cs_t must be incremented. Note that an overflow occurs, if all bits are already set to one for $tcs_{old}.cs_{ml-1}$.

The bit combinations of the first l cs components are assigned to the corresponding cs components of the new cs, the cs component $l-1$, i.e., the following level, is increased by one. The remaining bits are set to zero.

Note that the new computation preprocesses the dimension table tuples by sorting and computation of the compound surrogates. These tuples are then inserted into the empty dimension table via conventional bulk insert. If the dimension table has not been empty before the insert, all existing tuples have to be deleted and reinserted together with the new tuples. Thus this method is only useful for the initial bulk load of the dimension tables.

8.2 Operations on Dimension Table

In a DW, loading and maintaining data is an important issue. Each operation generally affects compound surrogates in an EHC organized DW schema. This section describes maintenance

operations on dimension tables such as inserting, deleting, and updating tuples. Section 8.3 describes these operations on the fact table.

Inserting a dimension tuple is straight forward. For every hierarchy (usually one per dimension) the corresponding surrogate is computed before the insert operation. Because of the "attribute nature" of the surrogates, the original tuple is completed by the computed compound surrogates.

The DML statement `DELETE FROM <dimension>` deletes dimension table tuples. Tuples in the fact table reference the dimension tuples. The tuples of the dimension table that are referenced by other tables cannot be removed, if no `ON DELETE CASCADE` clause on all tables referencing the dimension table is specified ([DD93]). A delete operation on the dimensions is twofold: First the fact table tuples are removed (e.g., via a join to the dimension table with the same predicate as for the deletion on the dimension table), second the dimension table tuples are deleted.

An `UPDATE` operation on the dimension table may cause re-computation of compound surrogates. Affected compound surrogates are propagated to the compound surrogate of the fact table tuples that reference the modified dimension table tuples.

The following sections describe the operations on dimension tables in more detail.

8.2.1 Insert

Basically, two different types of insertions are available in Transbase®. The standard SQL 92 `INSERT INTO <table>` ([DD93]) statement for single tuples (or `INSERT INTO <table> SELECT FROM` for tuples resulting from a SQL statement) inserts tuples via a statement (or from a query). The bulk load `SPOOL` statement inserts a complete spool (flat) file into a table.

In the current implementation of EHC, compound surrogates are treated nearly as conventional attributes. The calculation of compound surrogates is triggered when inserting the hierarchy path. For the compound surrogate, the value NULL must be specified:

```
INSERT INTO dim_segment VALUES ('Austria', 'East', 'Saturn_E', 'S5',
NULL)
```

for the Example 8-1. The NULL is replaced by the computed compound surrogate via a trigger before insert. If the compound surrogates in the `CREATE TABLE` statement are specified behind the last "conventional" attribute, the value for the compound surrogate can be omitted[19]:

```
INSERT INTO dim_segment VALUES ('Austria', 'East', 'Saturn_E', 'S5')
```

For the SPOOL statement, the values in the flat file for the compound surrogates also can be pre-computed, in order to load an archive of the DW into the DBMS. Note that the corresponding reference surrogates of the fact table must have the same pre-computed surrogate values. It must be ensured, that the surrogates are consistent. Pre-computed compound surrogates sometimes can be used to speed up initial bulk loading (see Section 8.1.4).

8.2.2 Delete

As already mentioned, deleting dimension table tuples is only allowed if they are not referenced by tuples in another table, if no `ON DELETE CASCADE` constraint exists. In this case, the tuples of the referencing tables must be removed before deleting the dimension table tuples.

[19] If the `SURROGATE` clause is specified in between the other attributes of the table, it is not possible to assign the values to the contributing attributes correctly and type errors may occur, if no explicit value is specified.

Deletion of tuples means that hierarchy paths are removed. This means that bit combinations of one or more hierarchy levels (depending on the predicate of the delete operation) become vacant (see also Section 8.2.3) and can be reused for new dimension tuples in a *hole search* (see Section 8.1.3). The bit combination for the leaf level component always becomes vacant. Bit combination of other hierarchy levels only get free, if all paths through the member of a hierarchy level are removed, e.g., via the predicate describing a prefix path: `WHERE` $h^t=v_t$ `AND` $h^{t-1}=v_{t-1}$ `AND` … `AND` $h^k=v_k$. Here all tuples of the dimension table containing the prefix path from root h^t to level h^k are deleted, i.e., the bit combination of h^k with the prefix path $(h^t, h^{t-1}, \ldots, h^{k+1})$ becomes vacant.

The propagation of delete operations in a dimension table to the fact table requires efficient searching of the fact tuples that have to be deleted (if the delete operation is cascaded). In order to find these tuples, the reference surrogates of the deleted tuples in the dimension table are used to identify the fact tuples, because they are usually index attributes of the UB-Tree (see also Section 8.2.3).

An alternative for the hole search is to store all deleted surrogates in a maintenance table and use these information for the calculation of the compound surrogates. The same holds for the update operations.

8.2.3 Update

An `update` operation of a tuple semantically is equal to a `delete` followed by an `insert` operation of the modified tuple. The insert operation triggers the computation of the compound surrogates for the dimension table. The modification of the compound surrogate results in an additional update operation on the fact table, because the reference surrogates of the fact table reference the compound surrogates of the dimension tables. Usually a large number of fact tuples reference one tuple in the dimension table. Thus, a new computation of the compound surrogate requires an update of many fact tuples.

However, not every update operation on the dimension table leads to a re-computation of the compound surrogate cs (e.g., if feature attributes are modified or a hierarchy member is renamed) and the original cs can be reused. We try to avoid re-computation as far as possible. This section describes the conditions when surrogate computation can be omitted and gives implementation hints for efficient update processing.

Basically there are two ways to recognize whether a new computation is necessary or not. The first method uses structural information, i.e., the query predicates and catalog information. The second method is dynamic and uses the result of the predicate to decide whether to compute a new compound surrogate or not. In this thesis, we concentrate on static rules (see also Section 9).

8.2.3.1 Update Template

An `UPDATE` statement contains a list of attributes to change and a search condition to specify the tuples that are changed. The syntax for the SQL update statement is the following:

```
UPDATE TableName
     SET AssignList
     [WHERE LOCPRED]

AssignList ::= Assignment [, Assignment]

Assignment ::= Fieldname = Expression
```

LOCPRED is a predicate on the table to update, i.e., a number of restrictions that can contain sub-queries. *Expression* specifies the new value of the attribute *Fieldname*, where this expression can be formula that may be computed by a sub-query.

This general update template is simplified w.r. to an update in the DW scenario. We talk about a dimension table update for the following cases of *LOCPRED*. The attributes involved with *LOCPRED* are either feature attributes or hierarchical attributes. The restrictions of *LOCPRED* are equality restrictions, i.e., $attr_k$ = *Expression* for all attributes, where the restrictions are connected via AND operators:

UPDATE … WHERE $attr_i$ = Expression AND $attr_j$ = Expression AND …

Expression can be any formula returning one single value (e.g., a constant value, arithmetic formula or sub-query returning one value).

8.2.3.2 Update Classes

Depending on the type of update compound surrogate computation can be omitted or optimized. For a formal discussion we define four update classes:

- (U1): *Feature Update*
- (U2): *Renaming Hierarchy Members*
- (U3): *Moving a Hierarchy Sub-Tree*
- (U4): *Other Updates*

Definition 8-2 (Feature Update):

A *feature update* is an update of the dimension table on a non-hierarchical attribute, i.e., an attribute that does not occur in a hierarchy path of the hierarchies of one dimension. □

An update is a feature update, if *AssignList* only contains feature attributes:

UPDATE D_j SET f=f_{val} WHERE h_k = val

Definition 8-3 (Renaming Hierarchy Members):

Renaming hierarchy members is an update operation on a dimension table that does not change the structure of the hierarchy. Only the value of one or more hierarchy members is changed. An update on leaf level members (if no other hierarchical attributes are changed) always is a renaming operation. □

UPDATE D_j SET h_k = val_{new} WHERE h_t = val_t AND h_{t-1} = val_{t-1} AND … AND h_k = val_{old}

An update on leaf level means that the path from the top level to the father node of the leaf remains unchanged, the cs components remain. More generally, if the cs component for the leaf level is still valid, no new computation of cs is necessary. Thus, an update on leaf level always is a renaming of a hierarchy member.

Definition 8-4 (Moving Hierarchy Sub-trees):

If all paths of a hierarchy sub-tree are changed consistently, i.e., the prefix paths of the father member of the sub-tree are the same, we call this update operation *moving hierarchy sub-trees*. □

UPDATE D_j SET h_t = val_t^{new}, h_{t-1} = val_{t-1}^{new}, …, h_k = val_k^{new}
WHERE h_t = val_t AND h_{t-1} = val_{t-1} AND … AND h_k = val_k AND h_{k-1} = val_{k-1}

In this example, we specify the hierarchy path from the top level h^t to level h^k and change the path for the complete sub-tree. An example for the geographic hierarchy of the sample DW is shown in the following:

UPDATE segment SET country = ‘Germany’, region = ‘south’ WHERE
country = ‘Germany’ AND region = ‘north’ AND micromarket = ‘$Aldi_N$’

This update class is needed for structural reorganization of dimensions, e.g., if customers are re-classified, product groups are changed etc. In Figure 8-6 we show the reclassification of the *Region* level of the *Segment* dimension and the effect for the compound surrogates.

Definition 8-5 (Other Updates):

If an update operation is none of the classes above, we call it *other updates*. □

8.2.3.3 Conditions for Update Classes

The update class depends on the predicate and the assign list. Table 8-1 contains the conditions for the update classes. In this table, we consider dimension tables with the attributes h_t, h_{t-1}, …, h_1 for the hierarchy levels and f_1, f_2, …, f_k for the feature attributes. The assign list is denoted by assignments a_i = *expr* on attribute a_i. *LOCPRED* is described as a set of equality restrictions on the dimension table *{R(a_i)}*, i.e., a_i=*expr* for attribute a_i and expression *expr*.

Class	LOCPRED	AssignList
Feature Update	any local predicate on D_i	$\{ f_j = expr_j \}, 1 \le j \le k$
Renaming Hierarchy Members	$\{ R(h_p) \}, R(h_j)$, where $1 \le j < p \le t$	$h_j = expr_j$
Moving Sub-Trees	$\{R(h_t), R(h_{t-1}), \ldots, R(h_k)\}$,where $1 \le k \le t$	$h_j=expr_j$, $h_{j-1}=expr_{j-1}$, …, $h_k=expr_k$, where $k \le j \le t$
Other Updates	Any local predicate on D_i	any assign list

Table 8-1: Conditions for Update Classes

A *feature update* has only feature attributes in the assign list, but all attributes of the dimension table can be involved in *LOCPRED*. No hierarchical attributes are modified and the hierarchy and the cs components remain unchanged. There is no effect on tuples of the fact table, so this operation is executed very efficiently.

When *renaming hierarchy members* the hierarchical attributes involved in *LOCPRED* specify a prefix path (or a number of prefix paths), i.e., *{ R(h_p) }* of Table 8-1. A hierarchy member m_j^k of hierarchy level h_j is specified additionally in *LOCPRED*, where $j < p$, i.e., *R(h_j)* of Table 8-1. This hierarchy member is modified as specified in the *AssignList*: $h_j = expr_j$. Since each hierarchy member is assigned a unique surrogate value within its siblings, the modification with the conditions described does not change the hierarchy structure and the compound surrogates remain valid. An update on a leaf member ($j = 1$ in Table 8-1) leaves the compound surrogate valid, too, because the path from the top level to the leaf level is the prefix path and the cs components remain the same. Note that renaming more than one hierarchy member can be done in a sequence of these update operations. In Figure 8-5, we show the renaming of the hierarchy member 'Saturn$_W$' to 'Mediamarkt$_W$'. The SQL statement for this operation is the following:

```
UPDATE segment SET micromarket = 'Mediamarkt_W' WHERE Country =
'Germany' AND region = 'West' AND micromarket = 'Saturn_W'
```

If a sub-tree of a hierarchy is moved to another position in the hierarchy, i.e., the prefix path from the top level of the hierarchy to the top level of the sub-tree is changed, we have the update class *moving sub-trees*. *LOCPRED* specifies the complete prefix path from the hierarchy top level h_t to a hierarchy member of the top level of the sub-tree m_k^j. Moving means, that the prefix path changes, i.e., the outgoing edge of m_k^j is changed. Such a reorganization requires a corresponding modification of the compound surrogates (in the dimension table and in the fact table). Since the compound surrogates of a hierarchy sub-tree form an interval, the lookup of the fact table tuples can be optimized. Note that a

key attribute update (as it is necessary for the reference surrogate of the fact table) in a clustering index requires a delete and an insert operation, we first delete the tuples that are modified, write them to a tuple stream, modify the compound surrogates accordingly, sort them and insert them into the fact table via bulk update.

In Figure 8-6, we show *moving of sub-trees* of the member 'West' to the member 'East'. The SQL statement for this operation is the following:

```
UPDATE segment SET region = 'East' WHERE Country = 'Germany' AND
region = 'West'
```

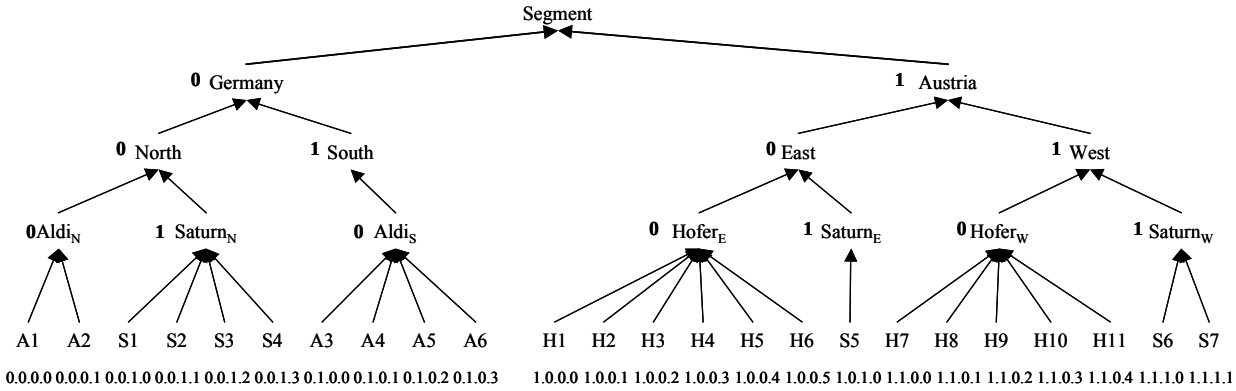


Figure 8-4: Sample Hierarchy

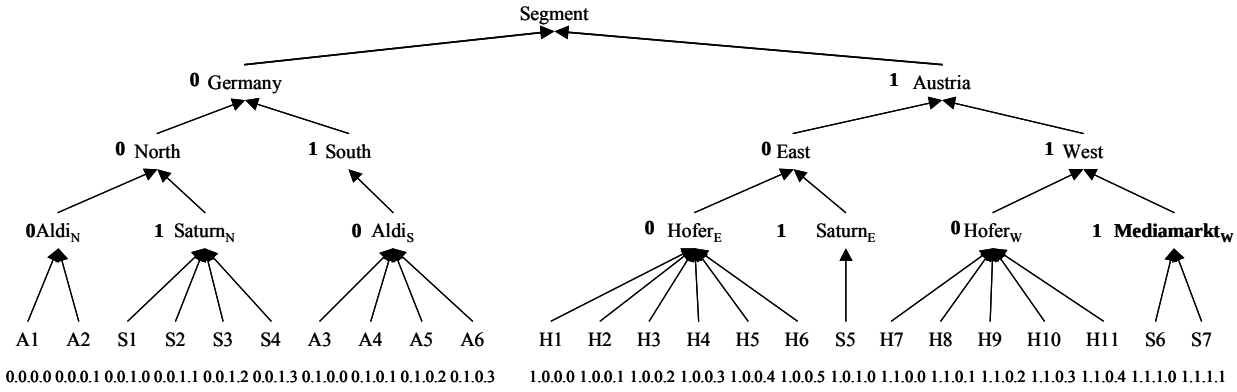


Figure 8-5: Renaming Hierarchy Members: Saturn$_W$ ➔ Mediamarkt$_W$

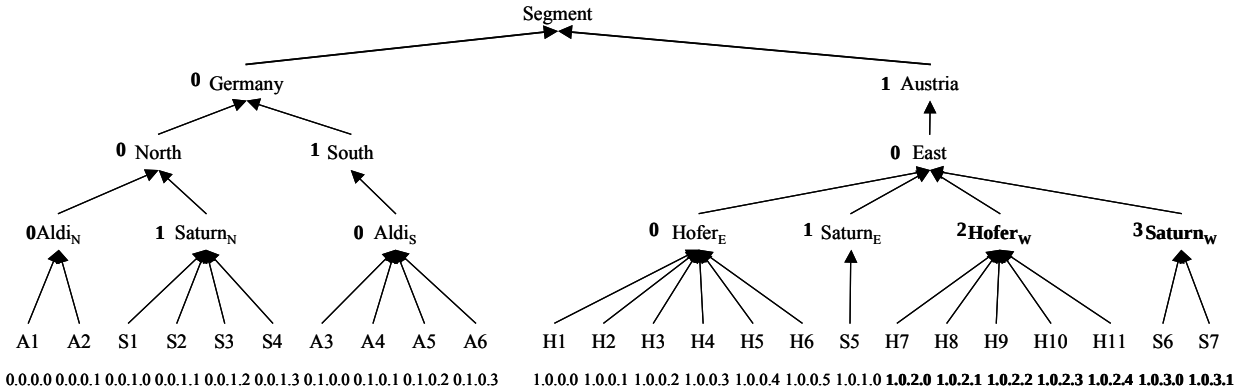


Figure 8-6: Moving Sub-Tree: West ➔ East

8.2.3.4 Processing of Update Statements

The update classes (U1) and (U2) do not require special processing algorithms because no compound surrogates are changed and the update is performed efficiently. An update of the leaf level (i.e., the dimension key attribute) will require a corresponding modification on the dimension key attribute of the fact table (not a primary index attribute). In this case for every dimension tuple that is updated, the corresponding fact tuples are searched via the reference surrogate. For the classes (U3) and (U4), we need special methods to execute the dimension update and the following fact table update as efficiently as possible.

For update class *moving sub-trees* (U3) the new compound surrogates have the same cs prefix (see also Figure 8-6). The corresponding dimension table tuples can be specified via an interval on the compound surrogate, because a hierarchy sub-tree is encoded by a compound surrogate interval (see Section 5.3). Thus, the affected fact tuples are specified via an interval on the reference surrogate on this interval, i.e., a range query on one dimension. We delete these fact tuples and collect them in a temporary container (e.g., main memory, if sufficient, or a file), modify the cs prefixes for the new prefix path, sort them w.r.t. the z-value of the dimensions and insert them via bulk insert into the fact table.

Figure 8-6 shows an example for this update class (U3). The paths containing hierarchy member *'West'* now contain hierarchy member *'East'* instead. The cs components of the hierarchy paths for hierarchy level Region are changed accordingly (from 1 to 0). Note that the successor members also have to be modified (*"$Hofer_W$" = 2* and *"$Saturn_W$" = 3*).

For the update class *other updates* (U4) we propose a general method how to process such update statements. Any dimension tuple may be affected by the update predicate *LOCPRED* with a modification of the compound surrogates. The changed compound surrogate requires an update on the reference surrogate of the referencing fact tuples. Such an update on the fact table is a primary index attribute update, i.e., the tuples physically are deleted from the UB-Tree and inserted with the new index attribute value. We delete the fact tuples for every changed dimension tuple, collect them like for (U3), assign the new reference surrogate, sort them and insert them into the fact table. Depending on the number of tuples in the fact table concerned by the update statement, this operation can take a long time.

8.2.3.5 Alternative Update Processing

A naïve method to cascade updates from the dimension table to the fact table is to perform for every tuple t_D of the dimension table that is updated, a lookup in the fact table. The tuples of the fact table found by the lookup are updated correspondingly. The lookup is a multidimensional range query, where one dimension is restricted to a point and the other dimensions are unspecified. This leads to a very poor performance, because such a degenerated query box intersects a large number of pages and thus takes a long time to perform. A large number of dimension table tuples t_D leads to a corresponding iterative processing (nested loop) of the update sequence. This processing serves the query boxes in a pipeline manner. Often, the same pages are accessed by more than one query box.

We propose to use an optimized execution plan, where first all query boxes are collected, in order to apply an extended multi-query box algorithm as described in Section 10.3. With this algorithm we first collect the page identifiers of the fact table before actually reading them from disk. For each page, we post-filter the tuples by all query boxes intersecting this page. Thus, we avoid multiple reading of pages.

8.3 Operations on the Fact Table

The amount of data stored in the fact table usually is very large compared to the dimension tables. Maintenance of the fact table (i.e., insert, delete, and update) often affects a large number of tuples and can take a long time to perform.

The initial load of the fact table inserts a large number of tuples (usually tens of millions or more). The physical organization of the fact table with the clustering UB-Tree and the reference surrogates as index attributes requires a lookup in each dimension table for every tuple. We additionally must ensure the consistency of the dimension keys in the fact table and the dimension tables (reference constraints). Special methods are necessary to speed up inserting data into the fact table.

Operations on the fact table often use dimension keys as restrictions to identify the tuples that are maintained (the concatenation of the dimension keys is the primary key of the fact table). The dimension keys, however, are not index attributes of the UB-Tree. A full table scan is necessary to perform such operations. Section 8.3.1 describes how to improve the performance via a transformation of dimension key restrictions to reference surrogate restrictions.

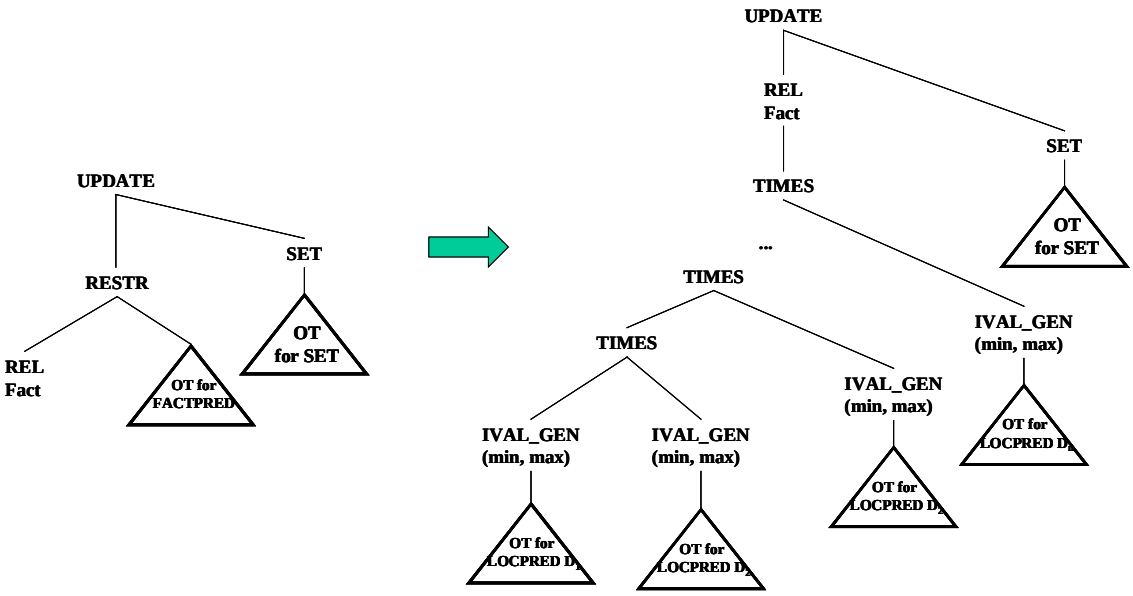
8.3.1 Transforming Restrictions

In the fact table, the dimension keys are not indexed, i.e., the dimension keys are not index attributes of the UB-Tree and usually are not indexed via secondary indexes. Thus, restrictions on dimension keys are evaluated via a full table scan and are performed slowly. The physical model, however, assigns each dimension key a reference surrogate that is an index attribute of the UB-Tree. Transforming the restriction on the dimension keys into a restriction on the reference surrogates efficiently evaluates the result via index access.

In the following, the predicate on the fact table is denoted by `LOCPRED(fact)` consisting of a sequence of `PRED(fact.d`$_i$`)` (dimension keys) and of `PRED(fact.m`$_k$`)` (measure attributes) connected via `AND` operators. The predicates `PRED(a)` can be any SQL predicate determining attribute a, e.g., `a = expr` or a `BETWEEN expr`$_1$ `AND expr`$_2$ etc.

The transformation maps a predicate `PRED(fact.d`$_i$`)` of the dimension key to a predicate `(fact.cs`$_i$`)` of the corresponding (indexed) reference surrogate. This mapping is provided by an artificial join to the corresponding dimension table, i.e., a lookup into the dimension table with the statement `SELECT cs FROM D`$_i$ `WHERE PRED(D`$_i$`.h`$_1$`)`. `PRED(D`$_i$`.h`$_1$`)` means that in the original predicate `PRED(fact.d`$_i$`)` we replace each occurrence of *fact.*d_i with $D_i.h_1$, i.e., the leaf level of the hierarchy of the dimension table D_i. We get a set of compound surrogates (i.e., the corresponding reference surrogates for d_i) that is used as predicate on the reference surrogate.

The resulting more complex operator tree is optimized in the conventional way as described in Sections 9 and 10. Especially, the combination of several dimensions leads to multidimensional point queries, because the set of reference surrogates for each dimension contains single values. An optimization to recognize intervals on the set of single values can speed up processing significantly. We use the *cs2ival* operator described in Section 10.1.8.1 that returns a number of intervals from a set of compound surrogate values using the *DXcs* index.



(a) Original Operator Tree **(b) Transformed Operator Tree**

Figure 8-7: Operator Tree for Update on Fact Table

Example 8-2: (Transforming a Restriction):

Figure 8-7 shows the operator tree for the transformed restrictions in the case of an UPDATE statement with a SET clause and a predicate (see Section 8.3.5). The original operator tree (not optimized for EHC) is shown in Figure 8-7 (a). The restriction evaluated by the RESTR operator usually performs a full table scan on the fact table *(*REL(Fact)*)*, because no index access is possible on the UB-Tree. The SET operator modifies the corresponding attributes of the fact table result tuples as specified in the SET clause of the UPDATE statement. The transformed operator tree (Figure 8-7 (b)) consists of three parts, the restrictions shifted to the dimension tables, the range query access to the fact table and the SET operator for the UPDATE SET clause.

The IVAL_GEN operator is an abstract operator for the interval generation of the local restriction of h_1 of dimension D_i. Depending on the restriction, IVAL_GEN returns a set I of cs intervals, $I = \{(min_{cs}, max_{cs})\}$. Each element of I is joined with the cs intervals of the other restricted dimensions for a number n of multidimensional range queries, $n = n_{I1} * n_{I2} *\ldots * n_{Ik}$ for k restricted dimensions. In addition to the multidimensional range query, local predicates on feature attributes can further restrict the fact table. These restrictions are summarized in the tree LOCPRED for the fact table. For each resulting tuple the SET predicate is applied to update the corresponding attributes. Note that by the optimization, a special algorithm to handle multiple query boxes is applied (see Section 10.3).

8.3.2 Loading Data

The most performance critical maintenance operations in data warehouse systems is the periodic incremental load of the DW. The DW is updated in periodic intervals (e.g., once a day, a week or month) during a time window where no traffic is on the DBMS (dedicated load). Sometimes there are no time windows because the warehouse must be available all the time (24 x 7 availability). In this case, updates are performed during normal work load and may slow down retrieval performance.

This section describes how to speed up bulk loading in order to minimize the time window and reduce the time to stress the DBMS by this operation.

Inserting a tuple into the fact table requires the lookup of the reference surrogates (see Section 8.3.3). A lookup for every reference surrogate (dimension) into the corresponding dimension table via the

dimension key (i.e., `SELECT` D_i`.cs FROM FACT,` D_i `WHERE FACT.`d_i=D_i.h_1) returns the corresponding surrogate. The lookup is necessary for each dimension and therefore slows down the performance to process the load (one `INSERT` operation requires *n* index access operations, if a fact table has *n* reference surrogates).

The lookup is processed via the primary index of the dimension table. An index lookup has the effort of a B-Tree search. Sometimes the required pages of the dimension tables can be cached in main memory (if the insert is local with respect to the dimensions), but a B-Tree organization is not the most efficient main memory access structure. Often the primary indexes of all fact tables do not fit into the main memory caches of the DBMS.

One solution is to store the necessary attributes, i.e., the dimension key and compound surrogate in a hash table. Each dimension is associated a hash table that is stored in main memory. A lookup in a hash table is very performant (depending on the length of the collision chains). There is a constant overhead to read all dimension tables for the bulk load and store the two attributes – the dimension key and the cs values – in the hash tables. The consecutive lookups are very fast.

It is important to chose a reasonable cardinality of the hash values depending on the number of objects to insert into the hash table. A good tradeoff between chain length and space utilization is to use a cardinality of *n/3* if *n* objects are inserted into the hash table. In this case we have an average chain length of three with an average of three main memory accesses for one lookup (one for the address of the first object, another one to check whether it is the searched object and in the average one for the address of the next object in the chain).

Compared to the insert into the fact table this lookup takes a short time. However, the amortization of the constant effort to load the hash tables depends on the number of tuples that are inserted via the bulk load. Thus, the bulk optimization should be applied, when the sum of the time to build the hash tables and for the bulk insert is less than inserting all tuples with single inserts. In Section 8.4 we show the insert behavior for both methods.

The reference constraints between the dimension attributes of the fact table and the dimension keys of the dimension tables require a consistency check. The consistency check usually is performed in a separate lookup in the referenced table. In the case of a reference surrogate lookup, however, this check is done implicitly. If a reference surrogate for a dimension key has been found, a corresponding dimension key in the dimension table also is found. The reference constraint is fulfilled. Thus, we can omit the reference checks (for the reference constraints for the dimension keys) when inserting fact tuples.

8.3.3 Insert

Inserting a tuple into the fact table requires a lookup into all dimension tables to get the reference surrogates corresponding to the dimension key attributes. Each tuple is completed with the reference surrogates for all dimensions. The complete tuple is inserted into the fact table, i.e., into the UB-Tree.

The lookup is done via primary index on the dimension table D_i that has $D_i.h_1$ as index key with the condition $FACT.d_i = D_i.h_1$. Thus the cost to insert the tuple into the fact table is $n*Iacc+UBIns$, where *n* is the number of dimensions for the fact table and *Iacc* is the cost for a direct index access (positioning). *UBIns* is the cost to insert a tuple into the UB-Tree.

Inserts concern a small number of tuples (compared to bulk insert, periodic updates and initial load). The overhead to fill hash tables in order to speed up the surrogate lookup is too large. A hybrid method is possible to get efficient lookup in the hash table and avoid the overhead to fill the hash table. A hash table is created for each dimension without filling it. For every tuple to insert into the fact table we search in the hash tables whether the reference surrogates are already stored. If the dimension key is not found, we perform a lookup in the dimension table and store the new dimension key (and reference surrogate) in the hash table. Because of the short lookup time for hash lookup, the

performance in this case is nearly the same as for index lookup solely. This method makes also sense, if a relatively small number of tuples is inserted via bulk insert (compared to the cardinality of the dimension tables).

8.3.4 Delete

Deletion of tuples is a standard B-Tree (or UB-Tree) operation. To find the tuples that are deleted a predicate is specified:

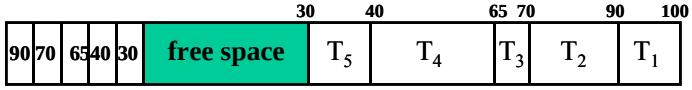
```
DELETE FROM table WHERE LOCPRED
```

LOCPRED can be any predicate specifying the tuples to delete. If the restrictions of LOCPRED concern dimension keys, the restriction can be mapped to a restriction on the reference surrogates via a restriction transformation (see Section 8.3.1) to get the tuples to delete efficiently via UB-Tree access.

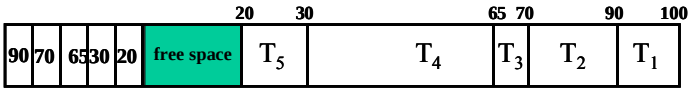
8.3.5 Update

An `UPDATE` statement specifies the tuples to modify in the same way as a `DELETE` statement (see Section 8.3.4) via the predicate *LOCPRED*. To efficiently find the corresponding tuples, the same predicate transformation (dimension key restrictions are transformed to restrictions on the reference surrogates via a lookup in the dimension tables) is used (see Section 8.3.1).

The modified tuples are not deleted and inserted into the UB-Tree as long as the primary index key attributes (for the UB-Tree) are not changed, i.e., an *update in place*. An update in place computes for each tuple the new attribute values. Some minor reorganization occurs, if the size of the tuple changes by the update. In this case, the tuples on the same page that are "larger" (according to the index key order) are shifted to provide enough space for the new tuple or to reuse the free space from the update (see Figure 8-8). This shift can result in a page overflow with a consecutive B-Tree page split[20]. Figure 8-8 shows the effect of an update in place on tuple T_4 where the size of T_4 increases by 10 byte. The illustration depicts one page with the tuples T_1, T_2, T_3, T_4, T_5 with the order $O(T_1) < O(T_2) < O(T_3) < O(T_4) < O(T_5)$. The position array on the beginning (left side) of the disk page points to the position of the corresponding tuples, i.e., the smallest tuple T_1 begins on position 90 etc. An update on T_4 increases the size of T_4 and shifts the larger tuples to the left. If the size is reduced, T_5 is shifted to the right.



(a) before UPDATE Tuple T_4



(b) after UPDATE Tuple T_4

Figure 8-8: Reorganization for Update in Place

If the primary key is changed by the `UPDATE` statement, i.e., an update of one of the reference surrogates, the tuple is deleted and inserted with the new value of the attributes. A change on the reference surrogates occurs, if the dimension key of a fact tuple is changed. A lookup in the corresponding dimension table fetches the new reference surrogate values.

[20] The consequences depend on the implementation of the B-Tree operations. E.g., a page underflow could occur and pages are merged etc.

The most frequent updates on primary key attributes (reference surrogates) are triggered updates. Triggered updates occur, if the compound surrogate of a dimension table is modified, and this modification is propagated to the tuples in the fact table (see Section 8.2.3).

8.4 Measurements

This section describes some measurements with the implemented concepts. We describe maintenance performance for dimension and fact tables. The measurements on the Sales DW are performed on a two processor PC Pentium II, 400 MHz, with 768 MB RAM and a SCSI hard disk. Operating system is Windows 2000.

8.4.1 Dimension Table Maintenance

An insert into a dimension table triggers the computation of the compound surrogates of the table (for details see Section 8.1). In this section we compare mass loading of the dimension table customer of the Sales DW with and without computation of the compound surrogate. There is one hierarchy defined on the customer dimension: *Country – Department – County – City – Area – Customer*. Thus, the compound surrogate consists of six components, one for each hierarchy level. The calculation requires several index accesses. We loaded 10, 100, 1.000, 10.000, 100.000, and 1.000.000 tuples via flat file into the empty customer dimension table. For each tuple the kernel first computes the compound surrogate and inserts it into the B*-Tree immediately with maintenance of the two system indexes *DXh* and *DXcs*. The index maintenance is necessary, because actual indexes are required for the computation of the compound surrogates.

Note that single insert statements would have about the same performance, because they compute the compound surrogates in the same way and require corresponding index maintenance.

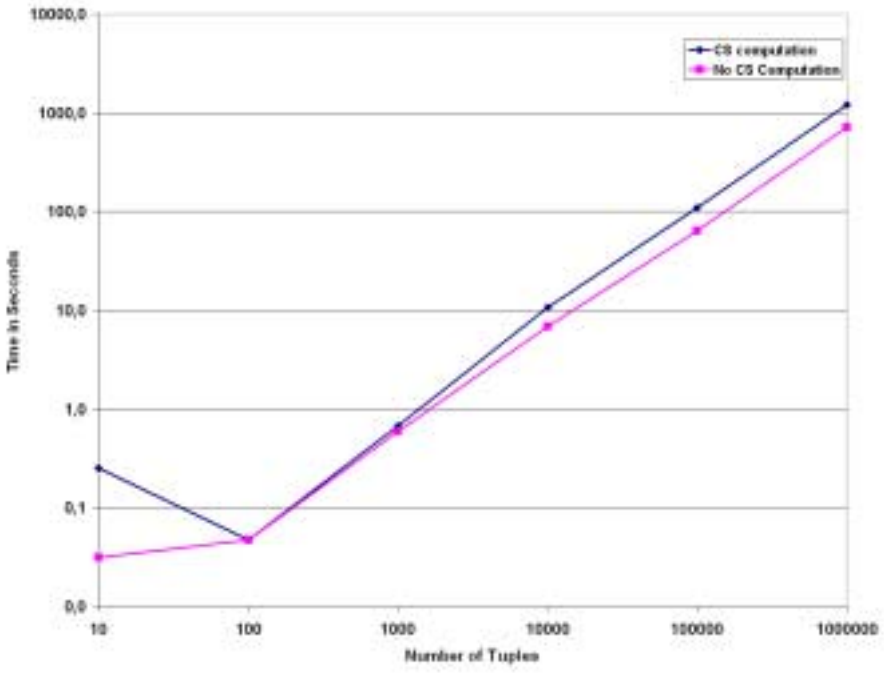


Figure 8-9: Load Performance of Dimension Tables

Figure 8-9 shows the comparison of inserting tuples with and without computation of compound surrogates. Note that the axes are both logarithmic. Generally speaking, the insert complexity is linear with the number of inserted tuples. The computation overhead is between 10% and 70% for more than 1.000 inserted tuples. For smaller amounts of tuples, there are further effects that make interpretation difficult (e.g., cache effects).

In the overall, the insert rate is about 1.000 tuples per second for large number of tuples. This usually is enough for the daily update of changing dimensions.

8.4.2 Fact Table Maintenance

We measure the load performance of the fact table for the Sales DW with a modified fact table: The number of measure attributes is reduced to 10. Five dimensions are used for MHC: *product, warehouse, calendar, transaction* and *sales payment*. Each dimension has a compound surrogate in the dimension table and one corresponding reference surrogate in the fact table. Table 8-2 shows the dimensions and the cardinalities. Note that all dimensions are relatively small. The product dimension is the largest dimension with 27.929 tuples.

Dimension	Cardinality
Product	27929
Warehouse	226
Calendar	2922
Transaction	91
Sales Payment	44

Table 8-2: Dimensions and Cardinalities for the Measurements

In Figure 8-10 we show the fact table load time for different load mechanisms. We compare the load of 10, 100, 1.000, 10.000, 100.000 and 1.000.000 tuples with the four different methods:

- *Random Insert*: Tuples are inserted via single `INSERT INTO` SQL statements. Each reference surrogate requires a lookup in the corresponding dimension table (index access).
- *Spool Hash Lookup*: Tuples are bulk loaded (via flat file) and the hash lookup optimization method is used. Hash lookup means that all compound surrogates and the dimension key values of all dimensions are loaded into internal hash tables before the load operation. The reference surrogate lookup is done in-memory.
- *Spool with CS*: The reference surrogates are already in the flat file. No reference surrogate lookup is necessary.
- *Spool B-Tree*: The flat file is spooled into a standard B*-Tree.

Note that the scale is logarithmic for both axes.

The random inserts are almost linear w.r.t. the number of tuples inserted. We did not measure the random insert for 1 million tuples, since the time is about 12 hours. For the bulk load operations, the time is also almost linear for larger tuple sets (from 1 thousand to 1 million tuples). For both bulk load methods, a constant overhead for loading the hash tables with the compound surrogates is necessary. In these measurements, this constant time is very short (about 0,5 seconds), because the dimensions are not very large. For the hash table load, the complete *DXh* index is read. Since this is a full index scan, it is very fast: 0,5 seconds for all dimensions.

After reading the tuples from the flat file, they are supplemented (by the reference surrogate) and sorted. Note that the sorter cache was large enough to hold them in cache (no external sorting has been applied). We use heap sort as in-memory sort method. For 100 tuples, the time is less than for 10 tuples, because the index pages are already in cache.

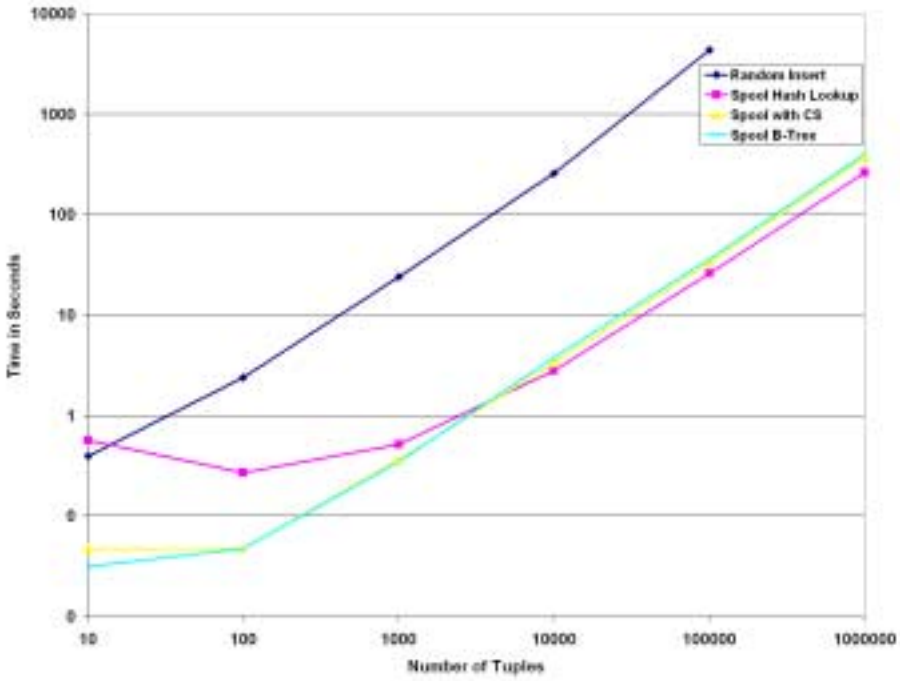


Figure 8-10: Comparison of Fact Table Load Performance

The time for *Spool with CS* is more than for the hash lookup method. This method is implemented by loading a UB-Tree with the dimension keys as the reference surrogates, but without defining reference surrogates. In this case, the tuples from the flat file are inserted with a computation of the z-value. However, the time for the correctness check of the dimension attribute values (whether they are in the proper range) takes a while, since it is implemented as a constraint operator tree and each tuple runs through this operator tree. This means that the hash lookup is very efficient.

We do not need these range tests for standard B*-Trees. However, the load time is still not faster than for the hash lookup. Some additional effects affect the performance. As for the other methods, sorting is necessary. For B*-Trees we sort w.r.t. the compound key (in this case the reference surrogates), not the z-value. Depending on the sort performance, there may be some deviation from the measured results.

In general, the mass loading method is more than a factor of 100 faster than random insert (especially for large number of tuples). With the implemented mass loading method, we load more then 3.800 tuples per second, i.e., 1 million tuples are loaded within 259 seconds.

9 Query Processing

The users of a data warehouse gain information about the stored data via queries on the DBMS. We call SQL statements on a star (or snowflake) schema which include the fact table *star queries*. The processing of star queries considering MHC is described in this section.

One of the most important parts of a star query is the processing of the star join, i.e., the join of the large fact table with the dimension tables. Star join processing has been studied extensively and specific solutions have been implemented in commercial products. See [CD97b] for an overview.

Bitmap indexes are used frequently to speed up the access to the fact table. The bitmaps corresponding to the different dimension values are ANDed or ORed depending on the selection condition. The resulting bitmap is used to extract tuples from the fact table ([NG95], [NQ97]). When the query selectivity is high, only a few bits in the result bitmap are set. If there is no particular order among the fact table tuples, we can expect each bit to access a tuple in a different page. Thus there will be as many I/Os as there are bits set.

Multidimensional clustering has been discussed in the field of multidimensional access methods (e.g., [GG97] and [Sam90]). [ZSL98] addresses the issue of hierarchical clustering for the one-dimensional case. The importance of good physical clustering in OLAP has been shown in [KR98], where packed R-trees are exploited for storing the results of the data cube operator ([GBLP96]). In [DRSN98], the benefits of hierarchical clustering for star queries was observed as a side effect of using a chunked file organization for enabling caching with chunk as the caching unit.

Several aspects of processing and optimizing star join queries on hierarchically clustered fact tables are also presented in [TT01]. The paper considers a star schema with UB-Tree organized fact tables and dimension tables stored sorted w.r.t. a composite surrogate key. For a particular class of star join queries, the authors investigate the usage of sort-merge joins and a set of other heuristic optimizations.

Some further optimization w.r.t. grouping with hierarchically clustered data is the pre-grouping method as described in Section 9.4. The publications contributing most to the pre-grouping methods described in this thesis are [CS94] and [YL94]. [CS94] describes three principles of pre-grouping, i.e., *invariant grouping, simple coalescing* and *generalized coalescing*. [YL94] and [YL95] describe an *early grouping and aggregation* method very similar to the coalescing methods of [CS94]. However, these methods do not consider hierarchical pre-grouping.

In the last years two additional publications, [Lar97] and [Lar02], discuss pre-grouping. [Lar97] compares different grouping methods and introduces a mathematical model to estimate group sizes and [Lar02] extends pre-grouping by practical implementation issues such as partial pre-grouping.

[SN95] describes parallel query processing, where grouping is done on partial results (per node). The final grouping operation merges the groups of the local results.

Other transformations are described in [GHQ95] and [LMS94]. We first published the basic abstract execution plan as described in Section 9.3 in [KTS+02]. We present the basic abstract execution plan for star queries on a schema with hierarchical encoding and a primary clustering multidimensional index on the fact table. Also the concept of pre-grouping is introduced shortly. In this thesis we extend the discussion of pre-grouping by implementation issues and an exact performance evaluation.

9.1 Conventional Approach of Star Query Processing

The standard query processing algorithm to execute so called star join queries first evaluates the predicates on the dimension tables (either on normalized (snowflake) or de-normalized (star) schemata) resulting in a set R_i of m_i tuples of dimension D_i, and then builds a cartesian product of the dimension result tuples $(R_1 \times R_2 \times \ldots \times R_n)$ for dimensions $D_1, D_2, \ldots, D_n$. The number of tuples $t=(f_1, f_2, \ldots, f_n)$, where $f_1 \in R_1, f_2 \in R_2, \ldots, f_n \in R_n$, is $m_1 * m_2 * \ldots * m_n$. The tuples t are used for a direct index access on the compound index on the fact table. For non-sparse fact tables and queries that restrict most dimensions of the compound index in the order of the index attributes the access to the fact tuples is quite fast. The next processing step joins the resulting fact tuples with the dimension tables for grouping and aggregating.

However, for sparse fact tables and high dimensionality, such a query processing does not work efficiently enough for large data volumes. The number of cartesian tuples resulting from the dimension predicates grows very fast, whereas the number of affected tuples in the fact table may be comparably small.

Due to the usually relatively small cardinality of the dimension keys compared to the number of fact tuples, some DBMS index the dimension keys in the fact table with bitmap indexes ([CY98], [WB98]). With the use of bitmap indexes, a different star join processing is possible (called star transformation in Oracle, [Ora01]). The star transformation first transforms the restrictions on the dimension tables into local sub-queries, each containing the dimension restriction, that qualifies a number of dimension keys. The dimension keys serve as restrictions on the bitmap indexes. The bitmap intersection returns a number of tuple identifiers for the resulting tuples of the fact table. The materialization of the resulting fact tuples, eventually reduced by additional local restrictions on the fact table, retrieves the tuples in base granularity. These tuples now are joined with the dimension tables, in order to group and aggregate them and get the resulting tuples of the query.

For example, consider the following query:

```
SELECT SUM(F.sales), P.group, L.area, T.month
FROM Fact F, Product P, Location L, Time T
WHERE F.item_id = P.item AND F.store_id = L.store AND
      F.day_id = T.day AND P.category = 'cat1' AND
      L.population > 1000000 AND T.year = 2002
GROUP BY P.group, L.area, T.month
```

This query is rewritten into the following statement:

```
SELECT SUM(F.sales), P.group, L.area, T.month
FROM Fact F, Product P, Location L, Time t
WHERE F.item_id = P.item AND F.store_id = L.store AND
  F.day_id = T.day AND
  F.item_id IN
      (SELECT item FROM Product WHERE category = 'cat1') AND
  F.store_id IN
      (SELECT store FROM Location WHERE population > 1000000) AND
  F.day_id IN (SELECT day FROM Time WHERE year = 2002)
GROUP BY P.group, L.area, T.month
```

With this query rewriting, the secondary indexes (usually bitmap indexes in Oracle) on the dimension keys of the fact table can be evaluated and intersected. The index intersection returns a set of tuple identifiers *TID* that qualify tuples of the fact table. The *TIDs* are sorted, in order to read each page only once. These tuples t = *(item_id, store_id, day_id, sales)* are materialized, i.e., read from the fact table via the *TIDs* (*ROWID* in Oracle, [Ora01]). The tuples t now are joined with the dimension tables and grouped and aggregated as specified in the query.

This method allows for a fast evaluation of the qualified tuples on the fact table, however, the materialization step can be very expensive, because for each TID, usually a random read access to the secondary storage is necessary. Generally, the fact table tuples are not clustered w.r.t. one or more dimensions as it is for clustering composite B-Trees or for clustering multidimensional indexes. Thus, a very small number of tuples on one disk page contribute to the result. We usually have to read *n* pages for *n* TIDs. However, the tuples can be "naturally" clustered, if they are inserted into the warehouse w.r.t. the time dimension. Often, data is loaded periodically on the smallest time granularity, e.g., every day. The load operation is an append to the data in fact table. Thus, each disk page contains only data of one day.

The basic concept is the transformation from an *equi-join* (the dimension key attributes of the dimension tables and the fact table) into a *semi-join*, in order to evaluate the fact table tuples efficiently.

The concept of *IBM DB2 UDB* also implements a semi-join for the fact table access. The dimension key attributes of the fact table are indexed by secondary indexes (usually standard B-Tree indexes). The restriction on the dimensions are evaluated resulting in a number of dimension keys for each leaf dimension table. These dimension keys are joined with the secondary indexes of the fact table. Each join results in a set of row identifiers (RID). One set of RIDs R_i for dimension D_i is used to build a dynamic bitmap that sets the bit for the row, if it is in R_i. The remaining bits are set to 0.

The next dimension D_j is chosen to compare it with the RID bitmap, in order to set bits that do not occur in R_j to 0. With this step, all rows (tuples) of the fact table are identified that are qualified by the restrictions of dimensions D_i and D_j. The remaining restricted dimensions are compared with the RID bitmap correspondingly. After this iterative comparison and bit setting, the RID bitmap determines all tuples of the fact table that have to be materialized. These tuples are read from the fact table (*materialization*) and are joined with the dimension tables again, in order to get the dimension attribute values necessary for grouping and aggregation. The tuples then are aggregated, grouped and sorted as far as required.

There are some optimization issues to speed up the RID bitmap intersection. The position in the RID bitmap *B* for the row id is not a bijective mapping (e.g., the identity: *RID = 1* ➔ *B[1]* is set etc.), but a mathematical optimized hashing function is used, in order to occupy less space for the bitmap. However, there are sometimes collisions resulting from the hash function. This means that some tuples of the fact table are materialized that do not contribute to the dimension predicates. The residual join has to post-filter, in order to decide that only correct tuples are processed.

The decision which dimension is used for building the bitmap depends on statistic information of the optimizer. With a reasonable cardinality and initial dimension, the time to evaluate the dimension intersection can be reduced significantly.

9.2 Star Query Template

In SQL Statement 9-1, we depict a SQL query template for ad hoc star-queries. The notation *{X}* represents a set of *X* objects. The above template defines typical star queries and uses abstract terms that act as placeholders. Note that the queries that conform to this template generally have a structure that is a subset of the above template and they instantiate all abstract terms.

Our template is applied on a schema similar to the star and snowflake schemata as described in Section 2.3 and 2.4. We use the term star queries for queries on star schemata and snowflake schemata. The *JOINPRED* expressions define the star join. Each dimension D_i is joined with the fact table via the leaf dimension table D_i^1: $D_i^1.h_1 = FACT.d_i$. The higher dimension tables D_i^k of Dimension D_i are joined depending on the snowflake schema (Section 2.4).

Apart from the star-join, there is a `GROUP BY` and `HAVING` clause. In general any attribute (hierarchical, feature, or measure) can appear in a `GROUP BY` clause. However, most queries will impose a grouping on a number of hierarchical and/or feature attributes. Finally, the `ORDER BY` clause controls the order of the presented results.

```
SELECT    {D.h}, {D.f}, {FT.m},
          {AGGR(FT.m) AS AM}, {AGGR(D.h) AS AH}, AGGR(D.f) AS AF}
FROM      FT,{D1},…,{Dn}
WHERE     JOINPRED(D1) AND
          JOINPRED(D2) AND
          ...
          JOINPRED(Dn) AND
          DIMPRED(D1)  AND
          DIMPRED(D2)  AND
          ...
          DIMPRED(DN)  AND
          FRESTR({FT.m})
GROUP BY  {D.h},{D.f},{FT.m}
HAVING    HPRED({AM},{AH},{AF})
ORDER BY  <ordering fields>
```

SQL Statement 9-1 : Query Template

`AGGR(x)` stands for a standard SQL aggregate function (`MIN, MAX, SUM, COUNT, AVG`). We use atomic expressions for illustration. However, we will extend the expression complexity later and discuss methods to handle complex expressions.

`DIMPRED(Di)` contains restrictions on dimension D_i. This restriction includes predicates on the leaf dimension table D_i and on the higher dimension tables D_i^k. It triggers the filtering on each dimension table. Also sub-queries can be part of `DIMPRED`. However, no joins with dimension tables of other dimensions are allowed (see Section 10.1).

`FRESTR({FT.m})` is a restriction that contains any constraints on measure attributes of the fact table, e.g., to ask for sales that exceed a certain value threshold.

We assume that each dimension has one hierarchy with the hierarchy levels h_t, h_{t-1}, …, h_1, where h_t is the most aggregated and h_1 the most detailed level of the hierarchy. A dimension with more than one hierarchies can be split into several dimensions each containing one hierarchy. We will discuss dimensions with multiple hierarchies in Section 10.6.

9.2.1 Hierarchical Surrogates

In Section 6 we have already discussed the effect of hierarchical clustering and the usage of compound surrogates. Hierarchical point restrictions are mapped to interval restrictions by using the hierarchical surrogate encoding. For the discussion of query processing we abstract from the compound surrogate encoding method, in order to discuss a general hierarchical encoding method.

We encode hierarchy paths by concatenating the hierarchy levels h_t, h_{t-1}, …, h_1 (h_t being the most aggregated level and h_1 the most detailed one). We use the notation $h_t/h_{t-1}/\ldots/h_1$ for these hierarchical surrogates, called *hierarchical surrogate key (hsk)* or simply *h-surrogate*.

The components h_k of the h-surrogate $h_t/h_{t-1}/\ldots/h_1$ correspond to the (encoded) surrogate components of the compound surrogates cs^k (see Definition 5-5). Other equivalent encoding are also possible.

9.2.2 Sample Schema for Grouping

As example throughout this section, we use a simple star DW schema as depicted in Figure 9-1. This data warehouse stores sales transactions recorded per item, store, customer, and date. It contains one fact table *SALES_FACT*, which is defined over the dimensions: *PRODUCT*, *CUSTOMER*, *DATE,* and *LOCATION* with the obvious meanings. The measures of *SALES_FACT* are *price, quantity,* and *sales* representing the values for an item bought by a customer at a store at a specific day. The schema of the fact and dimension tables is shown in Figure 9-1 and the dimension hierarchies are depicted in Figure 9-2. The meanings of the hierarchies are obvious.

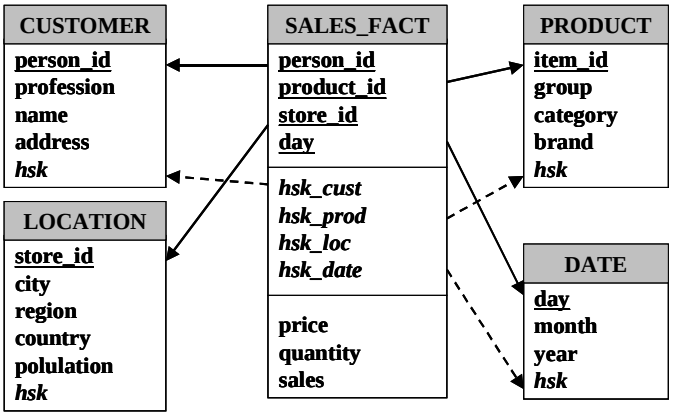


Figure 9-1. Sample schema

The *CUSTOMER* dimension has two hierarchical attributes (*person_id, profession*) and two feature attributes (*name, address*). The dimension *LOCATION* has four hierarchical attributes (*store_id, city, region, country*) and one feature attribute (*population*) that is assigned to the city level.

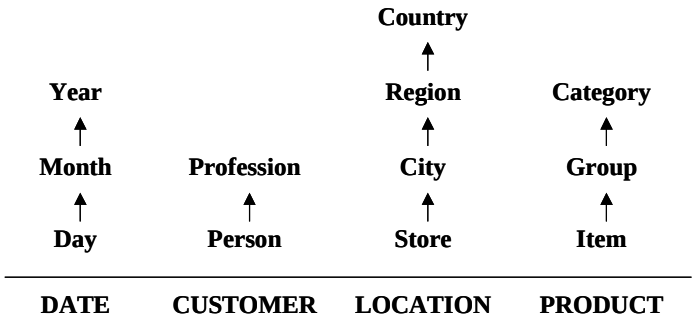


Figure 9-2. The dimension hierarchies of the example

Finally, the *PRODUCT* dimension is organized into three levels: *item – group – category* . The attribute *brand* characterizing each item is a feature attribute.

Note that all dimension tables contain the attribute *hsk*, i.e., the h-surrogate for the dimension hierarchy. These h-surrogates are referenced by *hsk_cust*, *hsk_prod*, *hsk_loc,* and *hsk_date* in the fact table.

9.2.3 Predicate Specification

The `WHERE` clause of a query instance for the star query template consists of join constraints and predicates. All predicates determining one dimension are called *dimension restriction (DIMRESTR).* All predicates specifying measure attributes are called *fact restriction (FRESTR)* and all restrictions of

one query are called *query restriction*. Note that we use the terms *restriction* and *predicate* equivalently.

Basically two different kinds of restrictions occur in star queries: *hierarchical* and *non-hierarchical* restrictions.

9.2.3.1 Hierarchical Restrictions

Hierarchical restrictions restrict dimension hierarchy level attributes to one or several points. Also an interval predicate on an enumeration data type can be seen as point restriction that specifies a number of points, .e.g., `year BETWEEN 2000 and 2002` is the same as `year IN (2000, 2001, 2002)`.

Hierarchical restrictions result in one or more intervals for hierarchical surrogates. We use the notation $h_t/h_{t-1}/*$ for an interval that contains all paths with the prefix h_t/h_{t-1}. A hierarchical restriction always leads to exactly one interval, if a hierarchy prefix is restricted to one point, e.g., `year = 2000 and month = 5`. We call such a restriction a *hierarchy prefix path (HPP)* restriction.

Definition 9-1 (Hierarchy Prefix Path, HPP):

A *hierarchy prefix path* restriction, *HPP*, is a restriction on hierarchy level attributes of a dimension, where alle levels from the most aggregated level h_t to a hierarchy level h_k are restricted to a point: $h_t = v_t$ AND $h_{t-1} = v_{t-1}$ AND … AND $h_k = v_k$. h_t is the most aggregated level (top level) of the hierarchy. A *HPP* always leads to one interval of the corresponding hierarchical surrogates: $h_t/h_{t-1}/\ldots/h_k/*$. □

If more than one *HPP* restrictions exist on one dimension, several intervals on the h-surrogate qualify the dimension result and consequently restrict the fact table.

A restriction on hierarchical predicates that do not form a hierarchy prefix is called hierarchy *non prefix path (HNPP)* restriction.

Definition 9-2 (Hierarchy Non Prefix Path, HNPP):

A *hierarchy non prefix path* restriction, *HNPP*, is a predicate on hierarchy level attributes of a dimension, where any hierarchy level attributes are restricted: $h_{k1} = v_{k1}$ AND $h_{k2}=v_{k2}$ AND … AND $h_{kn} = v_{kn}$, where $1 \leq k_i \leq t$ and is not a hierarchy prefix path restriction. A *HNPP* restriction leads to one or more intervals. □

A HNPP restriction usually specifies several intervals, because the hierarchy prefix is not restricted completely. For example consider the predicate `month = 5`. There are usually several years where the month number 5, e.g., “May”, occurs. Assume that the years 2000, 2001 and 2002 are stored in the hierarchy. Then the restriction `month = 5`, equal to */5/*, can be mapped to three HPP restrictions: `(year = 2000 AND month = 5) OR (year = 2001 AND month = 5) OR (year = 2002 AND month = 5)` resulting in the three intervals *2000/5/**, *2001/5/**, and *2002/5/**.

Note that it cannot be decided statically (at query compile time), whether a HNPP restriction results in one or several intervals without looking at the dimension table.

9.2.3.2 Non-Hierarchical Restrictions

Restrictions where not only hierarchy level attributes are involved, are called non-hierarchical restrictions. Such predicates usually contain restrictions on feature attributes or measure attributes. We consider non-hierarchical predicates for the dimensions for this predicate evaluation step. Measure attributes of the fact table are handled later in the fact table access.

A restriction on dimension D_i is *non-hierarchical*, if it restricts at least one feature attribute f_j of D_i. Non-hierarchical restrictions on dimensions lead to several intervals. Feature attributes can be orthogonal to the hierarchy and split an interval defined by a *HPP* or *HNPP* restriction to a large number of very small intervals, sometimes even points.

Consider for example a customer hierarchy with geographic hierarchy levels and the feature attribute *age* (of the customer). Restricting the hierarchy to all customers living in "Germany" with an age of 30 results in a number of customers not necessarily forming an interval. We introduce a method how to use hierarchical properties even in these cases (see Section 10.1.8 for more details).

Restrictions on the fact table measure attributes are always non-hierarchical restrictions. Thus, a query with a predicate *FRESTR({FT.m_j})* (see SQL Statement 9-1) has a non-hierarchical restriction on the fact table.

9.3 Abstract Execution Plan with Interval Generation

In this section we describe the major processing steps entailed when we want to answer star-queries over a hierarchically clustered fact table. It has been shown ([DRSN98], [MRB99], [KS01]) that evaluating queries including hierarchical restrictions on a hierarchically clustered fact table provides significant gains in performance.

We introduce an abstract *execution plan* (AEP) to illustrate the basic processing steps by *abstract operators*. The abstract operators only define a semantic, e.g., a join operator does not specify how the join is implemented (via sort merge, nested loop or hash join).

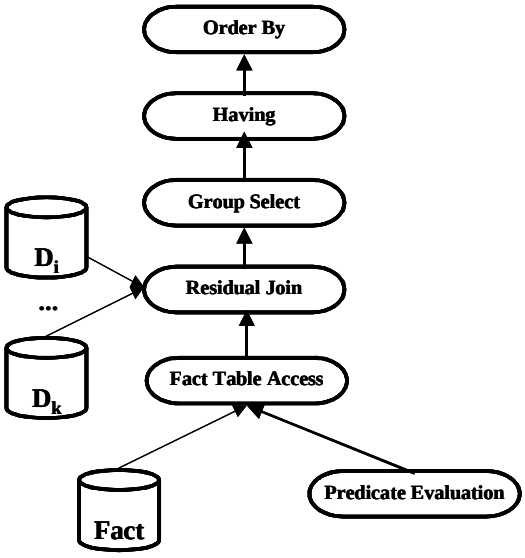


Figure 9-3: Standard Abstract Execution Plan

The processing begins with the evaluation of the restrictions on the individual dimension tables, i.e., the evaluation of the predicates (Section 9.2) in the *Predicate Evaluation* operator. We call the tuples retrieved from the fact table *fact table result tuples*. These tuples are joined with the dimension tables that are necessary for further processing, in order to retrieve dimension table attributes for grouping, aggregating and projection. The *Residual Join* operator can be implemented also as n-way join, depending on the capabilities of the DBMS. The *Group Select* operator groups and aggregates the joined tuples according to the `GROUP BY` and `SELECT` clause. If post-filtering by a `HAVING` clause is required, groups are removed from the result groups. Finally, the *Order By* operator sorts the result set w.r.t. the attributes as specified in the `ORDER BY` clause of the query.

Not all operators in the abstract plan may be needed for the execution of a particular query. The plan rather represents the most complex abstract plan that might be required to answer a supported query. For example, if the result records are not required in a specific order then the final *Order By* operator will not be applied. Also, many queries will not restrict all available dimensions nor require feature or hierarchical attributes from all dimension tables. This means that only a restricted number of *Residual Join* operators will be used. In the simplest possible query (`SELECT * FROM` Fact) only the *Fact Table Access* operator is needed.

The following sections describe the operators in more detail.

9.3.1 Predicate Evaluation

Recall that we assume a star schema with a hierarchically multidimensional clustered fact table. Also the dimension tables are organized by hierarchical encoding. With this physical organization, we get a special predicate evaluation optimization.

As described in Section 5.3, a hierarchical prefix restriction on a dimension hierarchy, e.g., h_t = v_t AND h_{t-1} = v_{t-1} AND … AND h_k = v_k describes an interval on the h-surrogates, i.e., $h_t/h_{t-1}/\ldots/h_k/*$. This interval can be evaluated efficiently by the underlying multidimensional index on the fact table.

The original restriction `DIMPRED` of a dimension D_i that is equi-joined with the fact table is transformed to a semi-join specifying an interval for the fact table access.

Figure 9-4 shows the abstract execution plan divided into two processing phases, the *h-surrogate processing* and the *main execution phase*. The h-surrogate processing phase illustrates the semi-join transformation by computing the intervals resulting from the `DIMPRED` restrictions for the dimensions of the query.

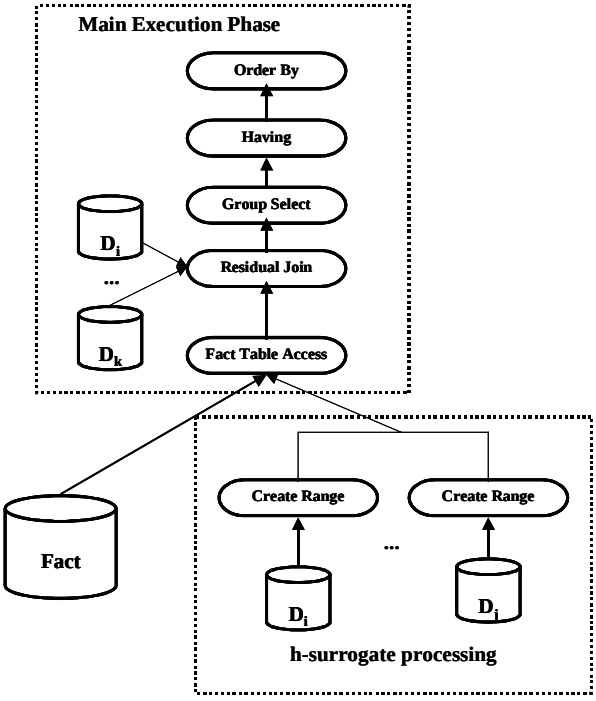


Figure 9-4: Abstract Execution Plan with Main Processing Phases

Create Range is responsible for evaluating the restrictions on each dimension table. This evaluation results in an h-surrogate specification (set of ranges) for each dimension. All these together define one (or more, disjoint) hyper-rectangle(s) in the multidimensional space of the fact table.

Fact Table Access receives as input the h-surrogate specifications from the *Create Range* operators and performs a "range query" on the underlying multidimensional structure that holds the fact table data. Apart from the selection of data points that fall into the desired ranges, this operator can perform further filtering based on predicates on the measure values and projection (without duplicate elimination) of fact table attributes. In the case of many query boxes generated by the dimension predicates, special optimization is necessary (see Section 10.3 for more details).

For the vast majority of dimension restrictions, the *Create Range* operator can be implemented very efficiently. If we consider hierarchical prefix path (*HPP*) restrictions (see Section 9.2.3), then the first matching tuple on each dimension suffices in order to retrieve the appropriate h-surrogate value that generates the ranges. For example, if we have the restriction `PRODUCT.category = "air condition" AND PRODUCT.class = "A"`, then essentially what we want is all the leaves of the sub-tree with root "air condition"/"A"/ defined in the tree instantiating the hierarchy of dimension *PRODUCT*. Therefore, if we retrieve the h-surrogate value of the first tuple that qualifies and truncate the part from the right that corresponds to level *Item*, then this will be the same for all matching tuples. Next we can use this truncated h-surrogate value in order to create a range.

Moreover, if we have stored more information on the correlation between the attributes of a dimension, apart from the definition of the hierarchy, then we can benefit from the above processing scheme, even for *non-HPP* restrictions. Suppose we have a hierarchy $h_t, h_{t-1},\ldots, h_1$ on a dimension and we have a restriction of the form: $h_k = c_1$ AND $h_p = c_2$ AND … $h_i = c_i$, where $h_k, h_p,\ldots,h_i$ do not form a prefix of $(h_t,\ldots,h_1)$ and h_i is the most detailed of the referenced attributes. If we know that h_i functionally determines h_j, for all $j > i$, then we can still apply the above strategy. For example, for the restriction `DATE.month = "AUG99"`, we know that the *month* attribute determines the *year* attribute and thus only the first tuple that has this value for month suffices for our processing needs. Similar observations hold for the restrictions on the feature attributes as well.

A very simple but also quite drastic optimization strategy for the processing of the *Create Range* operator, is the use of a composite (B*-Tree) index, for each dimension table D_i, defined over the attributes $h_t, h_{t-1},\ldots,h_1, hsk$. This index's purpose is twofold: (a) it can be used to speedup the retrieval of D_i tuples, when a hierarchical prefix path restriction appears in a local predicate for D_i and (b) it can also be used as a table that stores the mapping between hierarchical prefix paths and h-surrogate values. The former use is the classic exploitation of an index. The latter gives us the opportunity to use this index solely to evaluate all predicates that contain restrictions on hierarchical attributes only (and not on feature attributes), without accessing D_i. This is possible regardless of the existence, or not, of a hierarchical prefix match. Even if we do not have a match with the search-key of the index and we have to fully scan the index, this will be obviously more efficient than scanning the dimension table D_i. Naturally, smaller tuples of the index will deliver us the required h-surrogate values with much less I/O cost than if we had to read the D_i tuples.

Such a physical schema, i.e., a secondary index *DXh* on the hierarchy level attributes and the h-surrogate resp. compound surrogate are introduced in Section 4.4.4.

Another issue worth mentioning is that in some cases, dimension restrictions can result in a number of intervals or even distinct h-surrogate values. This inevitably will result in a large number of range queries. However, very often this evaluation produces a set of h-surrogate values that belong to the same "family" in the hierarchy and thus can be merged into a single interval, reducing this way the total number of intervals created. For example, a restriction on the *LOCATION* dimension with `population > 1000000`, could result in two areas that can be expressed by two intervals. The restriction, however, may qualify a large number of hierarchy paths with a corresponding number of distinct h-surrogates. A clever h-surrogate processing phase can detect such cases and reduce the number of intervals by merging h-surrogates of the same area. This would generate two intervals instead of a large number. We call such a method "values to interval". An example for an implementation that merges a number of h-surrogates (compound surrogates) to a set of intervals is the *cs2ival* operator as described in Section 10.1.8.1.

We describe the implementation of predicate classes, i.e., *HPP, HNPP,* and *NH* in more detail in Section 10.1.8.

9.3.2 Residual Join

Residual Join is a join on a key-foreign key equality condition among a dimension table and the tuples originating from the *Fact Table Access* operator. Recall that the schema requires a foreign key relationship between the dimension attributes of the fact table and the most detailed level of the dimension table: `Fact.d`$_i$ `= D`$_i$`.h`$_1$. Thus, each incoming fact table tuple is joined with *at most one* dimension table record. The join is performed in order to enrich the fact table records with the required dimension table attributes. These attributes may be required in the `SELECT`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses.

For star queries we often have to join a large number of fact table result tuples. Since each tuple is joined with a number of dimensions, the overall number of join operations is very high. Assume that we have 100.000 fact table result tuples and four dimensions to join. The number of join operations is 4 x 100.000 = 400.000. Generally the number of join operations is *n* * *fnr*, where *n* is the number of dimensions to join and *fnr* is the number of fact table result tuples.

For this purpose we suggest an implementation alternative. We use equivalence classes of dimension attributes of the fact table and perform the join only once per equivalence class. Usually the join is defined via the dimension key d_i of the fact table and the most detailed hierarchy level h_1 of dimension D_i. An equivalence class is determined by a common h-surrogate prefix. The hierarchy levels h_t, h_{t-1}, …, h_k are the same for all tuples of D_i that have the same h-surrogate prefix $h_t/h_{t-1}/\ldots/h_k$. Thus, it is enough to perform one join operation with $Fact.d_i$ resp. $D_i.h_1$ for all fact table result tuples with the same prefix h_t, h_{t-1}, …, h_k (or functional dependent attributes).

We create a hash table for each dimension that has to be joined. In this hash table we use the h-surrogate prefix $h_t/h_{t-1}/\ldots/h_k$ as hash key and store the (joined) dimension table attributes for this h-surrogate prefix as additional attributes in the hash table. For each fact table result tuple we look into the corresponding hash table, if there is already a tuple with the same h-surrogate prefix. If there is no tuple, the join `Fact.d`$_i$ `= D`$_i$`.h`$_1$ is performed and the result is stored in the hash table.

With this method, the number of lookups in the dimension table can be reduced significantly. However, if feature attributes are needed that are not known to be functionally dependent on one of the hierarchy levels of the h-surrogate prefix, we must perform the join conventionally, i.e., one join per tuple and required dimension.

An additional reduction of join operations is to push down the Group Select operator, in order to group on the h-surrogates of the fact table and thus reduce the number of fact table result tuples significantly. The consecutive join affects a significantly smaller number of tuples. We discuss this optimization in Section 9.4.

9.3.3 Group Select

The *Group Select* operator is responsible for the grouping of the fact table result tuples. Grouping is usually done on hierarchical attributes or feature attributes. These attributes are not available in the fact table and must have been joined by the former residual join, in order to have the grouping values available.

The implementation of the Group Select operator depends on the optimizer decisions. In our implementation we use a simple hash group algorithm.

The *Group Select* operator reduces the number of result tuples significantly, since we have usually hierarchical grouping. For hierarchical grouping a performance optimization is possible. We group on

h-surrogate prefixes of the fact table reducing the number of result tuples and thus the number of join operations. This method is described in Section 9.4 in more detail.

9.3.4 Having

The `HAVING` clause, implemented by the *Having* operator post-filters the result of the query by removing groups. The groups are removed w.r.t. predicates specified in the `HAVING` clause (see SQL-92 standard and [DD93]).

It is used usually in combination with a group by clause, in order to define predicates on aggregation results:

```
SELECT gfield, AGG(afield), …
…
GROUP BY gfield
HAVING HAVING_CONDITION
```

where `HAVING_CONDITION` is a sequence of expressions consisting of expressions like *AGG(afield)) OP expr*. *OP* is an arithmetic operator, e.g., <, >, = and *expr* is an expression. An example for a SQL statement with a `HAVING` clause is:

```
SELECT area, SUM(population)
FROM …
WHERE …
GROUP BY area
HAVING SUM(population) > 1000000
```

This statement returns the areas with a population larger than 1.000.000.

9.3.5 Order By

If the result of a star query has to be sorted, the sort attributes and the sort order are specified in the `ORDER BY` clause. The sort operation is performed as last operation in the AEP by the *Order By* operator. The implementation of the sort operation depends on the DBMS. Consider that sorting is performed externally, if the available main memory is not sufficient.

9.3.6 Measurements

In this section, we present a comparison between conventional star join processing with secondary indexes and the proposed execution plan with MHC and UB-Tree organized fact table. The measurements are performed on a schema similar to the Sales DW. It consists of a fact table with three dimensions *Customer*, *Product,* and *Calendar* and three measures: *quantity*, *value,* and *unit_price*. The *Customer* dimension contains 1,4 million records, *Product* consists of 27.000 products and the *Calendar* dimension covers 7 years on day granularity. The fact table data was enlarged to 15.543.380 records, amounting to 1,5 GB.

The query workload consists of 220 ad hoc star queries from the real world Sales DW application. We classify the queries into three groups according to their selectivity on the fact table (i.e., number of tuples retrieved from the fact table):

- [0.0-0.1]: 0% to 0.1% of fact table, i.e., 0 to about 15K records
- [0.1-1.0]: 0.1% to 1% of fact table, i.e., 15K to 160K records
- [1.0-5.0]: 1.0% to 5.0% of fact table, i.e., 160K to 780K records

The goal of the performance evaluation was to measure two alternative execution plans:

- the conventional star join plan (*STAR*),
- the abstract execution plan as described before (called *AEP*)

STAR uses secondary indexes that are created on the dimension keys of the fact table. The restrictions on the dimension tables are evaluated and the resulting dimension keys are used for index intersection on the fact table. The resulting records are joined with the dimension tables, in order to perform grouping and get the final result. This is the typical processing of star queries in commercial DBMSs (e.g., *star transformation* as described in Section 4). This processing has two major steps: the index intersection and the tuple materialization. While the index intersection has largely been optimized (e.g., with bitmap indexes [NQ97]) the materialization of results is still the bottleneck of non-clustering indexes. Consequently, we neglect the index intersection time for STAR and just measure the time for fact record materialization, residual joins and grouping. For AEP the complete processing including index access is measured, therefore favoring STAR.

FT Sel. %	[0.0-0.1]		[0.1-1.0]		[1.0-5.0]	
	STAR	**AEP**	**STAR**	**AEP**	**STAR**	**AEP**
MIN	0	0	65	2	274	11
MAX	30	6	290	9	1219	47
MEDIAN	1	1	182	8	477	23
STD-DEV	5	1	76	3	346	14

Table 9-1: Response Time in Seconds of Secondary Indexes and MHC

Table 9-1 shows the response time analysis (in seconds) for the three alternative processing plans. As the three classes contain queries with different result set size and thus different response times we use the maximum, minimum, median time and the standard deviation to analyze the performance.

Our results show that the standard STAR processing is outperformed by our approaches. However, for small queries, i.e., the class [0.0-0.1], the speedup is below an order of magnitude. In general, for small result sets, the advantage of clustering over non-clustering is not that large. The picture changes drastically, when we consider larger queries (classes [0.1-1.0] and [1.0-5.0]), which are more typical for OLAP applications. The hierarchical clustering of AEP leads to an average speedup compared to STAR of 24.

Note also that STAR has a very high deviation in the response times for queries within one class. This is mainly for two reasons: (a) STAR performance deteriorates very fast as the fact table selectivity is increased and (b) since the fact table is not stored clustered the number of performed I/Os may differ significantly from one query to another. On the other hand, the deviation for AEP and OPT remains low, showing a much more stable behavior.

9.4 Grouping Optimization: Pre-Grouping

Pre-grouping techniques have been proposed to reduce the number of join tuples by introducing a grouping and aggregation phase before the first residual join [CS94]. These techniques rely on the ability to exploit functional dependencies among attributes of a table. In DW applications, the grouping and aggregation is usually specified on the hierarchies of the dimensions, limiting the effect of these pre-grouping methods.

While the h-surrogates were originally designed to improve the clustering and indexing of the fact table, they can also be used efficiently in pre-grouping. With the hierarchy semantics encapsulated in the h-surrogates the pre-grouping algorithms can exploit more functional dependencies and thus achieve a much higher reduction of the number of join tuples.

A simple example illustrates the effects of the hierarchical pre-grouping method: Assume we have a DW with a time dimension (besides other dimensions) categorized by *year – month – day* and a well populated fact table. A query restricting the result to year 2001 (besides other restrictions) qualifies 100.000 fact table records. If the result has to be grouped w.r.t. month, we have to join 100.000 records with the time dimension table. When applying pre-grouping, the number of join operations is reduced only marginally (depending on grouping on other dimensions). When applying hierarchical pre-grouping, the number of join operations is reduced by a factor of 30, because all days of one month are aggregated to the month using the hierarchical encoding information in the fact table. In [PER+03a] we introduced the concept of hierarchical pre-grouping.

We first explain the group operations in SQL queries and then discuss the concept of pre-grouping, especially hierarchical pre-grouping.

9.4.1 Grouping

In SQL statements, it is possible to partition tuples into groups. All tuples that belong to one group (same grouping attribute values) are represented by one tuple, where the non-grouping attribute values are aggregated according to the aggregation functions.

9.4.1.1 General Description of Grouping

Grouping is a standard SQL operator. This `GROUP BY` operator partitions a set of tuples into disjoint tuple sets S_k and then aggregates over each set resulting in one tuple t_k' per tuple set S_k.

Definition 9-3 (Grouping Attribute, Grouping Value):

The *grouping attributes* are the attributes that are specified in the `GROUP BY` operator. All tuples with the same *grouping values*, i.e. the values of the grouping attributes, are merged to one tuple □

Definition 9-4 (Aggregation Attribute):

The aggregation attributes occur in the `SELECT` list or `HAVING` clause and are used in aggregation functions. □

Definition 9-5 (Aggregation Function):

An aggregation function $agg: 2^{\Delta 1} \rightarrow \Delta_2$, $agg(\{a_i\}) = a$ returns a value a computed from a multi set of values a_i of domain Δ_1. The domains Δ_1 and Δ_2 can be different domains. Aggregation functions of the SQL-92 standard are `MIN`, `MAX`, `SUM`, `COUNT`, `AVG`. □

Note that in SQL also duplicates are allowed and are considered in the aggregation calculation. Thus, we do not deal with relations in the sense of relational algebra, but with multi sets.

Figure 9-5 shows the basic idea of grouping. The original set of tuples R is partitioned into n sets $\{ S_1, S_2, \ldots, S_n \}$ where the grouping attributes g_1 and g_2 have the same values for each set S_k. The aggregation attributes are aggregated according to the aggregation function, i.e., *MIN* for a_1, *MAX* for a_2 and *SUM* for a_3. The result of the grouping and aggregation operations is a set $G = \{ t_1', t_2', \ldots, t_n' \}$, i.e., one tuple for each group S_k.

Grouping is a mapping g from a set R of tuples to a set G of tuples: $g_{a1,a2,\ldots,ak,agg1,agg2,\ldots,aggj}: 2^T \rightarrow 2^{T'}$, where T and T' are a set of tuples. $a_1, a_2, \ldots, a_k$ are the grouping attributes and $agg_1, agg_2, \ldots, agg_j$ are the aggregation functions. Using R and G, grouping is the mapping $g_{\{ai\}\{aggj\}}(R) = G$, where R is the source tuple set, $R = \{ t_i \}$ and G is the set of resulting tuples, $G = \{ t_i' \}$. Note further that $| R | \geq | G |$, i.e., the grouping operation reduces the number of tuples.

After partitioning R into disjoint tuple sets S^k, $S^k \in 2^T$ the tuples of S^k are merged to one tuple via the aggregation operation, i.e., $f: 2^T \rightarrow T$, $f(S^k) = t$.

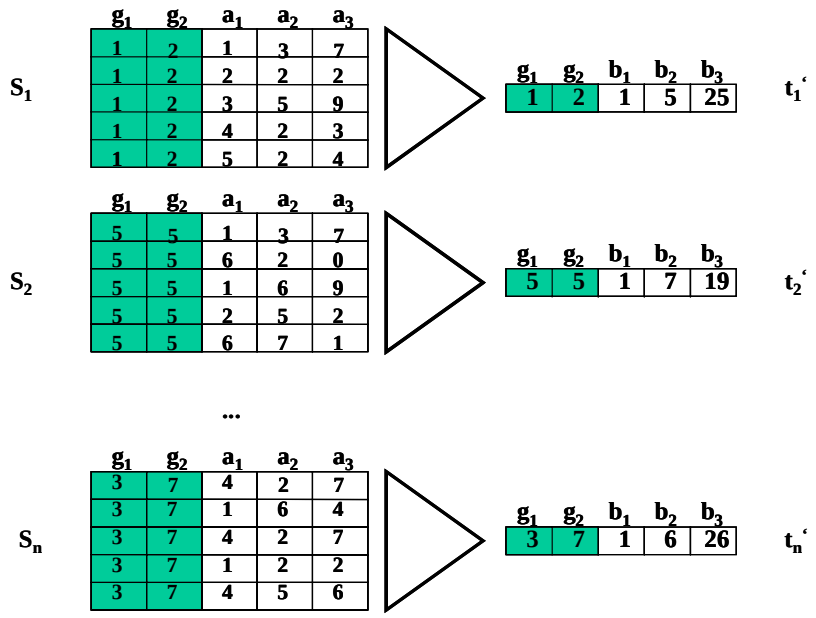


Figure 9-5: GROUP Operator: SELECT g_1, g_2, MIN(a_1), MAX(a_2), SUM(a_3) FROM R GROUP BY g_1, g_2

9.4.1.2 Grouping Properties

This section considers some properties of grouping w.r.t. functional dependencies of attributes. We discuss grouping extensions and grouping equivalences.

Definition 9-6 (Row Equivalence):

Consider a table *R(…, K, …)*, where $K = \{k_1, k_2, \ldots, k_n\}$ is a set of attributes. Two rows *t, t' ∈ R* are equivalent, w.r.t. *K*, if: $\bigwedge_{i=1,\ldots,n} (t.k_i = t'.k_i)$, which we also write as $t.K \overset{n}{=} t'.K$. □

Definition 9-7 (Functional Dependency, Functional Determination):

Consider a table *R(K, A, …)*, where $K = \{k_1, k_2, \ldots, k_n\}$ and $A = \{a_1, a_2, \ldots, a_j\}$ are a set of attributes. *K functionally determines A*, denoted by $K \rightarrow A$, if the following condition holds:

$\forall t, t' \in R, \{(t.K \overset{n}{=} t'.K) \Rightarrow (t.A \overset{n}{=} t'.A)\}$. We also say, *A* is *functionally dependent* on *K*. □

The key attributes (and candidate key attributes) always functionally determine all remaining attributes of *R*. There also can be chains of functional dependencies: $a_i \rightarrow a_j \rightarrow \ldots \rightarrow a_k$. In this case, a_k is also functionally dependent on a_i: $a_i \rightarrow a_k$ For two different values of a_j there can be the same value for a_i, if $a_j \rightarrow a_i$.

h-surrogates *hsk* are built from the combination of the corresponding hierarchy levels, $hsk = h_t/h_{t-1}/\ldots/h_1$, where h_t is the most aggregated (top level) and h_1 the most detailed level (leaf level) of the hierarchy. Recall that due to the hierarchical relationship of the hierarchy levels $h_1, h_2, \ldots, h_t$ the following hierarchical (functional) dependency holds: $h_1 \rightarrow h_2 \rightarrow \ldots \rightarrow h_t$.

In SQL-92 we are not able to express functional relationships $h_1 \rightarrow h_2 \rightarrow \ldots \rightarrow h_k$ within one table. Thus, additional meta data is necessary for this information. For example, in a time hierarchy we usually have the functional dependency chain: *Day* → *Month* → *Year* (e.g., "20020630" → "June2002" → "2002").

We use a similar notation as introduced with generalized projections in [GHQ95]. According to [GHQ95], grouping can be considered as a projection. The framework described in [GHQ95] is an intuitive framework for aggregation operators as an extension of duplicate elminating projection operators. Duplicate elimination projection is the simplest form of aggregation, because it can be expressed as a simple `GROUP BY` statement that does not compute any aggregates. We write

$\pi_{D}(R)$ for `SELECT D FROM R GROUP BY D` and

$\pi_{agg(a),D}(R)$ for `SELECT agg(a), D FROM R GROUP BY D`.

In general, we write

$\pi_{\{agg\},\{g\}}(R)$ for `SELECT { agg }, { g } D FROM R GROUP BY { g }`,

where `{agg}` is a set of aggregate functions `agg(a)`, e.g., `SUM(a)` and `{g}` is a set of grouping attributes. For example we use

$\pi_{SUM(a),c,d}(R)$ for `SELECT SUM(a), c, d FROM R GROUP BY c,d`

Consequently, for nested queries like

```
SELECT SUM(a'), f(D) FROM
      SELECT SUM(a) AS a', D FROM R GROUP BY D
GROUP BY f(D)
```

we write

$\pi_{sum(a'),f(D)}(\pi_{sum(a)\ as\ a',D}(R))$

f(D) is for example a prefix of *D*, if *D* consists of components $D = (d_1, d_2, \ldots, d_k)$.

Thus, a generalized projection produces from a relation *R* a new relation according to the subscript. The generalized projection results in exactly one tuple for each value of the grouping attributes and produces not duplicates in its output ([GHQ95]).

In this section we consider only `MIN`, `MAX` and `SUM` as aggregation functions, because these functions have the same implementation for a sequence of grouping operations. The aggregation functions `COUNT` and `AVG` are explained later.

We use $\Pi_a(R)$ as conventional projection operator doing the statement `SELECT a FROM R`. We do not consider aggregation explicitly in the following lemmata. Special computation algorithms are necessary to do aggregation as explained in Sections 9.4.2.2 and 10.2.

Lemma 1:

$\pi_a(R) = \Pi_a(\pi_{a,b}(R))$, if a → b. □

Proof:

From the grouping properties follows, that for $\pi_a(R)$ for each tuple t_i of the set $S^j = \{ t_i \}$ the value v_a of *a* is the same. From Definition 9-7 follows that for each value v_a the value v_b of the functional dependent attribute *b* is the same. Thus all tuples $t_i \in S^j$ have the same values for *(a, b)* and $\pi_a(R)$ is equivalent to $\pi_{a,b}(R)$ with a projection to a, i.e., $\Pi_a(\pi_{a,b}(R))$. □

Lemma 2:

$\pi_a(R) = \Pi_a(\pi_{a,b}(R))$, if $a \xrightarrow{*} b$. □

Proof:

For the functional dependency chain $a \xrightarrow{*} b = a \rightarrow a_{k1} \rightarrow a_{k2} \rightarrow \ldots \rightarrow a_{kn} \rightarrow b$ the following grouping equivalences hold (w.r.t Lemma 1):

$\pi_a(R) = \Pi_a(\pi_{a,ak1}(R)) = \Pi_a(\pi_{a,ak1,ak2}(R)) = \ldots = \Pi_a(\pi_{a,ak1,ak2,\ \ldots,\ akn}(R)) = \Pi_a(\pi_{a,ak1,ak2,\ \ldots,\ akn,b}(R)) = \Pi_a(\pi_{a,ak1,ak2,\ \ldots,\ akn-1,b}(R)) = \ldots = \Pi_a(\pi_{a,ak1,ak2,b}(R)) = \Pi_a(\pi_{a,ak1,b}(R)) = \Pi_a(\pi_{a,b}(R))$ □

Lemma 3:

$\pi_b(\pi_{a,b}(R)) = \pi_b(R)$, if a → b. □

Proof:

$\pi_{a,b}(R)$ results in a set of tuples $G = \{ t_i' \}$ where each tuple t_i' has a unique attribute combination *a, b*. From Lemma 1 follows that each tuple t_i' has a unique value for *a*. Since *b* can have the same value for different *a* (functional dependency property), a grouping $\pi_b(R)$ can further reduce the numbers of result tuples, but contains the same values *a*. □

9.4.1.3 Grouping and h-Surrogates

This section describes functional dependencies of h-surrogates and hierarchy levels. The h-surrogate *hsk* is built from the combination of the corresponding hierarchy levels, $hsk = h^t/h^{t-1}/\ldots/h^1$, where h^t is the most aggregated (top level) and h^1 the most detailed level (leaf level) of the hierarchy. Without loss of generality, we assume that the hierarchical dependency for the hierarchy levels $h^1, h^2, \ldots, h^t$ is given: $h^1 \rightarrow h^2 \rightarrow \ldots \rightarrow h^t$.

In general, we have the following functional dependencies: $h_1 \rightarrow \{h_2, h_3, \ldots, h_t\} \rightarrow \{h_3, h_4, \ldots, h_t\} \rightarrow \ldots \rightarrow \{h_{t-1}, hat\} \rightarrow h_t$. The hierarchy path „20020427“ – „200204“ – „2002“ is an example for a *Date* hierarchy *Day* → *{Month, Year}* → *Year*.

In the following, we use hierarchy levels with $h^1 \rightarrow h^2 \rightarrow \ldots \rightarrow h^t$. Such hierarchy levels are unique, e.g., a hierarchy path for the Date dimension is “20020427” – “200204” – “2002”.

Definition 9-8 (h-surrogate infix, hsk infix):

An *h-surrogate infix* (*hsk infix*) hsk(m:n) consists of a subset of the h-surrogate components: $hsk(m:n) = h^m/h^{m-1}/\ldots/h^{n+1}/h^n$, where $1 \le n \le m \le t$, i.e., all hierarchy levels from h^m to h^n.

□

Informally speaking, *hsk(m:n)* specifies a sub-tree of the hierarchy with root level h^m and leaf level h^n. Note that for the funcional dependency $h^1 \rightarrow h^2 \rightarrow \ldots h^m \rightarrow h^{m+1} \rightarrow \ldots \rightarrow h^n \rightarrow h^{n+1} \rightarrow \ldots \rightarrow h^t$ the path from h^m to h^t is defined completely and *hsk(m:n)* also specifies the hierarchy prefix.

Examples for hsk(m:n):

$hsk(t:1) = h^t/h^{t-1}/\ldots/h^1 = hsk$

$hsk(t:t) = h^t$ (top level)

$hsk(t:k) = h^t/h^{t-1}/\ldots/h^k$ (h-surrogate prefix)

Definition 9-9 (h-surrogate prefix, hsk prefix):

An *h-surrogate prefix* (*hsk prefix*) is an h-surrogate infix with the top level h^t as upper bound and a level h^k as lower bound: $hsk(t:k) = h^t/h^{t-1}/\ldots/h^k$, $1 \leq k \leq t$. □

h-surrogate prefixes play an important role for hierarchical pre-grouping as described in Section 9.4.2. We now discuss functional dependency properties between h-surrogate prefixes and hierarchy levels.

Lemma 4:

$hsk(t:k) \rightarrow \{h^t, h^{t-1}, \ldots, h^k\}$ □

Proof:

Trivial by construction of *hsk(t:k)*, see Definition 9-9. □

Lemma 5:

$hsk(t:k) \rightarrow h^k$ □

Proof:

$hsk(t:k) = h^t/h^{t-1}/.../h^k \rightarrow h^k$, because $h^k \rightarrow h^{k+1} \rightarrow ... \rightarrow h^t$ (first Armstrong Axiom). □

Lemma 6:

$hsk(t:k) \rightarrow h^j$, where $k \leq j \leq t$. □

Proof:

$j = t$: $hsk(t:t) = h^t \rightarrow h^t = h^j$.

$j = k$: $hsk(t:j) = h^t/h^{t-1}/.../h^j \rightarrow h^j$.

$k < j < t$: $hsk(t:k) = h^t/h^{t-1}/.../h^k \rightarrow h^t/h^{t-1}/.../h^{k+1} \rightarrow ... \rightarrow h^t/h^{t-1}/.../h^j = hsk(t:j) \rightarrow h^j$ □

Corollary:

$hsk(t:k)$ functionally determines all h^j, $k \leq j \leq t$ and all combinations $(h^{j1}, h^{j2}, ..., h^{jn})$, where $k \leq j_1, j_2, \ldots, j_n \leq t$. □

Proof:

Applying Lemma 4, Lemma 5, Lemma 6. □

9.4.2 Concept of Pre-Grouping

9.4.2.1 Early Grouping

The *early grouping* technique as proposed in [YL94] and [CS94] uses knowledge about functional dependencies of the join tables. Consider a query restricting a table *A* and grouping on attribute *g* of table *B*: *B.g*. The tables *A* and *B* are joined by the attributes *a* of *A* and *b* of *B*: *A.a* = *B.b*, where *b* is primary key of *B*. We can group on *A.a* before joining with *B*, because *A.a* functionally determines *B.g* via the join condition and the key properties.

The resulting groups of the early grouping step are a subset of the final groups, because *B.g* can have the same value for distinct *B.b*. An additional final grouping step is necessary.

The abstract execution plan is the same as for hierarchical pre-grouping and is shown in Figure 9-6. However, the implementation of the `Pre-Group` operator differs from the early group operator.

The dimension tables D_{ei}, …, D_{ej} are joined *before* post-grouping (e = early), D_{l1}, …, D_{ln} are joined *after* post-grouping (l = late). The second residual join (late residual join) is an optimization to delay the residual join for the dimension tables D_{l1}, …, D_{ln}. This delay is possible, if the early grouping is already exact (see also Section 10.2).

9.4.2.2 Hierarchical Pre-Grouping

Hierarchical pre-grouping is an extension of early grouping. We use h-surrogate prefixes instead of the grouping attributes as specified in the `GROUP BY` clause. We assume that the DBMS has information about hierarchical relationships of the dimension attributes, e.g., via meta data in the data dictionary. These metadata are used by the optimizer to determine the parameters of the pre-group operator.

Instead of applying pre-grouping on the user defined join attribute which has the finest granularity of the hierarchy, we group on the hierarchy level h_k as specified in the `GROUP BY` clause. Note that h_k is not yet available in this pre-grouping step. Thus we use the corresponding h-surrogate (available in the fact table). The h-surrogate prefix $h_t/h_{t-1}/\ldots/h_k$ reduces the number of resulting groups dramatically. Thus pre-grouping takes place on the fact table result tuples. Usually final grouping is necessary to merge the groups of the pre-grouping phase.

The groups of the pre-grouping operation are joined with the dimension tables, in order to get the values for the (user defined) grouping attributes. Figure 9-6 shows the execution plan with the `Pre-Group` operator.

Example 9-1 (Star Query Example)

```
SELECT
  P.category, L.city, max(L.population), D.month, SUM(F.sales)
FROM
  SALESFACT F, LOCATION L, DATE D, PRODUCT P
WHERE
  F.day = D.day AND F.store_id = L.store_id AND
  F.product_id = P.product_id AND
  D.year='1999' and L.region='North'
GROUP BY
  I.category, L.city, D.month
```

In this example query based on the schema of Figure 9-1, we can pre-group on the dimensions *Product* on *category* level (*hsk(3:3)*), *Location* on *city* level (*hsk(4:2)*), if the feature attribute *population* is known to be functionally dependent on *area*, on store *level* (*hsk(4:1)*) otherwise. Pre-grouping on the *Date* dimension can be done on the *year* level (*hsk(3:3)*). For the *Location* dimension, we need final grouping and a residual join between the groups of the pre-group phase and the *Location* dimension table. After final grouping, we join the dimension tables *Date* and *Product*.

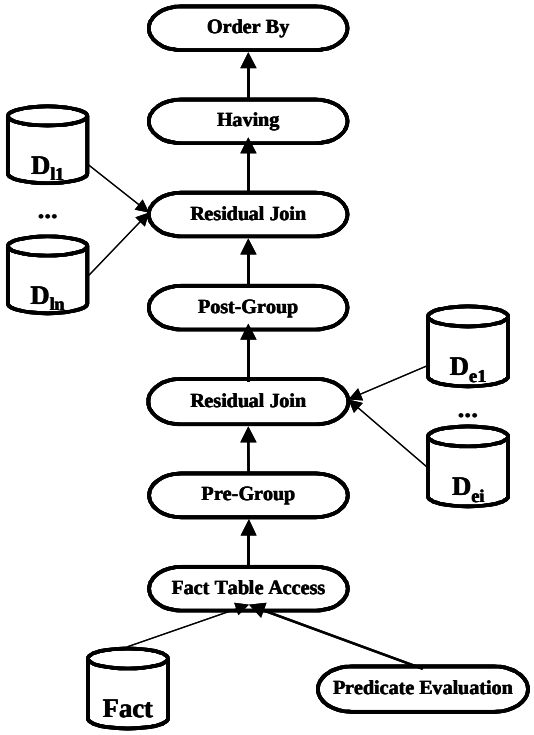


Figure 9-6: Abstract Execution Plan with Pre-Grouping

For a formal description of pre-grouping we need the following definitions.

Definition 9-10 (Hlevel):

The *Hlevel* of a hierarchy attribute h_k in a dimension table is defined to be k. The Hlevel of a feature attribute f of a dimension is k, if f is known to be functionally dependent on h_k, 1 otherwise. □

Definition 9-11 (Grouping Order, GO):

Let $g_1, \ldots, g_k$ be the set of grouping attributes of the `GROUP BY` clause which belong to dimension D_i. For dimension D_i, the *grouping order* $GO(D_i)$ is defined to be the minimum $Hlevel(g_i)$ for $1 \leq i \leq k$.

□

Note that the grouping attributes can be any attributes of dimension D_i.

Definition 9-12 (Aggregation Order, AO):

Let $a_1, \ldots, a_k$ be the set of aggregation attributes in the `SELECT` or `HAVING` clause that belong to dimension D_i. The *aggregation order* $AO(D_i)$ for D_i is defined to be the minimum $Hlevel(a_i)$ for $1 \leq i \leq k$. □

Definition 9-13 (Dimension Order, DO):

The dimension order $DO(D_i)$ for D_i is defined to be the minimum among $AO(D_i)$ and $GO(D_i)$. If $AO(D_i)$ and $GO(D_i)$ are not defined then $DO(D_i)=\infty$. □

Example 9-2 (Dimension Order):

For Example 9-1, we have the following Orders:

`GO(LOCATION) = 2,`

$$AO(LOCATION) = \begin{cases} 2 & \text{if population is functional dependent on City} \\ 1 & \textit{otherwise} \end{cases}$$

`DO(LOCATION) = MIN(GO, AO)`

`GO(DATE) = DO(DATE) = 2` (AO(DATE) is not defined)

`GO(PRODUCT) = DO(PRODUCT) = 3` (AO(PRODUCT) is not defined)

Pre-grouping reduces the number of fact records which are subject to the residual join significantly depending on *DO* (the higher the order, i.e., a shorter hsk prefix, the more pre-grouping reduces the groups), since groups of fact records are combined by aggregation functions on measure attributes to one record per group.

The following residual joins fetch the actual values of the grouping attributes. If the grouping is not exact, the groups of the pre-grouping phase may be condensed further by the *Post-Group* operator (Figure 9-6).

Post-grouping takes place on the (joined) dimension attributes and on the remaining dimensions occurring in the `SELECT`, `HAVING` or `GROUPING` clause.

All dimensions that are not yet joined and are necessary in order to evaluate the `SELECT` or `HAVGING` clause, are joined after the final grouping in a residual join operation.

Definition 9-14 (exact grouping):

Let $G = \{g_1, g_2, \ldots, g_k\}$ be a set of grouping attributes of the `GROUP BY` clause for dimension D_i. We say that G is exact for D_i *(exact grouping)*, if all dimension attributes of D_i occurring in G form a hierarchical prefix $h_t, h_{t-1}, \ldots, h_k$, where h_t is the most aggregated level of the hierarchy. □

For the grouping attributes of dimension D_i that fulfill the exact grouping criterium, i.e., $exactGrouping(D_i) = TRUE$, we can delay the residual join with D_i after the post-grouping phase. If none of the hierarchy attributes occur in the `SELECT` or `HAVING` clause, we even can omit the residual join.

If we have exact grouping for all dimensions of the grouping attributes, the post-grouping step can be omitted.

For example, grouping on *country* and *region* is an exact grouping, grouping on *region* only is not an exact grouping. The same holds for grouping on *population*.

9.4.3 Cost Estimation

Pre-grouping is not superior in all cases. Thus we need rules to decide in which cases an optimizer should generate a plan with pre-grouping and in which not.

Consider a query where the result is grouped w.r.t. several dimensions. The result is a number of groups that is almost the same as the fact table result tuples. In this case, two grouping operations, one for pre-grouping and the second for post-grouping are applied, while the tuples for the residual join are not reduced considerably. The additional grouping operation can be more expensive than the savings by cardinality reduction of the pre-grouping.

Since statistics for multidimensional data are limited, we only consider the data distribution for a single dimension. A simple heuristic is the following:

Compute the upper bound of the number of groups by the combination of the grouping attributes of the query. If this number of groups is smaller than the number of estimated fact table result tuples, then apply pre-grouping. Generally speaking, the optimizer has to decide whether the reduction of join attributes is more beneficial than an additional grouping operation.

9.4.4 Algorithm for Execution Plan with Pre-Grouping

In this section we present an algorithm that builds an optimized AEP (OAEP) with pre-grouping.

The OAEP (Algorithm 9-1) contains the operators *Fact Table Access*, *ResidualJoin*, *Having*, and *OrderBy* (as in the standard AEP) and *Pre Group* and *Post Group* as new operators (Figure 9-6).

Algorithm 9-1 (generating the OAEP):

```
OAEP = Fact_Table_Access(LOCPRED1(D1), …, LOCPREDn(Dn), MPRED(FT.m1), …, MPRED(FT.mk))
GD = groupingDimensions()
NormalAgg = { FT.mk | FT.mk ∈ AGG_SELECT }
SpecialAgg = {Di.hk | Di.hk ∈ AGG_SELECT} ∪ {Di.fk | Di.fk ∈ AGG_SELECT}
OAEP += PreGroup ({(GDi, GO(GDi))}, {FT.mk}, normalAgg)
FOR ALL {GDi ∈ GD| ¬exactGroup(GDi) ∧ extended(GDi)}
     OAEP += ResJoin(GDi)
hsk = HSK({ GDi ∈ GD| exactGroup(GDi) ∧ ¬extended(GDi)})
dattr = DimAtt({ GDi ∈ GD| ¬exactGroup(GDi) ∧ ¬extended(GDi)})
OAEP += PostGroup(hsk, dattr, {FT.mk}, normalAgg, SpecialAgg)
ResJoinNecessary = { GDi ∈ GD| exactGroup(GDi) ∧ ¬extended(GDi) ∧ GDi ∈ SelectClause}
FOR ALL { GDi ∈ ResJoinNecessary }
     OAEP += ResJoin(GDi)
OAEP += Having + OrderBy
```

The generation of the execution plan is done by the concatenation of operators (built by the corresponding function calls). The first operator is *Fact Table Access* specified by the local predicates of the dimension tables and the predicates on the fact table.

The function *groupingDimensions* specifies the dimensions that are needed for further processing, i.e., the dimensions necessary for the `SELECT`, `GROUP BY` and `HAVING` clauses. The dimensions are stored in $GD = \{GD_1, \ldots, GD_k\}$, if k dimensions are needed.

NormalAgg contains the fact table measure attributes, *SpecialAgg* contains the dimension attributes that are aggregated due to the `SELECT` and `HAVING` clause specification. The aggregation attributes are qualified by *AGG_SELECT*. The measure attributes are aggregated in a conventional way, whereas the dimension attributes are handled differently (see Section 10.2). The *PreGroup* operator contains all dimensions of *GD*. For each dimension, grouping is done on h-surrogate prefixes with grouping order *GO* depending on the grouping attributes (see Section 9.4.2) The measure attributes of the grouping clause are also used as grouping attributes.

The *ResidualJoin* operators are appended after the pre-grouping phase. For each dimension of *GD* that is not an exact dimension w.r.t. the *exactGroup* property of Section 10.2, a residual join is appended. In addition to these dimensions we add the residual joins for dimensions that are extended dimensions due to the grouping extension. An extended dimensions is a dimension that contains aggregation attributes but does not have grouping attributes. Such a dimension is considered also for pre-grouping.

The *PostGroup* operator uses h-surrogates, (joined) dimension attributes and measure attributes of the fact table (if occurring in the `GROUP BY` clause) as grouping attributes. The h-surrogate attributes (*hsk*) are used for the dimensions which are "exact" and not extended. For not exact groups the real attribute values (*dattr*) are used as grouping attributes in order to get the final granularity of the groups. The functions *HSK* resp. *DimAtt* return h-surrogate resp. dimension attributes. Aggregation is applied on all attributes to aggregate. The special aggregate semantic is used for dimension attributes as described in Section 10.2.3.

All dimensions that are not yet joined and needed for further processing are joined via *ResidualJoin* operators. These dimensions are "exact" and not extended and occur in the select clause[21].

Finally the *Having* and *OrderBy* operators are appended.

9.4.5 Measurements

In this section we describe measurements showing the benefit of pre-grouping.

The measurements are performed on a two processor PC Pentium II, 400 MHz, with 768 MB RAM and a SCSI hard disk running Windows 2000. All data is stored on one disk. The queries are executed with cold cache, i.e., cache effects (operating system and DBMS) are eliminated.

The data warehouse used for the measurements is the Sales DW with five dimensions organized with MHC: *Warehouse*, *Product*, *Calendar*, *Transaction,* and *Sales Payment* and 49 measures, such as sales, total etc. The tuples are very large (average size of the tuples is 349 byte) and the space overhead of h-surrogates is 12 byte, i.e., about 3% of the complete tuple. The dimensions are organized w.r.t. the hierarchies and cardinalities described in Section 10.8.1. The dimensions *Transaction* and *Sales Payment* have no hierarchy. The fact table has 8.579.458 records, i.e., 2,79 GB raw data.

The query workload consists of 880 ad hoc star queries from a real-world application. We classify the queries into three groups according to their selectivity on the fact table (i.e., number of tuples retrieved from the fact table):

- C_1 = [0.0-0.25]: 0% to 0.25% of fact table, i.e., 0 to about 21K records (502 queries)
- C_2 = [0.25-1.0]: 0.1% to 1% of fact table, i.e., 21K to 85K records (234 queries)
- C_3 = [1.0-10.0]: 1.0% to 10.0% of fact table, i.e., 85K to 858K records (144 queries)

The classification *All* is used in the measurement results and is the union of C_1, C_2, and C_3: $All = C_1 \cup C_2 \cup C_3$.

The queries vary in the following parameters:

- Dimension Predicates: different hierarchy levels
- Grouping Attributes: different grouping attributes (Table 9-2) and different number of grouping dimensions (from 0 to 3, see Table 9-3)

In Table 9-2 we show the occurrences of the hierarchy levels of the dimensions in the queries. Table 9-3 lists the number of grouping dimensions in the queries. Note that in most queries we have two or three grouping dimensions, but there are also some queries without a grouping clause.

Warehouse		**Product**		**Date**	
Country	116	**category**	314	**Year**	139
Geodept	119	**Group**	301	**Halfyear**	136
County	183			**Quarter**	212
City	170			**Month**	258

[21] We require that reference constraints exist for the foreign key relationship of the dimension attributes in the fact table to the most detailed level of the corresponding dimension tables. Otherwise the residual join would further restrict the result set, if groups exist that contain grouping values not existing in the corresponding dimension table.

Area	175				

Table 9-2: Grouping Attributes in Queries

0 Dimensions:	7
1 Dimensions:	72
2 Dimensions:	352
3 Dimensions:	449

Table 9-3: Number of Dimensions in GROUP BY

The goal of the performance evaluation was to measure three alternative execution plans:

- the abstract execution plan as described in Section 9.3 (NOPREGROUP),
- the early grouping without hierarchical surrogates as explained in Section 9.4.2.1 (EARLYGROUP) and
- the hierarchical pre-grouping as described in Section 9.4.2.2 (PREGROUP).

Since the time for the grouping and residual join phases covers a large part of the complete query execution time (more than 50%), an optimization of this phase reduces query execution time significantly. As join strategy, we use a nested loop join. Table 9-4 shows the average time that this third phase consumes compared to the complete query execution in the case for NOPREGROUP, EARLYGROUP, and PREGROUP for the Transbase® implementation.

For NOPREGROUP the phase is by far the longest. Both optimizations, i.e., EARLYGROUP and PREGROUP, reduce the time for grouping and residual join. The phase requires more time of the queries if the fact table result set is larger.

	All	**C_1**	**C_2**	**C_3**
NOPREGROUP	58%	46%	72%	79%
EARLYGROUP	47%	39%	57%	61%
PREGROUP	30%	22%	38%	43%

Table 9-4: Average Time of 3rd Query Processing Phase

	NOPREGROUP/EARLYGROUP				**NOPREGROUP/PREGROUP**			
	All	**C_1**	**C_2**	**C_3**	**All**	**C_1**	**C_2**	**C_3**
MIN	1,0	1,0	1,1	1,3	3,6	3,6	21,3	46,0
1. Quartile	1,5	1,4	1,7	1,8	245,8	135,1	911,3	816,2
MEDIAN	2,6	2,1	4,9	3,9	1.139,5	531,6	2.270,4	5.938,9
3. Quartile	14,1	7,2	29,1	32,5	4.708,0	1.905,6	9.747,5	25.409,6
MAX	530.931,0	1.210,6	78.384,0	53.0931,0	593.280,0	19.340,0	78.384,0	593.280,0

Table 9-5: Comparison of the Grouping Cardinality

In, Table 9-5 we show the improvement of pre-grouping w.r.t. the grouping cardinality. NOPREGROUP/EARLYGROUP contains the reduction of numbers of groups of EARLYGROUP compared to NOPREGROUP. A value of 1,0 means that there is no reduction, a value of 2 means that the resulting number of groups for EARLYGROUP is 50% of NOPREGROUP etc.

The improvement of EARLYGROUP is low compared to NOPREGROUP. 50% of all queries have an improvement between 1,5 and 14,1, where the median is 2,6. For the PREGROUP case, the improvement is 245,8 to 4.708,0 with a median of 1.139,5.

This explains the advantage of hierarchical grouping and thus the speedup of the query execution times (Table 9-6).

	EARLYGROUP/ NOPREGROUP				PREGROUP/ NOPREGROUP				PREGROUP/ EARLYGROUP			
	All	**C_1**	**C_2**	**C_3**	**All**	**C_1**	**C_2**	**C_3**	**All**	**C_1**	**C_2**	**C_3**
MIN	0,3	0,3	0,6	0,6	0,3	0,3	0,8	0,6	0,4	0,4	0,4	0,4
1. Quartile	0,9	0,9	1,0	0,9	3,0	2,4	3,9	4,6	1,0	1,2	0,9	0,8
MEDIAN	1,2	1,1	1,8	1,7	4,4	3,6	5,8	6,6	2,4	2,3	2,7	3,9
3. Quartile	3,4	2,2	4,5	5,9	6,5	5,2	7,2	7,8	5,8	4,9	7,0	7,4
MAX	15,2	8,8	15,2	13,2	25,5	14,3	25,5	12,6	35,1	19,7	35,1	14,9

Table 9-6: Speedup of EARLYGROUP and PREGROUP compared to NOPREGROUP

Table 9-6 shows the comparison of the complete user queries against the warehouse described above. The table presents the speedup, i.e., *EARLYGROUP/NOPREGROUP* means the speedup of EARLYGROUP compared to NOPREGROUP etc. The column *All* contains all queries, the column C_1 contains the results for queries of class C_1 etc.

As one can see, the speedup of EARLYGROUP compared to the standard execution plan (NOPREGROUP) is between 1,1 and 1,7 for the median. The speedup of PREGROUP compared to EARLYGROUP is again between 2,3 and 3,9. This leads to a speedup of PREGROUP compared to NOPREGROUP from 3,6 to 6,6 for the median. The speedup depends on the query classes. Generally speaking, the speedup is the higher the more tuples belong to the fact table result set, i.e., for query classes C_3, because the number of join operations can be reduced significantly.

The speedup of PREGROUP compared to NOPREGROUP of 50% of all queries lies between 3,0 and 6,5 (the range from first to third quartile). The speedup of PREGROUP compared to EARLYGROUP of 50% of all queries is between 1,0 and 5,8.

The maximum speedup of PREGROUP/EARLYGROUP of 35,1 comes from the fact that in this query EARLYGROUP (14,8 seconds) is slower than NOPRGROUP (10,7 seconds) and PREGROUP (0.4 seconds) is much faster than both. Thus, the speedup of PREGROUP compared to EARLYGROUP is larger than compared to NOPREGROUP.

Considering the overall query execution times with optimized fact table access and pre-grouping optimization, the queries do not take longer than 70 seconds (276 seconds for NOPREGROUP). In the average for query class C_3 they take about 16 seconds (56 seconds for NOPREGROUP). Thus, even queries covering a large fraction of the fact table are executed within acceptable time frames on a machine with comparably low performance characteristics.

9.5 Secondary Dimensions

The discussion so far is based on the assumption that the tuples retrieved from the fact table (*Predicate Evaluation* and *Fact Table Access*) is exact w.r.t. the predicates of the query.

However, in some cases, the optimizer could decide to retrieve a super set and reduce the tuples via a post-filtering operation. For example, a superset can be the consequence, if an index on a dimension

key is not used or does not exist. Usually a residual join is necessary to evaluate the final fact table result tuples (the residual join acts as post-filtering operation).

9.5.1 Description of Secondary Dimensions

A *secondary dimension* is a dimension that is not used to restrict the tuples in the *Fact Table Access* operator in the AEP. The restriction on such a dimension is evaluated after the *Fact Table Access* operator, usually by post-filtering or residual join methods. There are several reasons, why a dimension is used for delayed restriction evaluation. It depends on the optimizer decisions, which access plans are generated for the fact table:

- The dimension is not indexed.
- The dimension is not an index attribute of the primary clustering index.
- An alternative access method (not via index) to the fact table is cheaper w.r.t. the costs calculated by the optimizer

We call a dimension that is not a secondary dimension a *primary dimension*.

For a physical schema as proposed in Section 6.2.2 and used for the generation of AEP as described in Section 9.3, we assume that the fact table is organized by a primary clustering multidimensional index with the h-surrogates as index attributes. Typically, we use compound surrogates as h-surrogate implementation and the UB-Tree as multidimensional index. The AEP evaluates the intervals resulting from the restrictions on the dimensions and builds one or more multidimensional query boxes as restrictions on the UB-Tree.

We assume that secondary dimensions are also organized by h-surrogates that occur in the dimension tables and in the fact table analogously to primary dimensions. The only difference is that the h-surrogate in the fact table (reference surrogate in the implementation of Transbase®) is not part of the clustering multidimensional index.

A secondary dimension typically is a dimension of the DW schema that is not used as index attribute of the multidimensional index, because the index cannot handle a large number of dimensions or the dimension is not regarded to be as important as other dimensions. Thus, secondary dimensions are relevant in practice, because real data warehouses often have more than ten dimensions. Not all of them can be used as index attributes of the multidimensional index.

9.5.2 Star Query Processing

The standard way of processing star queries with secondary dimensions is similar to the standard abstract execution plan of Section 9.3. The Predicate Evaluation of the dimensions for the fact table access contains only primary dimensions D^P_i (see Figure 9-7). The Fact Table Access operator delivers a number of fact table result tuples R. R is the result of the restrictions on the primary dimensions D^P_i. Thus R is a super set of the final fact table result tuples with the secondary dimensions D^S_k considered. R is joined with the secondary dimensions D^S_k. This residual join evaluates the restriction on the secondary dimensions and removes tuples that do not fulfill these predicates. The remaining steps in the execution plan are equal to the standard execution plan and are described in Section 9.3.

The residual join with secondary dimensions is performed before the residual join with the primary dimensions, because the reduction of the fact table result tuples result in less join operations of the *Primary Dimension Residual Join*.

Since the residual joins are used for post-filtering, the order of the residual joins influences the query performance. If the first residual join reduces the fact table result set significantly, the next joins have less join operations. Thus, we first perform the residual join that is estimated to reduce the cardinality most etc.

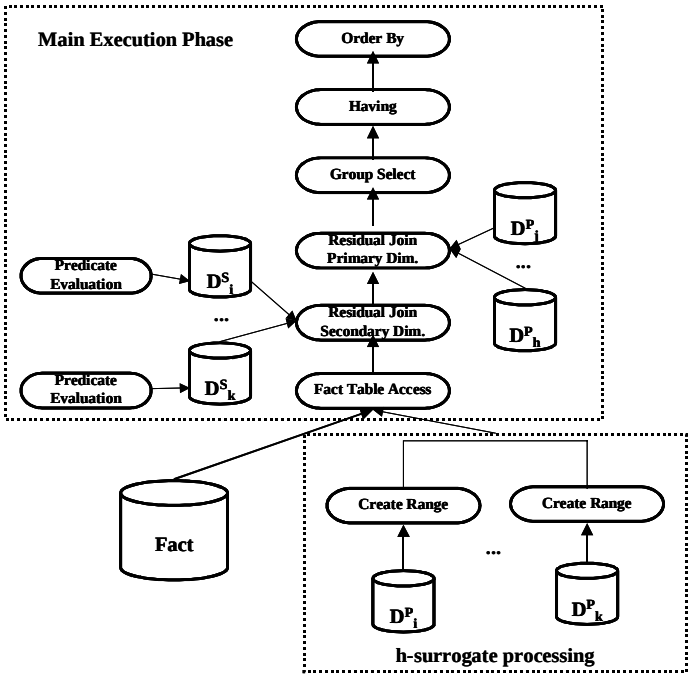


Figure 9-7: Standard Abstract Execution Plan with Secondary Dimensions

9.5.3 Pre-Grouping

9.5.3.1 Standard Pre-Grouping

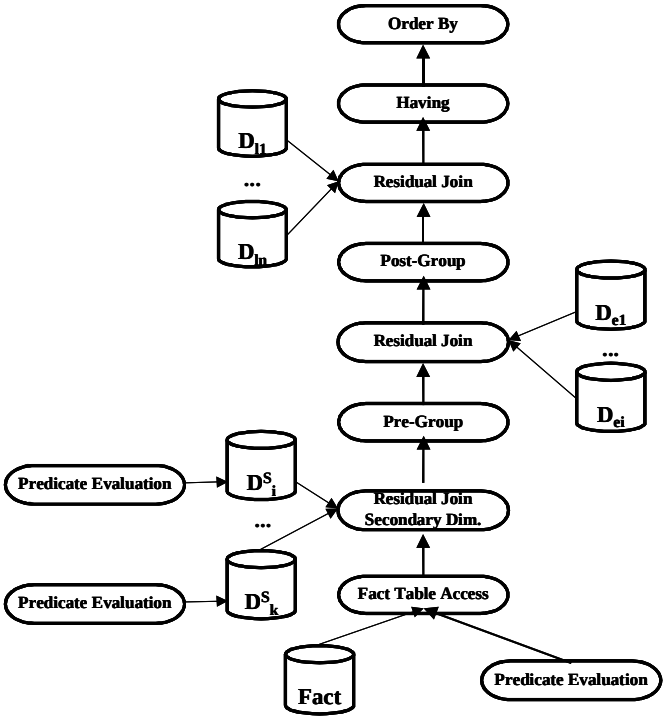


Figure 9-8: Pre-Grouping Plan with Secondary Dimensions

The pre-grouping optimization as described in Section 9.4 is adapted straight forward to secondary dimensions. The first approach is not to include secondary dimensions into pre-grouping (see Figure 9-8). The *Predicate Evaluation* and the *Fact Table Access* is done in the same way as in Section 9.5.2. The Residual Join with the secondary dimensions is performed after the *Fact Table Access*. This

residual join with secondary dimensions reduces the fact table result tuples and speeds up pre-grouping and the remaining steps of the execution plan. Pre-grouping is done on h-surrogate prefixes of the primary dimensions as described in Section 9.4.

The advantage of this pre-grouping processing is that the grouping cardinality is not increased due to additional grouping attributes of the secondary dimensions. After the residual restriction join with the secondary dimensions, the attributes of these dimensions are available and can be used for grouping and projection. For the primary dimensions, we apply pre-grouping on h-surrogates conventionally. Note that each residual join with a secondary dimension usually reduces the fact table result set correspondingly.

9.5.3.2 Pre-Grouping on Secondary Dimensions

In order to further optimize the query processing, we propose to apply pre-grouping on secondary dimensions. Instead of performing the residual join on the fact table result set we first pre-group the tuples w.r.t. h-surrogate prefixes of the primary and secondary dimensions. For the h-surrogate prefix of a secondary dimension D_i, we extend the dimension order *DO* for dimension D_i as $DO(D_i) = MIN(GO(D_i), AO(D_i), RO(D_i))$, where $RO(D_i)$ is the *restriction order* of D_i and corresponds to the minimum hierarchy level of the restricted dimension.

Definition 9-15 (Restriction Order):

A *restriction order* $RO(D_i)$ of a dimension D_i is the minimum Hlevel that is restricted in the query. □

This leads to an extension of the definition of the dimension order (see Section 9.4.2.2).

Definition 9-16 (Dimension Order, DO):

The dimension order $DO(D_i)$ for D_i is defined to be the minimum among $AO(D_i)$, $GO(D_i)$, and $RO(D_i)$. If $AO(D_i)$, $GO(D_i)$, and $RO(D_i)$ are not defined then $DO(D_i)=\infty$. □

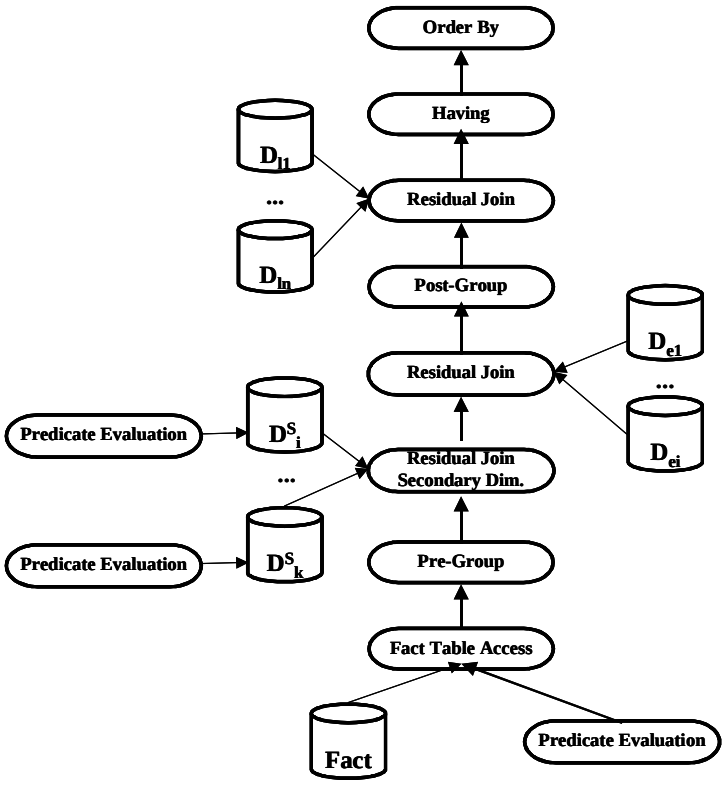


Figure 9-9: Pre-Grouing on Secondary Dimensions

The tuples of the fact table result set are pre-grouped w.r.t. h-surrogate prefixes of all participating dimensions. The subsequent residual join filters the groups according to the restrictions on the secondary dimensions (see Figure 9-9).

Depending on the grouping attributes and on the data, the cardinality of the tuples can be reduced by the pre-grouping step significantly. If h-surrogates do not exist, we have to pre-group on the most detailed hierarchy level (the primary key) of the dimension. This can lead to severe disadvantages, because the cardinality cannot be reduced in the same way as with pre-grouping on "higher" hierarchy levels. Thus, the decision which pre-grouping method to use is an important issue for the optimizer.

9.5.3.3 Step-wise Pre-Grouping

In this section we propose a method to deal with secondary dimensions without h-surrogates. This method also can be applied, if the grouping attributes of a dimension are feature attributes. Recall that grouping on feature attributes that are not known to be functionally dependent on hierarchy attributes, results in a pre-grouping on the complete h-surrogate instead on a prefix.

We therefore introduce *step-wise pre-grouping*, i.e., an iterative pre-grouping – residual join sequence, one for each dimension, where we have to pre-group on the complete h-surrogate (or h_1 correspondingly). The corresponding abstract execution plan is shown in Figure 9-10. We first perform the residual join with the secondary dimension that reduces most the cardinality of the fact table result set (dimension D^S_k in the plan). Then we pre-group on all dimensions. This means that we use the h-surrogate prefixes of all concerned dimensions in order to perform pre-grouping on the reduced fact table result set. Then we join with the next secondary dimension and pre-group again. If no h-surrogates exist on a secondary dimension, we use the dimension key attribute of the fact table for pre-grouping.

If all secondary dimensions are joined, we perform the residual join with primary dimensions D^P_{eq}, …, D^P_{eh}. The joined attribute allows pre-grouping on a higher hierarchy level and therefore a more effective pre-grouping. The same is done with all other primary dimensions with a pre-grouping h-surrogate prefix that corresponds to the complete h-surrogate. The remaining execution plan is the same as described in Section 9.4.2.

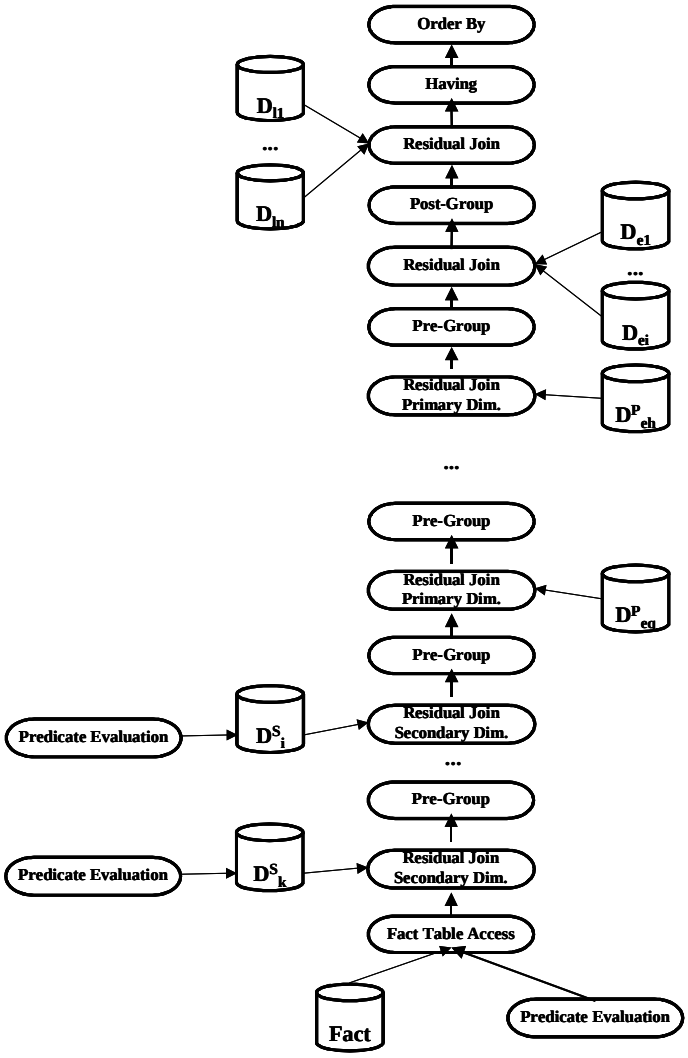


Figure 9-10: Step-Wise Pre-Grouping with Primary and Secondary Dimensions

Note that a pre-grouping operation requires effort that depends on the number of the input tuples. Depending on the implementation of the grouping operator, the costs can vary. It therefore is important to estimate the cost of additional grouping operators compared to the cost of residual joins on more tuples. In Section 10.9 we show some performance measurements with secondary dimensions.

There can be also implemented a multi-way residual join, i.e., an optimization to join more than one secondary (or primary) dimensions. This depends on the implementation of the DBMS.

10 Implementation Issues

In Section 9 we discussed the abstract query processing plan. We described concepts of the overall query processing. This section gives more implementation hints, i.e., how are particular operators implemented. We show some algorithms and operator trees of the Transbase® implementation.

We first describe how to recognize an MHC schema, i.e., the dimensions and hierarchies. The knowledge about the schema is necessary for query processing. In a DBMS there can be any physical schema, but only some special schemata are proper for star query processing.

We further discuss some advanced algorithms how to handle complex query requirements, such as complex expressions, multiple query boxes, join orders etc.

A query first is transformed to a "raw operator tree" without any optimization. Several optimization methods are applied, in order to optimize the operator tree and minimize query execution time. The optimization steps depend on pre-defined rules (in Transbase®).

The entry point where star query optimization takes place is the join optimization. This is the third optimization step in the optimization procedure. The first optimization step is the so called *prodnorm* optimization that combines inner join sequences to `TIMES` cluster. `TIMES` cluster enable efficient join optimization. The second optimization step handles local restrictions, i.e., restrictions on tables each of which is represented by a `RESTR` node. Restrictions of one table are coalesced to one `RESTR` node by "ANDing" the predicates. Transitive restrictions are simplified and some further optimization for special cases is done.

The third optimization step is the *join optimization*. This step was extended by the MHC star query optimization. The following sections describe the star query optimization concerning the join optimization, if not mentioned otherwise.

10.1 Recognizing the Schema and Building the Operator Tree

A star join query has a center fact table and surrounding dimension tables. We require a correctly specified physical star or snowflake schema with the necessary reference constraints, compound and reference surrogate attributes.

To check for a MHC star join scenario, internal structures are built and maintained. After analyzing the operator tree, we get the star join decomposed into edges and vertices, where edges are the join conditions and vertices are the corresponding relations.

Basically, we first look for the fact table, i.e., the center of the star join. The second step is to check the correctness of join conditions to the dimensions and the specification of hierarchies and surrogates. Now the dimensions can be identified and partitioned into dimension clusters, where every cluster contains all relations belonging to the dimension. Every dimension cluster is handled individually according to the restrictions. Further analyze steps modify the join conditions and dimension relations by adding or removing joins and relations according to the predicates of the query. The next step categorizes the dimension with respect to a query class (*HPP, HNPP, NH*) to build the interval generation operator trees. Finally, a new MHC operator tree is built by generating a new operator tree consisting of the operator trees of the dimensions and the fact table. The grouping and residual join optimizations build the final operator tree.

As example, we use the following query for the Sales DW as described in Section 10.8 for further explanations:

```
SELECT
  country_str, dept_str, cat_str, grp_str, year, quarter, month,
  SUM(val), SUM(qty)
FROM
  customer c, customer_country c_country, customer_dept c_dept,
  product p, product_category p_cat, product_group p_group, date d,
  fact f
WHERE
  f.custkey = c.customer AND
  f.prodkey = p.item_key AND
  f.datekey = d.day AND
  c.country = c_country.country AND
  c.dept = c_dept.dept AND
  p.category = p_cat.category AND
  p.grp = p_group.grp AND
  c_country.country_str = 'GERMANY' AND
  c_dept.dept_str = 'SOUTH' AND
  p_category.cat_str = 'TV' AND
  d.month = '10/2002' AND
  d.quarter = '4q2002' AND
  d.year = '2002'
GROUP BY
  country_str, dept_str, cat_str, grp_str, year, quarter, month
```

SQL Statement 10-1: Sample Query

For the *customer* dimension, we use a snowflake dimension (field normalized dimension) with the leaf dimension table *customer* and the higher dimension tables *customer_dept* and *customer_country*. The *product* dimension is modeled in the same way with *product* as leaf dimension table and *product_group* and *product_category* as higher dimension tables.

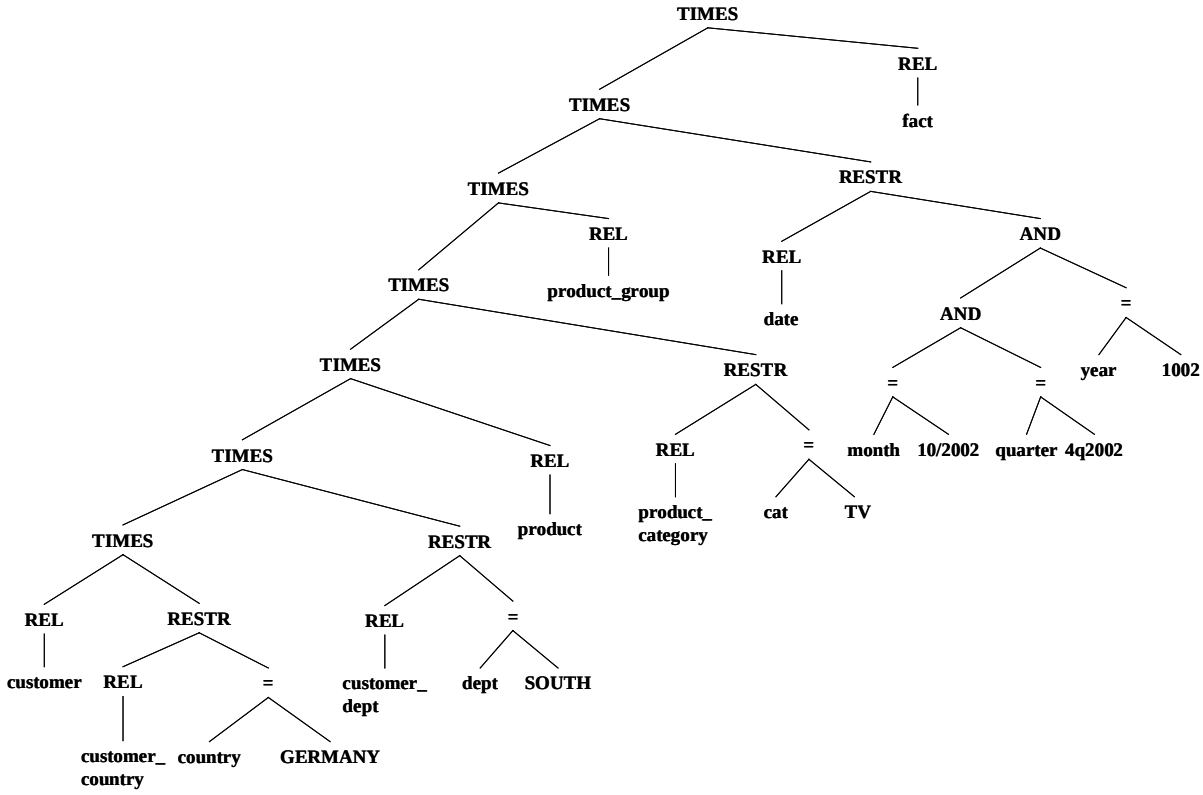


Figure 10-1: TIMES Cluster of the initial Operator Tree

The query results into a (non-optimized) operator tree generated by the first two phases of the optimization procedure (*prodnorm* and *local restrictions*). The complete operator tree in Transbase® notation is shown in Figure 10-2. The main part, i.e., TIMES cluster, is shown in Figure 10-1.

```
 (N0:sel  {   9 "country_str" "dept_str" "cat_str" "grp_str" "year"
"quarter" "month " "column_7" "column_8"  }
   (N1:proj
      (N2:group  { [ 11 14 21 23 24 28 29 ] sum[36] sum[37] }
         (N3:sort  {  +11 +14 +21 +23 +24 +28 +29 }
            (N4:restr
               (N5:times  {    }
                  (N6:times  {    }
                     (N7:times  {    }
                        (N8:times  {    }
                           (N9:times  {    }
                              (N10:times  {    }
                                   (N11:times  {    }
                                      (N12:rel  { "customer"  })
                                    (N13:restr
                                       (N14:rel  { "customer_country"  })
                                       (N15:eq  { }
                                          (N16:attr  { N13[3] } )
                                          (N17:const{'GERMANY'char(3)} ))))
                                 (N18:restr
                                    (N19:rel  { "customer_dept"  })
                                    (N20:eq  { }
                                       (N21:attr  { N18[3] } )
                                       (N22:const  { 'SOUTH'  char(4)  ))))
                              (N23:rel  { "product"  }))
                           (N24:restr
                              (N25:rel  { "product_category"  })
                              (N26:eq  { }
                                 (N27:attr  { N24[2] } )
                                 (N28:const  { 'TV'  char(2) } ))))
                        (N29:rel  { "product_group"  }))
                     (N30:restr
                        (N31:rel  { "date"  })
                        (N32:and
                           (N33:and
                              (N34:eq  { }
                                 (N35:attr  { N30[6] } )
                                 (N36:const  { '10/2002'  char(7) } ))
                              (N37:eq  { }
                                 (N38:attr  { N30[5] } )
                                 (N39:const  { '4q2002'  char(6) } )))
                           (N40:eq  { }
                              (N41:attr  { N30[1] } )
                              (N42:const  { '2002'  char(4) } )))))
                  (N43:rel  { "fact" }))
               (N44:and
                  (N45:and
                     (N46:and
                        (N47:and
                           (N48:and
                              (N49:and
                                 (N50:eq  { }
                                    (N51:attr  { N4[33] } )
                                    (N52:attr  { N4[6] } ))
                                 (N53:eq  { }
                                    (N54:attr  { N4[1] } )
                                    (N55:attr  { N4[9] } )))
                              (N56:eq  { }
```

```
                        (N57:attr  { N4[2] } )
                        (N58:attr  { N4[12] } )))
                  (N59:eq  { }
                     (N60:attr  { N4[34] } )
                     (N61:attr  { N4[17] } )))
               (N62:eq  { }
                  (N63:attr  { N4[15] } )
                  (N64:attr  { N4[20] } )))
            (N65:eq  { }
               (N66:attr  { N4[16] } )
               (N67:attr  { N4[22] } )))
         (N68:eq  { }
            (N69:attr  { N4[35] } )
            (N70:attr  { N4[27] } ))))))
(N71:build
   (N72:attr  { N1[1] } )
   (N73:attr  { N1[2] } )
   (N74:attr  { N1[3] } )
   (N75:attr  { N1[4] } )
   (N76:attr  { N1[5] } )
   (N77:attr  { N1[6] } )
   (N78:attr  { N1[7] } )
   (N79:attr  { N1[9] } )
   (N80:attr  { N1[8] } ))))
```

Figure 10-2: Operator Tree of non-Join Optimized Sample Query

The operator tree is presented in Transbase® notation, where `N<nr>` denotes the node with number *nr*. These nodes can be referenced by other nodes. If a specific attribute of node N_j is referenced, we use the notation $N_j[k]$ for the k^{th} attribute. The operators are denoted by the name of the operation. We have the operator `rel`, which stands for the access to a B-tree (index or table) via a primary key (index key or primary key of the table)[22]. For example, `(N19:rel { "customer" } )` means that the table "customer" (dimension table) is accessed via the primary key, i.e., *customer*.

The operator `restr` is a restriction with two successors, usually a data source and an expression tree. The fragment

```
(N13:restr
   (N14:rel  { "customer_country"  })
   (N15:eq  { }
      (N16:attr  { N13[3] } )
      (N17:const  { 'GERMANY'  char(3) } ))))
```

is an access to the table *"customer_country"* and returns all tuples where attribue N13[3] (*country* in the sample query) is equal to *'GERMANY'*.

The `times` node represents a join operation. Between two sub-trees, usually a `times` cluster (or a single `restr` node) for the left and a single `restr` node for the right son are placed.

The `sort` node is a sort operation on some attributes denoted by the consecutive attribute positions (in this case of the flattened join of all participating tables). The `group` node represents grouping (usually via sorting) with the grouping attributes denoted as attribute positions and the aggregation functions, `sum` in the sample query. The `proj` node does the projection for the query, i.e., some attributes are removed. Finally, the `sel` node is the projection with the names of the returned columns (attributes).

[22] Note that in Transbase® every table is stored in a clustering B-Tree with the primary key as index key.

The operator tree of Figure 10-2 contains a TIMES cluster for the cartesian product of all tables that are joined (*fact, customer, customer_country, customer_dept, product, product_category, product_group* and *date*). Figure 10-1 shows this TIMES cluster in a graphical representation for the ease of understanding. All tables are joined via a two-way join with local restrictions (on the dimension tables).

In Figure 10-3, we show the overall operator tree, with the times cluster as (abstracted) sub-tree. The sequence (from top of the tree) SEL, PROJ, GROUP, SORT, RESTR (with consecutive join restrictions) represent the overall query processing plan. All join conditions occur in the join condition cluster (right son of the RESTR node). The tuples resulting from the times cluster and the join condition cluster are sorted, in order to prepare for grouping and are finally grouped w.r.t. grouping attributes of the query.

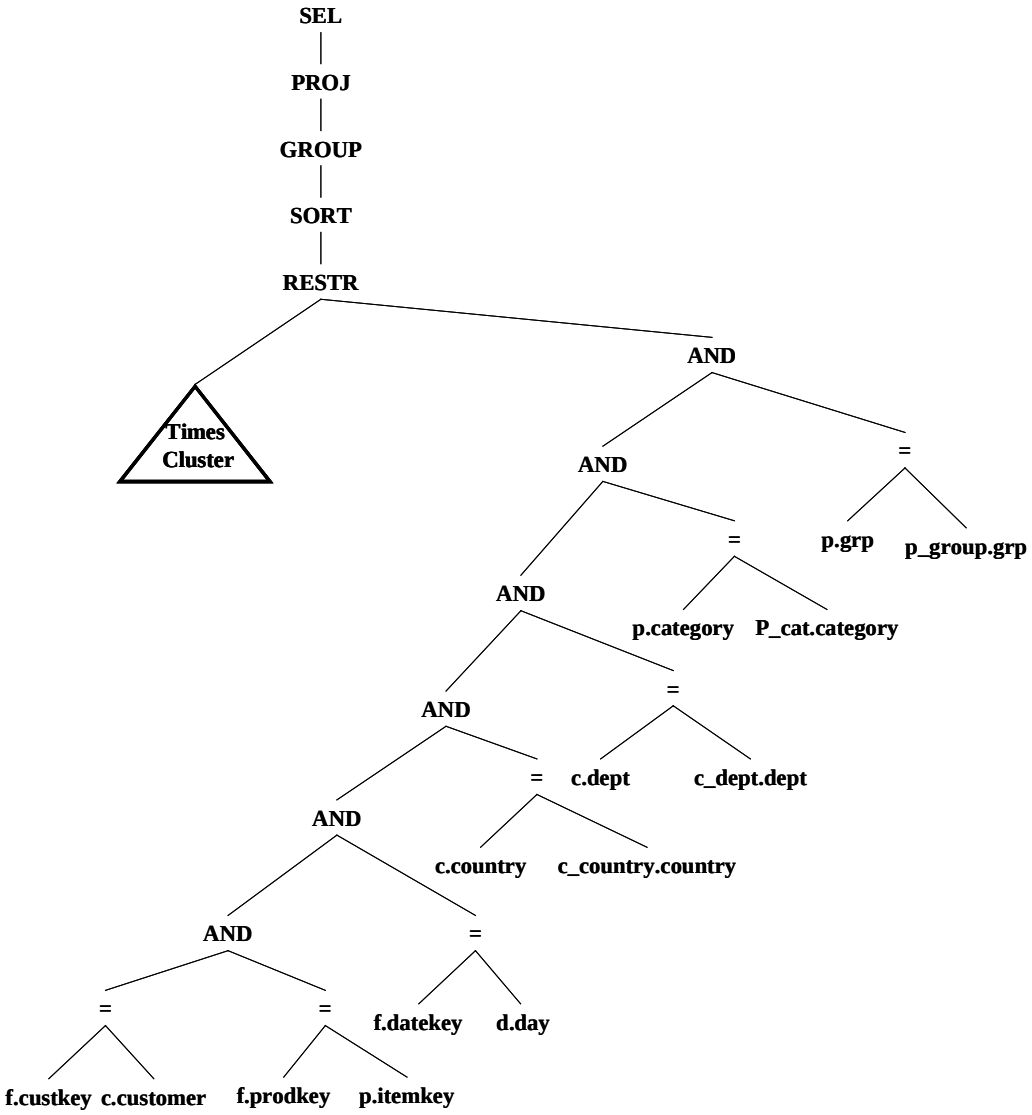


Figure 10-3: Overall Operator Tree

Note that this operator tree is not optimized, e.g., early projection is not done, special join optimization is missing etc.

For the MHC schema check, we decompose the operator tree into another data structure representation, i.e., into *join nodes* called *joininput* and *edges* for further processing. *joininput* represents the tables and nodes in the original operator tree. It contains information about

- The node in the operator tree, i.e., a pointer to the corresponding `TIMES` node.
- Catalog information about the table, e.g., attributes, indexes, surrogates etc.
- Additional information for the MHC optimization, e.g., properties like flags to denote leaf dimension tables, dimension number etc.

Each joininput represents a table in the join graph. The *edges* data structure captures information about

- the operator tree, i.e., pointer to the join predicate.
- properties of the join edge, e.g., `is_leaf_dimension_join`, properties for the marking algorithm, properties for handling the MHC optimization etc.

With the data structures operator tree, joininput, and edges, we can analyze the operator tree and generate various execution plans. *joininput* and *edges* are modified during the analyze steps, in order to store additional information.

10.1.1 Finding the Fact Table

The first step is to find the fact table within a star join. For the optimization of queries, it is crucial to recognize special patterns or dependencies, in order to apply special optimization methods. SQL-92 allows to formulate a vast number of queries that the DBMS has to support. Thus, there must be an automatism to recognize some basic query classes, such as star join queries. For each query, the check for star query is done. It is important to recognize very fast that the query does not contain a star join, in order to proceed with standard query optimization. If there is no valid fact table, we can skip further star query testing and continue with standard optimizing (in Transbase®).

The fact table *fact* as center of the star join must have the following properties:

- *fact* must be organized as UB-Tree.
- The joins with fact must be correct, i.e., for primary dimensions there must be valid reference and compound surrogates, for secondary dimensions, there must be correct reference constraints.

Thus, we go through the elements of *joininput*, in order to check whether the current table is organized as UB-Tree. Second, we test the joins for all dimensions that are joined with this table by looking at the *edges* structure, in order to decide, if each such leaf dimension table is joined correctly with the fact table (inner join via dimension key attribute). We further check, if the foreign key references and the reference and compound surrogates match with the query (see Section 6.2). Each leaf dimension table is marked. A leaf dimension table is not necessarily organized by compound surrogates and needs not have a corresponding reference surrogate in the fact table. However, there must be a valid foreign key relationship between the leaf dimension table and the fact table. If there is a table that is joined with the fact table, but does not fulfill the leaf dimension criteria, we abort the star query processing and proceed with standard optimizing.

After successfully checking and marking the leaf dimension tables, we build a join graph, in order to further check for a correct star query schema.

If more than one fact tables exist (and are joined either with one or more fact tables or via common dimension tables), we use one fact table as primary fact table and handle the remaining fact tables either as secondary dimensions (if they are joined with the primary fact table) or as special higher dimension table (if joined via common dimension tables). Section 10.7 discusses the order of fact tables and further methods to deal with multiple fact tables.

10.1.2 Isolating the Dimensions

For the remaining steps, the dimensions are handled individually. Thus, we first partition the edges and vertices according to the dimensions. The leaf dimension tables are handled in a special way and are positioned before the dimension tables. Every dimension table belongs to one dimension.

Otherwise there are cross references between dimensions. We mark the edges with a dimension id *dimid*.

For better handling of the edges and vertices, we sort the array storing the vertices according to *dimid*, where the vertex of the leaf dimension table is the first vertex in the dimension cluster. Also the edges are sorted in this way with the edge having an LDT as vertex ranked first in the edges cluster.

After partitioning the dimensions, we check the correctness of the joins between dimensions. A join between two dimension tables D_i^k and D_i^j of dimension D_i, where D_i^k is the higher dimension table, is correct, if for $D_i^k.h_m = D_i^j.h_m$ the attribute $D_i^k.h_m$ is "unique". A *higher* dimension table is a dimension table in a snowflake schema that is more on the edge of the snowflake than the other dimension table (see Section 2.4). A *unique* attribute means that $D_i^k.h_m$ is either the primary key of D_i^k or has a unique index. We have to ensure that no duplicates occur for the dimension predicate evaluation, because these duplicates are propagated to the leaf dimension table. It is not clear, what are the consequences for the interval generation and evaluation of these intervals on the fact table. There might be tuples with and without duplicates within one interval. This cannot be handled correctly, because we transform the equi-join between the fact and leaf dimension tables to a semi-join.

Thus, we check for unique join conditions beginning with the joins between leaf dimension table and the next higher dimension table. Then all further dimension tables are examined. Each table with a valid join predicate is marked with the dimension number. If all tables of one dimension can be marked in this way, the dimension join is correct.

During this procedure, all edges participating in unique joins are marked and represent a *spanning tree* for the star join, i.e., they form the necessary join conditions. Notice that there could be additional joins that are redundant or not necessary. Such joins are not considered, when the spanning tree has been created. Not necessary joins are joins with a higher dimension table that itself has no local predicate and is joined with other higher dimension tables that also have no local predicates. However, attributes of such dimension tables can occur in the `GROUP BY` or `SELECT` clause. In this case, we consider such a dimension table later in the residual join graph. Unnecessary dimensions often occur in queries that are generated w.r.t. query templates. For example, a template can be provided that contains all join predicates and the application has to fill in the dimension predicates, grouping and select attributes.

If the correctness of the joins are verified, we know that we have an MHC star join on a valid MHC schema. We now can proceed with the interval generation on the dimension tables and the optimization of the fact table processing.

10.1.3 Dimension Predicate Collection and Fact Table Predicate Mapping

The predicates of the dimension tables are collected in a data structure, in order to recognize query classes. Thus, we traverse the operator tree and check for each `RESTR` node, whether it belongs to a dimension table. The predicates are connected to the corresponding dimension entry in the *joininput* structure.

Before evaluating the predicates on the dimension tables, we check whether there are restrictions on the dimension attributes of the fact table as well. These dimension attributes on the fact table are not supported by an index, because the index attributes of the multidimensional index are the reference surrogates. However, such predicates can be mapped to predicates on the leaf dimension tables and evaluated efficiently by the standard dimension predicate evaluation method (see Section 8.3.1).

In the case of a local dimension key restriction on the fact table, we map this restriction to a restriction on the key of the corresponding leaf dimension table and proceed with the MHC optimization with this modified predicate. If there is not yet a join to this dimension table, we add a join condition according to the reference constraint of the dimension key in the fact table.

10.1.4 Finding Predicate Class for each Dimension

It is crucial for the optimization to get the predicate class for each dimension, i.e., whether the dimension is of class *HPP*, *HNPP* or *NH* (see Section 9.2.3 for further details). For this purpose, we look at the local restrictions on the dimension tables and on restrictions via join conditions.

A hierarchy consists of t levels $h_t, h_{t-1}, \ldots, h_1$, where h_t is the top level and h_1 is the leaf level. The predicate class depends on the sequence of the restricted hierarchy levels. If a hierarchy level is restricted locally in a dimension table or it is restricted via a join condition, the field denoting the hierarchy levels for the corresponding dimension is set to `TRUE`, `h_lev_restricted[i] = TRUE`, i.e., it is marked as *restricted*. A join condition on $D_i^k.h_j = D_i^h.h_j$ is equivalent to a restriction `WHERE` D_i^k`.`h_j `in select` h_j `from` D_i^h and thus is also considered for the query class. Also snowflake dimensions are recognized in this way.

If a feature attribute (i.e., a non-hierarchical attribute) is restricted, the class is *NH* and the evaluation of the query classes is finished. However, if a restricted feature attribute is correlated to a hierarchy level, i.e., if a feature attribute is located in a higher dimension table, we also have a hierarchical restriction, because the corresponding hierarchy level is restricted implicitly.

Depending on the values of `h_lev_restricted[]`, the predicate class of the dimension is *HPP* ($h_t, h_{t-1}, \ldots, h_k$ are set to *TRUE*) or *HNPP* ($h_{j1}, h_{j2}, \ldots, h_{jk}$ are set to *TRUE*, where $1 \leq j_1 < j_2 < \ldots < j_k \leq t$). h_t is the most aggregated level of the hierarchy. The *hierarchy degree hdg* is the lowest restricted hierarchy level, i.e., $hdg_{HPP} = k$ and $hdg_{HNPP} = j_1$.

For *NH* predicate class, the hierarchy degree is 1, because a feature attribute formally depends on the key of the dimension table(h_1).

In Section 10.1.8 we show some examples of the query classes and hierarchy degrees.

10.1.5 Building Dimension Join Operator Tree

With the information collected so far, we are able to build the operator trees for the interval generation of the dimension tables. The operator trees of the dimensions are combined, in order to establish the fact table access via a set of multidimensional intervals. We first describe the operator trees for the dimension tables and then show the overall operator tree with the fact table access (Section 10.1.6).

The operator tree for a dimension table depends on the dimension predicates. Various access methods are possible for the query classes. For a join with a secondary dimension, a different operator tree is built, because the access to the multidimensional index is not supported. We then build an operator tree that returns the values for the equi-join instead of a number of intervals.

The basic dimension operator trees are built without considering the query class, since only join predicates and local dimension predicates occur. The information about the query class is stored as a parameter to the operator tree of the dimension. A later optimization step (the *index access optimization*) cares about the interval generation and optimizes the index access (see Section 10.1.8).

We divide the description about building the dimension operator tree into *snowflake dimensions* and *star dimensions*. A snowflake dimension consists of a leaf dimension table and one or more higher dimension tables. A star dimension consists of the leaf dimension table only.

10.1.5.1 Snowflake Dimension

If we have a snowflake schema with higher dimension tables in the join graph of the query (after removing redundant joins), the join operator tree of the higher dimension tables is constructed. The schema used in the illustrations is the path normalized Sales DW schema as described in Section 10.8. No join optimization is done at this step, i.e., there is a `TIMES` cluster resulting eventually in a

cartesian product (depending on the join restrictions and on the schema). In a later optimization step, the standard join optimization is applied to this TIMES cluster.

The TIMES cluster is combined with a RESTR node, in order to specify the join conditions. Figure 10-4 shows an example for such a join operator tree of the customer dimension from the sample query of SQL Statement 10-1. The RESTR tree is used as driver table for a nested loop join into the leaf dimension table. The join with the leaf dimension table is constructed in the same way, i.e., a TIMES node with a RESTR node for the join condition between the leaf dimension table and the join cluster.

Above this RESTR node, we add a conceptual COMPSURR node. The COMPSURR (compute surrogates) node is used in a later optimization step, in order to generate the interval generation for that dimension.

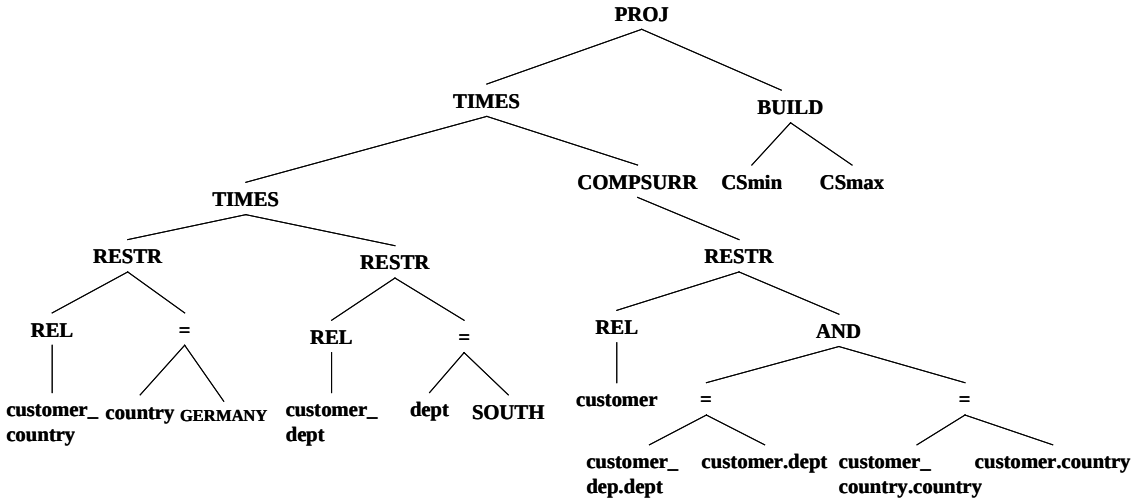


Figure 10-4: Operator Tree for Customer Dimension

For the interval generation, we add a PROJ (projection) node at the top of the operator tree. The BUILD node specifies the projection attributes. We restrict the projection to the compound surrogates (minimum and maximum) that specifies the interval for the *customer* dimension.

The operator tree for the *product* dimension is built analogously and is not shown explicitly here.

10.1.5.2 Star Dimension

For a dimension without higher dimension tables, i.e., the dimension consists of the leaf dimension table, there is no dimension join. The operator tree is simpler and contains only dimension predicates on the leaf dimension table. The preliminary COMPSURR node is again on top of the operator tree.

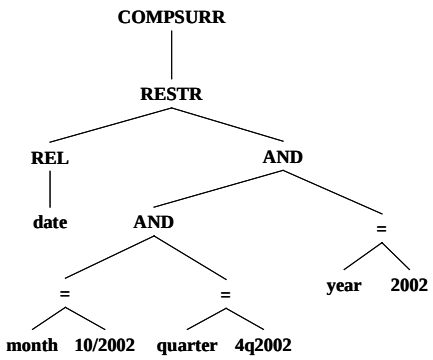


Figure 10-5: Operator Tree for Date Dimension

The *date* dimension of the sample query is a dimension that consists of the leaf dimension table *date*. The resulting operator tree is shown in Figure 10-5.

10.1.5.3 Secondary Dimension Join

The join operator tree of a secondary dimension is a standard join operator tree, if the secondary dimension is a *secondary snowflake dimension*. A secondary snowflake dimension is a secondary dimension with a leaf dimension table and one or more higher dimension tables comparable to a primary snowflake dimension. If the secondary dimension consists of the leaf dimension table only, the operator tree contains the restrictions without additional join nodes.

The predicates of the secondary dimension are evaluated and used for the residual join, in order to post-filter the fact table result tuples (see Section 10.5). If a secondary dimension is also organized with hierarchies, i.e., the leaf dimension table of secondary dimension also has compound surrogates, these information can be used for pre-grouping the secondary dimension (see Section 9.5.3).

10.1.6 Combining Fact Operator Tree with Dimension Operator Trees

The operator trees of the dimensions are combined to a complete operator tree of the query. We show the corresponding operator tree for the three dimensions *country*, *product* and *date* for the sample query in Figure 10-6. Note that this operator tree is not complete w.r.t. the interval generation, since the COMPSURR nodes have to be resolved (see Section 10.1.8). In Appendix C in Figure 14-24, we show the operator tree in Transbase® notation.

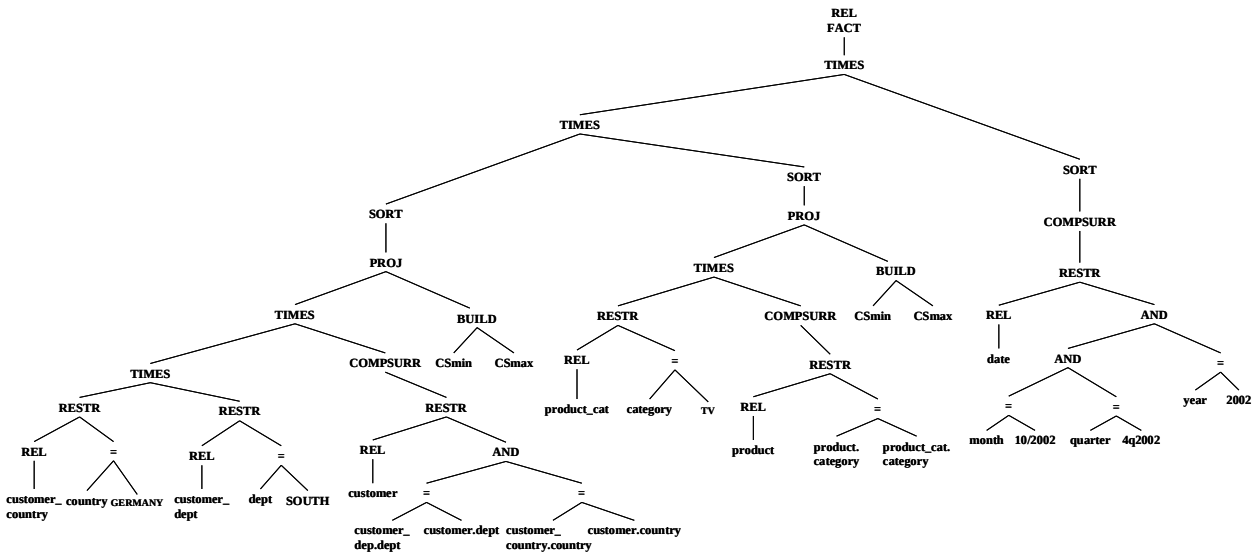


Figure 10-6: Combined Operator Tree

This operator tree does not yet contain the grouping and residual join operations, since it is only the concatenation of the dimension operator trees for the dimension interval generation.

For the combination of the dimension operator trees, each operator tree is extended by a SORT node. These nodes are necessary, because the dimension evaluation results in a set of intervals. The intervals of one dimension are disjoint and must be sorted for the use of the extended range query algorithm as described in Section 10.3. The range query algorithm requires a sorted and disjoint stream of intervals. Multiple intervals of one dimension cause multiple multidimensional query boxes. The number of query boxes can be very large and special algorithms are necessary, in order to handle large number of query boxes efficiently (see Section 10.3 for more details).

10.1.7 Building the Grouping Operators and Residual Joins

The basic operator tree already contains a grouping node. With MHC, however, grouping is handled in a special way. We construct a new operator tree with pre-grouping, if possible. For a detailed discussion about the concept of pre-grouping see Section 9.4.

Grouping cannot be discussed isolated from the residual join handling, because these two optimization steps influence each other.

After the combination of the fact table access operator tree with the operator trees of the dimension, we have an operator tree as shown in Figure 10-6 with the fact table access and evaluation of the dimension tables. The grouping and residual join operators are still missing. We generate these operations w.r.t. the following order (from bottom to top):

- Residual Join of Secondary Dimensions (Aggval Join)
- Pre-Group operation
- Residual Join (Group Exact Join)
- Post-Group operation
- Residual Join (Group Value Join)
- Standard Joins

The operator tree of Figure 10-7 shows the basic structure for the general grouping optimization. This operator tree contains the residual join for secondary dimensions (see Section 9.5), immediately above the operator tree for the fact table access at the bottom of Figure 10-7. The secondary dimensions are denoted by D^S_k. The `PROJ` node ensures that only attributes are pipelined that are necessary for further processing.

The `GROUP` node above the residual join of the secondary dimensions is the pre-grouping operator and groups on h-surrogate prefixes as described in Section 9.4. The pre-grouping optimization has to consider the aggregate functions that may contain feature attributes of the fact table or hierarchy or feature attributes of the dimension tables. Attributes of the dimension tables (except dimension key attributes) are not available at pre-grouping and must be replaced by h-surrogate prefixes. The actual values are fetched in the residual join. If an aggregate function contains an arithmetic expression, it depends on the expression whether we can apply pre-grouping or not (see Section 10.2.4).

After the pre-grouping step, the residual join to fetch the dimension attribute values is done, denoted by the following `TIMES` cluster and consecutive `RESTR` (inclusively the operator tree with the join conditions) with the "early" dimensions D_{ek}. The second grouping step is represented by the `GROUP` node with a residual join operation, shown by the `TIMES` cluster with the corresponding `RESTR` node and the "late" dimensions D_{lk}. The `RESTR` node finalizes the operator tree.

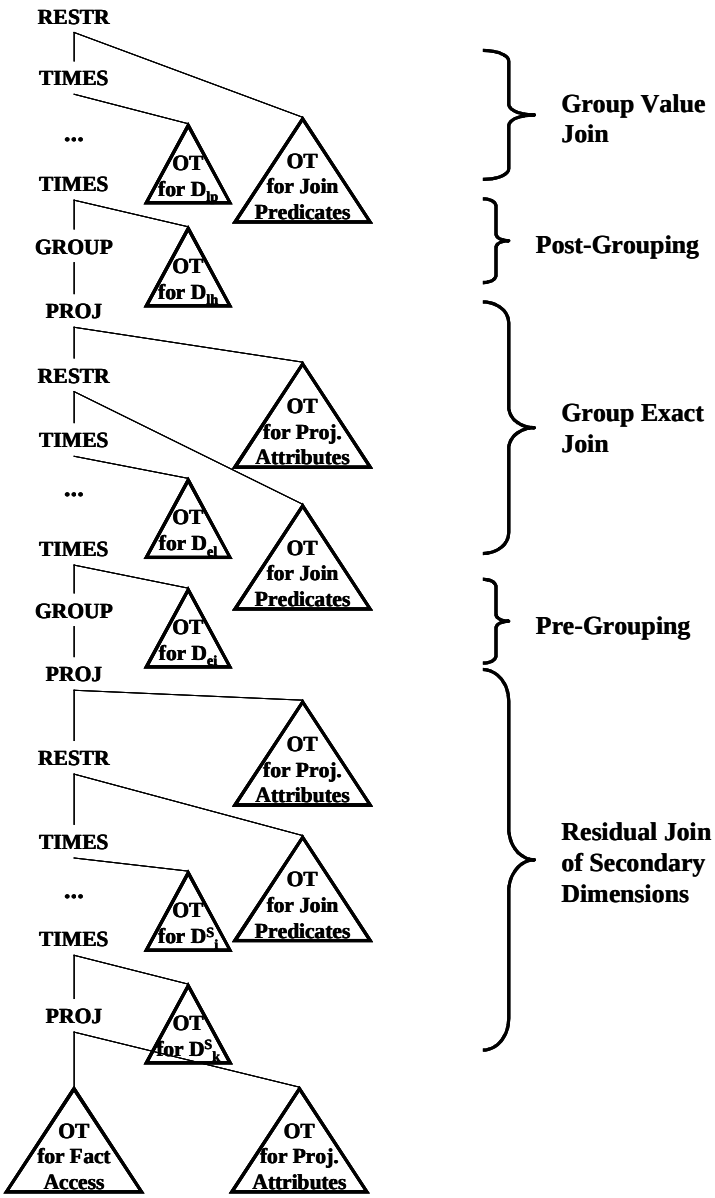


Figure 10-7: Basic Operator Tree for Grouping and Residual Join

The algorithm (in a high level syntax) is shown in Algorithm 10-1.

Algorithm 10-1 (Grouping and Residual Join):

```
makeOTforGroupingAndResJoin:
     handleGroupBy()
     makeResidualJoin()
     performStdJoin()

handleGroupBy:
     searchMHCGroupBy()
     getGroupProperties()
     analyzeSplitAggregationFunctions()
     makeResidualJoin()
     makeGroup()
```

```
makeResidualJoin:
     makeTimesForResidJoin()
     makeRestrForResidJoin()
```

Algorithm 10-1 shows the basic steps how to build the operator tree for grouping and residual join. The main routine is `makeOTforGroupingAndResJoin` with the sub-routines for the grouping, for the final residual join and the standard join of the dimension tables (only for snowflake dimensions). The algorithm first searches for the original `GROUP` node in the operator tree (`searchMHCGroupBy`) by a recursive search (traversed in pre-order, i.e., first the root, then the sub-trees from left to right etc.). The routine `getGroupProperties` calculates the properties of the grouping, e.g., the types of the aggregation function arguments. If we cannot apply hash grouping, grouping is done via sorting and grouping (the standard method how Transbase® handles grouping). This occurs, if the `DISTINCT` operator occurs within an aggregation function, e.g., `SUM(DISTINCT a)`. For further details refer to Section 10.2.5.

The function `analyzeSplitAggregationFunctions` is explained in Section 10.2 in more detail. Basically it analyzes the aggregation function w.r.t. the available attributes and expressions to split and builds a structure to keep these information. Splitting of expressions is necessary for aggregates on dimension attributes. `makeResidualJoin` in `handleGroupBy` is responsible for the residual join of secondary dimensions. This residual join comes before the pre-grouping step, in order to perform post-filtering on the secondary dimensions and delivers the required attributes for the grouping operations. This call is also necessary, if no secondary dimensions occur, because it makes the basic operator tree from the tables represented by the vertices and edges structures. The routine `makeGroup` does the actual grouping. We show the steps in Algorithm 10-2 in more detail.

The `makeResidualJoin` function constructs the residual join depending on the status, i.e., an *aggval join* or *exact grouping join* etc. It consists of the methods `makeTimesForResidJoin` and `makeRestrForResidJoin` to build the `TIMES` cluster and `RESTR` node with the join predicates.

In Algorithm 10-2 we show how to build the grouping nodes. The routine `makeGroup` that is called by `handleGroupBy` (see Algorithm 10-1) first does the pre-grouping (`makePreGroup`) and the residual join and post-grouping, if necessary. Finally the *groupValueJoin* is done, in order to fetch the needed attribute values. A special `PROJ` node is added for the projection of the grouping and aggregation attributes.

Algorithm 10-2 (Make Grouping):

```
makeGroup:
     makePreGroup()
     if needPostGroup
          makeResidualJoin()
          makePostGroup
     if groupValueJoinNeeded
          makeResidualJoin()
          makeFinalGroupProjection()

makePreGroup:
     computeGroupTruncLevel()
     FOR EACH grouping field gf
          if gf available
               initialize for exact grouping
          else       // grouped by ref. surrogate or joined field
               getSurrogateInformation()
```

```
            extendBySurrOrNormalField
      addSecDimJoinFieldToGroupingFields()
      FOR EACH aggregate field af
            if af available
                  applyMappingToAttr()
            else
                  setNeededAttr()
      FOR EACH primary dimension
            addRepresFunction()
      IF noHashGroup
            makeConventionalSort()

makePostGroup:
      addProjectionForNotAvailableFields()
      handleSpecialAggregateCompuation()
      FOR EACH dimension
            addRepresFunction()
```

The pre-grouping is the most complex operation in the generation of the grouping and residual join operators. The mapping from the original attributes to h-surrogate prefixes must be computed for all cases (see Section 9.4 for more details of the concepts).

First, we compute the prefix of the h-surrogates (`computeGroupTruncLevel`) for every dimension. For each grouping field we check, whether it is already available, i.e., in the fact table or delivered by an already performed join (usually secondary dimensions). If the field is available, some initialization parameters are set (we do not enlarge on this) and grouping is done with standard methods. If the field is not available, we have to handle it via an h-surrogate (see Section 9.4.2.2). For this purpose, we need some information about the corresponding surrogate (`getSurrogate-Information`) and extend the grouping field list by the surrogate, if it is not already listed (`extendBySurrOrNormalField`).

If a field of a secondary dimension occurs in the grouping clause, we add the corresponding field in the grouping list (`addSecDimJoinFieldToGroupingFields`).

The aggregation attributes can be available (similar to the grouping attributes), e.g., fact table measure attributes. Otherwise they are missing attributes (e.g., dimension attributes) that are available later after the residual join. If the attribute is available, we use the attribute w.r.t. the aggregation function (`SUM`, `COUNT` etc.). The value of the aggregation is computed during grouping. Not available aggregation attributes are marked (`setNeededAttr`). These attributes are computed after the residual join, and the calculation of the final result of the aggregate is done at this step (see Section 10.2.3).

For the optimization of the residual join, we add a so called representative function (`addRepresFunction`). With this representative operation, denoted as parameter in the `GROUP` node by `repres`, this join is implemented as conventional nested loop join, but as a quasi hash join.

If no hash group can be applied, we have to add a normal sort operator, in order to perform standard grouping via a previous sorting (`makeConventionalSort`). Hash group cannot be applied, if a `DISTINCT` aggregation occurs within an aggregation function, e.g., `SUM(DISTINCT a)`. A hash group does aggregation at once and does not store the single values of the aggregation attribute. Thus, the `DISTINCT` *a* operation fails when using hash grouping (see Section 10.2.1).

Now we describe the algorithm for `makePostGroup`. So far the pre-group and residual join operators have been added to the operator tree. The post-group operator first builds the projection for the attributes that have been joined by the previous residual join by the routine `addProjectionForNotAvailableFields`. This projection is necessary for the additional computation resulting from the post-calculation of the dimension attributes that are available after the actual aggregation calculation (at the residual join). This calculation is the product of the dimension attribute value with the number of group members. For more complex aggregations, we call the method `handleSpecialAggregateCompuation`. Depending on the arithmetic expression within the aggregation function, special calculation is necessary (see Section 10.2.4). For each dimension to join, we again add the representative function, in order to speed up the following residual join (`addRepresFunction`).

In the following we present the overall operator tree for SQL Statement 10-1. The operator tree is generated by the algorithm described. The operator tree as generated for the interval calculation is not included, it already was shown in Figure 10-6. Above this operator tree, there is a `PROJ` node with three `SUBRG` nodes. Each `SUBRG` node specifies for one reference surrogate, which bits are used for pre-grouping. For example, for the surrogate $cs_{customer}$ the first seven bits are used for pre-grouping. The `GROUP` node represents the pre-grouping step. After the pre-grouping operation, six dimension tables are joined: *customer*, *product*, *customer_country*, *customer_dept*, *product_cat* and *product_group*. Now the post-group operation is performed (`GROUP` node) and the dimension table *date* is joined (final `TIMES` node). `SEL` represents a node, in order to indicate a `SELECT` query and name the attributes.

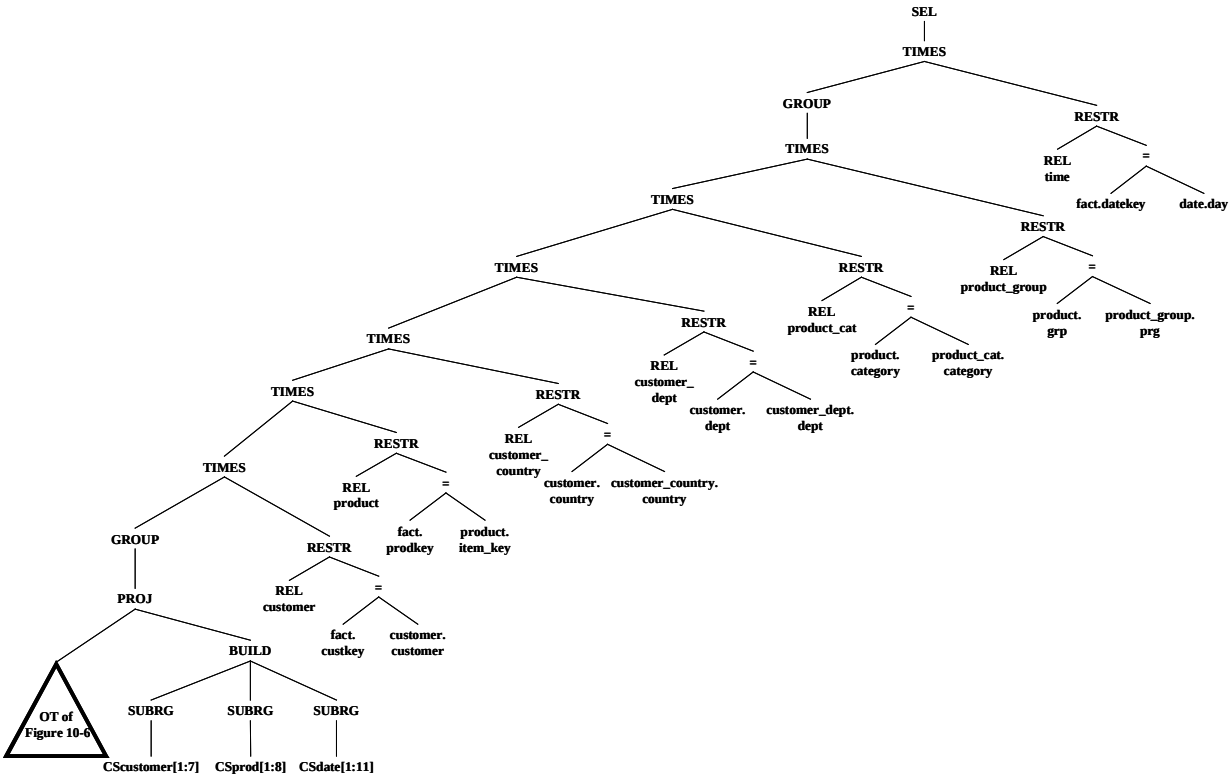


Figure 10-8: Operator Tree with Grouping and Residual Join Optimization

We show the same operator tree in Figure 14-25 (Appendix C) in Transbase® notation. This notation contains additional information (especially for the grouping and the join operations). For example, the `TIMES` nodes are marked with the corresponding residual join function (*groupexactjoin*, *groupvaluejoin*). The representative join attributes are marked in the `GROUP` nodes by *repres*. The join operators (`TIMES`) with the projection nodes enlarge the operator tree very much, because the projection optimization is done in a later optimization step and the combination of all attributes of all

dimensions occurs in this step in the `PROJ` node. Thus, we shorten the illustration by leaving some of the nodes out (denoted by "…").

10.1.8 Interval Generation and Index Access

A later optimization step (*indexacc*) transforms the preliminary interval generation node `COMPSURR` to the final interval generation operator tree depending on the query classes. This optimization step also optimizes the index access, in order to efficiently retrieve the tuples and evaluate the dimension predicates. The interval generation is done at this later step (not in the join optimization phase), because the access to the indexes depends on the interval generation. The standard index opimization method handles the access to specific indexes (primary or secondary) depending on the knowledge about the access and needed and known attributes. The interval generation depends on the access plan and can be done at this step. Thus, the index access intelligence is isolated from the MHC join optimization.

The result is more or less the final operator tree, only some minor optimizations are performed after the *indexacc* optimization (at least for star join queries).

We describe the interval generation for each query classe, i.e., *HPP*, *HNPP,* and *NH*. The query class is coded into the `COMPUSRR` node as parameter. A `COMPSURR` node contains the parameters `COMPSURR { attrpos hdg }`, where *attrpos* denotes the attribute position of the compound surrogate and *hdg* is the hierarchy degree of the dimension.

$$hdg \begin{cases} > 0 & \text{if predicate class is HPP} \\ < 0 & \text{if predicate class is HNPP} \\ = 0 & \text{if predicate class is NH} \end{cases}$$

In the sample query of SQL Statement 10-1 (field normalized snowflake schema of Sales DW as described in Section 10.8), $hdg_{customer}$=5 for the *customer* dimension, i.e., the hierarchy prefix of the hierarchy levels h_6 (*country*) and h_5 (*department*) is restricted. For the product *dimension*, $hdg_{product}$=3, because the *category* level is restricted only. Note that the group level is restricted via a join predicate to the dimension table *product_group*. However, there is no local predicate on that hierarchy level resp. dimension table. If the foreign key constraint

```
FOREIGN KEY customer (group) REFERENCES customer_group (group)
```

exists, the restriction is superfluous and is not considered for the hierarchy search degree, because all values of *customer.group* occur in *customer_group.group*. Otherwise, $hdg_{product}$=2.

The *time* dimension has a hdg_{time}=2, because the hierarchy levels *year, quarter,* and *month* are restricted.

For the query of SQL Statement 10-2, we have the following hierarchy degrees:

- $hdg_{customer}$ = 5 (*HPP*, i.e., `COMPSURR { 8 5 }`), because the hierarchy prefix *country* and *dept* are restricted,
- $hdg_{product}$ = -2 (*HNPP*, i.e., `COMPSURR { 5 -2 }`), because the hierarchy level *grp* (h_2) is restricted,
- hdg_{date} = 0 (*NH*, i.e., `COMPSURR { 9 0 }`), because the feature attribute *day_of_week* is restricted.

In the following, we first discuss the operator `CS2IVAL` that is necessary for the interval generation of *HNPP* and *NH* predicate classes. Then we describe the interval generation for each hierarchy class and use as example the sample query of SQL Statement 10-2.

```
SELECT
 country_str, dept_str, grp, year, quarter, month, SUM(val),SUM(qty)
FROM
  customer c, customer_country c_country, customer_dept c_dept,
  product p, date d, fact f
WHERE
  f.custkey = c.customer AND c.country = c_country.country AND
  c.dept=c_dept.dept AND f.prodkey = p.item_key AND f.timekey=d.day
  AND country_str = 'GERMANY' AND dept_str = 'SOUTH' AND
  grp = 'COLOR' AND day_of_week = 'Monday'
GROUP BY
  country_str, dept_str, grp, year, quarter, month
```

SQL Statement 10-2: Query with HPP, HNPP and NH Predicates

10.1.8.1 CS2IVAL Operator

The CS2IVAL operator is used to find intervals from a set of compound surrogates cs resulting from a dimension predicate. Not only consecutive cs values form an interval, but also two compound surrogates cs_1 and cs_2, $cs_1 < cs_2\text{-}1$ and there is no cs_3 with $cs_1 < cs_3 < cs_2$. This means that there is no (used) cs value between two compound surrogates. Due to the relationship of the reference surrogates of the fact table and the compound surrogates of the dimension tables, all reference surrogates occur in the dimension table and thus an interval generated by CS2IVAL operator covers the correct values also in the fact table.

The CS2IVAL node works by processing its input against the index *DXcs*. Remember that *DXcs* is a secondary index on the compound surrogate of the leaf dimension table.

CS2IVAL starts to fetch its sorted input sequence cs_1 cs_2 ,.., and first makes a direct access into *DXcs* using cs_1 as search value (this and all following direct accesses must lead to a hit). After the direct access, the input sequence is compared with the cs values which follow in the *DXcs* sequence in the index. When a cs_k input value is not found in the index as a direct successor of the previously found value cs_{k-1}, then a result interval is built with (cs_1 , cs_{k-1}) as lower bound and upper bound values. Then the next direct access with cs_k as search value is made and the method continues (cs_k will be the lower bound of the next interval to be constructed).

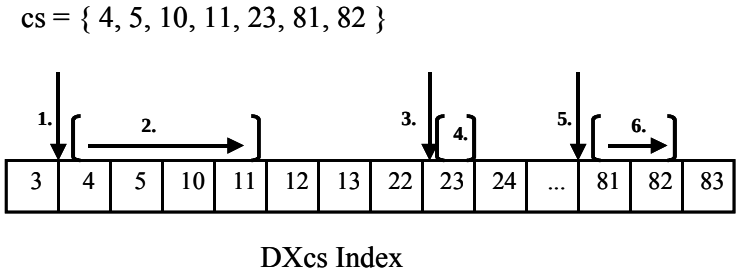


Figure 10-9: CS2IVAL Operator

Figure 10-9 shows an example for the CS2IVAL operator. We use a decimal representation of the compound surrogates instead of bit strings. The total order is the same for bit strings (with equal length) and decimal numbers. Assume that the cs values {4, 5, 10, 11, 23, 81, 82} result from the local predicate of the dimension table. All of these values must also occur in the *DXcs* index. The first cs value, i.e., '4' is used as search argument for a direct search in the index (1.). Then we compare the consecutive cs values of the input with the consecutive index entries until the incoming cs value is larger than the next index entry (2.). In this case, it is the value '23' that is larger than the next index

entry '12'. The cs value '23' is used again as direct search argument (3.). The next cs input value '81' is larger than the next index entry (4.) and a direct search is done (5.) etc.

The result of the CS2IVAL operator of Figure 10-9 are three intervals, i.e., [4, 11], [23, 23] and [81, 82] instead of seven cs values.

The CS2IVAL method generates compound surrogate intervals and the number of created intervals is minimal and thus is optimal w.r.t. the resulting fact table accesses. The effort is linear with the number *n* of incoming cs values. A maximum of *n* direct accesses to the index is performed. However, usually less than *n* direct index searches are necessary. The number of next read operations and comparisons is equal to the number of incoming cs values. The overall maximal cost of CS2IVAL is $c_{CS2IVAL} = n*c_{directaccess} + n*c_{nextread}$.

10.1.8.2 Predicate Class HPP

The dimension operator tree for the *customer* dimension contains a *HPP* restriction with predicates on the dimension tables *customer_country* and *customer_dept*. The access to the leaf dimension tables is done via the secondary index *customerDXh* that contains the hierarchy levels and the compound surrogate (Figure 10-10):

```
customerDXh = (country, dept, county, city, area, customer, cs)
```

It is enough to evaluate one tuple of the index, because the predicates restrict a prefix of the hierarchy, i.e., `country = 'Germany' AND dept = 'SOUTH'`. This is denoted by the property *singletup* of the INDEX node. We pipeline two attribute values from this tuple. The PROJ node generates these values. These values are two compound surrogates that are computed from the compound surrogate of the single tuple access to the index. The remaining bits of the compound surrogate (except the prefix bits) are filled with '0' for the minimum compound surrogate and with '1' for the maximum surrogate (see Section 5.3). For this purpose, we mask the compound surrogate with a BITAND operation with `'111…100…000'` for the minimum compound surrogate. With this mask operation, the prefix bits are unchanged by 'ANDing' with `'111…111'` and the remaining bits are set to 0 by 'ANDing' with `'000…000'`. The maximum compound surrogate is masked by BITOR `'000…011…111'`, i.e., the prefix bits are 'ORed' by `'000…000'` and thus left unchanged, and the remaining bits are set to 1.

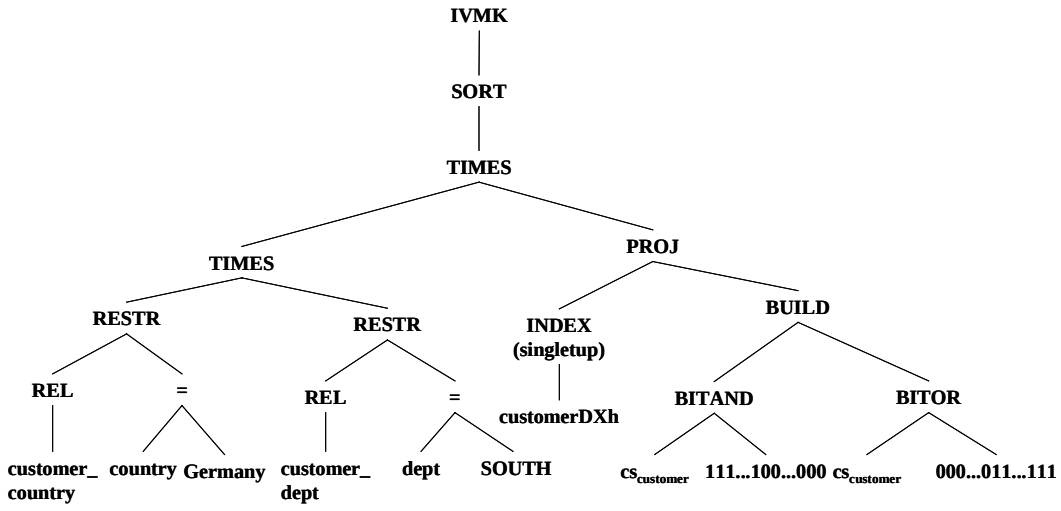


Figure 10-10: Operator Tree for Dimension Customer (HPP) with Interval Generation

The resulting compound surrogate pairs (cs_{min}, cs_{max}) are sorted in the SORT node. Each pair is transformed to an interval in the IVMK (interval make) node.

This example of a *HPP* operator tree is typical, however, quite complex, for this predicate class. For a star dimension (only leaf dimension table), the join between the index access to the leaf dimension table and the higher dimension tables does not occur and the predicates on the dimension tables are local predicates on the leaf dimension table. The index access and interval generation is the same as in Figure 10-10.

10.1.8.3 Predicate Class HNPP

The operator tree for the *product* dimension of SQL Statement 10-2 contains a restriction on the hierarchy level *group*: `grp = 'COLOR'`. In the physical schema, this hierarchy level is indexed by a standard secondary index `group_secIX (group, cs)` that additionally contains the compound surrogate. This dimension has the predicate class *HNPP*, because only hierarchy attributes are restricted.

The interval generation for dimensions of query class *HNPP* generally is done by evaluating the local predicates on the dimension. For each tuple that qualifies the dimension predicate, we retrieve the compound surrogate (either by accessing an index that is used for the evaluation and contains the compound surrogate, or by materializing the compound surrogate from the dimension table itself). The compound surrogates are combined to intervals as far as possible by applying the `CS2IVAL` operator (see Section 10.1.8.1) to the set of compound surrogates. This operator needs a sorted stream of compound surrogates and delivers pairs of compound surrogates as lower and upper bounds of the intervals.

The evaluation of the compound surrogates usually does not return sorted cs values. Thus, a `SORT` operator is used before the `CS2IVAL` node. In Figure 10-11 no `SORT` operator is necessary, because the index access (a point restriction) returns the index tuples *(group, cs)*, where group is restricted to `group = 'COLOR'`. The index tuples are stored sorted by *(group, cs, IK)*.

The `PROJ` node reduces the index tuple to the cs value only.

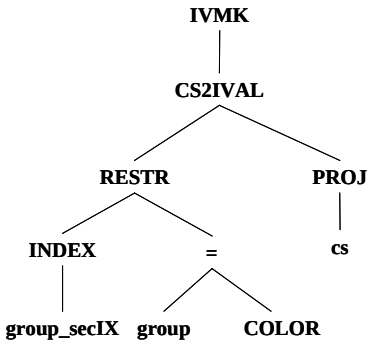


Figure 10-11: Operator Tree for Dimension Product (HNPP) with Interval Generation

The intervals generated by a *HNPP* query can be described as a set of cs intervals that cover a number of compound surrogates (depending on the minimum restricted hierarchy level). The cardinality of the intervals depends on the hierarchy degree *hdg* and on the hierarchy instance. Typically we have a number of hierarchy sub-trees that correspond to the restriction. For example consider a restriction of a date hierarchy: `WHERE month = 'July'`. The result of this restriction are 31*3 = 93 cs values for '*July 2000*', '*July 2001*', and '*July 2002*', if these three years are stored in the date dimension table. The `CS2IVAL` operator transforms the 93 cs values into three intervals.

10.1.8.4 Predicate Class NH

The operator tree for *NH* predicate classes is similar to *HNPP*. The predicate of the dimension results in a number of compound surrogates that are mapped to a (smaller) number of intervals by the `CS2IVAL` operator (see Section 10.1.8.1).

Figure 10-12 shows the operator tree for the date dimension of SQL Statement 10-2, where the feature attribute *day_of_week* is restricted to *'Monday'*. In this case, the *date* dimension table (basic relation) is accessed via a full table scan and the cs values for all tuples qualified by the predicate (`RESTR` node) are returned (`PROJ`). These cs values are sorted (`SORT`) and serve as input for the `CS2IVAL` operator. The result of the operator tree is a set of intervals built by the `IVMK` node that is minimal w.r.t. the compound surrogates. The `IVMK` node builds the internal representation of an interval.

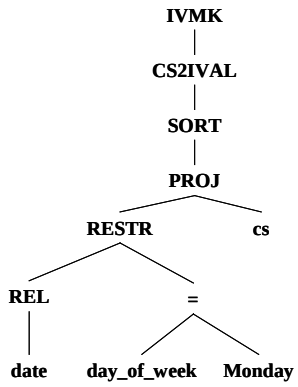


Figure 10-12: Operator Tree for Dimension Date (NH) with Interval Generation

Non-hierarchical predicates can lead to a large number of intervals, since feature attributes often are orthogonal to the hierarchy (e.g., the population of cities in a geographical hierarchy). With the `CS2IVAL` operator, we are able to build a minimal set of intervals for the restriction. However, the overall number of intervals can be still very large and consequently the number of query boxes. If a large number of intervals is generated for several dimensions, the number of multidimensional query boxes is the cross product of the intervals of each dimensions. This can lead to serious performance problems and special optimization is necessary to handle such situations (see Section 10.3).

In some hierarchies, the feature attributes also can correspond to hierarchy levels and determine hierarchy levels, if the feature attributes are restricted. This leads to a small number of intervals (comparable to *HNPP* restrictions). For example, a time hierarchy with *year – month – day* with an additional feature attribute *week*. With the predicate `year = '2002' AND week = 49`, the following days are restricted: '2002-12-01', '2002-12-02', '2002-12-03', '2002-12-04', '2002-12-05', '2002-12-06', '2002-12-07', These days form an interval (also for the compound surrogates) and the `CS2IVAL` operator transforms the set of seven single values into one interval. Note that for the time hierarchy, such relationships of feature attributes to hierarchy levels are very common. For other hierarchies, e.g., the product hierarchy with a feature attribute *color*, additional effort is necessary, in order to optimize the interval generation (see Section 11). With the `CS2IVAL` operator we benefit from such relationships.

Hierarchical pre-grouping for *NH* restrictions is not as efficient as for hierarchical restrictions, because feature attributes usually are only functionally dependent on the most detailed hierarchy level h_1. This leads to h-surrogate prefixes consisting of the complete compound surrogate with only minor advantages for pre-grouping. Feature attributes with hierarchical nature are suitable for pre-grouping. A corresponding snowflake schema design enables the optimizer to recognize such hierarchical relationships.

10.1.9 Remarks

The methods for the recognition of the schema discussed so far do not consider all possible schemata that are suitable for star query processing. For example, a schema could be designed without correct definition of reference constraints. In this case, duplicates on the dimension tables might occur that cause duplicate fact table tuples. According to SQL-92, such duplicates can be evaluated and must be considered for the result of the query. The semi-join methods used in our star query processing algorithm do not handle such duplicates and special algorithms must be implemented. However, in reality, duplicates do not occur although there are missing constraints to prohibit duplicates (due to bad schema design). Thus, we could use the standard star query processing algorithms without loss of tuples. Some dynamic optimization aspects can solve this problem: Proceed with the proposed query processing algorithms as long as there are not duplicates. If duplicates on one dimension occur, use standard query processing methods that can be imperformant but deliver the correct result. This dynamic optimization is not implemented since dynamic optimization requires changes in the overall operator tree generation and execution.

Another problem is the join over several fact tables. Queries may contain a number of fact tables or self joins on the fact table. For example, one of the business queries on the Sales DW requires a self join on the fact table. We implemented a method to deal with multiple fact table within one query (see Section 10.7 for more details).

10.2 Aggregation and Grouping

In SQL-92 the aggregation functions `MIN`, `MAX`, `COUNT`, `SUM`, and `AVG` exist. Each aggregation function has at least one parameter, usually a single attribute. For `COUNT` two different applications are possible: `COUNT(E)`, where *E* is an expression, computes the number of tuples where *E* does not contain a `NULL` value. `COUNT(*)` computes the overall number of tuples. The other aggregation functions also can contain expressions, e.g., `MIN(a+b)`, where *a* and *b* are attributes. If one of the participating attributes for `MIN` or `MAX` is `NULL`, the expression is ignored for the result[23]. If the value of an attribute participating in an expression for `SUM` or `AVG` is `NULL`, then the complete expression for the tuple is 0[24]. `SUM(E)` computes the sum of the expression *E* for all tuples, `AVG(E)` computes the average value of *E*.

The implementation of the aggregation functions is straightforward. `AVG(E)` is equal to `SUM(E)/COUNT(E)`. Thus, for `AVG(E)` we store count and sum and do the cacluclation at the end.

In the context of pre-grouping, aggregation is more difficult, because for aggregates on dimension attributes, the aggregation values are not available when pre-grouping (and aggregating). The following sections describe how to do aggregation in such cases.

10.2.1 Implementation of Hash-based Grouping

We use a hash table implementation for grouping in MHC environments in Transbase®. The key of the hash table is calculated w.r.t. one or more attributes. These attributes are the parameters for the hash function. The hash function itself is quite simple, but provides good distribution for most data investigated so far.

The crucial point when using hash table implementation is to estimate the number of objects (tuples) to store in the hash table. Since we do not have statistics about the multidimensional data distribution on the fact table, an (even rough) estimation of the number of resulting groups is not possible.

[23] If each tuple contains a NULL value in the expression, then the result is NULL.

[24] Note, that also `COUNT(a)` returns the number of tuples where *a* is not NULL. Thus, AVG is computed correctly also for a set of tuples where one or more tuples exist with *a* = NULL.

Thus, we use a fixed cardinality of the hash table for pre-grouping (currently 10.000). The average length of the collision chains is then *n/10.000*, if the number of groups is *n*. For most queries, 30.000 groups (an average chain length of three is still performant) are enough. The initialization of the hash table with a cardinality of 10.000 requires 10.000 * 4 Byte + 16.384 Byte = 56.384 Byte, where the 16KB are used for the first (at the beginning empty) object container. Thus, the overhead of the cardinality does not have large effects on the overall memory consumption and the fixed cardinality of 10.000 can be increased or decreased, if other sizes turn out to be reasonable.

The cardinality n_{post} of the hash table for the post-grouping step depends on the number of groups resulting from pre-grouping n_{pre}, because $n_{pre} \geq n_{post}$.

In the case, that there is not enough main memory available for the remaining objects, we established a mechanism to write these tuples into a temporary container to disk. The size of the available main memory depends on the database parameters. The parameter for the size of the local cache determines how much space is available for the database kernel for, e.g., hash tables.

The implementation of grouping on hash tables is straightforward. The grouping attributes $g_1, g_2, \ldots, g_n$ form the key for the hash value *hv*: $hv = f_{hash}(g_1, g_2, \ldots, g_n)$. The hash lookup with *hv* returns a pointer to the first object with the same hash value. All objects in the collision chain are tested, whether the grouping attributes of the object (tuple) are equivalent with $g_1, g_2, \ldots, g_n$. In this case, a group already exists and the aggregation value must be re-calculated. For example, for an aggregation function *SUM(a)*, the attribute *a* in the group is adapted: $a := a + a_{new}$, where a_{new} is the value of the attribute *a* for the new tuple belonging to the group. For performance reasons, we calculate an update of *a* in place, i.e., we write the new value of *a* at the corresponding memory position without constructing and writing the complete group tuple. For attributes with variable length data type, e.g., `CHAR(*), VARCHAR(n), BINCHAR(*)` etc., we use a special method. An update that enlarges the attribute value would cause a reconstruction of the tuple (change of the attribute pointers and the required space) and a re-insertion of the tuple, because the object container is occupied densely. A delete (and re-insert) would leave a hole in the object container that cannot be re-used without free-space handling overhead. Thus, we do not store the attribute values for such data types (`CHAR(*), VARCHAR(n), BINCHAR(*), BINCHAR(n), BITS(*), BITS(n)`). We store a pointer to a special *string container* holding the values of these attributes. Within the container, it is relative easy to handle string enlargements by re-allocation of heap space.

If no tuple with the same grouping attributes $g_1, g_2, \ldots, g_n$ compared to the current tuple exists, the group is appended to the chain of the hash objects with the same hash value. A new entry in the object table is created that holds the grouping tuple and the collision chain is maintained accordingly.

After the calculation of all groups, i.e., all incoming tuples are handled, the groups are returned by fetching all objects in the hash table. The first group is the first entry with the smallest hash value, the second is the next object in the collision chain with the same hash value (if existing). After the objects of the chain have been returned, the first object of the next smallest used hash value is returned etc.

Thus, the order of the returned groups depends on the hash function and insertion order of the groups. Depending on the SQL statement, there can be an `ORDER BY` clause. In this case, the groups are sorted in a later step.

In SQL we have to care about `NULL` values. Grouping on attributes that may contain `NULL` values (e.g., measure attributes or dimension table feature attributes) is done in a standard way, where we interpret the `NULL` value of such an attribute as a special value. The hash function returns a special value for the `NULL` value. We do not have to care about further operations, because two tuples with a `NULL` value in the grouping attribute are coalesced to one group.

10.2.2 Basic Aggregation

For pre- and post-grouping, the aggregation functions `MIN`, `MAX` and `SUM` are implemented straightforward. `COUNT` (and `AVG`) has to be modified for the post-grouping step:

The aggregation function values for `COUNT` of the pre-grouping phase have to be added for all merged groups (`AVG=SUM/COUNT` and therefore is affected by the new `COUNT` computation). For example, consider the following SQL statement that aggregates a measure *FACT.a* and groups w.r.t. hierarchy level h_k of dimension *D*, where h_k is any hierarchy level of the hierarchy $h_t, h_{t-1}, \ldots, h_k, h_{k-1}, \ldots, h_1$ with h_t ist the top level and h_1 the leaf level of the hierarchy.

```
SELECT COUNT(FACT.a) FROM FACT, D, … WHERE … GROUP BY D.hk
```

This statement can be transformed to the following SQL statement, in order to show the aggregation computation:

```
SELECT SUM(cnt) FROM
      SELECT COUNT(FACT.a) AS cnt, D.hk AS H FROM FACT, D WHERE …
      GROUP BY FACT.hsk(k)
GROUP BY H
```

Pre-grouping is done on the h-surrogate prefix for dimension *D* on level h_k specified informally by *hsk(k)*, i.e., *hsk(k)* contains the components $(h_t, h_{t-1}, \ldots, h_k)$. After pre-grouping, we add the counters and group w.r.t. the actual grouping attribute h_k.

More sophisticated aggregation semantic is necessary for aggregation on dimension attributes (Section 10.2.3) and for complex expressions in aggregation functions (Section 10.2.4).

10.2.3 Aggregation of Dimension Attributes

In most cases, star queries aggregate fact table measure attributes. The aggregation operation has to be modified, if a dimension attribute a_d is involved in the aggregation. Pre-Grouping is done on the h-surrogate prefix w.r.t. the grouping order *GO* of the dimension (see Section 9.4.2.2). Each resulting group from the pre-group operation represents a number of tuples with the same value of a_d. The value of a_d, however, is still unknown. The aggregation operations `MIN` and `MAX` are implemented straightforward and evaluated after the residual join. `SUM` has to be modified: We additionally compute `COUNT(*)` in the pre-group step and multiply it with the value of a_d after the residual join.

For example, the SQL statement

```
SELECT SUM(D.ad), D.hk, … FROM FACT, D, … WHERE … GROUP BY D.hk
```

groups the tuples according to $D.h_k$. If a_d is known to be functionally dependent on h_k, we perform hierarchical pre-grouping on the compound surrogate prefix of h_k. The groups from the pre-grouping step contain the following attributes: $(?, cs(h_k), \ldots, COUNT)$, where "*?*" is a placeholder for the still unknown value of a_d, $cs(h_k)$ is the compound surrogate prefix corresponding to h_k and *COUNT* is the number of fact table tuples contributing to this group. After the residual join with *D*, the group tuple is modified to $(a_d * COUNT, D.h_k, \ldots)$, i.e., $SUM(D.a_d)$ is computed by $D.a_d$ multiplied with the number of tuples in this group.

`AVG` is computed by `SUM/COUNT` and is also affected by the special `SUM` computation.

For example consider a query with `SUM(LOCATION.population)` in the `SELECT` clause. For each tuple t_k' representing group S_k of the pre-grouping phase we store in t_k'.*cnt* the number of original

fact table result tuples contributing to this group (`COUNT(*)`). In the residual join phase with *LOCATION*, we compute the aggregation value: $a = t_k'.cnt * LOCATION.population$.

10.2.4 Expressions in Aggregation

A more difficult problem are complex aggregations, i.e., expressions in the aggregation functions, especially, if fact and dimension attributes occur within such an expression. An example is a measure with different currencies and exchange rates. In this case, we have a currency dimension table `Currency (cid, exchrate)` and the following SQL statement:

```
SELECT SUM(F.turnover * C.exchrate) FROM Fact F, Currency C WHERE
F.dcurrency = C.cid AND …
```

At pre-grouping *F.turnover* is known. Thus special preparation is necessary: We have to split the expression into an expression that can be computed at pre-grouping and an expression that is computed later (similar to Section 10.2.3). In the example above, we have to delay the computation of *C.exchrate* after the residual join.

Basically, the general form of an aggregation function is *AGG(exp)*, where *exp* is an arithmetic expression. *exp* also can contain the `DISTINCT` operator, e.g., `SUM(DISTINCT (F.turnover))`. In the implementation, we can handle a special class of arithmetic expressions *exp*: exp must be a product consisting of factors, where each factor is an arithmetic expression that either contains only fact table attributes or only dimension table attributes. Such an expression is split into a part that is evaluated during pre-grouping (aggregation of fact table attributes) and a part that is evaluated after the residual join (dimension table attributes). The final result of the aggregation is computed by these two results and multiplied with the number of tuples for this group (`COUNT(*)`).

The decision whether to split the aggregation function and apply pre-grouping or not is done in the routine `analyzeSplitAggregationFunctions` (see Section 10.1.7). The expression of the aggregation is represented by an expression operator tree in the `PROJ` node above the `GROUP` node (see Figure 10-13). The `BUILD` node in Figure 10-13 contains the expression, represented by the `MULT` node and a number of other attributes needed for grouping. The `SUBRG` nodes represent the compound surrogate prefixes, in this case, for pre-grouping.

We traverse the expression operator tree until we find a `MULT` node (for multiplication). We recursively traverse the sons of the `MULT` node, in order to check that only fields of either the fact table or the dimension tables occur. If there is an inconsistency, i.e., a mixture of fact and dimension table attributes, we cannot split this expression and thus cannot use pre-grouping for the dimensions that occur in the expression.

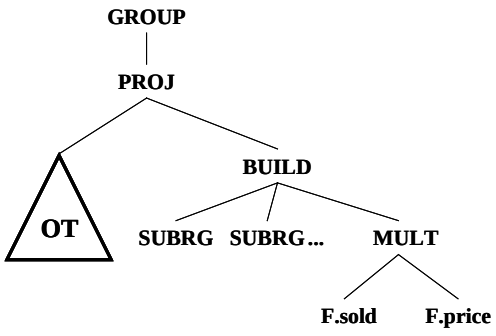


Figure 10-13: Example Aggregate Expression

The following expressions allow pre-grouping:

```
SUM(fact.sold * fact.price)

SUM(fact.sold * fact.price * 1.99434)

SUM((fact.sold + 10) * fact.price * 1.99434)

SUM(fact.sold * fact.price * (currency.exchrate + cust_country.tax))
```

The following expression does not allow pre-grouping:

```
SUM(fact.sold * fact.price * (currency.exchrate + fact.credit))
```

The following properties for complex expressions must be fulfilled, in order to allow for pre-grouping:

- Aggregation functions SUM, AVG
- Product Form: $e = f_1 * f_2 * \ldots * f_n$
- Each fact fi is an expression that either contains fact table attributes or dimension attributes

Note that various expressions can be split into expressions that can be handled as described, e.g., $SUM(F.m_i*D.h_j + F.m_k*D.h_l)$ is split into $SUM(F.m_i*D.h_j) + SUM(F.m_k*D.h_l)$. Special information has to be stored within the groups, i.e., the atomic expressions and the arithmetic operations. These information contains the split operators of the arithmetic expression and eventually some rules how to handle special cases. Handling of quotients is similar to products, but has not been implemented. We must care about the division with zero. The divisor and dividend each may contain either fact or dimension table attributes again.

Additionally, we must care about the NULL semantic. If one of the attributes involved in the expression is NULL, the complete expression is 0 for SUM (and thus for AVG). Therefore, we have to check if any attribute is NULL: for measure attributes at pre-grouping, for dimension attributes after the residual join.

Complex expressions can be handled for the aggregation functions SUM and AVG. Complex MIN and MAX expressions cause a large effort to implement pre-grouping (see Section 10.2.5).

10.2.5 Failure of Pre-Grouping for Aggregations

Some aggregation functions cannot be handled in the current Transbase® implementation by pre-grouping. Such functions are expressions containing a combination of fact table and dimension table attributes like $MIN(F.m_j * D.h_k)$. For fact table processing it is not clear whether $MIN(F.m_j)$ or $MAX(F.m_j)$ has to be used for the aggregate, because $D.h_k$ can have opposite signs compared to $MIN(F.m_j)$. Then $MIN(F.m_j) * MIN(D.h_k) > MAX(F.m_j) * MIN(D.h_k)$.

Such expressions can be handled only with huge effort and are not easy to implement for the general case (consider complex expressions where all cases have to be stored for each group).

Another class of expressions where pre-grouping cannot be applied is the DISTINCT operator within an aggregation function, e.g., SUM(DISTINCT D.a). Here, the aggregation in the grouping process does not store the single values of *a* of dimension table *D* for the aggregation. The computation of DISTINCT fails.

If the foreign key reference constraints between the fact table and a secondary dimension table is not specified, the dimension table must be post-filtered. The post-filtering takes place immediately after the fact table access and before pre-grouping. Thus, for such dimensions, pre-grouping cannot be applied.

Summing up, some special expressions can be handled easily with pre-grouping. However, the vast majority of expressions cause problems and huge implementation effort. Most data warehouses, however, do not use such complex expressions. Complex metrics that reflect business content are implemented in front-end tools.

10.2.6 Memory Consumption

Hierarchical pre-grouping reduces the amount of memory that is necessary for a hash based implementation of grouping. Each group corresponds to one object (group tuple) in the hash table. If a tuple belongs to an already existing group, only the aggregation values are modified. We therefore need $m*t_{size}$, if m is the number of groups resulting from pre-grouping and t_{size} is the average size of one tuple. Additionally we need some overhead for the implementation of hash grouping, i.e., $k*4$ byte for the mapping of the hash values to the object container, where k is the cardinality of the hash table. For each pointer to the next object in the case of collisions we again need a four byte pointer, i.e., $k*l_{chain}*4$, if l_{chain} is the average chain length in the hash table.

The overall main memory for grouping therefore is:

$m*t_{size} + k*4 + k*l_{chain}*4$ byte.

If the available main memory is not enough, we have two alternative strategies, in order to deal with this too large number of groups.

The first alternative is to perform grouping until the available main memory is exhausted. Each tuple that belongs to one of the already hashed groups can be handled by modifying the group. All other tuples are written to a temporary container on disk. After all tuples have been handled, we empty the hash table by pipelining all groups to the parent node in the operator tree and proceed with the pre-grouping with the tuples of the container on disk. Note, that several iterations of this process can be necessary, in order to group all tuples. This solution generates the same groups (number and values) as a full in-memory hash-grouping.

The alternative is to pipeline and remove the group that has not been modified for the longest time (some kind of LRU characteristics) from the hash table ([Lar02]). Now a new group can be added to the hash table. With this approach, the number of resulting groups is increased, but Larson shows in his measurements that this method does not produce extremely larger numbers of groups. The additional groups are coalesced at the post-grouping step. So this method requires a post-grouping operator with all dimensions of the pre-grouping step. This approach needs no storing of tuples on the disk, however, with the larger number of groups, the residual join will take longer correspondingly. The post-grouping also has to deal with a larger number of incoming groups

Depending on the dimension predicates, hierarchical pre-grouping reduces the number of groups dramatically (compared to the number of tuples resulting from the fact table access). Thus, hierarchical pre-grouping reduces the necessary amount of memory. Less overall memory is necessary or more parallel users can share the available memory.

10.2.7 Remarks

Handling complex expressions with pre-grouping is an interesting topic. Basically, if an aggregation function can be decomposed, it is possible to apply pre-grouping on the expressions. However, a large implementation and run-time effort must be spent for a general solution. This effort even might exceed the benefit of pre-grouping. Further investigation is necessary to develop algorithms, data structures, and implementation to generally handle complex expressions in aggregation functions.

The additional effort must be compared to the achieval speed-up by delaying residual joins after the pre-grouping step. So a cost-based approach is necessary, in order to decide which optimization to apply.

10.3 Multi-Query-Box Handling

One serious problem for star queries on a database schema modeled with MHC is the occurrence of many query boxes. If a hierarchical restriction in dimension D_i leads to m_i intervals, then m_i query boxes are created in combination with the other dimensions. If another dimension D_j leads to m_j intervals, then $m_i * m_j$ query boxes must be handled by the underlying indexing structure. Generally, the number of query boxes for n dimensions $D_1, D_2, \ldots, D_n$ is $\prod_{i=1}^{n} m_i$. Some hierarchical restrictions lead to a large number of intervals, e.g., restrictions on feature attributes. For example consider a query restricting the *age* of customers, where *age* is a feature attribute to the usually geographical hierarchy. The number of intervals can be several thousands.

A query like

```
SELECT SUM(turnover)
FROM Fact F, Customer C, Date D
WHERE F.cid = C.cid AND F.did = D.did AND C.age BETWEEN 25 AND 30
AND D.year = 2002
```

can be rewritten into

```
SELECT SUM(turnover)
FROM Fact F
WHERE
  (F.hsk_cust BETWEEN 1 AND 4 AND F.hsk_date BETWEEN 1 AND 364) OR
  (F.hsk_cust BETWEEN 7 AND 8 AND F.hsk_date BETWEEN 1 AND 364) OR …
```

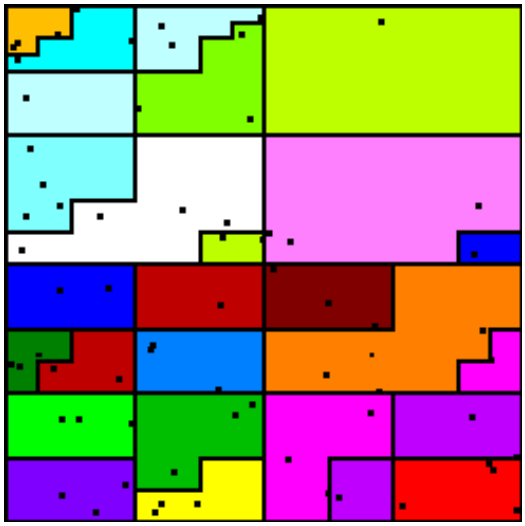


Figure 10-14: Two dimensional UB-Tree

Those intervals depend on the data, but the number of intervals can be very large. Often multiple query boxes even intersect the same disk pages. The range query algorithm is performed on every query box and requires multiple loading of the same pages (see Figure 10-15).

For illustration purpose we use a two dimensional UB-Tree with a page capacity of four tuples. Figure 10-14 shows the UB-Tree with the tuples.

In Figure 10-15 multiple query boxes are shown. The query boxes are illustrated by transparent rectangles. Not all query boxes contain tuples. We call such query boxes *empty query boxes*. For example the four query boxes in the right upper quadrant of the universe all intersect the same two

pages. Thus, these two pages are post-filtered four times with the predicates of the corresponding query box.

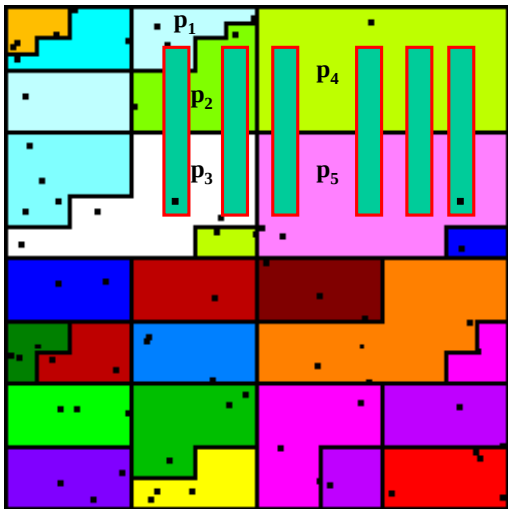


Figure 10-15: Two dimensional UB-Tree with multiple Query Boxes

10.3.1 Standard Query Box Algorithm

The implementation of the UB-Tree in Transbase® is on top of the standard B-Trees (see [RMF+00]). The index part of the B-Tree uses z-addresses as separators, whereas on the leaf levels (leaf pages) the original tuples are stored. These tuples contain the n-dimensional UB-Tree key $(d_1, d_2, \ldots, d_n)$ and are ordered by their z-value $Z(d_1, d_2, \ldots, d_n)$. We have to post-filter the tuples on the leaf pages, in order to decide which of them are inside the query box and which are not.

The standard query box algorithm is *query box oriented*, i.e., for each query box the sequence *compute next page – fetch page – filter tuples on page* is executed. A simplified algorithm is shown in Algorithm 10-3. See [Ram02] for more details.

Algorithm 10-3 (Standard Query Box Algorithm):

```
For each QBi ∈ QB
     start = Z(ql)
     end = Z(qh)
     cur = start
     WHILE(TRUE)
          cur = getRegionsSeparator(cur)
          postFilterPage(GetPage(cur), QBi)
          if cur ≥ end
               break;
          cur = getNextJumpIn(cur, QBi)
```

The query boxes are stored in $QB = \{ QB_i \}$. For each query box we execute the range query algorithm. A query box is determined by a minimum *ql* and maximum *qh* corner. *ql* and *qh* are tuples, *Z(ql)* (*Z(qh)*) computes the z-value of *ql* (*qh*), where $Z(ql) \leq Z(qh)$. See [Ram02] for more details on computing the z-value. We start with the minimum z-value of the query box and compute all pages that are intersected by the query. The pages are computed by iteratively calculating the z-value for the next intersection point of the z-curve with the query box. This continues until a z-value for *cur* is found that is larger than *end* (*Z(qh)*).

The function *getRegionsSeparator* returns the separator for the current z-value. The function *postFilterPage* tests all tuples on the page if they are contained in the query box specified by *ql* and *qh*. *getPage(p)* requests page *p* from the storage manager.

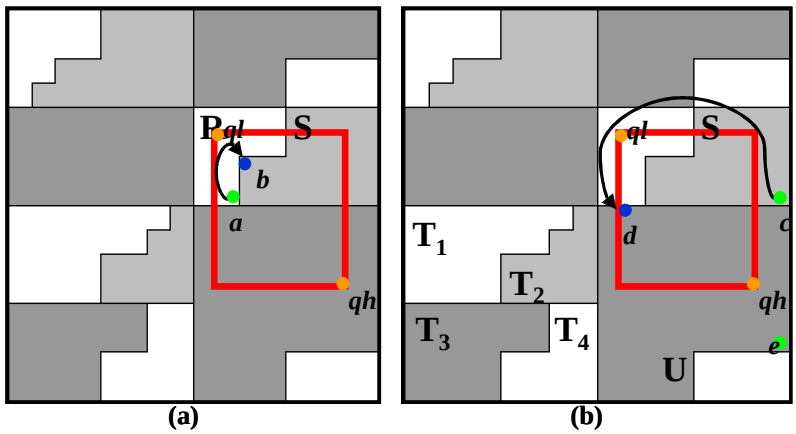
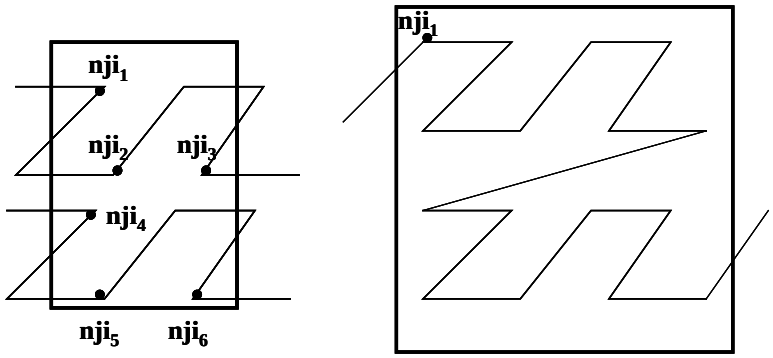


Figure 10-16: Range Query Algorithm

Figure 10-16 shows an example of the range query algorithm. First we fetch the page with region *R* that contains *ql* of the query box (Figure 10-16 (a)). *R* is post-filtered and the next intersection point *b* is calculated, i.e., the largest (according to the z-order) point *a+1* of *R*: *b* = *a+1* in region *S*. After post-filtering *S*, the next intersection point is calculated again. Because the largest point *c* of *S* is not inside the query box, the following *next jump in* computation computes point *d* of region *U*. Note that the regions T_1, T_2, T_3, and T_4 are skipped. After post-filtering *U*, the algorithm is finished, because the largest point *e* of *U* has a larger z-address than *qh*.

A detailed description about the range query algorithm is found in [Ram02] and [RMF+00]. See also Section 2.5 for more details about the UB-Tree and the implementation in Transbase®. For the Transbase® implementation, we first position on the leaf page of the UB-Tree that contains the tuple *Z(ql)*. Note that the tuple *Z(ql)* needs not to exist, but the separator *s*, $s \leq Z(ql)$, of the index part of the tree points to that page. If the page has been found, we post-filter all tuples on that page by the ranges of the query box specified by *(ql, qh)*. The z-value of the last tuple, i.e., largest tuple w.r.t. z-order, is used for the computation of the *next jump in* point *nji* into the query box. *nji* is the first point on the z-curve that is inside the query box. In Figure 10-17 two scenarios of a query box and the z-curve are illustrated. On the left side, the z-curve enters and leaves the query box several times, on the right side, it enters the query box one time and all points on the z-curve are contained in the query box until the z-curve leaves the query box.



(a) Many Jump In/Out (b) One Jump In/Out

Figure 10-17: Query Box Algorithm

nji is used for the B-Tree search in the UB-Tree, i.e., we search for a separator that points to the page containing *nji*. Depending on the B-Tree implementation, the search for *nji* can be done very efficiently, if the search algorithm can jump on the index pages instead of performing a full path search from the top of the B-Tree to the bottom index level (see Figure 10-18). A corresponding B-Tree implementation with pointer to siblings on index page level allows for such jump optimization.

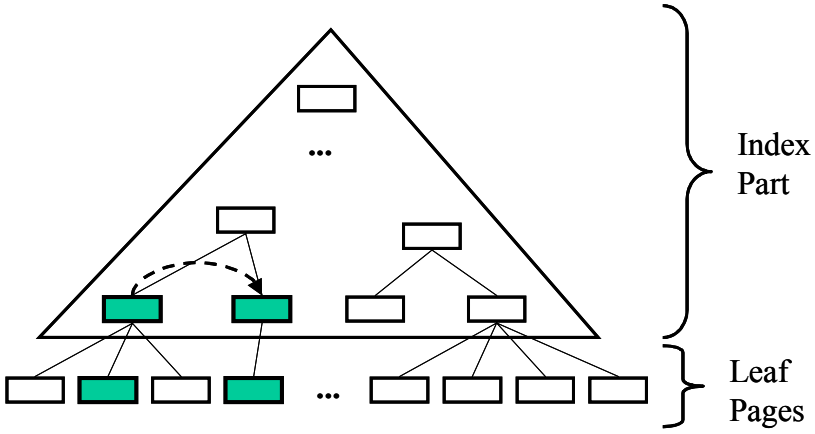


Figure 10-18: B-Tree Jump Algorithm

10.3.2 Optimization on UB-Tree Level

Special algorithms are necessary to handle large numbers of query boxes. For the UB-Tree, see [Fen98] for an algorithm. This method first sorts the minimum corner of the query boxes w.r.t. z-order and processes the first page/region with the smallest z-address, i.e., the query box on top of the list. After applying the range query algorithm on this region, the new region (resulting from the range query algorithm) is inserted into the list at the corresponding place. With this iterative processing and sorting method, we avoid fetching the same page multiple times.

For other multidimensional indexing structures see [PM98] where especially for the R-Tree some algorithms are proposed that are similar to [Fen98]. A formula is developed to decide when it is applicable to merge two or more query boxes and execute them as one (see also Section 10.3.3).

We propose a different solution for the integration of the UB-Tree with many query boxes in Transbase®. The main problem is that with the standard query box algorithm, a page (region) is read several times depending on the number of query boxes that intersect the page. With the proposed algorithm, we read and process every page only once.

Compared to [Fen98] and other approaches, we sort the pages to test w.r.t. page numbers. These page numbers are chosen by Transbase® and do not reflect any multidimensional order.

10.3.2.1 Collecting Pages

In the first step, we collect the page numbers by applying the range query algorithm in the index part only. In the second step we fetch each page via the storage manager and post-filter the page with all predicates contributing to this page. Note that in the case of MHC, multiple query boxes are in disjunctive normal form, i.e., if a tuple lies inside one query box, it belongs to the result set independently on other attributes. Sophisticated predicate testing algorithms are possible to speed up post-filtering.

Collecting the page numbers requires a modification of the original UB-Tree range query algorithm. We perform the *next jump in* algorithm `getNextJumpIn` on the index part of the UB-Tree only. The use of separators as parameters for `getNextJumpIn` enables to get the identifier of the leaf page

without accessing the leaf pages and finds all leaf pages that may contain tuples of the specified query box (see Figure 10-19). The modified algorithm is presented in Algorithm 10-4.

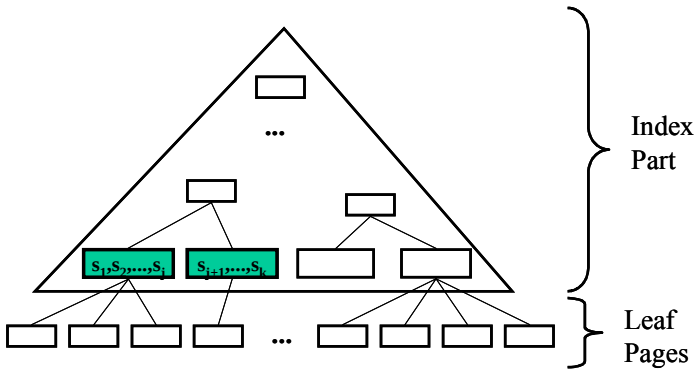


Figure 10-19: Range Query Algorithm on Separators

In Algorithm 10-4, the list of page numbers *PNO* is filled iteratively with page numbers resulting from the `getNextJumpIn` call for the corresponding separators. The overall algorithm is very similar to the original range query algorithm (see Algorithm 10-3). Instead of post-filtering the tuples, we add the page numbers that may contain result tuples and delay post-filtering to a later step. The list of page numbers is sorted after the evaluation of all query boxes, because several query boxes can intersect the same pages. In this case, pages occur several times in *PNO*. Thus, we use the function `sortDistinct` to sort and eliminate duplicate pages. At the end of the algorithm, *PNO* contains a list of page numbers with result tuples of all query boxes. Each page number occurs once.

Algorithm 10-4 (Range Query Algorithm on Separators):

```
PNO = { }
For each QBi ∈ QB
     start = Z(ql)
     end = Z(qh)
     curSep = start
     WHILE(TRUE)
          curSep = getRegionSeparator(curSep)
          PNO = PNO ∪ getPno(curSep)
          if curSep ≥ end
               break;
          curSep = getNextJumpIn(curSep, QBi)
PNO = sortDistinct(PNO)
```

The computation of the page number list accesses only index pages. Some index pages are accessed multiple times, if one of its leaf pages is intersected by multiple query boxes. The comparable small number of index pages allows keeping them in the cache and does not require to load them from disk for each access.

After the page number collecting phase, we have a list *PNO* of page numbers $PNO = \{p_1, p_2, \ldots, p_k\}$ that contains all pages that are intersected by any query box of $QB = \{ QB^1, QB^2, \ldots, QB^m\}$. The query boxes are generated by the *Create Range* operators of the processing plan. Assume that we have n dimensions $D_1, D_2, \ldots, D_n$. The query boxes are specified by:

$QB^1 = ((l_1^1, l_2^1, \ldots, l_n^1) , (h_1^1, h_2^1, \ldots, h_n^1))$

$QB^2 = ((l_1^2, l_2^2, \ldots, l_n^2) , (h_1^2, h_2^2, \ldots, h_n^2))$

...

$QB^m = ((l_1^m, l_2^m, \ldots, l_n^m) , (h_1^m, h_2^m, \ldots, h_n^m))$

The multidimensional interval for QB^1 is specified by a lower bound $l_1^1, l_2^1, \ldots, l_n^1$ in each dimension and an upper bound $h_1^1, h_2^1, \ldots, h_n^1$. For example the query boxes of Figure 10-15 resulting from the intervals [3.0, 3.5] [4.5, 5.0] [5.5, 6.0] [7.0, 7.5] [8.0, 8.5] [9.0, 9.5] for dimension D_1 and [6.0 , 8.0] for dimension D_2 could have the following ranges:

$QB^1 = ((3.0 , 6.0) , (3.5 , 8.0))$

$QB^2 = ((4.5 , 6.0) , (5.0 , 8.0))$

$QB^3 = ((5.5 , 6.0) , (6.0 , 8.0))$

$QB^4 = ((7.0 , 6.0) , (7.5 , 8.0))$

$QB^5 = ((8.0 , 6.0) , (8.5 , 8.0))$

$QB^6 = ((9.0 , 6.0) , (9.5 , 8.0))$

The query boxes are mapped to a *predicate matrix* structure that holds the ranges of the query boxes for each dimension:

D_1: $[l_1^1 , h_1^1] [l_1^2 , h_1^2] \ldots [l_1^m , h_1^m]$

D_2: $[l_2^1 , h_2^1] [l_2^2 , h_2^2] \ldots [l_2^m , h_2^m]$

…

D_n: $[l_n^1 , h_n^1] [l_n^2 , h_n^2] \ldots [l_n^m , h_n^m]$

Thus each query box QB^i is reflected in the predicate matrix by the column

$[l_1^i , h_1^i] [l_2^i , h_2^i] \ldots [l_n^i , h_n^i]$.

Lemma 7:

The ranges of one dimension are either equal or disjoint. □

Proof:

Follows from the Create Range Operator that returns disjoint intervals for each dimension. The query boxes therefore are disjoint, too. q.e.d

The sample query boxes $QB^1, QB^2, \ldots QB^6$ result in the following predicate matrix:

D_1: [3.0, 3.5] [4.5, 5.0] [5.5, 6.0] [7.0, 7.5] [8.0, 8.5] [9.0, 9.5]

D_2: [6.0, 8.0] [6.0, 8.0] [6.0, 8.0] [6.0, 8.0] [6.0, 8.0] [6.0, 8.0]

In the basic implementation, all tuples on pages of *PNO* are tested with the complete predicate matrix, i.e., for each tuple one column of the predicate matrix is used as a multidimensional restriction. If the tuple is inside one of the query boxes, then it is qualified by the restrictions of the range query. The problem with this approach is, that query boxes are tested for tuples and pages that do not intersect the page, because all tuples on a page are post-filtered with all predicates resulting from the predicate matrix. If a tuple corresponds to one predicate, it is in the result of the query boxes.

10.3.2.2 Optimizing Post-filtering: Predicate Bitmap

We introduce a predicate mapping structure holding for each page, which query boxes intersect this page. This bitmap is TRUE for each query box intersecting the page: $PBM = \{ b_1, b_2, \ldots, b_m \}$, where b_i are bits with the status 0 (FALSE) or 1 (TRUE). We test for each tuple on page p all query boxes where $PBM_p[b_i] = 1$. PBM_p represents the bitmap for page p. For example, the bitmaps for the pages $p_1, p_2, \ldots, p_5$ that are accessed in the example look like the following:

PBM_{p1} = { 1, 0, 0, 0, 0, 0 }

PBM_{p2} = { 1, 1, 0, 0, 0, 0 }

PBM_{p3} = { 1, 1, 0, 0, 0, 0 }

PBM_{p4} = { 0, 0, 1, 1, 1, 1 }

PBM_{p5} = { 0, 0, 1, 1, 1, 1 }

With this optimization we reduce the number of predicate comparisons for all tuples on the retrieved pages, because the tuples on one page are post-filtered with the query boxes that intersect that page. The overall number of comparisons still can be very large.

10.3.2.3 Optimization of Post-filtering: Predicate Structure

We suggest a further optimization. We sort the range of the predicate matrix, in order to speed up the test whether a tuple lies within an interval or not. For this purpose, we modify the predicate matrix and do not store the ranges of one query box as a column of the matrix, but sort the ranges for each dimension ascending. Duplicate ranges for one dimension are removed.

D_1: $[l_1^{k1}, h_1^{k1}]$ $[l_1^{k2}, h_1^{k2}]$ … $[l_1^{km1}, h_1^{km1}]$

D_2: $[l_2^{k1}, h_2^{k1}]$ $[l_2^{k2}, h_2^{k2}]$ … $[l_2^{km2}, h_2^{km2}]$

…

D_n: $[l_n^{k1}, h_n^{k1}]$ $[l_n^{k2}, h_n^{k2}]$ … $[l_n^{kmn}, h_n^{kmn}]$

Thus not all fields of the matrix are filled out. Each row i contains k_{mi} ranges. The sample matrix is modified to:

D_1: [3.0, 3.5] [4.5, 5.0] [5.5, 6.0] [7.0, 7.5] [8.0, 8.5] [9.0, 9.5]

D_2: [6.0, 8.0]

It is important that all combinations of ranges of all dimensions are valid query boxes for the query. This comes from the fact that intervals are created in all dimensions as cartesian product (*Create Range* operator of the abstract execution plan, see Figure 9-4). We enumerate the query boxes by combination of the ranges in the dimensions. The ranges in the dimensions are numbered starting from *0* to *k-1*, if *k* ranges exist for that dimension:

D_1: R_1^1 R_1^2 ... R_1^{k1}

D_2: R_2^1 R_2^2 ... R_2^{k2}

…

D_n: R_n^1 R_n^2 ... R_n^{kn}

The enumeration creates a list *LQB* of n-dimensional query boxes:

```
LQB = { }
FOR d1 = 0 to k1
     FOR d2 = 0 to k2
     …
          FOR dn = 0 TO kn
               LQB = LQB ∪ {(R1^d1, R2^d2, …, Rk^dk)}
          NEXT kn
     …
     NEXT k2
NEXT k1
```

The elements of *LQB* are numbered from *0* to $k_1*k_2*\ldots*k_n$ *- 1*. For the sample query, the enumeration order of the query boxes corresponds to the order of the query boxes: *LQB = { QB^1, QB^2, QB^3, QB^4, QB^5, QB^6 }*. Figure 10-20 shows the enumeration of the query boxes with two dimensions.

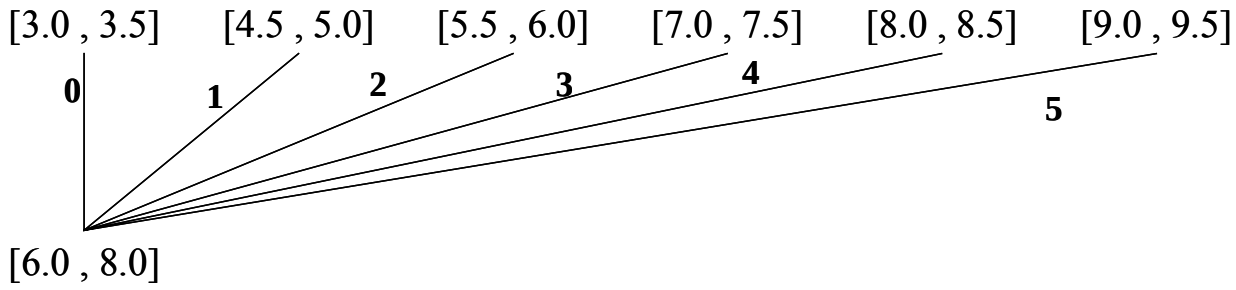


Figure 10-20: Query Box Enumeration

Each page p_k has a list of query boxes PL_k that intersect the page:

$PL_1 = \{0\}$, $PL_2 = \{0, 1\}$, $PL_3 = \{0, 1\}$, $PL_4 = \{2, 3, 4, 5\}$,

$PL_5 = \{2, 3, 4, 5\}$

Given a page p_k where we have to post-filter the tuples, we proceed in the following way: We iteratively check for each tuple *t* of p_k, if one of the query boxes of PL_j contains *t*. If such a query box is found, we add the tuple to the result set (see Algorithm 10-5).

Algorithm 10-5 (Query Box Testing Algorithm):

```
p = {t1, t2, …, tj}
FOR EACH ti
     FOR EACH nqb ∈ PLp
          IF nqb CONTAINS ti
               R = R ∪ ti
               break;
```

The algorithm for query box testing has a maximum of $j * \overline{n_{qb}}$ comparisons, where *j* is the number of tuples that are tested and $\overline{n_{qb}}$ is the average number of query boxes that intersect one page. The *CONTAINS* operation checks whether tuple t_i is inside query box n_{qb}. n_{qb} is the number of the query box.

An optimization is possible for the query box testing. We do not test complete query boxes, but ranges in the dimensions. We begin with dimension D_1 and test, to which range $t.d_1$ belongs. $t.d_1$ denotes the attribute d_1 of t contributing to dimension D_1. Note that there is only one range, where d_1 belongs to. If this range exists and contributes to a valid query box for the page, we proceed in the same way for the remaining dimensions. If we find for all dimensions a corresponding range, the tuple belongs to the result. The search of the ranges can be done very efficiently by binary search.

10.3.2.4 Optimization w.r.t. Order of Query Boxes

In some situations, a high number of query boxes hits almost the same leaf pages (regions) of the UB-Tree. In such a case, the page number collecting algorithm traverses the UB-Tree index part for each query box and accesses and fetches the index pages multiple times. The final result can be very small compared to the number of index page accesses. The algorithm therefore is CPU bound (the index pages are usually in the cache). We can reduce the index part traversal by modifying the range query algorithm in the following way.

We sort the query boxes w.r.t. z-order of *ql* and process not complete query boxes, but split up query boxes into regions. This means that the algorithm is not performed for each query box. We collect each region only once. Since multiple query boxes can intersect the same regions, we can skip the calculation and thus the UB-Tree index access for all other query boxes for this region.

For the new algorithm we need a description of query boxes. A query box is specified by *QB = (ql, qh, α, qnr)*, where *ql* is the lower and *qh* the upper corner of the query box, α is a running z-address initialized with $\alpha = Z(ql)$ and *qnr* specifies the query box number. *QBL* is a sorted list of query boxes $QBL=\{QB_i\}$. The query boxes are sorted w.r.t. $\alpha = Z(ql)$. The algorithm is shown in Algorithm 10-6.

Algorithm 10-6 (Query Box Algorithm with Query Box Splitting):

```
μ = -∞, PNO = { }
WHILE QBL ≠ { }
     QB = pop(QBL)
     IF QB.α > μ
          PNO = PNO ∪ getPno(QB.α)
          μ = nextSep(QB.α)
     QB.α = getNextJumpIn(μ , QB)
     IF QB.α ≤ Z(QB.qh)
          push(QBL, QB)
```

We assume that the query box list is organized as heap ([Knu99]) with the operations `head`, `pop`, and `push`. `head` qualifies the first (smallest w.r.t. heap order) element of the heap. The expression `head(QBL).α` returns the value of α (running z-address) of the first query box in the list. `pop` deletes the first element of the heap and the next larger element is on the first position of the heap[25]. `push` inserts a new element *e* into the heap. If *e* is the smallest element, it is on top of the heap and can be accessed by the `head` operation.

In the query box algorithm, we use μ for the current separator of the index part. μ is initialized with -∞. *PNO* is again the list of page numbers and is the empty set at the beginning. We run the calculations as long as there are query boxes left in *QBL*. We remove the first query box from the heap (`pop`), in order to resort the heap with the new computed *next jump in* address of the next separator. First we check if the anchor α of the query box is larger than the current separator. This is an

[25] An implementation for this operation is to replace the first element with the last element in the heap array and resort the heap by exchanging elements in the heap (logarithmic complexity due to the organization of heaps).

optimization, because the regions for all query boxes with the anchor α smaller than the current separator have already been inserted into *PNO*.

If α is larger than the current separator, we compute the page number and add it to *PNO*. The function `nextSep` returns the following separator in the UB-Tree index part. We calculate the address of the next jump in (`getNextJumpIn`) for μ and store it as the anchor α of the query box. If α is larger than the right corner of the query box, i.e., *Z(qh)*, the query box is finished. Otherwise we re-insert the query box into the heap again (`push`) to the position w.r.t. the computed *next jump in* address.

If one or more query boxes have smaller anchors than the current separator, the region of the anchor can be skipped and they are re-inserted with a new anchor (resulting from the next jump in calculation).

This algorithm reduces the number of UB-Tree index page operations significantly if many query boxes occur that intersect the same data pages.

The idea of the algorithm is very similar to [Fen98] and the basic idea of sorting the query boxes w.r.t. the z-order of the query boxes is taken from the proposed algorithm. However, predicate testing is different in the meaning that the predicate optimization as described before is done.

10.3.2.5 Impact to the Operator Tree

With the original handling of query boxes with MHC, the operator tree is designed to process each query box in a nested loop (`TIMES` node). The left son is an operator subtree which delivers the set of *n*-dimensional query boxes. Its right son is the access to the fact table with the current query box actually delivered by the left son. In this way, a completely independent processing of each query box is implemented (see Figure 10-21).

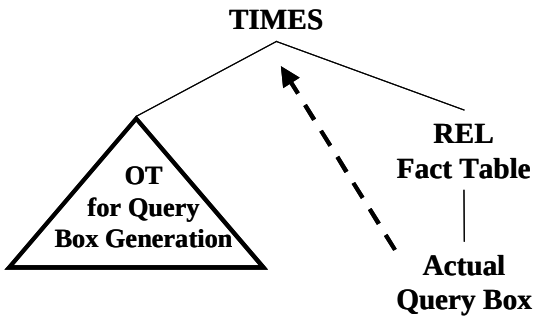


Figure 10-21: Operator Tree for handling Query Boxes sequentially

With the new technique, the computation of the query boxes is arranged under the `REL` node (which does the fact table access) – in addition, the query boxes are not explicitly constructed but only the constituents (one dimensional intervals) are constructed (see Figure 10-22). In this way, we only get $m_1 + m_2 + \ldots + m_n$ elements instead of $m_1 * m_2 * \ldots * m_n$ query boxes, where m_k is the number of intervals resulting from the predicates of dimension D_m.

Of course, the nature of this improvement is closely coupled with the independence of the one-dimensional intervals among the dimensions, in other words, the explicit construction of query boxes as it has been done introduces a lot of redundancy.

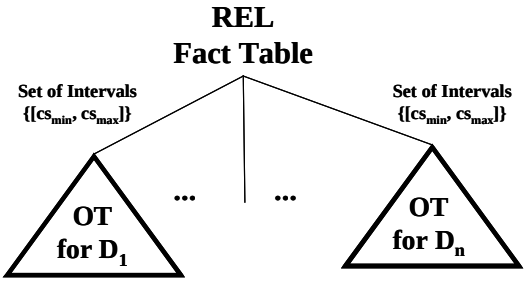


Figure 10-22: Operator Tree for collecting Query Boxes with Predicate Matrix

10.3.2.6 Remarks

The optimization of the multi-query-box algorithm speeds up handling of many query boxes significantly. The division into two phases, i.e., the page number collecting phase and the evaluation of the tuples, allows for more parallelism. The first phase uses the range query algorithm and returns a list of page numbers on which tuples for the query boxes are located. If a page number has been found, the storage manager fetches the corresponding page either from the cache or from secondary storage. This can be done in parallel to computing the remaining page numbers. The processing of each retrieved page also can be done in parallel.

Thus, we have a three layer parallelism. Depending on the hardware environment and the implementation of the DBMS this parallelism can speed up the query processing step significantly. We have implemented a separate I/O process to perform page fetching and tuple evaluation on the pages in parallel. However, first measurements showed that this optimization has only minor effects on speeding up the overall query processing (probably caused by the implementation).

Another optimization can be applied, if the *next jump out* algorithm ([Ram02]) is available. When processing the UB-Tree range query algorithm, we can check whether the page is completely within the query box without accessing the tuples of the page. The tuples of such a page are not post-filtered. Further investigation is necessary to see how likely a page is completely within a query box.

Note that the optimization, especially the third optimization (Section 10.3.2.3) is adapted for star queries on an MHC designed schema. The optimizer has to generate the corresponding execution plan with multiple disjoint ranges for every dimension. The proposed algorithm is **not** a general multiple query box algorithm to handle many query boxes.

10.3.3 Merging Intervals

The optimization of the query box algorithm as discussed in Section 10.3.2 suffers, if the number of predicates to test for the pages is very large. Especially if many query boxes are empty, this method is inefficient. In this case, all tuples on the page are tested with many predicates (all query boxes intersecting this page). Such problems occur especially, if there are many dimension tuples resulting from the dimension restrictions that do not occur in the fact table. Also very sparse fact tables can cause such problems.

Note that in this case, secondary indexes on the fact table dimension attributes prevent from testing on non-existent dimension values. In secondary indexes only existing tuples are stored.

In order to reduce the number of predicates to test, we can merge query boxes. Instead of testing for a large number of predicates, we test only for one multidimensional query box. The optimal case is to merge query boxes such that no additional tuples lie within the new larger query box. In Figure 10-23 the right enlarged query box contains the same number of tuples as the original three single query boxes, i.e., one tuple. The left enlarged query box, however, contains an additional tuple between the middle and right single query boxes. This tuple must be removed after result set evaluation.

Removing the superfluous tuples can be delayed to the residual join step in the overall star query execution plan. We can pre-group the super set of tuples and post-filter in the residual join (similar to secondary dimensions). The predicate of the dimension must be evaluated again and must be used as additional join restriction. The grouping order of pre-grouping (see Section 9.4.1.2) is the minimum of the conventional grouping order and the minimal hierarchy level of the dimension predicate, because the granularity of post-filtering must be suitable for the dimension predicate. This kind of optimization is only possible for range query algorithm in the context of star queries with MHC.

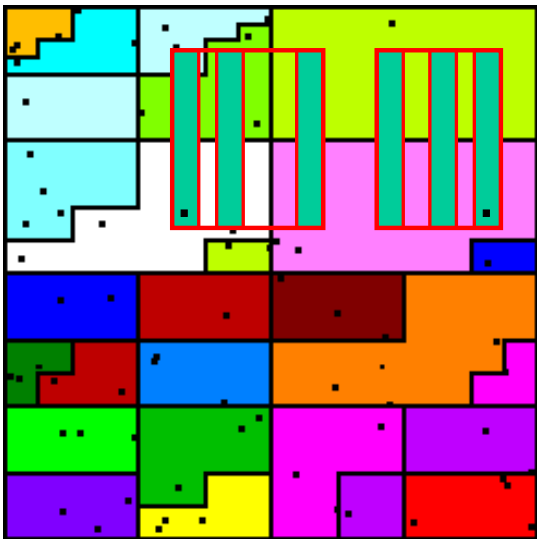


Figure 10-23: Two dimensional UB-Tree with merged Query Boxes

The decision which query boxes to merge depends on the data stored in the UB-Tree. For this purpose, statistics about the data distribution are necessary. If no statistics are available, as for the Transbase® implementation, we have to decide w.r.t. rules at query compile time. In general, the goal is to get a minimum *m* of merged query boxes from a number *n* of original query boxes that intersect the same number of pages and do not contain significantly more tuples than the original query boxes. The overhead for handling the superfluous tuples should be less than the effort to perform post-filtering of many query boxes.

One strategy is to combine query boxes on compound surrogate level, i.e., to analyze the compound surrogates that are the edges of the intervals of the query boxes. With an ordered list of intervals for each dimension, we can decide, which intervals should be merged. For example, compound surrogates that are distinct in some of the lower hierarchy levels are likely to form intervals that are near together. Also the number of tuples that are superfluous after merging such intervals, is likely smaller than for intervals with larger distances.

For the decision which intervals to merge, we introduce a distance arithmetic *d(a, b)* where *a* and *b* are compound surrogates and *a* is the upper bound of interval I_1 and *b* is the lower bound of interval I_2. The distance is $b - a$, i.e., the number of compound surrogates that can lie between *a* and *b*. Since not all compound surrogates are used in the hierarchy, such a computation can be very imprecise.

We suggest to compute the distance from the number of existing compound surrogates between *a* and *b*. This number can be computed (or estimated) efficiently by processing the *DXcs* index (similar to the `CS2IVAL` operator). The distance can be computed on the fly when processing the set of compound surrogate intervals in the `CS2IVAL` operator. The overhead of the distance computation is comparably low.

The merging process merges intervals with small distances. The decision depends on hierarchy properties such as height of the hierarchy, fan-out etc. Also some statistical properties can be used, e.g., the distribution of the hierarchy paths. Merging is done by replacing a number of intervals by a

larger interval in the predicate structure. The range query algorithm is the same as described in Section 10.3.2.

The proposed method is a kind of dynamic optimization without modifying the operator tree. If after the range query generation phase of the dimensions the number of query boxes is larger than a threshold, we perform interval merging and reduce the number of query boxes. However, pre-grouping and residual join with predicate evaluation must be foreseen in the operator tree, in order to do correct post-filtering.

Note that merging query boxes is orthogonal to the optimization of multiple query boxes as described in Section 10.3.2. The alternative query box algorithm can be used in addition to merging query boxes.

10.3.4 Measurements

The measurements of this section address the multi query box optimization. We performed a set of queries with

- the original query box algorithm (sequential processing of query boxes): *Original*
- the optimized query box algorithm with the collection of pages and post-filtering optimization as implemented in Transbase®: *Optimized*

The queries are executed on the Sales-DW (see Section 10.8) on a two processor PC Pentium II, 400 MHz, with 768 MB RAM and a SCSI hard disk. Operating system is Windows 2000. The queries cover basically the same fact table data and the corresponding fact table leaf pages are in the DBMS cache. Thus, we concentrate on the performance of the query box algorithms themselves. The template for all queries is the same:

```
SELECT COUNT(*)
FROM fact f, product prd, calendar cal, warehouse zap
WHERE f.cal_trans_dwh_key=cal.dwh_key and f.prd_dwh_key=prd.dwh_key
  and f.whs_issue_dwh_key=zap.dwh_key
  and zap.country_code = '1' and zap.geodept_code='1'
  and zap.county_code='8'
  and prd.category_code='21' and prd.group_code = '21001'
  and cal.year = '1999' and half_year = '1h1999'
  and quarter = '2q1999'and month = '06/1999'
  and day_of_week='Thursday'
```

With the feature predicate `day_of_week='Thursday'`, the hierarchical restriction is split into distinct compound surrogates *cs*, e.g., four *cs* values in the previous query. In order to increase the number of intervals, we modified the feature predicate to `(day_of_week = 'Tuesday' OR day_of_week ='Thursday')`. This leads to eight distinct *cs* values. With a modification of the hierarchical restriction for the *Calendar* dimension, we further increase the number of intervals, e.g., a restriction to `year = '1999'` only leads to 52 ("Thursday") resp. 104 ("Tuesday or Thursday") intervals. For further increase of the number of intervals, we replaced the hierarchical restriction on the *Product* dimension by `prd.dwh_key/<a>*<a>=prd.dwh_key`. This results to $\frac{1}{a} * |\mathrm{Pr}oduct|$ intervals assuming that dwh_key is numbered from 1 to 27.929. |*Product*| is the number of records in the *Product* dimension. For *a=2* means that 13.964 intervals are generated. We used the values 4, 8, 16, 32 for the queries. In combination with the weekday feature restriction, we have up to 86.008 query boxes.

The results are shown in Figure 10-24. Both algorithms are linear w.r.t. the number of query boxes, where the *optimized* algorithm is always faster by a factor of two to three.

The original algorithm processes the leaf pages of the UB-Tree for each query box, triggered by the UB-Tree range query algorithm. This results in loading the leaf pages multiple times, if more than one query box intersects the same UB-Tree regions (see Section 10.3.1). In contrast, the optimized algorithm only stores the page numbers and processes each page number once for all query boxes.

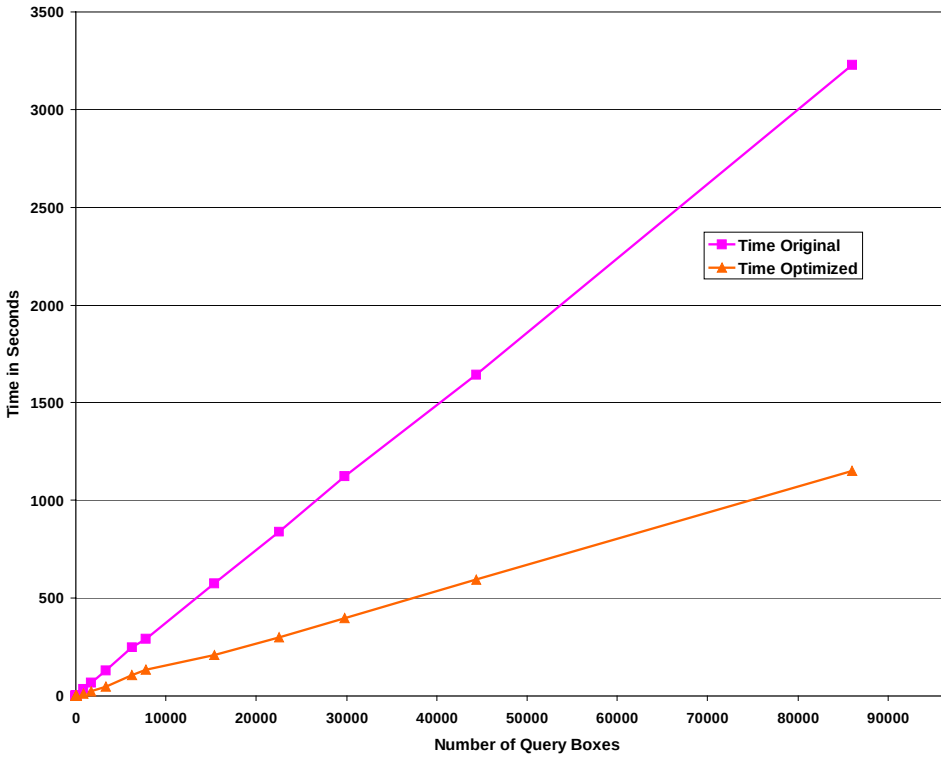


Figure 10-24: Comparison Multi Query Box Algorithm

Figure 10-25 shows the comparison of the numbers of read pages for both algorithms. The total number of pages accessed by the original algorithm (*TotalPagesOriginal*) is up to 25 million, whereas for the optimized algorithm (*TotalPagesOptimized*) it is 3,2 million. For the original algorithm, the number of index (*IndexPagesOriginal*) and leaf page accesses (*LeafPagesOriginal*) both is high. For the optimized algorithm, the number of index pages accesses (*IndexPagesOptimized*) is high, but the number of leaf pages (*LeafPagesOptimized*) is very low because of the unique access of leaf pages. Thus, the overall number of accessed pages is almost the same as the number of accessed index pages. The index pages often are cached and therefore are not expensive to access.

In the overall, the optimized algorithm accesses an order of magnitude less pages. Mainly index pages are read, which are usually cached in memory. For multi-user environment, the optimized algorithm will be even more beneficial, because leaf pages can be swapped out of the DBMS cache (depending on the cache strategies).

However, there is a drawback for the optimized algorithm. The page numbers are stored within a list in memory. If the list becomes too long to hold it in memory, it has to be written to a temporary file. Sort operations and sequential read is done on this file and is slower compared to main memory operations. This holds for very large number of query boxes, e.g., 1 million query boxes require $1*10^6*4$ Bytes (page numbers are integer values). Such large number of query boxes occur very seldom. The customers running Transbase® with MHC usually have queries with a maximum of 10.000 query boxes. Most queries have between 1 and 100 query boxes.

Note that the number of result tuples often does not correlate to the number of query boxes. Usually, in queries with a large number of queries many query boxes are empty. For example, the last query with 86.008 query boxes qualifies 600 fact table tuples. There are still 3,2 million index pagge accesses

necessary, in order to evaluate 600 tuples! With secondary indexes, such a query would be faster, because the most effort is to evaluate the index intersection. The materialization requires at most 600 fact table page accesses. When executing the same query with an other commercial DBMS with bitmap indexes (see Section 10.9.4), the time is 16 seconds.

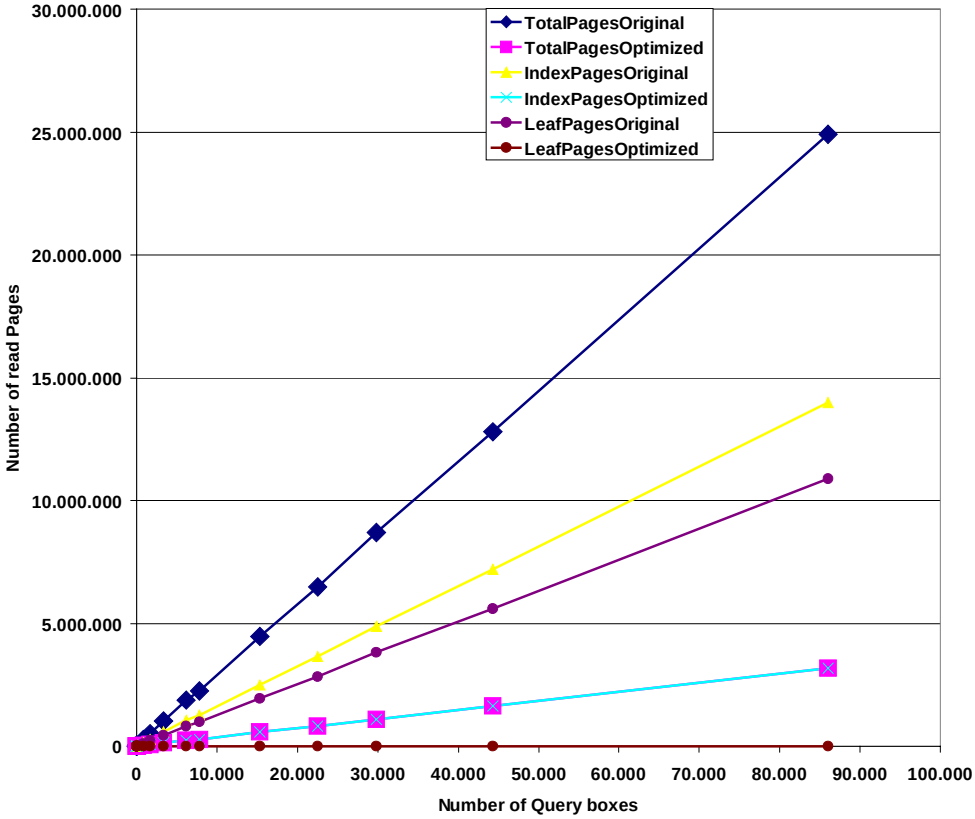


Figure 10-25: Comparison of Page Accesses

When merging query boxes, we can reduce the 86.008 query boxes to below 100 and therefore reduce the time (and number of page accesses) to evaluate the query by several orders of magnitude.

10.3.5 Remarks

In the previous sections, we sometimes mentioned *dynamic optimization*. Dynamic optimization is a method to modify the query execution operator tree during the execution of the operator tree. This is a very interesting and quite new topic and for the best of our knowledge not much investigation has been done so far. However, we believe that this will be one of the future methods to do query optimization.

The problem is that statistics turn out to be not sufficient to choose the most appropriate operator trees for complex queries. After the first join, it is not possible any more to estimate the number of resulting tuples and large deviations occur. Thus, the generated operator tree often is not optmimal w.r.t. the costs of the query. A dynamic behavior of the query optimizer with changes of some basic operators can adapt optimally to query execution. So far, most DBMS (inclusive Transbase®) do not support dynamic optimization. Beyond the mentioned approaches of dynamic optimization, we have to investigate further conditions and adaptions.

10.4 MHC and Partitioning of UB-Tree Data

The partitioning of a UB-Tree usually adapts very good to the data distribution (see Figure 10-26). This means, that the partitioning of the regions (disk pages) of the UB-Tree is very close to the distribution of the data. Multidimensional query boxes that approximate the partitioning grid thus are very performant, because the number of accessed pages is almost minimal.

However, for certain data distributions and queries the so called puff pastry effect increases the number of loaded pages dramatically (see [Mar99]). In this section we discuss the distribution of the regions and data of a MHC organized fact table.

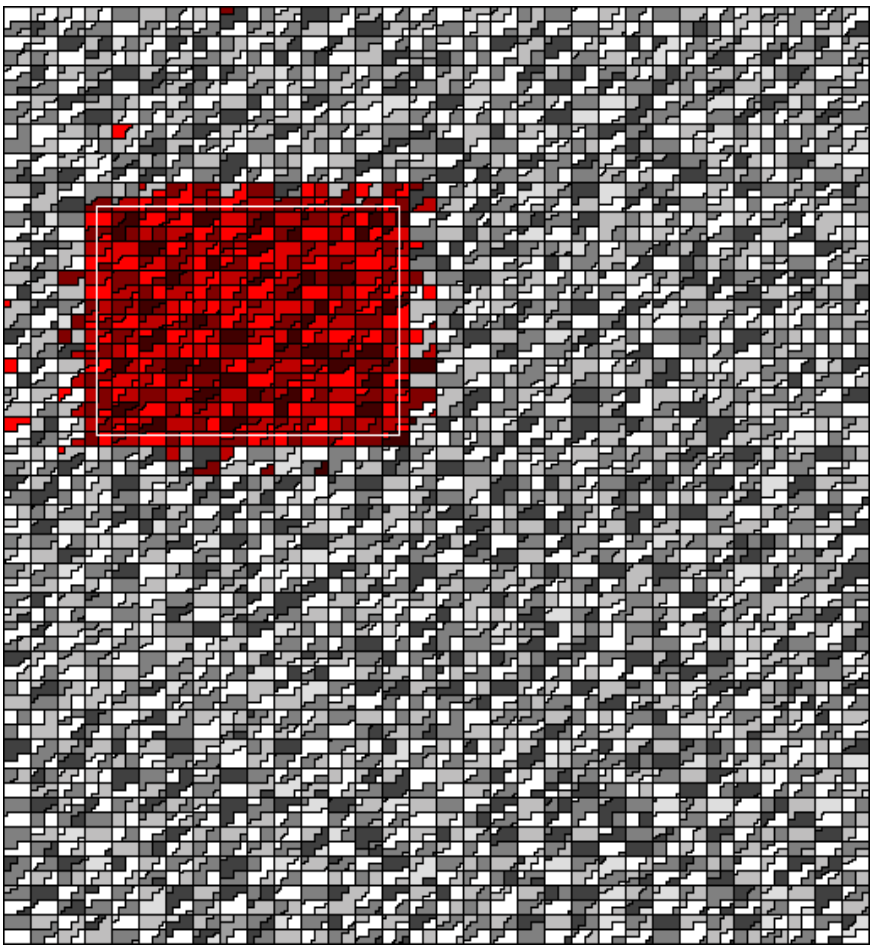


Figure 10-26: Partitioning and Range Query for UB-Trees

10.4.1 Partitioning

For MHC, the dimensions of the UB-Tree usually are compound surrogates. A compound surrogate represents a hierarchy path and consists of surrogates, one for each level of the hierarchy (see Section 5.3). Each surrogate is encoded by a bit string. The number of bits reserved for the surrogate (hierarchy level) depends on the expected maximum number of siblings for this level. Since the maximum number of siblings often is higher than the number of used bit combinations for an instance of a hierarchy, a large number of bit combinations is unused, so called *holes*. The occupation of the bit combinations depends on the order of insertion of the hierarchy paths (see Section 5.3.3). Usually, the used bit combinations form an interval from 0 to k, if the hierarchy node has k siblings[26]. All bit combinations, i.e., from $k+1$ to n, where $n = 2^j-1$ for j reserved bits for the hierarchy level, remain

[26] All siblings have the same father hierarchy node.

unused. This characteristic of the distribution applies to all dimensions, and thus results in data clusters.

Figure 10-27 shows an example distribution for two dimensions, i.e., the customer and calendar dimensions of the Sales DW. The x-axis represents the customer dimension, the y-axis represents the calendar dimension. The customer dimension contains one country and four departments (the vertical clusters). The calendar dimension contains two years with three quarters and seven months for each year. We use seven months and three quarters for illustration purpose, in order to show the data distribution.

The first department in the country contains most of the data (very small and numerous regions), whereas the distribution w.r.t. the time is quite uniform.

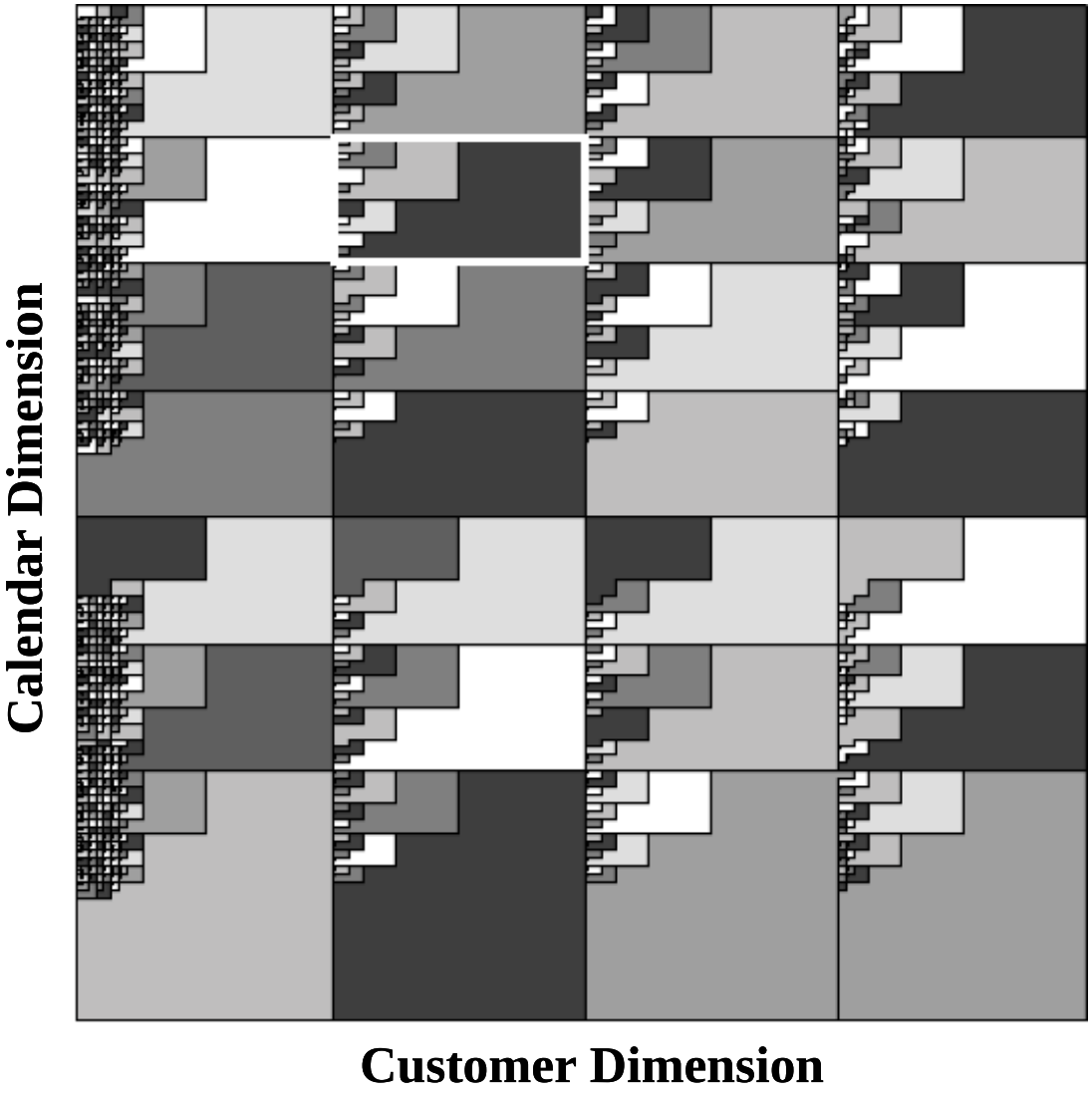


Figure 10-27: MHC Data Partitioning

A query on a MHC organized star schema generates intervals representing prefixes of the hierarchy. These prefixes of the hierarchy form intervals including complete clusters (see the marked rectangle in Figure 10-27). Such a query box is optimal w.r.t. the number of pages intersected by the query box.

The concept of MHC clusters the fact table, in order to support hierarchical restrictions optimally. This means that the multidimensional query boxes adapt very good to the data distribution of MHC organized data. Conceptually, the query boxes resulting from hierarchical predicates are adapted to the data and thus load a minimal number of pages.

Jump regions can cause additional pages to load (see [Mar99]). A jump region is a region that is not a continuous region in the multidimensional space, but comes from the z-curve that "jumps" into another quarter of the space (see [Mar99] for more details). Also not optimal regions originating from page splitting cause additional pages to load. So called fringes can require loading of additional pages. In Figure 10-28 we show an example where such a page splitting requires the access to one additional page.

We can adapt the page split algorithm, in order to get rectangular regions for MHC organized data. This can easily be done by modifying the computation of the split separators. The split points are calculated considering the bit positions of the hierarchy levels. Then no fringes occur. See [Mar99] for more information about modifying the split algorithm.

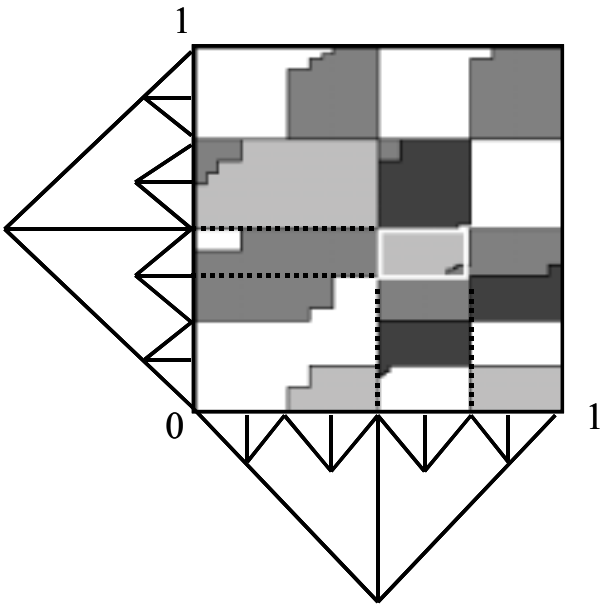


Figure 10-28: Query Box intersecting two Regions and Predicate Grid

In Figure 10-28 we show a two dimensional space with a number of z-regions. Possible query boxes from hierarchical predicates on a MHC organized UB-Tree are defined along the grid originating from the recursive partitioning of the data space. These intervals are shown in Figure 10-28 on the left and on the bottom side of the figure. The space is divided into 64 (8 x 8) rectangles. The query box is defined by [4/8 ; 6/8] in *x* and [3/8 ; 4/8] in *y* direction, if the domain is [0 ; 1] for both dimensions.

An MHC predicate is a multidimensional interval originating from all dimensions of the UB-Tree. Each dimension is represented by a compound surrogate consisting of surrogates for each hierarchy level. Each hierarchy level is represented by a bit combination, i.e., each hierarchy level has 2^k sub-trees, each of which is represented by an interval, if *k* bits are used for the mapping of the hierarchy level. In Figure 10-28, the top level is partitioned into two intervals, the next level into four intervals. The remaining levels are not shown in the figure. Thus, when restricting the *Y* dimension by h^t = a AND h^{t-1} = b and the *X* dimension by h^t = d AND h^{t-1} BETWEEN e AND f, the query box of Figure 10-28 is generated.

Figure 10-29: Impossible Query Box with MHC Predicate

Figure 10-29 shows a query box that cannot be defined by a hierarchical MHC interval, because the ranges of the query box exceed the hierarchy levels. This is the reason why the clustering of an MHC organized fact table is good for queries generated by MHC hierarchical predicates. In Section 10.4.4 we show with experimental results that the interval generation method for range queries is almost optimal using hierarchical prefix paths of compound surrogates.

10.4.2 Hierarchical Clustering Effect

The partitioning of data in the multidimensional space using the z-curve is restricted to a number of bits, i.e., a prefix of the z-value (see [Mar99] for more details). The bit strings of all dimensions are interleaved.

The compound surrogates representing the dimensions are constructed from the surrogates of the hierarchy levels. For the Sales DW, the compound surrogates consist of the components as listed in Figure 10-30. For each dimension product, warehouse, calendar, transaction and sales payment, we list the bits, where p_i has the values *1* or *0* etc.

$$cs_{product} = \underbrace{p_{22}p_{21}p_{20}p_{19}p_{18}}_{h^3}\underbrace{p_{17}p_{16}p_{15}p_{14}p_{13}p_{12}}_{h^2}\underbrace{p_{11}p_{10}p_9...p_1}_{h^1}$$

$$cs_{warehouse} = \underbrace{w_{22}}_{h^6}\underbrace{w_{21}w_{20}w_{19}}_{h^5}\underbrace{w_{18}w_{17}w_{16}}_{h^4}\underbrace{w_{15}w_{14}w_{13}}_{h^3}\underbrace{w_{12}w_{11}w_{10}w_9w_8w_7}_{h^2}\underbrace{w_6...\ w_1}_{h^1}$$

$$cs_{calendar} = \underbrace{c_{12}c_{11}c_{10}}_{h^5}\underbrace{c_9}_{h^4}\underbrace{c_8}_{h^3}\underbrace{c_7c_6}_{h^2}\underbrace{c_5c_4c_3c_2c_1}_{h^1}$$

$$cs_{transaction} = \underbrace{t_7\ t_6\ t_5\ t_4\ t_3\ t_2\ t_1}_{h^1}$$

$$cs_{salespayment} = \underbrace{s_6\ s_5\ s_4\ s_3\ s_2\ s_1}_{h^1}$$

Figure 10-30: Compound Surrogates for the Dimensions of the Sales DW

The z-value for the five dimensions is calculated by interleaving the bits of the compound surrogates of all dimensions starting with the longest compound surrogate:

$$z = p_{22}w_{22}c_{12}t_7s_6\ p_{21}w_{21}c_{11}t_6s_5\ p_{20}w_{20}c_{10}t_5s_4\ p_{19}w_{19}c_9t_4s_3\ p_{18}w_{18}c_8t_3s_2\ p_{17}w_{17}c_7t_2s_1\ p_{16}w_{16}c_6t_1\ p_{15}w_{15}c_5\ p_{14}w_{14}c_4\ p_{13}w_{13}c_3\ p_{12}w_{12}c_2\ p_{11}w_{11}c_1\ p_{10}w_{10}\ p_9w_9\ p_8w_8\ p_7w_7\ p_6w_6\ p_5w_5\ p_4w_4\ p_3w_3\ p_2w_2\ p_1w_1$$

The number *P* of pages necessary to store the data of the fact table determines the number *k* of bits that are used for the physical clustering ([Mar99]):

$$k = \log_2 P$$

In the Sales DW, we have 448.650 data pages of the UB-Tree for the fact table for 8.579.458 tuples. Thus, we can use $k = \log_2 P = \log_2 448650 = 18{,}8$ bits for physical clustering (see Figure 10-31). This means that only 18 of 69 bits are used for clustering. The highest bits of the compound surrogates are preferred leading to a better clustering for the higher hierarchy levels. Figure 10-31 shows which bits are used for clustering.

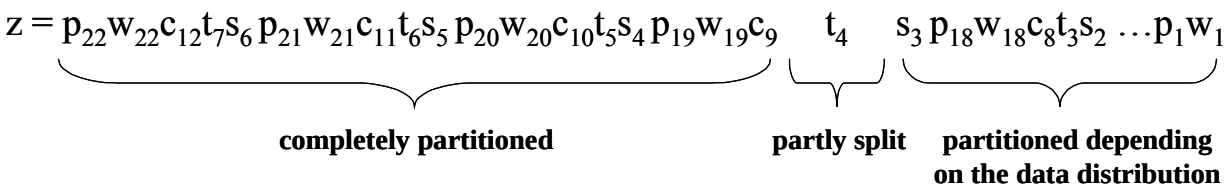


Figure 10-31: Split Levels for the z-Value

The z-value can be divided into three parts. The first part (bits 1 to 18) contributes to the clustering independently on the data distribution (see [Mar99]). The second part (bit 19, i.e., t_4) is sometimes split and sometimes not, i.e., it can be used for clustering in some situations. The remaining bits (bit 20 to 69) are used, if the data is not uniformly distributed. Otherwise these bits will not contribute to the partitioning of the universe.

This means that for the product dimension only a subset (the first four bits of five) of h^3 (highest hierarchy level) is granted to be used for partitioning. For the warehouse dimension, h^6 and h^5, for calendar, h^5 and h^4 are used for partitioning. The flat dimensions transaction and sales payment can use the first three bits.

In order to use h_3 for the product dimension completely, we have to increase the number of pages to at least $2^{21}=2.097.152$. This means that the number of tuples must be at least *40.103.483*, i.e., about five times of the current amount, if the page size remains at 8 KB.

Since in real data warehouses, however, the dimensions are not independent, i.e., they are correlated highly, even these lower bits are used for the physical clustering. Indeed, for the Sales DW we have an average split level for the z-value of 49 bits. This means that the first 49 bits contribute to the clustering! The following z-value shows which hierarchy levels contribute to the clustering:

$$z = p_{22}w_{22}c_{12}t_7s_6\ p_{21}w_{21}c_{11}t_6s_5\ p_{20}w_{20}c_{10}t_5s_4\ p_{19}w_{19}c_9t_4s_3\ p_{18}w_{18}c_8t_3s_2\ p_{17}w_{17}c_7t_2s_1\ p_{16}w_{16}c_6t_1\ p_{15}w_{15}c_5\ p_{14}w_{14}c_4\ p_{13}w_{13}c_3\ p_{12}w_{12}c_2\ p_{11}w_{11}c_1 \mid p_{10}w_{10}\ p_9w_9\ p_8w_8\ p_7w_7\ p_6w_6\ p_5w_5\ p_4w_4\ p_3w_3\ p_2w_2\ p_1w_1$$

The dimensions *calendar*, *transaction* and *sales payment* contribute with the clustering in all hierarchy levels, whereas the lower levels of the dimensions *product* (h_1) and *warehouse* (h_2 and h_1) do not (see Figure 10-30). For the impact of correlation, split level and clustering properties, refer to [Ram02].

Usually not all bit combinations are used for all instances of the hierarchy levels, because the number of bits reserved for one hierarchy level depends on the maximum number of siblings of all hierarchy level instances of this hierarchy level. Most hierarchies, however, are not distributed uniformly, i.e.,

the hierarchy level instances vary in the number of siblings. Thus, for most hierarchy level instances, not all bit combinations are used.

The calculation of the bit combinations $b_k, b_{k-1}, \ldots, b_0$ starts with the bit combination 00..00, the next bit combination is 00..01 etc. (see Section 5.3.3). Therefore, the highest bits are always 0. Since a restricted number of bits contributes to the physical clustering, e.g., $b_k, b_{k-1}, \ldots, b_{k-j}$, these bits should be used for clustering. Thus, a different bit calculation method can improve clustering. The goal is to use as short common prefixes as possible. Assume that we use the bits $b_k, b_{k-1}, \ldots, b_{k-j}$ for clustering. The remaining bits $b_{k-j-1}, b_{k-j-2}, \ldots, b_0$ are not used. Then we assign the bit combinations to the *used bits* in the following way: 000..00, 100..00, 110..00 etc. If there is still the necessity of further bit combinations, we proceed with 010..00, 1010..00 etc. After all bit combinations of the used bits for clustering have been assigned, we have to distribute further bit combinations to the non-used bits.

10.4.3 Quality of MHC Partitioning

The quality of the UB-Tree w.r.t. the application and the queries can be seen when examining the number of tuples per page contributing to the fact table result set. If a large fraction of the tuples belongs to the result set, the partitioning is good, if only one (or even none) tuple of a page belongs to the result set, there is no advantage compared to non-clustered fact tables (or even a disadvantage).

The average number of tuples per page is the number of tuples in the fact table divided by the number of UB-Tree leaf pages: $\frac{8.579.458}{448.650} = 19{,}12$

This means that most pages contain 19 tuples and some pages contain 20 tuples. The average number of tuples that belong to the result set is calculated by the number n_r of tuples of the fact table result set divided by the number of pages n_p that are loaded to evaluate the multidimensional query box.

The clustering factor is the number of result set tuples divided by the number of loaded tuples:

$$\text{clusteringFactor} = \frac{\#\text{ResultTuples}}{\#\text{LoadedTuples}}$$

Since we do not have the number of loaded tuples, we use the number of leaf pages loaded, in order to estimate the fact table result tuples:

$$\text{clusteringFactor} = \frac{\#\text{ResultTuples}}{\#\text{loadedPages} * \text{avgTuplesPerPage}}$$

This calculation is not exact, because the number of actual tuples per page is not the same as the number of average tuples per page. For a sufficient number of loaded pages, however, the average number of tuples per page adapts well to the computed average number.

In order to show the clustering factor, we executed a query suite with 1061 queries with different fact table selectivities from 0% to 9%. We compare the average number of tuples on the loaded leaf pages of the UB-Tree (tuples per page) contributing to the fact table result set. Table 10-1 shows the number of average tuples per page and the average clustering factor in percent. 50% of the queries hit between 1,8 and 13,3 tuples per page, which is a clustering factor between 9,4 and 69,8 percent of the tuples per page. The median number is 7,8 (40,9 %). The comparable low page clustering factor comes from the distribution of the queries. Most queries have a very small result set: 50% of all queries have a fact table selectivity from 0,02% to 0,46 %, i.e., the number of fact table result tuples lies between 1.708 and 39.726. The median is 0,12%. For small selectivities, only a small number of pages is loaded. The pages can contain only a very small number of tuples contributing to the result set because of the ranges of the query boxes. For larger query boxes, the page clustering factor improves.

	avg. result tuples per page	avg. Clustering Factor in %
Minimum	0,00	0,02
1st Quartile	1,80	9,39
3rd Quartile	13,34	69,74
Maximum	20,60	107,71
Average	5,86	30,64
Median	7,83	40,94
Standard Deviation	6,53	34,16

Table 10-1: Clustering Factor

The maximum number of tuples is 20,60, which is larger than the page capacity. This comes from the fact that not all pages contain exact the same number of tuples. If pages are loaded that contain 20 tuples instead of 19, the page clustering factor is more than 100%.

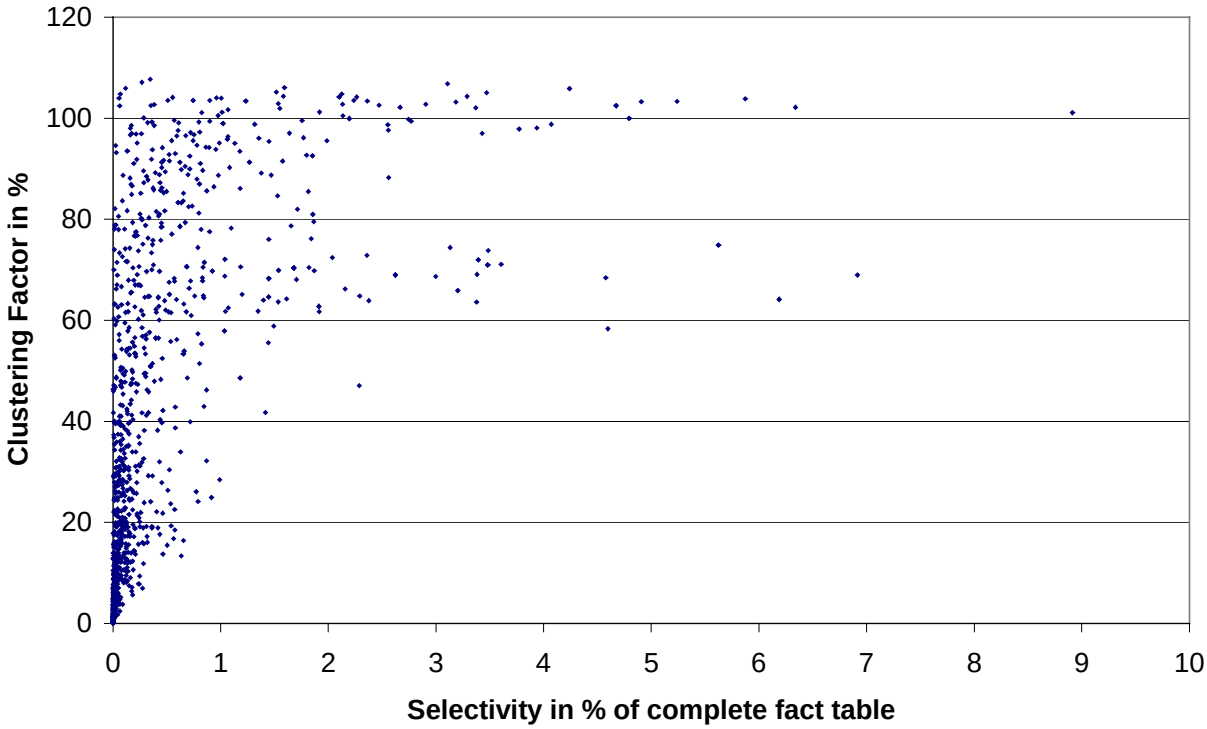


Figure 10-32: Clustering Factor of the 1061 Query Suite

Figure 10-32 shows the page clustering factor (in %) depending on the selectivity of the query boxes for the 1061 queries. Most queries are between 0% and 0,2% and have a clustering factor between 0% and 40%. It turns out that queries with more result set tuples, i.e., selectivities larger than 1%, have higher clustering factors (from 40% to 100%). Generally speaking, the results indicate that the higher the number of fact table result tuples is, the better is the page clustering factor.

Queries with small fact table result set size usually do not access many disk pages and are executed within a comparably small amount of time. Large fact table result set sizes require many disk pages to access, but are executed nearly optimally, because a large number of tuples per page contributes to the result set.

Figure 10-33 shows a qualitative comparison between different indexing techniques, i.e., the full table scan, secondary indexes and UB-Tree (with MHC). The UB-Tree is similar to Figure 10-32 and obtained from the experimental results. The full table scan and secondary index are based on considerations about number of pages of the fact table depending on query boxes with selectivities from 0% to 100%.

Since the full table scan reads the complete table, i.e., all pages of the table, the page clustering factor function is linear: $f_{fulltablescan} = (n_r/p_h)/n_p*100$ where n_r is the number of tuples of the result set, p_h is the number of pages hit by the query and $n_p = n_t/p_t$ is the number of tuples per page (n_t is the number of tuples of the table and p_t is the number of pages occupied by the table). Because $p_h = p_t$ for the full table scan, $f_{fulltablescan} = (n_r/p_t)/(n_t/p_t) *100 = a*n_r$, where $a = 100/n_t$ is constant. Thus, the clustering factor is equal to the selectivity.

Assuming that the secondary index has to access one page for each tuple in the result set (non-clustered fact table), the function is constant until a specific selectivity: $f_{secix} = (n_r/p_h)/n_p*100 = (n_r/n_r)/n_p*100 = 100/n_p$, because the number p_h of pages hit is the same as the number n_r of result tuples and p_t is the same for all queries: $100/n_p = 100/19{,}1 = 5{,}2$. If more than 5,2% of the fact table are in the result set, each page contains more than one tuple in the average. The function now is the same as for the full table scan. Note that the assumption that until 5,2%, each result tuple is stored in an own page and requires a separate page access, is a worst case estimation. In general, tuples often have a natural clustering and the constant clustering factor is shiftet to higher percentages.

The UB-Tree page clustering function is growing very fast to almost 100% and then is almost constant. However, there are no worst-case guarantees, only the full table scan is a worst case guarantee. Usually, for degenerated query boxes (e.g., with a point restriction in one dimension) the clustering factor will be bad. For most query boxes, however, the clustering factor is very high even for small fact table result sets.

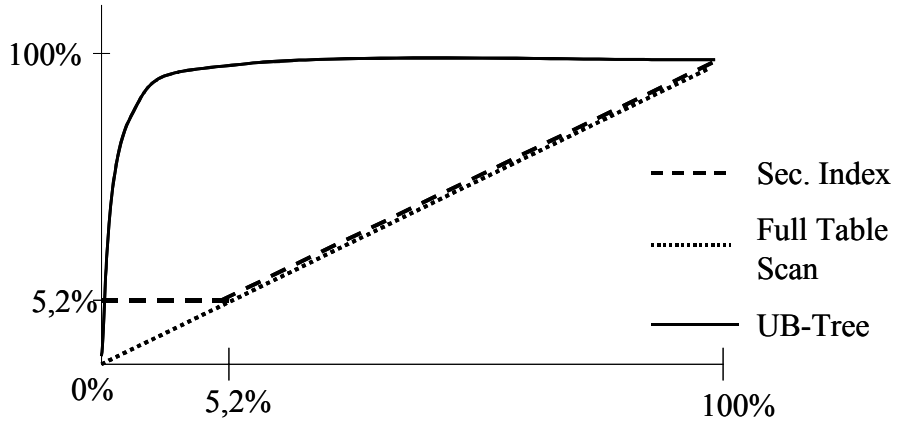


Figure 10-33: Qualitative comparison of Clustering Factor of Secondary Index, Full Table Scan and UB-Tree

In real data warehouses with non-clustered fact tables, the insertion order of the fact table tuples often provides a natural clustering w.r.t. the date dimension. Consider a warehouse scenario where data is loaded daily into the warehouse. If the DBMS provides non-clustered fact tables, the new tuples are appended.

This natural clustering can improve the clustering factor for secondary indexes depending on the predicates of the queries. For example, if the query is restricted to one day (without further predicates),

this method provides a perfect clustering. However, most queries have restrictions in additional dimensions. In such a case, the constant part of the page clustering function is shifted to a higher percentage, but is still far below the clustering factor of the UB-Tree.

10.4.4 Optimizing the Interval Generation

The current implementation of the interval generation builds intervals with the bits of the restricted hierarchy prefix and the remaining bits filled with 0 for the lower and with 1 for the upper bound. This results in the largest interval covering all possible hierarchy sub-trees of the restricted hierarchy prefix (see Section 9.3). We call this method *maximum intervals*. As an alternative, we can shrink the intervals to *minimum intervals*, i.e., the minimum and maximum used bit combinations of the compound surrogates covered by the hierarchical predicate form the interval.

For example, for a date hierarchy with the hierarchy levels year, month and day, over four years, the compound surrogates are *yy.mmmm.ddddd*, e.g., *01.0010.01010* for "*2001/February/10*". The maximum interval for a restriction "2001/February" is [01.0010.00000 ; 01.0010.11111]. If only the days 10, 11, 12, and 13 are used in the hierarchy, the minimum interval is [01.0010.01010 ; 01.0010.01101].

If the partitioning of the universe results in fringes of the regions, the maximal intervals can intersect these fringes and force pages to load, although there are no tuples contributing to the restriction. A minimal interval can prevent from intersecting the fringes and thus from loading superfluous disk pages.

In this section we discuss some experimental results that show the effect of minimizing the intervals. We compare the number of read leaf pages of the UB-Tree with maximum and minimum intervals. The queries are standard OLAP ad-hoc queries on the Sales DW with restrictions varying in the number of dimensions and hierarchy levels and in the selectivity of the qualifying fact table tuples. We use a set of 572 queries.

Minimum	0,00 %
1st Quartile	0,00 %
3rd Quartile	1,61 %
Maximum	22,49 %
Average	1,28 %
Median	0,38 %
Standard Deviation	2,40%

Table 10-2: Number of additional Pages in Percent

Table 10-2 shows the results for the number of pages for the maximum intervals compared to the minimal intervals. The maximum intervals never load less pages than the minimal intervals. 50% of the queries load between 0 and 1,61 percent more pages than the minimal intervals. The average overhead of maximal intervals is 1,28 percent, which is actually not very much. The maximal overhead of 22,49 % comes from a query where 676 leaf pages for the maximum interval and 524 for the minimal interval are read. The query with this result is:

```
SELECT cal.quarter_year, zap.city_code, SUM(val_gross)
FROM fact f, product p, calendar c, warehouse w
WHERE
  f.cal_key=c.dwh_key AND f.prd_key=prd.dwh_key AND
  f.whs_key=w.dwh_key AND c.year = '2001' AND c.half_year = '1h2001'
  AND p.category = '41' AND p.group = '41170' AND
  w.country = 'Greece' AND w.geodept = '3' AND w.county = '1'
group by c.quarter, w.city
```

Summing up, the overhead of maximal intervals compared to minimal intervals is very small. The generation of minimum intervals requires additional effort, because the minimum and maximum compound surrogate must be retrieved from the dimension table. This evaluation is an overhead compared to the maximum interval generation, where only one index access to the *DXh* index is necessary (see Section 10.1.8). When applying the optimized split level calculation as described in Section 10.4.1, the maximal interval generation method loads the same number of pages as the minimal interval method due to rectangular regions.

10.5 Handling of Secondary Dimensions

A fact table contains dimension key attributes and measure attributes. The dimension key attributes form the primary key of the fact table. The UB-Tree is used as primary clustering index with the dimension key attributes as key attributes, i.e., the index attributes of the UB-Tree are identical with the primary key of the fact table. However, for fact tables with a high number of dimensions, we cannot specify every dimension key as UB-Tree index attribute. This leads to a not-unique UB-Tree index. We distinguish between the primary key of the fact table and the clustering key of the UB-Tree. We use the clustering key as primary key for the description of the B*-Tree and UB-Tree algorithms, if not stated otherwise.

In Transbase®, the index attributes of a primary index correspond to the primary key of the table. This is no problem, when dealing with standard composite B*-Tree indexes. The index attributes can be supplemented by the attributes of the primary key without loss of storage or performance. In contrast, the performance of UB-Trees depends on the number of index attributes.

For the UB-Tree and data warehouses, we implemented *NOT UNIQUE primary indexes*. Such indexes do not have a logical primary key. The DDL extension of the `CREATE TABLE` statement is

```
CREATE TABLE t(a INTEGER, b INTEGER, …) KEY NOT UNIQUE IS a, b
```

or

```
CREATE TABLE t(a INTEGER, b INTEGER, …) HCKEY NOT UNIQUE IS a, b
```

for a table with the UB-Tree index (*Hypercube Index*). Some basic algorithms are modified, in order to deal with such indexes:

- Insertion into B*-Tree
- Search on B*-Tree
- Multiple query box algorithm for collecting page numbers

10.5.1 Modification of B*-Tree Algorithms

In a non-unique B*-Tree, the tuples on the leaf page have no total order any more. A new tuple t_{new} can be inserted at any position on the leaf page, such that $o(t_1) \leq o(t_2) \leq \ldots \leq o(t_{new}) \leq \ldots \leq o(t_k)$, where $o(t)$ is the order of the primary key attributes of tuple t. t_1 is the smallest tuple and t_k is the largest tuple on the leaf page.

Consequently, the following search algorithms (search on B*-Tree) must be adapted:

- Search via the separator
- Search within one leaf page

10.5.1.1 Search via the separator

In a B*-Tree, the separators are used to find the path from the root to the leaf page. The index pages in a B*-Tree contain the separators (see Figure 10-34). One index page contains *k* separators and *k+1* pointers to successor pages, i.e., one separator between two pointers. A separator specifies which path has to be followed, in order to find the page with the searched tuple (if existing).

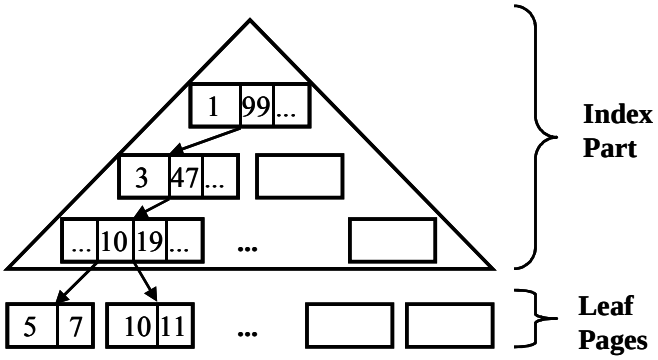


Figure 10-34: Separator Search on Standard B*-Tree

An example for a standard B*-Tree with the separators is shown in Figure 10-34. The index pages contain the index key (separators), the leaf pages contain the complete tuples, in this illustration the key attribute *a* only. If we search for the tuple with *a = 10*, we start at the root page and look for the largest separator that is smaller or equal to 10. The successor index page that is specified by the following pointer is loaded next. We again look for the largest separator that is smaller or equal to the search value and proceed as before. This search is repeated for every level of the index pages in the B*-Tree. The lowest index page points to the leaf pages. In Figure 10-34, the found separator of the lowest index page is 10 and the next pointer points to the leaf page with the tuples with the keys 10 and 11. Within the leaf page, usually binary search is done depending on the organization of the leaf page. For more details about B*-Trees and separators refer to [Com79].

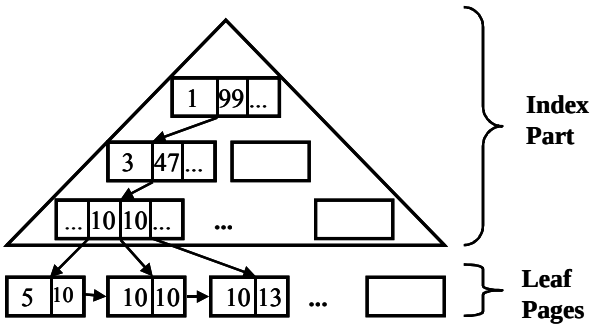


Figure 10-35: Separator Search on Duplicate B*-Tree

In Figure 10-35, we show an example for a non-unique B*-Tree. In non-unique B*-Trees, the definition of separators is modified. The left leaf page contains tuples that are smaller or equal to the separator, the right leaf page contains tuples that are larger or equal to the separator. This is the consequence of the case where only tuples with the same key are stored in the duplicate B*-Tree. More than one tuple with the same key may exist on the leaf pages. Consequently several identical separators can occur in the index pages of one level. The performance of the search of the leaf page is worse, because it can be necessary that two leaf pages must be read, if the separator has the same

value as the search value (see Figure 10-35). The left leaf page always must be read with a consecutive tuple search, and the right leaf page also must be read, if no tuple has been found on the left page.

If a leaf page is found that may contain tuples with the corresponding search value, the search within the page positions on the first ("smallest") matching tuple. If the largest tuple on the page has the same search value, tuples with the search value on the next page may exist. The access to the next leaf page is done via the next pointer on the leaf pages.

10.5.1.2 Search within one Leaf Page

The leaf page search of a tuple with search value *a* is done via binary search in the Transbase® B*-Tree implementation. Without loss of generality, we use a search key consisting of one single key attribute for the ease of description. The algorithms, however, are identical for compound search keys. The tuples on one page are stored ordered w.r.t. the order of the key attributes. Thus, a maximum of $\lceil \log_2 n \rceil$ search operations is necessary, where *n* is the number of tuples stored on the page.

For non-unique B*-Trees more than one tuple with the same key value can be stored on one page. The tuples are stored sorted again w.r.t. the order of the key attributes. The order of the tuples with the same key attribute value is not defined and depends on the insertion order in the current implementation. Thus, for the search of a tuple with key value *a*, the binary search is extended. If such a tuple is found via binary search, also the tuples *before and after* the found tuple have to be checked whether they also have the same key order. If the tuple before the found tuple has the same key order, we have to perform iterative *prev_read* operations, until we find a tuple with a key order smaller than the search key order. The semantic of the Transbase® *search(a)* operation is that the first tuple with the key value *a* is returned. The following tuples can be fetched via the *next_read* operation.

10.5.1.3 Range Queries on non-unique B-Trees*

The range query algorithm of the standard B*-Tree is not modified for non-unique B*-Trees. A query with a predicate `WHERE a BETWEEN` v_1 `and` v_2 is performed by searching for the first tuple with the key value $a \geq v_1$. For a non-unique B*-Tree, this point search is done as described in Sections 10.5.1.1 and 10.5.1.2. Iterative *next_read* operations are performed until a tuple is found with $a > v_2$.

For the range query algorithm with range predicates on more than one attributes (multidimensional query boxes), no extension is necessary for non-unique B*-Trees. The algorithm works in the same way as for one attribute.

The same holds for the multidimensional range query algorithm on non-unique UB-Trees. The UB-Tree is a one-dimensional B*-Tree with a computed key value (z-value of the UB-Tree index attributes). Thus, a range `WHERE z(`a_1`, `a_2`, …, `a_n`) BETWEEN z(`v^1_1`, `v^1_2`, …, `v^1_n`) AND z(`v^2_1`, `v^2_2`, …, `v^2_n`)` is performed by searching the first tuple with z-value $z(v^1_1, v^1_2, \ldots, v^1_n)$ and performing *next_read* operations until the z-value of the retrieved tuple is larger than $z(v^2_1, v^2_2, \ldots, v^2_n)$. Note that the range query algorithm on UB-Trees splits a multidimensional range query into a number of one dimensional intervals that are handled in a standard way by the B*-Tree.

10.5.1.4 Multiple Query Box Algorithm

The extended algorithm for multi-query-box handling has to be modified when collecting the page numbers from the separators of the index part (see Section 10.3.2.1). The reason is that tuples on the leaf pages can be placed on the left or on the right leaf page from the separator, if the separator has the same value as the search value (see Section 10.5.1.1). Thus, the number of page numbers resulting from the collection step is possibly larger than for unique UB-Trees.

10.5.2 Non-unique UB-Trees in Data Warehouses

As mentioned earlier, the fact tables in data warehouses are often organized by non-unique UB-Trees. With non-unique UB-Trees, the UB-Tree range query algorithm retrieves all tuples that are qualified by the multidimensional range query on the UB-Tree index attributes. A post-filtering step is necessary for additional restrictions on the secondary dimensions (see Section 9.5).

In fact tables, we typically have large tuples. This means that a comparably small number of tuples fits on one leaf page. For example, in the Sales DW, the tuple size is 361 Byte. For a page size of 8 KB, i.e., 8192 Byte, 22 tuples can be stored on one page. Thus, with one read operation, a maximum of 22 tuples can be read from disk. From these tuples, *p* are qualified by the predicates of the multidimensional query box and *22-p* do not contribute to the result. The advantage of the UB-Tree with good clustering and requiring a small number of disk accesses can be less for very large tuples.

Some DW have very large tuples and can only store a very small number of tuples, e.g., two to five, on one disk page. A good natural physical clustering of the tuples, e.g., w.r.t. the date dimension, can compete with the z-order when considering I/O. In such a case, a larger page size increases the number of tuples on one disk page. However, the page size is chosen w.r.t. the number of blocks that are read by one disk access from the operating system.

Another typical constellation in a fact table is a large number of secondary dimensions. The initial evaluation of the tuples contributing to the range query of the UB-Tree can be a large super set of the tuples qualified additionally by the predicates on the secondary dimensions. Post-filtering is very expensive, because a residual join with every restricted secondary dimension is necessary to evaluate the corresponding predicates for each restricted secondary dimension. A fast index intersection with secondary indexes on all dimension key attributes has the advantage that only tuples are retrieved from the fact table that are qualified by the restrictions of all dimensions. This comparably small result set is used for further processing.

We have to speed up the residual join using advanced join methods, e.g., hash join, in order to speed up post-filtering.

A further interesting approach for a fact table with many dimensions is to use secondary indexes on all dimensions, e.g., bitmap indexes, and perform a fast index intersection on the restricted predicates. If MHC is used on such a table, we can profit from hierarchical pre-grouping and speed up query processing significantly.

10.5.3 Residual Joins

Since the residual joins with secondary dimensions are performed with a very large number of tuples (see Section 9.5), an optimization of the residual join speeds up the overall query processing significantly.

The residual join is necessary for two purposes:

- applying the predicates of the secondary dimension
- retrieving the attribute values needed for further processing

The two join methods that are implemented in Transbase®, i.e., *nested loop* and *sort merge join*, do not support the requirements for fast joining in the context of the residual join of secondary dimensions. A sort merge join is not suitable, because the tuples resulting from the fact table are not sorted w.r.t. the secondary dimension and additional sort effort is necessary to apply sort merge join. Note that for the residual join of several secondary dimensions this sort effort is necessary for each secondary dimension. A nested loop join performs a lookup in the dimension table (or in a corresponding index), in order to test whether the tuple is qualified by the dimension predicate. The lookup is done via B*-Tree search and therefore is expensive, because of the overhead of a B*-Tree

search (locating the leaf page, performing a binary search on the page and extracting the tuple from the page). Usually, often the same key is used for the lookup, because many tuples of the fact table have the same dimension key of the secondary dimension. Even if the page is cached, the described overhead has influence on the performance.

We therefore propose to use a hash join implementation.

10.5.3.1 Description of Hash Join

We only give a short outline about the implementation of hash joins. A lot of investigation already has been done on this field ([CLYY92], [DG85], [HCY94]).

A hash-based join usually is applied on two sources, e.g., tables or intermediate results. One source is the *inner*, the other source the *outer table* (see Figure 10-36). We use the term table instead of source, because the sources are often tables. An intermediate result also can be seen as a set of tuples representing a table. The inner table is used to build the hash table, the outer table is used for the *tuple probing*. We split the hash join into the two phases

- table building phase and
- tuple probing phase.

It depends on the optimizer which table is chosen as inner and outer table (often the smaller will be the inner table). Special methods must be implemented, if the hash table built from the inner table does not fit into the main memory (see Section 10.2.6 for more details about table hash overflow mechanisms).

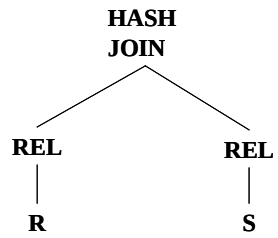


Figure 10-36: Operator Tree with Hash Join

The *table building phase* creates a hash table of the inner table using the hash function of the join attribute. The tuple probing phase tests for each tuple of the outer table, whether a tuple in the hash table exists with the same join key value. Note that with *right deep operator trees* ([CLYY92]), hash joins fit into the concept of pipelining in operator trees.

In Figure 10-37 we illustrate a sequence of hash joins applied to the residual join of secondary dimensions. The left son of the lowest `HASH JOIN` operator is the operator tree for the fact table access (see Section 10.1.6). The right son is a `RESTR` operator with the secondary dimension D^S_1 that is chosen to be joined first with the fact table result. In Section 10.5.3.5, we discuss the order of the residual joins of the secondary dimensions. The operator tree for the predicate evaluation of D^S_1 filters the tuples before the hash table phases. Thus only tuples are used to build the hash table that are qualified by the predicate of D^S_1. If these tuples are used for the tuple probing phase, only tuples that correspond to the dimension predicate are probed.

The `HASH JOIN` operator builds for each tuple that is probed successfully a new tuple as combination of the hashed tuples and the probed tuple with all attributes needed for further processing.

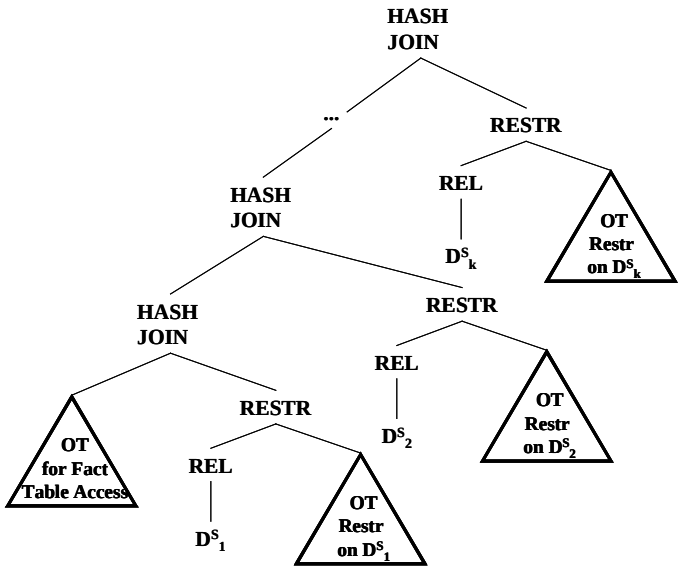


Figure 10-37: Residual Join Sequence of Secondary Dimensions with Hash Joins

10.5.3.2 Usage of Hash filters

An optimization of the probing phase is the usage of *hash filters*. A hash filter built by an inner table on the join attribute is an array of bits which are initialized with 0. The k^{th} bit of the hash filter is set to *one*, if there exists a hash function value *k* with a tuple in the hash table ([CHY93]). Thus, for each hash value that is occupied, the corresponding bit is set to *one* in the hash filter. It is very performant to check, whether a tuple from the probe phase *has not* a corresponding tuple in the hash table. There is not a corresponding tuple in the hash table, if the bit in the hash filter *hf* is 0 for the value of the hash function *h* of the tuple t_{probe} that is probed: $hf[h(t_{probe})] = 0$.

If $hf[h(t_{probe})] = 1$, the tuples with the corresponding hash value have to be checked whether the value of the join attribute of the probed and the hashed tuple are equal.

Note that these methods are a kind of semi-join.

10.5.3.3 Alternative Hash Join

A large dimension table results in a large hash table. If only a small subset of dimension table tuples occurs in the fact table result set (e.g., due to the dimension restriction), the overhead to build the hash table can exceed the benefit of the hash table join.

We therefore suggest an alternative, also hash-based join method. We access the dimension table via conventional index lookup and use a hash table as cache to store already needed tuples. For each fact table result tuple, we look into the hash table and use the tuple of the hash table, if it is already stored. Otherwise we have to access the dimension table. If a fact table result set contains a large number *m* of tuples with a small number *n* of distinct dimension key values, there are *m* hash table lookups and only *n* dimension table accesses. With the conventional hash join, we have also *m* hash table lookups and read the complete dimension table, in order to build the hash table. The lookup into the dimension table can be less efficient, because a higher number of tuples is stored in the hash table (each tuple of the dimension table) and the collision chains are longer and therefore the lookup time.

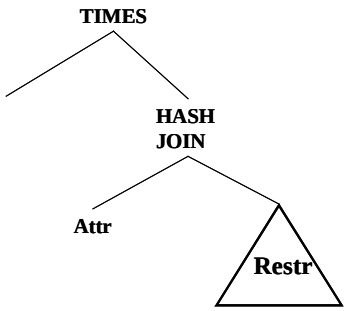


Figure 10-38: Alternative Hash Join

Figure 10-38 shows the basic operator tree for such a hash join organization. The `TIMES` operator represents the join between the fact table result set and the dimension table. The `HASH JOIN` operator is the special hash join with the lookup in the hash table and a lookup into the dimension table for non-existing tuples. `RESTR` is the dimension table restriction operator tree and `Attr` denotes the corresponding dimension key attribute necessary for the lookup into the dimension table.

Since secondary dimensions form an additional restriction on the fact table result set, some tuples are filtered out by this residual join. Such a dimension key value requires a lookup in the hash table (where it is not stored because of the dimension predicates) and then into the dimension table evaluating the dimension restriction. No such tuple is found and the fact table result tuple is discarded. For *k* fact table result tuples with such a dimension key value, this procedure is done *k* times. Especially the dimension restriction evaluation is very expensive. Thus, we propose to also store non-hit dimension key values in the hash table which are marked as *filtering tuples*. Each so found tuple denotes that the fact table result tuple is filtered out. As alternative we can build two hash tables, one for the hit tuples and the second for the filtering tuples. For each fact table result tuple we first look into the two hash tables. If none has a corresponding tuple, we look into the dimension table and store the result in one of the two hash tables.

This proposed method especially is better than conventional hash join, if the effort to build the hash table exceeds the iterative lookup into the dimension table and evaluation of the dimension predicate.

10.5.3.4 Parallelism

Hash-based residual join is suitable for parallel intra query execution. We do not enlarge on this, because parallelizing is beyond the scope of this thesis and a large amount of work has been spent on this subject.

In [CLYY92] and [HCY94], the authors describe a method to build operator trees that are suitable for parallel execution by using segmented right-deep trees for the execution of pipelined hash joins.

10.5.3.5 Order of Residual Joins

The order of residual joins is important for the performance of the query processing. A residual join can reduce the number of fact table result tuples due to the predicates of the secondary dimension. If the predicates reduce the result set significantly, the reduced result set is used for further processing, e.g., for the residual join with the next secondary dimension.

Thus, the residual join with the secondary dimension that reduces the result set most should be chosen as first residual join. For this decision a cost-based optimizer is necessary. However, it is not easy to decide which dimension reduces the result most, since the selectivity of the restriction on the fact table must be estimated. Statistics about the dimension tables and the fact table must be available as well as "join statistics". Heuristics depending on the hierarchy level of the dimension predicates also can be

used for the decision of the order of residual joins. But such a decision is very imprecise and can lead to bad operator trees.

10.5.4 Remarks

The hash join as optimization of the residual join for secondary dimensions can be also used for the residual join of primary dimensions. The hash tables can be built when evaluating the restrictions on the dimension table at the interval generating step. During the interval generating step, we already evaluate the complete dimension predicates and collect the tuples, actually the compound surrogates, for *HNPP* and *NH* predicate classes. During this evaluation we could insert the tuples with the attributes needed for further processing into a hash table and perform the residual join after the pre-grouping resp. post-grouping (see Section 9.4) by probing the tuples resulting from the grouping operations. Until now, hash joins are not implemented in Transbase®.

Non-unique UB-Trees that are necessary when dealing with secondary dimensions are also mandatory for other applications. For example, the UB-Tree string indexing method as described in Section 5.2.2 requires non-unique UB-Trees, because the calculated codes can be the same for different string prefixes.

10.6 Multiple Hierarchies

In real data warehouses, some dimensions have more than one hierarchy. For the physical schema, usually one hierarchy is chosen that is considered to be the most important hierarchy. However, sometimes more than one hierarchies are important for a large set of queries. Therefore, we have to deal with multiple hierarchies. The Transbase® implementation allows the definition of several compound surrogates for a dimension such as several reference surrogates referencing one dimension (see Section 7.6.4). The query processing must support multiple hierarchies, in order to evaluate such queries efficiently.

In Section 7.6.4, we introduced an example for a dimension with two alternative hierarchies, each of which repesented in a separate compound surrogate. Both surrogates occur in the fact table as reference surrogates. Since each of the reference surrogates is used as index key of the clustering multidimensional index on the fact table, predicates on each of the hierarchies determine the multidimensional range queries on the fact table.

Because the queries do not specify explicitly which hierarchy is used, the optimizer has to decide this when generating the execution plan. Depending on the predicate of the dimension, one or more interval restrictions are built from the dimension predicates. The selection of the hierarchy is done in the schema recognition phase (see Section 10.1). For each restricted hierarchy, we build the hierarchy characteristics, in order to generate the predicate classes and corresponding operator trees.

Each hierarchy restriction leads to a restriction on the multidimensional index and therefore is used as restriction on the fact table. The consequence is that one operator tree is generated for each hierarchy representing the “dimension” restriction of the multidimensional index. Note that this terminology of dimensions refers to the dimensions in the multidimensional index and does not necessarily correspond to the primary dimensions of the schema.

If a hierarchy level occurs in more than one hierarchies, we use the restriction on such a hierarchy level for all corresponding hierarchies. For example, the dimension key always is the leaf hierarchy level of all hierarchies. However, higher hierarchy levels also can occur in more than one hierarchy (see example of Section 7.6.4).

For a restriction of a feature attribute in the leaf dimension table, the optimizer has to decide to which hierarchy the feature attribute is assigned. Since there is no corresponding information in the schema, it is difficult to chose the suitable hierarchy. For example, some feature attributes are correlated highly

to one hierarchy level, but are orthogonal to other hierarchies. Thus, the number of intervals on the compound surrogates depends on the choice of the compound surrogate.

With the availability of statistics it is possible to compare the distribution of the feature attribute with the hierarchy levels. If the distribution is very similar to a hierarchy level, the probability to have correlated attributes is very high. A false decision can lead to a very high number of compound surrogate intervals with corresponding bad performance (see Section 10.3).

If no statistics are available, we can use one of the hierarchies or all. Note that using only one compound surrogates qualifies the same fact table tuples because the restriction on the alternative hierarchies serve as feature restrictions. Thus, a number of smaller intervals may follow. We can decide dynamically which hierarchy is used for the fact table evaluation by modifying the predicate structure of the page number collection phase (see Section 10.3.2).

For snowflake dimensions with hierarchy levels and corresponding feature attributes in higher dimension tables, it is easy to decide which hierarchy to chose for the interval generation. The dependency of feature attributes and hierarchy levels is know in such a schema.

If not all reference surrogates for the hierarchies of one dimension are used as index keys in the multidimensional index, the optimizer uses the hierarchy that is indexed in the fact table. A restriction on an alternative hierarchy that is not indexed, is mapped to a restriction on one of the indexed hierarchies, where the restricted hierarchy levels are treated as feature attributes. The interval generation is done with standard methods.

The definition of multiple dimensions increases the number of physical dimensions on the multidimensional clustering index on the fact table. If the number of these dimensions becomes high, the clustering of the fact table is not good any more and slows down the evaluation of multidimensional range queries. Thus, a reduction is necessary reducing the number of indexed dimensions. Usually, less important hierarchies are removed from the index attributes. In some cases, however, it is not decidable which hierarchies to remove. We therefore propose a method to merge hierarchies and provide comparable good query performance for a large number of predicates (see Section 11 for more details). We call this method the transformation of so called complex hierarchies to simple hierarchies.

10.7 Multiple Fact Tables

In some data warehouse applications several fact tables exist that share common dimension tables. Queries contain several fact tables in the `FROM` clause, i.e., the fact tables are joined (directly or indirectly). With the standard algorithms described so far, we cannot handle such queries and schemata. Figure 10-39 shows an example for such a schema. F^1, F^2 and F^3 denote the fact tables and D^i_j the corresponding dimension tables that belong the the fact table, where the dimensions D^1_j belongs to the fact table F^1 etc. Each D^i_j represents a complete dimension, i.e., the dimension can be a snowflake dimension and D^i_j represents the complete dimension join. The join tables between the fact tables are dimensions again, denoted by $D^{1,2}$ for the join between F^1 and F^2, and $D^{2,3}$ for the join between F^2 and F^3. Note that the join can be specified via such a *connector dimension table* or via a dimension key attribute itself.

The standard approach is to use conventional join optimization features of the DBMS optimizer. This results (in Transbase®) into a sort merge join between the fact tables and the dimension tables with long query evaluation times.

A better approach is to choose one of the fact tables as standard fact table with the proposed star join processing. The fact table result set is joined with the fact table result set of the remaining fact tables. The join method depends on some criteria that are explained later. However, it is necessary to recognize this kind of schema. In Section 10.1 we discuss how to recognize a typical star schema.

Similar methods are necessary to recognize a more complex data warehouse schema with multiple fact tables. The algorithms must be extended to deal with additional information, e.g., to which fact table each dimension table belongs. Especially shared dimensions often occur, and it must be checked to which fact table they belong. These methods are not implemented yet in Transbase®, but are planned for the future, in order to handle such complex schemata.

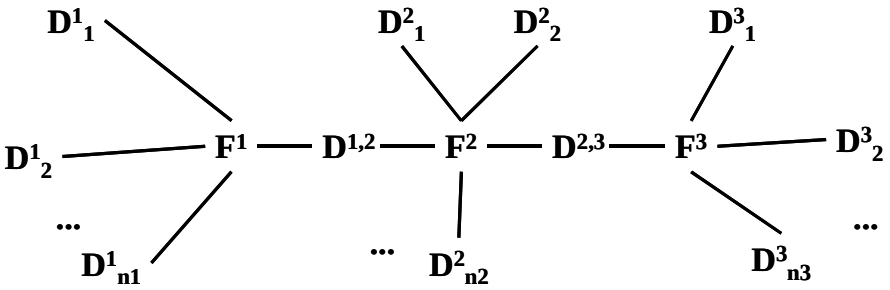


Figure 10-39: Schema with multiple Fact Tables

The join method between the fact tables is similar to the join optimization of secondary dimensions. We consider the join between two fact tables as a secondary dimension join. One of the fact tables is used as *primary fact table*, the other fact table is considered as secondary dimension on the primary fact table, i.e., as *secondary fact table*. The secondary fact table is evaluated before the join (via the proposed star join processing algorithms for the interval generation) and is joined via hash join with the result of the primary fact table.

Grouping and aggregation is done after the join. We cannot perform pre-grouping before joining with the secondary fact tables, because the join between fact tables must be done on the lowest cardinality. After the secondary fact table (dimension) join, however, pre-grouping is possible and advantageous.

We characterize three different kinds of schemata for multiple fact tables:

- Sequential Join
- Star Join
- Snowflake Join

The *sequential join* is a chain of fact tables F^1, F^2, …, F^N (see Figure 10-40), where the first fact table F^1 is joined via the connector dimension $D^{1,2}$ with F^2, this fact table is joined via $D^{2,3}$ with F^3 etc. Each fact table additionally can have a number of dimensions that are not shown in this figure.

The *star join* of fact tables is similar to a star schema with one fact table F^1 in the center and the other fact tables surrounding (Figure 10-41). As in the illustration for the sequential join, we do not show the dimensions for each fact table.

The *snowflake join* of fact tables is a general join, i.e., a join graph with one center fact table, F^1 in Figure 10-42. The sequential and star join of fact tables are special cases of the general snowflake join. Note that no cycles within the join graph are allowed (similar to connected dimensions for the star query algorithms). Otherwise it is difficult to get an appropriate order of the join processing of the fact tables. Shared dimensions that usually cause such cycles, are seen as separate dimensions for each fact table.

10.7.1 Sequential Join

There are some different methods how to perform sequential fact table joins. The most intuitive approach is to evaluate the first and the second fact table (F^1 and F^2 in Figure 10-40) and join the results, then evaluate F^3 and join with the previous results etc. until the last fact table is joined.

The second approach is to evaluate the fact tables in pairs and join them. These results are joined again in pairs etc until all fact tables have been joined.

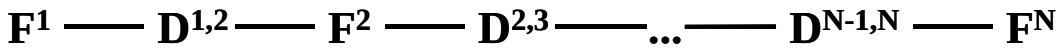


Figure 10-40: Sequential Join of multiple Fact Tables

The first method works with a very small result set cardinality, because each join does not enlarge the result set from one previous fact table join. The second method keeps the tuples resulting from each join short, because the tuples contain only attributes of two fact tables (and the necessary dimension tables).

10.7.2 Star Join

A star join of multiple fact tables is similar to the classic star schema with one fact table in the center and the surrounding dimension tables. There is one fact table, the primary fact table, in the center of the star and the remaining fact tables are connected via dimension tables with the primary fact table. We use a similar method for the join processing as for standard star query processing. First, the edges of the star are evaluated. We evaluate the result of all secondary fact tables via interval generation for each fact table as described earlier. Then, we evaluate the result of the primary fact table, F^1 in Figure 10-41.

The result of each fact table is considered as result of secondary dimensions for the primary fact table and joined accordingly.

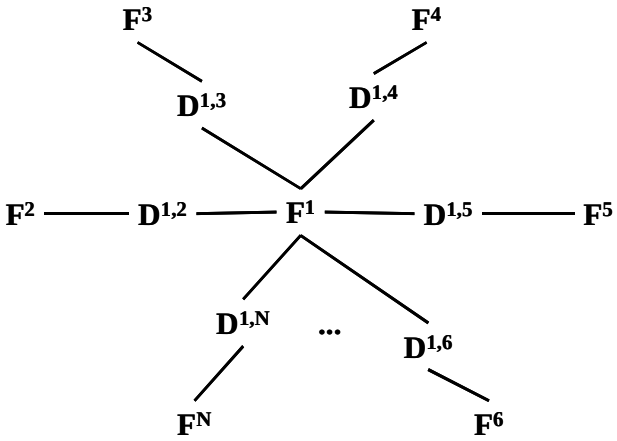


Figure 10-41: Star Join of multiple Fact Tables

10.7.3 Snowflake Join

We select one fact table as center of the snowflake join, usually one fact table that is surrounded by several other fact tables and has a comparably small result set. In Figure 10-42, we use F^1 as primary fact table and the remaining fact tables are secondary fact tables. The fact tables directly connected with the primary fact table are called *leaf fact tables* (similar to leaf dimension tables in the snowflake schema). We start with the evaluation of the fact tables at the edge of the snowflake join and join them with the next fact tables until all leaf fact tables are evaluated and joined with the remaining secondary fact tables.

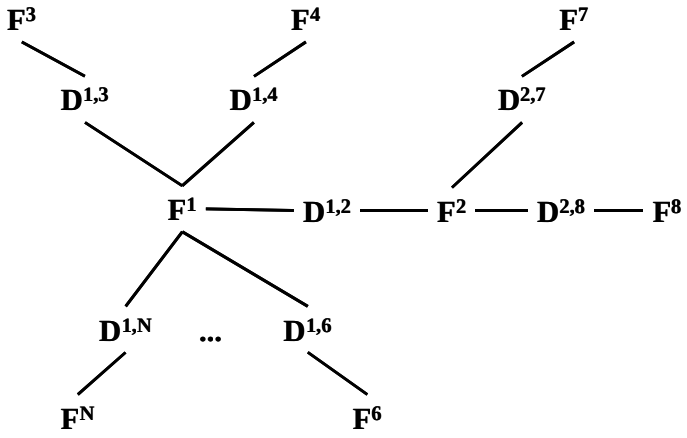


Figure 10-42: Snowflake Join of multiple Fact Tables

One method to build the execution plan is to start with the evaluation of F^8 and F^2 in Figure 10-42 and join the results via $D^{2,8}$. Then we evaluate F^7 and join with the previous result. After the evaluation of F^3, F^4, F^6, …, F^N we join each of the results with the primary fact table F^1. In general, we start with the outer fact tables and finally join all of them with the center fact table. The selection of the center fact table, however, is not deterministic, since different fact tables can be chosen.

As alternative, we can evaluate one fact table at the edge of the snowflake join and then iteratively join with the remaining fact tables. The order of the joins depends on the result set cardinality of each fact table.

10.8 Schema for the Measurements

This section describes the schema that is used in the following measurements. It is a real world data warehouse schema of a large electronic retailer. We call the warehouse *Sales DW*.

10.8.1 Conceptual Schema

The Sales DW is a star schema with the fact table *SALESFACT* and 14 dimensions. It contains measures for the daily business sales. In other words, each record in the fact table, gives values of the sales of a particular product, on a particular day, to a particular customer, who carried out a particular transaction type on a particular store with a particular salesman, along all the other dimensions, whose roles and descriptions follow.

10.8.1.1 Overview

The dimensions of Sales DW are described in the following:

- CALENDAR: The star schema's "time" dimension which is easily extendable to any arbitrary date in the future. It contains several descriptor fields that classify a given date into a certain higher level time period such as week, month, quarter, fiscal year, etc. There are actually three dimensions with a "timely" aspect, i.e., *transaction date*, *export date,* and *delivery date*. They all use the same dimension table in the logical schema. Depending on the time dimension, different views on *SALESFACT* are possible, one for each time dimension.
- HOUR: The "hour" dimension. Has exactly 24 records each one representing an hour of the day and includes classifier fields that group hours into, for example, peak or non-peak.
- PRODUCT: The "product" dimension represents all products classified among hierarchies as described later.
- WAREHOUSE: Contains each warehouse (store) of the business. The warehouse dimension, like the time dimension, has multiple roles for each fact record. There are three warehouses

for the fact records, which represent the *transaction warehouse*, the *export warehouse,* and the *delivery warehouse*. This means that actually the warehouse dimension table is used for three conceptual warehouse dimensions. Again, a transaction, an export, and a delivery view is possible on the data warehouse.

- CUSTOMER: The customer dimension is by far the largest dimension in our schema. It contains several transformed customer descriptors.
- TRANSACTION: This dimension represents the type of a sales transaction. Representative transaction types include for example an order, a return, or a wholesale order.
- OFFERING: Contains the offering types. It is used in the fact table, in the case that there is an offering involved in the transaction.
- SALESMAN: In this dimension, the salesmen are stored that are involved in a transaction.
- CASH_REGISTER: The cash register dimension describes the register that issued the invoice of the transaction.
- CURRENCY: The currency dimension contains the different currencies that may be accepted at a selling point.
- SALES_PAYMENT: Contains the payment ways for the transactions.
- LOAN_STATUS: This dimension describes the loan status for the cases that the transaction involves loans.
- SPECIAL_IDL: This dimension is actually the collection of other dimensions with very low cardinality. Thus, it can be decomposed to the dimensions
    1. The DELIVERY dimension which describes what kind of delivery method is used for the transaction.
    2. The RESERVED dimension which describes what kind of reservation method is used for the transaction.
    3. The COVERED dimension which describes whether the transaction is covered by another transaction.

Basically there are three different warehouses combined to one, i.e., the *transaction*, *export* and *delivery* warehouse. This has impact on the sparsity of the fact table. Each record in the fact table contains either values for transaction or export or for delivery.

10.8.1.2 Dimensions and Hierarchies

This section contains a description of the dimensions. In particular, we show the hierarchies that are necessary to understand the business context. Due to the large number of dimensions we concentrate on a subset of the most important dimensions. These dimensions are also used in the context of physical modelling for an MHC organized Sales DW.

The following sections describe some of the important dimensions. Appendix A shows the conceptual schema for all relevant dimensions.

10.8.1.2.1 Dimension Calendar

The Calendar dimension has three alternative hierarchies (Figure 10-43):

- Date – Month – Quarter – Half Year – Year
- Date – Week – Year
- Date – Day of Week

In addition to the hierarchy attributes, a number of feature attributes describe the hierarchy levels in more detail. As mentioned before, the calendar hierarchy is used by three dimensions: transaction, export, and delivery date.

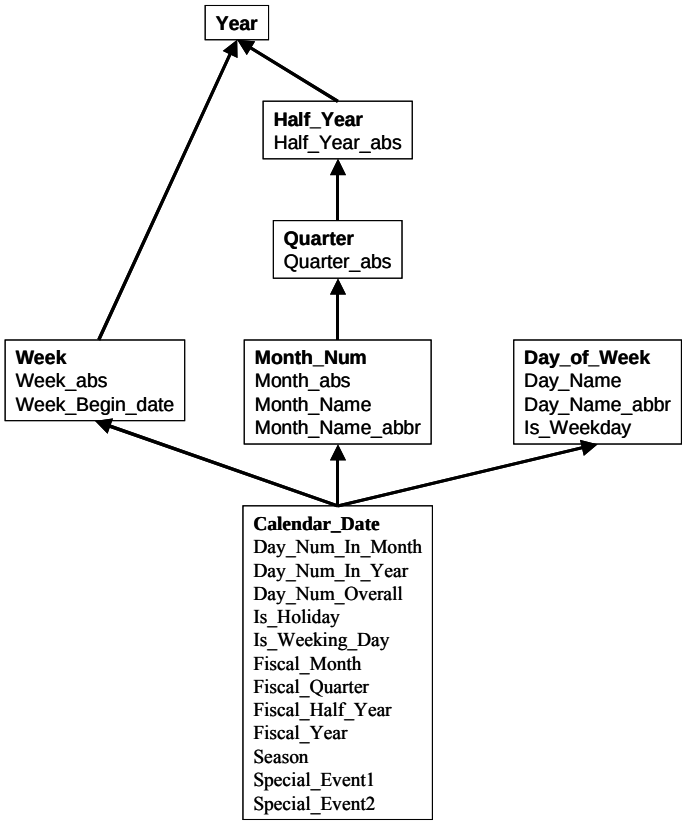


Figure 10-43: Calendar Hierarchy

We show the conceptual schema. Each box contains the fields of one hierarchy level. A hierarchy level consists of the hierarchy level attribute which is bold in the figures and of a number of feature attributes. The arros denote the hierarchical relationships between the hierarchy levels. Note that there are also hierarchy levels without feature attributes.

Basically, a feature attribute can be seen as a separate hierarchy level that is hierarchically dependent on the actual hierarchy level attribute. For the distinction between feature and hierarchy level attribute we need knowledge about the semantics of the fields of a dimension.

10.8.1.2.2 Dimension Hour

The hour dimension (Figure 10-44) is used, in order to express events within one day. For each hour, there is an entry in the dimension. Each hour is further categorized by a day period, e.g., noon, evening etc., and a boolean flag *is_peak* indicates, whether the hour is a peak hour.

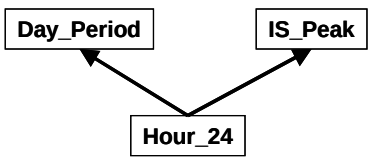


Figure 10-44: Hour Hierarchy

10.8.1.2.3 Dimension Product

In the product dimension (Figure 10-45), we have four alternative hierarchies:

- Item – Group – Category
- Item – Brand
- Item – ABC Category

- Item – Vat Category

The hierarchies contain feature attributes for a more exact description of the item, product group etc. The most frequently used hierarchy is the Item – Group – Category hierarchy.

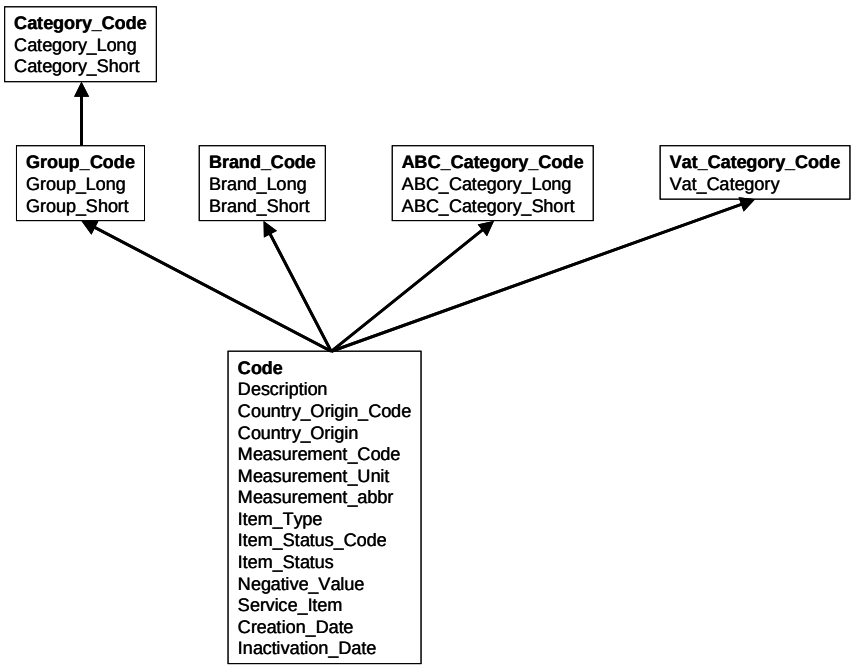


Figure 10-45: Product Hierarchy

10.8.1.2.4 Dimension Warehouse

The warehouse dimension (Figure 10-46) is complex. It has various alternative hierarchies. The most important one is the geographic hierarchy: Warehouse – Area – City – County – Department – Country. The remaining hierarchies are used for further evaluations.

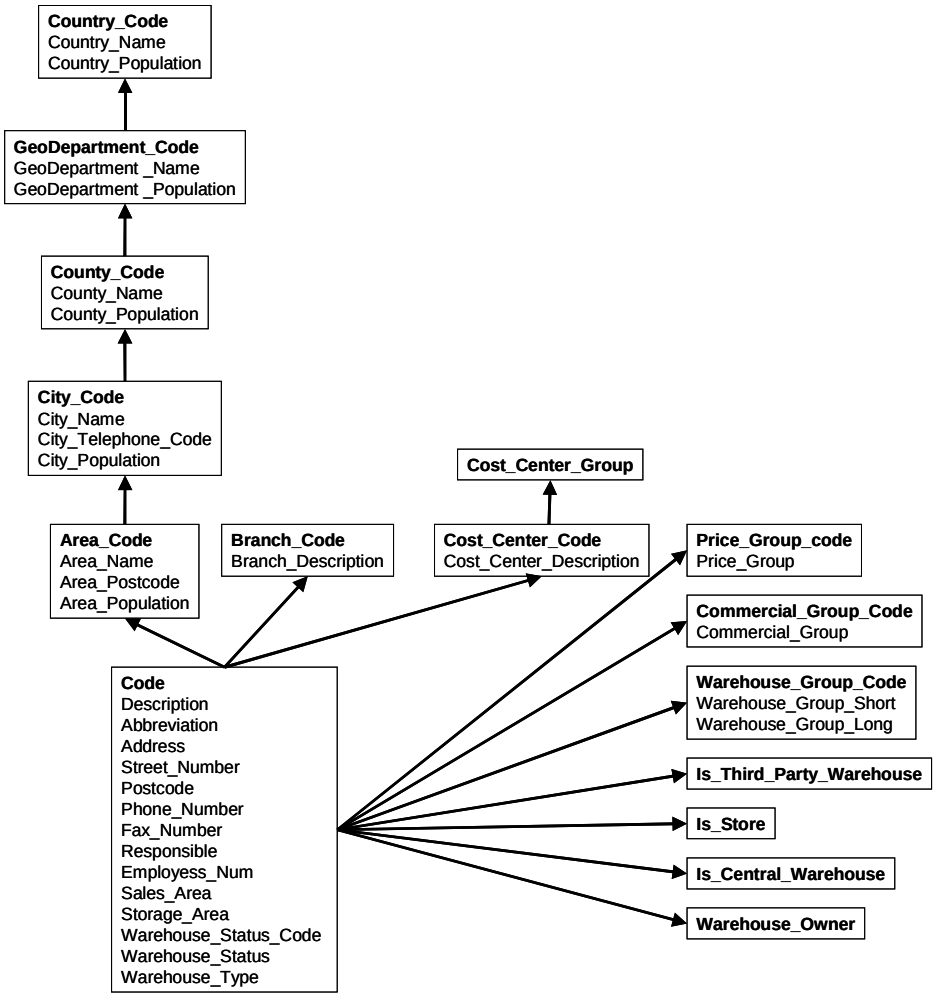


Figure 10-46: Warehouse Hierarhcy

10.8.2 Measures

The Retailer warehouse has a list of measures, in order to quantify business transactions. In the following, we list the measures that are stored in the fact table. The measures are divided into three categories:

- *Quantities*:
  - Qty_total,
  - Qty_Free,
  - Qty_Net
- *Unit Prices* and *Costs*:
  - Unit_Cost, Unit_Base_Cost
  - Unit_Sale_Price, Unit_Init_Sale_Price
  - Unit_Interest, Unit_Price_Cost
  - Unit_avg_cost, Unit_min_price
- Total Values – *Prices*, *Costs* and *Counts*:
  - Val_gross, Val_cost, Val_taxable, Val_vat
  - Val_gross_free, Val_vat_free
  - Val_discount, Val_surcharges
  - Val_interest, Val_capital, Val_total
  - Loans_Num, Total_instalments

We do not enlarge on an exact description of the measures, since they reflect the business content that is not of interest in this thesis.

10.8.3 Logical Schema for Measurements

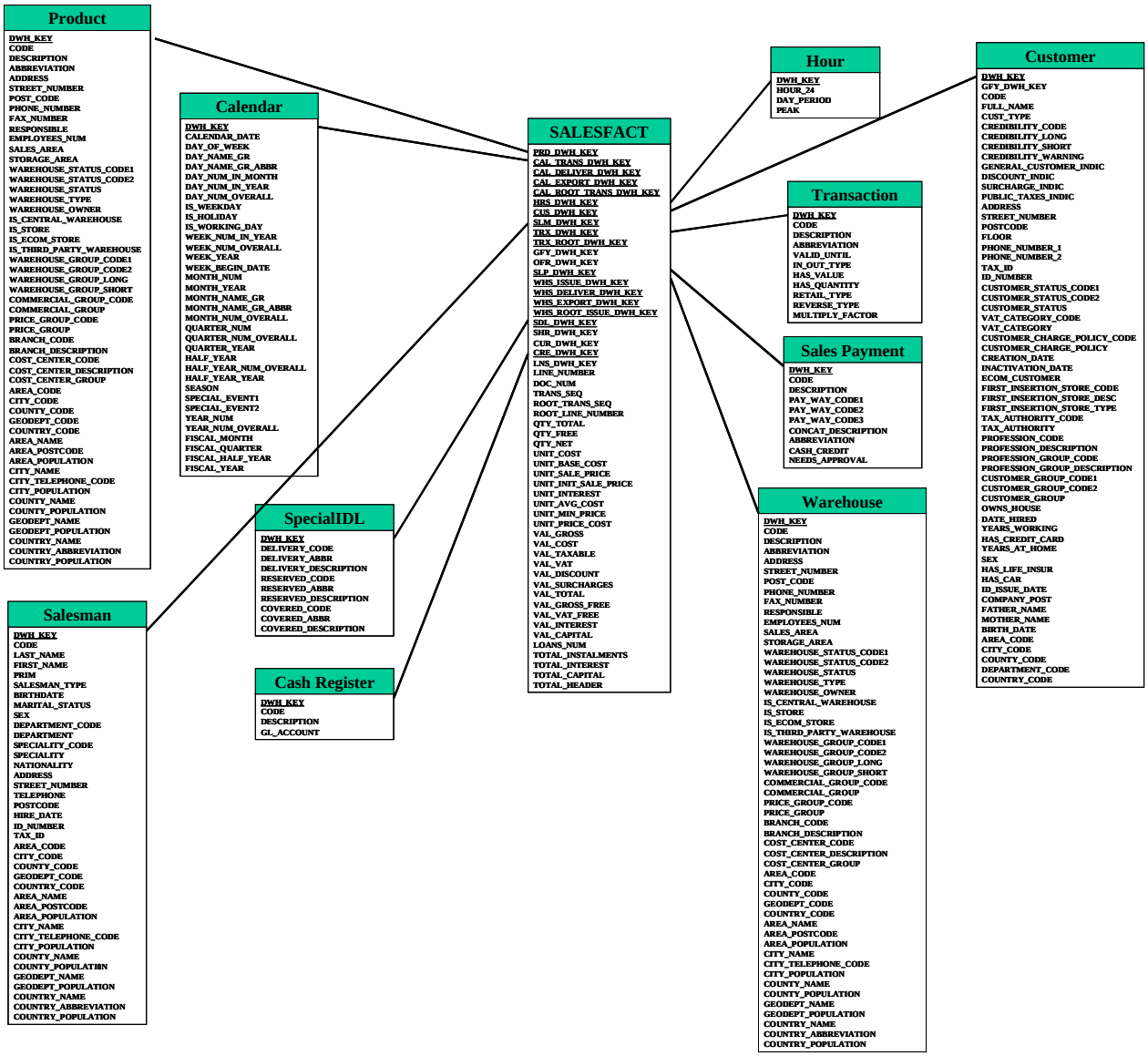


Figure 10-47: Logical Schema of Sales DW

The complete Retailer DW is very complex. Thus, for the measurements and analysis, we chose a subset of all dimensions. All attributes of the fact table are used, but the dimensions are restricted to the following 10 dimensions:

- Transaction Calendar
- Customer
- Hour
- Product
- Special IDL
- Warehouse
- Cash Register
- Salesman
- Sales Payment
- Transaction

In Figure 10-47 we show the logical schema for Sales DW used in the remaining thesis, if not described differently. The logical schema has a fact table *SALESFACT* and 10 dimension tables. The key attributes are marked with underlines and the foreign key relationships are expressed by lines between the corresponding attributes. E.g., *SALESFACT.PRD_DWH_KEY* is a foreign key to Product.*DWH_KEY*. The hierarchies are not shown explicitly since in the star schema, there is no hierarchical information available for the logical schema. Figure 10-47 shows the complete logical schema.

10.8.4 Data Distribution of the Fact Table

In this section we describe some aspects of the data stored in the Sales DW. In particular, we discuss the data distribution of the fact table.

The Fact table contains 8.579.458 tuples, i.e., the complete data of three years. In real data warehouses, the data is distributed extremely non-uniformly. This section contains an analysis of the data distribution w.r.t. the dimensions and hierarchies on the dimensions as they are used for the measurements. The data distribution is shown graphically. The figures show the number of fact table tuples that belong to the corresponding hierarchy members, e.g., 250.000 for "03/2000", i.e., March 2000, in the calendar dimension. The x-axis contains the hierarchy path (usually a prefix path depending on the hierarchy shown in the graph). For example, '1 1 1' for the customer dimension with country – department – county means the hierarchy path "1"/"1"/"1". The y-axis contains the number of tuples of the fact table belonging to the hierarchy prefix. Each chart contains only hierarchy members with corresponding tuples in the fact table. This means that members where no bars are visible, have a very small number of corresponding tuples in the fact table. In Appendix B, we show the data distribution of all dimensions.

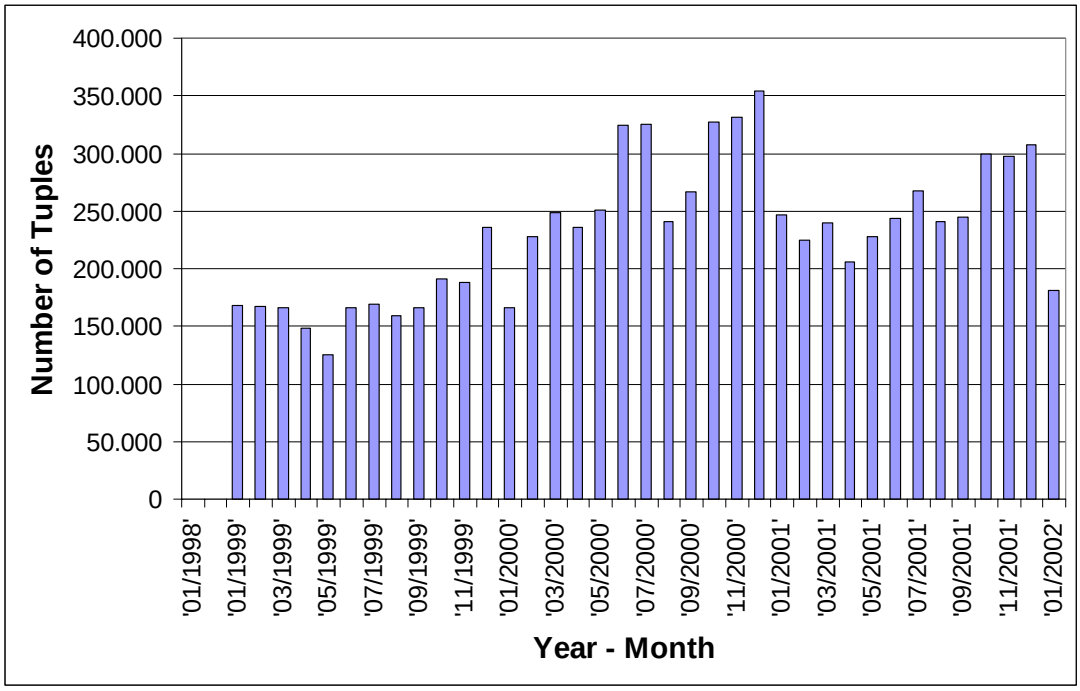


Figure 10-48: Data Distribution according to Calendar: Year – Month

Overall, most dimensions have a non-uniform data distribution. Only the calendar dimension contains a comparable amount of data for most months stored in Sales DW (see Figure 10-48). The geographic customer and warehouse dimension only contain data of one country. A large number of fact table tuples for the customer dimension is classified to one special path country – department – county – city (see Figure 10-50).

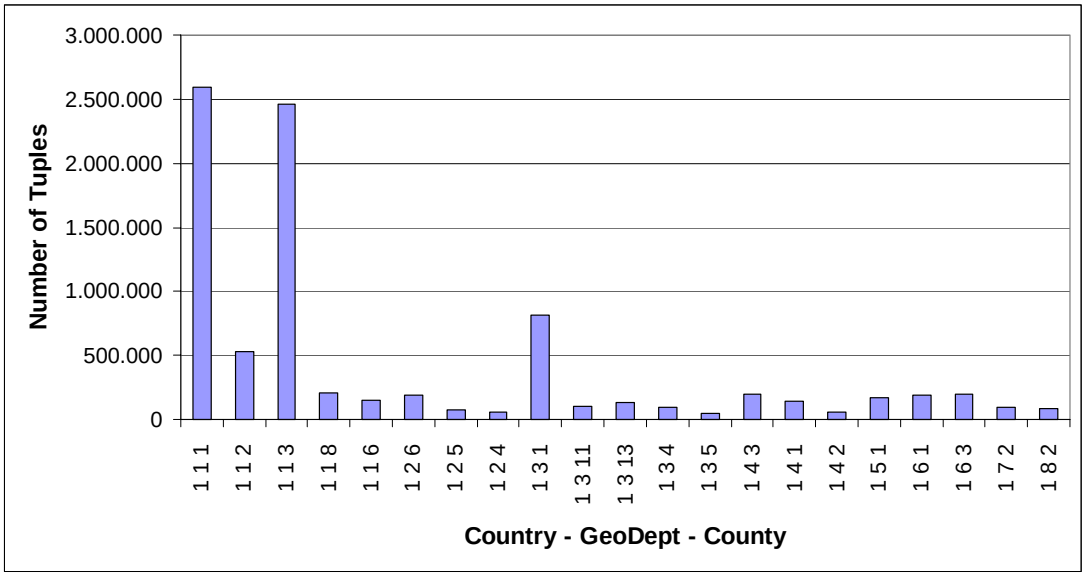


Figure 10-49: Data Distribution according to Warehouse: Country – GeoDepartment - County

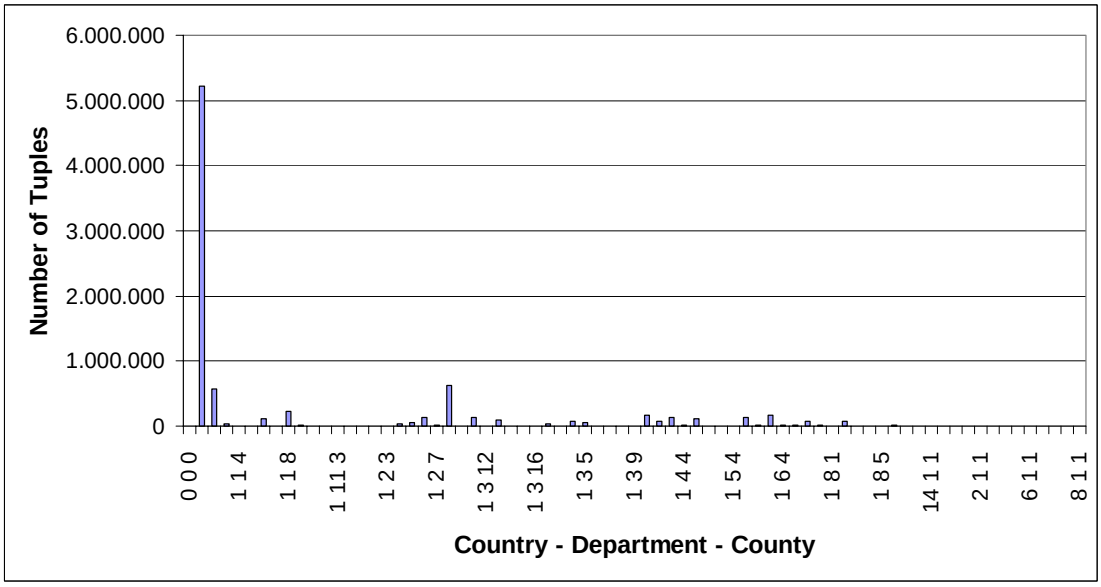


Figure 10-50: Data Distribution according to Customer: Country – Department - County

In the product dimension, there are several categories with a large number of tuples in the fact table. The two categories 41 and 21 occur most frequently in the fact table (see Figure 10-51), the products w.r.t. the product group is shown in Figure 10-52 with some data clusters around the categories with the most tuples.

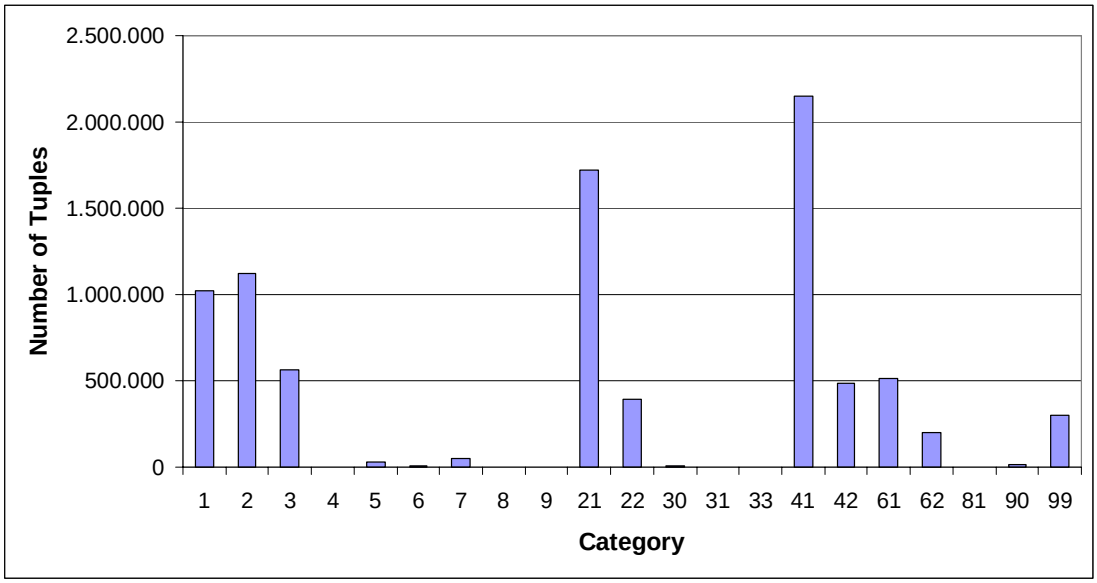


Figure 10-51: Data Distribution according to Product: Category

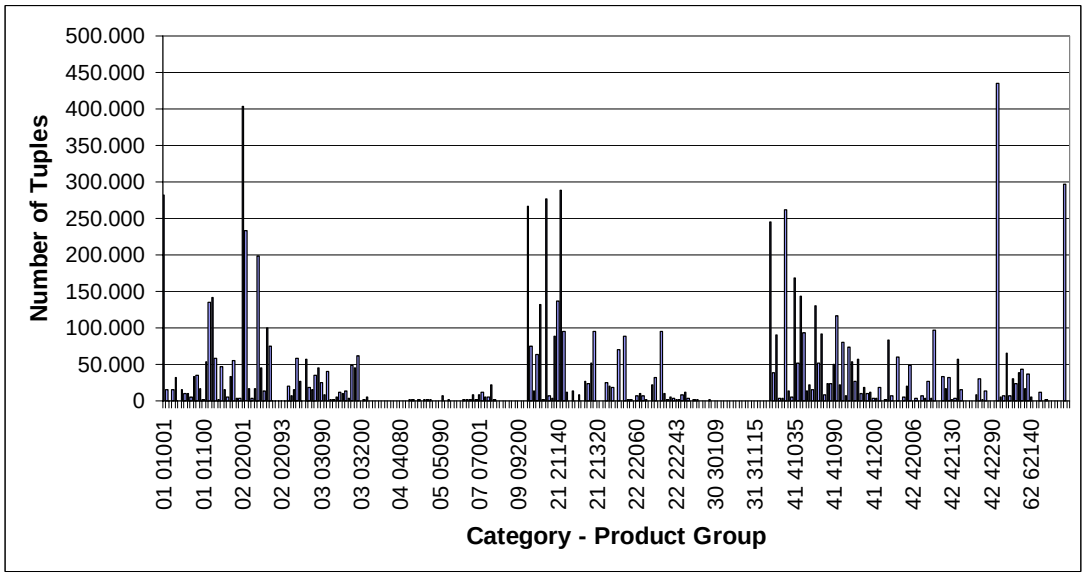


Figure 10-52: Data Distribution according to Product: Category – Product Group

Most tuples of the fact table have the sales payment '999999999' which is a "not categorized" classification. However, there are two sales payment types that belong to about 1.250.000 tuples in the fact table. Most of the sales payment ways have only a small number of tuples (see Figure 10-53).

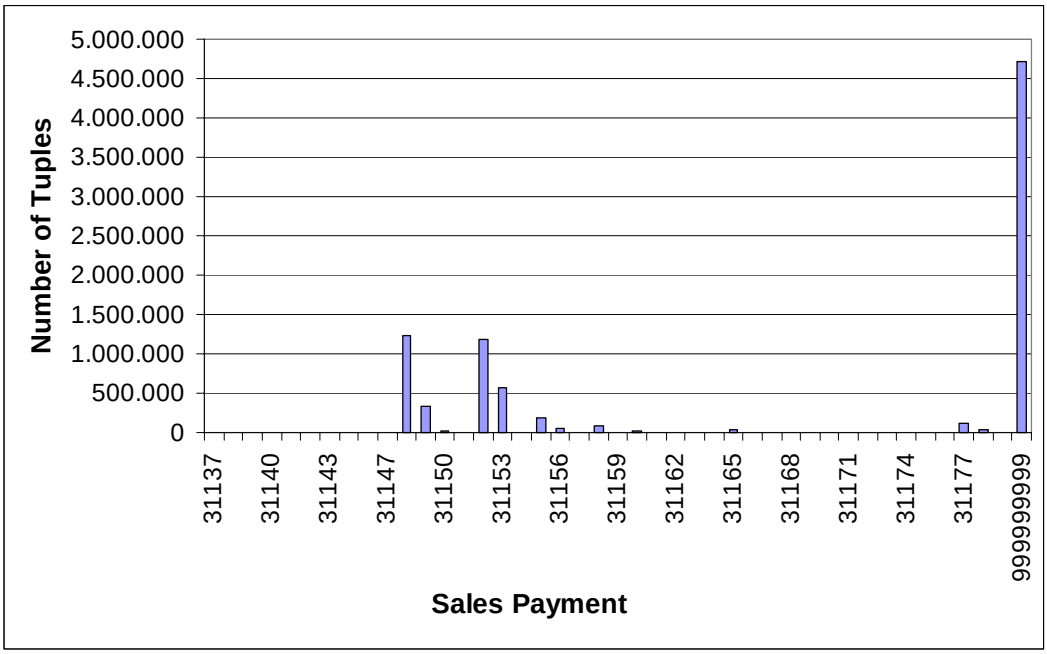


Figure 10-53: Data Distribution according to Sales Payment

In the transaction dimension, there are three transactions with a very large number of tuples, i.e., between 1.200.000 and 2.500.000., all other transaction types belong to less than 500.000 tuples of the fact table.

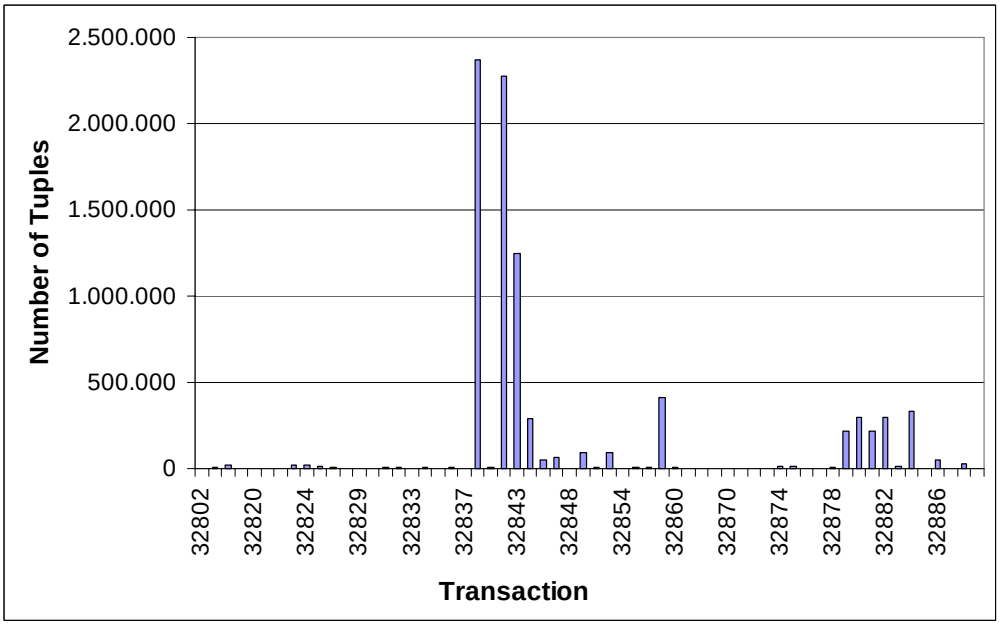


Figure 10-54: Data Distribution according to Transaction

10.9 Further Measurements

In this section, we describe some measurements that are not assigned to specific implementation sections:

- Scalability w.r.t. cache size
- Scalability w.r.t. fact table size
- Comparison with an other commercial DBMS

These measurements show the applicability of MHC in combination with the introduced optimization algorithms. We use the Sales DW as introduced in Section 10.8 with the five clustering dimensions *Product, Warehouse, Calendar, Transaction,* and *Sales Payment*. The remaining dimensions are secondary dimensions. The measurements are performed on a PC with 2 CPUs Pentium III Xeon, 866Mhz. The hard disks are IDE disks. Operating system is Windows NT4. All data is stored on one disk. The queries are executed with cold cache, i.e., cache effects are eliminated.

10.9.1 Description of the Queries

The queries are real business queries and are based on the application of the Sales DW. We defined 13 different templates, each of them describing one business case:

- Actual turnover per warehouse, month, hour
- Product analysis of sales
- Monthly analysis of sales for product groups
- Sales analysis
- Analysis of sales after 3pm
- Actual turnover per warehouse and product
- Net sales for consignment notes sales transactions
- Pending sales transactions
- Analysis of sales for root warehouses (stores)
- Cancelling transactions
- Credit cards revenues
- Credit sales per month
- Analysis of sales for Sales DW credit payment ways

Appendix D contains the query templates in SQL based on the logical schema as described in Section 10.8.3. The templates Q1 to Q13 corresponding to the different query types. The remaining templates are modifications, e.g., template Q203 corresponds to template Q3, Q208 to Q8 etc.

For the measurements we generated instances from the templates. Each of these templates has a date range parameter that was modified for different selectivities. Furthermore, the third and tenth templates imposed a range restriction on the *Product Group* level of the *Product* dimension. Likewise, template number eight imposed a restriction on the *Warehouse* dimension.

In order to generate a set of queries that would cover all possible cases (high selectivity and low selectivity queries), we have performed an analysis of the restrictions on the selectivity that the parameters of each template may enforce. During the analysis process, the number of tuples returned by each of the query templates was computed, when deleting the `GROUP BY` clause and varying the date range parameter.

Using these results, we have initially derived three classes for the group level of the *Product* dimension of the query template number three. Next, three more classes were derived for the *County* level of the *Warehouse* dimension of the query template number 8. Finally, we have defined three additional classes for the group level of the *Product* dimension of the query template number ten.

In order to simplify the query generation process, the query templates three, eight and ten have been transformed so that they had only one parameter: the date range. This process allowed us to use all templates in the same manner since all templates would have only one parameter. At the end of this process we had 22 query templates: 12 generated templates and 10 original. For each of these templates sixty query versions were generated using date ranges from one month to two years. The process described above generated the total number of 1320 queries that are used as the report query set for the performance measurements.

10.9.2 Cache Size Scalability Measurements

We first examine two different cache sizes:

- *Small*: Data cache of Transbase® is 30 MB ([Tra01]).
- *Large*: Data cache of Transbase® is 300 MB.

Because of the large number of measured queries, we discuss only a couple of query templates. The complete measurement figures are in Appendix E. Figure 10-55 shows the results for Transbase® with small (*TborigSmall*) and large (*TBorigLarge*) cache. *orig* stands for the original database size (about 3 GB raw data for the fact table).

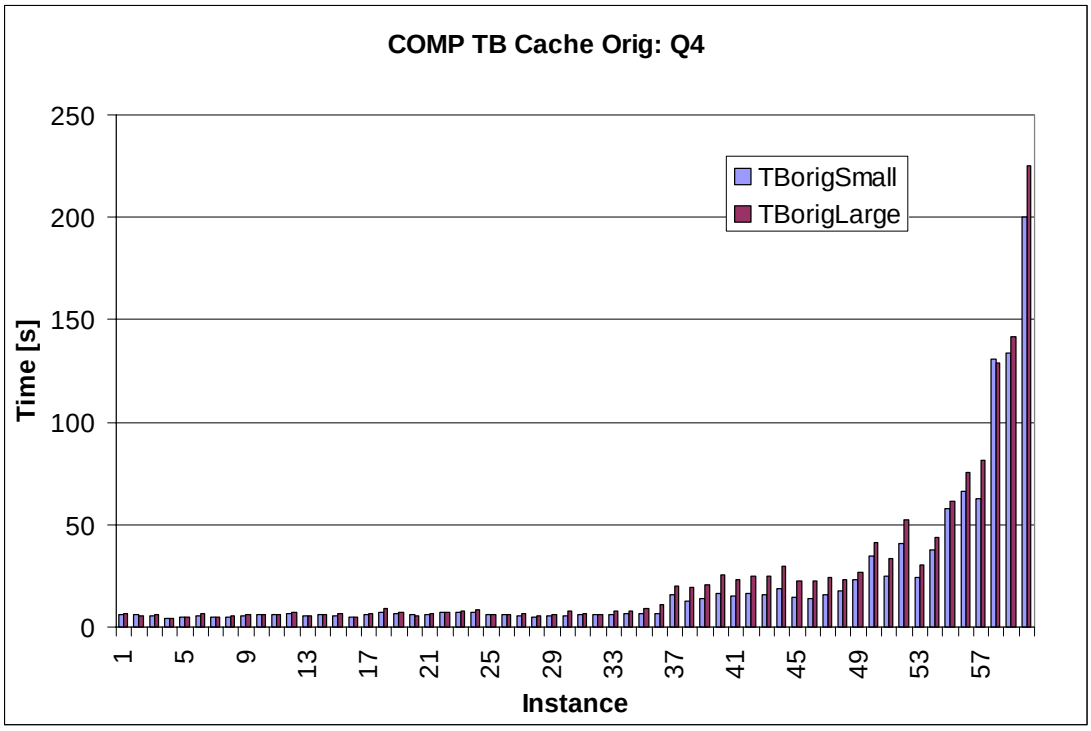


Figure 10-55: Comparison of different Cache Sizes for Template Q4

The queries are ordered by the instances. The intervals for the *calendar* restriction is one month for the first 36 queries and is enlarged up to three years (i.e., the complete time) for the last query. The rising number of fact table result tuples is reflected in the query response times. The left bar is for the small cache configuration, the right bar for the large one. Because of the elimination of DBMS cache effects (we reboot the database before each query), the large configuration cannot gain profit from the additional available memory. It even is slower in this environment to use a larger cache configuration, because the cache is filled during query execution. This leads to the requirement of additional shared memory for the Transbase® kernel. The operating system, however, uses most of the free memory as file cache and additional overhead occurs when this memory is requested by the Transbase® kernel. For larger caches, this overhead is larger and therefore the query execution time is slower. This is especially true for the queries specifying a large number of fact table tuples (many pages are written to the DBMS cache).

When not eliminating the cache effects, the results will be different. In this case, some (or most) of the requested data (and index) pages are already in the shared memory and can be used without accessing the secondary storage.

10.9.3 Fact Table Size Scalability Measurements

The objective of these measurements is to show the scalability w.r.t. the fact table size. We built a second database with more fact table records (a factor of 10, i.e., 30 GB raw data). The data was enriched by inserting tuples with consecuting calendar dates. Thus, the result of the queries is the same for both databases, the original and the 30 GB one. The queries are the same as described before. We run the queries with the small cache size.

Figure 10-56 shows the results for query Q6. *TBorigSmall* is the original database size and *TB30Small* stands for the large database size (with small cache configuration).

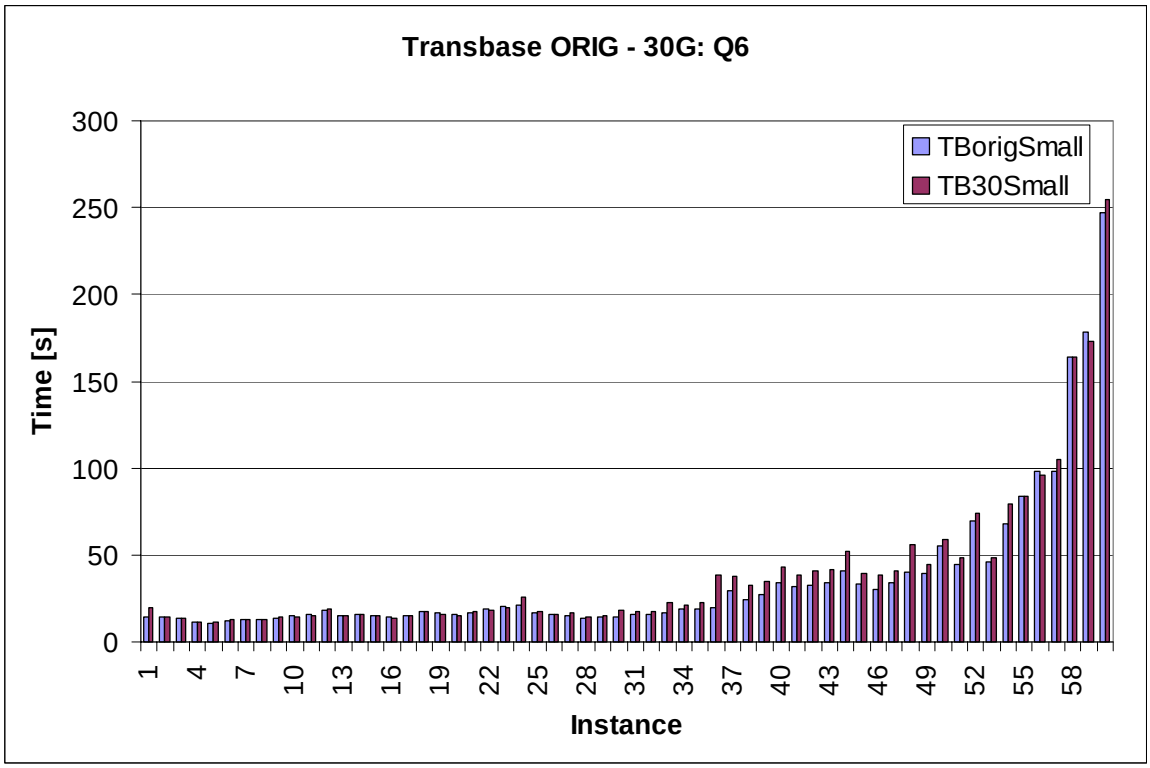


Figure 10-56: Comparison of different Database Sizes for Template Q6

The first 36 queries restrict one month again, the remaining queries restrict up to three years. Most of the queries have the same execution time for both database sizes. However, some of them take longer for the *TB30Small*. This comes from the modified clustering due to the inserted new data. The new data was generated by only modifying the calendar dates. The remaining dimensions are the same. Thus, tuples of later calendar dates are stored in z-regions of the UB-Tree with data of the query result. We therefore have to read more pages, in order to evaluate the queries.

For example, one page of *TBorigSmall* contains the tuples

p_1: (“2003-01-23”, p_1, w_1), (“2003-01-23”, p_2, w_2), (“2003-01-23”, p_3, w_3), (“2003-01-23”, p_4, w_4)

After the enlargement of the data for TB30Small the page is split into two pages:

p_1’: (“2003-01-23”, p_1, w_1), (“2003-01-24”, p_1, w_1), (“2003-01-23”, p_2, w_2), (“2003-01-24”, p_2, w_2)

p_1’: (“2003-01-23”, p_3, w_3), (“2003-01-24”, p_3, w_3), (“2003-01-23”, p_4, w_4), (“2003-01-24”, p_4, w_4)

The same query (restricting to “2003-01-23”) accesses two pages p_1’ and p_2’ for *TB30Small* instead of one page p_1.

Some queries are evaluated faster for the large database size. This also results from the different clustering properties. In general, the scalability is good. There are only minor effects to the query execution time for larger database sizes. A complete list of measurement result figures for the query templates Q1 to Q13 is shown in Appendix E.

10.9.4 Comparison with another commercial DBMS

In this section we compare the implementation of MHC in Transbase® with the implementation of a different technology, i.e., bitmap index intersection and star transformation, of a well known and popular commercial DBMS in its latest version with the best configuration according to our knowledge. Note that we are not allowed to mention the name of this DBMS due to license restrictions and call it *CommDBMS*.

We built the same star schema with bitmap indexes in *CommDBMS*. In the presented measurements we compared the original database size (3 GB fact table raw data). We also ran the queries on the 30 GB size, but no significant differences occurred. The queries are the same as described before and Appendix E contains the complete results for the templates Q1 to Q13 and Q208.

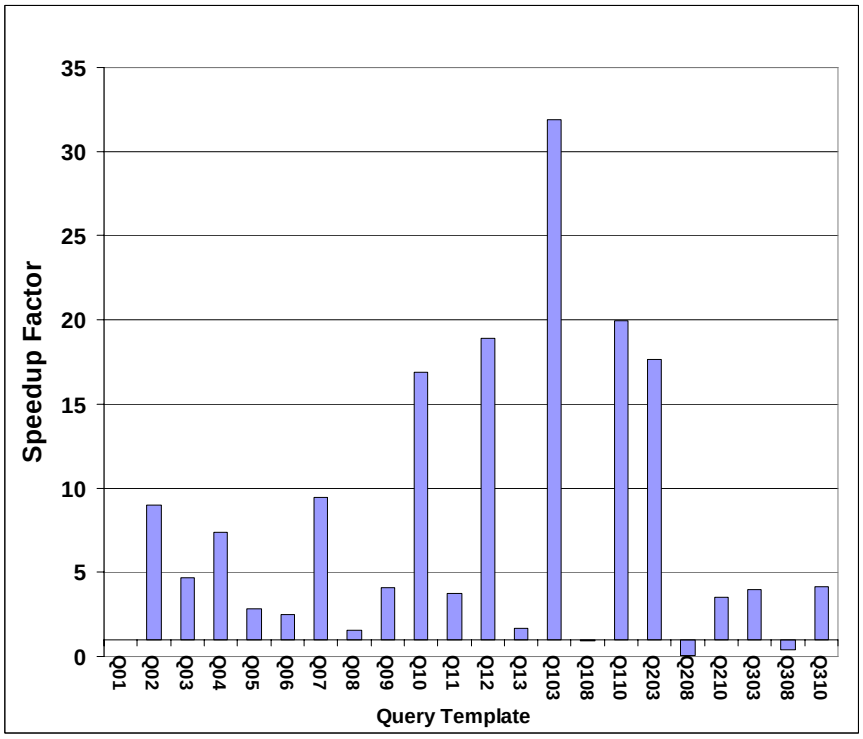


Figure 10-57: Speedup of Transbase® MHC vs. CommDBMS for all Query Templates

Figure 10-57 shows the speedup of Transbase® with MHC compared to *CommDBMS*. Each query template is listed in the figure. The templates where Transbase® is faster are pictured with bars above the 1 (equal) line. For templates where CommDBMS is faster (Q108, Q208 and Q308), the bars are below the equal line. A speedup of two means that Transbase® needs half the time to run all queries of the corresponding query template. A speedup of 0.5 means that *CommDBMS* needs half the time to run all queries of the corresponding template.

Note that the speed up is calculated on the aggregated query exeuction times per template. There are some templates with a very high speedup (Q10, Q12, Q103, Q110 and Q203), i.e., from an order of magnitude up to a speedup of 32. For these queries, the MHC technology is very beneficial.

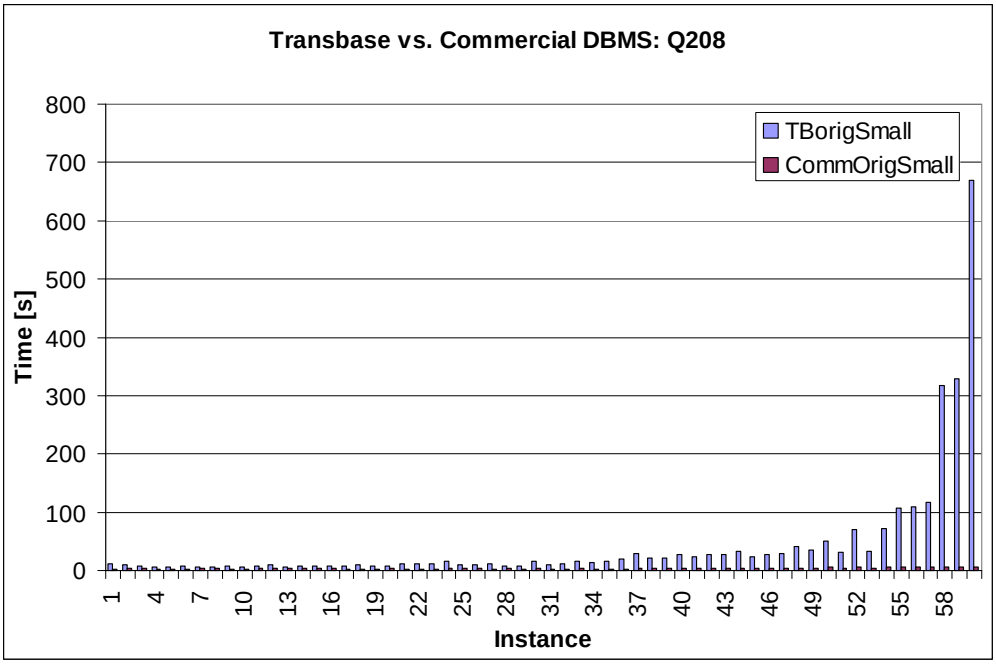


Figure 10-58: Transbase® compared to CommDBMS for Template Q208

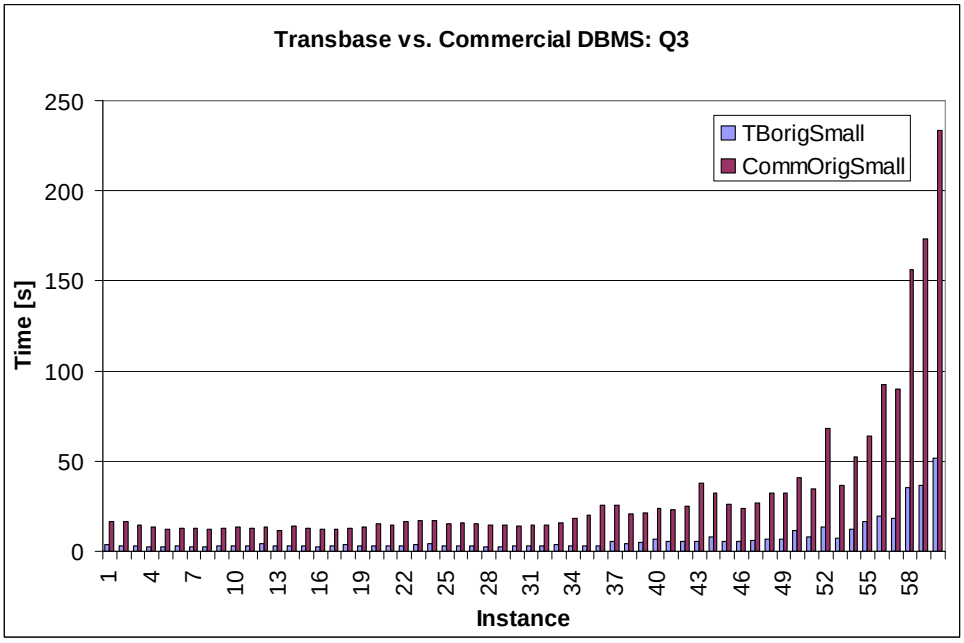


Figure 10-59: Comparison of Transbase® and CommDBMS for Template Q3

The query templates where *CommDBMS* is faster, secondary dimensions are used for grouping and restriction. This leads to expensive post-filtering after the fact table access via residual joins on a larger number of tuples (see Section 10.5). Figure 10-58 shows one of these templates in more detail. *CommDBMS* can use also the restrictions on the secondary dimensions via bitmap index intersection (all dimensions are indexed by bitmap indexes) and therefore does not suffer from secondary dimensions.

Most of the results are similar to Figure 10-59 where Transbase® is faster for all queries within the template.

In Figure 10-60 the cost based optimizer of *CommDBMS* choses a different plan for some queries (e.g., query 38) due to the analysis of the restrictions and statistics. Obviously that is a bad plan.

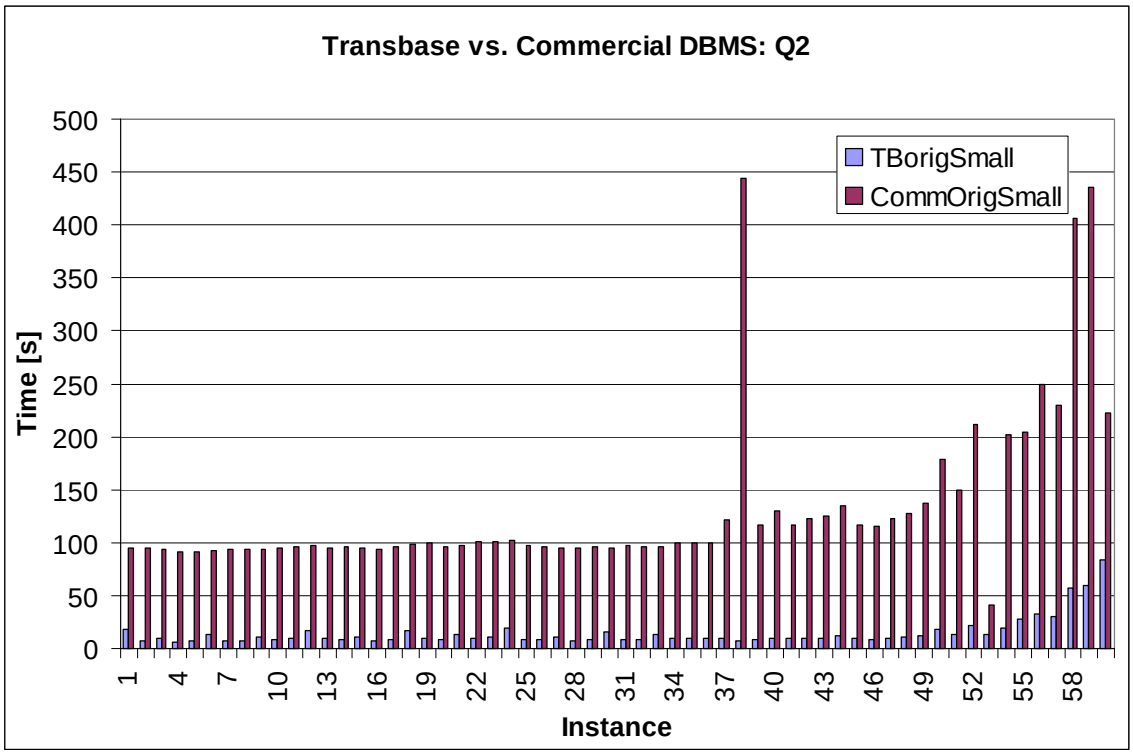


Figure 10-60: Comparison of Transbase® and CommDBMS for Template Q2

For the complete list of queries and results please refer to Appendix E.

10.9.5 APB Benchmark

In addition to the real world Sales DW example, we compared the MHC technology for a standard benchmark, the APB benchmark. The APB benchmark ([APB98]) is a standard benchmark for OLAP. It defines the whole process of loading, updating and querying a data warehouse. We compare the performance of Transbase® with MHC and *CommDBMS* (see Section 10.9.4).

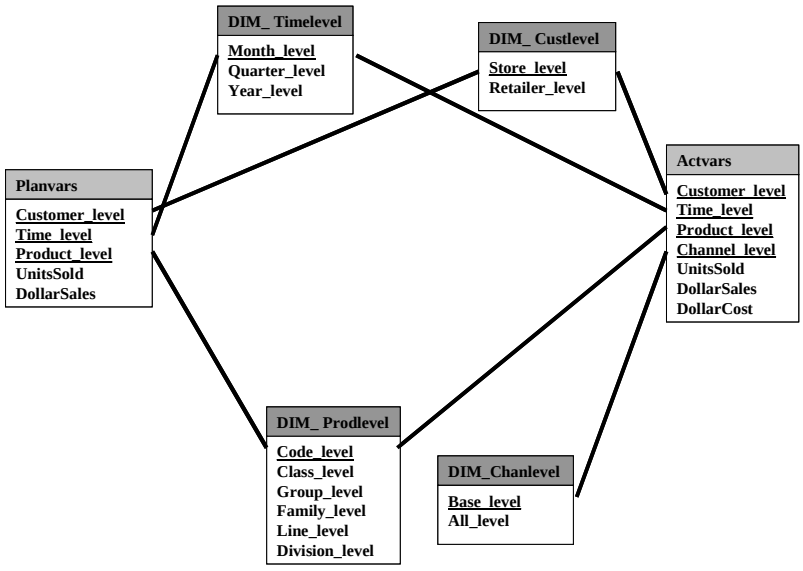


Figure 10-61: Logical Schema of the APB Benchmark with Main Fact Tables

The schema consists of two main fact tables, i.e., *Actvars* and *Planvars* and four dimensions *Time, Customer, Product,* and *Channel* (see Figure 10-61). The hierarchies of the dimensions are shown in Figure 10-62.

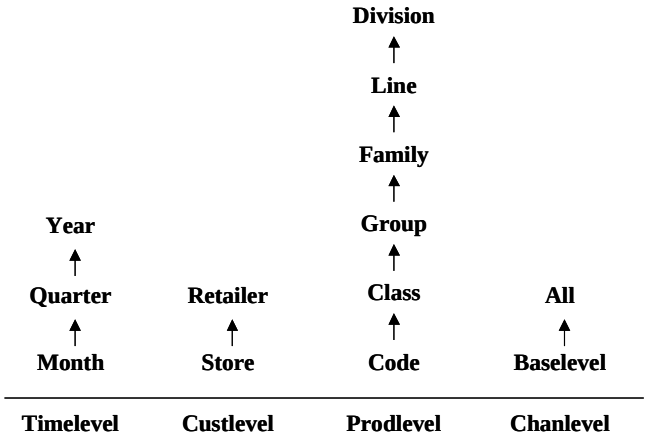


Figure 10-62: Hierarchies of the Dimensions

In addition to the main fact tables, four further fact tables are used in the APB benchmark (see Figure 10-63): *CurrInventory, HistInventory, StdShipCost,* and *StdProdCost*. For a detailed explanation about the economical background please refer to [APB98].

We implemented the APB benchmark with scaling factor 10, i.e., 124 million fact table records for the largest fact table *Actvars*. All data is stored on one disk. Table 10-3 shows the cardinality of all tables and the sizes in MB or GB for Transbase®. For *CommDBMS*, the sizes are very similar. We used MHC for all fact tables with all dimensions as specified in the schema. The compound surrogates of the dimension tables have the hierarchies as shown in Figure 10-62. For *CommDBMS*, we used bitmap indexes on the dimension attributes of the fact tables.

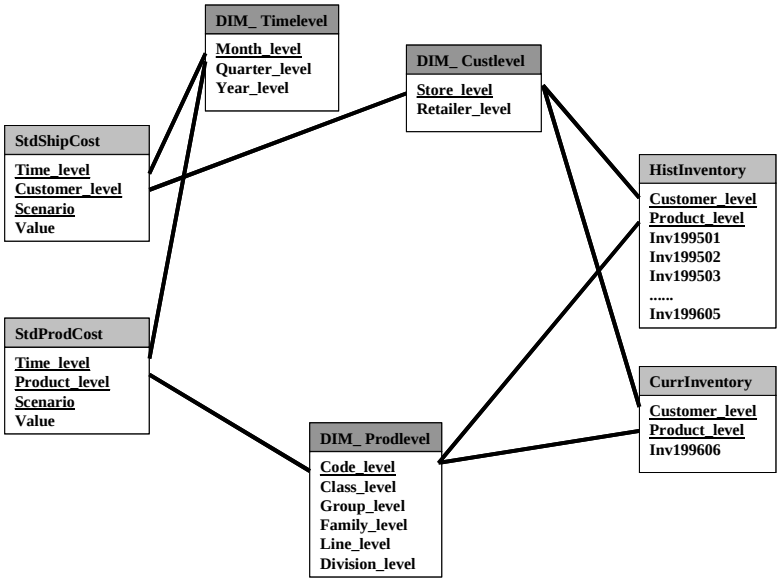


Figure 10-63: Logical Schema of the APB Benchmark with further Fact Tables

The measurements were performed on a two processor PC Pentium II, 400 MHz, with 768 MB RAM and a SCSI hard disk. Operating system is Windows 2000. We eliminated cache effects by restarting the DBMS before every query.

Fact Table	Cardinality	Size		Dimension Table	Cardinality
Actvars	123.930.000	15,6 GB		Product	9.000
Planvars	44.311.800	4,2 GB		Customer	900
CurrInventory	3.692.650	282,0 MB		Time	24
HistInventory	3.692.650	562,3 MB		Channel	9
StdShipcost	27.000	2,2 MB			
StdProdCost	270.000	22,6 MB			

Table 10-3: Sizes of the Tables of the APB Benchmark for Transbase®

In the APB benchmark 10 query templates are defined modelling economic processes such as channel sales analysis, customer margin analysis etc. In data warehouses usually different kinds of queries occur:

- Static, long running reports and analyses executed in regular time intervals and
- Dynamic ad hoc queries started by the user.

The instances of one query template differ in the restricted hierarchy level and in the value of the restrictions. Thus, the instances of one query template have different selectivities. Some of the queries consist of several single queries, e.g., query template Q08 consists of four parts, each of them is a separate SQL statement:

- Q08-P1: part 1 of the query
- Q08-P2: part 2 of the query
- Q08-P3: part 3 of the query
- Q08-Prep: preparation SQL statement for the query

The number of instances pertemplate is between 125 and 500. The overall number of queries is 6.251.

Figure 10-64 shows the comparison between Transbase® and *CommDBMS* for each query template. It contains the 1st, the 2nd and 3rd, and the 4th quartile of the speedup of the instances for each template. For example, for template Q05, the minimum speedup of Transbase® compared to *CommDBMS* is 0,53 (*CommDBMS* needs only half the time), 50% of the queries have a speedup between 13 and 18 (Transbase® is between a factor of 13 and 18 faster than *CommDBMS*) and the maximum speedup is 28,5.

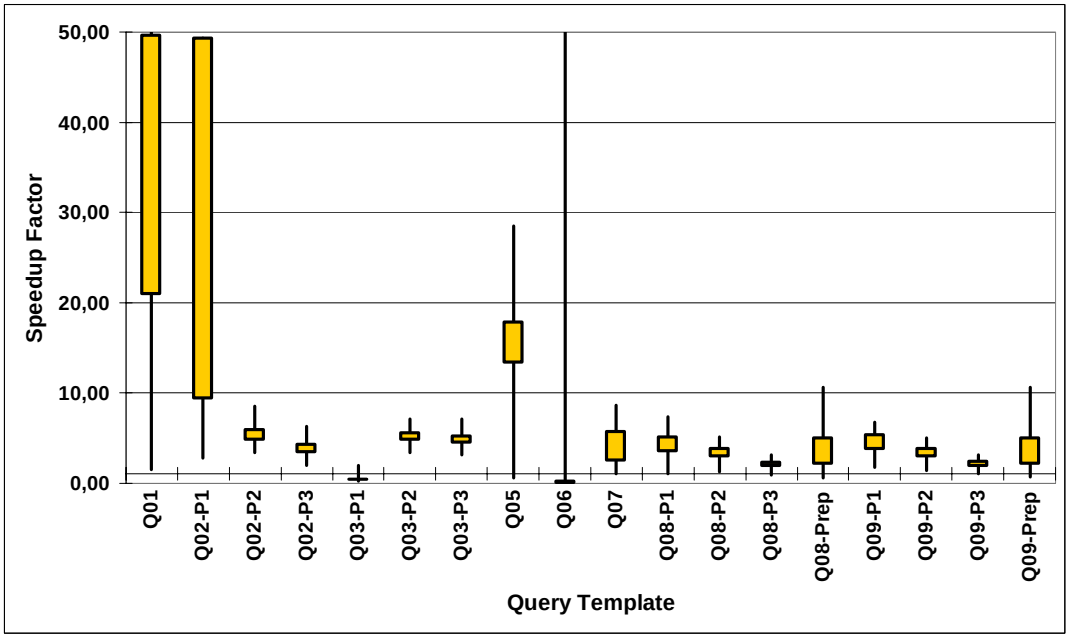


Figure 10-64: APB Benchmark: Speedup of Query Templates

Most of the queries are significantly faster for Transbase® compared to *CommDBMS*. Only the queries of Q03-P1 and Q6 are slower. In these queries, only two dimensions out of four are restricted. Most of the instances of the remaining query templates are between a factor of two until a factor of 50 (and even more) faster.

11 Handling Complex Hierarchies

11.1 Transforming Hierarchy Instances to Simple Hierarchies

Queries that restrict dimensions, have predicates on hierarchy levels. Such 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). These point sets can be replaced by a smaller set of interval restrictions depending on the predicate.

EHC is useful to transform a set of hierarchy paths to a small set of intervals (see Section 6). However, this encoding only is suitable for simple hierarchies. For complex hierarchies, it is necessary to select one of several simple hierarchies of the complex hierarchy. Therefore, restrictions on the other hierarchies probably will cause relatively bad performance. A large number of query boxes can be the consequence (see Section 10.3).

To enable hierarchy encoding for such a complex scenario, we present the transformation algorithm HINTA that transforms a DW-hierarchy to a simple hierarchy, preserving hierarchical relationships of the original levels. This algorithm has first been described in [PMB00].

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 2-2.

11.1.1 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 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 11-1 illustrates all possible hierarchy schemata of phi (phi_1, phi_2, phi_3).

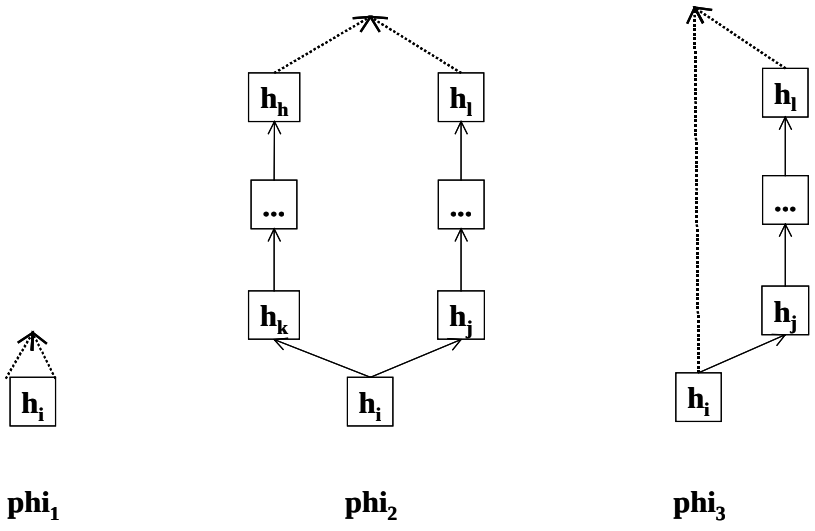


Figure 11-1: Hierarchy Schema for Primitive Hierarchy Instances

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 $h_k, \ldots, h_h$ for H_1 and $h_j, \ldots, h_l$ 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.

phi3 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 $h_j, \ldots, h_l$.

In Figure 11-2, the 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, C, D, 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 of type phi_2, and the alternative paths J and $J \rightarrow I$ are of type phi_3.

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

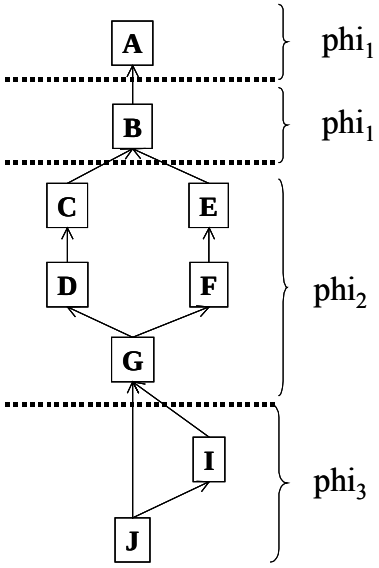


Figure 11-2: Example of phi's for a Hierarchy Schema

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

Definition 11-1 (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 11-1 (Primitive Hierarchy Instance):

This example shows the phi's of the sample hierarchy instance *H* of Figure 11-3. Figure 11-4 shows the schema of the hierarchy. *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 11-6).

H^{p3} is of type phi_2 and consists of two alternative paths with shared leaf level *Outlet* (see Figure 11-7)

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

E^{phi3} = {(A1, $Aldi_N$), (A2, $Aldi_N$), (S1, $Saturn_N$), (S2, $Saturn_N$), (A3, $Aldi_S$), (H1, $Hofer_E$), (H2, $Hofer_E$), (S4, $Saturn_E$), (H3, $Hofer_W$), (H4, $Hofer_W$), (S5, $Saturn_W$), (S6, $Saturn_W$),
(A1, TG1), (A2, TG2), (S1, TG5), (S2, TG2), (A3, TG2), (H1, TA2), (H2, TA1), (S4, TA1), (H3, TA1), (H4, TA1), (S5, TA2), (S6, TA1),
($Aldi_N$, North), ($Saturn_N$, North), ($Aldi_S$, South), ($Hofer_E$, East), ($Saturn_E$, East), ($Hofer_W$, West), ($Saturn_W$, West),
{(TG1, Germany), (TG2, Germany), (TG5, Germany), (TA1, Austria), (TA2, Austria),
(North, Germany), (South, Germany), (East, Austria), (West, Austria)} □

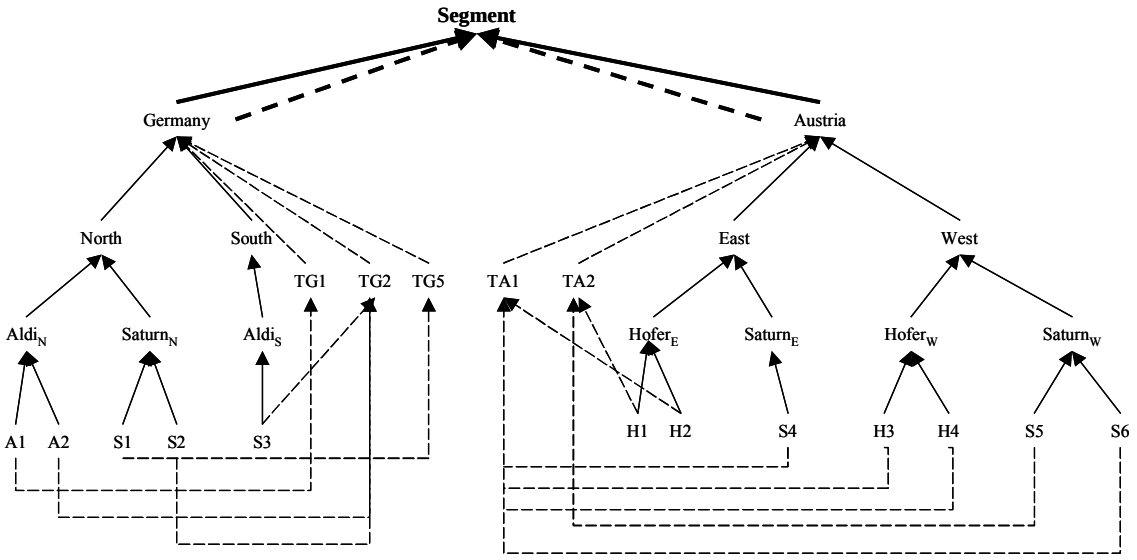


Figure 11-3: Sample Hierarchy

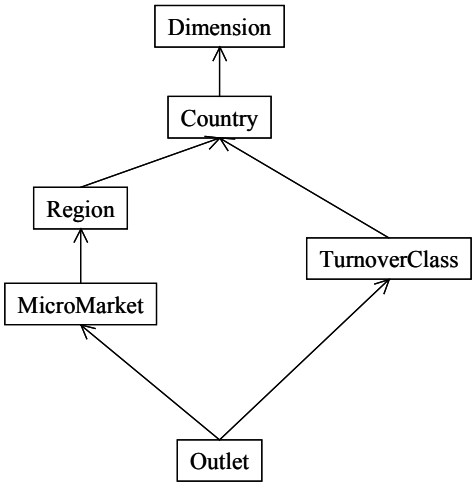


Figure 11-4: Schema of Sample Hierarchy

11.1.2 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) 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 algorithm is specified in pseudo code:

`TransformPhiToSimpleHierarchy:`

```
if type(H^phi)=phi_1, then
    forall edges (v_i, v_x)
        if not isPreferred(v_i, v_x) then
            delete edge(v_i, v_x)
        /* delete leaving edges of the non-preferred hierarchy,
           leaving edges of preferred hierarchy remain*/
if type(H^phi)=phi_2, then
    forall edges (v_l, v_x)
        if not isPreferred(v_l, v_x) then
            delete edges (v_l, v_x)
            /* delete leaving edges of the non-preferred
               hierarchy */
    forall edges (v_i, v_k)
        if not pathexists(v_i→v_j'→…→v_l'→v_k)
            insertpath(v_i→v_j'→…→v_l'→v_k)
```

```
        delete edges (v_i, v_k)
    forall edges (v_i, v_j) delete edges (v_i, v_j)
    /* make v_k indirect hierarchically dependent on v_i
       (instead of direct hierarchically dependent) by
       duplicating vertices and edges */
if type(H^phi)=phi_3, then
    for all edges (v_i, v_x)
        delete edges (v_i, v_x)
```

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 11-5). The operation `insertpath(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.

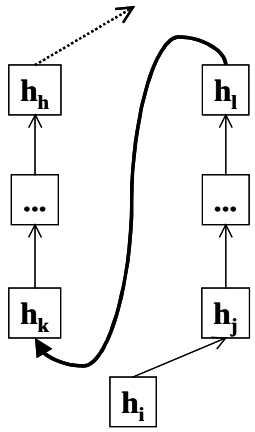


Figure 11-5: Transformation of a phi_2

11.1.3 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 hierarchies by splitting them into *phi's* and transform each *phi* with `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 proceeding in a more formal way:

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)) ∪
                    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.

11.1.4 Example of HINTA

To illustrate HINTA, we use the hierarchy instance $H=(V^H, E^H)$ of Figure 2-2, where *H* contains two simple hierarchies H_1^S and H_2^S (see Example 2-5). 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)

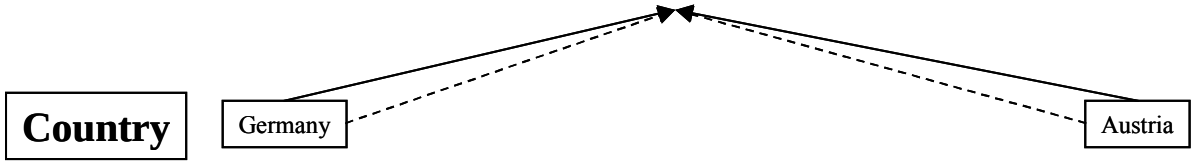


Figure 11-6: Schema and Instance of H^{phi2}

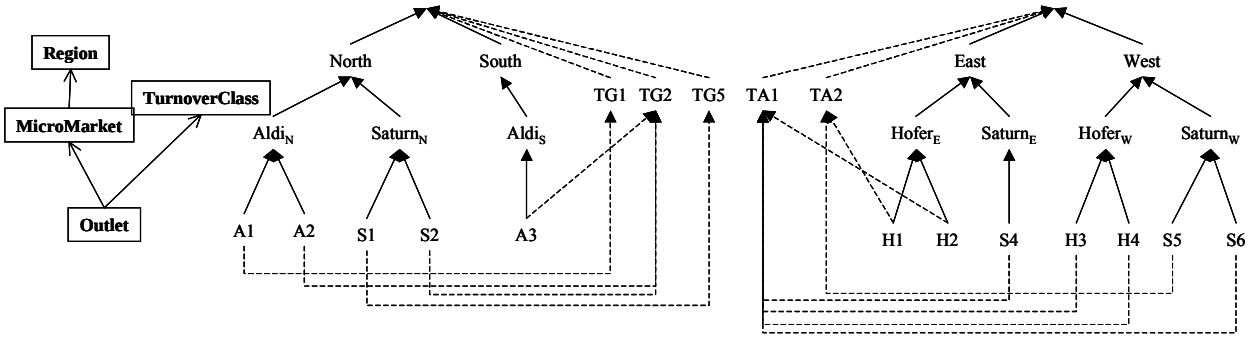


Figure 11-7: Schema and Instance of H^{phi3}

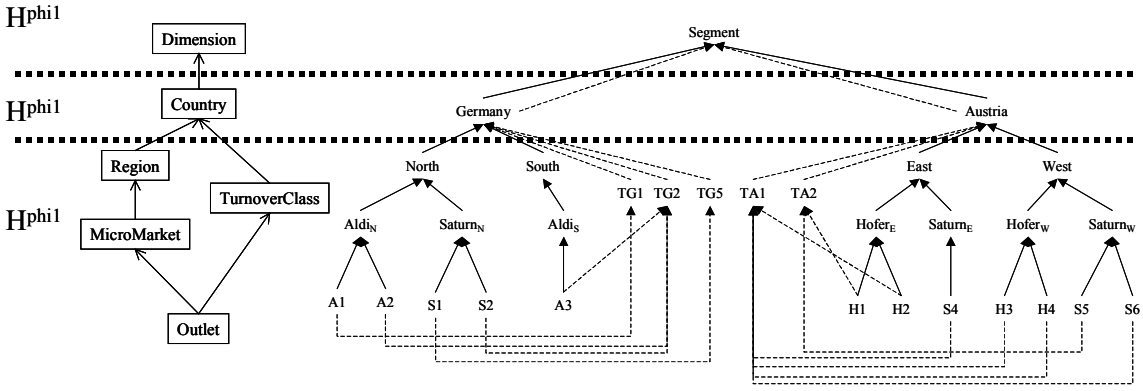


Figure 11-8: Transforming the Hierarchy

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 11-6 and Example 11-1).

H^{p3} *=(V, E)* (of type *phi*$_2$) consists of two alternative paths with shared leaf level *Outlet* (see Figure 11-7 and Example 11-1).

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)}*

Figure 11-9 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}*), (TG5*^{1}*, Saturn*$_N$*), (S2, TG2*^{2}*), (TG2*^{2}*, Saturn*$_N$*), (A3,TG2*^{3}*), (TG2*^{3}*, Aldi*$_S$*), (H1, TA2*^{1}*), (TA2*^{1}*, Hofer*$_E$*), (H2,TA1*^{1}*), (TA1*^{1}*, Hofer*$_E$*), (S4, TA1*^{2}*), (TA1*^{2}*, Saturn*$_E$*), (H3, TA1*^{3}*), (H4, TA1*^{3}*), (TA1*^{3}*, Hofer*$_W$*), (S5, TA2*^{2}*), (TA2*^{2}*, Saturn*$_W$*), (S6, TA1*^{4}*), (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}*), (H4, TA1*^{3}*), (S5, TA2*^{2}*), (S6, TA1*^{4}*),*
(TG1^1*, Aldi*$_N$*), (TG2*^{1}*, Aldi*$_N$*), (TG2*^{2}*, Saturn*$_N$*), (TG2*^{3}*, Aldi*$_S$*), (TG5*^{1}*, Saturn*$_N$*), (TA1*^{1}*, Hofer*$_E$*), (TA1*^{2}*, Saturn*$_E$*), (TA1*^{3}*, Hofer*$_W$*), (TA1*^{4}*, Saturn*$_W$*), (TA2*^{1}*, Hofer*$_E$*), (TA2*^{2}*, Saturn*$_W$*),*
(Aldi$_N$*, North), (Saturn*$_N$*, North), (Aldi*$_S$*, South), (Hofer*$_E$*, East), (Saturn*$_E$*, East), (Hofer*$_W$*, West), (Saturn*$_W$*, West),*
(North, Germany), (South, Germany), (East, Austria), (West, Austria)}

Figure 11-10 shows the new hierarchy instance with the inserted members and edges (including the hierarchy schema).

The resulting simple hierarchy of *HINTA* is the union of H^S_1, H^S_2 and H^S_3, as illustrated in Figure 11-11.

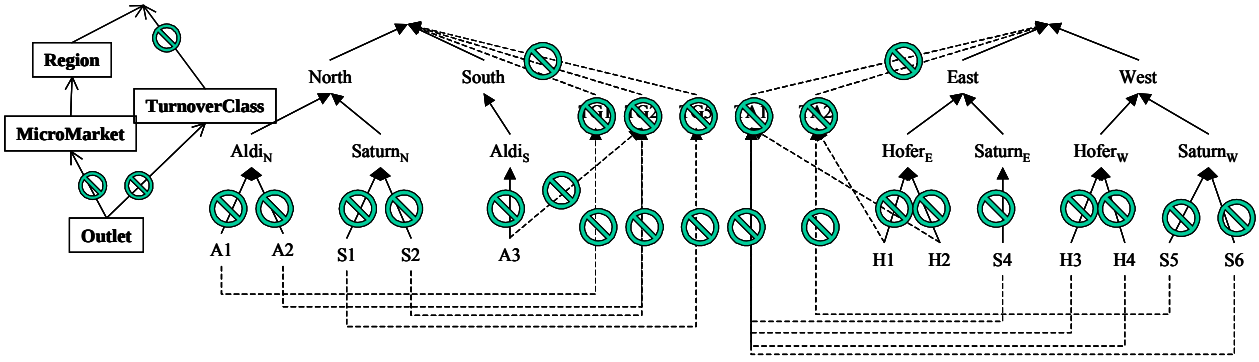


Figure 11-9: Deleting Members and Edges

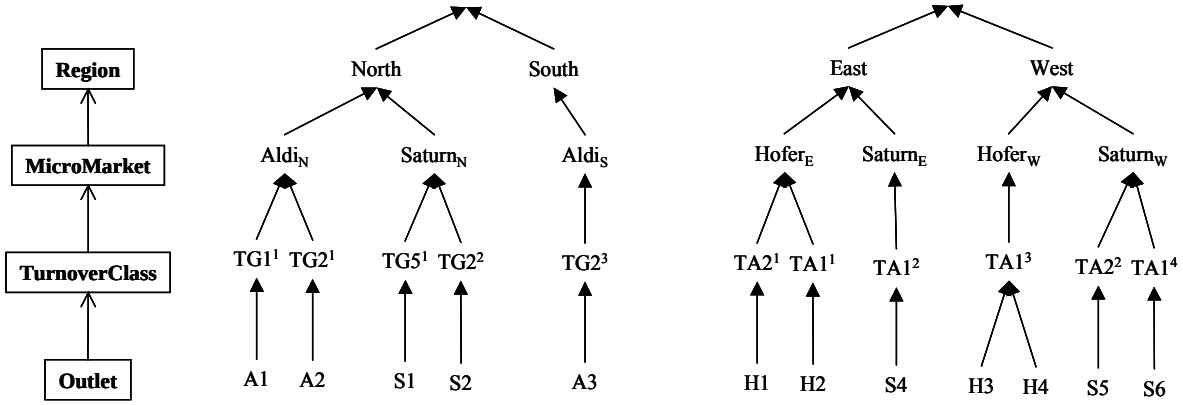


Figure 11-10: Inserting Members and Edges

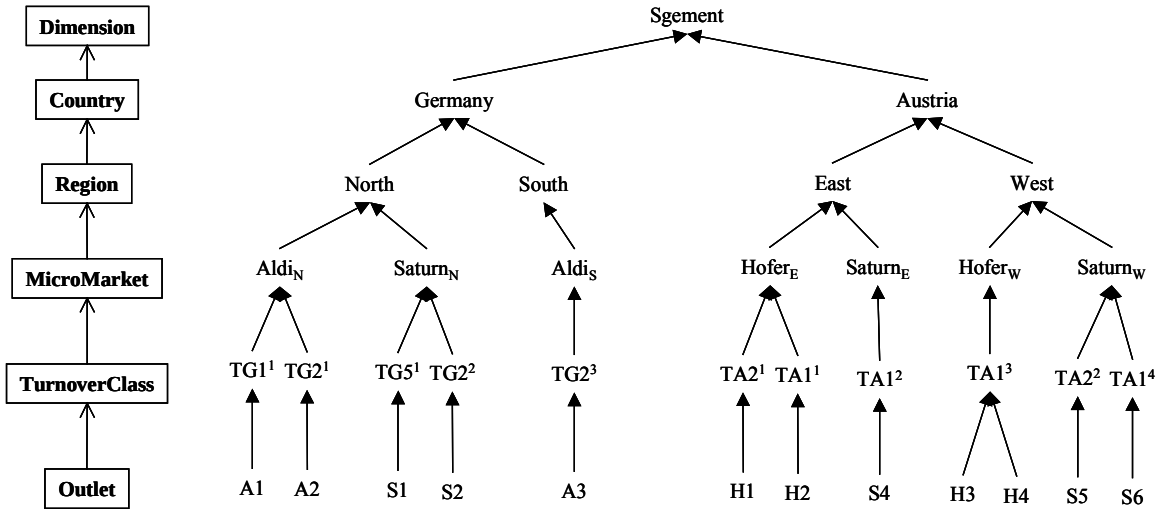


Figure 11-11: Final Simple Hierarchy Instance and Schema

12 Changes on Hierarchies (dynamic hierarchies)

Usually, dimensions in data warehouses have a static nature. This means that changes on dimension tables are very seldom or even do not occur. For example, a calendar dimension will not be changed, because the days of May 2002 will always belong to May 2002 and not to another month. However, some dimensions may change, e.g., a customer *c* can move to another city or country within the customer dimension. In this case, the hierarchy is modified, i.e., the path of *c* is modified. In the MHC schema, a change of a hierarchy results in a re-calculation of the compound surrogate and a (technical) deletion and insertion into the dimension table is performed (see Section 8.2.3). A MHC organized dimension table requires a reservation of a specific number of bits for each hierarchy level. This number corresponds to the maximum number of children (from a higher hierarchy level) that this level can hold. If this number is exceeded, the calculation of the compound surrogate fails (overflow).

In such a case, we must reorganize the dimension table. A reorganization requires the re-calculation of all compound surrogates in the dimension table which actually is a rebuild of the complete dimension table.

The problem with this approach is that the fact table contains reference surrogates, each of them references a compound surrogate in the dimension table. Thus, all tuples of the fact table have to be updated, too.

This section describes how to cope with the problem of surrogate overflow. We first describe how to avoid overflow as long as possible for deleting and update scenarios (Section 12.1). Then we introduce a method to handle overflow of surrogates without rebuilding the dimension and the fact table (Section 12.2). In Section 12.3 we present a script generator which easily supports all kinds of restructuring a dimension hierarchy including the necessary rebuilding steps on the database side. We further discuss a broader framework for autonomic MHC maintenance in Section 12.4 and finally mention some approaches that we have followed, but which unfortunately do not work in general (Section 12.5).

12.1 Reusing Deleted Surrogates

The calculation of compound surrogates assigns to each new member m_i^j of hierarchy level h_i the next higher surrogate, e.g., $surr(m_i^j) = s+1$, where $surr(m_i^{j-1}) = s$ and m_i^{j-1} is the highest member so far within the specific path. Thus, if a member that is not the last member of the hierarchy level is deleted, the unoccupied surrogate is not immediately reused but each new member is assigned a surrogate larger than the deleted one. The reason is that finding a new unused surrogate by the max+1 method is well supported by the surrogate index (see Section 8.1 for more details).

To reuse a deleted surrogate, the holeSearch is used as described in Section 8.1.3. This method finds holes in the sequence of used surrogates on an arbitrary member level. The overhead to find a hole is larger than the simple assignment of the next higher surrogate, so the hole search is done only when an overflow has occurred on a certain hierarchy level.

This method does not avoid overflows of a hierarchy level, but guarantees that even in the event of deletions and updates, the complete space as defined by the user is exhausted before an unavoidable overflow occurs.

12.2 Overloading of Surrogates

Overloading of surrogates is a technique which does not require redefinition of existing surrogates. So the fact table does not have to be rebuilt.

12.2.1 Principles of the Method

We allow a non-injective mapping from hierarchy members to surrogates. This means that several different values of a hierarchy field get the same surrogate encoding. As a special technique, for each level of the hierarchy, one special bit combination (e.g. 11..11) is reserved to encode all values for which no free surrogate combination has been found. Thus, all overflow values are mapped to this special bit combination.

E.g., let *D* be a leaf dimension table with hierarchy fields h_4, h_3, h_2, h_1 and the corresponding compound surrogate field *cs*. We assume for simplicity that two bits are used to encode each level resulting in surrogates with a length of eight bits. *F* is a fact table with reference surrogate field cs_{ref} referencing *cs* of *D*. Suppose that in dimension *D*, level h_2 has overflown and the bit combination 11 (as the third part of the surrogate) encodes all overflows of that level.

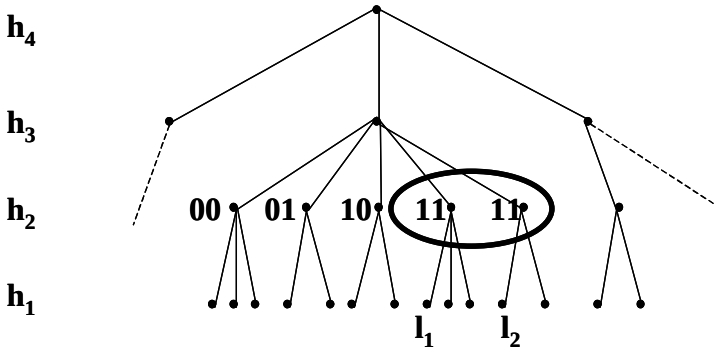


Figure 12-1: Hierarchy Tree with Surrogate Overloading

In the above example, we have one overflow element on hierarchy level h_2. The two leaves l_1 and l_2 now have the same surrogate encoding.

Surrogate overloading has some impacts on the query processing algorithms. The immediate consequence of this non-unique mapping is that, in the fact table, a reference surrogate cs_{ref} does not uniquely encode the hierarchy field values of the corresponding leaf dimension table.

For the sake of clearness, we say that

- a *cs* value has an overflow value on level h_i, if the compound surrogate component which belongs to level h_i is the bit combination which encodes all overflow values of level h_i ,
- a *cs* value has an overflow value if it has an overflow value on any of its level.

Recall that in the MHC processing schema, first the local restrictions on dimension side are evaluated and the results are internally represented by one or more compound surrogate intervals. Then the fact table is accessed using the *cs* intervals. For our discussion, it is sufficient to think of the *cs* intervals as a set of single *cs* values which represent the result set on dimension side.

It is clear now that with the above encoding schema, the result set on dimension side is no more uniquely representable as a set of *cs* values. In the above example, if a result value *v* has a surrogate encoding cs_v which has the 11 bit combination as the third part, the access to the fact table via cs_v produces result values on the fact table which might contain another value *v'* which has the same encoding but is not in the result set on dimension side. This can easily been seen in the above example if l_1 is in the result set but l_2 is not.

12.2.2 Introduction of Postfiltering by Surrogate Overloading

Therefore, in the general case, a superset of the desired result is materialized in the fact table. Of course it is possible to reduce the superset to the desired fact table result by a postfiltering approach

(compare with secondary dimensions in Section 10.5). This requires a residual join with *D* as shown in Figure 12-2. This execution plan is discussed in more detail in Section 9.

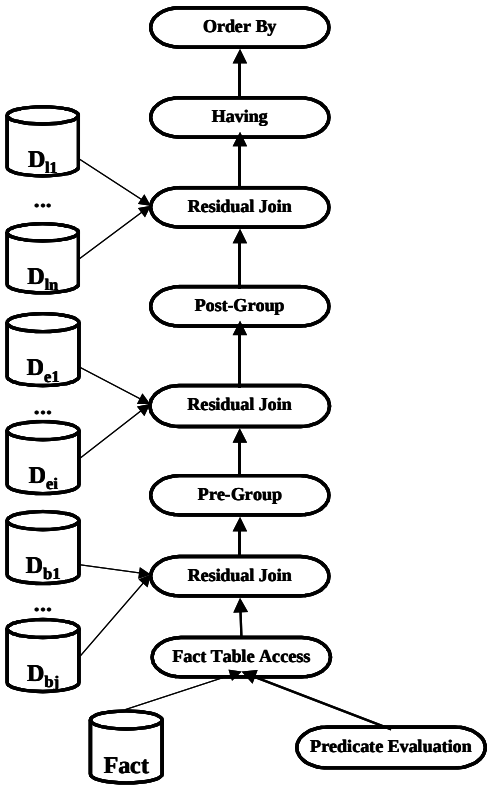


Figure 12-2: Overall Execution Plan

Obviously, the residual join before grouping is a drawback but the mechanism can be refined such that the additional residual join often can be omitted. This is described in the next two sections.

12.2.3 Suppression of Residual Join with Surrogate Overloading

For suppression of postfiltering (residual join), a rule can be formulated which can be checked statically. Informally, if overloading of surrogates occurs on a hierarchy level h_{ol} then predicates on higher levels are not affected. More precisely, if the hierarchy search degree of the search predicate (Section 10.1.4) has the value *sdg* and *sdg* > *ol* for all overloaded levels h_{ol} then postfiltering is not necessary. Thus, if we can avoid overloading in the higher hierarchy levels by reserving more bits for the surrogates, residual joins can be suppressed more likely.

The reason is that in the above case, the *cs* result set on dimension side consists of subtrees of height *sdg* – so whenever a non-uniquely coded value appears in the subtree then also all other values with the same encoding are in the result set. Thus, only hits (and of course all hits) are materialized in the fact table by the corresponding reference surrogates.

In our example, a search predicate on hierarchy level h_3 or h_4 would not require postfiltering by residual join whereas a predicate on h_2 or h_1 would require it.

If postfiltering can be suppressed then the next point of interest is the applicability of pre-grouping (Section 9.4). Similar arguments apply here. If the grouping fields have hierarchy degree greater than *ol* then the described pre-grouping techniques can be applied. This is because pre-grouping then is performed on surrogate prefixes which do not contain overloaded bit combinations.

It turns out that overloading of surrogates on a higher hierarchy level is worse than on a lower level because – with a high level - more query types have a performance penalty.

12.2.4 Additional System Requirements of Surrogate Overloading

Note that the technique of surrogate overloading requires additional entries in the system tables. This is due to two reasons: First, on insertion it must be clear which surrogate levels are allowed to be overloaded (and which bit combinations hold the overflows). Second, the query optimizer has to know about the hierarchy levels where overflowing is permitted to decide about optimal or suboptimal access plans on the fact table.

12.2.5 Suppression of Postfiltering with Surrogate Overloading - Dynamic Rules

The preceeding section described a sufficient rule for exactness of the fact table result in case of surrogate overloading. Another approach is the dynamic analysis of the result set on leaf dimension side from the dimension predicates. This means that the result set is analyzed before the fact table access is done.

We can state the following obvious rule:

Suppression of postfiltering is possible if for each v in the result set L of the dimension predicates it holds that L also contains all result values which have the same surrogate encoding as v.

This can only be checked at runtime. Consequently, it can only be decided at runtime whether postfiltering is necessary for the fact table result. In this case, a dynamic optimizer is necessary.

12.3 Redefinition of Surrogates

This section describes the steps which are necessary to reorganize the leaf dimension table in a most flexible manner. If the hierarchy changes then a recomputation of surrogates is the consequence.

The natural way to modify hierarchies and compound surrogates is the following sequence of actions:

- Spool the fact table to file.
- Drop the fact table.
- Spool the dimension table to file.
- Drop the dimension table.
- Create the dimension table (with modified compound surrogate definition).
- Spool the data into the dimension table.
- Create the fact table.
- Spool the data into the fact table.

Of course, this sequence of actions can be performed automatically when a redefinition of surrogates is inevitable.

Traditionally, Transbase® has supported a script generation for a most general redefinition of a table. The script generator is located in the interactive frontend *TBI* and (given a table name) creates a sequence of statements which is analogous to steps 3,4,5,6. The generated table redefinition script recreates the table in identical form. For the user, it provides the basis for desired changes of the table, for example adding or dropping columns or changing field types.

For MHC, the script generator has been extended. When a leaf dimension table with compound surrogates is provided to the generator, it creates the above sequence of actions (1. – 8.), in order to retain the consistency of compound surrogates and reference surrogates. If several fact tables reference the (modified) dimension table, we generate reloading for each of them.

12.4 Deferred fact table update

As mentioned in previous sections, updates on the dimension tables lead to cascading updates on the fact table if surrogates of dimension members have to be recomputed. Consequently, a small update on the dimension table may lead to a large and expensive operation on the fact table leading to large execution times of the update statement. As both steps, the dimension update and the triggered fact table update, have to be performed in one transaction, this may also influence multi-user performance, e.g., queries can not be processed during update. This is tolerable as long as the maintenance window is large enough, but in "24-7" environments other concepts have to be used.

One solution to this problem is to decouple the two steps, i.e., the dimension update and the updates on the fact table. In case of MHC this is possible, as the update on the dimension table only triggers a re-clustering of some tuples in the fact table – the tuple values (except the surrogates which determine the clustering) are not changed. Instead of doing the reorganization of the fact table online a background process does this in idle time. In the following, we briefly sketch the architecture of such an deferred update mechanism and address the main problems that have to be solved.

In order to break the complete update into two (or more) separate steps, it is necessary that the overall transactional semantics is already achieved after the first step. In case of MHC this means that it has to be guaranteed that the fact table tuples are correctly associated to the dimension members. In case of MHC processing we use the surrogates instead of the dimension keys to match the fact tuples to the dimension members. The question that arises is how can we guarantee the correct mapping if the surrogates change on the dimension side but we do not want to update the fact table immediately. This is achieved by introducing – theoretically - a level of indirection: for each dimension member we keep two surrogates, the current one and the last one. If these two are not identical, then the fact table has not been reorganized according to the latest dimension update, yet. Consequently, the old surrogate value has to be used to access the fact table. Thus it is guaranteed that queries always return the correct results. However, as the clustering of the fact table may not always reflect the latest dimension status, a small performance penalty has to be paid. A background process of the DBMS has then the possibility to reorganize the fact table in idle time. The smallest granularity is the reclustering of all fact tuples corresponding to a specific dimension member. Thus the locking overhead and the resulting influence on the multi-user performance is reduced.

For the implementation of such a scheme, the MHC data model has to be revised, adding a second surrogate field to the dimension tables. Also the query processing algorithms have to be extended to detect non-propagated dimension updates in order to create the correct surrogate restrictions on the fact table. An open issue is the extension of the advanced pre-grouping algorithms. Even though pre-grouping still can be applied on the fact table results, the post-processing step also has to take care of possible old surrogates. In such cases, more finer groups will be created by the pre-grouping phase, i.e., the intermediate result set size may not be reduced so much, and consequently more merging has to be done after the residual joins. For the deferred updates, the architecture for DBMS background processes has to be designed and implemented.

The concept of deferred fact table updates allows to provide faster response time to dimension table updates with only a modest decrease in query performance.

12.5 Further concepts

In this section, we briefly discuss concepts that have been considered to solve the dynamic MHC problem, but that turned out not to be applicable to the general case. However, we think it is worth mentioning these approaches as they provide nice solutions for some special cases.

If an overflow occurs at some hierarchy level, the fix-sized surrogates have to be extended leading to a complete reorganization of the fact table. The approach of surrogate overloading as described above is only a temporary solution. So the general question that we try to solve is, if there is a way to extend

the fix-length surrogates in a way that the z-order is still preserved but the complete fact table does not need to be reorganized.

12.5.1 Variable-length Surrogates

The problem of overflows stems from the fixed-size of surrogates, or to be more precise, the fix-sized binary representation of the dimension values required for the UB-Tree. Consequently, to overcome this deficit the natural idea is to work with variable length bit representations. This would give each dimension enough room to grow and further provide compression as only as many bits are used as are required to encode the value. The problem with this approach is the ordering function on such bit representations. If one does not want to store expanded bit strings then a lexicographic ordering function has to be used as we know for standard strings. The problem is that lexicographic comparison does not work for bit representations of domains that do not adhere to lexicographic ordering, e.g., all numeric domains. So, variable-length surrogates only work if all dimensions of the UB-Tree adhere to lexicographic order.

12.5.2 Expansion of Surrogates

If we are bound to fix-length surrogates, is there then a possibility of expanding the surrogates without violating the current clustering? The only way to extend the internal representation of UB-Tree keys so that the z-ordering is not violated is to add bits at the beginning of the bit strings. In addition, the comparison function has to be changed to take padded zero bits for the smaller values into account. Thus one dimension, the first dimension in interleaving order to be precise, could be extendable. However, the overall clustering would degenerate into a composite key clustering the more the dimension is expanded.

12.6 Remarks

In the previous sections we have discussed some approaches how to deal with dynamics in hierarchies. Most hierachies have a static nature, for slowly changing hierarchies, it is often not necessary to take care of bit overflow (when using the *hole search* method), because enough bit combinations can be reserved, in order to cover future hierarchies. Since the distribution of the hierarchy paths (see also the distribution of the Sales DW in Section 10.8.4) is very skewed, in most cases there is only one special path that is a candidate for overflows.

However, when reserving a large number of bits for higher hierarchy levels (more than are needed), the physical clustering for MHC will suffer (see Section 10.4). Thus, the best is to analyze the hierarchies and estimate the future behaviour. The techniques to deal with overflow situations have lack of performance (surrogate overloading) or require a large amount of time (reorganization).

13 Summary

In this thesis, we present a complete implementation of MHC into the relational DBMS Transbase® Hypercube and describe general concepts how to process and maintain hierarchically organized data warehouse schemata. Most of the described concepts have been implemented into Transbase® Hypercube. Several installations of Transbase® with MHC are already productive at customers of Transaction Software.

Multidimensional hierarchical clustering (MHC) is an encoding of hierarchy paths, in order to map hierarchical restrictions to interval restrictions. These interval restrictions can be used to efficiently access the records stored in a multidimensional clustering index and thus improve the evaluation of the fact table records. Since the hierarchical encoding is stored in the fact table in a data warehouse application, special optimization techniques are possible. For example, in conventional star query algorithms, grouping is done after the residual join with the dimension tables. With MHC, we can group the fact table records w.r.t. hierarchical attributes without joining the dimension tables and therefore speed up the query processing phase significantly. We call this method hierarchical pre-grouping. Special algorithms are required for correct aggregation in combination with pre-grouping. In this thesis, we describe the basic algorithms in the area of maintaining MHC, interval generation, query optimization, hierarchical pre-grouping and aggregating as well as requirements from real world scenarios, such as complex schemata (snowflake schema), complex aggregation expressions, multiple fact tables (galaxies), large multidimensional query boxes, a large numbers of dimensions (secondary dimensions), and multiple hierarchies per dimension.

The performance comparison of MHC with conventional techniques is very impressive, and the comparison with another very popular commercial DBMS shows the benefit of MHC in the context of Transbase® Hypercube. In combination with other “soft skills” of Transbase® Hypercube and Transaction Software, the results make us confident that the technique and the product will be accepted and used in a large scale.

In the following we outline some experiences, where MHC is beneficial and where not. MHC in combination with multidimensional clustering indexes, e.g., the UB-Tree, is suitable for data warehouse applications with a small number of “important” dimensions. This means that a relative small number of dimensions should be restricted in most queries. In reality, such dimensions are (among others) the calendar, product, or customer dimension. The scalability of MHC is very good, i.e., the query performance does not suffer from larger data sets. We found that in combination with the Transbase® DBMS, the hardware environment can be very small, in order to get acceptable query response times.

However, for dimensions with many alternative hierarchies, the modeling with MHC requires to choose one hierarchy for the physical clustering. Otherwise the number of dimensions for the multidimensional index is very large and the multidimensional physical clustering suffers. We can use HINTA, in order to deal with multiple hierarchies on dimensions, but this is only a “work around”.

If we do not use all dimensions for physical clustering, we have to deal with secondary dimensions. Depending on the number of queries that restrict the secondary dimensions query performance will also suffer.

It turned out that the maintenance overhead of MHC in a data warehouse is acceptable for loading new data. For data warehouses with a very dynamic nature, updates on hierarchical attributes of the dimensions may lead to significant performance problems for the update statements.

The implementation of MHC into the database kernel is quite smooth. A high effort is the extension of the optimizer to consider MHC and the physical clustering when generating query execution plans. Of

course not all optimizer rules are currently implemented in the Transbase® optimizer and a cost based or dynamic approach probably would improve the query execution plans. In this area, a lot of investigation still can be done.

13.1 Future Work

Of course, there is still work to do, especially in the integration of additional and extended concepts in the context of MHC and data warehousing.

13.1.1 Cost Model

Since Transbase® uses a rule-based optimizer, no costs are considered for optimizing query plans. This is a general disadvantage, but is very important for some optimizer decisions that currently may generate bad query plans. A cost model must be developed and used for the optimizer. However, we think that a combination of rule-based and cost-based optimizer is the easiest and most robust way.

13.1.2 Dynamic Optimization

Dynamic optimization has been mentioned several times in this thesis. However, not much research is available in the field of dynamic optimization and very new optimizer concepts and structures must be implemented. We think, that there are some application areas, especially in the field of optimizing MHC processing, where dynamic optimization would gain a significant advantage compared to rule-based and cost-based optimizers.

13.1.3 Hash Joins

The concept of hash joins is basically implemented into Transbase® Hypercube, but there is still the lack of using hash joins for conventional queries, also in the field of star queries. The effort to integrate hash joins, however, is considered to be comparably low.

13.1.4 Complex Aggregation Expressions

In general, complex aggregate expressions are already supported by pre-grouping. These expressions require a special structure, i.e., only products of factors where each fact contains either fact table or dimension table attributes are required. Additional effort is necessary to handle general complex expressions.

13.1.5 Multi Query Box Algorithm

We have implemented an algorithm to improve handling of many query boxes. However, there is still one optimization that should be implemented: We have to order the query boxes w.r.t. z-order and modify the processing of the query boxes correspondingly as described in Section 10.3.2.4.

13.1.6 Multiple Fact Tables and multiple Hierarchies

There is still research necessary for complex schemata, e.g., for schemata with multiple fact tables and multiple hierarchies. Some of the concepts are already implemented, but a cost model and corresponding optimizer rules are still missing.

13.1.7 Assistants and Wizards around Transbase® Hypercube

In order to provide a complete DW suite consisting of a DBMS, schema design tools and further assistants like tuning wizards etc., much work is necessary. Self-tuning wizards are a hot topic in current DBMS development. The motivation and priorities mainly depend on the requirements of the customers and how they understand the technology and can integrate and implement the concepts described in this thesis.

14 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.

[APB98] www.olapcouncil.org

[Bay97] R. Bayer. *The universal B-Tree for multi-dimensional Indexing: General Concepts.* WWCA '97. Tsukuba, Japan, LNCS, Springer Verlag, March, 1997.

[Bla00] M. Blaschka: *FIESTA: A Framework for Schema Evolution in Multidimensional Databases*, Ph.D. Thesis, Technische Universität München, 2000

[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.

[CD97b] S. Chaudhuri, U. Dayal: *Data Warehousing and OLAP for Decision Support* (Tutorial). SIGMOD Conference 1997: 507-508

[CHY93] M.-S. Chen, H.-I. Hsiao, P. S. Yu. *Applying Hash Filters to Improving the Execution of Bushy Trees*, Proceedings of the 19th International Conference on Very Large Data Bases, p. 505-516, 1993

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

[CLYY92] M.-S. Chen, M.-L. Lo, P. S. Yu, H. C. Young. *Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins*. Proceedings of the 18th International Conference on Ver Large Data Bases, p. 16-26, 1992

[Com79] D. Comer: *The ubiquitous B-tree*. ACM Computing Surverys, 11(2), 1979.

[CS94] S. Chaudhuri, K. Shim: *Including Group-By in Query Optimization*. VLDB 1994: 354-366

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

[DD93] C. J. Date, H. Darwen: *A guide to the SQL Standard*, 3rd Edition, Addison-Wesley Publishing Company, 1993

[DG85] D. J. DeWitt, R. Gerber. *Multiprocessor Hash-Based Join Algorithms*. Proceedings of the 11th international Conference on Very Large Data Bases, p. 151-162, 1985

[DRSN98] P. Deshpande, K. Ramasamy, A. Shukla, J. F. Naughton: *Caching Multidimensional Queries Using Chunks*. SIGMOD Conference 1998: 259-270

[CY98] C.-Y. Chan, Y.E. Ioannidis: *Bitmap Index Design and Evaluation*, Proc. Of the ACM SIGMOD Conference, 1998

[Fen98] R. Fenk. *Design and Implementation of a UB-Tree Range Query Algorithm for a Set of Query Boxes*. Master Thesis, Technische Universität München, 1998.

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

[GB+01] H. Günzel, A. Bauer. *Data Warehouse Systeme*: dpunkt.verlag, 1. Auflage, 2001

[GBLP96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh: Data Cube: *A Relational Aggregation Operator Generalizing Group-By*, Cross-Tab, and Sub-Total. ICDE 1996: 152-159

[GG97] V. Gaede and O. Günther. *Multidimensional Access Methods*. ACM Computing Surveys 30(2), 1997.

[GHQ95] A. Gupta, V. Harinarayan, D. Quass: *Aggregate-Query Processing in Data Warehousing Environments*. VLDB Conference 1995: 358-369

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

[HCY94] H.-I Hsiao, M.-S. Chen, P.S. Yu. *On parallel Execution of multiple pipelined Hash Joins*, Proc. of the ACM SIGMOD Conference, 1994

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

[Inm96] W.H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., 2nd edition, 1996.

[JKS00] H. V. Jagadish, N. Koudas, D. Srivastave: *On Effective Multi-Dimensional Indexing for Strings*, Proc. of the ACM SIGMOD Conference, 2000

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

[KK00] C. Kirchner, H. Kirchner.: *Rewriting Solving Proving*, http://www.loria.fr/~ckirchne/, 2000

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

[KR98] Y. Kotidis, N. Roussopoulos: *An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees*. Proc. of SIGMOD Conference 1998: 249-258

[KS01] N. Karayannidis, and T. Sellis: *SISYPHUS: A Chunk-Based Storage Manager for OLAP Cubes*. Proc. of DMDW 2001

[KTS+02] N. Karayannidis, A. Tsois, T. Sellis, R. Pieringer, V. Markl, F. Ramsak, R. Fenk, K. Elhardt, R. Bayer: *Processing Star Queries on Hierarchically-Clustered Fact Tables*. Proc. of VLDB 2002

[Lar97] P-Å. Larson: *Grouping and Duplicate Elimination: Benefits of Early Aggregation*. Technical Report MSR-TR-97-36, 1997 (available at http://www.research.microsoft.com/pubs)

[Lar02] P-Å. Larson: *Data Reduction by Partial Preaggregation*. Proc. of ICDE Conference 2002: 706-715

[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.

[LMS94] A. Y. Levy, I. S. Mumick, Y. Sagiv: *Query Optimization by Predicate Move-Around*. Proc. of VLDB Conference 1994: 96-107

[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.

[Mer99] S. Merkel. *Evaluation of the UB-Tree for a Market Research Data Warehouse*. Master Thesis TU-München, 1999

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

[NG95] P. E. O'Neil, G. Graefe: *Multi-Table Joins Through Bitmapped Join Indices*. SIGMOD Record 24(3): 8-11 (1995)

[NQ97] P. E. O'Neil, D. Quass: *Improved Query Performance with Variant Indexes*. SIGMOD Conference 1997: 38-49

[OM84] J. A. Orenstein and T.H. Merret. *A Class of Data Structures for Associate Searching*. Proc. of ACM SIGMOD-PODS Conf., Portland, Oregon, 1984, pp. 294-305

[Ora01] Oracle Documentation ; 2001

[PM98] A. N. Papadopoulos, Y. Manolopoulos. *Multiple Range Query Optimization in Spatial Databases*. ADBIS 1998

[PER+03a] R. Pieringer, K. Elhardt, F. Ramsak, V. Markl, R. Fenk, R. Bayer, N. Karayannidis, A. Tsois, T. Sellis. *Combining Hierarchy Encoding and Pre-Grouping: Intelligent Grouping in Star Join Processing*. Proc. of ICDE'03, Bangalore, India, 2003

[PER+03b] R. Pieringer, K. Elhardt, F. Ramsak, V. Markl, R. Fenk, R. Bayer . *Transbase: A leading-edge ROLAP Engine supporting multidimensional Indexing and Hierarchy Clustering*. Proc. of BTW'03, Leipzig, Germany, 2003

[PMB00] R. Pieringer, V. Markl, R. Bayer. *Modellierung und Verwaltung hierarchisch strukturierter Informationen in relationalen Datenbanksystemen*. Proc. of GI-Workshop Plön, Germany, 2000

[Ram02] F. Ramsak. *Towards a general-purpose, multidimensional index: Integration,Optimization, and Enhancement of UB-Trees*. Ph.D. Thesis, Technische Universität München, 2002.

[RMF+00] F. Ramsak, V. Markl, R. Fenk, M. Zirkel,K. Elhardt, R. Bayer.*Integrating the UB-Tree intoa Database System Kernel*. Proc. of VLDB2000, Cairo, Egypt, 2000.

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

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990

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

[SBH+98] C. Sapia, M.Blaschka, G. Höfling, B. Dinter: *Extending the E/R model for the Multidimensional Paradigma*. Advances in Database Technologies, Springer LNCS 1998.

[SN95] A. Shatdal, J. F. Naughton: *Adaptive Parallel Aggregation Algorithms*. Proc. of SIGMOD 1995, 297-306

[Tra01] Transbase® Documentation; www.transaction.de, 2001

[TT01] D. Theodoratos, A. Tsois: *Heuristic Optimization of OLAP Queries in Multidimensionally Hierarchically Clustered Databases*. DOLAP 2001.

[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

[WB98] M.-C. Wu, A. P. Buchmann: *Encoded Bitmap Indexing for Data Warehouses*. Proc. Of the ICDE 1998

[YL94] W. P. Yan, P-Å. Larson: *Performing Group-By before Join*. Proc. of ICDE 1994: 89-100

[YL95] W. P. Yan, P.-Å. Larson: *Eager Aggregation and Lazy Aggregation*. Proc. of VLDB Conference 1995

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

Appendices

Appendix A: Conceptual Schema of Sales DW

Dimension Calendar

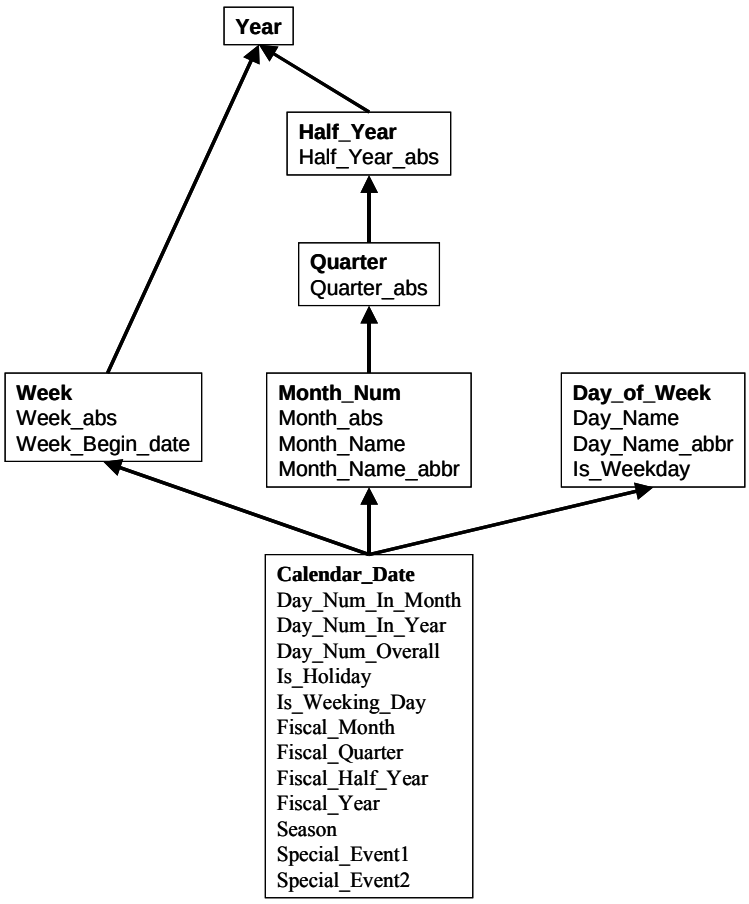


Figure 14-1: Calendar Hierarchy

Dimension Hour

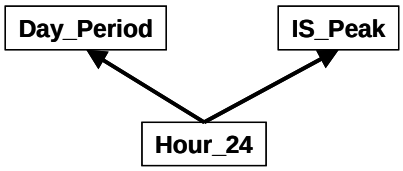


Figure 14-2: Hour Hierarchy

Dimension Product

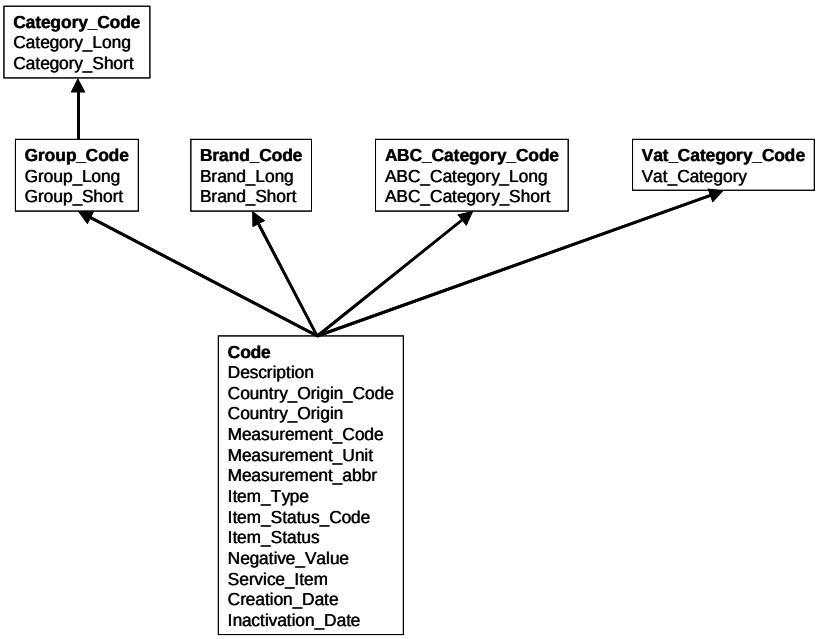


Figure 14-3: Product Hierarchy

Dimension Warehouse

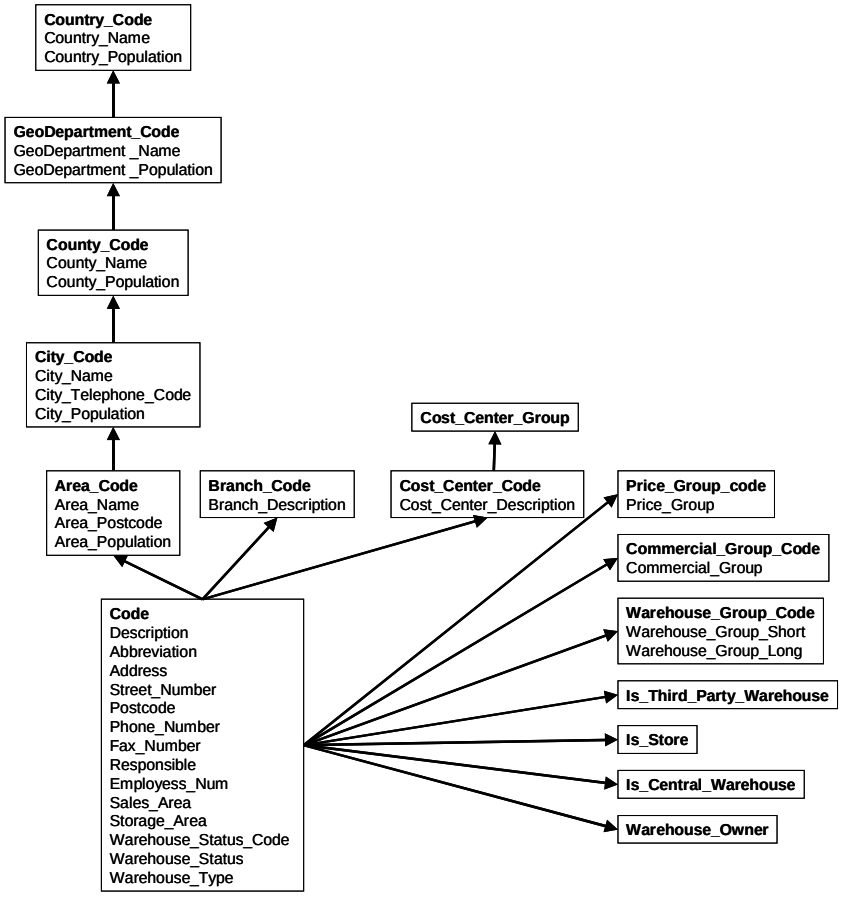


Figure 14-4: Warehouse Hierarhcy

Dimension Customer

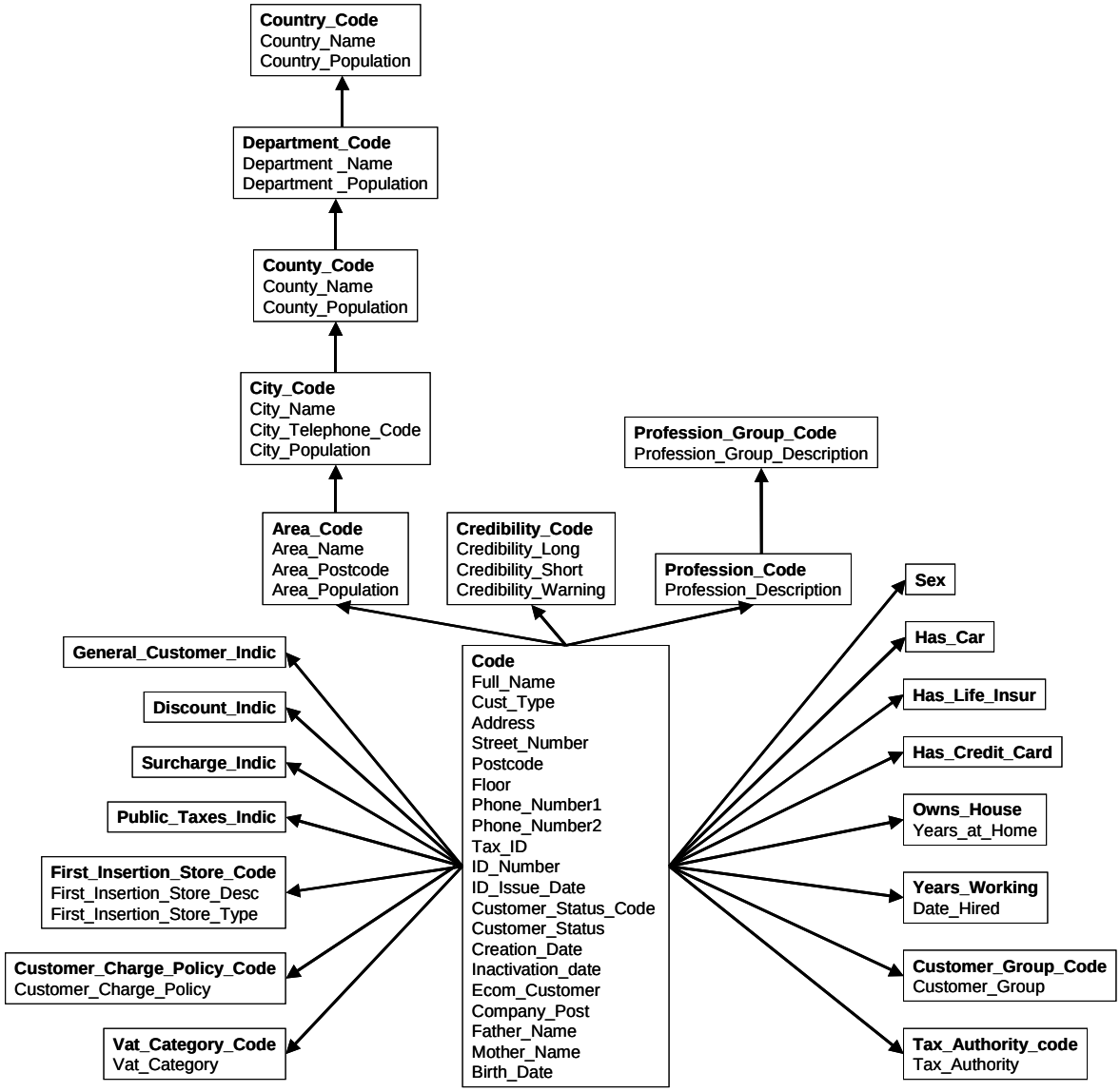


Figure 14-5: Customer Hierarchy

Dimension Sales Transaction

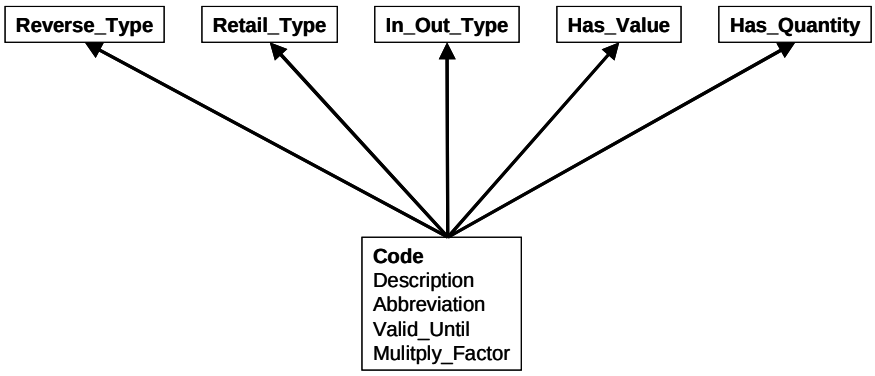


Figure 14-6: Sales Transaction Hierarchy

Dimension Offering

Figure 14-7: Offering Dimension

Dimension Salesman

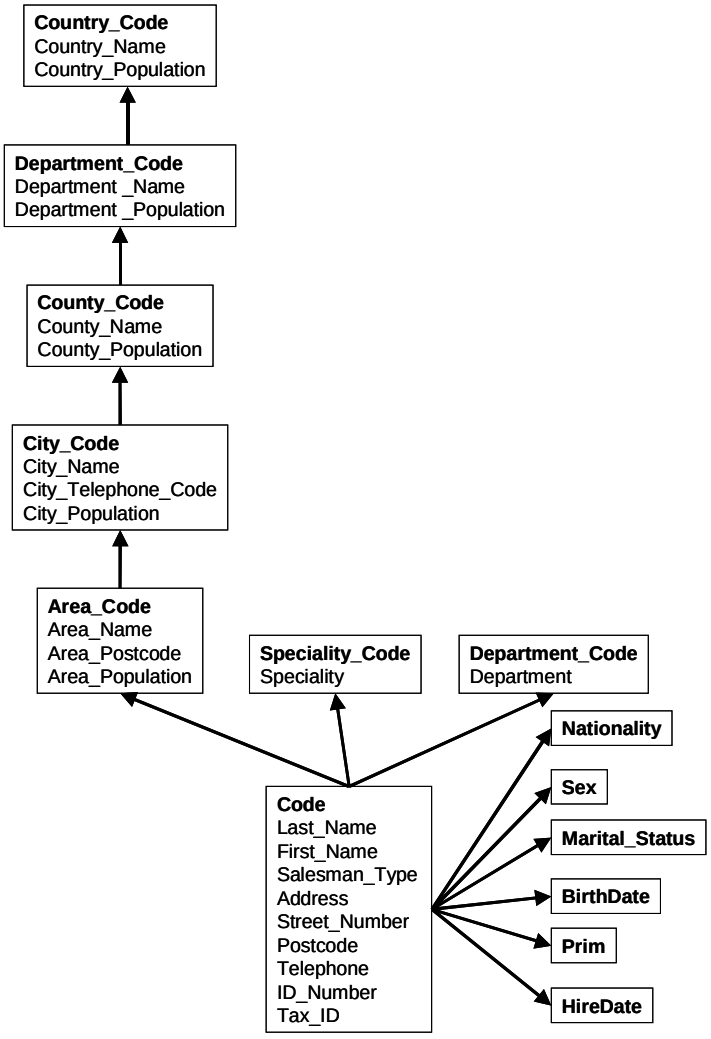


Figure 14-8: Salesman Hierarchy

Dimension Cash Register

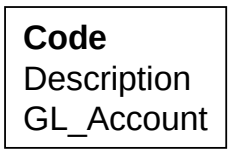


Figure 14-9: Cash Register Dimension

Dimension Currency

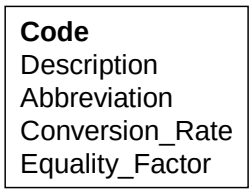


Figure 14-10: Currency Dimension

Dimension Sales Payment

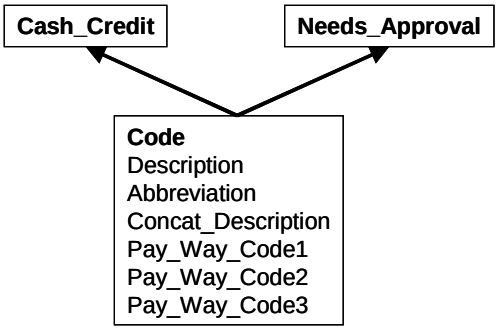


Figure 14-11: Sales Payment Hierarchy

Dimension Loan Status

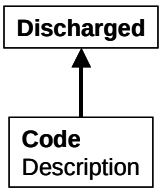


Figure 14-12: Loan Status Hierarchy

Dimension Special IDL

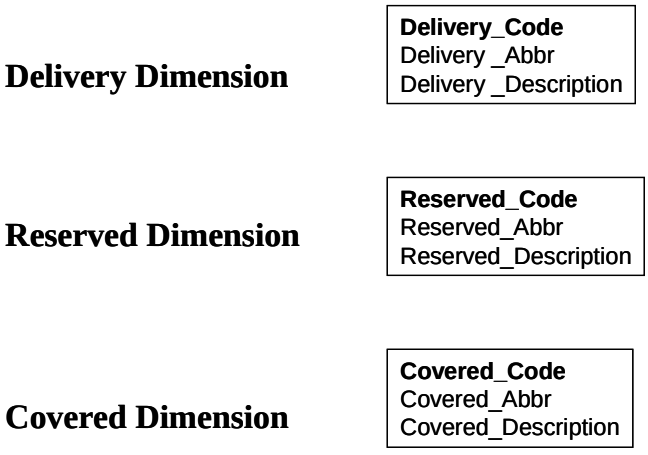


Figure 14-13: Three low-Cardinality Dimensions

Appendix B: Data Distribution of Sales DW

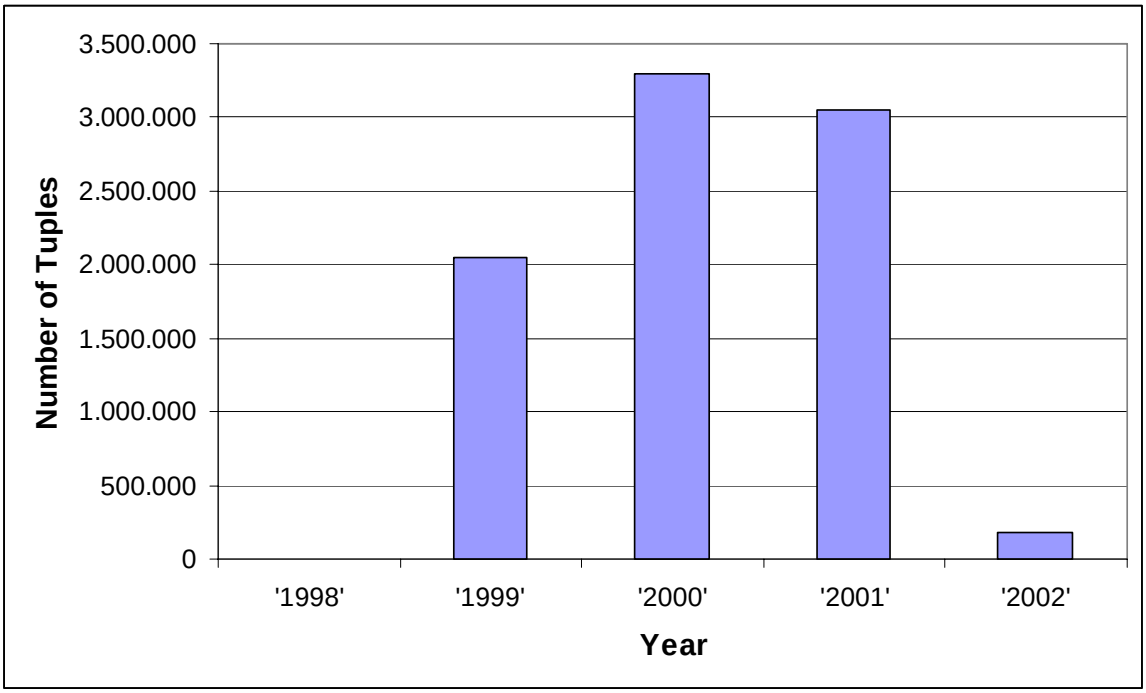


Figure 14-14: Data Distribution according to Calendar: Year

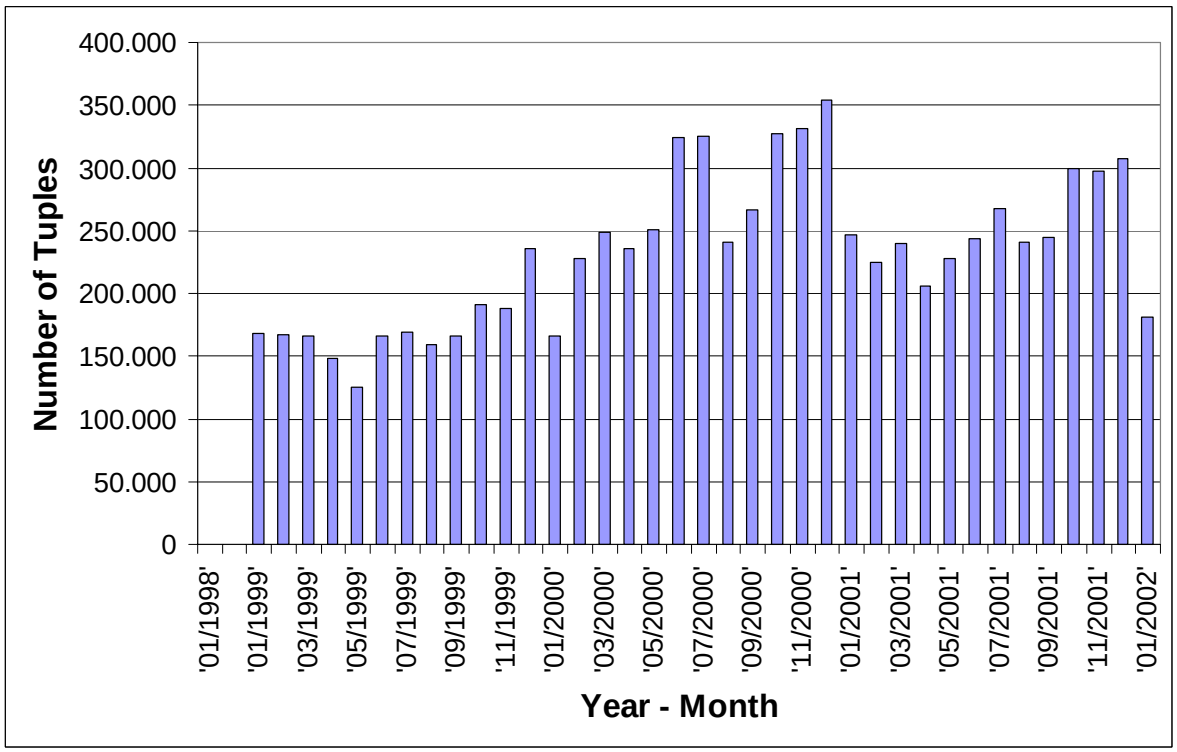


Figure 14-15: Data Distribution according to Calendar: Year – Month

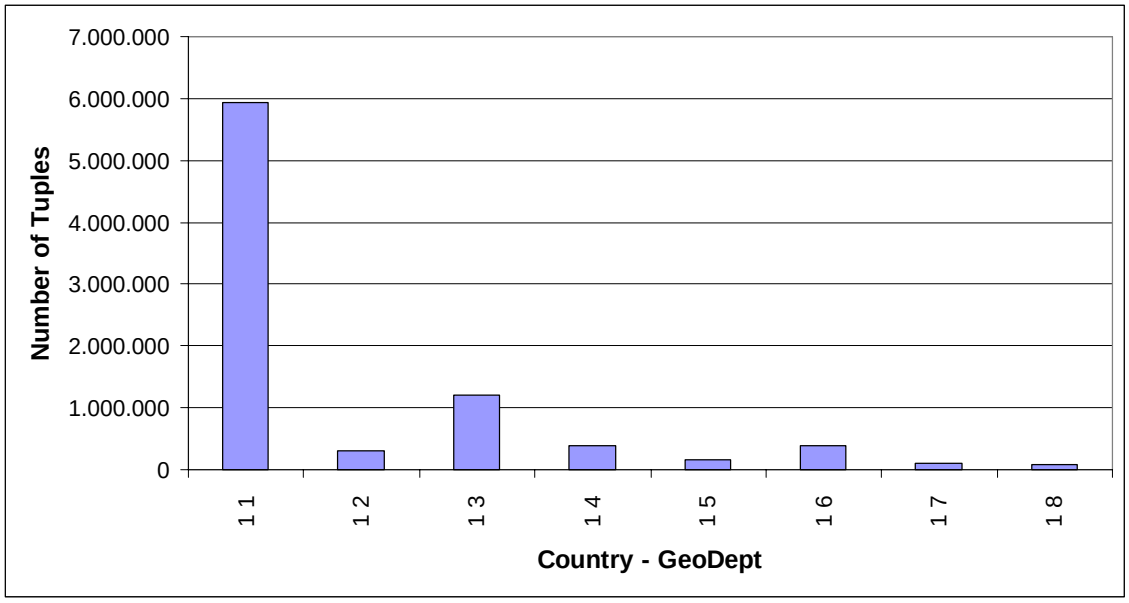


Figure 14-16: Data Distribution according to Warehouse: Country – GeoDepartment

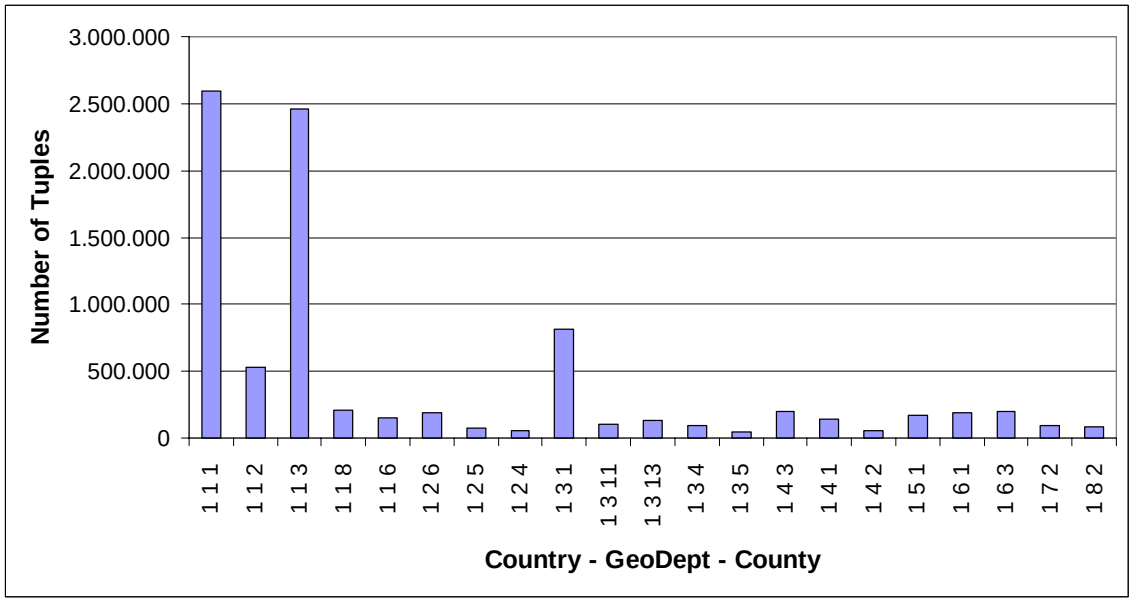


Figure 14-17: Data Distribution according to Warehouse: Country – GeoDepartment - County

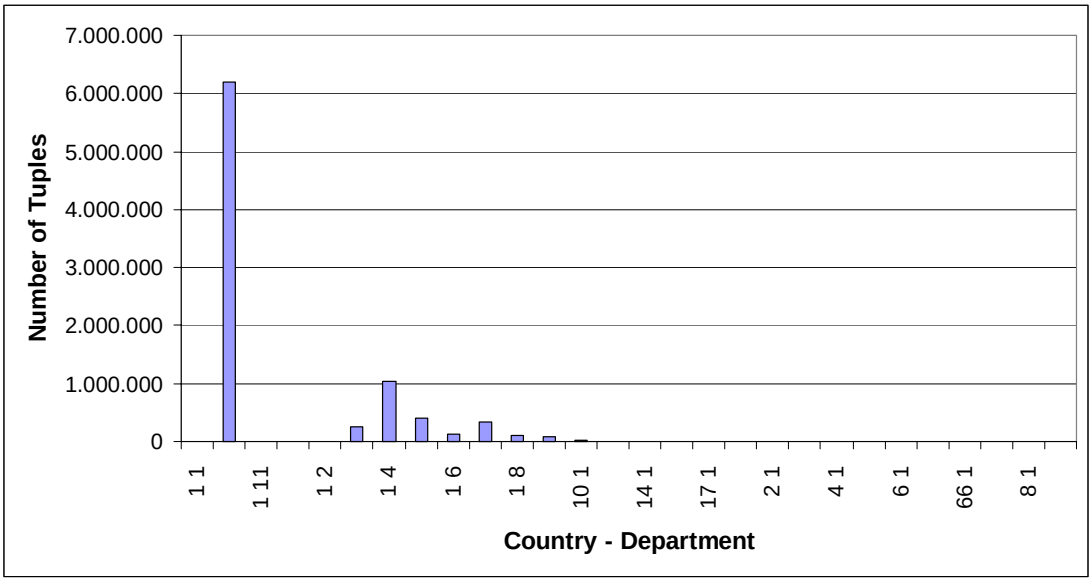


Figure 14-18: Data Distribution according to Customer: Country - Department

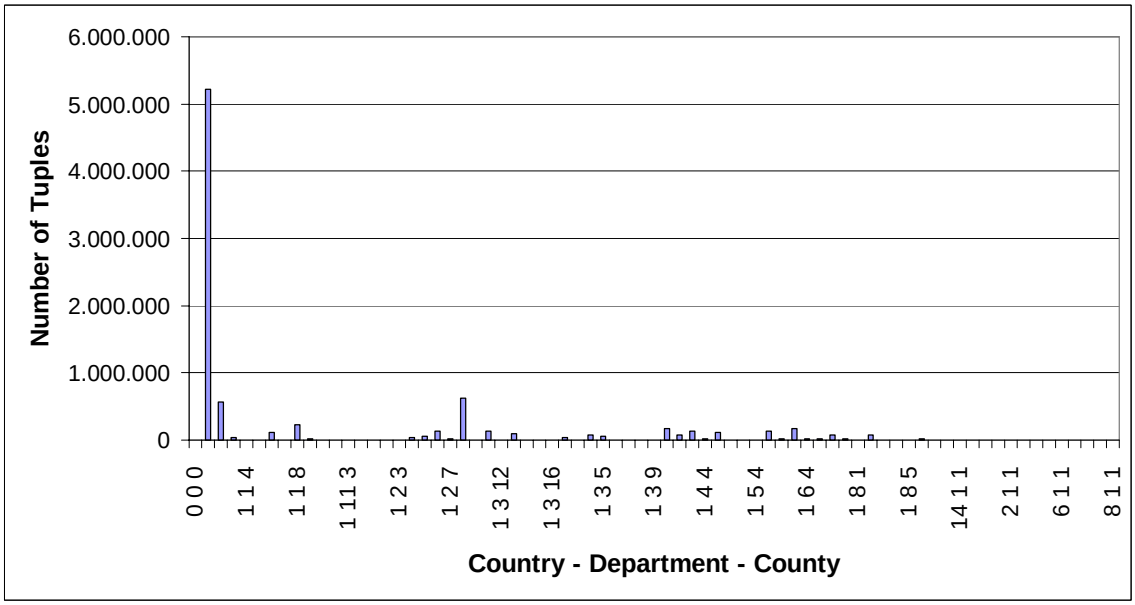


Figure 14-19: Data Distribution according to Customer: Country – Department - County

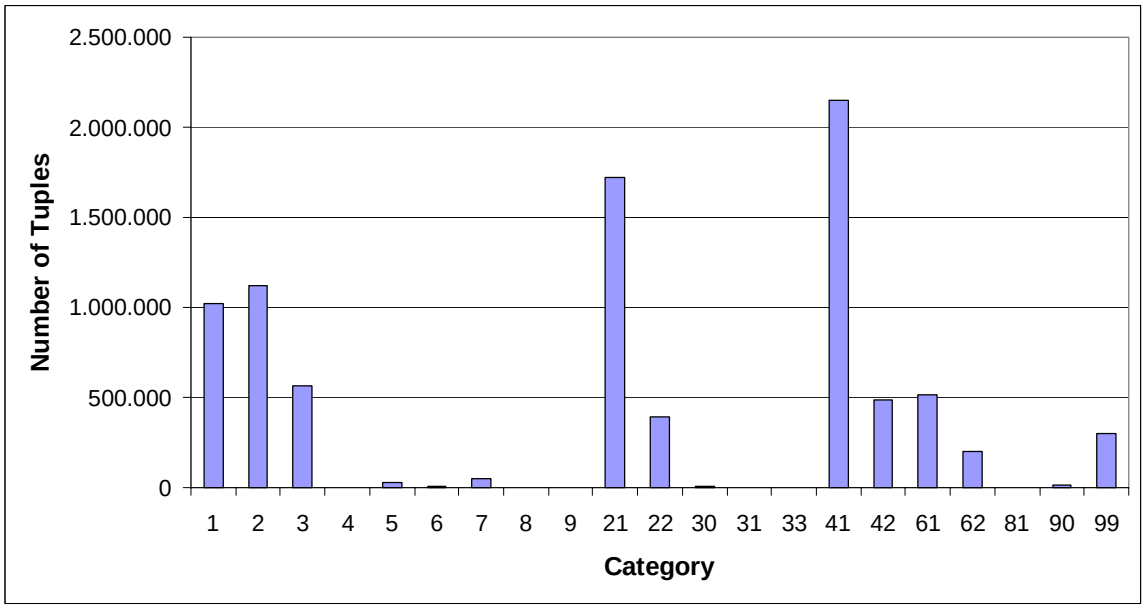


Figure 14-20: Data Distribution according to Product: Category

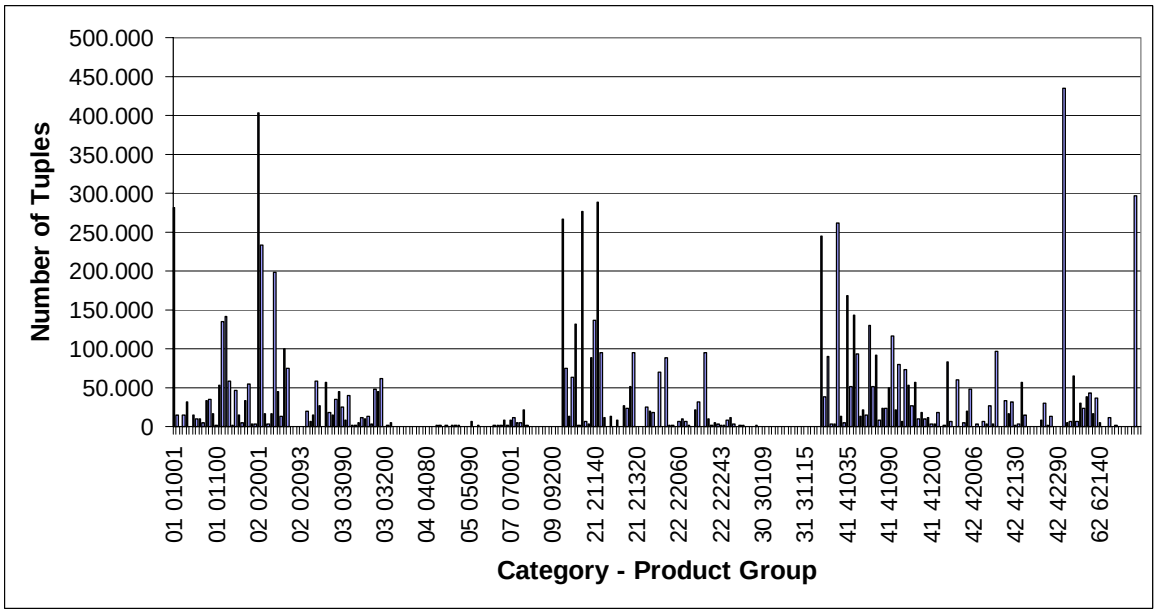


Figure 14-21: Data Distribution according to Product: Category – Product Group

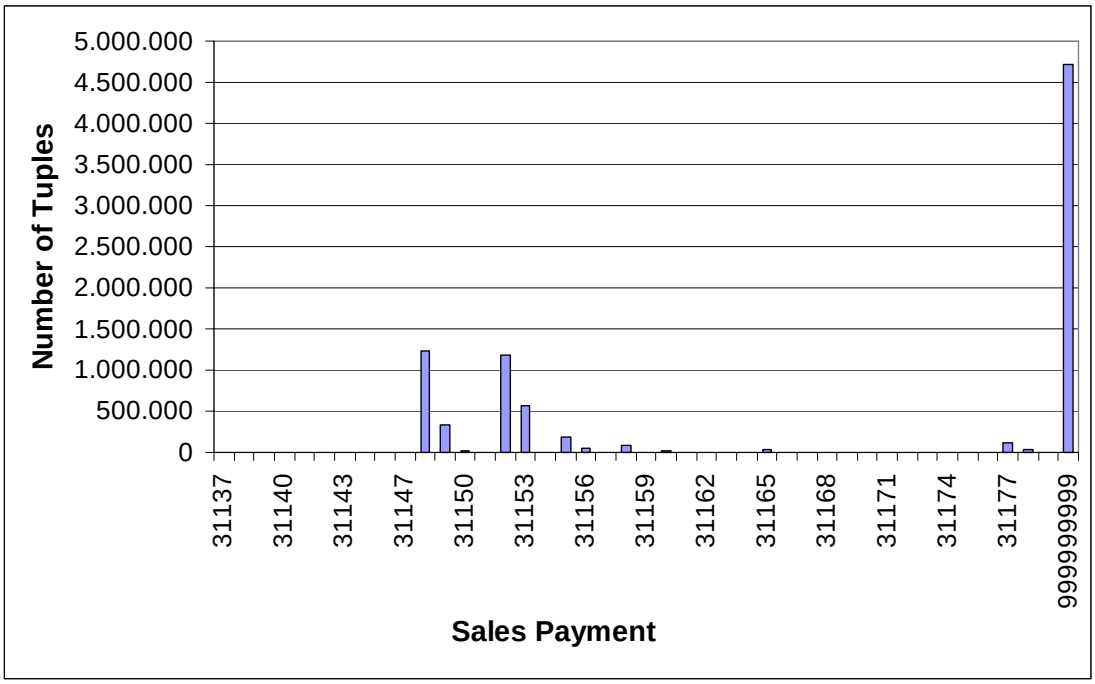


Figure 14-22: Data Distribution according to Sales Payment

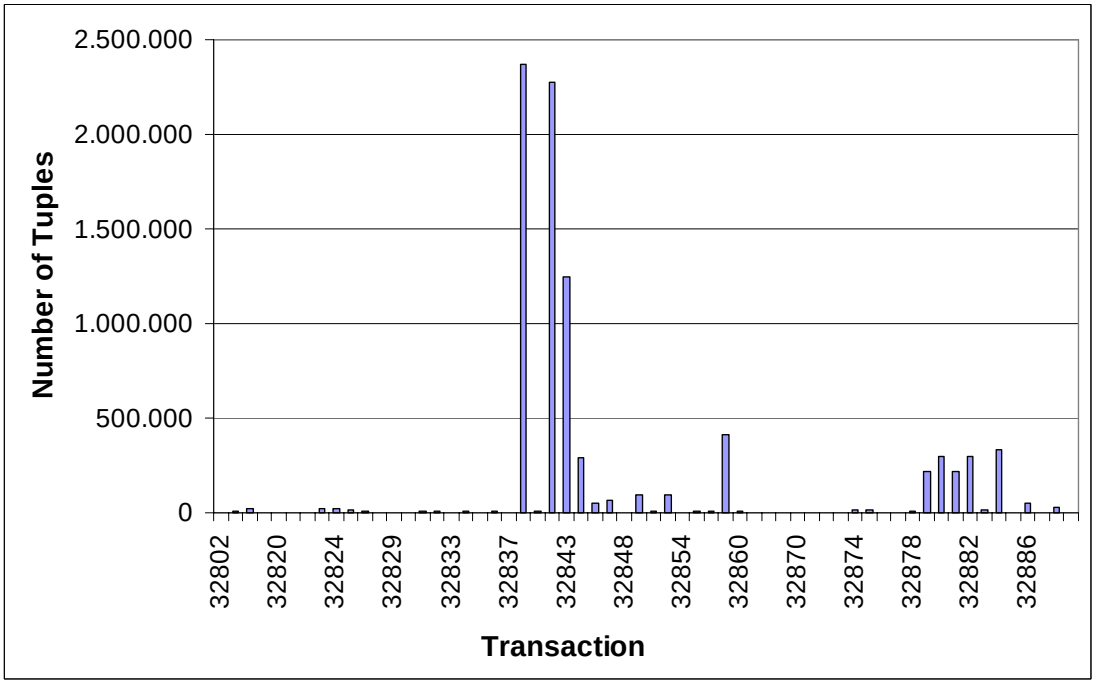


Figure 14-23: Data Distribution according to Transaction

Appendix C: Operator Trees

```
(N0:rel  { "fact" }
   (N1:times  {   keyaccess }
      (N2:times  {   keyaccess }
         (N3:ivmk  {  betw   }
            (N4:sort  {  +1 }
               (N5:proj
                  (N6:times  {   dimleaf }
                     (N7:times  {   dimhigh }
                        (N8:restr
                           (N9:rel  { "customer_country"  })
                           (N10:eq  { }
                              (N11:attr  { N8[3] } )
                              (N12:const  { 'GERMANY'  char(3) } )))
                        (N13:restr
                           (N14:rel  { "customer_dept"  })
                           (N15:eq  { }
                              (N16:attr  { N13[3] } )
                              (N17:const  { 'SOUTH'  char(4) } ))))
                     (N18:compsurr  { 8 5 }
                        (N19:restr
                           (N20:rel  { "customer"  })
                           (N21:and
                              (N22:eq  { }
                                 (N23:attr  { N19[1] } )
                                 (N24:attr  { N6[1] } ))
                              (N25:eq  { }
                                 (N26:attr  { N19[2] } )
                                 (N27:attr  { N6[4] } ))))))
                  (N28:build
                     (N29:attr  { N5[7] } )
                     (N30:attr  { N5[8] } )))))
         (N31:ivmk  {  betw   }
            (N32:sort  {  +1 }
               (N33:proj
                  (N34:times  {   dimleaf }
                     (N35:restr
                        (N36:rel  { "product_cat"  })
                        (N37:eq  { }
                           (N38:attr  { N35[2] } )
                           (N39:const  { 'TV'  char(2) } )))
                     (N40:compsurr  { 5 3 }
                        (N41:restr
                           (N42:rel  { "product"  })
                           (N43:eq  { }
                              (N44:attr  { N41[1] } )
                              (N45:attr  { N34[1] } )))))
                  (N46:build
                     (N47:attr  { N33[3] } )
                     (N48:attr  { N33[4] } ))))))
      (N49:ivmk  {  betw   }
         (N50:sort  {  +1 }
            (N51:compsurr  { 9 2 }
               (N52:restr
                  (N53:rel  { "date"  })
                  (N54:and
                     (N55:and
                        (N56:eq  { }
                           (N57:attr  { N52[6] } )
```

```
                        (N58:const  { '10/2002'  char(7) } ))
                     (N59:eq  { }
                        (N60:attr  { N52[5] } )
                        (N61:const  { '4q2002'  char(6) } )))
                  (N62:eq  { }
                     (N63:attr  { N52[1] } )
                     (N64:const  { '2002'  char(4) } )))))))))
```

Figure 14-24: Operator Tree of the combined Dimension Operator Trees

```
(N0:sel  {   9 "country_str" "dept_str" "cat_str" "grp_str" "year" "quarter
" "month " "column_7" "column_8"  }
   (N1:proj
      (N2:proj
         (N3:times  {   groupvaluejoin }
            (N4:group  { ghash [ 20 23 30 32 3 ] sum[5] sum[6] repres[9 5]}
               (N5:proj
                  (N6:proj
                     (N7:times  {   groupexactjoin }
                        (N8:times  {   groupexactjoin }
                           (N9:times  {   groupexactjoin }
                              (N10:times  {   groupexactjoin }
                                 (N11:times  {   groupexactjoin }
                                    (N12:times  {   groupexactjoin }
                                       (N13:group  {  ghash [ 1 2 3 ]
count[*] sum[4] sum[5] repres[6 1] repres[7 2] repres[8 -3] }
                                          (N14:proj
                                             (N15:rel  { "fact" }
```

< Operator Tree for rel (“fact”) as in Figure 14-24>

```
                                          (N72:build
                                             (N73:subrg  { false }
                                                (N74:attr  { N14[8] } )
                                                (N75:const { 1 integer})
                                                (N76:const {8 integer}))
                                             (N77:subrg  { false }
                                                (N78:attr  { N14[7] } )
                                                (N79:const { 1 integer})
                                                (N80:const{11 integer}))
                                             (N81:subrg  { false }
                                                (N82:attr  { N14[9] } )
                                                (N83:const  {1 integer})
                                                (N84:const {7 integer}))
                                             (N85:attr  { N14[4] } )
                                             (N86:attr  { N14[5] } )
                                             (N87:attr  { N14[1] } )
                                             (N88:attr  { N14[2] } )
                                             (N89:attr  { N14[3] } ))))
                                       (N90:restr
                                          (N91:rel  { "customer"  })
                                          (N92:eq  { nljoin }
                                             (N93:attr  { N12[7] } )
                                             (N94:attr  { N90[6] } ))))
                                    (N95:restr
                                       (N96:rel  { "product"  })
                                       (N97:eq  { nljoin }
                                          (N98:attr  { N11[8] } )
                                          (N99:attr  { N95[3] } ))))
                                 (N100:restr
                                    (N101:rel  { "customer_country"  })
```

```
                                    (N102:and
                                      (N103:eq  { }
                                        (N104:attr  { N10[10] } )
                                        (N105:attr  { N100[1] } ))
                                      (N106:eq  { }
                                        (N107:attr  { N100[3] } )
                                        (N108:const{'GERMANY' char(3)}))))))
                                 (N109:restr
                                    (N110:rel  { "customer_dept"  })
                                    (N111:and
                                       (N112:eq  { }
                                          (N113:attr  { N9[11] } )
                                          (N114:attr  { N109[1] } ))
                                       (N115:eq  { }
                                          (N116:attr  { N109[3] } )
                                          (N117:const  { 'SOUTH' char(4)}))))))
                              (N118:restr
                                 (N119:rel  { "product_cat"  })
                                 (N120:and
                                    (N121:eq  { }
                                       (N122:attr  { N8[18] } )
                                       (N123:attr  { N118[1] } ))
                                    (N124:eq  { }
                                       (N125:attr  { N118[2] } )
                                       (N126:const  { 'TV'  char(2) } )))))
                           (N127:restr
                              (N128:rel  { "product_grp"  })
                              (N129:eq  { }
                                 (N130:attr  { N7[19] } )
                                 (N131:attr  { N127[1] } ))))
                        (N132:build
                           (N133:attr  { N6[1] } )
                           (N134:attr  { N6[2] } )
…
                           (N164:attr  { N6[32] } )))
                     (N165:build
                        (N166:attr  { N5[1] } )
                        (N167:attr  { N5[2] } )
…
                        (N197:attr  { N5[32] } ))))
               (N198:restr
                  (N199:rel  { "time"  })
                  (N200:eq  { nljoin }
                     (N201:attr  { N3[8] } )
                     (N202:attr  { N198[4] } ))))
            (N203:build
               (N204:attr  { N2[1] } )
               (N205:attr  { N2[2] } )
…
               (N212:attr  { N2[7] } )))
         (N213:build
            (N214:attr  { N1[1] } )
            (N215:attr  { N1[2] }
…
            (N222:attr  { N1[8] } ))))
```

Figure 14-25: Operator Tree with Grouping and Residual Join Optimization

Appendix D: Business Query Templates for Measurements

Template Q01

```
select
      hrs.hour_24 as hour
       ,cal.month_year as month_year
       ,zap.description as warehouse_description
       ,zap.warehouse_group_short as warehouse_group_short_desc
       ,zap.area_name as warehouse_area_name
       ,grp.description as grp_description
       ,sum(trxgrp.multiply_factor * fact1.val_interest) as sum_fact_val_interest
       ,sum(trxgrp.multiply_factor * fact.val_gross) as sum_fact_val_gross

from
        hour            hrs
       ,calendar        cal
       ,warehouse       zap
       ,fact            fact
       ,sl_trans_groups trxgrp
       ,sl_groups_ref   grp
       ,transaction     trx
       ,fact            fact1
where cal.dwh_key             = fact.cal_trans_dwh_key
and   hrs.dwh_key               = fact.hrs_dwh_key
and   zap.dwh_key               = fact.whs_issue_dwh_key
and   trx.dwh_key               = fact.trx_dwh_key
and   trxgrp.trx_code             = trx.code
and   trxgrp.grp_code             = grp.code
and   fact.root_trans_seq         = fact1.trans_seq
and   fact.root_line_number       = fact1.line_number
and   grp.description             = 'ВО ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
and   zap.is_store                = 'ÍÁÉ'
and   zap.warehouse_owner         = 'ÊÙÔÓÏÂÏËÏÓ'
and   hrs.peak                    = 'ÍÁÉ'
and   cal.calendar_date           >= ${BEGIN_DATE}
and   cal.calendar_date            <= ${END_DATE}
group by
      hrs.hour_24
       ,cal.month_year
       ,zap.description
       ,zap.warehouse_group_short
       ,zap.area_name
       ,grp.description
```

Template Q02

```
select
     prd.description as product_description
    ,prd.brand_short as product_brand_short
    ,prd.prodgroup_short as product_group_short
    ,prd.category_short as product_category_short
    ,sum(trxgrp.multiply_factor*fact.val_total) as sum_fact_val_total
    ,sum(trxgrp.multiply_factor*fact.qty_total) as sum_fact_total_quantity
    ,sum(fact.val_cost) as sum_fact_val_cost
from
    product         prd
   ,calendar        cal
   ,fact            fact
   ,sl_trans_groups trxgrp
   ,sl_groups_ref   grp
   ,transaction     trx
where prd.dwh_key        = fact.prd_dwh_key
  and cal.dwh_key        = fact.cal_trans_dwh_key
  and trx.dwh_key        = fact.trx_dwh_key
  and trxgrp.trx_code    = trx.code
```

```
  and trxgrp.grp_code    = grp.code
  and grp.description    = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
  and cal.calendar_date  >= ${BEGIN_DATE}
  and cal.calendar_date  <= ${END_DATE}
group by
  prd.description
 ,prd.brand_short
 ,prd.prodgroup_short
 ,prd.category_short
```

Template Q03

```
select
     cal.calendar_date as calendar_date
    ,sum(trxgrp.multiply_factor*fact.val_total) as sum_fact_val_total
from
    product         prd
   ,calendar        cal
   ,fact            fact
   ,sl_trans_groups trxgrp
   ,sl_groups_ref   grp
   ,transaction     trx
where prd.dwh_key        = fact.prd_dwh_key
 and  cal.dwh_key        = fact.cal_trans_dwh_key
 and  trx.dwh_key        = fact.trx_dwh_key
 and  trxgrp.trx_code    = trx.code
 and  trxgrp.grp_code    = grp.code
 and  prd.prodgroup_code    <= '42160'
 and  prd.prodgroup_code    >= '01001'
 and  grp.description    = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
and   cal.calendar_date      >= ${BEGIN_DATE}
and   cal.calendar_date      <= ${END_DATE}
group  by
   cal.calendar_date
```

Template Q04

```
select
      prd.description as product_description
     ,prd.brand_short as product_short
     ,prd.omada_short as product_omada_short
     ,prd.kathgoria_short as product_kathgoria_short
     ,cal.month_year as month_year
     ,sum(trxgrp.multiply_factor*fact.val_gross) as sum_fact_val_gross
     ,sum(trxgrp.multiply_factor*fact.qty_total) as sum_fact_qty_total
     ,sum(trxgrp.multiply_factor*fact.val_cost)  as sum_fact_val_cost
from
    product               prd
   ,calendar              cal
   ,fact                  fact
   ,sl_trans_groups       trxgrp
   ,sl_groups_ref         grp
   ,transaction           trx
where prd.dwh_key        = fact.prd_dwh_key
  and cal.dwh_key        = fact.cal_trans_dwh_key
  and trx.dwh_key        = fact.trx_dwh_key
  and trxgrp.trx_code    = trx.code
  and trxgrp.grp_code    = grp.code
  and grp.description    = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
  and cal.calendar_date  >= ${BEGIN_DATE}
  and cal.calendar_date  <= ${END_DATE}
group by
     prd.description
    ,prd.brand_short
    ,prd.omada_short
    ,prd.kathgoria_short
    ,cal.month_year
```

Template Q05

```
select
     hrs.hour_24 as hour
    ,zap.description as warehouse_description
    ,trx.description as transaction_description
    ,sum(trxgrp.multiply_factor*fact.val_gross) as sum_fact_val_gross
from
    hour                hrs
   ,calendar            cal
   ,sl_trans_groups     trxgrp
   ,sl_groups_ref       grp
   ,transaction         trx
   ,warehouse           zap
   ,fact                fact
where hrs.dwh_key         = fact.hrs_dwh_key
  and cal.dwh_key         = fact.cal_trans_dwh_key
  and trx.dwh_key         = fact.trx_dwh_key
  and trxgrp.trx_code     = trx.code
  and trxgrp.grp_code     = grp.code
  and zap.dwh_key         = fact.whs_issue_dwh_key
  and grp.description     = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
  and cal.calendar_date   >= ${BEGIN_DATE}
  and cal.calendar_date   <= ${END_DATE}
  and hrs.day_period      = 'ÌÅÓÇÌÅÑÉ'
  and zap.is_store        = 'ÍÁÉ'
  and zap.warehouse_owner = 'ÊÙÔÓÏÂÏËÏÓ'
group by
  hrs.hour_24
 ,zap.description
 ,trx.description
```

Template Q06

```
select
     zap.code as warehouse_code
    ,prd.code as product_code
    ,prd.prodgroup_short as product_group_short
    ,prd.category_short as product_category_short
    ,sum(trxgrp.multiply_factor*fact.val_total) as sum_fact_val_total
    ,sum(trxgrp.multiply_factor*fact.val_vat) as sum_fact_val_vat
    ,sum(trxgrp.multiply_factor*fact.val_cost) as sum_fact_val_cost
    ,sum(trxgrp.multiply_factor*fact.val_interest) as sum_fact_val_interest
from
    product         prd
   ,calendar        cal
   ,fact            fact
   ,sl_trans_groups trxgrp
   ,sl_groups_ref   grp
   ,transaction     trx
   ,warehouse       zap
where prd.dwh_key         = fact.prd_dwh_key
  and cal.dwh_key         = fact.cal_trans_dwh_key
  and zap.dwh_key         = fact.whs_root_issue_dwh_key
  and trx.dwh_key         = fact.trx_dwh_key
  and trxgrp.trx_code     = trx.code
  and trxgrp.grp_code     = grp.code
  and grp.description     = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
  and zap.warehouse_owner = 'ÊÙÔÓÏÂÏËÏÓ'
  and cal.calendar_date   >= ${BEGIN_DATE}
  and cal.calendar_date   <= ${END_DATE}
group by
       zap.code
      ,prd.code
      ,prd.prodgroup_short
      ,prd.category_short
```

Template Q07

```
select
      cal.calendar_date     as calendar_date
     ,cre.code              as cash_register_code
     ,fact.doc_num          as fact_document_num
     ,fact.qty_total        as fact_quantity_total
     ,zap.code              as warehouse_code
     ,cus.code              as customer_code
     ,prd.code              as product_code
     ,sum(fact.val_taxable) as sum_val_taxable
from
      product       prd
     ,calendar      cal
     ,warehouse     zap
     ,cash_register cre
     ,customer      cus
     ,fact          fact
     ,transaction   trx
where  prd.dwh_key = fact.prd_dwh_key
and    cal.dwh_key = fact.cal_trans_dwh_key
and    cus.dwh_key = fact.cus_dwh_key
and    zap.dwh_key = fact.whs_issue_dwh_key
and    cre.dwh_key = fact.cre_dwh_key
and    trx.dwh_key = fact.trx_dwh_key
and    trx.abbreviation in ( 'ÁÐ9' )
and    cal.calendar_date      >= ${BEGIN_DATE}
and    cal.calendar_date      <= ${END_DATE}
group by
      cal.calendar_date
     ,cre.code
     ,fact.doc_num
     ,fact.qty_total
     ,zap.code
     ,cus.code
     ,prd.code
```

Template Q08

```
select
       slm.code            as salesman_code
      ,slm.last_name       as salesman_last_name
      ,slm.first_name      as salesman_first_name
      ,slm.salesman_type   as salesman_type
      ,cal.calendar_date   as calendar_date
      ,zap.code            as warehouse_code
      ,zap.description     as warehouse_description
      ,sdl.covered_code    as covered_code
      ,cus.code            as customer_code
      ,cus.full_name       as customer_full_name
      ,fact.doc_num        as fact_document_number
      ,trx.abbreviation    as transaction_abbreaviation
      ,sum(fact.val_total) as sum_fact_total_val
from
      salesman    slm
     ,calendar    cal
     ,warehouse   zap
     ,special_idl sdl
     ,customer    cus
     ,fact        fact
     ,transaction trx
where cal.dwh_key = fact.cal_trans_dwh_key
and   cus.dwh_key = fact.cus_dwh_key
and   slm.dwh_key = fact.slm_dwh_key
and   zap.dwh_key = fact.whs_issue_dwh_key
and   sdl.dwh_key = fact.sdl_dwh_key
and   trx.dwh_key = fact.trx_dwh_key
and   sdl.covered_code <> '3'
and   cal.calendar_date      >= ${BEGIN_DATE}
```

```
and   cal.calendar_date      <= ${END_DATE}
group by
       slm.code
      ,slm.last_name
      ,slm.first_name
      ,slm.salesman_type
      ,cal.calendar_date
      ,zap.code
      ,zap.description
      ,sdl.covered_code
      ,cus.code
      ,cus.full_name
      ,fact.doc_num
      ,trx.abbreviation
```

Template Q09

```
select
     zap.code                                        as warehouse_code
    ,sum(trxgrp.multiply_factor*fact.val_gross)   as sum_fact_val_gross
    ,sum(trxgrp.multiply_factor*fact.val_vat)     as sum_fact_val_vat
    ,sum(trxgrp.multiply_factor*fact.val_taxable) as sum_fact_val_taxable
from
    calendar            cal
   ,sl_trans_groups     trxgrp
   ,sl_groups_ref       grp
   ,transaction         trx
   ,warehouse           zap
   ,fact                fact
 where cal.dwh_key         = fact.cal_trans_dwh_key
  and  trx.dwh_key         = fact.trx_dwh_key
  and  trxgrp.trx_code     = trx.code
  and  trxgrp.grp_code     = grp.code
  and  zap.dwh_key         = fact.whs_issue_dwh_key
  and  grp.description     = 'ВО ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
  and cal.calendar_date    >= ${BEGIN_DATE}
  and cal.calendar_date    <= ${END_DATE}
  and zap.is_store         = 'ÍÁÉ'
  and zap.warehouse_owner  = 'ÊÙÔÓÏÂÏËÏÓ'
group by
       zap.code
```

Template Q10

```
select
      prd.prodgroup_code as product_group_code
     ,cal.calendar_date  as calendar_date
     ,zap.description    as warehouse_description
     ,fact.doc_num       as fact_document_num
     ,prd.code           as product_code
     ,trx.abbreviation   as transaction_abbreviation
from
     product     prd
    ,calendar    cal
    ,warehouse   zap
    ,fact        fact
    ,transaction trx
where prd.dwh_key = fact.prd_dwh_key
  and   cal.dwh_key = fact.cal_trans_dwh_key
  and   zap.dwh_key = fact.whs_issue_dwh_key
  and   trx.dwh_key = fact.trx_dwh_key
  and   trx.abbreviation in ('ÁÑÃ','ÐÄ3','ÐÄ4','ÐÄÐ','ÐË1','ÐË4','ÐÔ1','ÐÔ4')
  and   cal.calendar_date      >= ${BEGIN_DATE}
  and   cal.calendar_date      <= ${END_DATE}
  and   prd.prodgroup_code in ('61001','61010')
```

Template Q11

```
select
     slp.description     as payment_description
    ,sum(fact.val_total) as sum_fact_val_total
from
     fact          fact
    ,calendar      cal
    ,sales_payment slp
where  fact.slp_dwh_key       = slp.dwh_key
  and    fact.cal_trans_dwh_key = cal.dwh_key
  and    slp.cash_credit        = 'CREDIT'
  and    cal.calendar_date      >= ${BEGIN_DATE}
  and    cal.calendar_date      <= ${END_DATE}
group by
       slp.description
```

Template Q12

```
select
     slp.description     as payment_description
    ,cal.week_year       as week_year
    ,sum(fact.val_total) as sum_fact_val_total
from
     sales_payment slp
    ,calendar      cal
    ,fact          fact
where cal.dwh_key            = fact.cal_trans_dwh_key
  and slp.dwh_key            = fact.slp_dwh_key
  and slp.concat_description = 'ÐÉÓÔÙÛÉÊÇ ÊÁÑÔÁ'
  and cal.calendar_date      >= ${BEGIN_DATE}
  and cal.calendar_date      <= ${END_DATE}
group by
   slp.description
  ,cal.week_year
```

Template Q13

```
select
     prd.description     as product_description
    ,prd.brand_short     as product_brand_short
    ,prd.prodgroup_short as product_group_short
    ,prd.category_short  as product_category_short
    ,slp.description     as payment_description
    ,cal.week_year       as week_year
    ,zap.description     as warehouse_description
    ,sum(fact.qty_total) as sum_fact_qty_total
    ,sum(fact.val_gross) as sum_fact_val_gross
 from
    product       prd
   ,sales_payment slp
   ,calendar      cal
   ,fact          fact
   ,warehouse     zap
where  prd.dwh_key            = fact.prd_dwh_key
  and  cal.dwh_key            = fact.cal_trans_dwh_key
  and  slp.dwh_key            = fact.slp_dwh_key
  and  zap.dwh_key            = fact.whs_issue_dwh_key
  and  slp.concat_description = 'ÐÉÓÔÙÓÇ ÊÙÔÓÏÂÏËÏÓ'
  and cal.calendar_date       >= ${BEGIN_DATE}
  and cal.calendar_date       <= ${END_DATE}
group by
     prd.description
    ,prd.brand_short
    ,prd.prodgroup_short
    ,prd.category_short
    ,slp.description
    ,cal.week_year
    ,zap.description
```

Template Q103

```
select
     cal.calendar_date                          as calendar_date
    ,sum(trxgrp.multiply_factor*fact.val_total) as sum_fact_val_total
from
    product         prd
   ,calendar        cal
   ,fact            fact
   ,sl_trans_groups trxgrp
   ,sl_groups_ref   grp
   ,transaction     trx
where prd.dwh_key        = fact.prd_dwh_key
   and  cal.dwh_key        = fact.cal_trans_dwh_key
   and  trx.dwh_key        = fact.trx_dwh_key
   and  trxgrp.trx_code    = trx.code
   and  trxgrp.grp_code    = grp.code
   and  prd.prodgroup_code >= '90000'
   and  grp.description    = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
   and  cal.calendar_date  >= ${BEGIN_DATE}
   and  cal.calendar_date  <= ${END_DATE}
group  by
   cal.calendar_date
```

Template Q108

```
select
       slm.code            as salesman_code
      ,slm.last_name       as salesman_last_name
      ,slm.first_name      as salesman_first_name
      ,slm.salesman_type   as salesman_type
      ,cal.calendar_date   as calendar_date
      ,zap.code            as warehouse_code
      ,zap.description     as warehouse_description
      ,sdl.covered_code    as covered_code
      ,cus.code            as customer_code
      ,cus.full_name       as customer_full_name
      ,fact.doc_num        as fact_document_number
      ,trx.abbreviation    as transaction_abbreaviation
      ,sum(fact.val_total) as sum_fact_total_val
from
      salesman    slm
     ,calendar    cal
     ,warehouse   zap
     ,special_idl sdl
     ,customer    cus
     ,fact        fact
     ,transaction trx
where cal.dwh_key = fact.cal_trans_dwh_key
and   cus.dwh_key = fact.cus_dwh_key
and   slm.dwh_key = fact.slm_dwh_key
and   zap.dwh_key = fact.whs_issue_dwh_key
and   sdl.dwh_key = fact.sdl_dwh_key
and   trx.dwh_key = fact.trx_dwh_key
and   sdl.covered_code <> '3'
and   cal.calendar_date      >= ${BEGIN_DATE}
and   cal.calendar_date      <= ${END_DATE}
and   zap.county_code > '3'
group by
       slm.code
      ,slm.last_name
      ,slm.first_name
      ,slm.salesman_type
      ,cal.calendar_date
      ,zap.code
      ,zap.description
      ,sdl.covered_code
      ,cus.code
      ,cus.full_name
```

```
      ,fact.doc_num
      ,trx.abbreviation
```

Template Q110

```
select
      prd.omada_code    as product_omada_code
     ,cal.calendar_date as calendar_date
     ,zap.description   as warehouse_description
     ,fact.doc_num      as fact_document_num
     ,prd.code          as product_code
     ,trx.abbreviation  as transaction_abbreviation
from
     product     prd
    ,calendar    cal
    ,warehouse   zap
    ,fact        fact
    ,transaction trx
where prd.dwh_key = fact.prd_dwh_key
  and   cal.dwh_key = fact.cal_trans_dwh_key
  and   zap.dwh_key = fact.whs_issue_dwh_key
  and   trx.dwh_key = fact.trx_dwh_key
  and   trx.abbreviation in ('ÁÑÃ','ÐÄ3','ÐÄ4','ÐÄÐ','ÐË1','ÐË4','ÐÔ1','ÐÔ4')
  and   cal.calendar_date      >= ${BEGIN_DATE}
  and   cal.calendar_date      <= ${END_DATE}
  and   prd.prodgroup_code     >= '90000'
```

Template Q203

```
select
     cal.calendar_date                          as calendar_date
    ,sum(trxgrp.multiply_factor*fact.val_total) as sum_fact_val_total
from
    product         prd
   ,calendar        cal
   ,fact            fact
   ,sl_trans_groups trxgrp
   ,sl_groups_ref   grp
   ,transaction     trx
where prd.dwh_key        = fact.prd_dwh_key
 and  cal.dwh_key        = fact.cal_trans_dwh_key
 and  trx.dwh_key        = fact.trx_dwh_key
 and  trxgrp.trx_code    = trx.code
 and  trxgrp.grp_code    = grp.code
 and  prd.omada_code     < '90000'
 and  prd.omada_code     >= '60000'
 and  grp.description     = 'ВО ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
 and cal.calendar_date   >= ${BEGIN_DATE}
 and cal.calendar_date   <= ${END_DATE}
group  by
   cal.calendar_date
```

Template Q208

```
select
       slm.code            as salesman_code
      ,slm.last_name       as salesman_last_name
      ,slm.first_name      as salesman_first_name
      ,slm.salesman_type   as salesman_type
      ,cal.calendar_date   as calendar_date
      ,zap.code            as warehouse_code
      ,zap.description     as warehouse_description
      ,sdl.covered_code    as covered_code
      ,cus.code            as customer_code
      ,cus.full_name       as customer_full_name
      ,fact.doc_num        as fact_document_number
      ,trx.abbreviation    as transaction_abbreaviation
      ,sum(fact.val_total) as sum_fact_total_val
from
      salesman    slm
```

```
     ,calendar    cal
     ,warehouse   zap
     ,special_idl sdl
     ,customer    cus
     ,fact        fact
     ,transaction trx
where cal.dwh_key = fact.cal_trans_dwh_key
  and   cus.dwh_key = fact.cus_dwh_key
  and   slm.dwh_key = fact.slm_dwh_key
  and   zap.dwh_key = fact.whs_issue_dwh_key
  and   sdl.dwh_key = fact.sdl_dwh_key
  and   trx.dwh_key = fact.trx_dwh_key
  and   sdl.covered_code <> '3'
  and   cal.calendar_date      >= ${BEGIN_DATE}
  and   cal.calendar_date      <= ${END_DATE}
  and   zap.county_code < '3'
group by
       slm.code
      ,slm.last_name
      ,slm.first_name
      ,slm.salesman_type
      ,cal.calendar_date
      ,zap.code
      ,zap.description
      ,sdl.covered_code
      ,cus.code
      ,cus.full_name
      ,fact.doc_num
      ,trx.abbreviation
```

Template Q210

```
select
      prd.prodgroup_code as product_group_code
     ,cal.calendar_date  as calendar_date
     ,zap.description    as warehouse_description
     ,fact.doc_num       as fact_document_num
     ,prd.code           as product_code
     ,trx.abbreviation   as transaction_abbreviation
from
     product     prd
    ,calendar    cal
    ,warehouse   zap
    ,fact        fact
    ,transaction trx
where prd.dwh_key = fact.prd_dwh_key
  and   cal.dwh_key = fact.cal_trans_dwh_key
  and   zap.dwh_key = fact.whs_issue_dwh_key
  and   trx.dwh_key = fact.trx_dwh_key
  and   trx.abbreviation in ('ÁÑÃ','ÐÄ3','ÐÄ4','ÐÄÐ','ÐË1','ÐË4','ÐÔ1','ÐÔ4')
  and   cal.calendar_date      >= ${BEGIN_DATE}
  and   cal.calendar_date      <= ${END_DATE}
  and   prd.prodgroup_code <  '90000'
  and   prd.prodgroup_code >= '60000'
```

Template Q303

```
select
     cal.calendar_date                          as calendar_date
    ,sum(trxgrp.multiply_factor*fact.val_total) as sum_fact_val_total
from
    product         prd
   ,calendar        cal
   ,fact            fact
   ,sl_trans_groups trxgrp
   ,sl_groups_ref   grp
   ,transaction     trx
where prd.dwh_key        = fact.prd_dwh_key
 and  cal.dwh_key        = fact.cal_trans_dwh_key
 and  trx.dwh_key        = fact.trx_dwh_key
```

```
 and  trxgrp.trx_code    = trx.code
 and  trxgrp.grp_code    = grp.code
 and  prd.prodgroup_code < '60000'
 and  grp.description    = 'BO ÔÆÉÑÏÓ ÐÙËÇÓÅÙÍ'
 and  cal.calendar_date  >= ${BEGIN_DATE}
 and  cal.calendar_date  <= ${END_DATE}
group  by
   cal.calendar_date
```

Template Q308

```
select
       slm.code            as salesman_code
      ,slm.last_name       as salesman_last_name
      ,slm.first_name      as salesman_first_name
      ,slm.salesman_type   as salesman_type
      ,cal.calendar_date   as calendar_date
      ,zap.code            as warehouse_code
      ,zap.description     as warehouse_description
      ,sdl.covered_code    as covered_code
      ,cus.code            as customer_code
      ,cus.full_name       as customer_full_name
      ,fact.doc_num        as fact_document_number
      ,trx.abbreviation    as transaction_abbreaviation
      ,sum(fact.val_total) as sum_fact_total_val
from
      salesman    slm
     ,calendar    cal
     ,warehouse   zap
     ,special_idl sdl
     ,customer    cus
     ,fact      fact
     ,transaction trx
where cal.dwh_key = fact.cal_trans_dwh_key
  and   cus.dwh_key = fact.cus_dwh_key
  and   slm.dwh_key = fact.slm_dwh_key
  and   zap.dwh_key = fact.whs_issue_dwh_key
  and   sdl.dwh_key = fact.sdl_dwh_key
  and   trx.dwh_key = fact.trx_dwh_key
  and   sdl.covered_code <> '3'
  and   cal.calendar_date      >= ${BEGIN_DATE}
  and   cal.calendar_date      <= ${END_DATE}
  and   zap.county_code = '3'
group by
       slm.code
      ,slm.last_name
      ,slm.first_name
      ,slm.salesman_type
      ,cal.calendar_date
      ,zap.code
      ,zap.description
      ,sdl.covered_code
      ,cus.code
      ,cus.full_name
      ,fact.doc_num
      ,trx.abbreviation
```

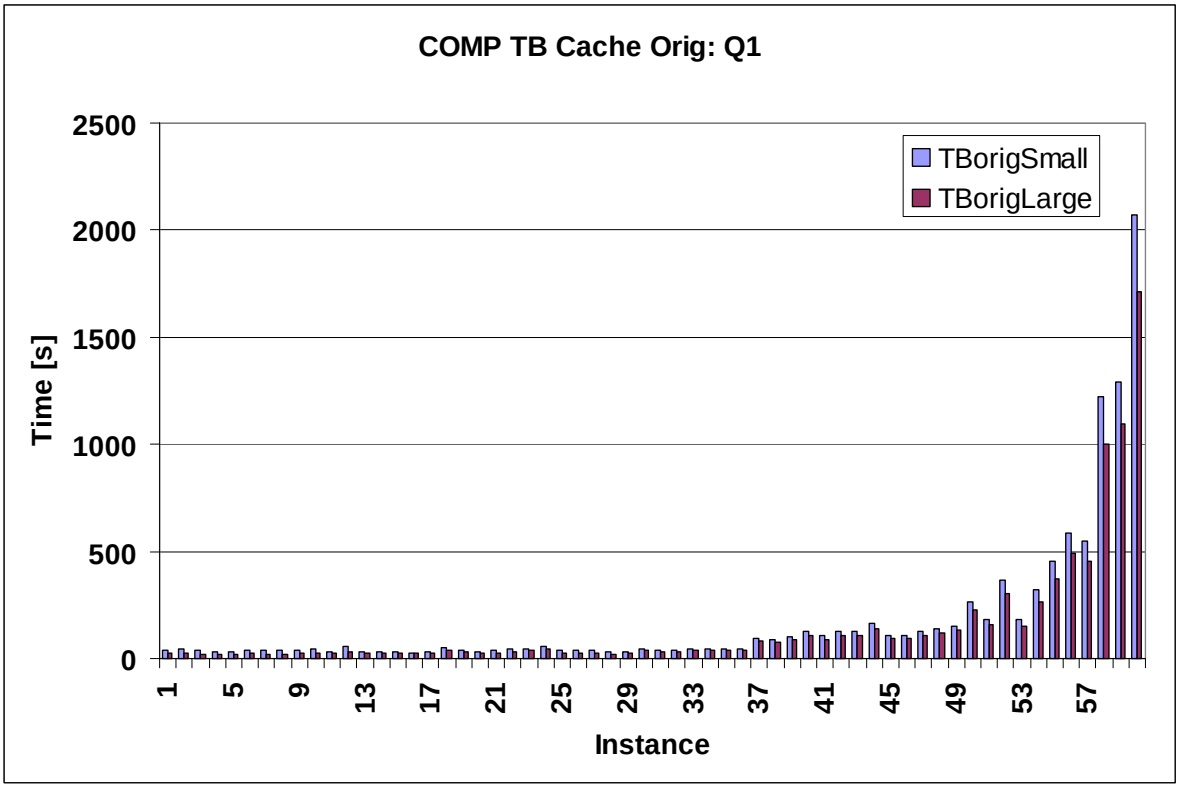
Template Q310

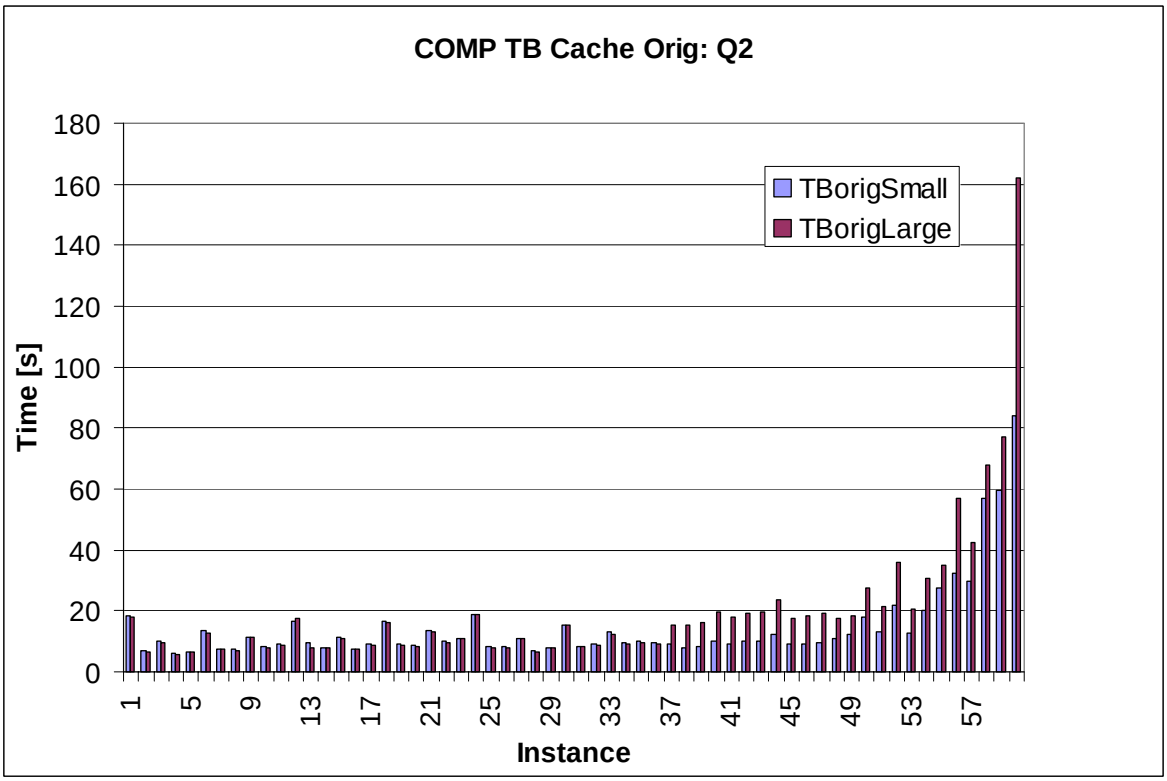
```
select
      prd.prodgroupo_code as product_group_code
     ,cal.calendar_date   as calendar_date
     ,zap.description     as warehouse_description
     ,fact.doc_num        as fact_document_num
     ,prd.code            as product_code
     ,trx.abbreviation    as transaction_abbreviation
from
     product     prd
    ,calendar    cal
    ,warehouse   zap
```

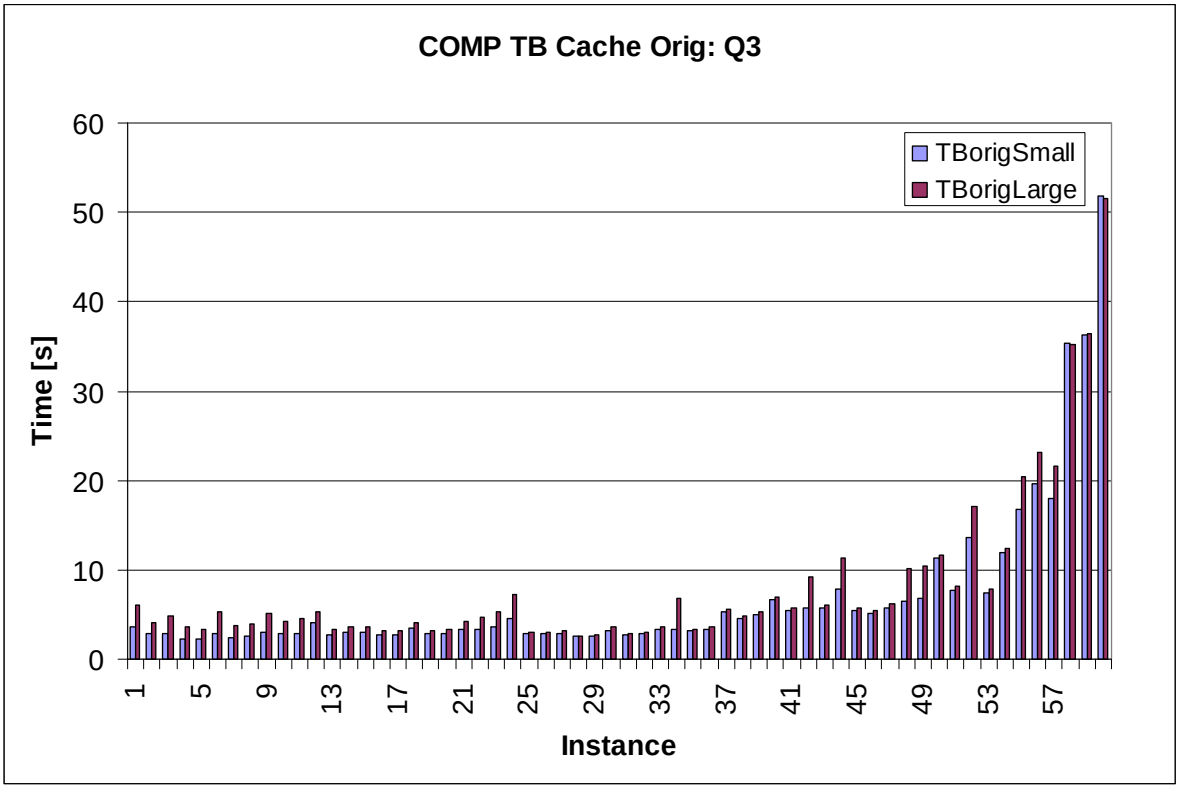
```
    ,fact        fact
    ,transaction trx
where prd.dwh_key = fact.prd_dwh_key
  and   cal.dwh_key = fact.cal_trans_dwh_key
  and   zap.dwh_key = fact.whs_issue_dwh_key
  and   trx.dwh_key = fact.trx_dwh_key
  and   trx.abbreviation in ('ÁÑÃ','ÐÄ3','ÐÄ4','ÐÄÐ','ÐË1','ÐË4','ÐÔ1','ÐÔ4')
  and   cal.calendar_date      >= ${BEGIN_DATE}
  and   cal.calendar_date      <= ${END_DATE}
  and   prd.prodgroup_code     < '60000'
```

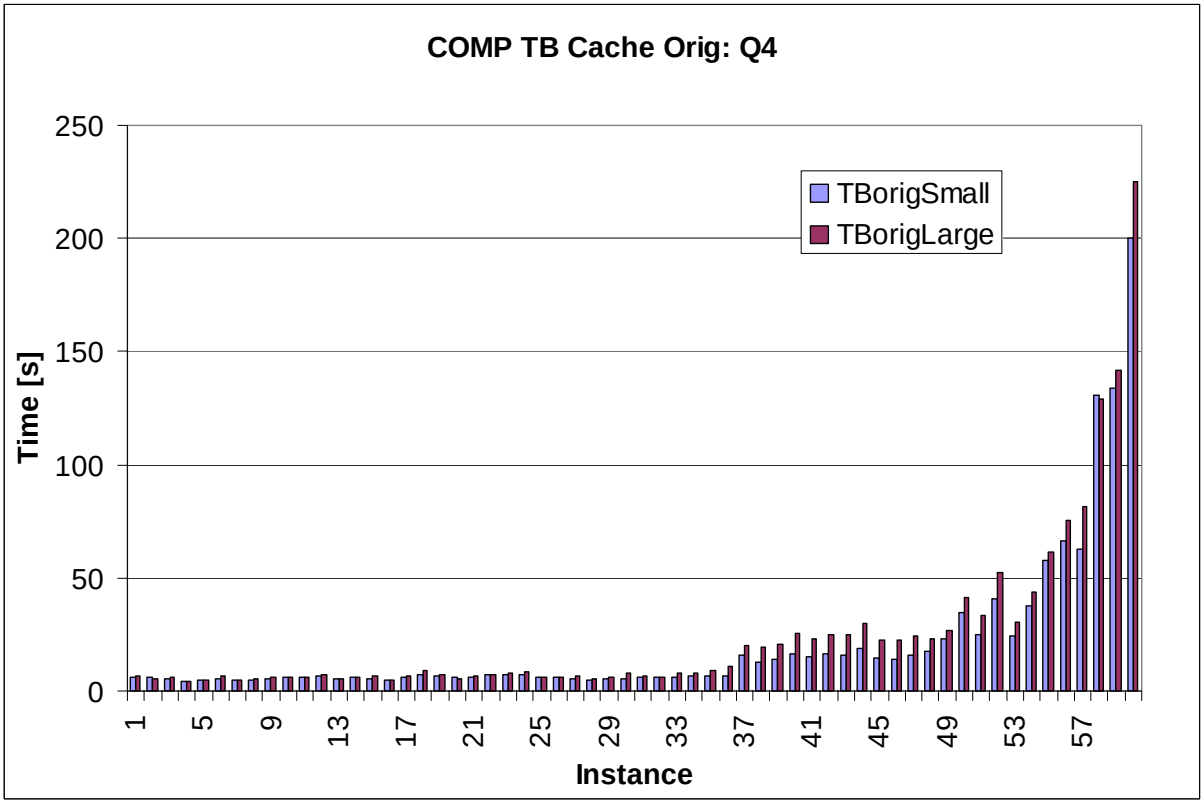
Appendix E: Complete Results for Measurements

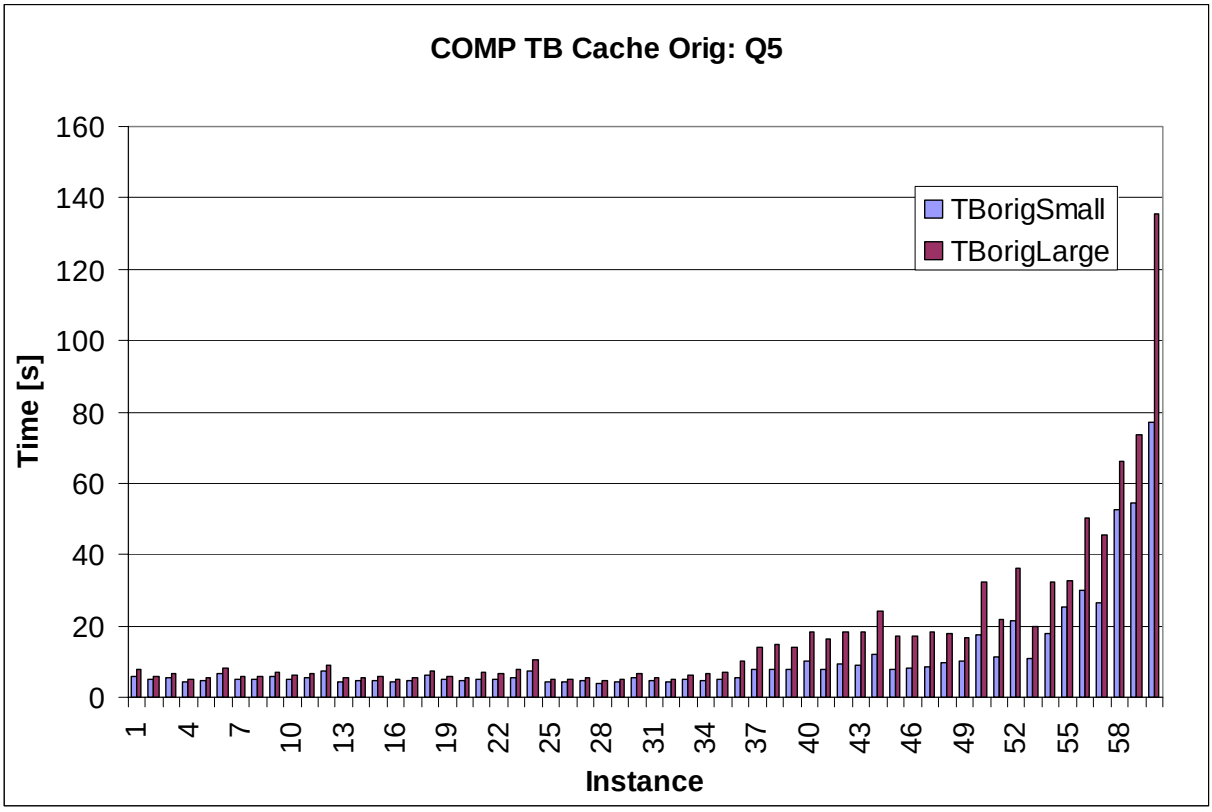
Comparison of Cache

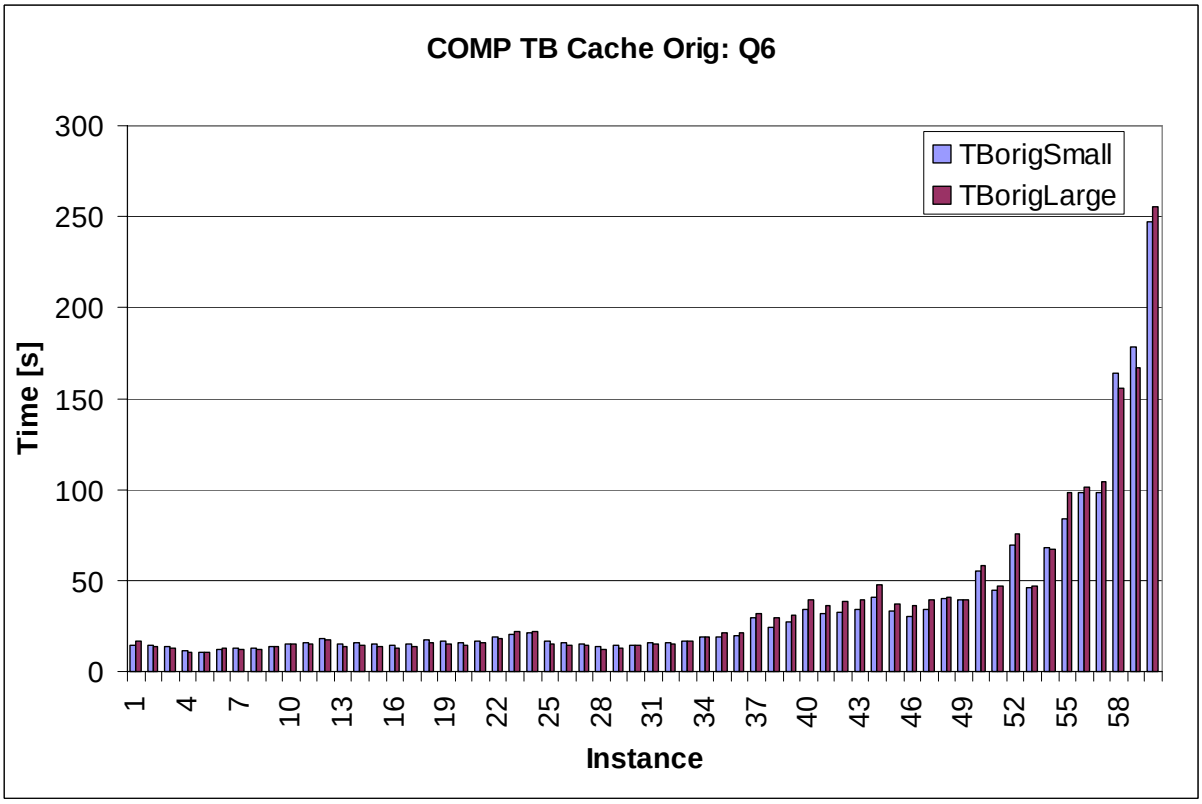


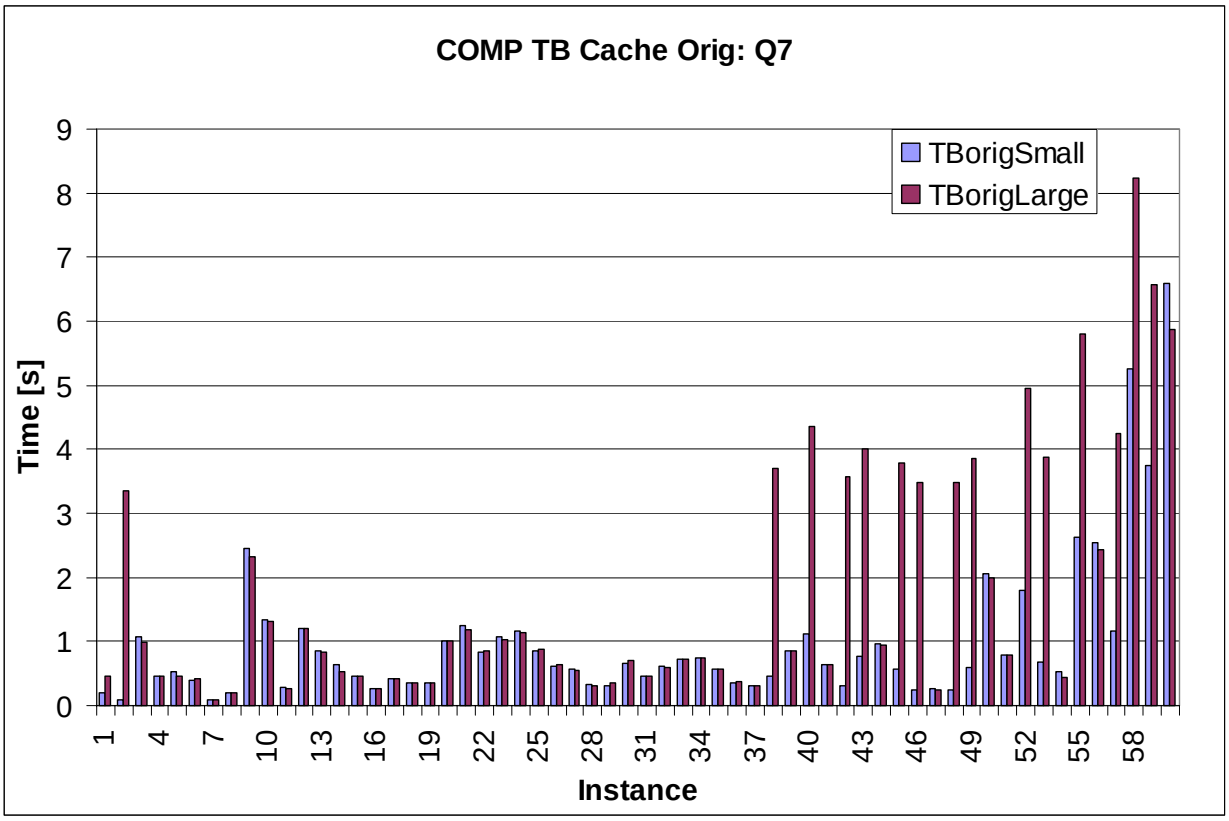


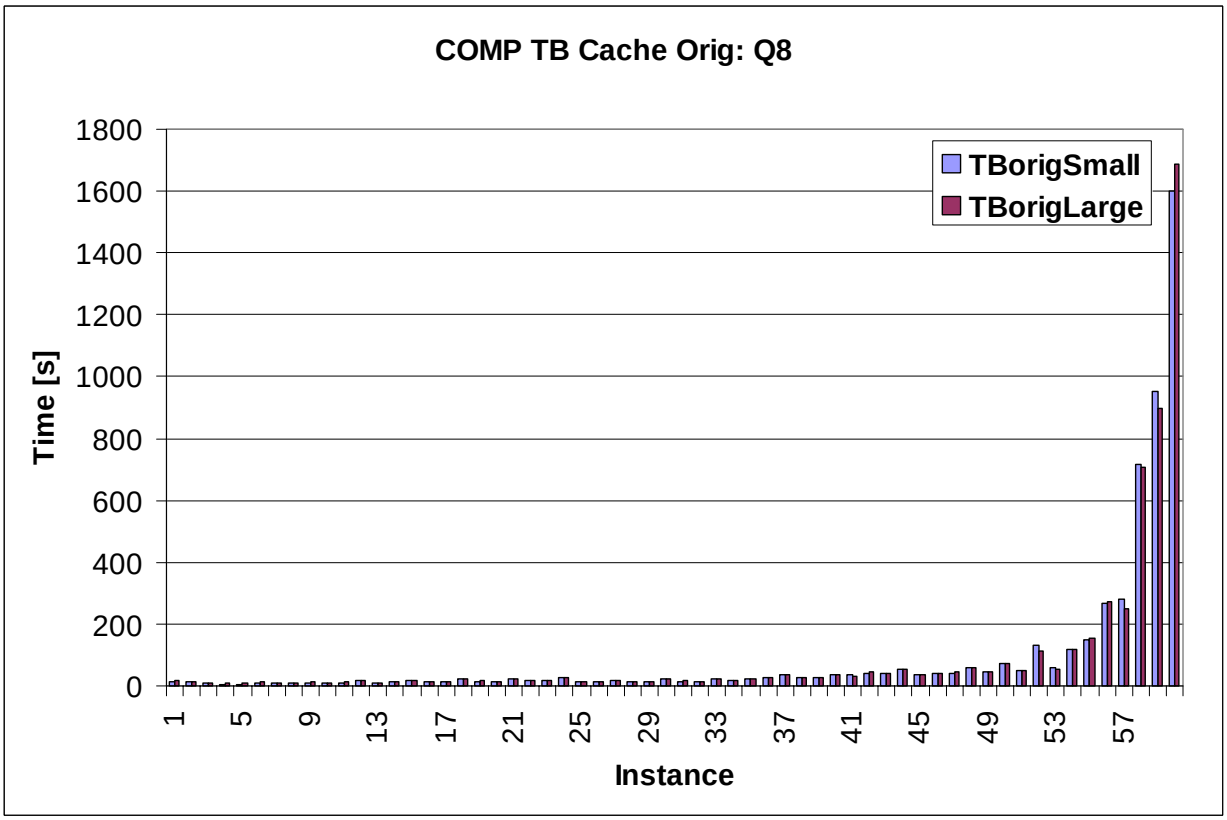

COMP TB Cache Orig: Q3
Time [s]
Instance
TBorigSmall
TBorigLarge
0
10
20
30
40
50
60
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57

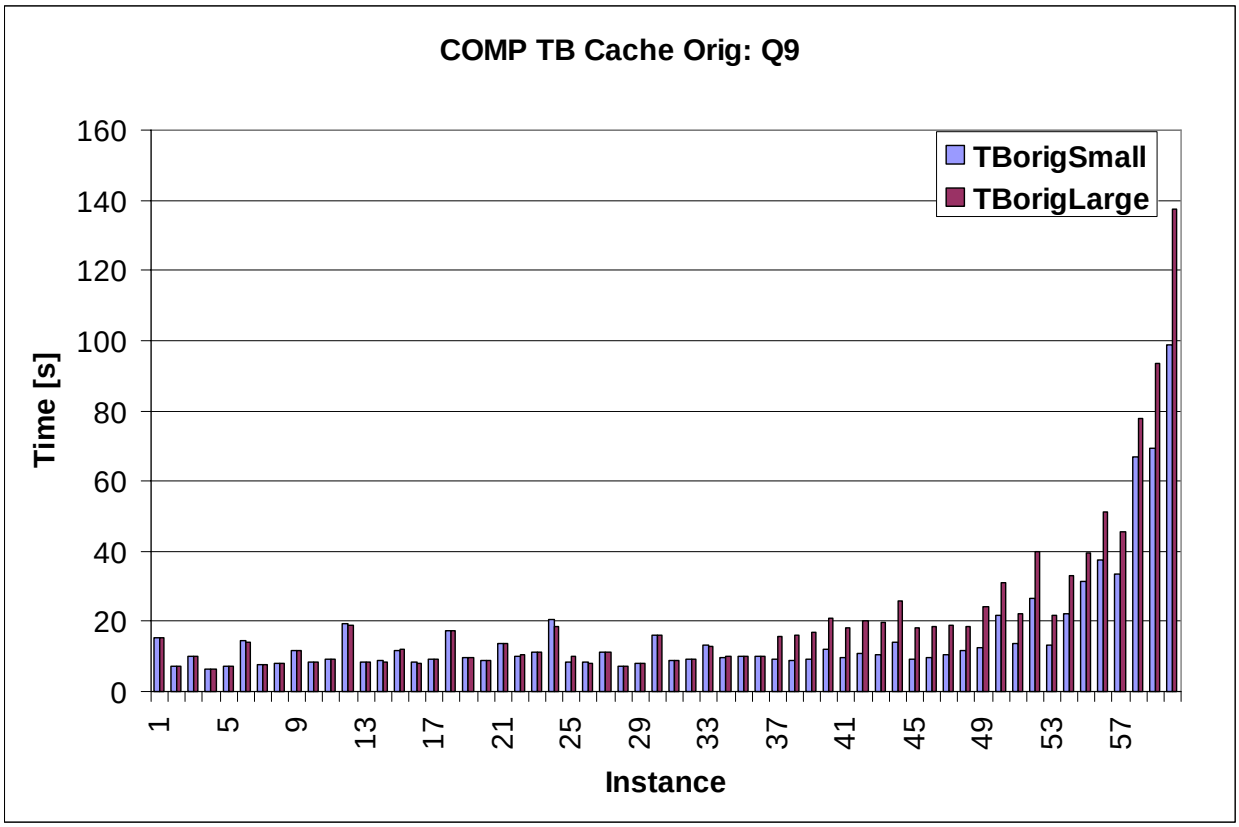

COMP TB Cache Orig: Q4
Time [s]
Instance
TBorigSmall
TBorigLarge
0
50
100
150
200
250
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57

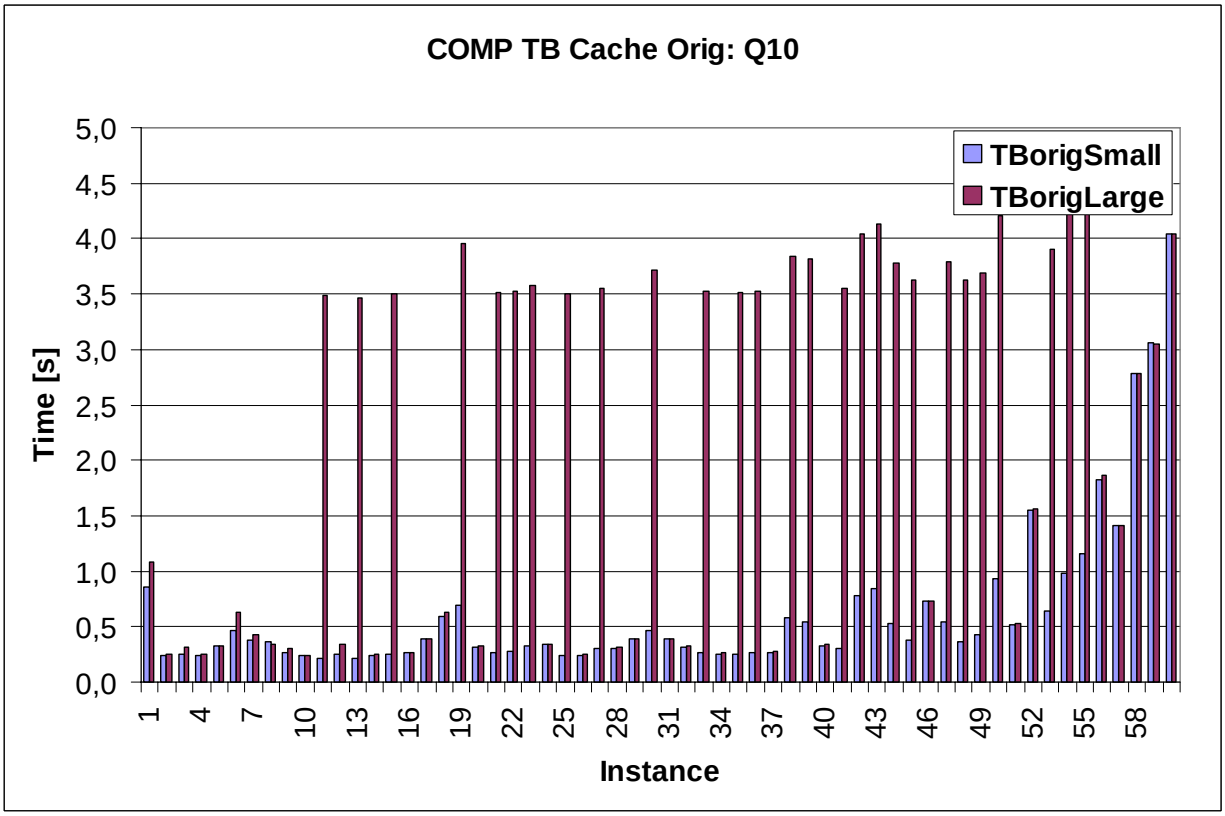

COMP TB Cache Orig: Q5
Time [s]
Instance
TBorigSmall
TBorigLarge
0
20
40
60
80
100
120
140
160
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

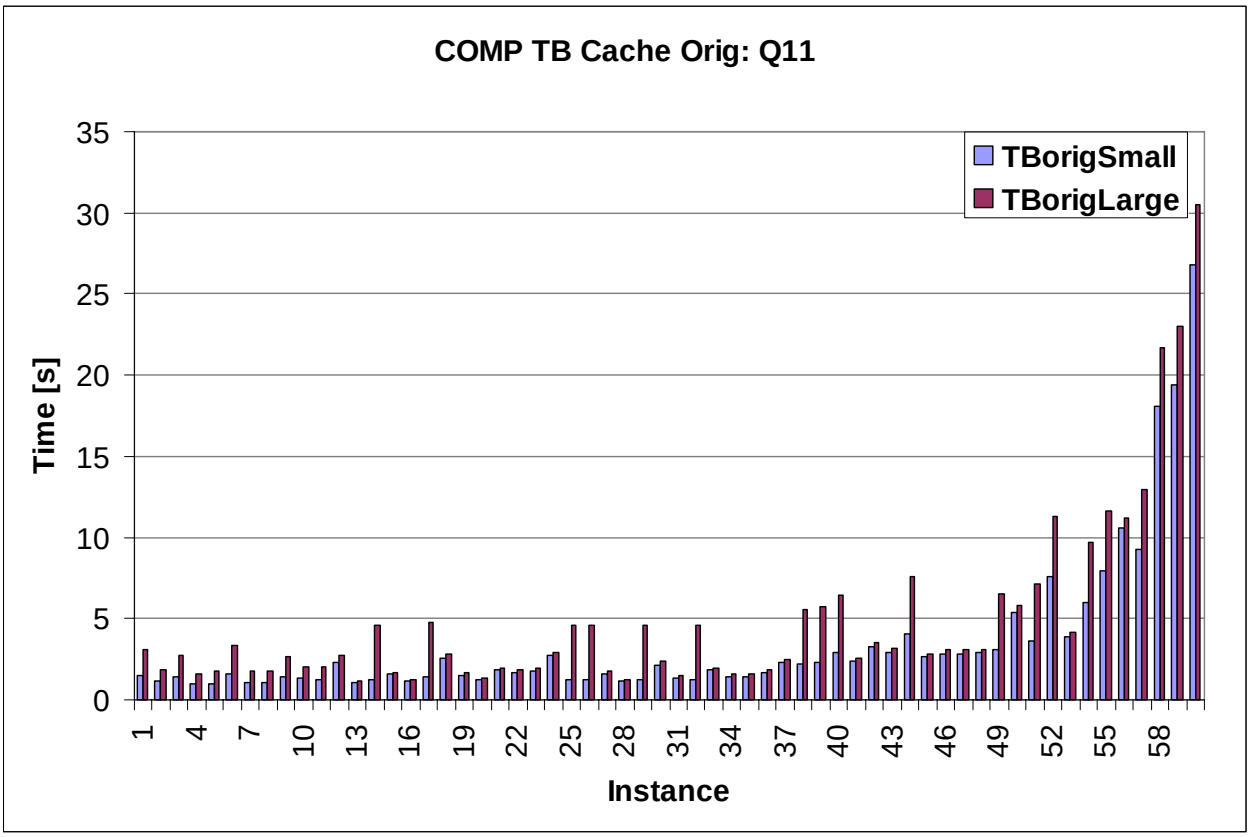

COMP TB Cache Orig: Q6
Time [s]
Instance
TBorigSmall
TBorigLarge
0
50
100
150
200
250
300
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

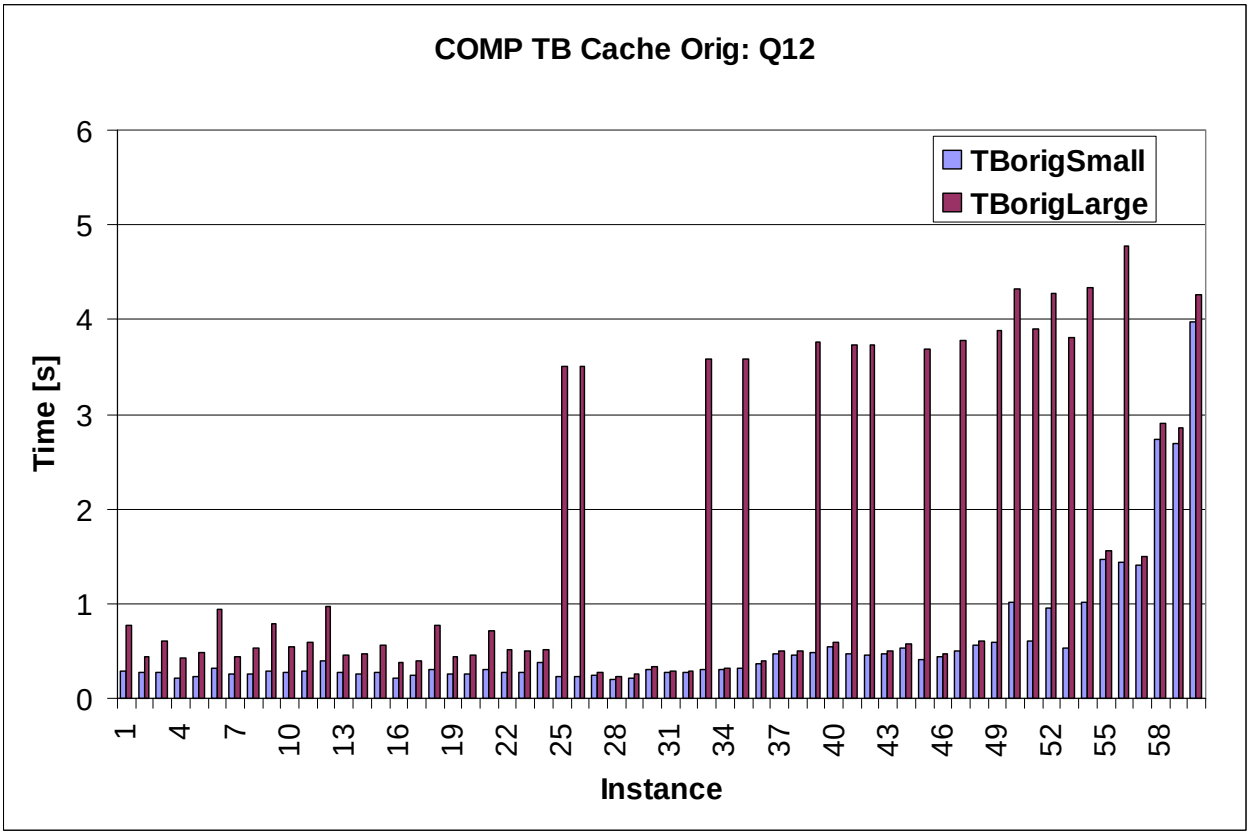

COMP TB Cache Orig: Q7
TBorigSmall
TBorigLarge
Time [s]
Instance
0
1
2
3
4
5
6
7
8
9
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

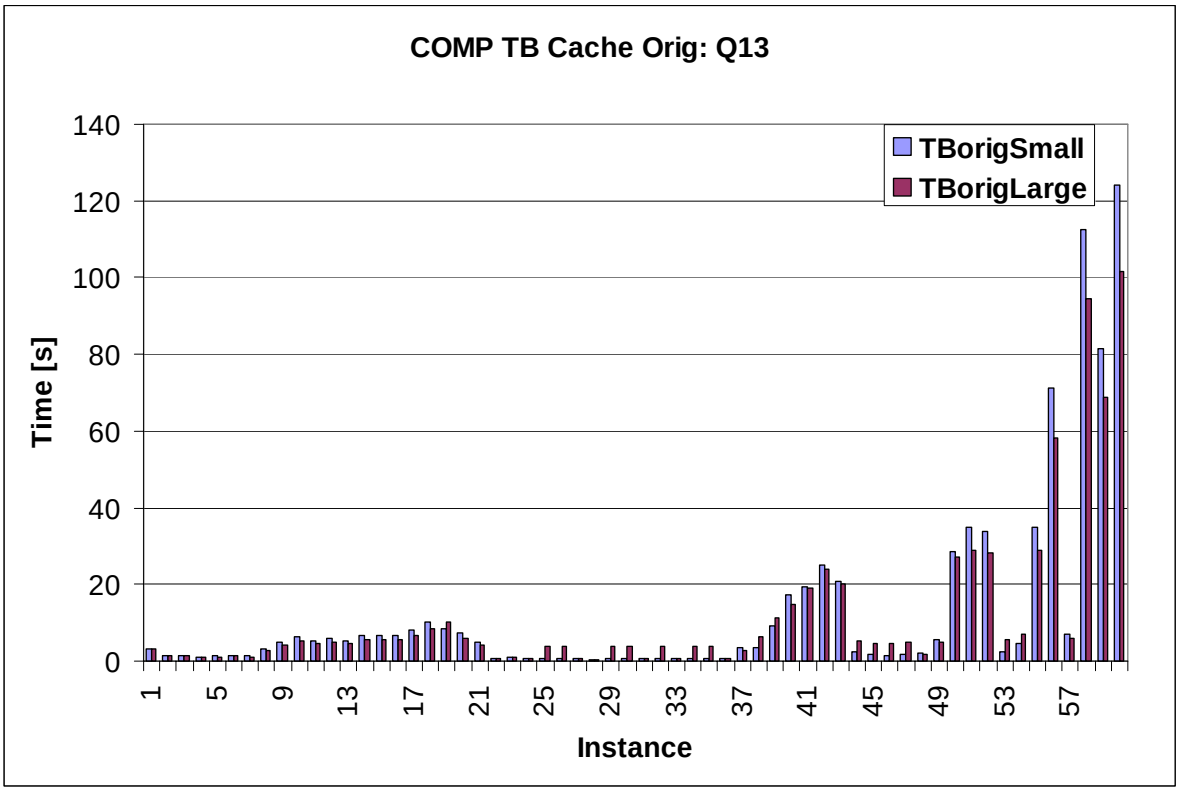

COMP TB Cache Orig: Q8
TBorigSmall
TBorigLarge
Time [s]
Instance
0
200
400
600
800
1000
1200
1400
1600
1800
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57

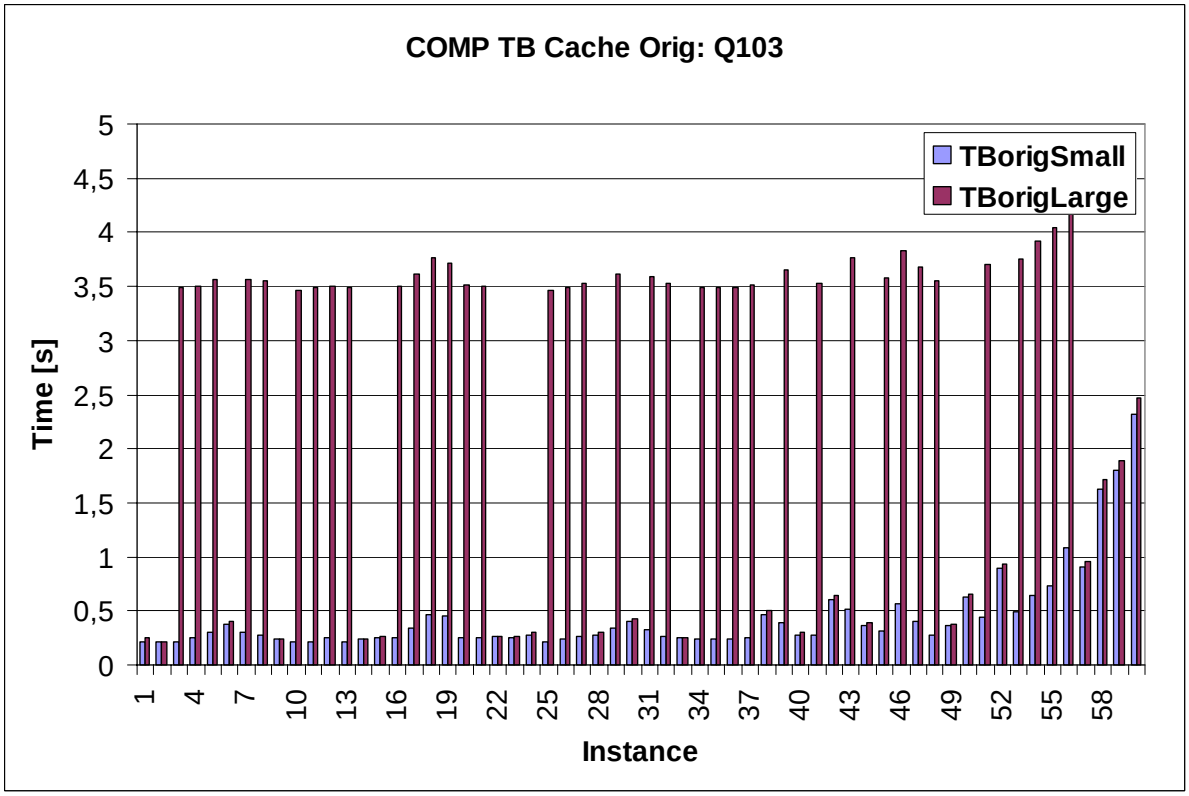

COMP TB Cache Orig: Q9
Time [s]
Instance
TBorigSmall
TBorigLarge
0
20
40
60
80
100
120
140
160
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57

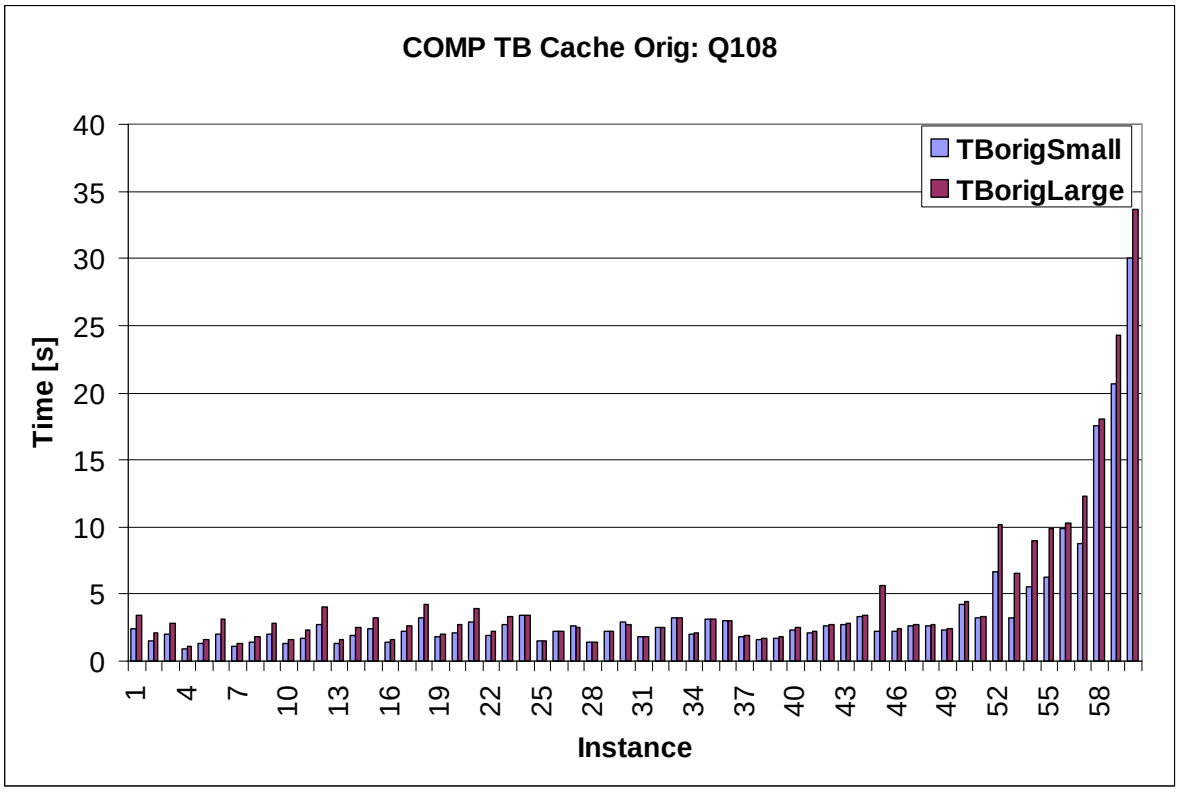

COMP TB Cache Orig: Q10
Time [s]
Instance
TBorigSmall
TBorigLarge
0,0
0,5
1,0
1,5
2,0
2,5
3,0
3,5
4,0
4,5
5,0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

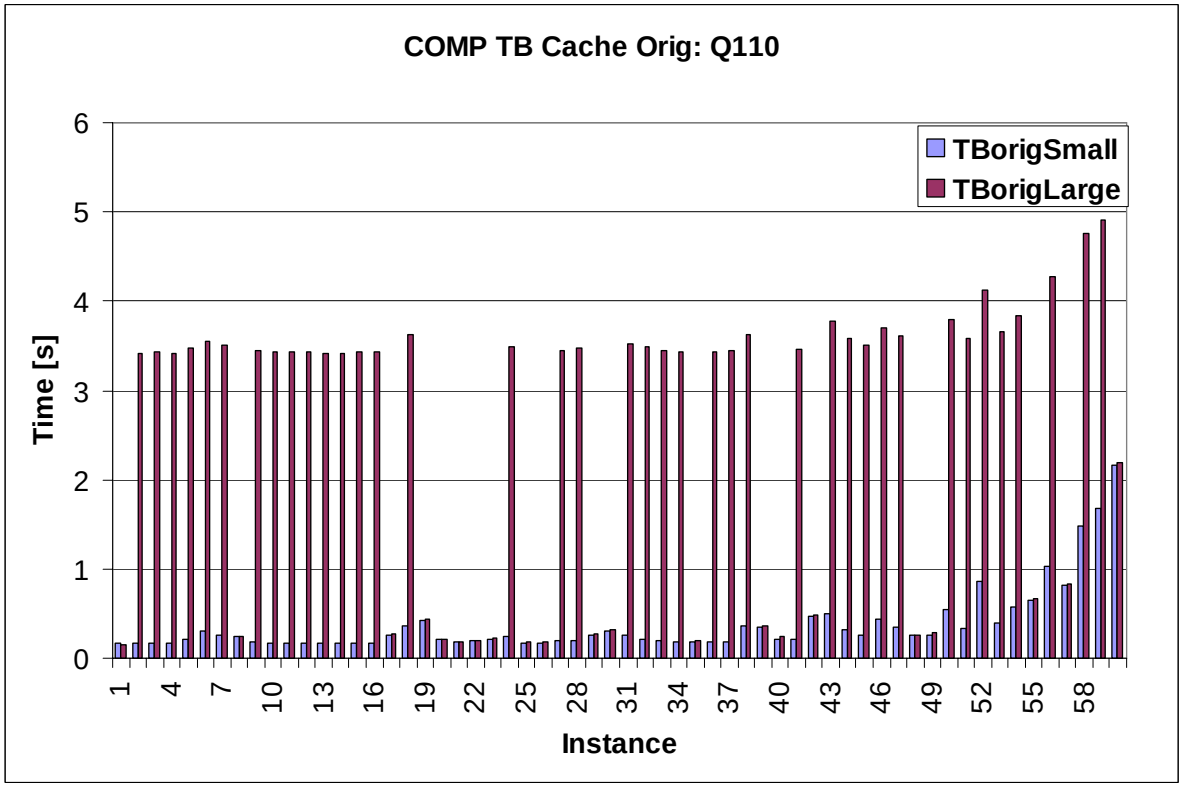

COMP TB Cache Orig: Q11
TBorigSmall
TBorigLarge
Time [s]
Instance
0
5
10
15
20
25
30
35
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

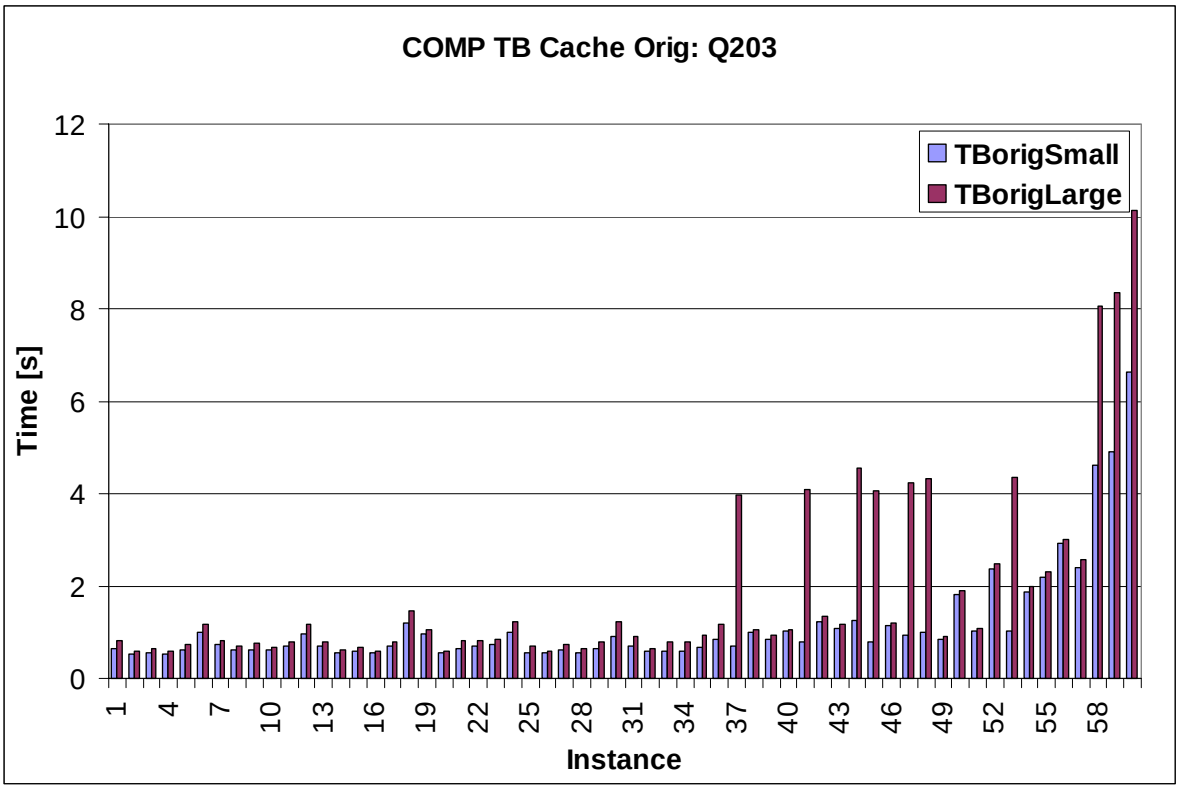

COMP TB Cache Orig: Q12
TBorigSmall
TBorigLarge
Time [s]
Instance
0
1
2
3
4
5
6
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

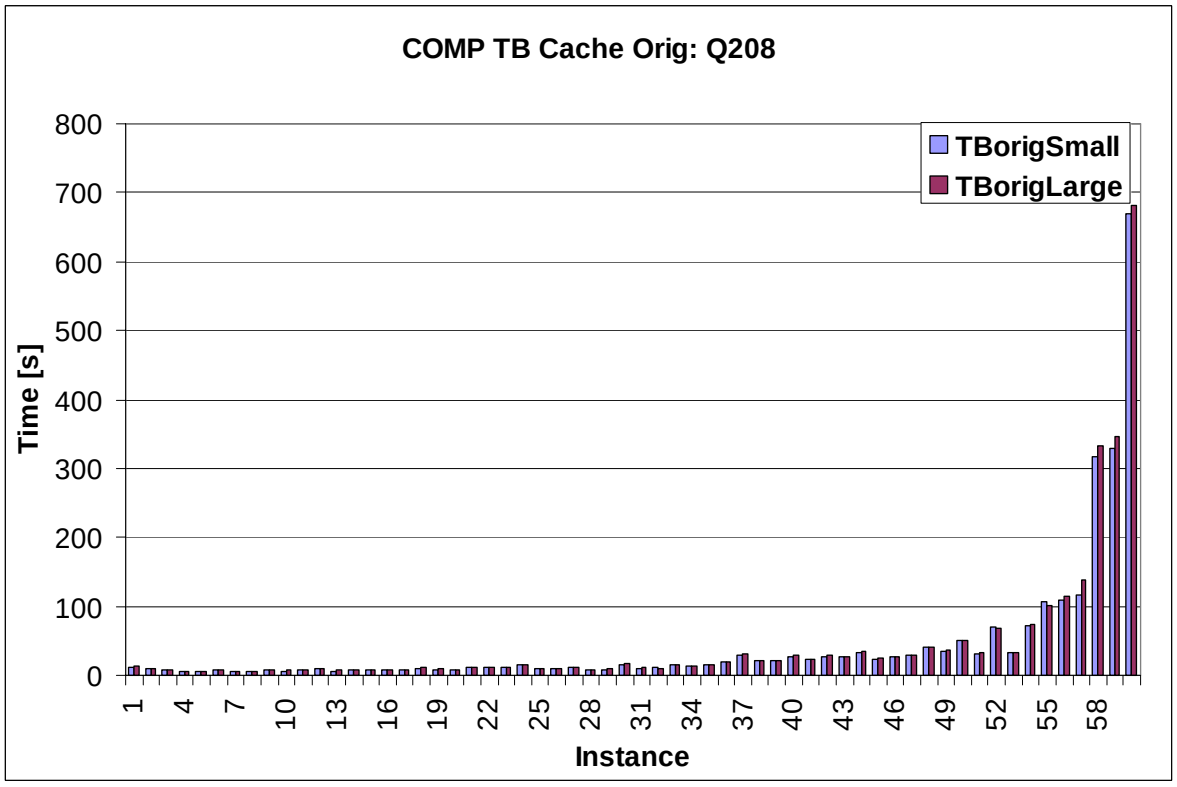

COMP TB Cache Orig: Q13
Time [s]
Instance
TBorigSmall
TBorigLarge
0
20
40
60
80
100
120
140
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57

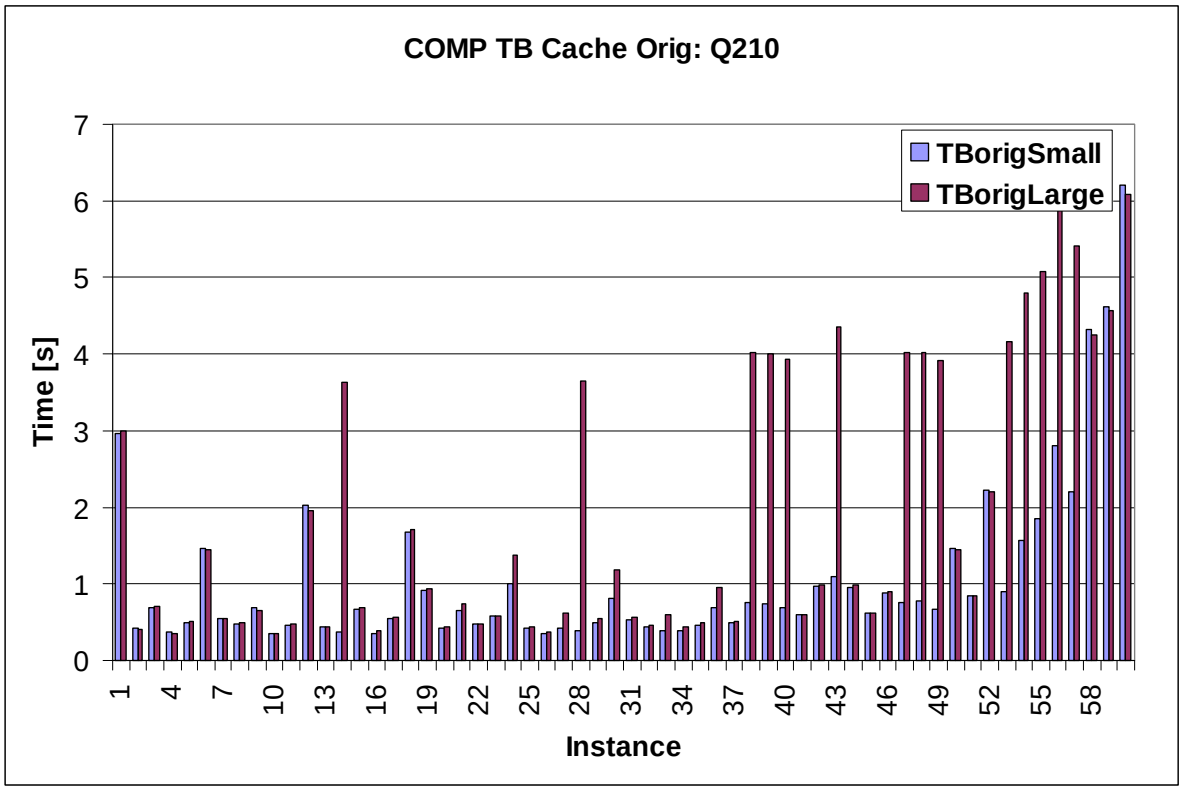

COMP TB Cache Orig: Q103
Time [s]
Instance
TBorigSmall
TBorigLarge
0
0,5
1
1,5
2
2,5
3
3,5
4
4,5
5
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

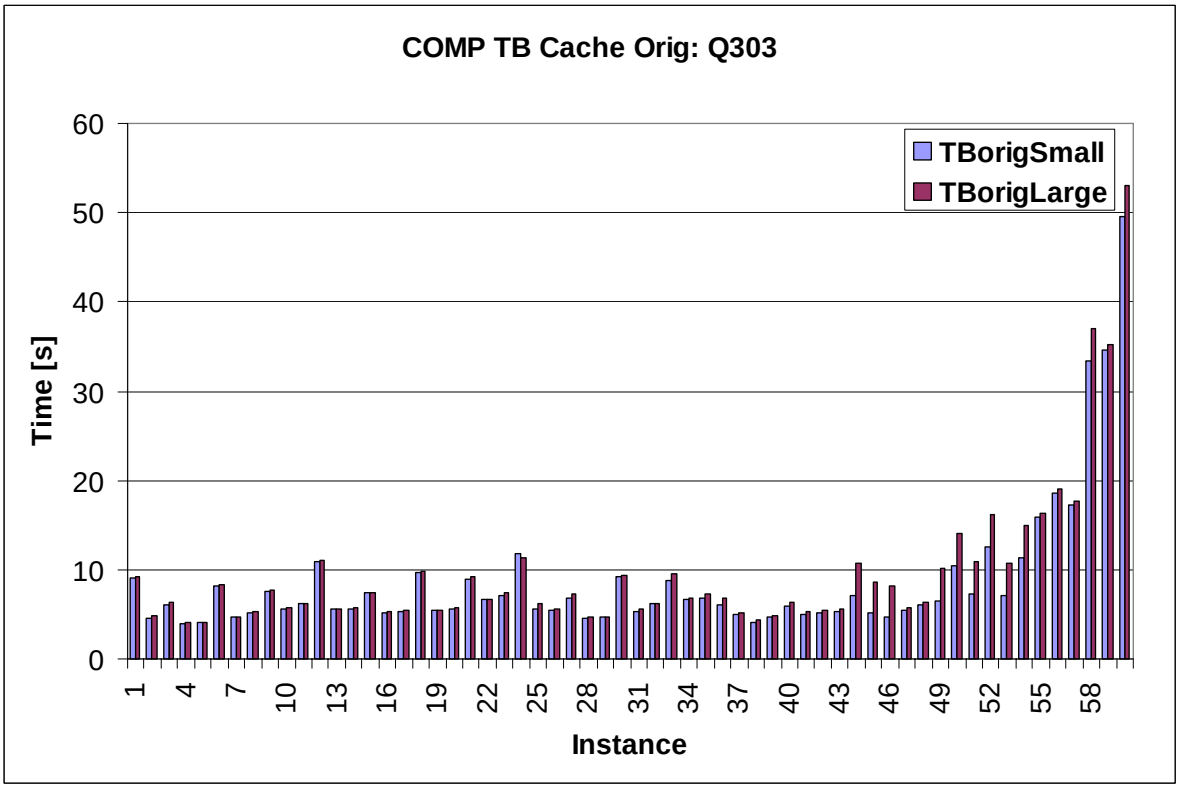

COMP TB Cache Orig: Q108
TBorigSmall
TBorigLarge
Time [s]
Instance
0
5
10
15
20
25
30
35
40
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

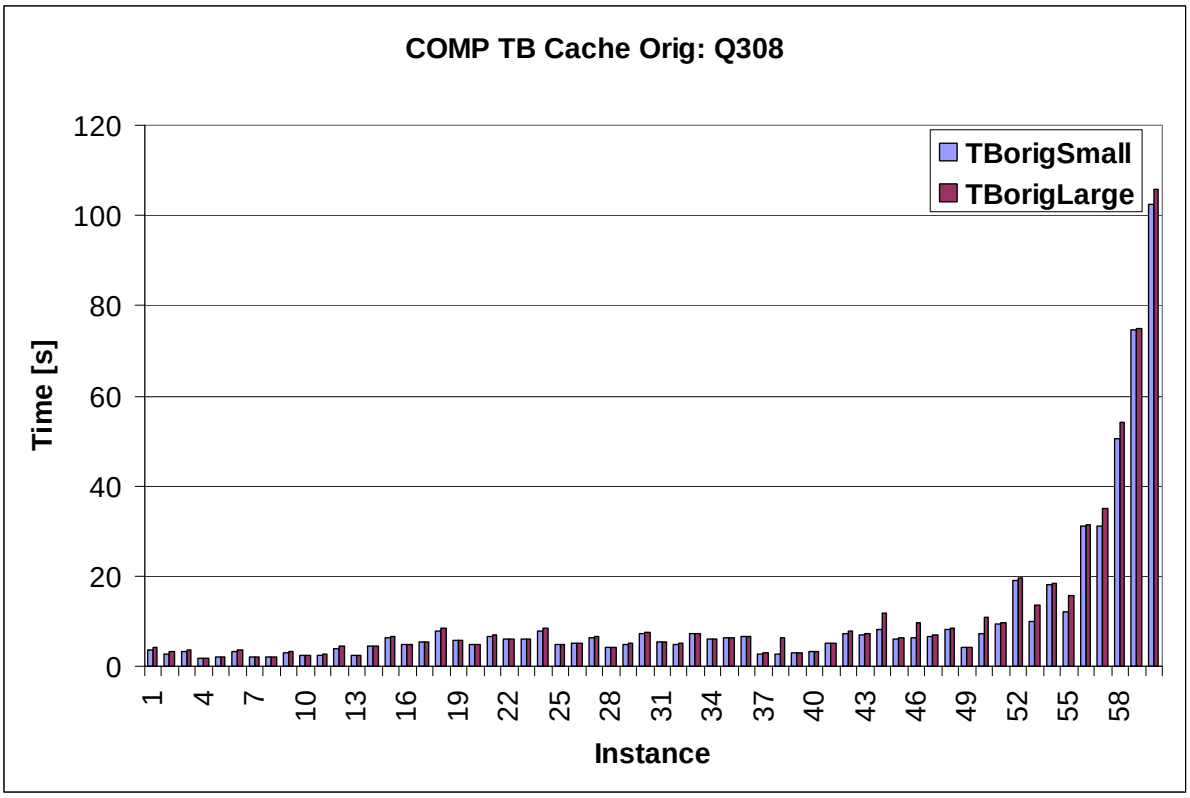

COMP TB Cache Orig: Q110
TBorigSmall
TBorigLarge
Time [s]
Instance
0
1
2
3
4
5
6
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

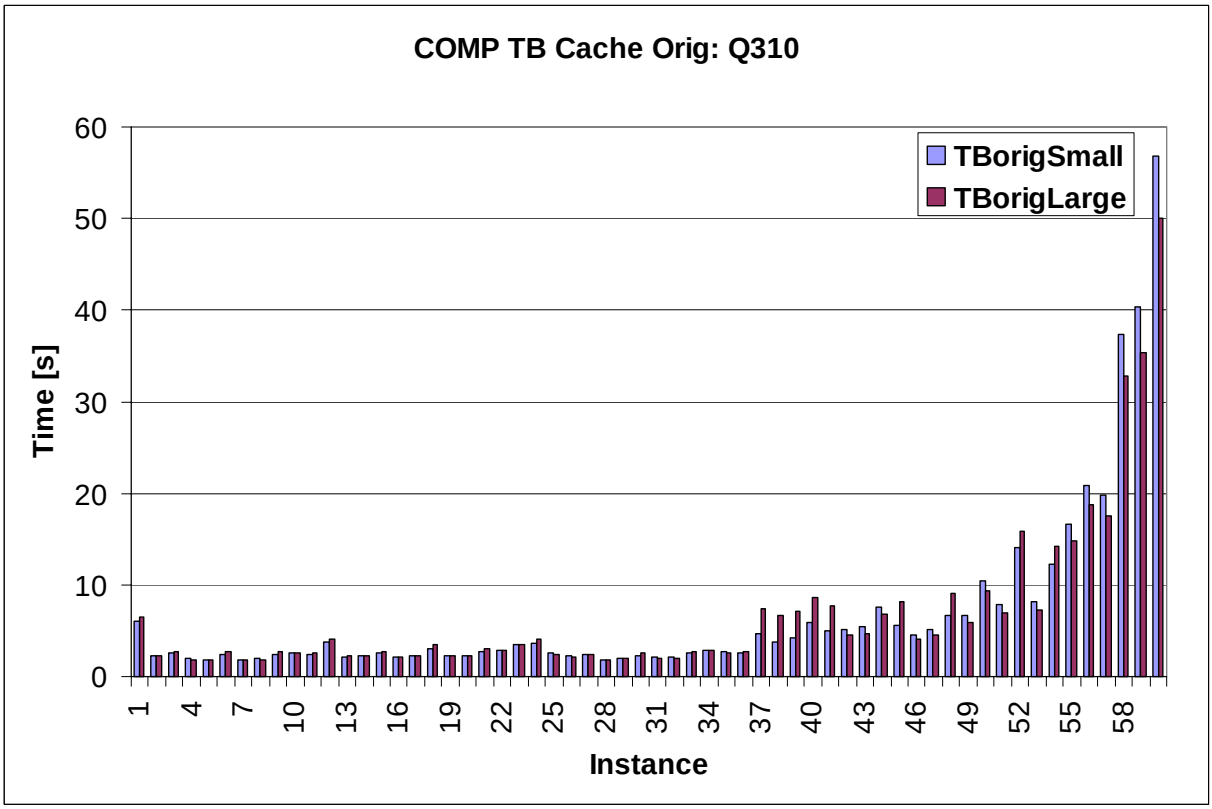

COMP TB Cache Orig: Q203
TBorigSmall
TBorigLarge
Time [s]
Instance
0
2
4
6
8
10
12
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58


COMP TB Cache Orig: Q208
TBorigSmall
TBorigLarge
Time [s]
Instance
0
100
200
300
400
500
600
700
800
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

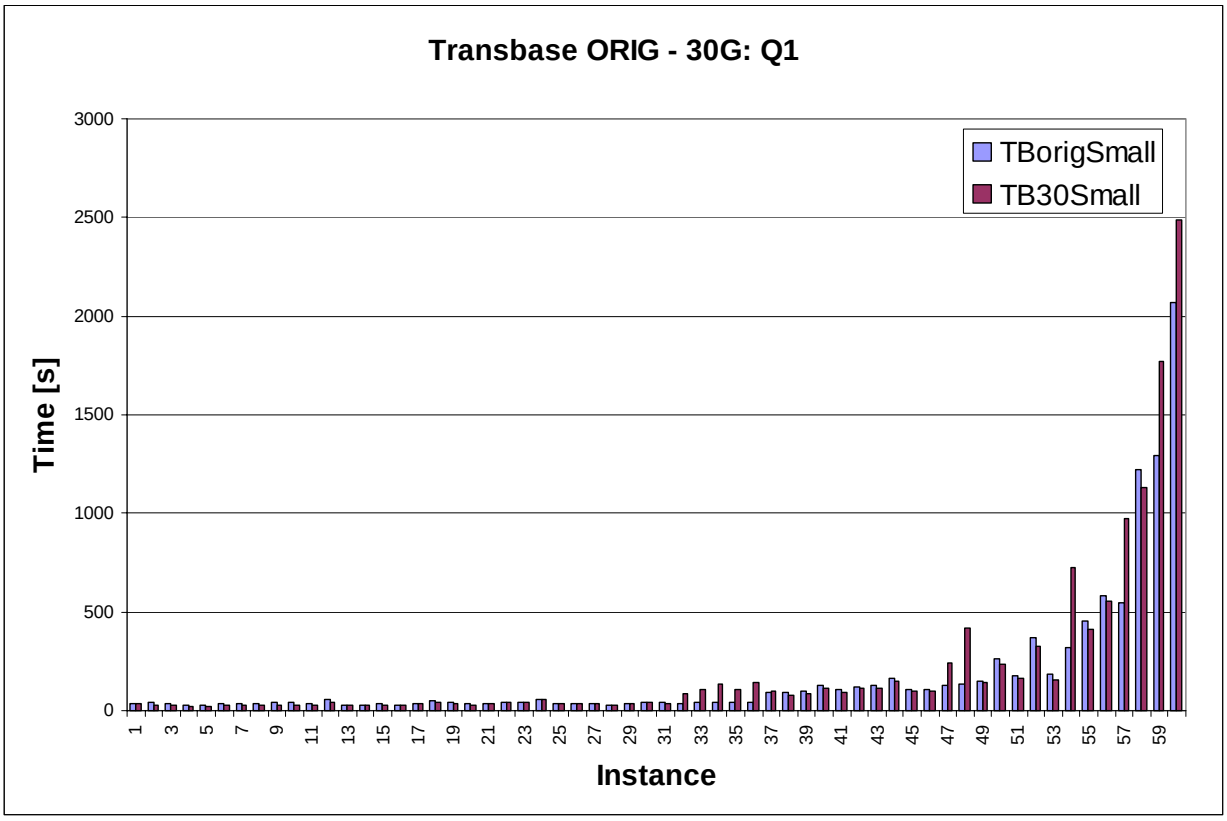

COMP TB Cache Orig: Q210
TBorigSmall
TBorigLarge
Time [s]
Instance
0
1
2
3
4
5
6
7
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

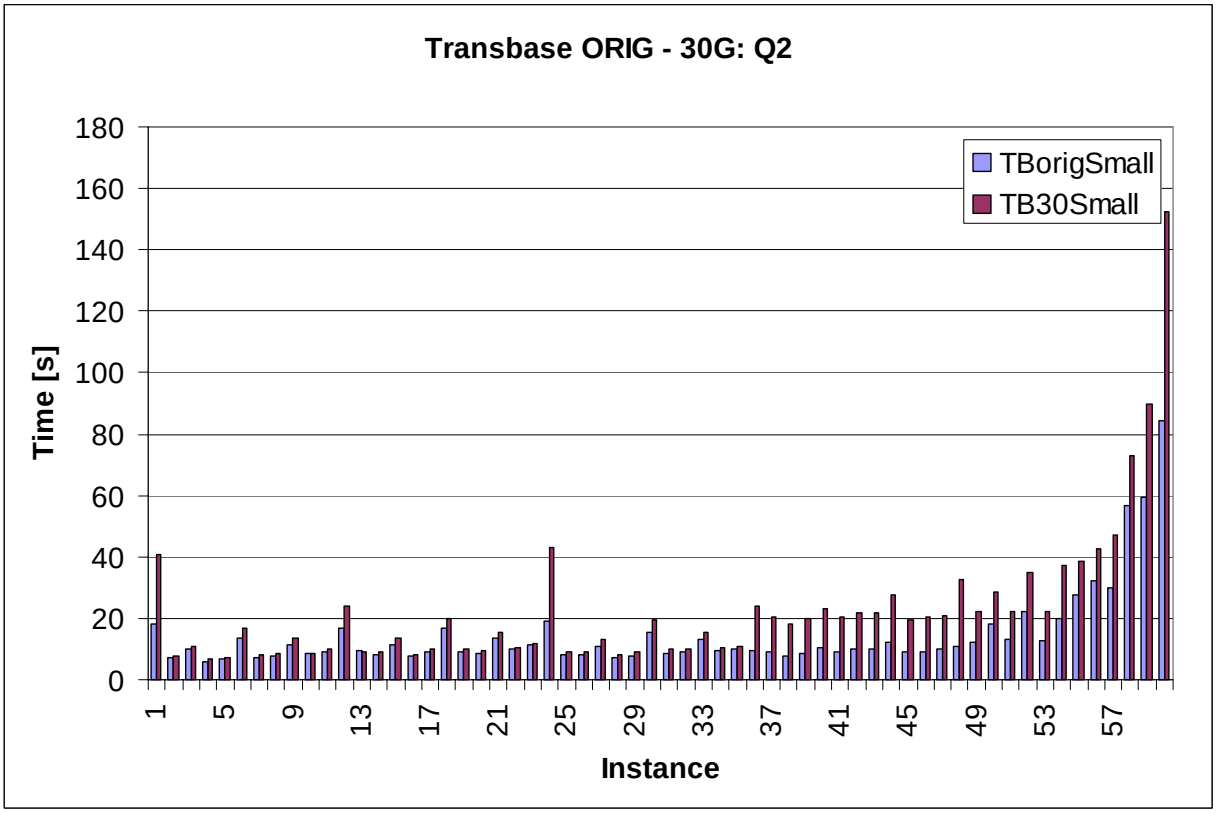

COMP TB Cache Orig: Q303
TBorigSmall
TBorigLarge
Time [s]
Instance
0
10
20
30
40
50
60
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

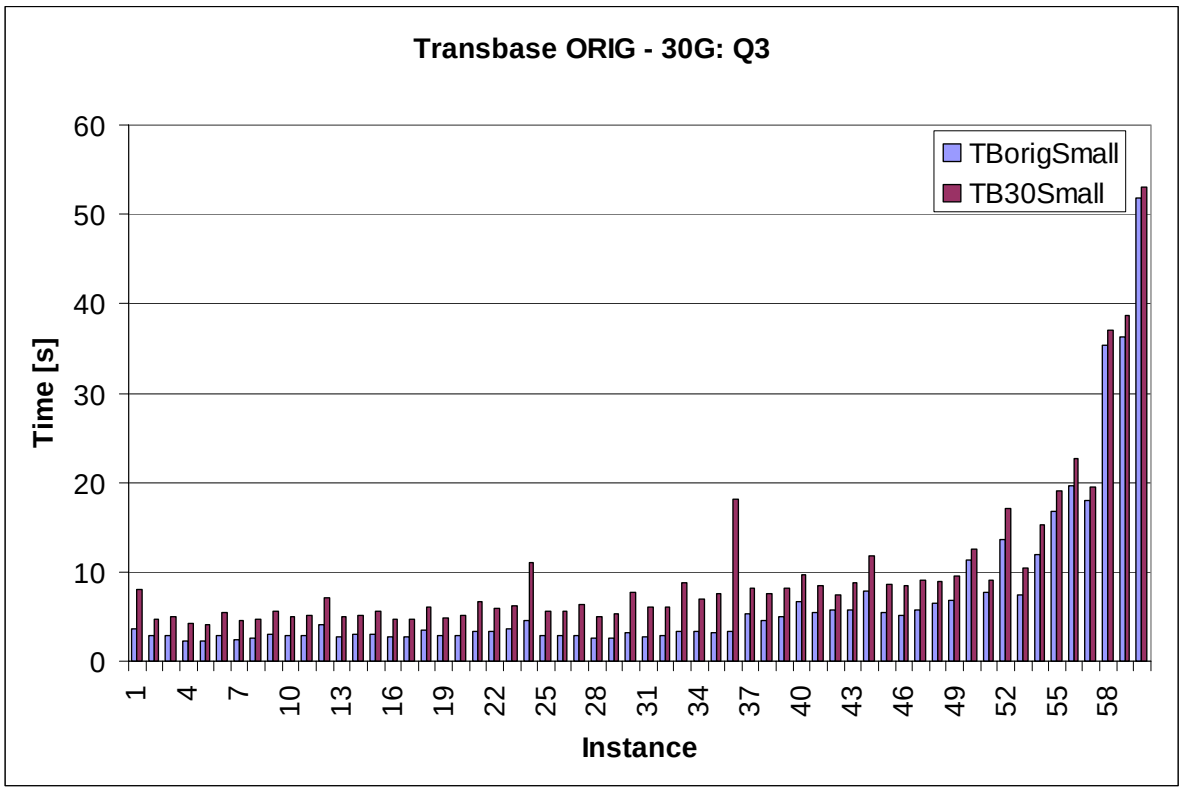

COMP TB Cache Orig: Q308
Time [s]
Instance
TBorigSmall
TBorigLarge
0
20
40
60
80
100
120
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

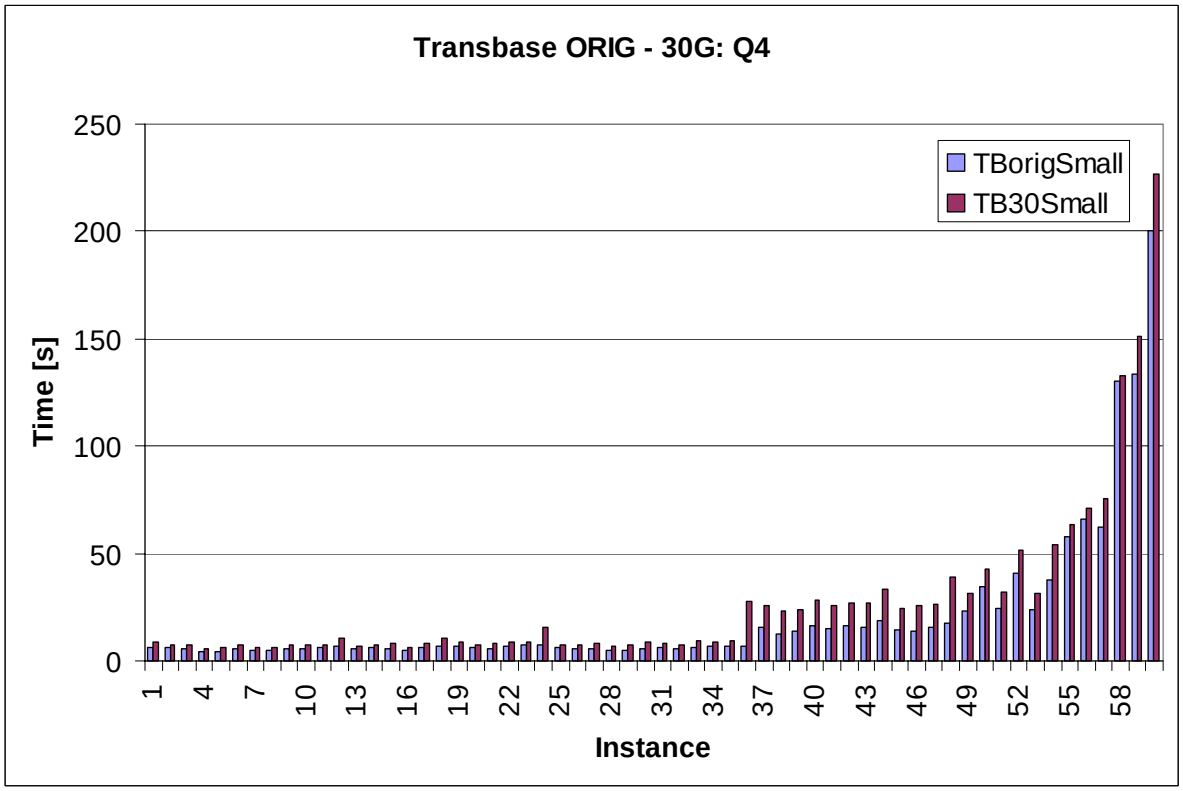

COMP TB Cache Orig: Q310
Time [s]
Instance
TBorigSmall
TBorigLarge
0
10
20
30
40
50
60
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

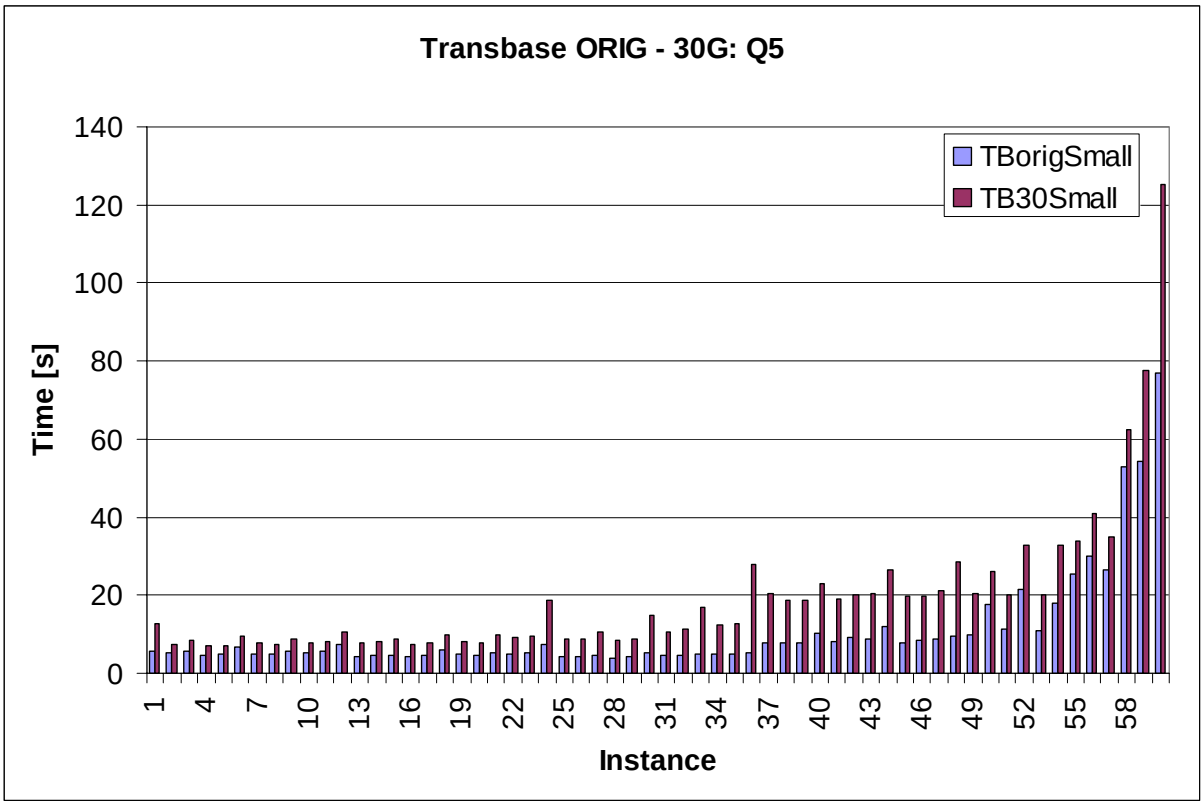
Comparison of Database Size

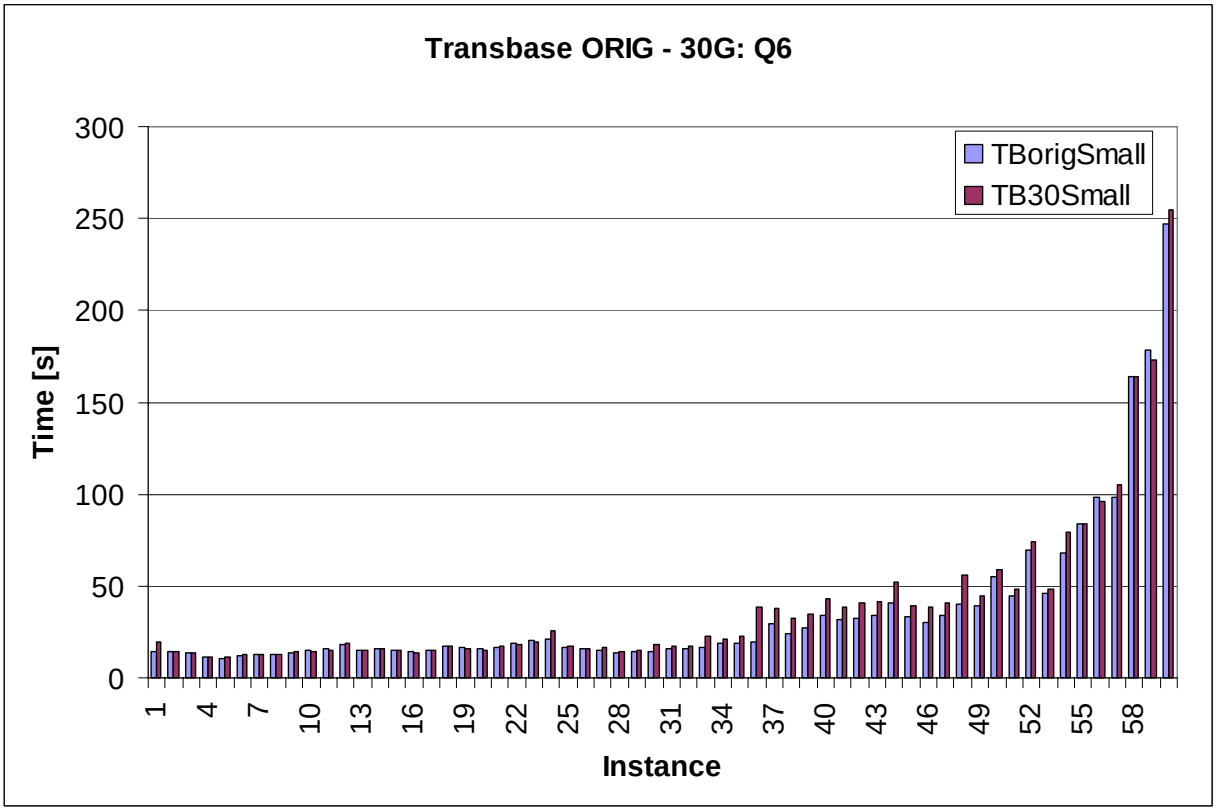


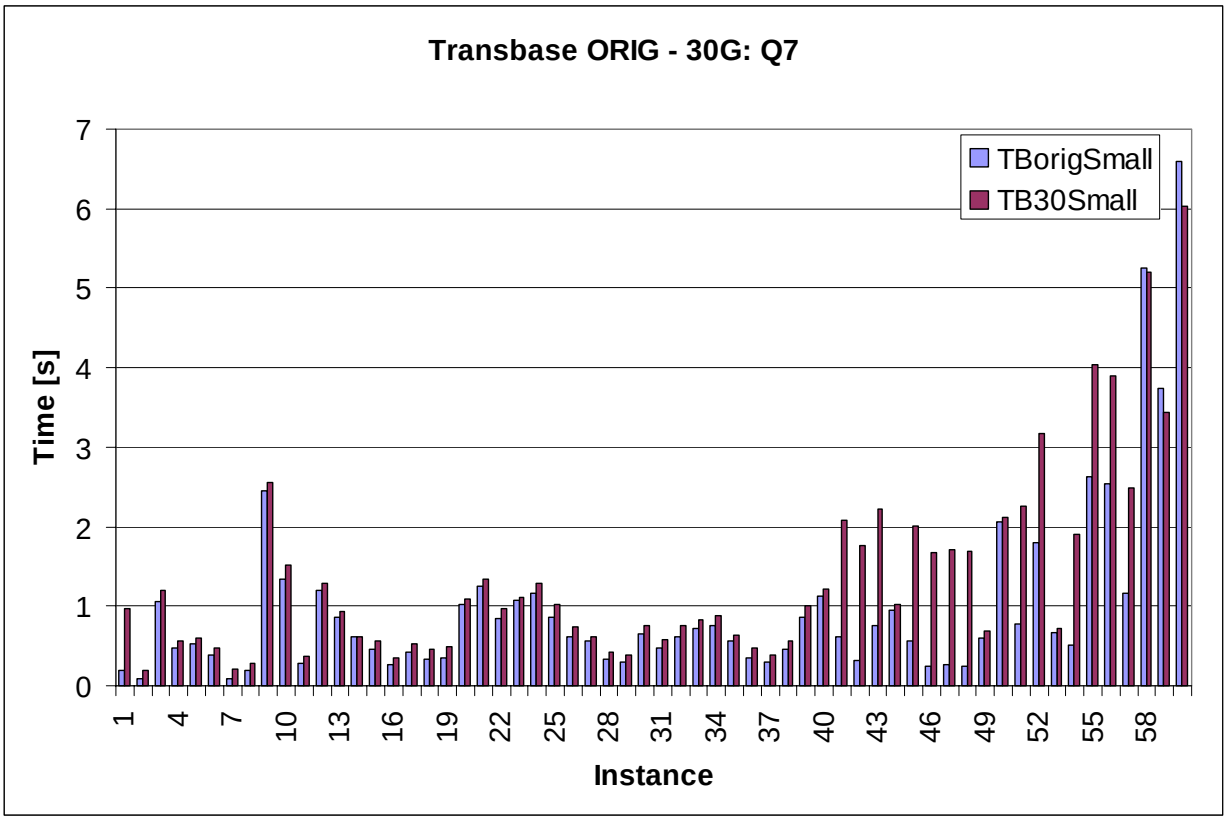


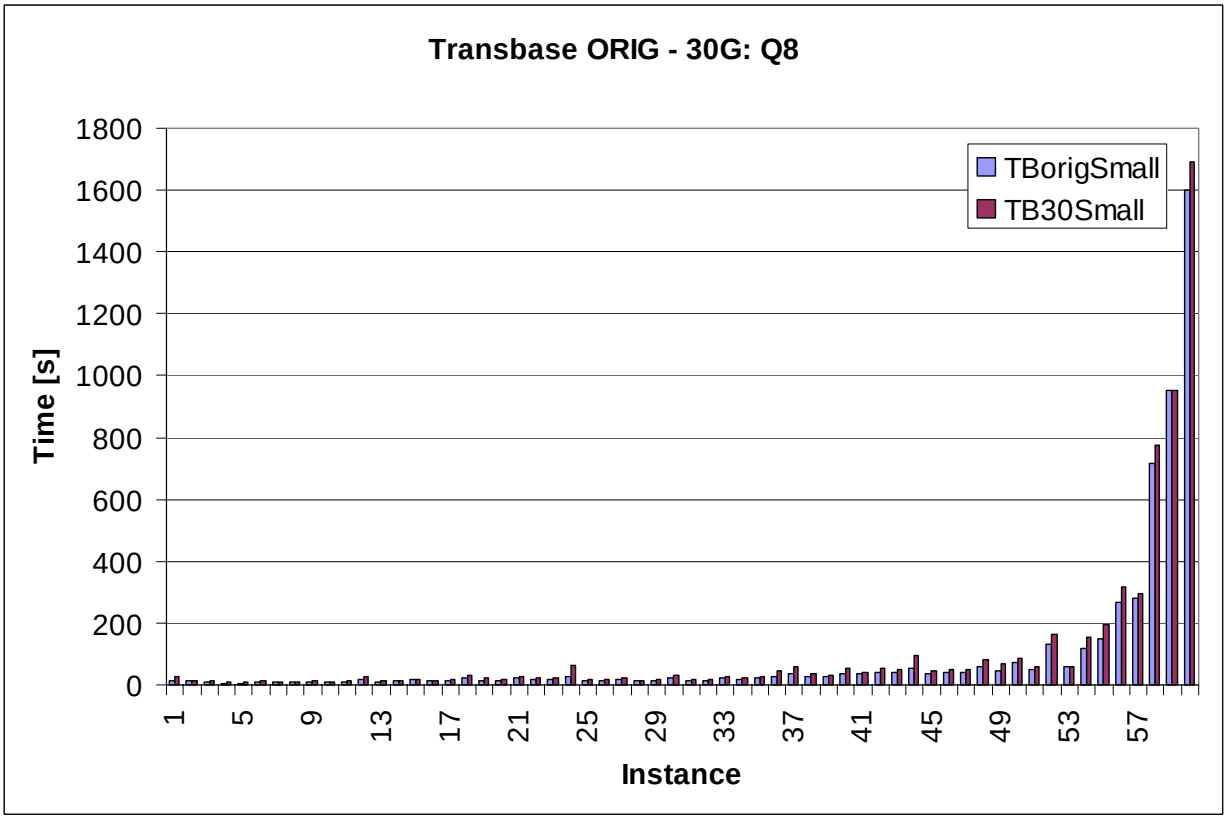

Transbase ORIG - 30G: Q3
Time [s]
Instance
TBorigSmall
TB30Small
0
10
20
30
40
50
60
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

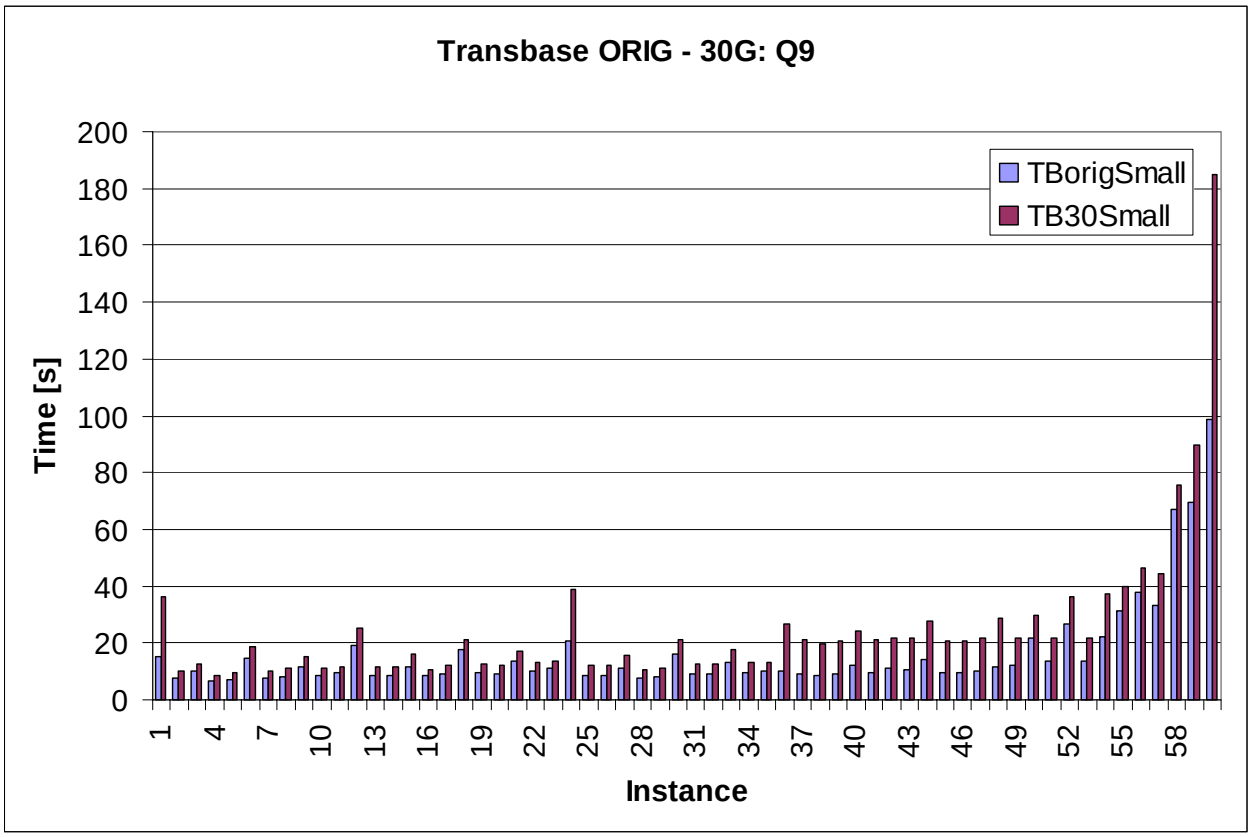

Transbase ORIG - 30G: Q4
Time [s]
Instance
TBorigSmall
TB30Small
0
50
100
150
200
250
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

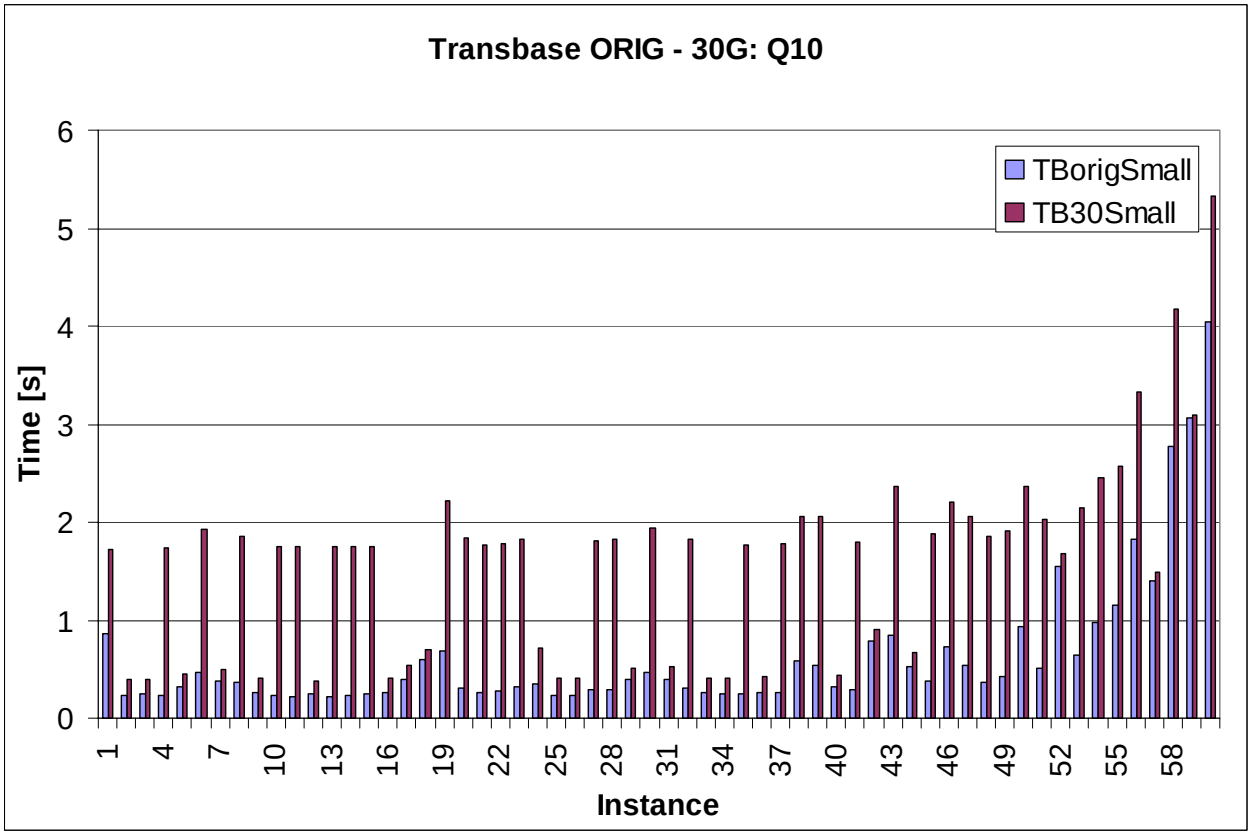

Transbase ORIG - 30G: Q5
Time [s]
Instance
TBorigSmall
TB30Small
0
20
40
60
80
100
120
140
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

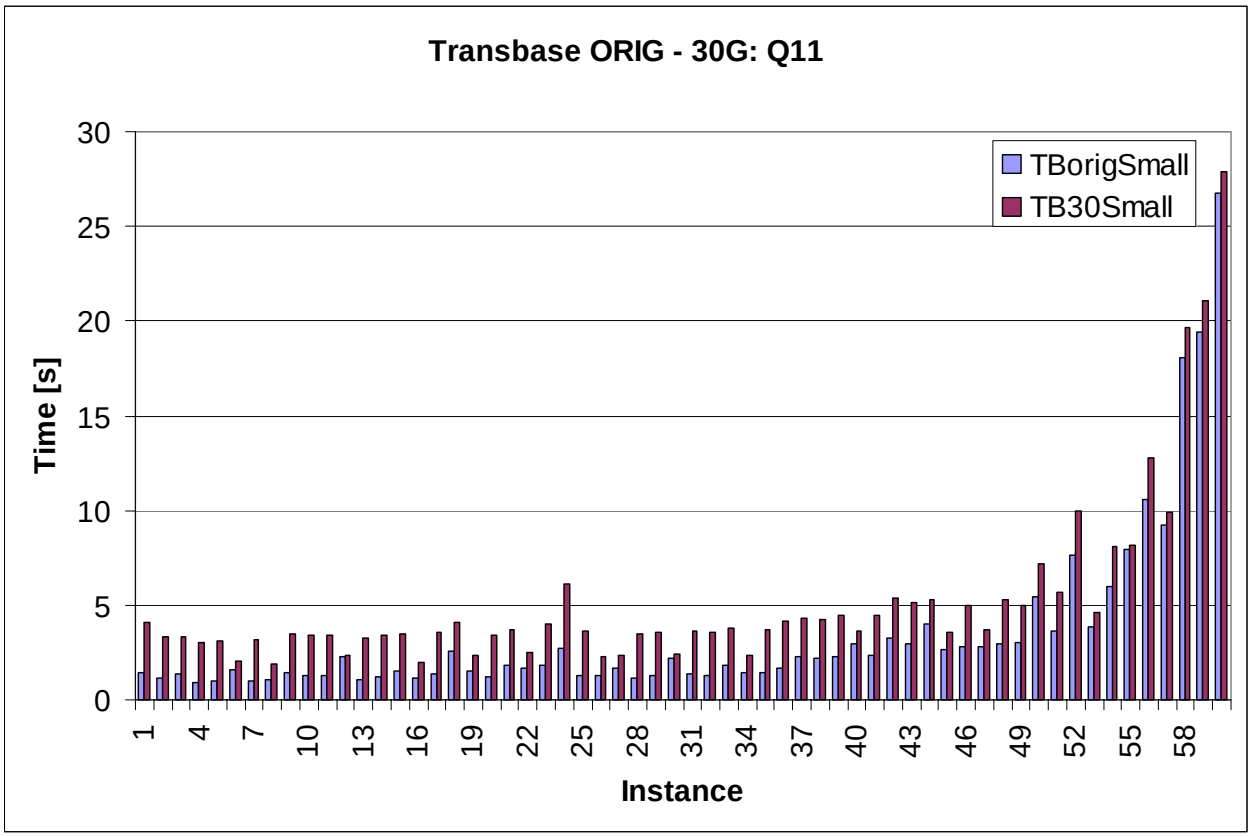

Transbase ORIG - 30G: Q6
Time [s]
Instance
TBorigSmall
TB30Small
0
50
100
150
200
250
300
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

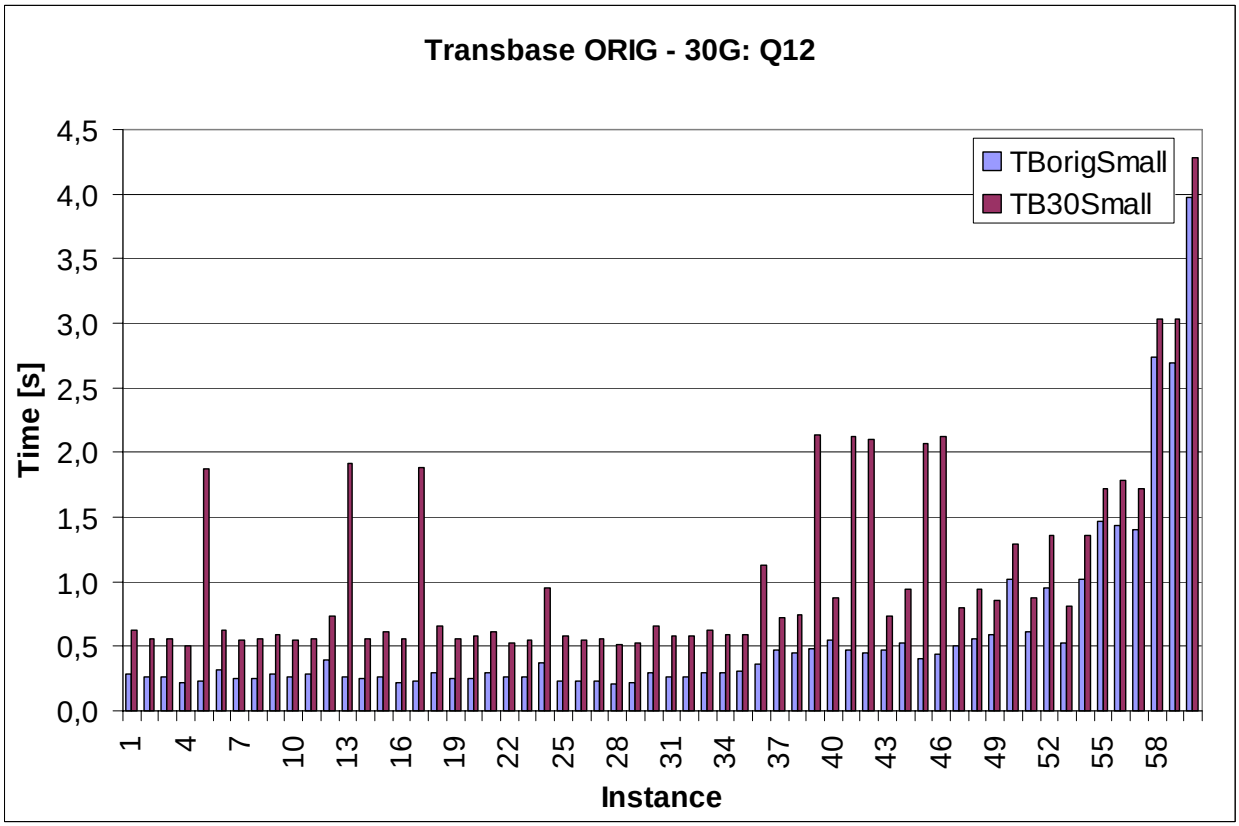

Transbase ORIG - 30G: Q7
Time [s]
Instance
TBorigSmall
TB30Small
0
1
2
3
4
5
6
7
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

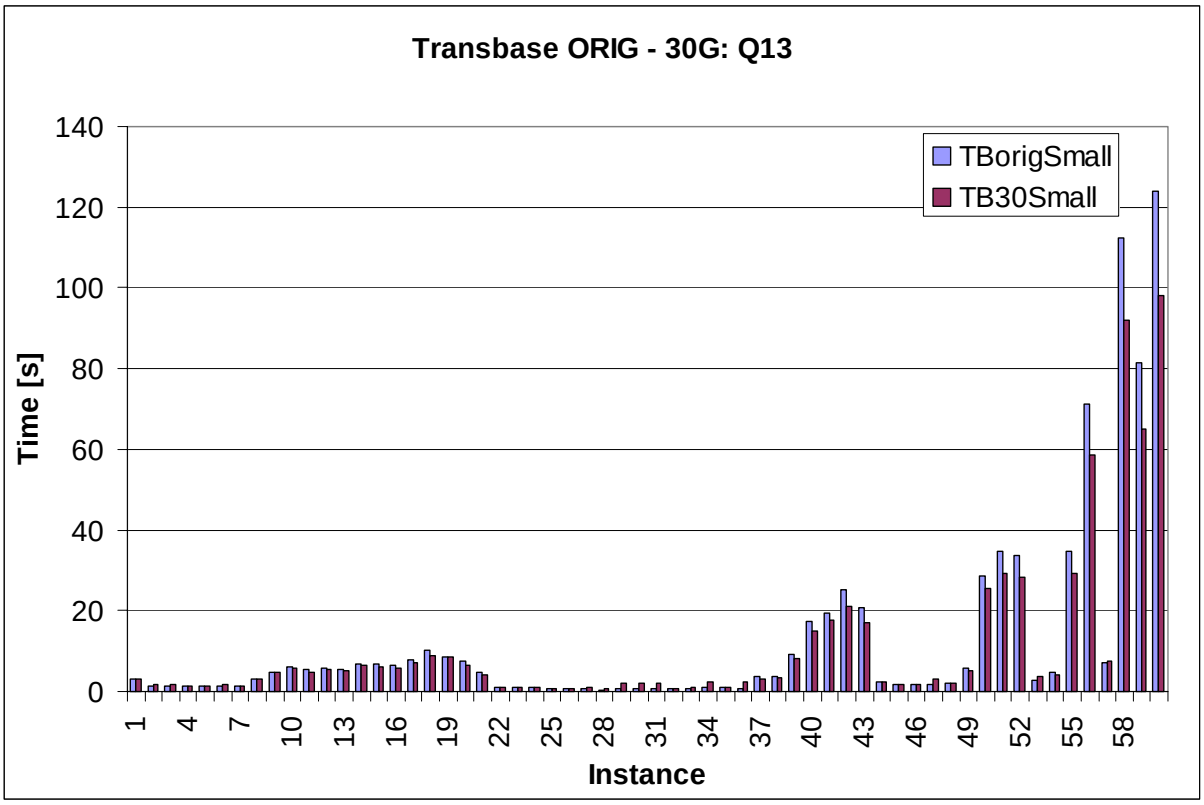

Transbase ORIG - 30G: Q8
Time [s]
Instance
TBorigSmall
TB30Small
0
200
400
600
800
1000
1200
1400
1600
1800
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57


Transbase ORIG - 30G: Q9
Time [s]
Instance
TBorigSmall
TB30Small
0
20
40
60
80
100
120
140
160
180
200
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

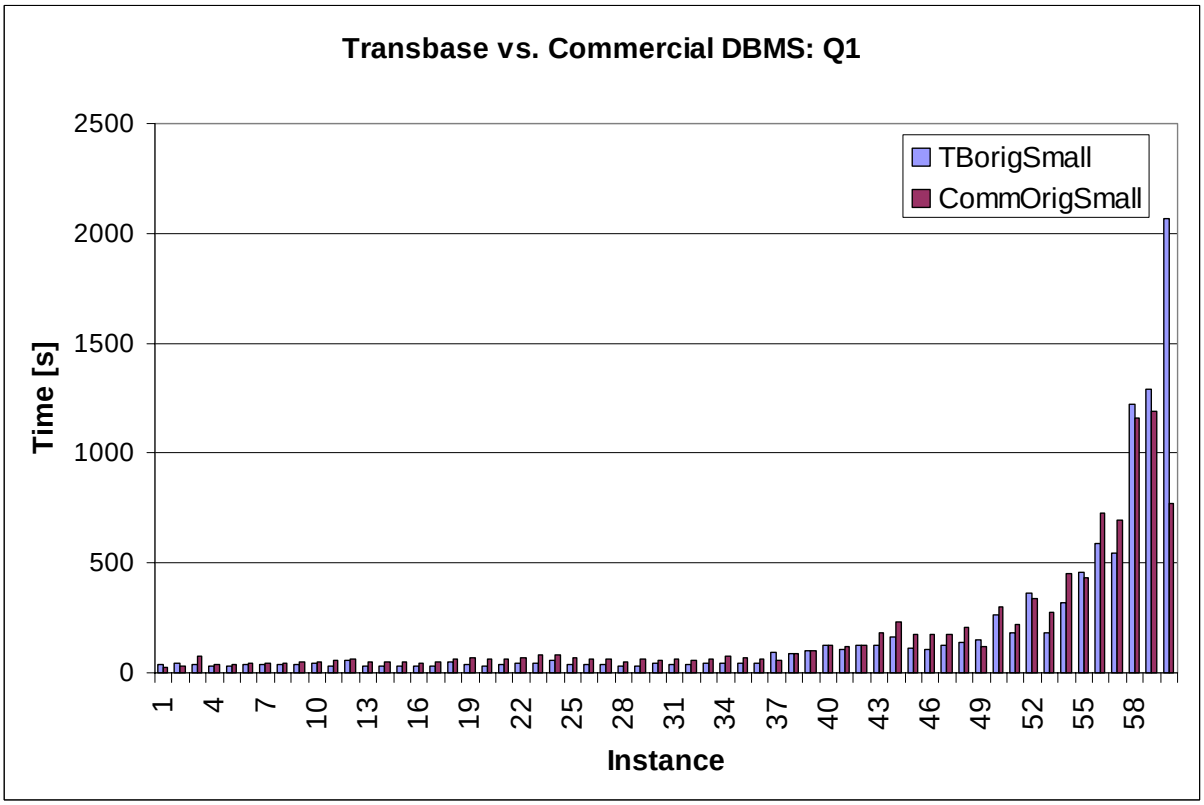

Transbase ORIG - 30G: Q10
Time [s]
Instance
TBorigSmall
TB30Small
0
1
2
3
4
5
6
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

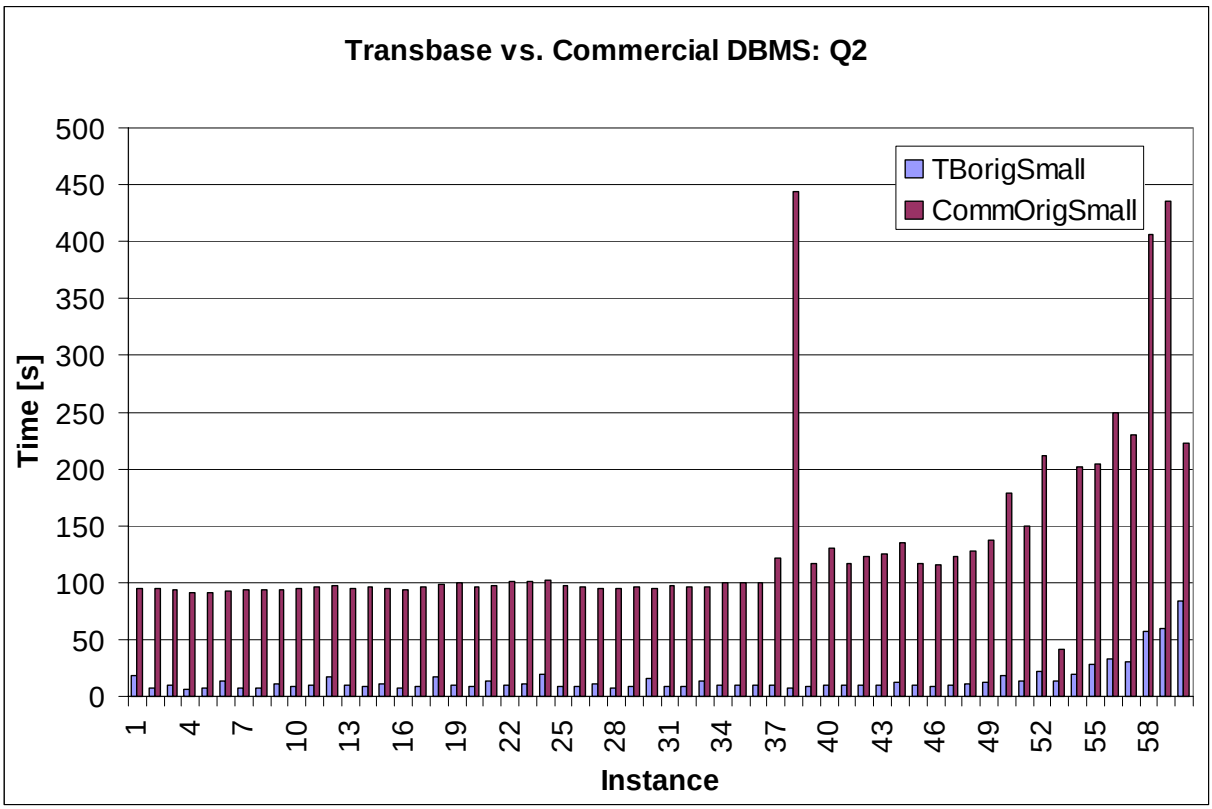

Transbase ORIG - 30G: Q11
Time [s]
Instance
TBorigSmall
TB30Small
0
5
10
15
20
25
30
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

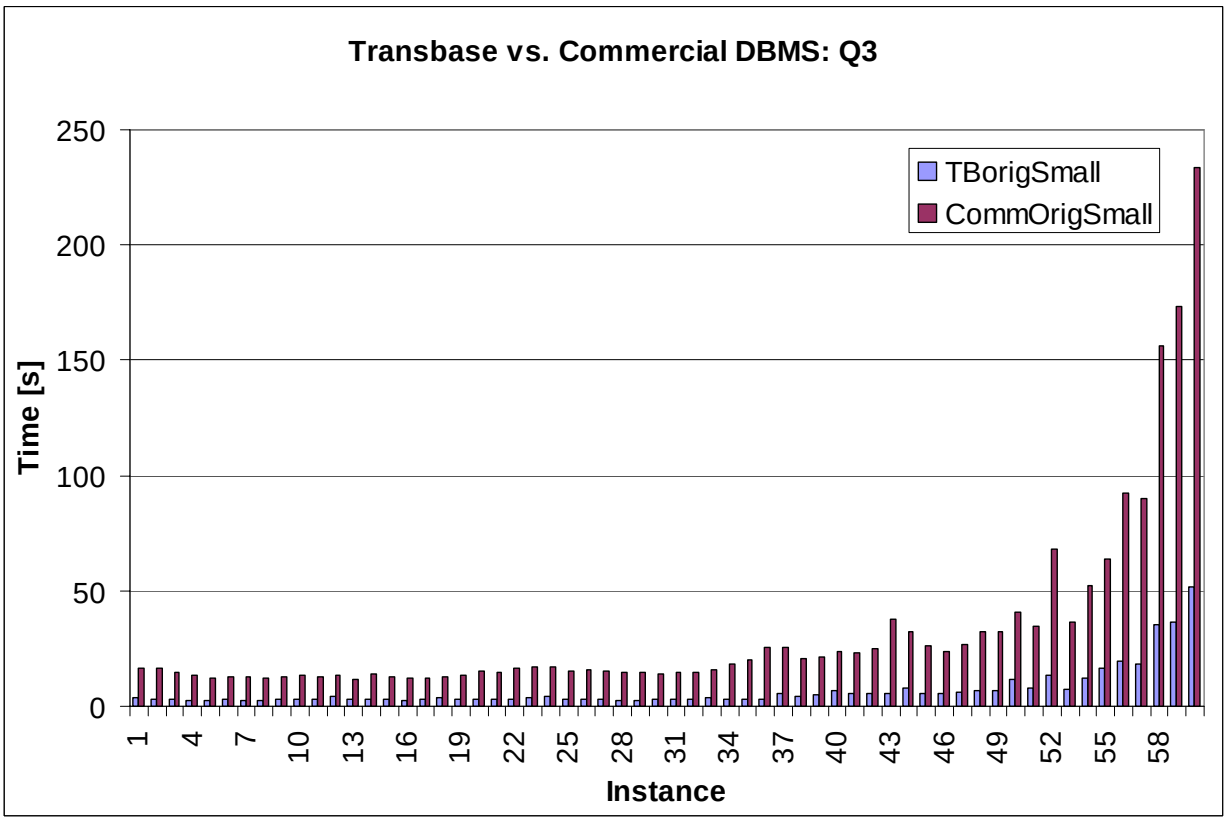

Transbase ORIG - 30G: Q12
Time [s]
Instance
TBorigSmall
TB30Small
0,0
0,5
1,0
1,5
2,0
2,5
3,0
3,5
4,0
4,5
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

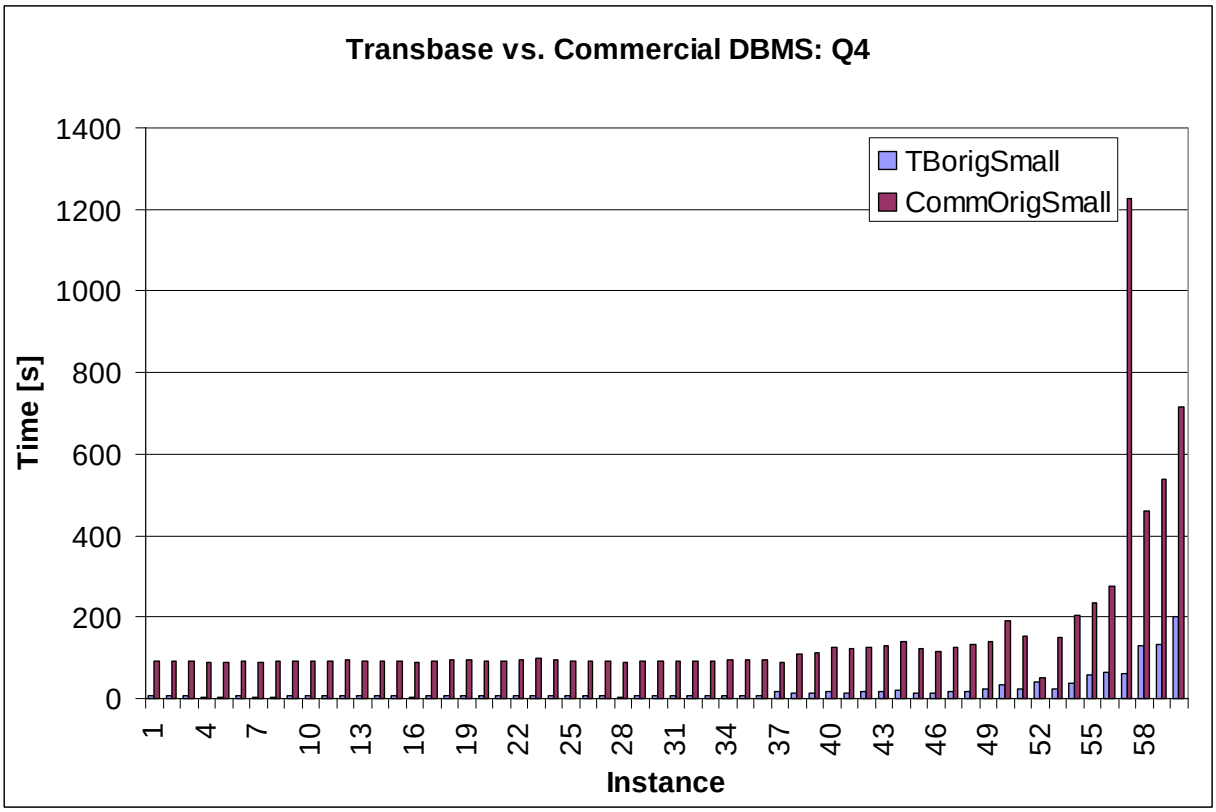


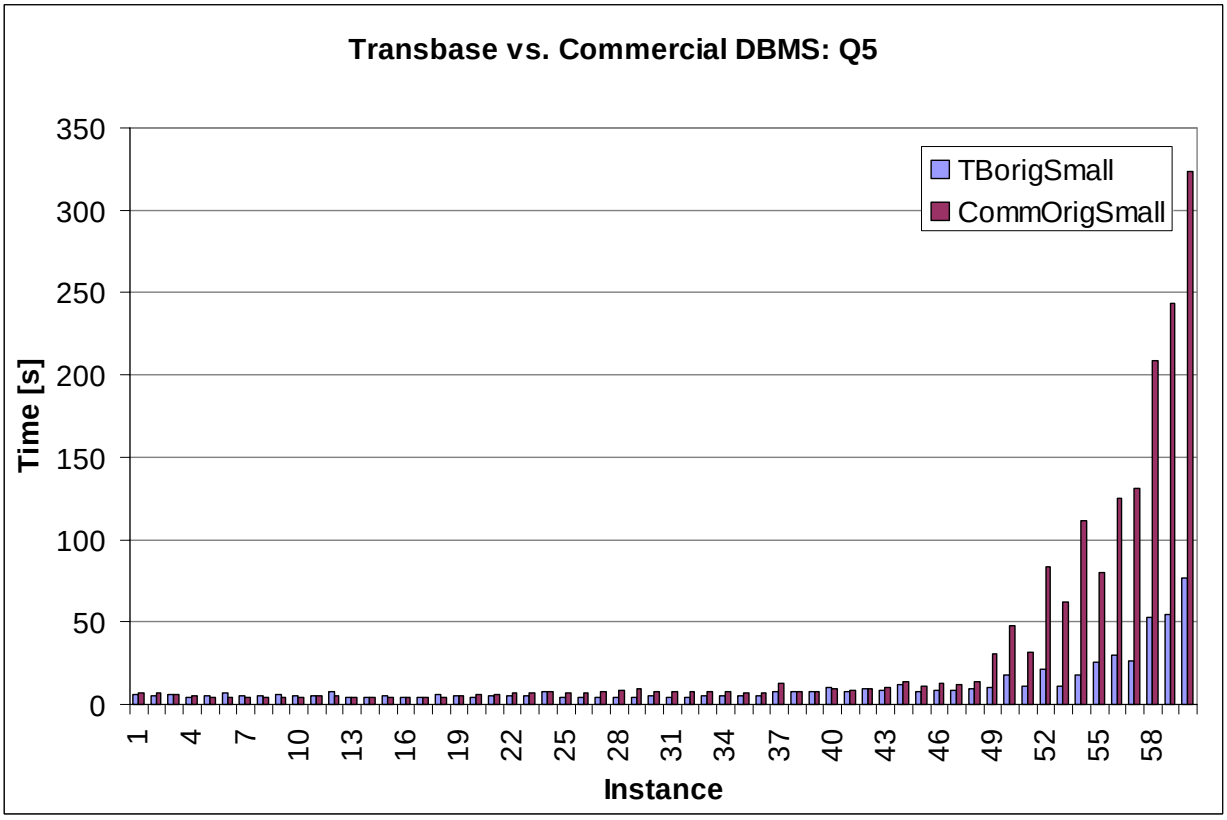
Comparison DBMS: Transbase® vs. Commercial DBMS

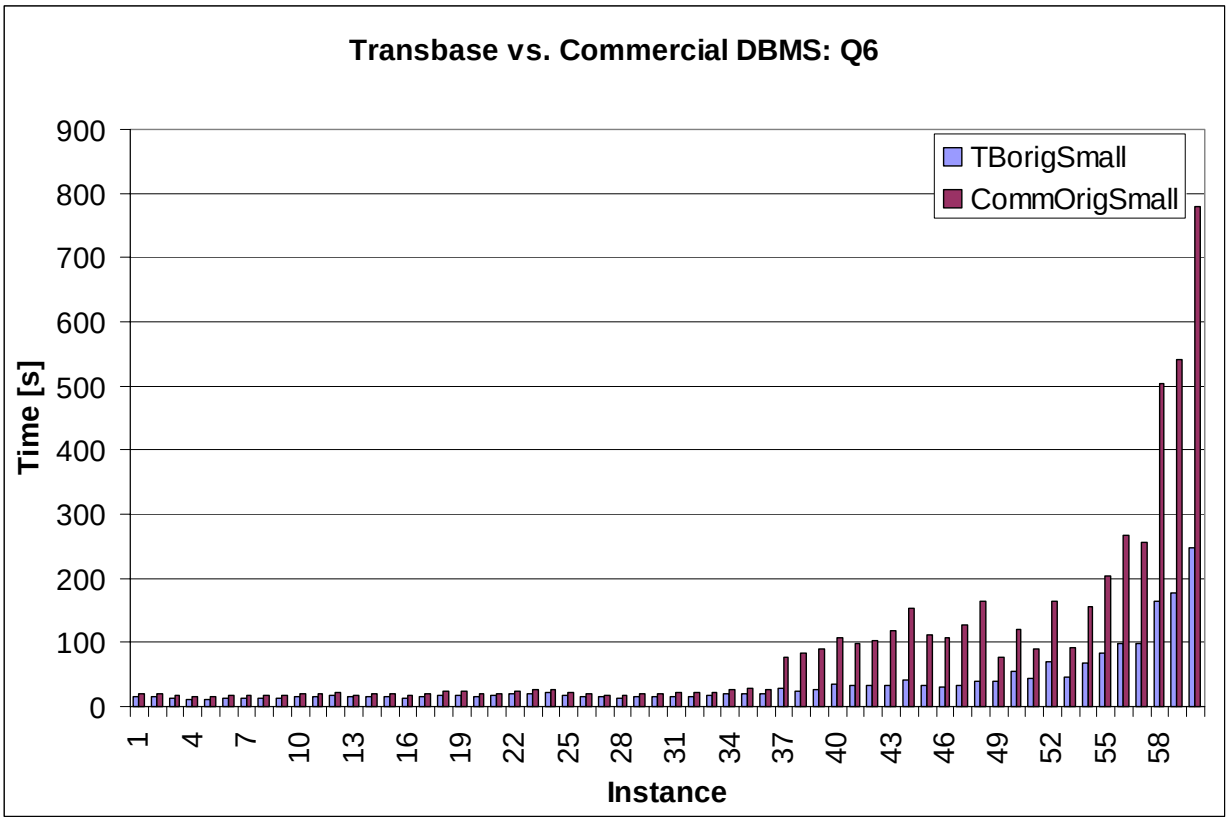


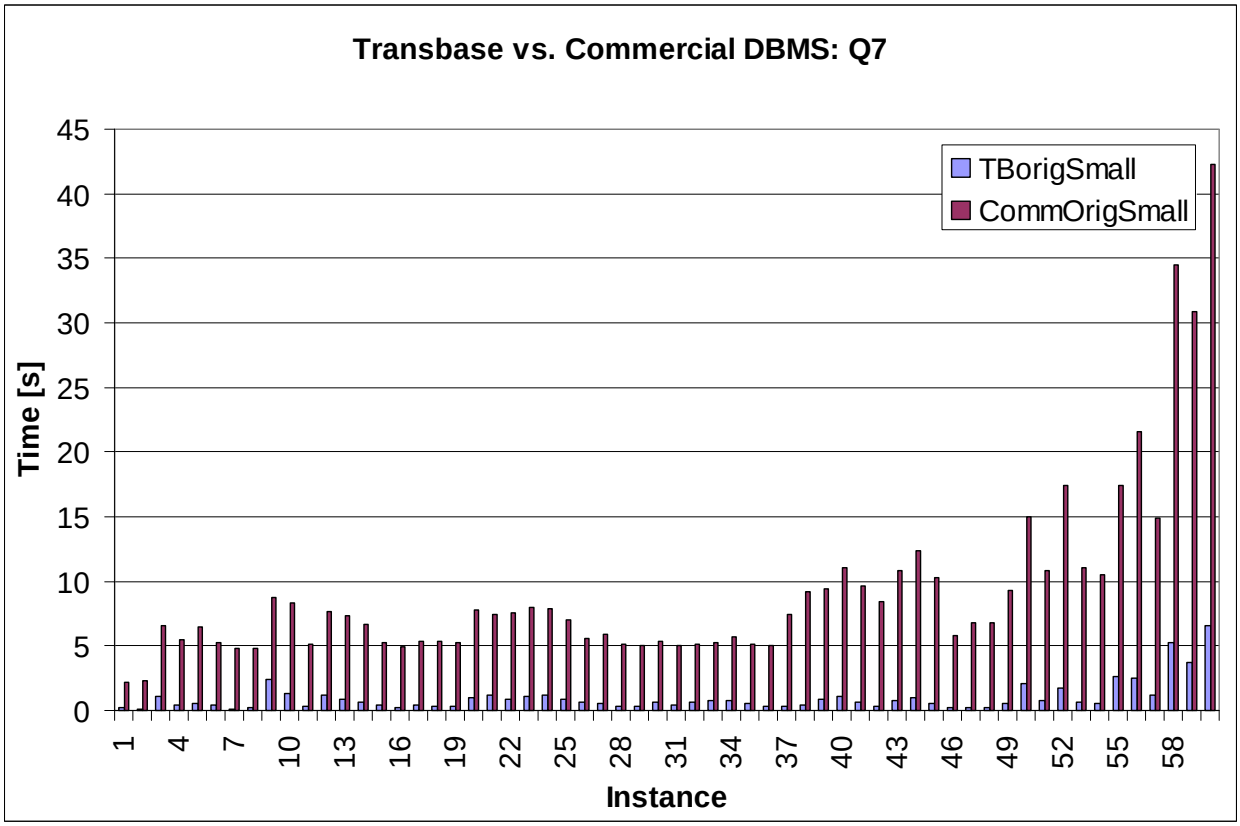

Transbase vs. Commercial DBMS: Q2
Time [s]
Instance
TBorigSmall
CommOrigSmall
0
50
100
150
200
250
300
350
400
450
500
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

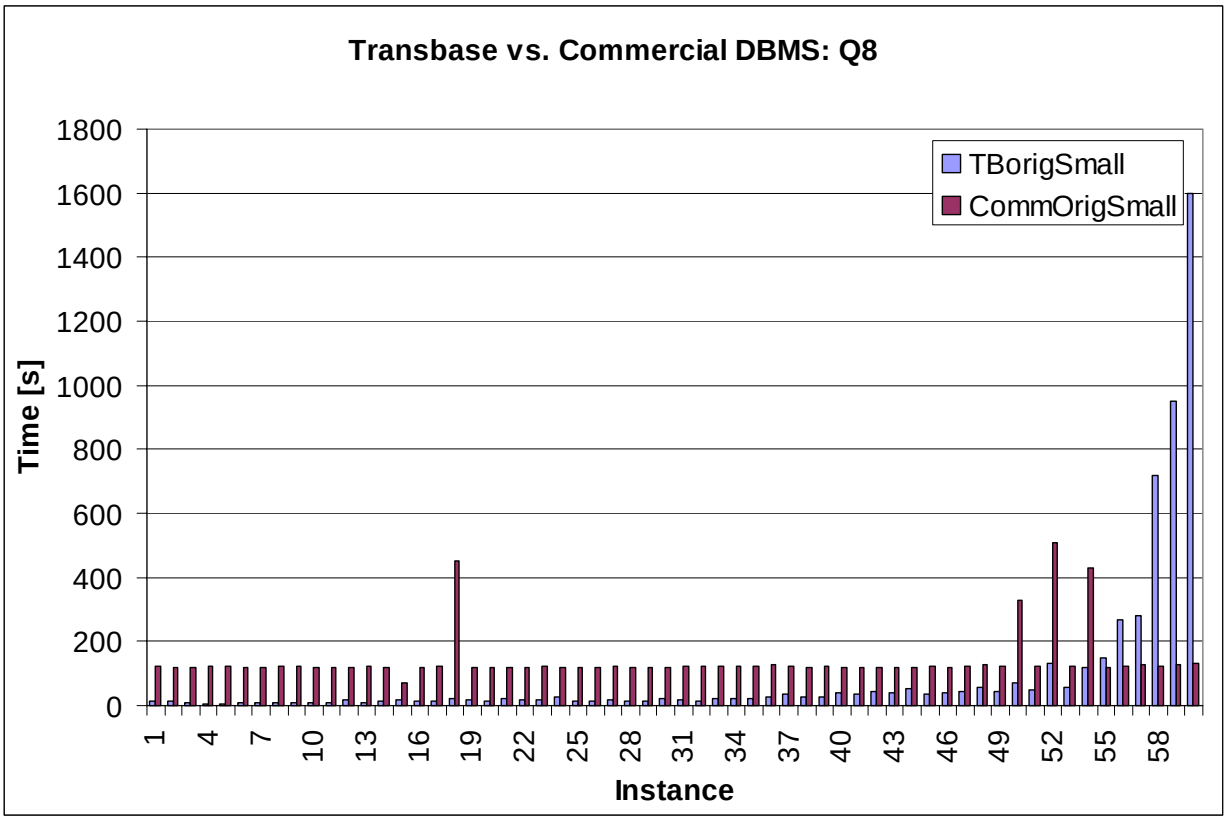

Transbase vs. Commercial DBMS: Q3
Time [s]
Instance
TBorigSmall
CommOrigSmall
0
50
100
150
200
250
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58

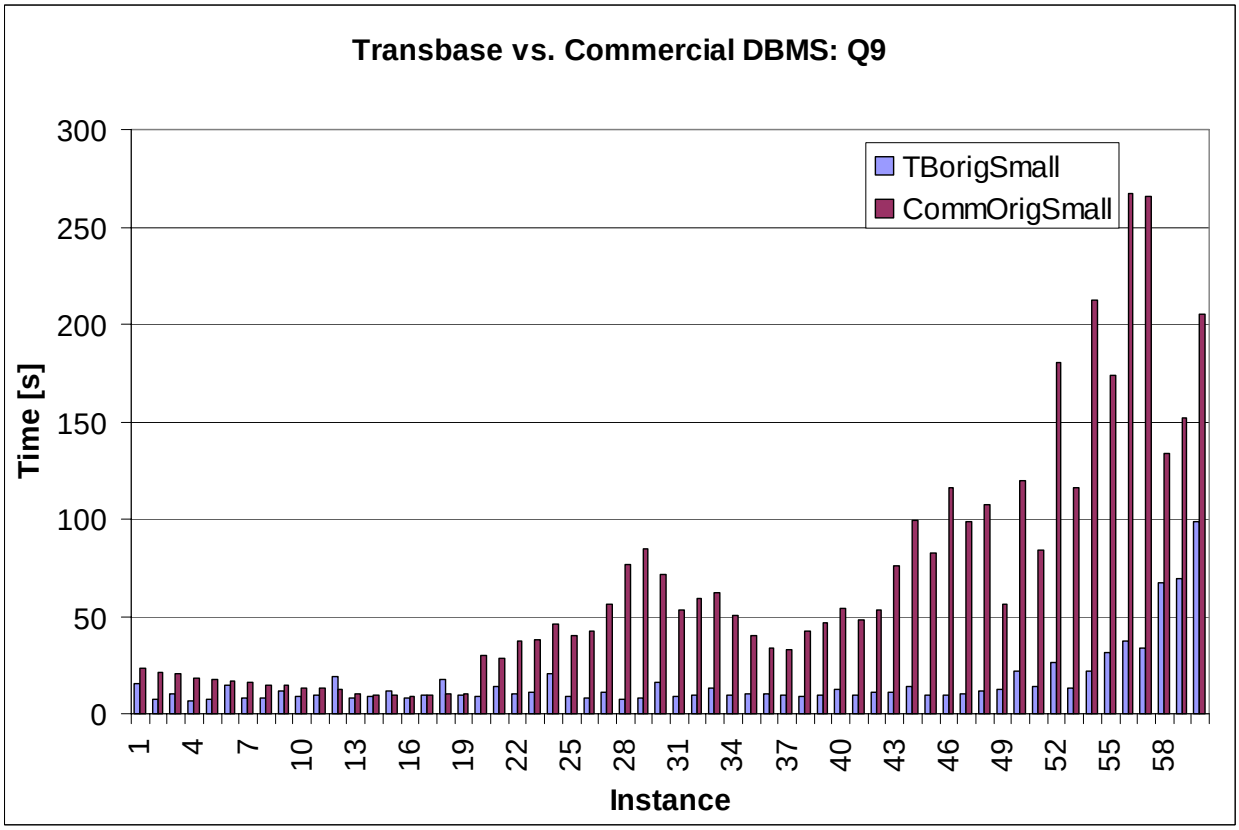

Transbase vs. Commercial DBMS: Q4
Time [s]
1400
1200
1000
800
600
400
200
0
TBorigSmall
CommOrigSmall
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance

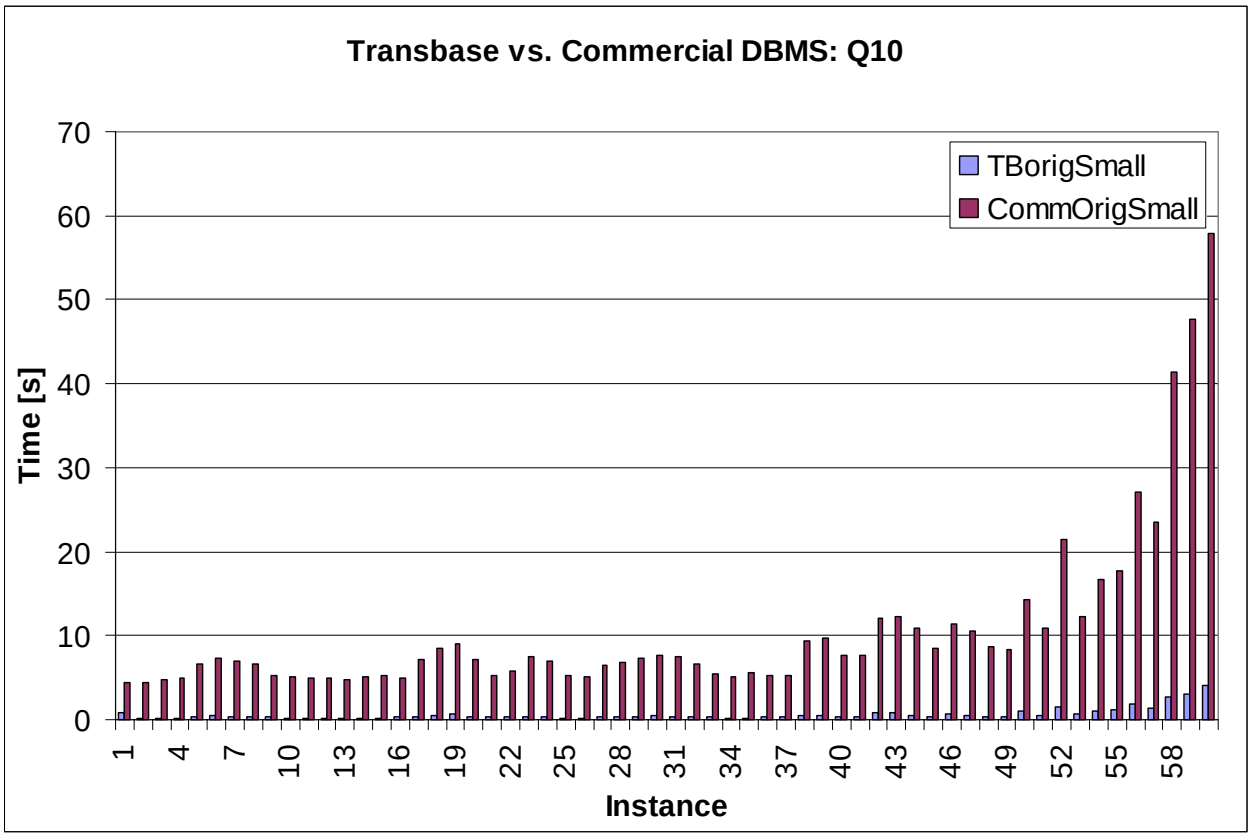

Transbase vs. Commercial DBMS: Q5
Time [s]
350
300
250
200
150
100
50
0
TBorigSmall
CommOrigSmall
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance

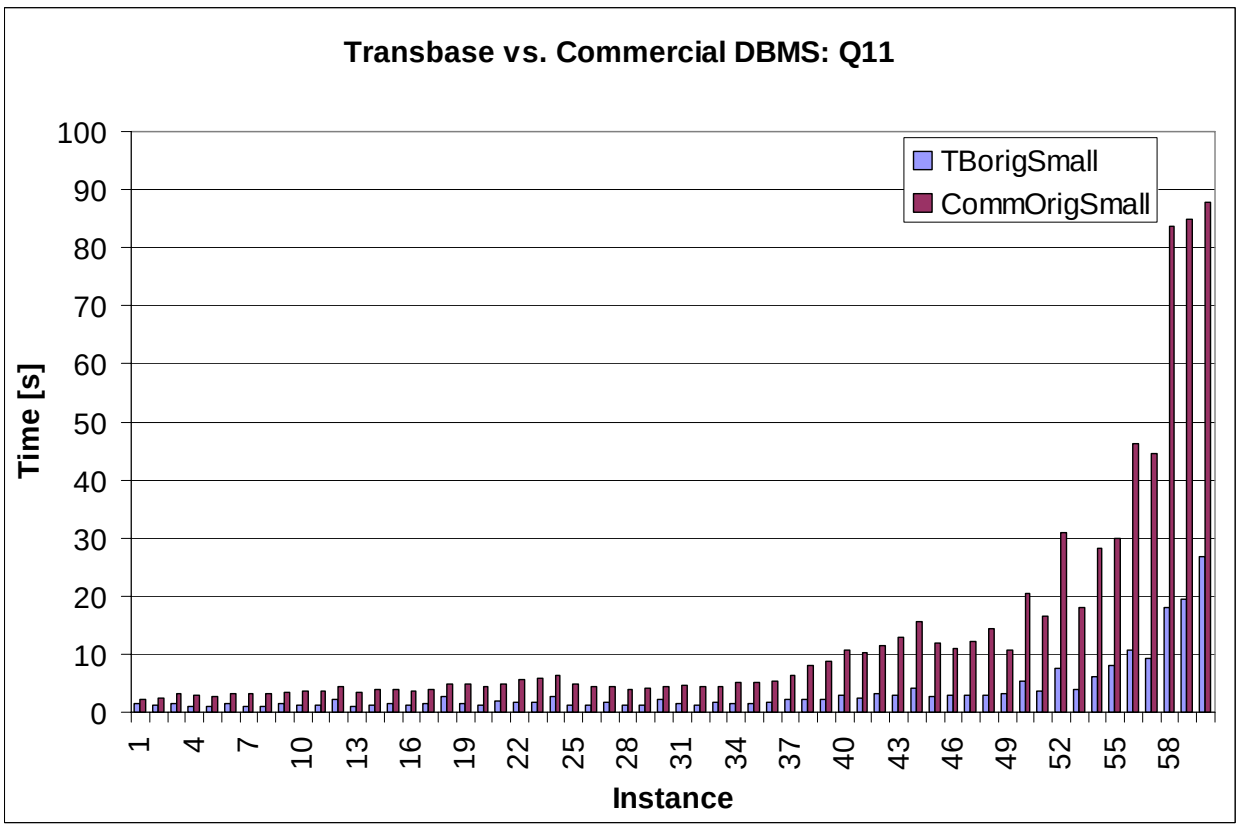

Transbase vs. Commercial DBMS: Q6
Time [s]
900
800
700
600
500
400
300
200
100
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance
TBorigSmall
CommOrigSmall

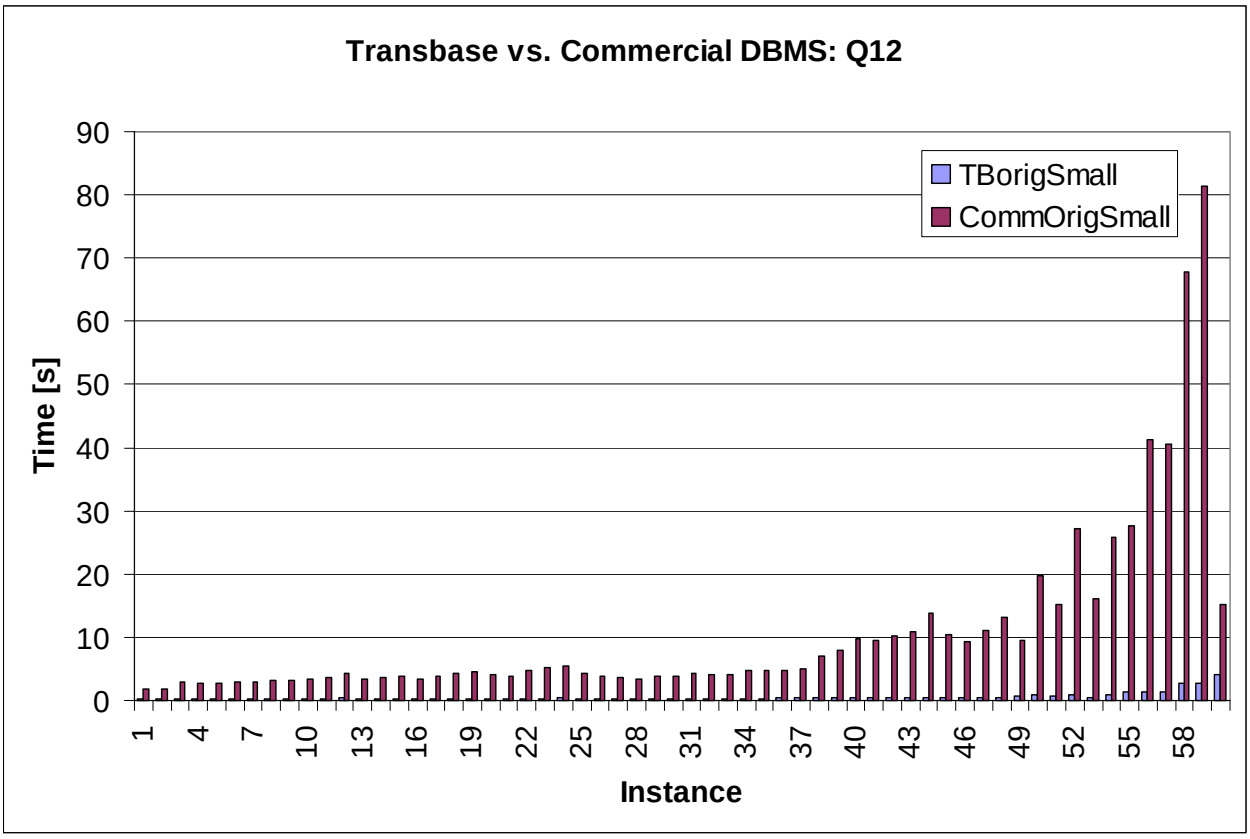

Transbase vs. Commercial DBMS: Q7
Time [s]
45
40
35
30
25
20
15
10
5
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance
TBorigSmall
CommOrigSmall

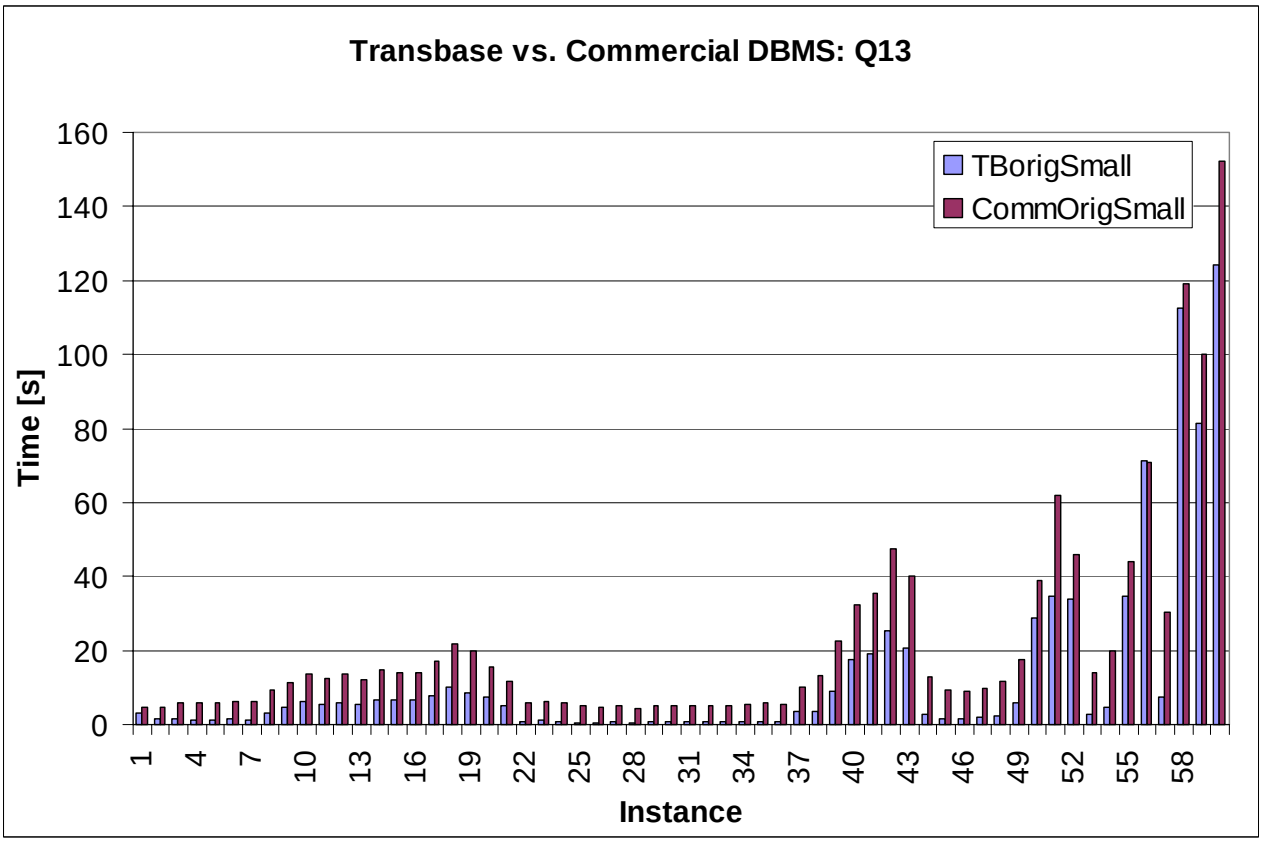

Transbase vs. Commercial DBMS: Q8
Time [s]
1800
1600
1400
1200
1000
800
600
400
200
0
TBorigSmall
CommOrigSmall
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance

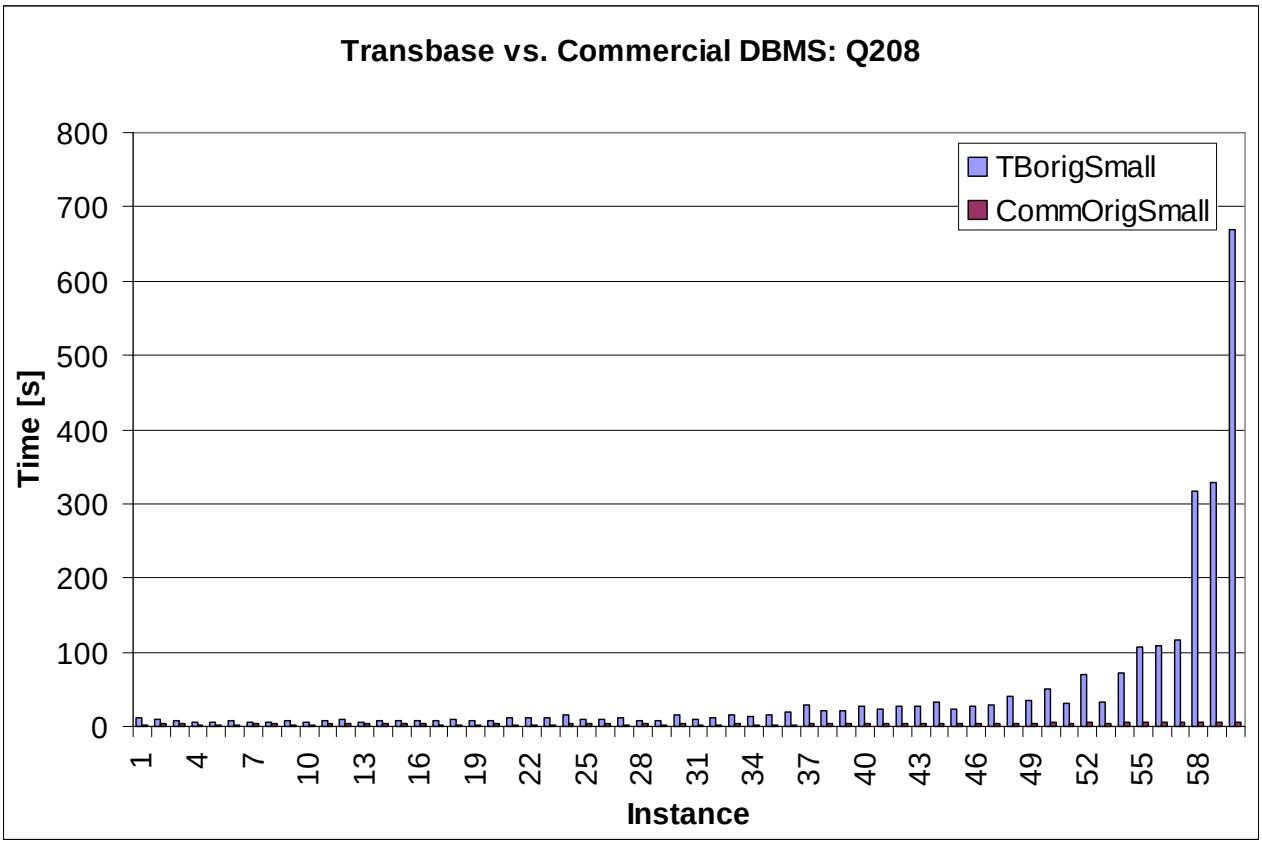

Transbase vs. Commercial DBMS: Q9
Time [s]
300
250
200
150
100
50
0
TBorigSmall
CommOrigSmall
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance


Transbase vs. Commercial DBMS: Q10
Time [s]
Instance
TBorigSmall
CommOrigSmall
0
10
20
30
40
50
60
70
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58


Transbase vs. Commercial DBMS: Q11
Time [s]
Instance
TBorigSmall
CommOrigSmall
0
10
20
30
40
50
60
70
80
90
100
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58


Transbase vs. Commercial DBMS: Q12
Time [s]
Instance
TBorigSmall
CommOrigSmall
0
10
20
30
40
50
60
70
80
90
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58


Transbase vs. Commercial DBMS: Q13
Time [s]
Instance
TBorigSmall
CommOrigSmall
0
20
40
60
80
100
120
140
160
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58


Transbase vs. Commercial DBMS: Q208
Time [s]
800
700
600
500
400
300
200
100
0
TBorigSmall
CommOrigSmall
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
Instance

Appendix F: Query Templates of APB Benchmark

Template Q01

```
select
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME,
       chanlevel.$CHAN_GIVEN_LEVEL_FNAME,
       timelevel.$TIME_CHILD_LEVEL_FNAME,
       sum (a.unitssold) AS Units,
       sum (a.dollarsales) AS Dollars,
       case when (sum(a.unitssold) <> 0) then sum (a.dollarsales) /
       sum (a.unitssold) else 0 end AS AvgSellingPrice
from
       fact_actvars a, dim_product prodlevel, dim_time timelevel,
       dim_customer custlevel, dim_channel chanlevel
where
           a.channel_level = '$CHANMEMBER'
       and a.product_level = prodlevel.code_level
       and a.time_level =  timelevel.month_level
       and a.customer_level = custlevel.store_level
       and a.channel_level = chanlevel.base_level
       and prodlevel.$PROD_CHILD_LEVEL_FNAME in (
         select member from dim_product_tree where parent = '$PRODMEMBER'
       )
       and timelevel.$TIME_CHILD_LEVEL_FNAME in (
         select member from dim_time_tree where parent = '$TIMEMEMBER'
       )
       and custlevel.$CUST_CHILD_LEVEL_FNAME in (
         select member from dim_customer_tree where parent = '$CUSTMEMBER'
       )
group by
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       chanlevel.$CHAN_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME,
       timelevel.$TIME_CHILD_LEVEL_FNAME
order by
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME,
       timelevel.$TIME_CHILD_LEVEL_FNAME
```

Template Q02

Q02-P1:

```
select
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       'Channel top',
       timelevel.$TIME_GIVEN_LEVEL_FNAME,
       sum (a.unitssold) AS Units,
       sum (a.dollarsales) as DollarSales,
       sum (a.dollarcost) as DollarCost,
       sum (a.dollarsales) - sum (a.dollarcost) as Margin,
       case when (sum(a.dollarsales) <> 0) then (sum (a.dollarsales) -
       sum (a.dollarcost)) / sum (a.dollarsales) else 0 end as MarginPct
from
       fact_actvars a, dim_product prodlevel, dim_time timelevel,
       dim_customer custlevel
where
           a.product_level = prodlevel.code_level
       and a.time_level =  timelevel.month_level
       and a.customer_level = custlevel.store_level
       and timelevel.$TIME_GIVEN_LEVEL_FNAME = '$TIMEMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
```

```
       and prodlevel.$PROD_CHILD_LEVEL_FNAME in (
         select member from dim_product_tree where parent = '$PRODMEMBER'
       )
group by
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME
order by
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME
```

Q02-P2:

```
select
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME,
       sum (a.val) as SPC
from
       fact_stdprodcost a, dim_product prodlevel, dim_time timelevel
where
           a.product_level = prodlevel.code_level
       and a.time_level =  timelevel.month_level
       and a.scenario = 'Actual'
       and timelevel.$TIME_GIVEN_LEVEL_FNAME = '$TIMEMEMBER'
       and prodlevel.$PROD_CHILD_LEVEL_FNAME in (
         select member from dim_product_tree where parent = '$PRODMEMBER'
       )
group by
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME
order by
       $PROD_CHILD_LEVEL_FNAME,
       $TIME_GIVEN_LEVEL_FNAME
```

Q02-P3:

```
select
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME,
       sum (a.val) as SSC
from
       fact_stdshipcost a, dim_time timelevel, dim_customer custlevel
where
           a.time_level =  timelevel.month_level
       and a.customer_level = custlevel.store_level
       and a.scenario = 'Actual'
       and timelevel.$TIME_GIVEN_LEVEL_FNAME = '$TIMEMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
group by
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME
order by
       $CUST_GIVEN_LEVEL_FNAME,
       $TIME_GIVEN_LEVEL_FNAME
```

Template Q03

Q03-P1:

```
select
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME,
       'Channel Top',
       timelevel.month_level,
       sum (a.unitssold) as Units,
       sum (a.dollarsales) as DollarSales
```

```
      , sum (a.dollarcost) as DollarCost
from
      fact_actvars a, dim_product prodlevel, dim_time timelevel,
      dim_customer custlevel
where
          a.product_level = prodlevel.code_level
      and a.time_level =  timelevel.month_level
      and a.customer_level = custlevel.store_level
      and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
      and custlevel.$CUST_CHILD_LEVEL_FNAME in (
        select member from dim_customer_tree where parent = '$CUSTMEMBER'
      )
group by
      prodlevel.$PROD_GIVEN_LEVEL_FNAME,
      custlevel.$CUST_CHILD_LEVEL_FNAME,
      timelevel.month_level
order by
      $PROD_GIVEN_LEVEL_FNAME,
      $CUST_CHILD_LEVEL_FNAME,
      month_level
```

Q03-P2:

```
select
      prodlevel.$PROD_GIVEN_LEVEL_FNAME,
      custlevel.$CUST_CHILD_LEVEL_FNAME,
      'Channel Top',
      sum (a.inv199501) as inv199501,
      sum (a.inv199502) as inv199502,
      sum (a.inv199503) as inv199503,
      sum (a.inv199504) as inv199504,
      sum (a.inv199505) as inv199505,
      sum (a.inv199506) as inv199506,
      sum (a.inv199507) as inv199507,
      sum (a.inv199508) as inv199508,
      sum (a.inv199509) as inv199509,
      sum (a.inv199510) as inv199510,
      sum (a.inv199511) as inv199511,
      sum (a.inv199512) as inv199512,
      sum (a.inv199601) as inv199601,
      sum (a.inv199602) as inv199602,
      sum (a.inv199603) as inv199603,
      sum (a.inv199604) as inv199604,
      sum (a.inv199605) as inv199605
from
      fact_histinventory a, dim_product prodlevel, dim_customer custlevel
where
          a.product_level = prodlevel.code_level
      and a.customer_level = custlevel.store_level
      and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
      and custlevel.$CUST_CHILD_LEVEL_FNAME in (
        select member from dim_customer_tree where parent = '$CUSTMEMBER'
      )
group by
      prodlevel.$PROD_GIVEN_LEVEL_FNAME,
      custlevel.$CUST_CHILD_LEVEL_FNAME
order by
      $PROD_GIVEN_LEVEL_FNAME,
      $CUST_CHILD_LEVEL_FNAME
```

Q03-P3:

```
select
      prodlevel.$PROD_GIVEN_LEVEL_FNAME,
      custlevel.$CUST_CHILD_LEVEL_FNAME,
      'Channel Top',
      sum (a.inv199606) as inv199605
from
```

```
       fact_currinventory a, dim_product prodlevel, dim_customer custlevel
where
           a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_CHILD_LEVEL_FNAME in (
         select member from dim_customer_tree where parent = '$CUSTMEMBER'
       )
group by
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME
order by
       $PROD_GIVEN_LEVEL_FNAME,
       $CUST_CHILD_LEVEL_FNAME
```

Template Q05

```
select
       'Product Top',
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.month_level,
       sum (a.unitssold)                         aS UnitsSold,
       sum (a.dollarsales)                       as DollarSales,
       case when (sum(a.unitssold) <> 0) then sum (a.dollarsales) /
       sum (a.unitssold) else 0 end as AvgSellingPrice
       ,sum (a.dollarcost)                       as DollarCost,
       sum (a.dollarsales) - sum (a.dollarcost) as Margin
from
       fact_planvars a, dim_time timelevel, dim_customer custlevel
where
           a.time_level = timelevel.month_level
       and a.customer_level = custlevel.store_level
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
group by
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.month_level
order by
       custlevel.$CUST_GIVEN_LEVEL_FNAME,
       timelevel.month_level
```

Template Q06

```
select
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       'Customer Top',
       timelevel.quarter_level,
       sum (a.unitssold)                         as UnitsSold,
       sum (a.dollarsales)                       as DollarSales,
       case when (sum(a.unitssold) <> 0) then sum (a.dollarsales) /
       sum (a.unitssold)  else 0 end             as AvgSellingPrice
       , sum (a.dollarcost)                      as DollarCost,
       sum (a.dollarsales) - sum (a.dollarcost) as MarginDollars
from
       fact_planvars a, dim_time timelevel, dim_product prodlevel
where
           a.time_level = timelevel.month_level
       and    a.product_level = prodlevel.code_level
       and timelevel.quarter_level in (
              select member from dim_time_tree where parent = '1996'
       )
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
group by
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       timelevel.quarter_level
order by
       $PROD_GIVEN_LEVEL_FNAME,
       quarter_level
```

Template Q07

```
select
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME,
       sum (a.unitssold)                          as SumUnits,
       sum (a.dollarsales)                        as SumDollars,
       case when (sum(a.unitssold) <> 0) then sum (a.dollarsales) /
       sum (a.unitssold)  else 0 end              as Price,
       sum (a.dollarcost)                         as SumCost,
       sum (a.dollarsales) - sum (a.dollarcost) as Margin
from
       fcstmonthstoreproduct a, dim_time timelevel, dim_product prodlevel,
       dim_customer custlevel
where
           a.time_level = timelevel.month_level
       and    a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and timelevel.$TIME_GIVEN_LEVEL_FNAME = '$TIMEMEMBER'
       and prodlevel.$PROD_CHILD_LEVEL_FNAME in (
         select member from dim_product_tree where parent = '$PRODMEMBER'
       )
       and custlevel.$CUST_CHILD_LEVEL_FNAME in (
         select member from dim_customer_tree where parent = '$CUSTMEMBER'
       )
group by
       prodlevel.$PROD_CHILD_LEVEL_FNAME,
       custlevel.$CUST_CHILD_LEVEL_FNAME,
       timelevel.$TIME_GIVEN_LEVEL_FNAME
order by
       $PROD_CHILD_LEVEL_FNAME,
       $CUST_CHILD_LEVEL_FNAME,
       $TIME_GIVEN_LEVEL_FNAME
```

Template Q08

Q08-Prep:

```
delete FROM q8a_tmp
;
insert into q8a_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level = '199606'
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
insert into q8a_tmp (
       product,
       customer,
       timeper,
```

```
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606YTD',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level in (
              '199601', '199602', '199603', '199604', '199605', '199606'
           )
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
delete FROM q8b_tmp
;
insert into q8b_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_planvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level = '199606'
       and a.product_level = prodlevel.code_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and a.customer_level = custlevel.store_level
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
insert into q8b_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606YTD',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_planvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level in (
              '199601', '199602', '199603', '199604', '199605', '199606'
              )
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
```

```
;

delete FROM q8c_tmp
;
insert into q8c_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199506',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level = '199506'
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
insert into q8c_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199506YTD',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level in (
                    '199501', '199502', '199503', '199504', '199505', '199506'
              )
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
```

Q08-P1:

```
SELECT
       a.product,
       a.customer,
       'actual',
       a.timeper,
       a.dollarsales
FROM
       q8a_tmp a
WHERE
       a.timeper in ('199606', '199606YTD')
UNION
SELECT
       a.product,
       a.customer,
```

```
       'budget',
       a.timeper,
       a.dollarsales
FROM
       q8b_tmp a
WHERE
       a.timeper in ('199606', '199606YTD')
```

Q08-P2:

```
SELECT
       a.product,
       a.customer,
       a.timeper,
       (a.dollarsales - b.dollarsales) AS ValDiff,
       case when (b.dollarsales <> 0) then (a.dollarsales / b.dollarsales)
       - 1 else 0 end AS PctDiff
FROM
       q8a_tmp a, q8b_tmp b
WHERE
           a.timeper = b.timeper
       and a.timeper in ('199606', '199606YTD')
       and a.product = b.product
       and a.customer = b.customer
```

Q08-P3:

```
SELECT
       a.product,
       a.customer,
       '199606YTY',
       (a.dollarsales - b.dollarsales) AS LastYearValDiff,
       case when (b.dollarsales <> 0) then (a.dollarsales / b.dollarsales)
       - 1 else 0 end AS LastYearPctDiff
FROM
       q8a_tmp a, q8c_tmp b
WHERE
           a.timeper = '199606'
       and b.timeper = '199506'
       and a.product = b.product
       and a.customer = b.customer
UNION
SELECT
       a.product,
       a.customer,
       '199606YTDYTY',
       (a.dollarsales - b.dollarsales) AS LastYearValDiff,
       case when (b.dollarsales <> 0) then (a.dollarsales / b.dollarsales)
       - 1 else 0 end AS LastYearPctDiff
FROM
       q8a_tmp a, q8c_tmp b
WHERE
           a.timeper = '199606YTD'
       and b.timeper = '199506YTD'
       and a.product = b.product
       and a.customer = b.customer
```

Template Q09

Q09-Prep:

```
delete FROM q9a_tmp
;
insert into q9a_tmp (
       product,
       customer,
       timeper,
```

```
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level = '199606'
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
insert into q9a_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606YTD',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level in (
              '199601', '199602', '199603', '199604', '199605', '199606'
              )
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
delete FROM q9b_tmp
;
insert into q9b_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606',
       sum (a.dollarsales) AS dollarsales
FROM
       fcstmonthstoreproduct a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level = '199606'
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
insert into q9b_tmp (
```

```
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199606YTD',
       sum (a.dollarsales) AS dollarsales
FROM
       fcstmonthstoreproduct a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level in (
              '199601', '199602', '199603', '199604', '199605', '199606'
              )
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
delete FROM q9c_tmp
;
insert into q9c_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199605',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level = '199605'
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
insert into q9c_tmp (
       product,
       customer,
       timeper,
       dollarsales
)
SELECT
       prodlevel.$PROD_GIVEN_LEVEL_FNAME AS product,
       custlevel.$CUST_GIVEN_LEVEL_FNAME AS customer,
       '199605YTD',
       sum (a.dollarsales) AS dollarsales
FROM
       fact_actvars a, dim_product prodlevel, dim_customer custlevel
WHERE
           a.time_level in (
                    '199601', '199602', '199603', '199604', '199605'
              )
       and a.product_level = prodlevel.code_level
       and a.customer_level = custlevel.store_level
       and prodlevel.$PROD_GIVEN_LEVEL_FNAME = '$PRODMEMBER'
       and custlevel.$CUST_GIVEN_LEVEL_FNAME = '$CUSTMEMBER'
```

```
GROUP BY
       prodlevel.$PROD_GIVEN_LEVEL_FNAME,
       custlevel.$CUST_GIVEN_LEVEL_FNAME
;
```

Q09-P1:

```
SELECT
       a.product,
       a.customer,
       'actual',
       a.timeper,
       a.dollarsales
FROM
       q9a_tmp a
WHERE
       a.timeper in ('199606', '199606YTD')
UNION
SELECT
       a.product,
       a.customer,
       'forcst',
       a.timeper,
       a.dollarsales
FROM
       q9b_tmp a
WHERE
       a.timeper in ('199606', '199606YTD')
```

Q09-P2:

```
SELECT
       a.product,
       a.customer,
       a.timeper,
       (a.dollarsales - b.dollarsales) AS ValDiff,
       case when (b.dollarsales <> 0) then (a.dollarsales / b.dollarsales)
       - 1 else 0 end AS PctDiff
FROM
       q9a_tmp a, q9b_tmp b
WHERE
           a.timeper = b.timeper
       and a.timeper in ('199606', '199606YTD')
       and a.product = b.product
       and a.customer = b.customer
```

Q09-P3:

```
SELECT
       a.product,
       a.customer,
       '199606PTP',
       (a.dollarsales - b.dollarsales) AS LastPeriodValDiff,
       case when (b.dollarsales <> 0) then (a.dollarsales / b.dollarsales)
       - 1 else 0 end AS LastPeriodPctDiff
FROM
       q9a_tmp a, q9c_tmp b
WHERE
           a.timeper = '199606'
       and b.timeper = '199605'
       and a.product = b.product
       and a.customer = b.customer
UNION
SELECT
       a.product,
       a.customer,
       '199606YTDPTP',
       (a.dollarsales - b.dollarsales) AS LastPeriodValDiff,
```

```
       case when (b.dollarsales <> 0) then (a.dollarsales / b.dollarsales)
       - 1 else 0 end AS LastPeriodPctDiff
FROM
       q9a_tmp a, q9c_tmp b
WHERE
           a.timeper = '199606YTD'
       and b.timeper = '199605YTD'
       and a.product = b.product
       and a.customer = b.customer
```

Index

List of Figures

List of Definitions

List of Examples