Extending Practical Pre-Aggregation in On-Line Analytical Processing

Torben Bach Pedersen† Christian S. Jensen‡ Curtis E. Dyreson*

† Center for Health Information Services, Kommunedata
P. O. Pedersens Vej 2, DK-8200 Århus N, Denmark, tbp@kmd.dk

‡ Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK–9220 Aalborg Ø, Denmark, csj@cs.auc.dk

* Department of Computer Science, James Cook University
Townswille, QLD 4811, Australia, curtis@cs.jcu.edu.au

September 20, 1999

Abstract

On-Line Analytical Processing (OLAP) based on a dimensional view of data is being used increasingly in traditional business applications as well as in applications such as health care for the purpose of analyzing very large amounts of data. Pre-aggregation, the prior materialization of aggregate queries for later use, is an essential technique for ensuring adequate response time during data analysis. Full pre-aggregation, where all combinations of aggregates are materialized, is infeasible. Instead, modern OLAP systems adopt the *practical pre-aggregation* approach of materializing only select combinations of aggregates and then re-use these for efficiently computing other aggregates. However, this re-use of aggregates is contingent on the dimension hierarchies and the relationships between facts and dimensions satisfying stringent constraints. This severely limits the scope of the practical pre-aggregation approach. This paper significantly extends the scope of practical pre-aggregation to cover a much wider range of realistic situations. Specifically, algorithms are given that transform "irregular" dimension hierarchies and fact-dimension relationships, which often occur in real-world OLAP applications, into well-behaved structures that, when used by existing OLAP systems, enable practical pre-aggregation. The algorithms have low computational complexity and may be applied incrementally to reduce the cost of updating OLAP structures.

Contents

1 Introduction

On-Line Analytical Processing (OLAP) systems, which aim to ease the process of extracting useful information from large amounts of detailed transactional data, have gained widespread acceptance in traditional business applications as well as in new applications such as health care. These systems generally offer a dimensional view of data, in which measured values, termed facts, are characterized by descriptive values, drawn from a number of dimensions; and the values of a dimension are typically organized in a containment-type hierarchy. A prototypical query applies an aggregate function, such as average, to the facts characterized by specific values from the dimensions.

Fast response times are required from these systems, even for queries that aggregate large amounts of data. The perhaps most central technique used for meeting this requirement is termed *pre-aggregation*, where the results of aggregate queries are pre-computed and stored, i.e., materialized, for later use during query processing. Pre-aggregation has attracted substantial attention in the research community, where it has been investigated how to optimally use pre-aggregated data for query optimization [7, 3] and how to maintain the pre-aggregated data when base data is updated [19, 23]. Further, the latest versions of commercial RDBMS products offer query optimization based on pre-computed aggregates and automatic maintenance of the stored aggregates when base data is updated [29].

The fastest response times may be achieved when materializing aggregate results corresponding to all combinations of dimension values across all dimensions, termed *full* (or eager) pre-aggregation. However, the required storage space grows rapidly, to quickly become prohibitive, as the complexity of the application increases. This phenomenon is called *data explosion* [4, 26, 21] and occurs because the number of possible aggregation combinations grows rapidly when the number of dimensions increase, while the sparseness of the multidimensional space decreases in higher dimension levels, meaning that aggregates at higher levels take up nearly as much space as lower-level aggregates. In some commercial applications, full pre-aggregation takes up as much as 200 times the space of the raw data [21]. Another problem with full pre-aggregation is that it takes too long to update the materialized aggregates when base data changes.

With the goal of avoiding data explosion, research has focused on how to select the best subset of aggregation levels given space constraints [11, 9, 31, 1, 27, 25] or maintenance time constraints [10], or the best combination of aggregate data and indices [8]. This approach is commonly referred to as *practical* (or partial or semi-eager [5, 11, 28]) pre-aggregation. Commercial OLAP systems now also exist that employ practical pre-aggregation, e.g., Microsoft Decision Support Services (Plato) [18] and Informix MetaCube [13].

The premise underlying the applicability of practical pre-aggregation is that lower-level aggregates can be *re-used* to compute higher-level aggregates, known as summarizability [16]. Summarizability occurs when the mappings in the dimension hierarchies are *onto* (all paths in the hierarchy have equal lengths), *covering* (only immediate parent and child values can be related), and *strict* (each child in a hierarchy has only one parent); and when also the relationships between facts and dimensions are many-to-one and facts are always mapped to the lowest levels in the dimensions [16]. However, the data encountered in many real-world applications fail to

comply with this rigid regime. This motivates the search for techniques that allow practical pre-aggregation to be used for a wider range of applications, the focus of this paper.

Specifically, this papers leverages research such as that cited above. It does so by showing how to transform dimension hierarchies to obtain summarizability, and by showing how to integrate the transformed hierarchies into current systems, transparently to the user, so that standard OLAP technology is re-used. Specifically, algorithms are presented that automatically transform dimension hierarchies to achieve summarizability for hierarchies that are non-onto, non-covering, and non-strict. The algorithms have low computational complexity, and are thus applicable to even very large databases. It is also described how to use the algorithms to contend with non-summarizable relationships between facts and dimensions, and it is shown how the algorithms may be modified to accommodate incremental computation, thus minimizing the maintenance cost associated with base-data updates.

To our knowledge, this work is the first to present algorithms to automatically achieve summarizability for non-covering and non-onto hierarchies. The research reported here is also the first to demonstrate techniques and algorithms for achieving summarizability in non-strict hierarchies. The integration of the techniques into current systems, transparently to the user, we believe is a novel feature. The only past research on the topic has been on how to manually, and not transparently to the user, achieve summarizability for non-covering hierarchies [24].

The next section presents a real-world clinical case study that exemplifies the non-summarizable properties of real-world applications. Section 3 proceeds to define the aspects of a multidimensional data model necessary for describing the new techniques, and defines also important properties related to summarizability. Sections 4 and 5 present algorithms that transform dimension hierarchies to achieve summarizability, then apply the algorithms to fix non-summarizable relationships between facts and dimensions. Section 6 demonstrates how the techniques may be integrated into current systems, transparently to the user. Section 7 summarizes and points to topics for future research. Appendix A describes how to modify the algorithms to accommodate incremental computation.

2 Motivation—A Case Study

This section presents a case study that illustrates the properties of real-world dimension hierarchies. The case study concerns patients in a hospital, their associated diagnoses, and their places of residence. The data analysis goal is to investigate whether some diagnoses occur more often in some areas than in others, in which case environmental or lifestyle factors might be contributing to the disease pattern. An ER diagram illustrating the underlying data is seen in Figure 1.

The most important entities are the *patients*, for which we record the name. We always want to count the number of patients, grouped by some properties of the patients. Thus, in multidimensional terms, the patients are the *facts*, and the other, describing, entities constitute the *dimensions*.

Each patient has a number of *diagnoses*, leading to a *many-to-many* relationship between facts and the diagnosis dimension. When registering diagnoses of patients, physicians use dif-

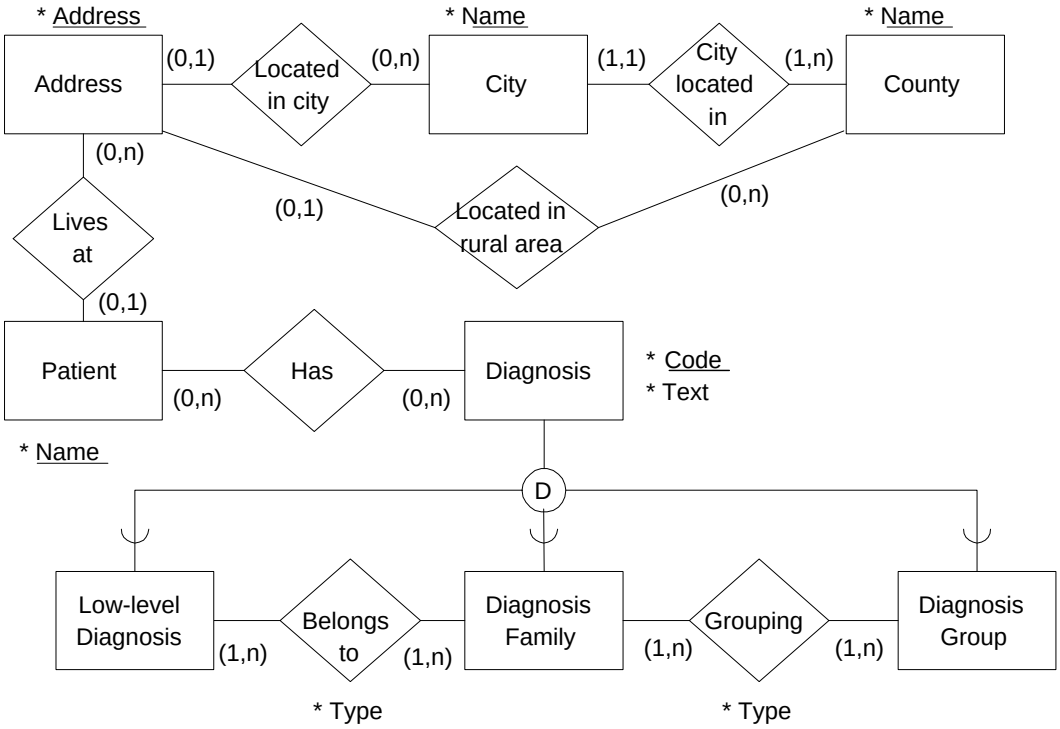

Figure 1: ER Schema of Case Study

ferent levels of granularity, ranging from very precise diagnoses, e.g., "Insulin dependent diabetes during pregnancy," to more imprecise diagnoses, e.g., "Diabetes," which cover wider ranges of patient conditions. To model this, the relationship from patient to diagnoses is to the supertype "Diagnosis," which then has three subtypes, corresponding to different levels of granularity, the *low-level diagnosis*, the *diagnosis family*, and the *diagnosis group*. Examples of these are "Insulin dependent diabetes during pregnancy," "Insulin dependent diabetes," and "Diabetes," respectively. The higher-level diagnoses are both (imprecise) diagnoses in their own right, but also serve as groups of lower-level diagnoses.

Each diagnosis has an alphanumeric code and a descriptive text, which are specified by some standard, here the World Health Organization's International Classification of Diseases (ICD-10) [30], or by the physicians themselves. Indeed, two hierarchies are captured: the standard hierarchy specified by the WHO, and the user-defined hierarchy, which is used for grouping diagnoses on an ad-hoc basis in other ways than given by the standard. The *Type* attribute on the relationships determines whether the relation between two entities is part of the standard or the user-defined hierarchy.

The hierarchy groups low-level diagnoses into *diagnosis families*, each of which consists of 2–20 related diagnoses. For example, the diagnosis "Insulin dependent diabetes during pregnancy[1]" is part of the family "Diabetes during pregnancy." In the WHO hierarchy, a low-level

[1]The reason for having a separate pregnancy related diagnosis is that diabetes must be monitored and controlled

diagnosis belongs to exactly one diagnosis family, whereas the user-defined hierarchy does not have this restriction. Thus, a low-level diagnosis can belong to several diagnosis families, e.g., the "Insulin dependent diabetes during pregnancy" diagnosis belongs to both the "Diabetes during pregnancy" and the "Insulin dependent diabetes" family. Next, diagnosis families are grouped into *diagnosis groups*, consisting of 2–10 families, and one family may be part of several groups. For example, the family "Diabetes during pregnancy" may the part of the "Diabetes" and "Other pregnancy related diseases" groups.

In the WHO hierarchy, a family belongs to exactly one group. In the WHO hierarchy, a lower-level value belongs to exactly one higher-level value, making it *strict* and *covering*. In the user-defined hierarchy, a lower-level value may belong to zero or more higher-level values, making it *non-strict* and *non-covering*. Properties of the hierarchies will be discussed in more detail in Section 3.2.

We also record the addresses of the patients. If the address is located in a city, we record the *city*; otherwise, if the address is in a rural area, we record the *county* in which the address is located. A city is located in exactly one county. As not all addresses are in cities, we cannot find all addresses in a county by going through the "City located in" relationship. Thus, the mapping from addresses to cities is *non-covering* w.r.t. addresses. For cities and counties, we just record the name. Not all counties have cities in them, so the mapping from cities to counties is *into* rather than *onto*.

In order to exemplify the data, we assume a standard mapping of the ER diagram to relational tables, i.e., one table per entity and relationship type. We also assume the use of surrogate keys, named *ID*, with globally unique values. The three subtypes of the Diagnosis type are mapped to a common Diagnosis table, and because of this, the "belongs to" and "grouping" relationships are mapped to a common "Grouping" table. The resulting tables with sample data are shown in Table 1 and will be used in examples throughout the paper.

If we apply pre-aggregation to the data from the case study, several problems occur. For example, if the counts of patients by City are pre-computed and we use these for computing the numbers of patients by county, an incorrect result will occur. In the data, the addresses "123 Rural Road" and "1 Sandy Dunes" (one of them is the address of a patient) are not in any city, making the mapping from City to County not *covering* w.r.t. addresses.

Next, if the counts of patients by Low-Level Diagnosis are pre-computed and we use these for computing the total count of patients, an incorrect result again ensues. First, patients only with lung cancer are not counted, as lung cancer is not present at the level of Low-Level Diagnosis; the mapping from Low-Level Diagnosis to Diagnosis Family is *into*. Second, patients such as "Jim Doe" only have higher-level diagnoses and will no be counted; the fact-to-dimension mapping has *varying granularity*. Third, patients such as "Jane Doe" have several diagnoses and will be counted several times; , the relationship between facts and dimensions is *many-to-many*. Fourth, Low-Level diagnoses such as "Insulin dependent diabetes during pregnancy" are part of several diagnosis families, which may also lead to "double" counting when computing higher-level counts; the dimension hierarchy is *non-strict*.

These problems yield "non-summarizable" dimension hierarchies that severely limit the

particularly intensely during a pregnancy to assure good health of both mother and child.

ID	Name
1	John Doe
2	Jane Doe
3	Jim Doe

Patient

PatientID	AddressID
1	50
2	51
3	52

LivesAt

ID	Address
50	21 Central Street
51	34 Main Street
52	123 Rural Road
53	1 Sandy Dunes

Address

PatientID	DiagnosisID	Type
1	9	Primary
2	5	Secondary
2	9	Primary
3	11	Primary

Has

ParentID	ChildID	Type
4	5	WHO
4	6	WHO
9	5	User-defined
10	6	User-defined
11	9	WHO
11	10	WHO
12	4	WHO
13	14	WHO

Grouping

ID	Code	Text	Type
4	O24	Diabetes during pregnancy	Family
5	O24.0	Insulin dependent diabetes during pregnancy	Low-Level
6	O24.1	Non insulin dependent diabetes during pregnancy	Low-Level
9	E10	Insulin dependent diabetes	Family
10	E11	Non insulin dependent diabetes	Family
11	E1	Diabetes	Group
12	O2	Other pregnancy related diseases	Group
13	A1	Cancer	Group
14	A11	Lung cancer	Family

Diagnosis

ID	Name
20	Sydney
21	Melbourne

City

AddressID	CityID
50	20
51	21

LocatedInCity

ID	Name
30	Sydney
31	Melbourne
32	Outback

County

ID	Name
52	31
53	32

LocatedInRuralArea

CityID	CountyID
20	30
21	31

CityLocatedIn

Table 1: Tables for the Case Study

applicability of practical pre-aggregation, leaving only full pre-aggregation, requirering huge amounts of storage, or no pre-aggregation, resulting in long response time for queries.

The properties described above are found in many other real-world applications. Many-to-many relationships between facts and dimensions occur between bank customers and accounts, between companies and Standard Industry Classifications (SICs), and between students and departments [15, 16]. Non-strict dimension hierarchies occur from cities to states in a Geography dimension [24] and from weeks to months in a Time dimension. In addition, hierarchies where the change over time is captured are generally non-strict. The mapping from holidays to weeks as well as organization hierarchies of varying depth [12] offer examples of "into" mappings. Non-covering relationships exist for days-holidays-weeks and for counties-cities-states, as well as in organization hierarchies [12].

Even though many real-world cases possess the properties described above, current techniques for practical pre-aggregation require that facts are in a many-to-one relationships to dimensions and that all hierarchies are strict, onto, and covering. Thus, current techniques cannot be applied when the hierarchies has these properties.

3 Method Context

This section defines the aspects of a multidimensional data model that are necessary to define the techniques that enable practical pre-aggregation in applications as the one just described. The full model is described elsewhere [22]. Next, the data model context is exploited for defining properties of hierarchies relevant to the techniques.

The particular data model has been chosen over other multidimensional data models because it quite naturally captures the data described in the case study and because it includes explicit concepts of dimensions and dimension hierarchies, which is very important for clearly presenting the techniques. However, the techniques are also applicable to other multidimensional or statistical data models, as will be discussed in Section 6.

3.1 A Concrete Data Model Context

For each part of the model, we define the *intension* and the *extension*, and we give an illustrating example.

An *n-dimensional fact schema* is a two-tuple $\mathcal{S} = (\mathcal{F}, \mathcal{D})$, where $\mathcal{F}$ is a *fact type* and $\mathcal{D} = \{\mathcal{T}_i, i = 1, .., n\}$ is its corresponding *dimension types*.

Example 1 In the case study from Section 2, *Patient* is the fact type, and *Diagnosis*, *Residence*, and *Name* are the dimension types. The intuition is that *everything* that characterizes the fact type is considered to be *dimensional*.

A dimension type $\mathcal{T}$ is a four-tuple $(\mathcal{C}, \leq_\mathcal{T}, \top_\mathcal{T}, \bot_\mathcal{T})$, where $\mathcal{C} = \{\mathcal{C}_j, j = 1, .., k\}$ are the *category types* of $\mathcal{T}$, $\leq_\mathcal{T}$ is a partial order on the $\mathcal{C}_j$'s, with $\top_\mathcal{T} \in \mathcal{C}$ and $\bot_\mathcal{T} \in \mathcal{C}$ being the top and bottom element of the ordering, respectively. Thus, the category types form a lattice. The intuition is that one category type is "greater than" another category type if members of the

former's extension logically contain members of the latter's extension, i.e., they have a larger value size. The top element of the ordering corresponds to the largest possible value size, that is, there is only one value in it's extension, logically containing all other values.

We say that $\mathcal{C}_j$ *is a category type of* $\mathcal{T}$, written $\mathcal{C}_j \in \mathcal{T}$, if $\mathcal{C}_j \in \mathcal{C}$.

Example 2 Low-level diagnoses are contained in diagnosis families, which are contained in diagnosis groups. Thus, the *Diagnosis* dimension type has the following order on its category types: $\perp_{Diagnosis}$ = *Low-level Diagnosis* < *Diagnosis Family* < *Diagnosis Group* < $\top_{Diagnosis}$. Other examples of category types are *Address*, *City*, and *County*. Figure 2, to be discussed in detail later, illustrates the dimension types of the case study.

A *category* C_j of type $\mathcal{C}_j$ is a set of *dimension values* e. A *dimension* D of type $\mathcal{T} = (\{\mathcal{C}_j\}, \leq_{\mathcal{T}}, \top_{\mathcal{T}}, \perp_{\mathcal{T}})$ is a two-tuple $D = (C, \leq)$, where $C = \{C_j\}$ is a set of categories C_j such that $Type(C_j) = \mathcal{C}_j$ and $\leq$ is a partial order on $\cup_j C_j$, the union of all dimension values in the individual categories. We assume a function $Pred : C \mapsto 2^C$ that gives the set of immediate predecessors of a category C_j. Similarly, we a assume a function $Desc : C \mapsto 2^C$ that gives the set of immediate descendants of a category C_j. For both $Pred$ and $Desc$, we "count" from the category $\top_{\mathcal{T}}$ (of type $\top_{\mathcal{T}}$), so that category $\top_{\mathcal{T}}$ is the ultimate predecessor and category $\perp_{\mathcal{T}}$ (of type $\perp_{\mathcal{T}}$) is the ultimate descendant.

The definition of the partial order is: given two values e_1, e_2 then $e_1 \leq e_2$ if e_1 is logically contained in e_2. We say that C_j is a category of D, written $C_j \in D$, if $C_j \in C$. For a dimension value e, we say that e is a dimensional value of D, written $e \in D$, if $e \in \cup_j C_j$.

The category of type $\perp_{\mathcal{T}}$ in dimension of type $\mathcal{T}$ contains the values with the smallest value size. The category with the largest value size, with type $\top_{\mathcal{T}}$, contains exactly one value, denoted $\top$. For all values e of the dimension D, $e \leq \top$. Value $\top$ is similar to the *ALL* construct of Gray et al. [6]. When the context is clear, we refer to a category of type $\top_{\mathcal{T}}$ as a $\top$ category, not to be confused with the $\top$ dimension value.

Example 3 In our *Diagnosis* dimension we have the following categories, named by their type. The numbers in parentheses are the ID values from the Diagnosis table in Table 1. *Low-level Diagnosis* = {"Insulin dependent diabetes during pregnancy" (5), "Non insulin dependent diabetes during pregnancy" (6)}, *Diagnosis Family* = {"Diabetes during pregnancy" (4), "Insulin dependent diabetes" (9), "Non insulin dependent diabetes" (10), "Lung cancer" (14)}, *Diagnosis Group* = {"Diabetes" (11), "Other pregnancy related diseases" (12), "Cancer" (13)}, and $\top_{Diagnosis} = \{\top\}$. We have that $Pred(\textit{Low-level Diagnosis}) = \{\textit{Diagnosis Family}\}$. The partial order $\leq$ is obtained by combining WHO and user-defined hierarchies, as given by the Grouping table in Table 1. Additionally, the top value $\top$ is greater than, i.e., logically contains, all the other diagnosis values.

Let F be a set of facts, and $D = (C = \{C_j\}, \leq)$ a dimension. A *fact-dimension relation* between F and D is a set $R = \{(f, e)\}$, where $f \in F$ and $e \in \cup_j C_j$. Thus R links facts to dimension values. We say that fact f is *characterized by* dimension value e, written $f \rightsquigarrow e$, if $\exists e_1 \in D\ ((f, e_1) \in R \wedge e_1 \leq e)$. We require that $\forall f \in F\ (\exists e \in \cup_j C_j\ ((f, e) \in R))$; thus, all fact maps to at least one dimension value in every dimension. The $\top$ value is used to represent

an unknown or missing value, as $\top$ logically contains all dimension values, and so a fact f is mapped to $\top$ if it cannot be characterized within the particular dimension.

Example 4 The fact-dimension relation R links patient facts to diagnosis dimension values as given by the Has table from the case study, so that $R = \{$("John Doe" (1), "Insulin dependent diabetes" (9)), ("Jane Doe" (2), "Insulin dependent diabetes during pregnancy" (5)), ("Jane Doe" (2), "Insulin dependent diabetes" (9)), ("Jim Doe" (3), "Diabetes" (11))$\}$. Note that facts may be related to values in higher-level categories. We do not require that e belongs to $\perp_{Diagnosis}$. For example, the fact "John Doe" (1) is related to the diagnosis "Insulin dependent diabetes" (5), which belongs to the *Diagnosis Family* category. This feature will be used later to explicitly capture the different granularities in the data. If no diagnosis was known for patient "John Doe" (1), we would have added the pair ("John Doe" (1),$\top$) to R.

A *multidimensional object* (MO) is a four-tuple $M = (\mathcal{S}, F, D, R)$, where $\mathcal{S} = (\mathcal{F}, \mathcal{D} = \{\mathcal{T}_i\})$ is the fact schema, $F = \{f\}$ is a set of *facts* f where $Type(f) = \mathcal{F}$, $D = \{D_i, i = 1, .., n\}$ is a set of *dimensions* where $Type(D_i) = \mathcal{T}_i$, and $R = \{R_i, i = 1, .., n\}$ is a set of fact-dimension relations, such that $\forall i((f, e) \in R_i \Rightarrow f \in F \wedge \exists C_j \in D_i(e \in C_j))$.

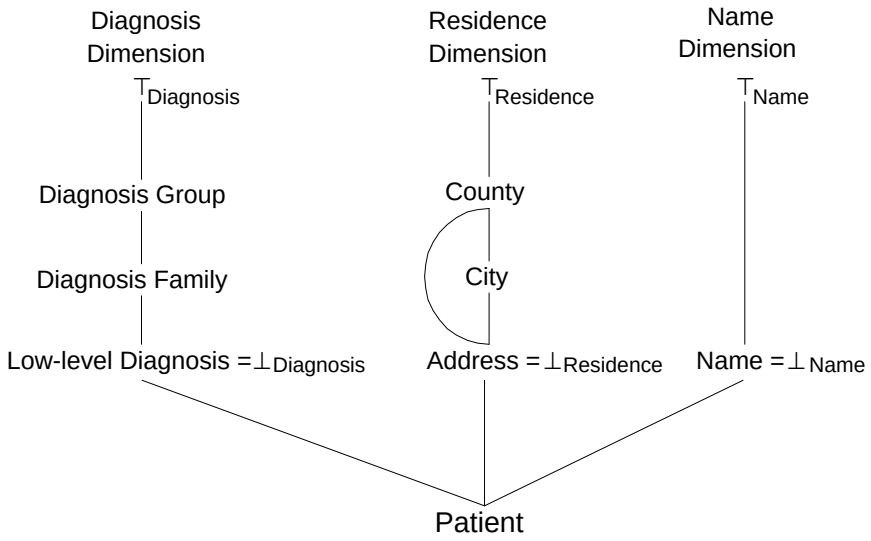

Figure 2: Schema of the Case Study

Example 5 For the case study, we get a three-dimensional MO $M = (\mathcal{S}, F, D, R)$, where $\mathcal{S} = (\textit{Patient}, \{\textit{Diagnosis, Name, Residence}\})$ and $F = \{$"John Doe" (1), "Jane Doe" (2), "Jim Doe" (3)$\}$. The definition of the diagnosis dimension and its corresponding fact-dimension relation was given in the previous examples. The Residence dimension has the categories *Address* ($= \perp_{Residence}$), *City*, *County*, and $\top_{Residence}$. The values of the categories are given by the corresponding tables in Table 1. The partial order is given by the relationship tables. Additionally, the only value in the $\top_{Residence}$ category is $\top$, which logically contains all the other values in

the Residence dimension. The Name dimension is simple, i.e., it just has a *Name* category ($= \perp_{Name}$) and a $\top$ category. We will refer to this MO as the "Patient" MO. A graphical illustration of the schema of the "Patient" MO is seen in Figure 2. Because some addresses map directly to counties, County is an immediate predecessor of Address.

The facts in an MO are objects with *value-independent identity*. We can test facts for equality, but do not assume an ordering on the facts. The combination of dimensions values that characterize the facts of a fact set is *not* a "key" for the fact set. Thus, several facts may be characterized by the same combination of dimension values. But, the facts of an MO is a *set*, so an MO does not have duplicate *facts*. The model formally defines quite general concepts of dimensions and dimension hierarchies, which is ideal for the presentation of our techniques. The presented techniques are not limited by the choice of data model.

3.2 Hierarchy Properties

In this section important properties of MOs that relate to the use of pre-computed aggregates are defined. The properties will be used in the following sections to state exactly what problems the proposed algorithms solve. The first important concept is *summarizability*, which intuitively means that higher-level aggregates may be obtained directly from lower-level aggregates.

Definition 1 Given a type T, a set $S = \{S_j, j = 1, .., k\}$, where $S_j \in 2^T$, and a function $g : 2^T \mapsto T$, we say that g is *summarizable* for S if $g(\{\{g(S_1), .., g(S_k)\}\}) = g(S_1 \cup .. \cup S_k)$. The argument on the left-hand side of the equation is a multiset, i.e., the same value may occur multiple times.

Summarizability is important as it is a condition for the flexible use of pre-computed aggregates. Without summarizability, lower-level results generally cannot be directly combined into higher-level results. This means that we cannot choose to pre-compute only a relevant selection of the possible aggregates and then use these to (efficiently) compute higher-level aggregates on-the-fly. Instead, we have to pre-compute the all the aggregate results of queries that we need fast answers to, while other aggregates must be computed from the base data. Space and time constraints can be prohibitive for pre-computing all results, while computing aggregates from base data is often inefficient.

It has been shown that summarizability is equivalent to the aggregate function (g) being *distributive*, all paths being *strict*, and the mappings between dimension values in the hierarchies being *covering* and *onto* [16]. These concepts are formally defined below. The definitions assume a dimension $D = (C, \leq)$ and an MO $M = (\mathcal{S}, F, D, R)$.

Definition 2 Given two categories, C_1, C_2 such that $C_2 \in Pred(C_1)$, we say that the mapping from C_1 to C_2 is *onto* iff $\forall e_2 \in C_2(\exists e_1 \in C_1\ (e_1 \leq e_2))$. Otherwise, it is *into*. If all mappings in a dimension are onto, we say that the dimension hierarchy is *onto*.

Mappings that are into typically occur when the dimension hierarchy has varying height. In the case study, there is no low-level cancer diagnosis, meaning that some parts of the hierarchy

have height 2, while most have height 3. It is thus not possible to use aggregates at the Low-level Diagnosis level for computing aggregates at the two higher levels. Mappings that are into also occur often in organization hierarchies.

Definition 3 Given three categories, C_1, C_2, and C_3 such that $Type(C_1) < Type(C_2) < Type(C_3)$, we say that the mapping from C_2 to C_3 is *covering with respect to* C_1 iff $\forall e_1 \in C_1\ (\forall e_3 \in C_3\ (e_1 \leq e_3 \Rightarrow \exists e_2 \in C_2\ (e_1 \leq e_2 \wedge e_2 \leq e_3)))$. Otherwise, it is *non-covering with respect to* C_1. If all mappings in a dimension are covering w.r.t. any category, we say that the dimension hierarchy is *covering*.

Non-covering mappings occur when some of the links between dimension values skip one or more levels and map directly to a value located higher up in the hierarchy. In the case study, this happens for the "1 Sandy Dunes" address, which maps directly to "Outback County" (there are no cities in Outback County). Thus, we cannot use aggregates at the City level for computing aggregates at the County level.

Definition 4 Given an MO $M = (\mathcal{S}, F, D, R)$, and two categories C_1 and C_2 that belong to the same dimension $D_i \in D$ such that $Type(C_1) < Type(C_2)$, we say that the mapping from C_1 to C_2 is *covering with respect to* F, the set of facts, iff $\forall f \in F\ (\forall e_2 \in C_2\ (f \rightsquigarrow_i e_2 \Rightarrow \exists e_1 \in C_1\ (f \rightsquigarrow_i e_1 \wedge e_1 \leq_i e_2)))$.

This case is similar to the one above, but now it is the mappings between facts and dimension values that may skip one or more levels and map facts directly to dimension values in categories above the bottom level. In the case study, the patients can map to diagnoses anywhere in the Diagnosis dimension, not just to Low-level Diagnoses. This means that we cannot use aggregates at the Low-level Diagnosis Level for computing aggregates higher up in the hierarchy.

Definition 5 Given two categories, C_1 and C_2 such that $C_2 \in Pred(C_1)$, we say that the mapping from C_1 to C_2 is *strict* iff $\forall e_1 \in C_1\ (\forall e_2, e_3 \in C_2\ (e_1 \leq e_2 \wedge e_1 \leq e_3 \Rightarrow e_2 = e_3))$. Otherwise, it is *non-strict*. The hierarchy in dimension D is *strict* if all mappings in it are strict; otherwise, it is *non-strict*. Given an MO $M = (\mathcal{S}, F, D, R)$ and a category C_j in some dimension $D_i \in D$, we say that there is a *strict path* from the set of facts F to C_j iff $\forall f \in F(f \rightsquigarrow_i e_1 \wedge f \rightsquigarrow_i e_2 \wedge e_1 \in C_j \wedge e_2 \in C_j \Rightarrow e_1 = e_2)$. (Note that the paths to the $\top_{\mathcal{T}}$ categories are always strict.)

Non-strict hierarchies occur when a dimension value has multiple parents. This occurs in the Diagnosis dimension in the case study where the "Insulin dependent diabetes during pregnancy" low-level diagnosis is part of both the "Insulin Dependent Diabetes" and the "Diabetes during pregnancy" diagnosis families, which in turn both are part of the "Diabetes" diagnosis group. This means that we cannot use aggregates at the Diagnosis Family level to compute aggregates at the Diagnosis Group level, since data for "Insulin dependent diabetes during pregnancy" would then be counted twice.

Definition 6 If the dimension hierarchy for a dimension D is *onto*, *covering*, and *strict*, we say that D is *normalized*. Otherwise, it is *un-normalized*. For an MO $M = (\mathcal{S}, D, F, R)$, if all dimensions $D_i \in D$ are normalized and $\forall R_i \in R\ ((f, e) \in R_i \Rightarrow e \in \perp_D)$ (, i.e., all facts map to dimension values in the bottom category), we say that M is *normalized*. Otherwise, it is *un-normalized*.

For normalized hierarchies and MOs, all mappings are summarizable, meaning that we can pre-aggregate values at any combination of dimension levels and safely re-use the pre-aggregated values to compute higher-level aggregate results. Thus, we want to normalize the dimension hierarchies and MOs for which we want to apply practical pre-aggregation.

We proceed to describe how the normalization of the dimension hierarchies and MOs used for aggregation is achieved. We first show how to perform transformations on dimension hierarchies, then later describe how the same techniques may be applied to eliminate the non-summarizable properties of fact-dimension relations.

4 Dimension Transformation Techniques

This section describes how dimensions can be transformed to achieve summarizability. Transforming dimensions on their own, separately from the facts, results in well-behaved dimensions that can be applied in a number of different systems or sold to third-party users. The transformation of the dimension hierarchies is a three-step operation. First, all mappings are transformed to be *covering*, by introducing extra "intermediate" values. Second, all mappings are transformed to be *onto*, by introducing "placeholder" values at lower levels for values without any children. Third, mappings are made *strict*, by "fusing" values together. The three steps are treated in separate sections. None of the algorithms introduce any non-summarizable properties, so applying each once is sufficient.

In general, the algorithms take as input a set of tables R_{C_1,C_2} that specifies the mapping from dimension values in category C_1 to values in category C_2. The input needs not contain all pairs of ancestors and descendants—only direct parent-child relationships are required. If there are non-covering mappings in the hierarchy, we have categories C, P, H such that $\{P, H\} \subseteq Pred(C)$ and $Type(P) < Type(H)$. In this case, the input must also contain $R_{P,H}$ tables that map P values to H values.

4.1 Non-Covering Hierarchies

The first algorithm renders all mappings in a dimension hierarchy covering w.r.t. any category. When a dimension value is mapped *directly* to another value in a category higher than the one immediately above it in the hierarchy, a new intermediate value is inserted into the category immediately above, and the two original dimension values are linked to this new value, rather than to each other.

Example 6 In the hierarchy for the Residence dimension, two links go from Address directly to County. The address "123 Rural Road" (52) is in "Melbourne County" (31), but not in a

city, and the address "1 Sandy Dunes" (53) is in "Outback County" (32), which does *not* have any cities at all. The algorithm inserts two new dimension values in the City category, **C31** and **C32**, which represent Melbourne and Outback county, respectively, and links them to their respective counties. The addresses "123 Rural Road" and "1 Sandy Dunes" are then linked to **C31** and **C32**, respectively. This occurs in the first call of procedure MakeCovering (on the Address category; the procedure is given below). When MakeCovering is called recursively on the City, County, and $\top$ categories, nothing happens, as all mappings are already covering. The transformation is illustrated graphically in Figure 3. The dotted lines show the "problematic" links, and the bold-face values and thick lines show the new dimension values and links.

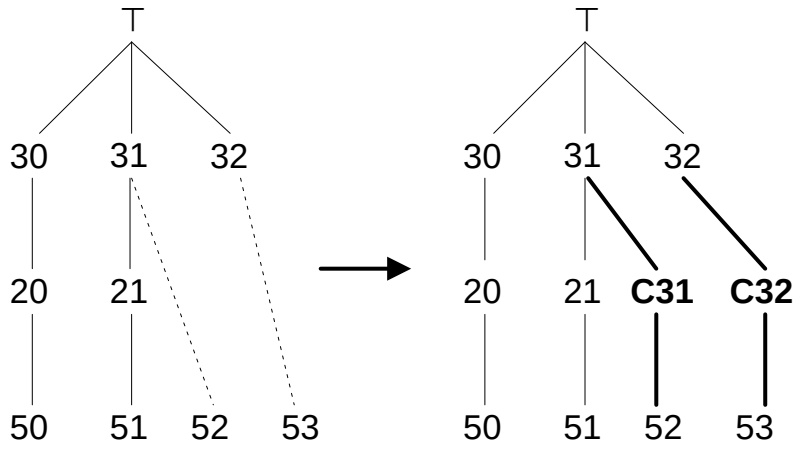

Figure 3: Transformations by the MakeCovering Algorithm

In the algorithm, C is a *child* category, P is a *parent* category, H is a "*higher*" category, L are the non-covering *links* from C to H, and N are the "higher" dimension values in L. The $\bowtie$ operator denotes natural join. The algorithm works as follows. Given the argument category C (initially the bottom category) in line (1), the algorithms goes through all C's parent categories P (2). For each parent category P, it looks for predecessor categories H of C that are "higher" in the hierarchy than P (4). If such an H exist, there might be links in the mapping from C to H that are not available by going through P. Line (6) finds these "non-covered" links, L, in the mapping from C to H by "subtracting" the links that *are* available by going through P from *all* the links in the mapping from C to H. Line (7) uses L to find the dimension values N in H that participate in the "non-covered" mappings. For each value in N, line 8 inserts a corresponding marked value into P; these marked values represent the N values in P. The marked values in P are then linked to the original values in H (9) and C (10). Line (12) contains a recursive call to the algorithm P, thus fixing mappings higher up in the hierarchy. The algorithm terminates when it reaches the $\top$ category, which has no predecessors.

All steps in the algorithm are expressed using standard relational algebra operators. The *general* worst-case complexity of join is $\mathcal{O}(n^2)$, where n is the size of the input. However, because the input to the algorithm are hierarchy definitions, the complexity of the join in the algorithm

will only be $\mathcal{O}(n \log n)$. Thus, all the operators used can be evaluated in time $\mathcal{O}(n \log n)$, where n is the size of the input. The $Mark$ operation can be performed in $\mathcal{O}(1)$ time. The inner loop of the algorithm is evaluated at most once for each link between categories, i.e., at most $k^2/2$ times, where k is the number af categories (if all categories are directly linked to all others). Thus, the overall big-$\mathcal{O}$ complexity of the algorithm is $\mathcal{O}(k^2 n \log n)$, where k is the number of categories and n is the size of the largest participating R_{C_1,C_2} relation. The worst-case complexity will not apply very often; in most cases, the inner loop will only be evaluated at most k times.

(1) **procedure** MakeCovering(C)
(2) **for** each $P \in Pred(C)$ **do**
(3) **begin**
(4) **for** each $H \in Pred(C)$ where $Type(H) > Type(P)$ **do**
(5) **begin**
(6) $L \leftarrow R_{C,H} \setminus \Pi_{C,H}(R_{C,P} \bowtie R_{P,H})$
(7) $N \leftarrow \Pi_H(L)$
(8) $P \leftarrow P \cup \{Mark(h) \mid h \in N\}$
(9) $R_{P,H} \leftarrow R_{P,H} \cup \{(Mark(h), h) \mid h \in N\}$
(10) $R_{C,P} \leftarrow R_{C,P} \cup \{(c, Mark(h)) \mid (c, h) \in L\}$
(11) **end**
(12) MakeCovering(P)
(13) **end**
(14) **end**

The algorithm inserts new values into the P category to ensure that the mappings from P to higher categories are summarizable, i.e., that pre-aggregated results for P can be directly combined into higher-level aggregate results. The new values in P mean that the cost of materializing aggregate results for P is higher for the transformed hierarchy than for the original. However, if the hierarchy was not transformed to achieve summarizability, we would have to materialize aggregates for G, and perhaps also for higher level categories. At most one new value is inserted into P for every value in G, meaning that the extra cost of materializing results for P is never greater than the cost of the (otherwise necessary) materialization of results for G. This is a very unlikely worst-case scenario—in the most common cases, the extra cost for P will be much lower than the the cost of materializing results for G, and the savings will be even greater because materialization of results for higher-level categories may also be avoided.

The correctness argument for the algorithm has two aspects. First, the mappings in the hierarchy should be *covering* upon termination. Second, the algorithm should only make transformations that are semantically correct, i.e., we should get the same results when computing results with the new hierarchy as with the old. The correctness follows from Theorem 1 and 2, below. As new values are inserted in the P category, we will get aggregate values for both the new and the original values when "grouping" by P. Results for the original values will be the same as before, so the original result set is a *subset* of the result set obtained with the transformed hierarchy.

Theorem 1 Algorithm MakeCovering terminates and the hierarchy for the resulting dimension D' is covering.

Proof: By induction in the height of the dimension lattice. *Base*: The height is 0, making the statement trivially true. *Induction Step*: We assume the statement is true for dimension lattices of height n, and consider lattices of height $n+1$. For termination, we note that there is a finite number of (P, H) pairs, all operations in the inner loop terminate, and the algorithm is called recursively on P, which is the root of a lattice of height n. For the covering property, we note that the insertion of intermediate, marked values into P means that the mapping from P to H is covering w.r.t. C. By the induction hypothesis, the mappings higher in the hierarchy are fixed by the recursive call of the algorithm.

Theorem 2 Given dimensions D and D' such that D' is the result of running MakeCovering on D, an aggregate result obtained using D is a subset of the result obtained using D'.

Proof: Follows easily from Lemma 1, next, as the inserted values are "internal" in the hierarchy.

Lemma 1 For the dimension $D' = (C', \leq')$ resulting from applying algorithm MakeCovering to dimension $D = (C, \leq)$, the following holds: $\forall e_1, e_2 \in D\,(e_1 \leq' e_2 \Leftrightarrow e_1 \leq e_2)$ (there is a path between any two original dimension values in the new dimension hierarchy iff there was a path between them in the original hierarchy).

Proof: By induction in the height of the dimension lattice. *Base*: The height is 0 making the statement trivially true. *Induction Step*: We assume the statement is true for dimension lattices of height n, and consider lattices of height $n+1$. Examing the inner loop, we see that the insertion of intermediate values into P, and the linking of values in C and H to these, only links values in C and H that were linked before. No links or values are destroyed by the inner loop. Thus, the statement is true for the links from C to P, and from C to H. By the induction hypothesis, the statement holds true for the transformations made by the recursive call on P.

We see that the original values in the hierarchy are still linked to exactly the same original values as before, as stated by Lemma 1, although new values might have been inserted in-between the original values. Thus, when evaluating a query using the transformed hierarchy, the results for the original values will be the same as when using the original hierarchy.

Assuming only the original result set is desired, results for the new values must be excluded, which is easy to accomplish. The new, "internal" values are marked with "mark=internal", whereas the original values have "mark=original". In order to exclude the new, internal values from the result set, the equivalent of an SQL HAVING clause condition of "mark=original" is introduced into the original query.

4.2 Non-Onto Hierarchies

The second algorithm renders all mappings in hierarchies onto, i.e., all dimension values in non-bottom categories have children. This is ensured by inserting placeholder values in lower

categories to represent the childless values. These new values are marked with the original values, making it possible to map facts to the new placeholder values instead of to the original values. This makes it possible to only map facts to the bottom category.

Example 7 In the Diagnosis dimension, the "Lung cancer" diagnosis family (ID = 14) has no children. When the algorithm reaches the Diagnosis Family category, it inserts a placeholder value (**L14**) into the Low-level Diagnosis category, representing the "Lung cancer" diagnosis, and links it to the original value. Facts mapped to the "Lung cancer" value may then instead be mapped to the new placeholder value, ensuring that facts are mapped only to the Low-level Diagnosis Category. A graphical illustration of the transformation is seen in Figure 4. The bold-faced **L14** value is the new value inserted, and the thick line between 14 and **L14** is the new link inserted.

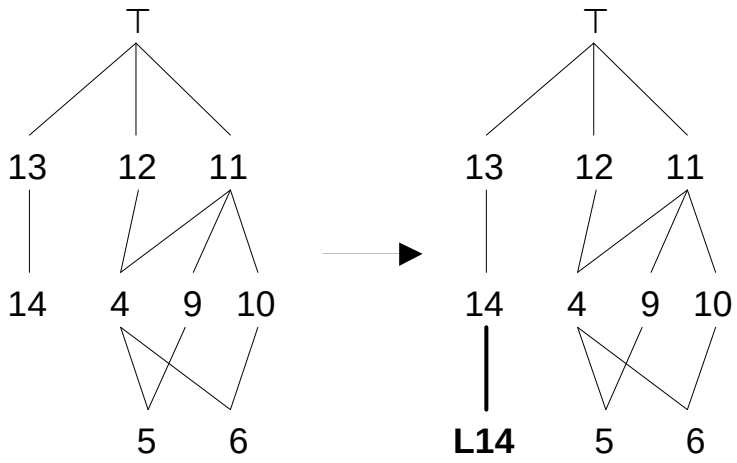

Figure 4: Transformations by the MakeOnto Algorithm

In the algorithm below, P is a *parent* category, C is a *child* category, and N holds the parent values with *no* children. The algorithm works as follows. Given a category P (initially the $\top$ category) in line (1), the algorithm goes through all categories C that are (immediate) descendants of P (2). For each C, line (4) finds the values N in P that have *no* children in C, by "subtracting" the values *with* children in C from the values in P. For each "childless" value in N, lines (5) and (6), respectively, insert into C a placeholder value marked with the parent value, and links the new value to the original. MakeOnto is then called recursively on C (7). The algorithms terminates when it reaches the $\bot$ category, which has no descendants.

Following the reasoning in Section 4.1, we find that the overall big-$\mathcal{O}$ complexity is equal to $\mathcal{O}(k^2 n \log n)$, where k is the number of categories and n is the size of the largest participating R_{C_1,C_2} relation. However, the complexity will only be $\mathcal{O}(kn \log n)$ for the most common cases.

The MakeOnto algorithm inserts new values into C to ensure that the mapping from C to P is summarizable. Again, this means that the cost of materializing results for C will be higher for

the transformed hierarchy than for the original. However, if the new values were not inserted, we would have to materialize results for P, and perhaps also higher categories, as well as C. At most one value is inserted in C for every value in P, meaning that the extra cost for C is never greater than the cost of materializing results for P. As before, this is a very unrealistic scenario, as it corresponds to the case where *no* values in P have children in C. In most cases, the extra cost for C will be a small percentage of the cost of materializing results for P, and the potential savings will be even greater, because pre-aggregation for higher-level categories may be avoided.

(1) **procedure** MakeOnto(P)
(2) **for** each $C \in Desc(P)$ **do**
(3) **begin**
(4) $N \leftarrow P \setminus \Pi_P(R_{C,P})$
(5) $C \leftarrow C \cup \{Mark(p) \mid p \in N\}$
(6) $R_{C,P} \leftarrow R_{C,P} \cup \{(Mark(p), p) \mid p \in N\}$
(7) MakeOnto(C)
(8) **end**
(9) **end**

As before, the correctness argument for the algorithm has two aspects. First, the mappings in the hierarchy should be *onto* upon termination. Second, the algorithm should only make transformations that are semantically correct. The correctness follows from Theorems 3 and 4, below. Again, the result set for the original values obtained using the original hierarchy will be a subset of the result set obtained using the transformed hierarchy. The results for the new values can be excluded from the result set by adding a HAVING clause condition.

Theorem 3 Algorithm MakeOnto terminates and the hierarchy for the resulting dimension D' is onto.

Proof: By induction in the height of the dimension lattice. *Base*: The height is 0, making the statement trivially true. *Induction Step*: We assume the statement is true for dimension lattices of height n, then consider lattices of height $n + 1$. For termination, we note that there is a finite number of descendants C for each P, that all operations in the loop terminate, and that the algorithm is called recursively on C, which is the top element in a lattice of height n. For the onto property, we note that the insertion of placeholder values into C makes the mapping from C to P onto. By the induction hypothesis, the mappings further down in the lattice are handled by the recursive call.

Theorem 4 Given dimensions D and D' such that D' is the result of applying the MakeOnto algorithm to D, an aggregate result obtained using D is a subset of the result obtained using D'.

Proof: Follows easily from the observation that "childless" dimension values are linked to new, placeholder values in lower categories in one-to-one relationships, meaning that data for childless values will still be counted exactly once in aggregate computations that use the new dimension.

4.3 Non-Strict Hierarchies

The third algorithm renders mappings in hierarchies strict, meaning that problems of "double-counting" will not occur. Non-strict hierarchies occur when one dimension value has several parent values.

The basic idea is to "fuse" a set of parent values into one "fused" value, then link the child value to this new value instead. The fused values are inserted into a new category in-between the child and parent categories. Data for the new fused category may safely be re-used for computation of higher-level aggregate results, as the hierarchy leading up to the new category is strict.

The fused value is also linked to the relevant parent values. This mapping is by nature non-strict, but this non-strictness is not a problem, as we prevent aggregate results for the parent category from being re-used higher up in the hierarchy. This is done by "unlinking" the parent category from its predecessor categories.

The categories higher up are instead reached through the fused category. This means that we can still get results for any original category, while being able to apply practical pre-aggregation throughout the hierarchy. In pre-aggregation terms, the "unlinking" of the parent categories means that we must prevent results for including this category from being materialized—only "safe" categories may be materialized. This should be given as a constraint to the pre-aggregation system that chooses which levels of aggregation to materialize.

We note that the algorithm does not introduce more *levels* in the hierarchy, only more categories, and that the number of "safe" categories in the result is the same as the number of original categories. This means that the complexity of the task of selecting the optimal aggregation levels to materialize is unaffected by the algorithm.

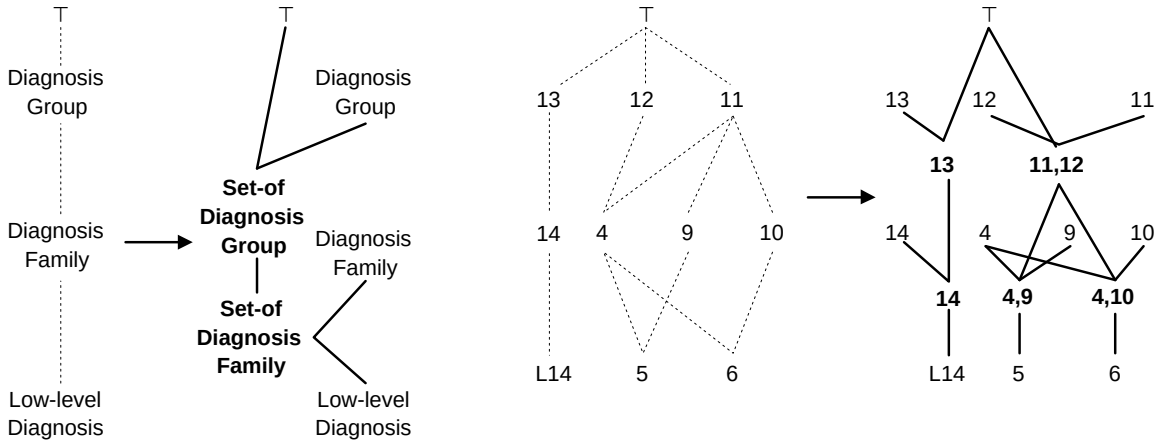

Figure 5: Schema and Value Transformations by the MakeStrict Algorithm.

Example 8 The result of running the algorithm on the Diagnosis dimension is seen in Figure 5. Because of the non-strictness in the mapping from Low-level Diagnosis to Diagnosis Family,

and from Diagnosis Family to Diagnosis Group, two new category types and the corresponding categories are introduced. The third picture indicates the argument to the algorithm; and, in addition, its dotted lines indicate the links deleted by the algorithm. The fourth picture gives the result of applying the algorithm; here, the bold-face values and thick lines indicate the values and links inserted by the algorithm.

In the first call of the algorithm the three Low-level Diagnosis values—"(low-level) Lung cancer" (L14); "Insulin dependent diabetes during pregnancy" (5); and "Non insulin dependent diabetes during pregnancy" (6)—are linked to the three new fused values—"(low-level) Lung cancer" (**14**); "Diabetes during pregnancy, Insulin dependent diabetes" (**4**, **9**); and "Diabetes during pregnancy, Non insulin dependent diabetes" (**4**, **10**)—and these are in turn linked to "Lung Cancer" (14); "Diabetes during pregnancy" (4); "Insulin dependent diabetes" (9); and "Non insulin dependent diabetes" (10). The these latter four values in the Diagnosis Family category are un-linked from their parents, as the Diagnosis Family category is "unsafe."

When called recursively on the Set-of Diagnosis Family category, the algorithm creates the new fused values "Cancer" (**13**) and "Diabetes, Other pregnancy related diseases" (**11**, **12**) in the Set-of Diagnosis Group category. These new values are linked to the values "Cancer" (13), "Diabetes" (11), and "Other pregnancy related diseases" (12) in the Diagnosis Group category, and to the $\top$ value; and the values in the Diagnosis Group category are un-linked from their parents. Note the importance of having a $\top$ value: the values not linked to $\top$ are exactly the unsafe values, for which aggregate results should not be re-used.

The algorithm assumes that all paths in the dimension hierarchy have equal length, i.e., all direct links are from children to their immediate parents. This is ensured by the MakeCovering and MakeOnto algorithms. In the algorithm below, C is a *child* category, P is a *parent* category, G is a *grandparent* category, N is the *new* category introduced to hold the "fused" values, and $\bowtie$ denotes natural join.

The algorithm takes a category C (initially the $\bot$ category) as input. I then goes through the set of immediate parent categories P of C (line (2)). Line (4) tests if there is non-strictness in the mapping from C to P *and* if P has any parents (4). If this test fails, there is no problem as aggregate results for P can either be safely re-used or are guaranteed not be re-used; and the algorithm in then invoked recursively, in line (20).

If the test succeeds, the algorithm creates a new fused category. First, a new, empty category N with domain 2^P is created in line (6). The values inserted into this category represent *sets* of values of P. For example. the value "**1**, **2**" represents the set consisting of precisely $1, 2$. Values in C are then linked to to new, fused values, representing their particular *combination* of parents in P (7). The new values are constructed using a Fuse function, that creates a distinct value for each combination of P values and stores the corresponding P values along with it.

The resulting links are used in line (8) to insert the fused values into their category N, and an "Unfuse" function, mapping fused values from N into the corresponding P values, is used in line (9) to map the values in N to those in P. In line (10), N is included in, and P is excluded from, the sets of predecessors of C. The set of predecessors of N is set to P in line (11), meaning that the new category N resides in-between C and P in the hierarchy.

For each grandparent category G, the algorithm links values in N to values in G, in line

(14), includes G in the predecessors of N, in line (15), and excludes G from the predecessors of P, in line (16), thereby also deleting the links from P to G from the hierarchy. The exclusion of the G categories from the predecessors of P means that aggregate results for P will not be re-used to compute results for the G categories.

In the end, the algorithm is called recursively on the new category, N. Note that the test for $Pred(P) \neq \emptyset$ in line (4) ensures that the mapping from N to P will not be altered, as P now has *no* predecessors.

(1) **procedure** MakeStrict (C)
(2) **for** each $P \in Pred(C)$ **do**
(3) **begin**
(4) **if** $(\exists e_1 \in C\ (\exists e_2, e_3 \in P\ (e_1 \leq e_2 \wedge e_1 \leq e_3 \wedge e_2 \neq e_3))) \wedge Pred(P) \neq \emptyset$ **then**
(5) **begin**
(6) $N \leftarrow CreateCategory(2^P)$
(7) $R_{C,N} \leftarrow \{(e_1, Fuse(\{e_2 \mid (e_1, e_2) \in R_{C,P}\}))\}$
(8) $N \leftarrow \Pi_N(R_{C,N})$
(9) $R_{N,P} \leftarrow \{(e_1, e_2) \mid e_1 \in N \wedge e_2 \in Unfuse(e_1)\}$
(10) $Pred(C) \leftarrow Pred(C) \cup \{N\} \setminus \{P\}$
(11) $Pred(N) \leftarrow \{P\}$
(12) **for** each $G \in Pred(P)$ **do**
(13) **begin**
(14) $R_{N,G} \leftarrow \Pi_{N,G}(R_{N,P} \bowtie R_{P,G})$
(15) $Pred(N) \leftarrow Pred(N) \cup \{G\}$
(16) $Pred(P) \leftarrow Pred(P) \setminus \{G\}$
(17) **end**
(18) MakeStrict(N)
(19) **end**
(20) **else** MakeStrict(P)
(21) **end**
(22) **end**

Following the reasoning in Section 4.1, we find that the overall big-$\mathcal{O}$ complexity is equal to $\mathcal{O}(pnk \log n \log k)$, where p is the number of immediate parent and children categories in the dimension type lattice, n is the size of the largest mapping in the hierarchy, and k is the maximum number of values fused together. For most realistic scenarios, p and k are small constants, yielding a low $\mathcal{O}(n \log n)$ complexity for the algorithm.

The MakeStrict algorithm constructs a new category N and insert fused values in N to achieve summarizability for the mapping from N to P, and from N to G. The algorithm only inserts the fused values for the combinations that are actually present in the mapping from C to P. This means that the cost of materializing results for N is never higher than the cost of materializing results for C. This is a worst-case scenario, for the most common cases the cost of materializing results for N will be be close to the cost of materializing results for P. However, without the introduction of N, we would have to materialize results not only for P, but also for

G and *all* higher-level categories. Thus, the potential savings in materialization costs are very high indeed.

Considering correctness, the mappings in the hierarchy should be *strict* upon termination, and the algorithm should only make transformations that are semantically correct. More specifically, it is acceptable that some mappings be non-strict, namely the ones from the new, fused categories to the unsafe parent categories. This is so because unsafe categories do *not* have predecessors in the resulting hierarchy, meaning that aggregate results for these categories will not be re-used.

The correctness follows from Theorems 5 and 6, below. When evaluating queries we get the same result for original values as when evaluating on the old hierarchy. The values that are deleted by the algorithm were not linked to any facts, meaning that these values did not contribute to the results in the original hierarchy. As all the new values are inserted into new categories that are unknown to the user, the aggregate result obtained will be the same for the original and transformed hierarchy. Thus, we do not need to modify the original query.

Theorem 5 Let D' be the dimension resulting from applying algorithm MakeStrict on dimension D. Then the following hold: Algorithm MakeStrict terminates and the hierarchy for the dimension D'', obtained by removing unsafe categories from D', is strict.

Proof: By induction in the height of the dimension lattice. *Base*: The height is 0, making the statement trivially true. *Induction Step*: Assuming that the statement is true for lattices of height n, lattices of height $n+1$ are considered. All steps in the algorithm terminate, and the algorithm is called recursively on either P (in the strict case) or N (in the non-strict case), both of which are the root of a lattice of height n, thus guaranteeing termination.

For the strictness property, there are three cases. If the mapping from C to P is already strict, this mapping is not changed, and by the induction hypothesis, the statement holds for the recursive call on P. If the mapping from C to P is non-strict, but P does not have any parents, strictness is ensured, as P is excluded from D''. If the mapping is non-strict and P has parents, the resulting mapping from C to N is strict. By the induction hypothesis, the statement holds true for the recursive call on N, as the introduction of N has not increased the height of the lattice.

Theorem 6 Given dimensions D and D' such that D' is the result of applying the MakeStrict algorithm to D, an aggregate obtained using D' is the same as that obtained using D.

Proof: Follows from Lemma 2, as all facts are mapped to values in the $\perp$ category, which is a safe category. Thus, there will be a path from a fact f to an original dimension value e iff there was one in the original hierarchy, meaning that aggregate results computed using the original and the new hierarchy will be same.

Lemma 2 For the dimension $D' = (C', \leq')$ resulting from applying algorithm MakeStrict to dimension $D = (C, \leq)$, the following holds. $\forall e_1, e_2 \in D\,(e_1 \in C_1 \wedge \mathit{Safe}(C_1) \wedge e_1 \leq' e_2 \Leftrightarrow e_1 \leq e_2)$ (there is a path between an original dimension value in a safe category and any other original dimension value in the new dimension hierarchy iff there was a path between them in the original hierarchy).

Proof: By induction in the height of the dimension lattice. *Base*: The height of the lattice is 0, making the statement trivially true. *Induction Step*: If either the mapping from C to P is strict, or P does not have any parents, the algorithm does not change the mappings, and by the indiction hypothesis, the statement is true for the recursive call on P. Otherwise, we observe that the creation of fused values in N, and the linking of C, P, and G values to these, only links *exactly* the values in C and P, or C and G, that were linked before. Because P is not safe, the links from P to G may be deleted. By the induction hypothesis, the statement is true for the recursive call on N.

5 Fact-Dimension Transformation Techniques

This section explains how the set of algorithms from Section 4 may also be applied to the relationships between facts and dimensions, thus providing a basis for enabling practical pre-aggregation on concrete MOs that include fact data.

The basic idea is to view the set of facts F as the bottom granularity in the lattice. The input to the algorithms then consists of the facts, F, the $R_{F,C}$ tables, describing the mappings from facts to dimension values, and the C and R_{C_1,C_2} tables, describing the dimension categories and the mappings between them.

Only the covering and strictness properties are considered because for the fact-dimension relationships, a mapping between facts and dimension values that is *into* means that not all dimension values in the bottom category have associated facts, which does not affect summarizability. As before, we first apply the MakeCovering algorithm, then the MakeStrict algorithm.

The computational complexity of the algorithms will now be dominated by the size, n, of the mapping between facts and dimension values, i.e., the complexity will be $\mathcal{O}(n \log n)$ if we assume the height of the lattice and the maximum number of values fused together to be small constants. This means that the algorithms can be applied to even very large databases.

5.1 Mixed Granularity Mappings

The first case to consider is the one where some of the mappings are non-covering w.r.t. the facts, meaning that not all facts can be reached through these mappings and thus resulting in these facts not being accounted for in aggregate computations. This occurs when some facts are mapped *directly* to dimension values in categories higher than the $\perp$ category, i.e., the facts are mapped to values of *mixed* granularities.

We use the MakeCovering algorithm to make the mappings covering, initially calling it on F, which is now the bottom of the lattice. The algorithm makes the mappings covering w.r.t. the facts by inserting new marked values, representing the parent values, in the intermediate categories, and by linking the facts to the new values instead of the parent values. As in Section 4.1, the marked values keep information about their original values, so that when new fact-dimension mappings are added, the links that are supposed to go *directly* to the original parent values now instead can be set to go to the marked value in the $\perp$ category.

Example 9 In the case study, the mapping between Patients and Diagnoses is of mixed granularity: "John Doe" (1) and "Jane Doe" are both mapped to the Diagnosis Family, "Insulin dependent diabetes" (9), "Jane Doe" is additionally mapped to the Low-level Diagnosis, "Insulin dependent diabetes during pregnancy" (5), and "Jim Doe" is mapped to "Diabetes" (11), a Diagnosis Group.

In the first call of the algorithm, two new Low-level Diagnoses are inserted: "**L9**," representing "Insulin dependent diabetes," and "**L11**," representing "Diabetes"; and the facts are mapped to these instead of the original values. In the recursive call on Low-level Diagnosis, an "**F11**" value representing "Diabetes" at the Diagnosis Family level is inserted between "Diabetes" and value "**L11**."

The transformations are illustrated in Figure 6, where dotted lines indicate links that are deleted by the algorithm and bold-face value and thick lines indicate dimension values and links inserted by the algorithm.

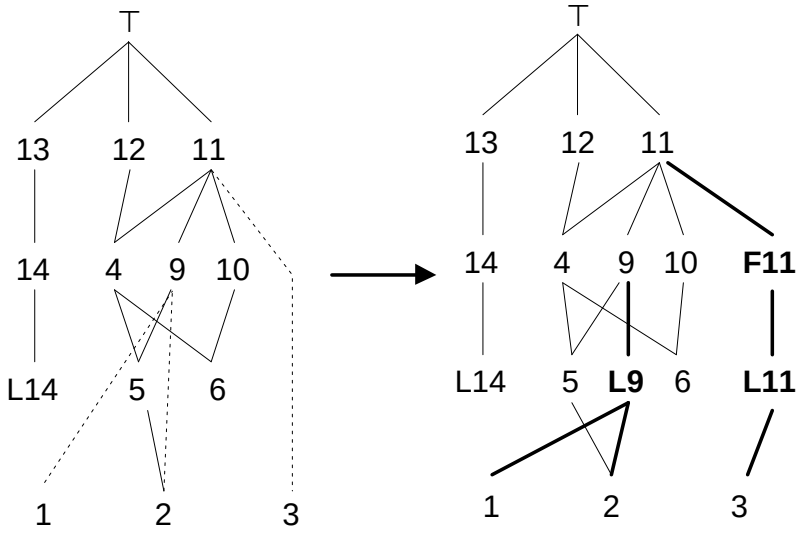

Figure 6: Transformations for Varying Granularities.

5.2 Many-To-Many Relationships

The second case occurs when relationships between facts and dimension values are many-to-many. This means that the hierarchy, with the facts as the bottom category, is non-strict, leading to possible double-counting of facts. It is enough to make the hierarchy partly strict, as described in Section 4.3. The MakeStrict algorithm is initially called on F, which is now the bottom of the hierarchy lattice. Because the MakeCovering algorithm has already been applied, all paths from facts to the $\top$ value have equal length, as required by the MakeStrict algorithm.

Some dimension values have no facts mapped to them, leading to an interesting side effect of the algorithm. When the algorithm fuses values and places the fused values in-between the original values, it also deletes the child-to-parent and parent-to-grandparent links. The fact-less dimension values are then left disconnected from the rest of the hierarchy, with no links to other values.

These fact-less dimension values do not contribute to any aggregate computations and are thus superfluous. To minimize the dimensions, an "Delete-unconnected" algorithm that deletes the fact-less dimension values by traversing the hierarchy starting at the facts is invoked in a postprocessing step. For a hierarchy of height k, this can be done in time $\mathcal{O}(kn \log n)$, where n is the size of the mapping between facts and dimensions. Thus, the overall computational complexity is not altered.

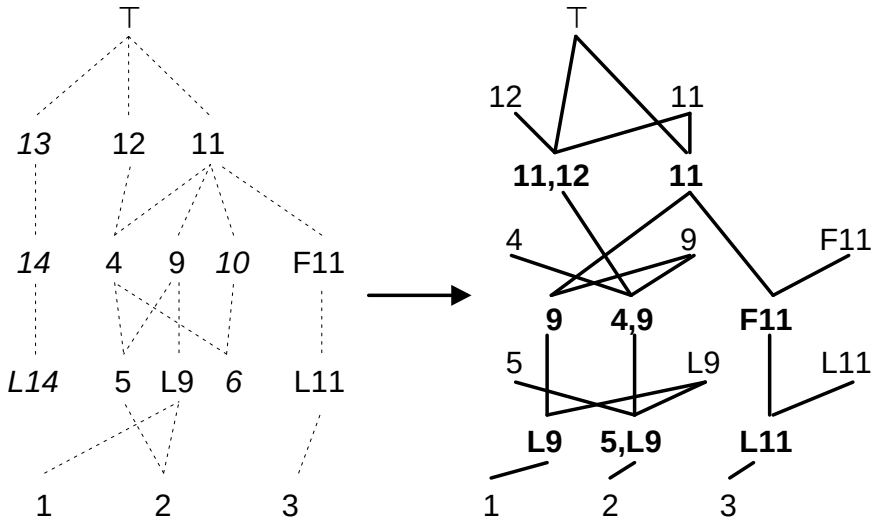

Figure 7: Transformations for Many-to-many Fact-Dimension Relationships

Example 10 The relationship between patients and diagnoses is many-to-many. In Example 9, the MO was transformed so that all mappings were covering, as seen in Figure 6; algorithm MakeStrict is applied to this MO. The final result of the application of the algorithm is seen to the right in Figure 7. Values in italics, e.g., *L14*, and dotted lines indicate deleted values and links. Bold-face values and thick lines denote values and links inserted by the algorithm.

Three new categories are introduced: "Set-of Low-level Diagnosis," "Set-of Diagnosis Family," and "Set-of Diagnosis Group," as non-strictness occurs at all levels. Fused values are inserted into these fused categories. For example, values "(low-level) Lung Cancer" (**L14**), "Insulin dependent diabetes during pregnancy, (low-level) Insulin dependent diabetes" (**5**, **L9**), and "(low-level) Insulin dependent diabetes" (**L9**) are inserted into the "Set-of Low-level Diagnosis" category; and the original values are linked to the new values.

Values "(low-level) Lung cancer" (**L14**), "Lung cancer" (14), "Cancer" (13), "Non insulin

dependent diabetes during pregnancy" (6), and "Non insulin dependent diabetes" (10) do not characterize any facts and are deleted by "Delete-unconnected."

6 Architectural Context

The overall idea presented in this paper is to take un-normalized MOs and transform them into normalized MOs that are well supported by the practical pre-aggregation techniques available in current OLAP systems. Queries are then evaluated on the transformed MOs. However, we still want the users to see only the original MOs, as they reflect the users' understanding of the domain. This prompts the need for means of handling both the original and the transformed MOs. This section explores this coexistence.

A current trend in commercial OLAP technology is the separation of the front-end presentation layer from the back-end database server. Modern OLAP applications consist of an OLAP client that handles the user interface and an OLAP server that manages the data and processes queries. The client communicates with the server using a standardized application programming interface (API), e.g., Microsoft's OLE DB for OLAP [17] or the OLAP Council's MDAPI [20]. The architecture of such a system is given to the left in Figure 8.

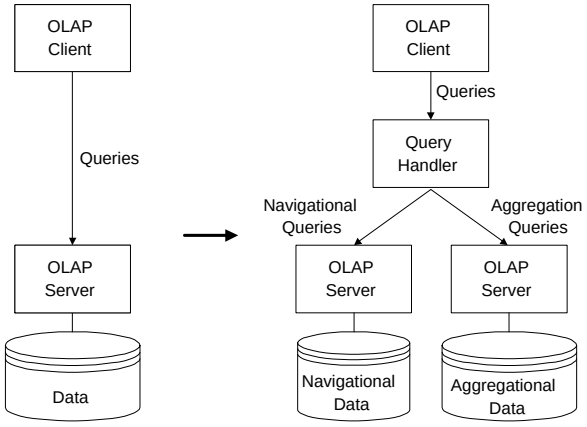

Figure 8: Architecture of Integration

This separation of client and server facilitates our desire to have the user see the original MO while queries are evaluated against the transformed MO. Studies have shown that queries on a data warehouse consist of 80% *navigational* queries that explore the dimension hierarchies and 20% *aggregation* queries that summarize the data at various levels of detail [14]. Examples of navigational and aggregation queries are "Show me the Low-Level Diagnoses contained in the Insulin-Dependent Diabetes Diagnosis Family" and "Show me the count of patients, grouped by Diagnosis Family," respectively. The navigational queries must be performed on the *original* MO, while the aggregation queries must be performed on the *transformed* MO. This is achieved by introducing an extra "Query Handler" component between the client and

the server. The OLAP client sends a query to the query handler, the primary task of which is to determine whether the query is a navigational query (internal to a dimension) or an aggregation query (involving the facts). Navigational queries are passed to one OLAP server that handles the original (navigational) data, while aggregation queries are passed to another OLAP server that manages the transformed (aggregation) data. This extended system architecture is seen to the right in Figure 8.

The OLAP server for navigation data needs to support dimension hierarchies which have non-summarizable properties, a requirement not yet supported by many commercial systems today. However, relational OLAP systems using snow-flake schemas [14] are able to support this type of hierarchies, as are some other OLAP systems, e.g., Hyperion (Arbor) Essbase [12]. If the OLAP system available does not have sufficiently flexible hierarchy support, one solution is to build a special-purpose OLAP server that conforms to the given API. This task is not as daunting as it may seem at first because only *navigational* queries need to be supported, meaning that multidimensional queries can be translated into simple SQL "lookup" queries.

We note that the only data needed to answer navigational queries is the hierarchy definitions. Thus, we only need to store the fact data (facts and fact-dimension relations, in our model) once, in the aggregational data, meaning that the overall storage requirement is only slightly larger than storing just the aggregational data. Navigational queries are evaluated on the original hierarchy definitions and do not need to be re-written by the query handler.

As described in Section 4, aggregation queries need to be re-written slightly by adding an extra HAVING clause condition to exclude results for the new values inserted by the transformation algorithms. This can easily be done automatically by the query handler, giving total transparency for the user. Even though the added HAVING clause conditions are only necessary for the covering and onto transformations, they can also be applied to hierarchies transformed to achieve strictness; this has no effect, but simplifies the query rewriting.

7 Conclusion and Future Work

Motivated by the increasing use of OLAP systems in many different applications, including in business and health care, this paper provides transformation techniques for multidimensional databases that leverage the existing, performance-enhancing technique, known as practical, or partial or semi-eager, preaggregation, by making this technique relevant to a much wider range of real-world applications.

Current pre-aggregation techniques assume that the dimensional structures are *summarizable*. Specifically, the mappings in dimension hierarchies must be *onto*, *covering*, and *strict*; the relationships between facts and dimensions must be many-to-one, and the facts must always be mapped to the lowest categories in dimensions. The paper presents novel transformation techniques that render dimensions with hierarchies that are non-onto, non-covering, and non-strict summarizable. The transformations have practically low computational complexity, they may be implemented using standard relational database technology, and the paper also describes how to integrate the transformed hierarchies in current OLAP systems, transparently to the user.

The paper also describes how to apply the transformations to the cases of non-summarizable

relationships between facts and dimensions, which also occur often in real-world applications. Finally, it is shown how to modify the algorithms to incrementally maintain the transformed hierarchies when the underlying data is modified.

Several directions for future research appear promising. The current techniques render the entire dimension hierarchies summarizable; extending the techniques to consider only the parts that have been selected for preaggregation appears attractive and possible. Another direction is to take into account the different types of aggregate functions to be applied, leading to local relaxation of the summarizability requirement. For example, *max* and *min* are insensitive to duplicate values, thus relaxing summarizability.

References

[1] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pp. 156–165, 1997.

[2] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. *Technical report, E.F. Codd and Associates*, 1993.

[3] S. Dar, H. V. Jagadish, A. Y. Levy, and D. Srivastava. Answering SQL Queries Using Views. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pp. 318–329, 1996.

[4] P. M. Deshpande, J. F. Naughton, K. Ramasamy, A. Shukla, K. Tufte, and Y. Zhao. Cubing Algorithms, Storage Estimation, and Storage and Processing Alternatives for OLAP. *IEEE Data Engineering Bulletin*, 20(1):3–11, 1997.

[5] C. E. Dyreson. Information Retrieval from an Incomplete Data Cube. In *Proceedings of the Twenty-Second Conference on Very Large Databases*, pp. 532–543, 1996.

[6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–54, 1997.

[7] A. Gupta, V. Harinarayan, and D. Quass. Aggregate Query Processing in Data Warehousing Environments. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pp. 358–369, 1995.

[8] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of he Thirteenth International Conference on Data Engineering*, pp. 208–219, 1997.

[9] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proceedings of the Sixth International Conference on Database Theory*, pp. 98–112, 1997.

[10] H. Gupta and I.S. Mumick. Selection of Views to Materialize Under a Maintenance-Time Constraint. In *Proceedings of the Seventh International Conference on Database Theory*, pp. 453–470, 1999.

[11] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 205–216, 1996.

[12] Hyperion Corporation. Hyperion Essbase OLAP Server. URL: <www.hyperion.com/downloads/essbaseolap.pdf>. Current as of February 17, 1999.

[13] Informix Corporation. Data Warehouse Administrator's Guide: MetaCube ROLAP Option for Informix Dynamic Server. URL: <www.informix.com/answers/english/pdf_docs/metacube/4189.pdf>. Current as of February 15, 1999.

[14] R. Kimball. *The Data Warehouse Toolkit*. Wiley Computer Publishing, 1996.

[15] R. Kimball. Data Warehouse Architect: Help with Multi-Valued Dimension. *DBMS Magazine*, 11(9), 1998.

[16] H. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Data Bases. In *Proceedings of the Ninth International Conference on Statistical and Scientific Database Management*, pp. 39–48, 1997.

[17] Microsoft Corporation. OLE DB for OLAP Version 1.0 Specification. Microsoft Technical Document, 1998.

[18] Microsoft Corporation. OLAP Services White Paper. URL: <www.microsoft.com/sql/70/whpprs/olapoverview.htm>. Current as of February 9, 1999.

[19] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 100–111, 1997.

[20] The OLAP Council. *MDAPI Specification Version 2.0*. OLAP Council Technical Document, 1998.

[21] The OLAP Report. *Database Explosion*. URL: <www.olapreport.com/DatabaseExplosion.htm>. Current as of February 10, 1999.

[22] T. B. Pedersen and C. S. Jensen. Multidimensional Data Modeling for Complex Data. In *Proceedings of the Fifteenth International Conference on Data Engineering*, 1999. Extended version available as TimeCenter Technical Report TR-37, URL: <www.cs.auc.dk/TimeCenter>, 1998.

[23] D. Quass and J. Widom. On-Line Warehouse View Maintenance for Batch Updates. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 393–404, 1997.

[24] M. Rafanelli and A. Shoshani. STORM: A Statistical Object Representation Model. In *Proceedings of the Fifth International Conference on Statistical and Scientific Database Management*, pp. 14–29, 1990.

[25] A. Segev and J. L. Zhao. Selective View Materialization in Data Warehousing Systems. *Working paper*, URL: <ftp://segev.lbl.gov/pub/LBL_DB_PUBLICATIONS/1997/aggreg-dw.ps>. Current as of February 9, 1999.

[26] A. Shukla, P. M. Deshpande, J. F. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pp. 522–531, 1996.

[27] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pp. 126–135, 1997.

[28] J. Widom. Research Problems in Data Warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management.*, pp. 25–30, 1995.

[29] R. Winter. Databases: Back in the OLAP game. *Intelligent Enterprise Magazine*, 1(4):60–64, 1998.

[30] World Health Organization. *International Classification of Diseases (ICD-10)*. Tenth Revision, 1992.

[31] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in a data warehousing environment. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pp. 136–145, 1997.

A Incremental Computation

When dimension hierarchies or fact data are updated, the transformed hierarchies must be updated correspondingly. One solution is to recompute the hierarchies using the new data. This straightforward solution is attractive when updating small dimension hierarchies that only change infrequently, or when large bulks of updates are processed. However, for massive hierarchies and frequent updates, and for updates of small parts of the hierarchies in general, it is desirable if the algorithms need only consider the *changed* parts of data, which will only be a small fraction of the total data volume. This section briefly describes how to incrementalize the algorithms.

In addition to modifying the transformed hierarchies, it is also necessary to update the actual pre-aggregated data when the underlying base data is modified. The modified hierarchies resulting from the algorithms given in this section differ only locally from the argument hierarchies. This means that the cost of updating the pre-aggregated data will not be greatly affected by the hierarchy transformations.

In the incremental algorithms, updates are modeled as deletions followed by insertions, so we consider only the latter two modification operations. We use prefix Δ_i to denote inserted

values, Δ_d to denote deleted values, and Δ to denote all modifications. For example, $\Delta_i C$ denotes the values inserted into C. The category and links tables in the algorithms refer to the states *after* modifications; and when a hierarchy value is deleted, all links to that value are also assumed to be deleted in the same set of modifications.

	Insert	Delete
C	yes	no
P	no	yes
H	no	yes
$R_{C,H}$	yes	no
$R_{C,P}$	no	yes
$R_{P,H}$	no	yes

Covering

	Insert	Delete
C	no	yes
P	yes	no
$R_{C,P}$	no	yes

Onto

	Insert	Delete
C	no	yes
P	no	yes
G	no	yes
$R_{C,P}$	yes	yes
$R_{P,G}$	yes	yes

Strict

Table 2: Effects of Insertions and Deletions on the Covering, Onto, and Strictness Properties

A.1 Covering Hierarchies

Modifications may render *covering* hierarchies non-covering in several ways. The the left-most table in Table 2, named "Covering" and discussed next, indicates whether an insertion ("Insert") or a deletion ("Delete") on the different parts of the input to MakeCovering may render the modified hierarchy non-covering.

Problems may arise if links are inserted into $R_{C,H}$ that are not covered by insertions into $R_{C,P}$ and $R_{P,H}$, or if links are deleted in $R_{C,P}$ or $R_{P,H}$, but the corresponding C-to-H links are not deleted in $R_{C,H}$. If values are deleted in P or H, their links will be deleted too, which is handled by the case above. Values cannot be inserted into C without any links, as all values in the original hierarchy must at least be linked to the $\top$ value.

The incremental version of MakeCovering algorithm starts by finding (in line (6)) the links L from C to H that are not covered by the links from C to P and P to H. These links are used as the base for the rest of the transformation. Thus, line (6) of the algorithm becomes the following expression.

$$L \leftarrow \Delta_i R_{C,H} \cup \Pi_{C,H}(\Delta_d R_{C,P} \bowtie R_{P,H}) \cup \Pi_{C,H}(R_{C,P} \bowtie \Delta_d R_{P,H}) \\ \setminus \Pi_{C,H}(\Delta_i R_{C,P} \bowtie \Delta_i R_{P,H}) \setminus \Delta_d R_{C,H}$$

A.2 Onto Hierarchies

The effects on the *onto* property of insertions and deletions are outlined in the middle table in Table 2. Insertion of values into P, deletion of values in C, and deletion of links in $R_{C,P}$ may cause the hierarchy to become non-onto. The incremental version of the MakeOnto algorithm

thus starts by finding (in line (4)) the "childless" values N from P with no children in C. As a result, line (4) of the algorithm becomes the following expression.

$$N \leftarrow \Delta_i P \cup \Pi_P(\Delta_D R_{C,P}) \setminus \Pi_P(\Delta_d P) \setminus \Pi_P(\Delta_i R_{C,P})$$

A.3 Strict Hierarchies

The case of maintaining the *strictness* property of hierarchies is more complicated because a new category N is introduced by the algorithm. We assume that all new categories have already been created before the incremental algorithm is used, i.e., if non-strictness is introduced in new parts of the hierarchy, we have to recompute the transformed hierarchy. The introduction of non-strictness requires major restructuring of both the hierarchy and the pre-aggregated data, so this is reasonable.

An overview of the effect on strictness of insertions and deletions in the input to algorithm MakeStrict is given in the right-most table in Table 2. If links are inserted into, or deleted from, $R_{C,P}$ or $R_{P,G}$, the links to N for the affected C, P, and G values must be recomputed.

Insertions into, or deletion from, C, P, or G will be accompanied by corresponding link insertions and deletions, so they are handled by the above case. The incremental MakeStrict, given below, works by finding the affected C, P, and G values, then recomputes their links to N and deletes the old links, and finally inserting the new links. As before, it is followed by a step that deletes the disconnected parts of the hierarchy.

(1) **procedure** IncrementalMakeStrict(C)
(2) **for** each $P \in Pred(C)$ such that $Pred(P) \neq \emptyset$ **do**
(3) **begin**
(4) $dC \leftarrow \Pi_C(\Delta R_{C,P})$
(5) $dR_{C,N} \leftarrow \{(c, Fuse(\{p \mid (c,p) \in dC \bowtie R_{C,P}\})\}$
(6) $dN \leftarrow \Pi_N(dR_{C,N})$
(7) $N \leftarrow N \cup dN$
(8) $R_{C,N} \leftarrow R_{C,N} \setminus \{(c,n) \mid c \in dC\} \cup dR_{C,N}$
(9) $dP \leftarrow \Pi_P(\Delta R_{C,P})$
(10) $dR_{N,P} \leftarrow \{(n,p) \mid n \in dN \wedge p \in dP \cap UnFuse(n)\}$
(11) $R_{N,P} \leftarrow R_{N,P} \setminus \{(n,p) \mid p \in dP\} \cup dR_{N,P}$
(12) **for** each $G \in Pred(P)$ **do**
(13) **begin**
(14) $dG \leftarrow \Pi_G(\Delta R_{P,G} \cup (dP \bowtie R_{P,G}))$
(15) $R_{N,G} \leftarrow R_{N,G}\ \{(n,g) \mid g \in dG\} \cup \Pi_{N,G}(R_{N,P} \bowtie R_{P,G} \bowtie dG)$
(16) **end**
(17) IncrementalMakeStrict(N)
(18) **end**
(19) **end**