An algebra for OLAP

Bart Kuijpers[a,*] and Alejandro Vaisman[b]

[a]*Databases and Theoretical Computer Science Research Group, UHasselt – Hasselt University and Transnational University Limburg, Agoralaan, 3590 Diepenbeek, Belgium* [b]*Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina*

Abstract. Online Analytical Processing (OLAP) comprises tools and algorithms that allow querying multidimensional databases. It is based on the multidimensional model, where data can be seen as a cube, where each cell contains one or more measures can be aggregated along dimensions. Despite the extensive corpus of work in the field, a standard language for OLAP is still needed, since there is no well-defined, accepted semantics, for many of the usual OLAP operations. In this paper, we address this problem, and present a set of operations for manipulating a data cube. We clearly define the semantics of these operations, and prove that they can be composed, yielding a language powerful enough to express complex OLAP queries. We express these operations as a sequence of atomic transformations over a fixed multidimensional matrix, whose cells contain a sequence of measures. Each atomic transformation produces a new measure. When a sequence of transformations defines an OLAP operation, a flag is produced indicating which cells must be considered as input for the next operation. In this way, an elegant algebra is defined. Our main contribution, with respect to other similar efforts in the field is that, for the first time, a formal proof of the correctness of the operations is given, thus providing a clear semantics for them. We believe the present work will serve as a basis to build more solid practical tools for data analysis.

Keywords: OLAP, data warehousing, algebra, data cube, dimension hierarchy

1. Introduction

Online Analytical Processing (OLAP) [9] comprises a set of tools and algorithms that allow efficiently querying multidimensional (MD) databases containing large amounts of data, usually called Data Warehouses (DW). Conceptually, in the MD model, data can be seen as a *cube*, where each cell contains one or more *measures* of interest, that quantify *facts*. Measure values can be aggregated along *dimensions*, which give context to facts. At the logical level, OLAP data are typically organized as a set of *dimension and fact tables.* Current database technology allows alphanumerical warehouse data to be integrated for example, with geographical or social network data, for decision making. In the era of so-called "Big Data", the kinds of data that could be handled by data management tools, are likely to increase in the near future. Moreover, OLAP and Business Intelligence (BI) tools allow to capture, integrate, manage, and query, different kinds of information. For example, alphanumerical data coming from a local DW, spatial data (e.g., temperature) represented as rasterized images, and/or economical data published on the semantic web. This wide variety of data types thus requires a BI user not only to know some OLAP language (or OLAP graphic tool), but also to deal with spatial data, and even with SPARQL (the standard query language for the semantic web). Ideally, this user would just like to deal with what she knows

Correspoding author: Bart Kuijpers, Databases and Theoretical Computer Science Research Group, UHasselt – Hasselt University and Transnational University Limburg, Agoralaan, Gebouw D, 3590 Diepenbeek, Belgium. E-mail: bart.kuijpers@uhasselt.be.

well, namely the data cube, using only the classical OLAP operators, like *Roll-up*, *Drill-down*, *Slice*, and *Dice* (among other ones), regardless the cube's underlying data type. Data types should only be handled at the logical and physical levels, not at the conceptual level. Building on this idea, Ciferri et al. [2] proposed a *conceptual*, *user-oriented* model, independent of technologies like ROLAP (for Relational OLAP), MOLAP (for Multidimensional OLAP) or HOLAP (for Hybrid OLAP). In this model, the user only manipulates a data cube. Associated with the model, there is a query language providing high-level operations over the cube. This language, called Cube Algebra, was sketched informally in the mentioned work. Extensive examples on the use of Cube Algebra were presented in [?], suggesting that this idea can lead to a language much more intuitive and simple than MDX [8], the *de facto* standard for OLAP. Nevertheless, these works do not give any evidence of the correctness of the languages and operations proposed, other than examples at various degrees of comprehensiveness. In fact, surprisingly, and in spite of the extensive corpus of work in the field, a formally-defined reference language for OLAP is still needed [11]. There is not even a well-defined, accepted semantics, for many of the usual OLAP operations. We believe that, far for being just a problem of classical OLAP, this formalization is also needed in current "Big Data" scenarios, where there is a need to efficiently perform real-time OLAP operations [3], that, of course, must be well defined.

1.1. Contributions

In this paper we address the problem described above. To this end, we:

- introduce a collection of operators that manipulate a data cube, and clearly define their semantics; and
- prove, formally, that our operators can be composed, yielding a language powerful enough to express complex queries and cube navigation ("*à la* OLAP") paths.

We achieve the above representing the data cube as a fixed d-dimensional matrix, and a set of k measures, and expressing each OLAP operation as a sequence of atomic transformations. Each transformation produces a new measure, and, additionally, when a sequence forms an OLAP operation, a flag that indicates which are the cells that must be considered as input for the next operation. This formalism allows us to elegantly define an algebra as a collection of operations, whose proof of correctness we provide in the paper. In this paper we limit ourselves to the most usual operations, namely slice, dice, roll-up and drill-down, which constitute the core of all practical OLAP tools. We denote these the *classical OLAP operations*. This allows us to focus on our main interest, which is, to prove the feasibility of the approach. Other not-so-usual operations, and operations between two or more cubes, are left for future work.

The main contribution of our work, with respect to other similar efforts in the field is that, for the first time, a formal proof to practical problems is given, so the present work will serve as a basis to build more solid tools for data analysis. As we show in the next section, existing work either lacks of formalism, or of applicability, and no work of any of these kinds give sound mathematical prove of its claims.

1.2. Paper organization

The remainder of the paper is organized as follows. In Section 2, we review and discuss related work. In Section 3, we present our MD data model, on which we base the rest of our work. Section 4 presents the atomic transformations that we use to build the OLAP operations. In Section 5 we discuss the classical OLAP operations in terms of the transformations, show how they can be composed to address complex queries, and give proofs of all of our claims. We conclude in Section ??. Additional proofs are given in the appendix.

2. Related work

Although the need of an algebra for OLAP has long been acknowledged in the literature (see for example [11]), just a few works have addressed this problem so far, and in a limited way. Further, most of the proposals have addressed this issue from a logical point of view, rather than from a *conceptual* one. Conceptually, one would like to define an algebra that can be intuitive for typical OLAP users. These users know how to manipulate and query data cubes. With this idea in mind, Viswanathan and Schneider [?] proposed what they called a user-centric approach for modeling MD data. As a sequel, the authors presented a query language based entirely on the abstract data cube metaphor. They called this language CAL (standing for Cube Analysis Language). CAL includes a Cube Definition Language (CDL), and a Cube Manipulation Language (CML). Similar ideas were presented in [2], where, as we said, a Cube Algebra was proposed. Actually, CAL and Cube Algebra suggest similar OLAP operations. However, and like in all proposals for query languages for OLAP, no formal definition of the semantics of the operations is presented. Although our work is based on the proposals above, we next review other approaches to querying MD databases.

The MD model proposed by Gyssens and Lakshmanan [7] defines a data manipulation language that can express a so-called cube operator. The authors propose an algebra (and an equivalent calculus), which includes set operations (like selection, projection, Cartesian product), operations for summarization, and for re-structuring (fold and unfold). This model largely simplifies typical MD models for OLAP (for example, dimension hierarchies are considered in a very limited way), and the operations only address simple cases rather than complex, real-world queries.

Along similar lines, Agrawal et al. [1] proposed a data model that supports multiple hierarchies along each dimension, and the possibility of performing ad-hoc aggregates. They also define a minimal set of algebraic operators that is composed of the following ones: push and pull, destroy dimension, restriction (slice and dice), join, and associate. These operations are introduced in an informal way.

In real-world scenarios, the composition of basic operations is usually required (e.g., we would need to roll-up to a certain level in a dimension hierarchy, and then select a portion of the result). The proposals above do not completely address this problem. On the contrary, we show that our approach can be applied to address typical operations composition, since our algebra is closed: all operations receive a matrix and return a matrix.

Macedo and Oliveira [10] presented an approach that can be considered close to ours, since they represent MD data as a matrix, with the idea of expressing OLAP operations in linear algebra (LA). The paper, like in our case, is aimed at filling the theoretical gap in the field. The proposal expresses some simple OLAP operations, mainly cross tabulations, as a combination of matrix multiplication, transposition, and a variant of the Kronecker product. However, it is very preliminary, as the authors acknowledge, and no justification of the approach is provided. Further, as it is, the work is oriented to Excel spreadsheets rather than to OLAP, and it has yet to be incorporated into a typical MD model for OLAP.

With a more OLAP-oriented flavor, Vassiliadis [?] presented a classic MD model, which includes the concepts of dimensions, hierarchies, and cubes. The author also proposes a set of operations using the notion of *base cube*, e.g., a cube defined at the finest granularity level. These operations are: level climbing, packing, function application, projection, navigation, slicing, and dicing. Again, no formal language is presented. To overcome these drawbacks, Ravat et al. [5] also proposed an OLAP algebra at the conceptual level. Again, no formal semantics is defined for the algebra presented by the authors. Stolte et al. [?] introduced Polaris, an interface for exploring large MD databases, based on the Pivot table paradigm. The tool makes use of a table algebra as a formal mechanism to specify table configurations. This algebra includes operations allowing performing the union and Cartesian product of tables, as well as a nesting operation. Again, the proposal lacks of a formal proof of the correctness of the operations, or a formal definition of their semantics.

Some works have already made use of the cube algebra proposed in [2]. Gómez el al. [6] used cube algebra to manipulate different kinds of spatial data: discrete spatial data, and continuous fields implemented in several different ways, like Voronoi diagrams and rasterized data. They implemented Cube Algebra at a conceptual level, and all the machinery needed to manipulate the heterogeneous cubes at the logical and physical levels. Similar work, but with semantic web data, is presented in [?]. We envision that, given the "Big Data" wave, applications like these are likely to grow in number and variety. Thus, the definition of an algebra at the conceptual level (i.e., based on the data cube metaphor), is clearly needed. Along the same lines, Viswanathan and Schneider [?] extended their proposal to spatial OLAP.

We remark once more that from all the works discussed above, only CAL and Cube Algebra are the ones that are defined from a user-centric point of view, at a *conceptual* level, providing high-level query operations for the user. For a comprehensive survey, we refer the reader to [2], where a deep study that shows that none of the existing approaches consider the data cube as a first-class citizen of the MD model.

3. The OLAP data model

In this section we describe the OLAP data model, based on the MD data cube. We first define the notion of MD matrix. In our model, the "empty" matrix serves as a placeholder for the measures in the data cube. We also define the notions of dimension schema (with hierarchies and levels) and dimension instance (level instance, hierarchy instance and dimension graph). We conclude with a discussion of ordered domains and the representation of higher-level objects.

3.1. Multidimensional matrix

We next give the definitions of multidimensional matrix schema and instance. Throughout this paper, d is a natural number, with $d > 1$, which represents the number of dimensions of a data cube.

Definition 1 (Matrix Schema). A *d-dimensional matrix schema* is a sequence $(D_1,D_2,\dots,D_d)$ of d dimension names. □

Dimension names can be considered to be strings. As illustrated in the following example, the convention will be that dimension names start with a capital letter.

Example 1. The running example we use throughout the paper, deals with sales information of certain products, at certain locations, at certain moments in time. For this purpose, we will define a 3dimensional matrix schema (D_1,D_2,D_3) = (*Product, Location, Time*). □

Definition 2 (Matrix Instance). A *d-dimensional matrix instance* (*matrix*, for short) over the $d$dimensional matrix schema $(D_1,D_2,\ldots,D_d)$ is the product

$$dom(D_1) \times dom(D_2) \times \cdots \times dom(D_d),$$

$i = 1,2,\ldots,d$, where $dom(D_i)$ is a non-empty, finite, ordered set, called the *domain*, that is associated with the dimension name D_i. For all $i = 1,2,\ldots,d$, we denote by $<$, the order that we assume on the elements of $dom(D_i)$. For $a_1 \in dom(D_1), a_2 \in dom(D_2),\ldots,a_d \in dom(D_d)$, we call the tuple $(a_1,a_2,\ldots,a_d)$, a *cell* of the matrix. □

The cells of a matrix serve as placeholders for the measures that are contained in the data cube (see Definition 7 below). Note that, as it is common practice in OLAP, we assumed an order < on the domain. The role of the order is further discussed in Section 3.4.

As a notational convention, elements of the domains $dom(D_i)$ start with a lower case letter, as it is shown in the following example.

Example 2. For the 3-dimensional matrix schema (D_1,D_2,D_3) = (*Product, Location, Time*) of Example 1, the non-empty sets $dom(D_1)$ = {*lego, brio, apples, oranges*}, $dom(D_2)$ = {*antwerp, brussels, paris, marseille*}, and $dom(D_3)$ = {1/1/2014,…,31/1/2014} produce the matrix instance $dom(D_1) \times dom(D_2) \times dom(D_3)$.

This matrix is depicted in Fig. 3. The cells of the matrix contain the sales for each combination of values in the domain. In $dom(D_2)$, we have, for instance, the order *antwerp* < *brussels* < *paris* < *marseille.* Over the dimension *Time*, we have the usual temporal order.

3.2. Level instance, hierachy instance and dimension graph

We now define the notions of dimension schema (with hierarchies and levels), and dimension graph (or dimension instance).

Definition 3 (Dimension Schema, Hierarchy and Level). Let D be a name for a dimension. A *dimension schema* $\sigma(D)$ *for* D is a lattice, with a unique top-node, called *All* (which has only incoming edges) and a unique bottom-node, called *Bottom* (which has only outgoing edges), such that all maximal-length paths in the graph go from *Bottom* to *All*. Any path from *Bottom* to *All* in a dimension schema $\sigma(D)$ is called a *hierarchy* of $\sigma(D)$. Each node in a hierarchy (i.e., in a dimension schema) is called a *level* (of $\sigma(D)$). □

We remark that a lattice is always a partial order set.

As a notational convention, level names start with a capital letter. Note that the *Bottom* node is often renamed, depending on the application, as is illustrated in the following example. This example also introduces a non-graphical notation for hierarchies.

Example 3. Figure 1 gives examples of dimension schemas $\sigma(Location)$ and $\sigma(Time)$ for the dimensions *Location* and *Time* in Example 1.

For the dimension *Location*, we have *Bottom* = *City*, and there is only one hierarchy, which we denote as

City → *Region* → *Country* → *All*.

The node *Region* is an example of a level in this hierarchy.

For the dimension *Time*, we have *Bottom* = *Da*, and two hierarchies, namely *Day* → *Month* → *Semester* → *Year* → *All* and *Day* → *Week* → *All*.

We remark that for the dimension *Location*, we have a linear lattice as a dimension schema. In this example, this is not the case for the dimension *Time*. □

Definition 4 (Level Instance, Hierachy Instance, Dimension Graph). Let D be a dimension with schema $\sigma(D)$, and let ℓ be a level of $\sigma(D)$. A *level instance of* ℓ is a non-empty, finite set *dom*$(D.\ell)$. If ℓ = *All*, then *dom(D.All)* is the singleton $\{all\}$. If ℓ = *Bottom*, then *dom(D.Bottom)* is the the domain of the dimension D, that is, *dom*(D) (as in Definition 2).

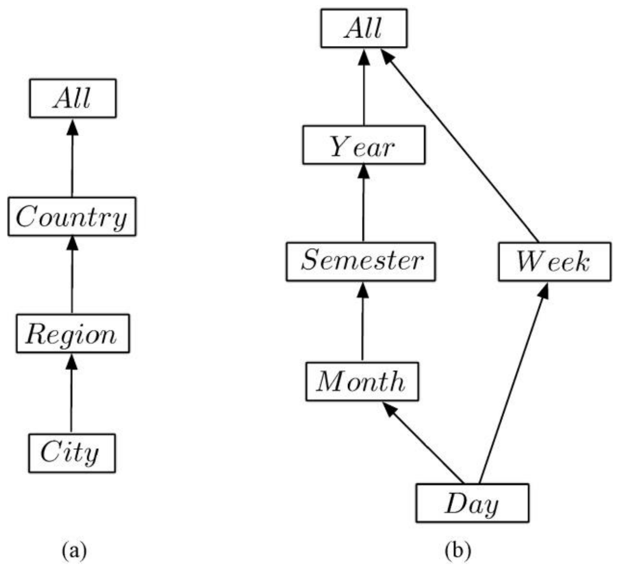

Fig. 1. Dimension schemas for the dimensions *Location*, in (a), and *Time* , in (b).

A *dimension graph (or instance)* $I(\sigma(D))$ over the dimension schema $\sigma(D)$ is a directed acyclic graph with node set

$$\bigcup_{\ell} dom(D.\ell),$$

where the union is taken over all levels in $\sigma(D)$. The edge set of this directed acyclic graph is defined as follows. Let ℓ and ℓ^0 be two levels of $\sigma(D)$, and let $a \in dom(D.\ell)$ and $a^0 \in dom(D.\ell^0)$. Then, only if there is a directed edge from ℓ to ℓ^0 in $\sigma(D)$, there can be a directed edge in $I(\sigma(D))$ from a to a^0.

If H is a hierarchy in $\sigma(D)$, then the *hierarchy instance* (relative to the dimension instance $I(\sigma(D))$) is the subgraph of $I(\sigma(D))$ with nodes from *dom*$(D.\ell)$, for ℓ appearing in H. This subgraph is denoted $I_H(\sigma(D))$. □

As notational convention, the names of objects in a set $dom(D.`)$ start with a lower case character.

We remark that a hierarchy instance $I_H(\sigma(D))$ is always a (directed) tree, since a hierarchy is a linear lattice. We also use the following terminology. If a and b are two nodes in a hierarchy instance $I_H(\sigma(D))$, such that (a,b) is in the transitive closure of the edge relation of $I_H(\sigma(D))$, then we say that a *rolls-up* to b and we denote this by $\rho_H(a,b)$ (or $\rho(a,b)$ if H is clear from the context). The following example illustrates these concepts.

Example 4. We continue with Example 3, and focus on dimension *Location*, whose schema $\sigma(Location)$, is given in Fig. 1 (a). From Example 2, we have *dom(Location)* = {*antwerp, brussels*, *paris, marseille*}, which is *dom(Location.Bottom)*, or *dom(Location.City)*.

For the levels *Region* and *Country*, we have *dom(Location.Region)* = {*flanders, capital, north, south*}, and *dom(Location.Country)* = {*belgium, france*}, respectively. An example of a dimension instance $I(\sigma($*Location*$))$ is depicted in Fig. 2. This example expresses, for instance, that the city *brussels* is located in the region *capital* which is part of the country *belgium*, meaning that *brussels* rolls-up to *capital* and to *belgium*, that is, $\rho($*brussels, captial*$)$ and $\rho($*brussels, belgium*$)$. Note that the dimension instance of Fig. 2 is indeed a tree. □

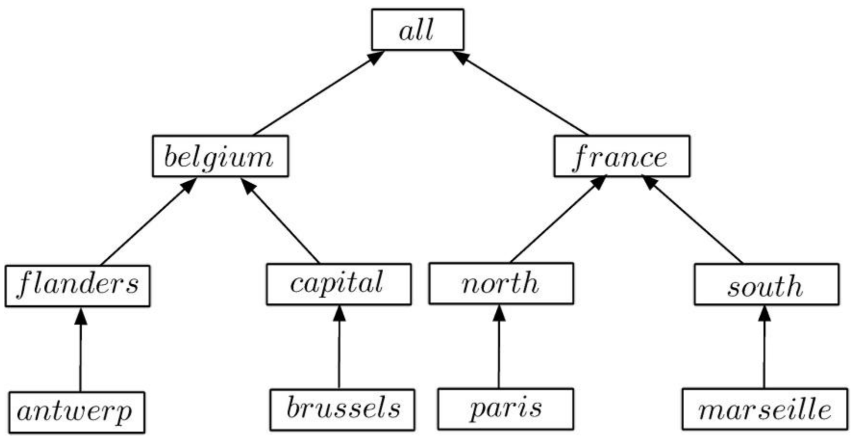

Fig. 2. An example of a dimension graph (or instance) $I(\sigma($*Location*$))$.

In a dimension graph with multiple hierarchies, elements in some levels may be reachable from elements in the *Bottom* level, in multiple ways. However, it is important to guarantee that rolling-up in different ways (i.e., through different paths) gives the same results. This is formalized by the concept of "sound" dimension graph.

Definition 5 (Sound Dimension Graph). Let $I(\sigma(D))$ be a dimension graph (as in Definition 4). We call this dimension graph *sound*, if for any level ` in $\sigma(D)$ and any two hierarchies $H_1$ and $H_2$ that reach ` from the *Bottom* level and any $a \in$ *dom (D)* and $b_1, b_2 \in$ $dom(D.`)$, we have that $\rho_{H1}(a,b_1)$ and $\rho_{H2}(a,b_2)$ imply that $b_1 = b_2$. □

In this paper, we assume that dimension graphs are always sound (as specified in Definition 7). Note that this poses an integrity constraint over the dimension instances. This constraint, together with the ones implied in the model's definition, are typical in so-called balanced (or homogeneous) dimensions [?]. We use this assumption in Property 1, at the end of Section 3.4.

3.3. Multidimensional data cube

In this section we define the concepts of MD data cube schema and instance. Essentially, a data cube is a matrix in which the cells are filled with measures that are taken from some *value domain* Γ. For many applications, Γ will be the set of real or rational numbers. But we may also think of applications where Γ includes spatial regions or other geometric objects, for instance.

Definition 6 (Data Cube Schema). A *d-dimensional data cube schema* consists of

- a d-dimensional matrix schema $(D_1,D_2,\ldots,D_d)$; and
- a hierarchy schema $\sigma(D_i)$ for each dimension D_i, with $i = 1,2,\ldots,d$. □

Definition 7 (Data Cube Instance). Let Γ be a non-empty set of "values". A *d-dimensional, k-ary data cube instance* (or *data cube*, for short) $\mathcal{D}$ over the d-dimensional matrix schema $(D_1,D_2,\ldots,D_d)$ and hierarchy schemas $\sigma(D_i)$ for D_i, for $i = 1,2,\ldots,d$, with values from Γ, consists of

- a d-dimensional matrix instance over the matrix schema $(D_1,D_2,\ldots,D_d)$, denoted $M(\mathcal{D})$;
- for each $i = 1,2,\ldots,d$, a *sound* dimension graph $I(\sigma(D_i))$ over $\sigma(D_i)$;
- *k measures* $\mu_1,\mu_2,\ldots,\mu_k$, which are functions from *dom*$(D_1)\times$*dom*$(D_2)\times\cdots\times$*dom*(D_d) to the value domain Γ; and

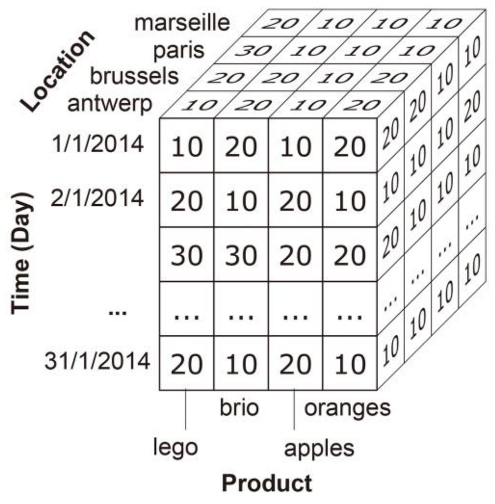

Fig. 3. An example of a data cube with one measure: μ_1 = *sales*.

- a *flag* ϕ , which is a function from *dom*(D_1) × *dom*(D_2) × ⋯ × *dom*(D_d) to the set $\{0,1\}$. □

In the remainder of this paper we assume that $\Gamma = \mathbf{Q}$, the set of the rational numbers. For most applications, this suffices. Also, as a notational convention, we use calligraphic characters, like $\mathcal{D}$, to represent data cube instances.

The flag ϕ can be considered as a $(k+1)$-st Boolean measure. The role of ϕ is to indicate which of the matrix cells are currently "active". The active cells have a flag value 1 and the others have a flag value 0. When we operate over a data cube, flags are used to indicate the input or output parts of the matrix of the cube. Typically, in the beginning of the operations, all cells have a flag value of 1. The role of flags will become more clear in the next sections, when we discuss OLAP transformations and operations.

Example 5. We build on the previous examples. Figure 3 shows a 3-dimensional 1-ary data cube instance over the matrix schema (*Product, Location, Time*), and dimension schemas σ(*Product*), σ(*Location*), and σ(*Time*) (two of which were given in Example 3), with the set of the rational numbers as value domain. The matrix cells contain one measure, namely μ_1 = *sales*, which expresses the sales amount per product, per location and per time instant. Initially, the flag ϕ may be, for instance, 1 for all matrix cells (not indicated in Fig. 3), telling that all cells of the matrix are currently active. □

3.4. Ordered domains and the representation of higher-level objects

In the process of performing OLAP transformations and operations, we may need to store aggregate information about certain measures up to some level above the *Bottom* one. We do not want to use extra space for this in the data cube. Instead, we use the available cells of the original data cube to store this aggregate information, yielding a more elegant solution, since this allows us to manipulate always the same cube schema while we perform a sequence of operations over the cube, as we will see later.

Recall that in Definition 2 we have assumed an order < for the domains $dom(D_i)$. We make use of this order for the representation of high-level objects by *Bottom*-level objects. The following definition specifies how this is achieved.

Definition 8. Let $D \in \{D_1, D_2, \ldots, D_d\}$ be an arbitrary dimension with domain *dom(D) = dom(D.Bottom)*. Let ` be a level of $\sigma(D)$. An element $b \in dom(D.`)$ is *represented* by the smallest element $a \in dom(D)$ (according to <) for which $\rho(a,b)$ holds. We denote this as $rep(b) = a$, and say that a *represents* b. □

We remark that, since we assume dimension graphs to be sound, this notion of representation is well defined, because the smallest element of the bottom level will always reach the same element in any level, regardless of the path traversed. The following example illustrates the concept of representation.

Example 6. Continuing with the previous examples, we consider the dimension $Location$ with *dom(Location)* = {*antwerp, brussels, paris, marseille*}. Note that this is actually *dom(Location.City)*. On this set, we *assume* the order *antwerp < brussels < paris < marseille*. For this dimension, we have the hierarchy *City* → *Region* → *Country* → *All*, and the dimension instance $I(\sigma($*Location*$))$, given in Fig. 2.

At the *Bottom* = *City* level, cities represent themselves. At higher levels, regions and countries are represented by their "first" city in *dom(Location)* (according to <). Thus, *flanders* and *belgium* are represented by *antwerp*, *france* is represented by *paris*, and *south* is represented by *marseille*. Let us explain further explain this. For the level *Region*, we have *dom(Location.Region)* = {*flanders, capital, north, south*}. At this level, *antwerp* represents *flanders* and *marseille* represents *south*, since they are the first (and, in this case only) domain elements that roll-up to these regions. So, we have *rep(flanders)* = *antwerp*. For the level *Country*, we have *dom(Location.Country)* = {*belgium, france*}. At this level *antwerp* represents *belgium* and *paris* represents *france*, since they are the first (but not only) domain elements that roll-up to these countries. At the level *All*, *antwerp* represents *all.* □

We will use the convention above, to encode outputs of OLAP transformations and operations at different levels. That means, in our running example, if we want to represent an output at the *Country* level, we will flag *antwerp* and *paris* to represent *belgium* and *france*. The idea is to store aggregate information for higher-level objects in the cells of their *Bottom*-level representatives. In an output cube that contains this aggregate information, we have these representatives flagged 1 and other cells flagged 0. We remark once more, that, in this way, we do not need extra cells in the matrix in order to represent aggregate information.

Remark 1. In the previous example we can see that, if we aggregate information at the level *Region*, with *dom(Location.Region)* = {*flanders, capital, north, south*}, then all cities of *dom(Location)* = {*antwerp, brussels, paris, marseille*} become flagged. At this point, it would not be clear if the cube contains information at the level *City* or at the level *Region*. A practical solution would be to keep a log of the OLAP operations that are performed, making the level of aggregation clear. □

The following property shows how the order on the *Bottom* level induces and order on higher levels. This property depends on the soundness of the dimension graph. Its proof is straightforward and we omit it.

Property 1. Let $D \in \{D_1, D_2, \ldots, D_d\}$ be a dimension of a data cube D and let ℓ be a level in the dimension schema $\sigma(D)$. The order < on *dom(D)* induces an order (also denoted <) on *dom*$(D.\ell)$ as follows. If $b_1, b_2 \in$ *dom*$(D.\ell)$, then $b_1 < b_2$ if and only if *rep*(b_1) < *rep*(b_2). □

4. OLAP transformations and operations

A typical OLAP user manipulates a data cube by means of well-known operations. Just to mention the most popular ones, *Roll-Up* aggregates measures up to a certain level in a dimension, *Drill-Down* disaggregates measures down to a certain level in a dimension, *Slice* drops a whole dimension, and *Dice* keeps only the cells in a cube satisfying a certain Boolean condition. These operations, which we will formally define later in this paper, actually express queries over the data cube, typically submitted using some graphic tool, and translated into an underlying query language. The result can then be displayed in graphic or tabular format. An OLAP query can then be considered as a sequence of these individual operations, which receive a cube as input, and return a cube as output. For instance, using our running example, an apparently simple query like “Total sales by region, for regions in Belgium or France”, is actually expressed as a sequence of operations, whose semantics should be clearly defined, and which can be applied in different order (with the same result). For example, we can first apply a *Roll-Up* to the *Country* level, and once at that level apply a *Dice* operation, which keeps the cube cells corresponding to Belgium or France. Finally, a *Drill-Down* disaggregates the sales down to the level *Region*, returning the desired result. Note that since the sales not occurred in Belgium and France have been eliminated, this last operation must only consider the remaining members in *Country*. Thus, the *Drill-Down* operation is not a just an undo of the previous *Roll-Up*, as it is sometimes considered to be. In day-to-day practice, this problem appears regardless of the querying interface, that is, whether the operation sequence is submitted as an expression in a query language, or processed during the user’s navigation through a graphic tool.

In what follows, we regard OLAP operations as the result of sequences of “atomic” OLAP transformations, which are measure-creating updates to a data cube. First, we give the definition of an OLAP transformation. Next, we show how these transformations can be combined to define OLAP operations, and how these OLAP operations can be composed. Finally, we give an overview of our arsenal of atomic OLAP transformations.

4.1. Introduction to OLAP transformations and operations

An *atomic OLAP transformation* acts on a data cube instance, by adding a measure to the existing data cube measures. OLAP operations like the ones informally introduced above are defined, in our approach, as a sequence of transformations. The process of OLAP transformations starts from a given *input data cube* D_{in}. We assume that this original data cube has k given measures $\mu_1,\mu_2,\dots,\mu_k$ (as in Definition 7). These k measures have a special status in the sense that they are "protected" and can never be altered (see Section 4.3). However, there is one exception to this protection. These original measures can be "destroyed" in some cells (see further on and Section 4.2), for instance, as the result of slice or dice operations, which are destructive by nature. Operations of these types destroy the content of some matrix cells and remove even the protected measures in it.

Typically, the input-flag ϕ of the original data cube D_{in} is set to 1 in every cell and signals that every cell of $M(\mathrm{D}_{in})$ is part of the input cube.

Atomic OLAP transformations can be applied to data cubes. They add (or create) new measures to the sequence of existing measures by adding new measure values in each cell of the data cube's matrix. At any moment in this process, we may assume that the data cube D has $k + l$ measures $\mu_1,\mu_2,\dots,\mu_k;\tau_1,\dots,\tau_l$, where the first k are the original measures of D_{in}, and the last l (with $l > 0$) ones have been created subsequently by l OLAP transformations (where $\tau_1,\dots,\tau_l$ is the empty sequence of τ's, for $l = 0$). The next OLAP transformation adds a new measure τ_{l+1} to the matrix cells.

We have said that we use OLAP transformations to compute OLAP operations. In this sense, we can see that many of the measures $\tau_1,\tau_2,\dots,\tau_l$ added by the transformations in a process, may represent the result of intermediate computations that are not really relevant to the output of an OLAP *operation*. We indicate that the computation of an OLAP operation O is finished by creating an m-ary output flag $\varphi_O^{(m)}$. This output flag is a Boolean measure, that is created like other measures, via atomic OLAP transformations. It indicates which of the cells of $M(\mathrm{D})$ should be considered as belonging to the output of O. It is m-ary in the sense that it keeps the last m created measures $\tau_{l-m+1},\tau_{l-m+2},\dots,\tau_l$ and "trashes" $\tau_1,\tau_2,\dots,\tau_{l-m}$. It also removes the previous flag, which it replaces. The initial measures $\mu_1,\mu_2,\dots,\mu_k$ of the input data cube D_{in} are never removed (unless they are "destroyed" in some cells). They are "protected" and remain in the cube throughout the process of applying one OLAP operation after another to D_{in}. So, at any stage, we can use the given measures $\mu_1,\mu_2,\dots,\mu_k$ (except in destroyed cells).

Summarizing the above, after an OLAP operation of output arity m is completed on some cube D, the measures in the cells of the output data cube $\mathrm{D}^0 = O(\mathrm{D})$ are of the form

$$\underline{\mu_1,\mu_2,\dots,\mu_k};\tau_{l-m+1},\tau_{l-m+2},\dots,\tau_l;\varphi_O^{(m)}.$$

In the previous expression, the underlining indicates the protected status of these measures. After each OLAP operation, we do a "cleaning" by renaming the unprotected measures with the symbols $\tau_1,\tau_2,\dots,\tau_m$ and the output measures become

$$\underline{\mu_1,\mu_2,\dots,\mu_k};\tau_1,\tau_2,\dots,\tau_m;\varphi_O^{(m)}.$$

The next OLAP operation O^0 can then act on D^0 and use in its computation all the measures $\underline{\mu_1,\mu_2,\dots,\mu_k};\tau_1,\tau_2,\dots,\tau_m;\varphi_O^{(m)}$. When O^0 finishes its computation after adding l^0 measures $\tau_{m+1},\tau_{m+2},\dots,\tau_{m+l^0}$ and producing an m^0-ary output, the new measures in the cells will look like

$\underline{\mu_1, \mu_2, \ldots, \mu_k}; \tau_{m+l'-m'+1}, \tau_{m+l'-m'+2}, \ldots, \tau_{m+l'}; \varphi_{O'}^{(m')}$.

The last m^0 measures before the flag are renamed $\tau_1, \tau_2, \ldots, \tau_{m^0}$, again. In this way, the composition of OLAP operations should be viewed.

We remark that the dimensions, the hierarchy schemas and instances of D remain unaltered during the entire OLAP process.

We end this description of our view of OLAP transformations, OLAP operations and their composition, with a remark on *destructors*. Destructors are similar to flags, in the sense that they are computed by some sequence of atomic OLAP transformations and that they are Boolean. A destructor, optionally, precedes the creation of an output flag. A destructor δ takes the value 1 for some cells of the matrix of a data cube, and 0 on other cells. When δ is invoked (and activated by the output flag that follows it) on a data cube D with measures $\underline{\mu_1, \mu_2, \ldots, \mu_k}; \tau_1, \tau_2, \ldots, \tau_m$ and flag $\varphi_O^{(m)}$, it empties all cells for which the value of the destructor δ is 0 by removing all measures from them, even the protected ones, thereby effectively "destroying" these cells. This is the only case where the protected measures are altered. For example, this happens when the OLAP operation is a slice or a dice. Operations of these types destroy part of the cube and make them inaccessible for further use. In this context, the output of a destructive operation O looks like

$\underline{\mu_1, \mu_2, \ldots, \mu_k}; \tau_1, \tau_2, \ldots, \tau_l; \delta; \varphi_O^{(m)}$,

in which the destructor precedes the output flag. The effect of the presence of a destructor is the following. A cell such that $\delta = 0$ is emptied, after which it contains no more measures and flag. For cells with $\delta = 1$, the sequence of measures $\underline{\mu_1, \mu_2, \ldots, \mu_k}; \tau_1, \tau_2, \ldots, \tau_l; \delta; \varphi_O^{(m)}$; is transformed to

$\underline{\mu_1, \mu_2, \ldots, \mu_k}; \tau_{l-m+1}, \tau_{l-m+2}, \ldots, \tau_l; \varphi_O^{(m)}$; which is renamed as $\underline{\mu_1, \mu_2, \ldots, \mu_k}; \tau_1, \tau_2, \ldots, \tau_m; \phi$; before the next transformation takes place. This transformation will act, cell per cell, on the matrix of a cube, with the understanding that it does nothing with emptied cells. That is, no new measure can ever be added to a destroyed cell.

4.2. OLAP transformations

The following definition specifies how an OLAP transformation acts on a data cube. We then address in detail each atomic OLAP transformation appearing in this definition.

Definition 9 (OLAP transformation). Let D be a d-dimensional, $(k + l)$-ary data cube instance with given (or protected) measures $\mu_1, \mu_2, \ldots, \mu_k$, created measures $\tau_1, \ldots, \tau_l$ (with $l > 0$) and flag ϕ over some value domain Γ. An *OLAP transformation* T, applied to D, results in the creation of a new measure τ_{l+1} in D. The transformation T adds the measure τ_{l+1} to non-empty cells of $M(\mathrm{D})$. The measure τ_{l+1} is produced from

- $\mu_1, \mu_2, \ldots, \mu_k$ (in non-empty cells);
- ϕ (in non-empty cells);
- $\tau_1, \tau_2, \ldots, \tau_l$ (in non-empty cells) and
- the hierarchy schemas and instances of D and belongs to one of the following classes:

1. Arithmetic transformations (see Definition 11);
2. Boolean transformations (see Definition 12);
3. Selectors (see Definition 13);
4. Counting, sum, and min-max (see Definitions 14 and 19);

5. Grouping (see Definition 18).

An OLAP transformation can also result in the creation of a measure that is an output flag $\phi^{(m)}$ of arity m. This should be a measure with a Boolean value. In order to indicate that it is a flag of arity m, we use the reserved symbol $\phi^{(m)}$ instead of τ_{l+1}. An output flag $\phi^{(m)}$ may (optionally) be preceded by a destructor δ (which is created following the same rules as for other measures, but which has a special status, expressed by the reserved symbol δ). This should be a measure with a Boolean value (to indicate which cells are destroyed). We use the reserved symbol δ instead of τ_{l+1}. □

The effect of output flags and destructors was discussed in Section 4.1. We remark that atomic OLAP transformations update the cells of the matrix $M(\mathrm{D})$ cell per cell and that empty cells of $M(\mathrm{D})$ are unaffected by transformations.

4.3. OLAP Operations and their composition

Before we give the definition of an OLAP operation, we describe the *input* to the OLAP process (this process may involve multiple OLAP operations). Such input is a d-dimensional, k-ary data cube instance D_{in}, with measures $\mu_1,\mu_2,\ldots,\mu_k$ and flag ϕ. These measures are *protected* in the sense that they remain the first k measures throughout the entire OLAP process and are never altered or removed unless they are destroyed in some cells. The cube D_{in} has also a Boolean flag ϕ, which typically has value 1 in all cells of $M(\mathrm{D}_{in})$, indicating that all the matrix cells are relevant for the input. Thus, the measures of the input cube D_{in} are denoted as follows:

$$\underline{\mu_1},\underline{\mu_2},\ldots,\underline{\mu_k};\phi.$$

After applying a sequence of OLAP operations to D_{in}, we obtain a data cube D, defined next.

Definition 10 (OLAP operation). Let D be a d-dimensional, $(k + l)$-ary *input* data cube instance with given measures $\mu_1,\mu_2,\ldots,\mu_k$, computed measures $\tau_1,\ldots,\tau_l$ and flag ϕ. The data cube D acts as the input of an *OLAP operation O* (of arity m), which consists of a sequence of n consecutive OLAP transformations that create the additional measures $\tau_{l+1},\ldots,\tau_{l+n}$, followed by the creation of an m-ary flag $\varphi_O^{(m)}$ (possibly preceded by a destructor δ). As the result of the creation of $\varphi_O^{(m)}$, the measures in the cells of the data cube are changed from $\underline{\mu_1},\underline{\mu_2},\ldots,\underline{\mu_k};\tau_1,\ldots,\tau_l;\phi;\tau_{l+1},\ldots,\tau_{l+n}$ to

$$\underline{\mu_1,\mu_2,\ldots,\mu_k};\tau_{l+n-m+1},\ldots,\tau_{l+n};\varphi_O^{(m)},$$

which become $\underline{\mu_1},\underline{\mu_2},\ldots,\underline{\mu_k};\tau_1,\ldots,\tau_m;\phi$,

after renaming. The output cube $\mathrm{D}^0 = O(\mathrm{D})$ has the same dimensions, hierarchy schemas and instances as D, and measures $\underline{\mu_1},\underline{\mu_2},\ldots,\underline{\mu_k};\tau_1,\ldots,\tau_m;\phi$.

In the case where $\varphi_O^{(m)}$ is preceded by a destructor δ, the same procedure is followed, except for the cells of $M(\mathrm{D})$ for which δ takes the value 0. These cells of $M(\mathrm{D})$ are emptied, contain no measures, and become inaccessible for future transformations or operations. □

We remark that the output $D^0 = O(D)$ of the OLAP operation O on input D can serve as input to a next OLAP operation. We later illustrate the composition of two operations in Example 11 and other examples.

4.4. Atomic OLAP transformations

In this section, we address the five classes of atomic OLAP transformations described in Definition 9. In the remainder of this section, we use the following notational convention. For a measure α, we write $\alpha(x_1,x_2,\dots,x_d)$ to indicate the value of α in the cell $(x_1,x_2,\dots,x_d) \in dom(D_1) \times dom(D_2) \times \cdots \times dom(D_d)$. We remark that $\alpha(x_1,x_2,\dots,x_d)$ does not exist for empty cells and it is therefore not considered in computations (such as sums). Also, we assume that there are *protected* measures $\mu_1,\mu_2,\dots,\mu_k$, and *computed* measures $\tau_1,\dots,\tau_l$ in the non-empty cells and that the next measure we compute is called τ_{l+1}.

Throughout this section, we continue with the running example given in Section 3, where we have a data cube D over the 3-dimensional matrix schema (D_1,D_2,D_3) = (*Product, Location, Time*), with measure μ_1 = *sales*, indicating the sales of certain products, at certain locations, at certain moments in time.

4.4.1. Arithmetic transformations

We start with the first class of transformations in Definition 9, namely, the arithmetic ones.

Definition 11 (Arithmetic transformations). The following creations of a new measure τ_{l+1} are *arithmetic transformations*:

1. (Rational constant) $\tau_{l+1} = \alpha$, with $\alpha \in \mathbf{Q}$, a rational number.
2. (Sum) $\tau_{l+1} = \alpha + \beta$, with $\alpha,\beta \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$.
3. (Product) $\tau_{l+1} = \alpha \cdot \beta$, with $\alpha,\beta \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$.
4. (Quotient) $\tau_{l+1} = \alpha/\beta$, with $\alpha,\beta \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$. Here, by convention, $a/0 := a$ for all $a \in \mathbf{Q}$.□

Example 7. If μ_1 = *sales* is the only measure, the following sequence of transformations computes the 10% of the sales:

– $\tau_1 = 0.1$ (rational constant); – $\tau_2 = \tau_1 \cdot \mu_1$ (product).

As another example, to create a Boolean measure that indicates whether a cell contains non-zero sales, we can write

– $\tau_3 = \mu_1/\mu_1$ (quotient).

The value of τ_3 is 1 if *sales* > 0 and 0 if *sales* = 0 (our definition of quotient says that 0/0 = 0).

□

4.4.2. Boolean transformations

We now address the second class of transformations in Definition 9, the boolean ones.

Definition 12 (Boolean transformations). The following creations of a new measure τ_{l+1} are *Boolean transformations*:

1. (Equality test on measures) $\tau_{l+1} = (\alpha = \beta)$, with $\alpha,\beta \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$. Here, the result of the comparison $(\alpha = \beta)$ is a Boolean 1 or 0 (cell per cell in the non-empty cells of the matrix).

2. (Comparison test on measures) $\tau_{l+1} = (\alpha < \beta)$, with $\alpha,\beta \in \{\mu_1,\mu_2,\ldots,\mu_k,\tau_1,\tau_2,\ldots,\tau_l\}$. Here, the result of the comparison $(\alpha < \beta)$ is a Boolean 1 or 0 (cell per cell in the non-empty cells of the matrix).
3. (Equality test on levels) For a level ` in the dimension schema $\sigma(D_i)$ of dimension $D_i$, and a constant object $c \in dom(D_i.$`$)$, $\tau_{l+1}(x_1,x_2,\ldots,x_d) =$ (` $= c)$ is an "equality" test. Here, the result of the comparison (` $= c)$ is a Boolean 1 or 0 (cell per cell in the non-empty cells of the matrix) such that $\tau_{l+1}(x_1,x_2,\ldots,x_d)$ is 1 if and only if x_i rolls-up to c at level `, that is $\rho(x_i,c)$.
4. (Comparison test on levels) For a level ` in the dimension schema $\sigma(D_i)$ of dimension $D_i$, and a constant object $c \in dom(D_i.$`$)$, $\tau_{l+1}(x_1,x_2,\ldots,x_d) =$ (` <` $c)$ is a "comparison" test. The result of the comparison (` <` $c)$ is a Boolean 1 or 0 (cell per cell in the non-empty cells of the matrix), such that $\tau_{l+1}(x_1,x_2,\ldots,x_d)$ is 1 if and only if x_i rolls-up to an object b at level ` for which $b$ <` c. The order <` can be any order that is defined on level `. The transformation $\tau_{l+1}(x_1,x_2,\ldots,x_d) = (c$ <` `$)$ is defined similarly. □

We remark that the above equality tests are superfluous since they can be expressed as a Boolean combination of comparison tests, but we include them for obvious practical reasons. Indeed, $a = b$ is equivalent to $\neg(a < b \vee b < a)$. Finally, note that the comparison test on levels uses the order <`, which may be the order (derived from) <, but, in practice, it will often be a lexicographical or alphabetical order, particular to the level domain.

Example 8. We illustrate the use of Boolean transformations by means of a sequence of transformations that implement a "dice" (see Section 5.2 for more details), that is, an OLAP operation that returns a portion of the cells in a data cube. The query

DICE(D,sales > 50)

asks for the cells in the matrix of D which contain sales that are higher than 50. Again, we assume that $\mu_1 =$ *sales* is the only available measure in the input cube. So, the measures in the cells are <u>sales</u>;ϕ. This query can be implemented by the following sequence of transformations:

- $\tau_1 = 49.99$ (rational constant);
- $\tau_2 = (\tau_1 <$ *sales*$)$ (comparison test on measures);
- $\tau_3 = \mu_1 \cdot \tau_2$ (product);
- $\delta = \tau_2$ (destructor); and – $\phi^{(1)} = \tau_2$ (unary flag)

The measure τ_3 contains the *sales* values larger than or equal to 50 (and a 0 if the *sales* are lower). The destructor δ destroys the cells that contain a O. Finally, the flag $\phi^{(1)}$ selects all cells from the input as output cells (it will contain a 1 for all such cells that satisfy the condition), and concludes the DICE(D,*sales* > 50) operation. The output of this operation is <u>sales</u>;τ_3;$\phi^{(1)}$, which is then renamed to <u>sales</u>;τ_1;ϕ. □

4.4.3. Selectors

We now address selector transformations.

Definition 13 (Selector transformations). The following creations of a new measure τ_{l+1} are *selector transformations* (or *selectors*), and their definition is (as always) cell per cell of $M(\mathrm{D})$:

1. (Constant selector) For a level ` in the dimension schema $\sigma(D_i)$ of a dimension $D_i$, and $c \in dom(D_i.$`$)$, τ_{l+1} can be a *constant-selector for c*, denoted $\sigma_{D_i.\text{`}=c}$, and it corresponds to the equality test on levels $\tau_{l+1}(x_1,x_2,\ldots,x_d) =$ (` $= c)$.

2. (Level selector) For a level ` in the dimension schema $\sigma(D_i)$ of a dimension $D_i$, $\tau_{l+1}$ can be a *level-selector for* `, denoted by $\sigma_{D_i.`}$, which means that we have, for all $x_j \in dom(D_j)$ with j 6= i,

□

1 if $a = rep(b)$

□

$\tau_{l+1}(x_1,...,x_{i1},a,x_{i+1},...,x_d)$ = for some $b \in dom(D_i.`)$, □0 otherwise.

□

The *constant* selector in Definition 13, corresponds to the equality test on levels (see 3. in Definition 12). Here, this transformation appears with a different functionality and we reserve a special notation for it, and we repeated it. Also, note that the *level* selector selects all representatives (at the *Bottom* level) of objects at level ` of dimension D_i.

Example 9. As a second example of a dice operation, we look at the query

DICE(D,*Location.City = antwerp*), which asks for the sales in the city of *antwerp*. This operation is destructive, since it destroys all the information in cells that do not belong to *antwerp*. This query can be implemented by the following sequence of transformations:

– $\tau 1 = \sigma_{Location.City=antwerp}$ (constant selector);
– $\tau_2 = \tau_1 \cdot \mu_1$ (product);
– $\delta = \tau_1$ (destroys the cells outside *antwerp*); – $\phi^{(1)} = \tau_1$ (unary flag creation).

The output arity of the query DICE(D,*Location.City = antwerp*) is 1. The measure τ_2 selects the sales in *antwerp* only, and the flag $\phi^{(1)}$ is a selector on the constant *antwerp*. The destructor δ, that precedes the flag, empties the cells outside *antwerp*. □

Example 10. As a next example, we look at the query

DICE(D,*Location.City = antwerp OR Location.City = brussels),*

which asks for the sales in the cities of *antwerp* and *brussels*. This query can be implemented by the following sequence of transformations:

– $\tau 1 = \sigma_{Location.City=antwerp}$ (constant selector);
– $\tau 2 = \sigma_{Location.City=brussels}$ (constant selector);
– $\tau_3 = \tau_1 + \tau_2$ (sum);
– $\tau_4 = \tau_3 \cdot \mu_1$ (product);
– $\delta = \tau_3$ (destroys the cells outside *antwerp* and *brussels*); – $\phi^{(1)} = \tau_3$ (unary flag creation).

The logical connective *OR* is implemented by the sum in τ_3, which can take values 0 or 1, since the cities *antwerp* and *brussels* do not overlap. Thus, this sum implements their union. Then, measure, ϕ_4 selects the sales in *antwerp* and *brussels* only. The flag $\phi^{(1)}$ is a selector on the constants *antwerp* and *brussels* and indicates that the cells of both these cities belong to the output. The destructor δ, that precedes the flag, empties the cells outside *antwerp* and *brussels*. □

Note that in the two previous examples, the flag and the destructor do the same double work. However, this will not be the case in most situations, and, in practice, it would not have impact.

Our final example in this section, combines the two previous ones, illustrating a dice operation on a measure (*sales*) and a dimension member (*brussels*).

Example 11. We consider the query

DICE$(\mathrm{D}$,*sales* > 50 *AND Location.City* = *brussels*$)$.

We can implement this by the operation DICE$(\mathrm{D}$,*sales* $> 50)$ followed by the operation DICE$(\mathrm{D}$,*Location.City* = *brussels*$)$.

Let $\underline{sales};\phi$ be the input measures. The query DICE$(\mathrm{D}$,*sales* $> 50)$ is the same as in Example 8. The output of this operation is $\underline{sales};\tau_3;\phi^{(1)}$, which is then renamed to $\underline{sales};\tau_1;\phi$. Next, the query DICE$(\mathrm{D}$,*Location, City* = *brussels*$)$ is implemented as

– $\tau 2 = \sigma_{Location.City=brussels}$ (constant selector);

– $\tau_3 = \tau_2 \cdot \mu_1$ (*product*);

– $\delta = \tau_2$ (destroys the cells outside *brussels*);

– $\phi^{(1)} = \tau_2 \cdot \phi$ (product and unary flag creation).

The output of this operation is $\underline{sales};\tau_3;\phi^{(1)}$, which we rename to $\underline{sales};\tau_1;\phi$.

Note that, of course, the query can also be implemented as DICE$(\mathrm{D}$,*Location.City* = *brussels*$)$ followed by DICE$(\mathrm{D}$,*sales* $> 50)$. □

4.4.4. Counting, sum and min-max

Now, we give transformations for counting different measure values, for summing all values of a measure in a matrix, and for determining the minimum and maximum value of a measure in a matrix. Later on, in Definition 19, we extend the counting and min-max transformations to be used together with grouping ones.

Definition 14 (Counting, sum, and min-max transformations). The creations of a new measure τ_{l+1} defined next, are denoted *counting, sum and min-max transformations*:

1. (Count-Distinct) $\tau_{l+1} = \#_{6=}(\alpha)$, with $\alpha \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$ counts the number of distinct values of the measure α in the complete matrix $M(\mathrm{D})$ of the data cube.
2. (d-dimensional sum)

$$\tau_{l+1} = \underset{(x_1,x_2,\dots,x_d)\in M(\mathrm{D})}{\mathrm{X}} \alpha(x_1,x_2\dots,x_d),$$

with $\alpha \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$, gives the sum of the measure α over all non-empty matrix cells. We abbreviate this operation by writing

$$\tau_{l+1} = \mathrm{SUM}_d(\alpha)$$

and call this transformation the *d-dimensional sum*.

3. (Min-Max) $\tau_{l+1} = \min(\alpha)$, with $\alpha \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$, gives the smallest value of the measure α in non-empty cells of the matrix $M(\mathrm{D})$. Similarly, $\tau_{l+1} = \max(\alpha)$, gives the largest value of the measure α in the matrix $M(\mathrm{D})$. □

It is important to remark that the above transformations create the *same new measure value* for all cells of the matrix $M(\mathrm{D})$.

We now give two examples of the use of d-dimensional sum. Examples of the use of $\#_{6=}(\alpha)$ are given in the following sections.

Example 12. Let us consider the query "*(grand total) average sales*". This is calculated as the total sales (over all cities, products and dates), divided by the total number of cells in the matrix of the data cube. This query can be computed as follows, given μ_1 = *sales*:

- $\tau_1 = \mathrm{SUM}_3(\mu_1)$ (this is the grand total of sales, stored in all cells in the matrix);
- $\tau_2 = \sigma_{Location.Bottom}$ (this puts a 1 in every cell of the matrix)
- $\tau_3 = \mathrm{SUM}_3(\tau_2)$ (this is the grand total of cells in the matrix);
- $\tau_4 = \tau_1/\tau_3$ (this is the average);
- $\phi^{(1)} = \tau_2$ (this flag creation selects all cells of the matrix).

The output measures are $\underline{sales};\tau_4;\phi^{(1)}$, which are renamed $\underline{sales};\tau_1;\phi$. This means that the grand total of average sales is now available in every cell of the matrix of the cube. □

Example 13. Now, we look at the query "total sales in *antwerp*". This query is asking for the total sales (over all products and dates) in the city of *antwerp*. The query can be computed as follows, given μ_1 = *sales*:

- $\tau 1 = \sigma_{Location.City=antwerp}$ (constant selector on *antwerp*);
- $\tau_2 = \tau_1 \cdot \mu_1$ (product that selects the sales in *antwerp*, puts a 0 in all the other ones);
- $\tau_3 = \mathrm{SUM}_3(\tau_2)$ (this is the total sales in *antwerp* in every cell); – $\tau_4 = \tau_3 \cdot \tau_1$ (this is the total sales in *antwerp* in the cells of *antwerp*); – $\phi^{(1)} = \tau_1$ (this flag creation selects the cells of *antwerp*).

The output measures are $\underline{sales};\tau_4;\phi^{(1)}$, which are renamed $\underline{sales};\tau_1;\phi$. Thus, the value of the total of sales in *antwerp* is now available in every cell corresponding to *antwerp*. For the cells outside *antwerp* there is a 0. We remark that this example can be modified with a destructor that effectively empties cells outside *antwerp*. □

Example 14. We look at the query "maximum sales", which should return all cells containing the maximum value in the cube. This query can be computed as follows:

- $\tau_1 = \max(\mu_1)$ (the maximum sales amount);
- $\tau^1 = (\tau_1 = \mu_1)$ (equality test to determine if a cell reaches the maximum);
- $\tau_3 = \tau_2 \cdot \mu_1$ (only the maximum sales remain; the others turn 0); – $\phi^{(1)} = \sigma_{Location.Bottom}$ (this flag creation selects all cells).

The output measures are $\underline{sales};\tau_3;\phi^{(1)}$, which are renamed $\underline{sales};\tau_1;\phi$. We remark that this example, like the previous one, can be modified with a destructor that effectively empties cells with strictly less than maximum sales. □

4.4.5. Grouping

The most common OLAP operations (e.g., roll-up, slice), require grouping data before aggregating them. For example, typically we will ask queries like "total sales by city", which requires grouping facts

---

$^1 \cdot \quad 7 \cdot \quad 11 = \quad 154$. Each cell in $A_1 \times A_2 \times A_3$ gets a unique prime product label.

by city, and, for each group, sum all of its sales; or, we can ask for the "total sales by city and day", meaning that, for each city-day combination, we sum all the sales. Therefore, we need a transformation to express "grouping". We address this issue next.

To deal with grouping, we use the concept of "prime labels" for sets and products of sets. We will use these labels to identify elements in dimensions and in dimension levels. Before giving the definition of the grouping transformations, we elaborate on prime labels and product of prime labels. As we show, these prime labels work in the context of measures that take rational values (as it is often the case, in practice).

The following definition specifies our infinite supply of prime labels.

Definition 15 (Prime labels).√Let√p_n denote the√ √ √n-th prime number, for√ $n > 1$. We define the sequence of *prime labels*√as follows: $1, \overline{2}, 3, 5, 7, 11,\ldots, p_n,\ldots$ We denote the set of all prime labels by $\overline{\mathrm{P}}$. □

Now, we define a prime labeling of a finite set and of a Cartesian product of finite sets.

Definition 16 (Prime labeling of sets).√Let $A, A_1,A_2,\ldots,A_n$ be (finite) sets. A *prime labeling* of the set A is an injective function $w : A \to \overline{\mathrm{P}}$. For $a \in A$, we call $w(a)$ the *prime label* of a (for the prime labeling w).

Let I be a subset of $\{1,2,\ldots,n\}$, which serves as an index set. A *prime product I-labeling* of the Cartesian product $A_1 \times A_2 \times \cdots \times A_n$ consists of prime labelings w_i of the sets A_i, for $i \in I$, that satisfy the condition that $w_i(A_i) \cap w_j(A_j)$ is empty for $i,j \in I$ and $i =6 j$. For $(a_1,a_2,\ldots,a_n) \in A_1 \times A_2 \times\cdots\times A_n$, we call Q$_{i\in I}\, w_i(a_i)$ the *prime product I-label* of $(a_1,a_2,\ldots,a_n)$ (given the prime labelings w_i, for $i \in I$). When I is a strict subset of $\{1,2,\ldots,n\}$, we speak about a *partial prime product labeling* and when $I = \{1,2,\ldots,n\}$, we speak about a *full prime product labeling*. □

In the sequel, whenever I is clear from the context, we can omit referencing√ I. In practice, to label a set $A_1\times A_2\times\cdots\times A_n$, we use consecutive, available labels from $\overline{\mathrm{P}}$ to label the sets $A_1,A_2,\ldots,A_n$, as the following example illustrates. Further on, we apply this labeling to domains of dimensions (possibly at different levels).

Example 15. Let $A_1 = \{a_1,a_2,a_3\}$,$A_2 = \{b_1,b_2\}$ and $A_3 = \{c_1,c_2\}$. To create a full prime product label for the elements of $A_1 \times A\sqrt{2} \times A_3$, we can use the prime labelings√ √ w_1√,w_2 and w_3, defined as√ follows: $w\sqrt{1}(a_1) = 1$,$w_1(a_2) = 2$,$w_1(a_3) = 3$, $w_2(b_1) = 5$,$w_2(\sqrt{b_2}) = 7$, $w_3(c_1) = 11$ and $w_3(c_3) = 13$. We remark that we have used consecutive elements of P (with respect to the natural

√order of natural numbers). These labelings give the tuple√ √ √ (a_2,b_2,c_1) the label $w_1(a_2)\cdot w_2(b_2)\cdot w_3(c_1) =$

To create a partial prime product label for $I = \{1,2\}$, we can use w_1 and w_2, as given above. In this case, for any $a \in A_1$ and $b \in A_2$, the cells (a,b,c_1) and (a,b,c_2) get the same (partial) prime product label. □

If we view a Cartesian product $A_1 \times A_2 \times \cdots \times A_n$ as a finite matrix, whose cells contain rationalvalued measures, we can use prime (product) labelings as follows in the aggregation process. Let us assume that the cells of $A_1 \times A_2 \times\cdots\times A_n$ contain rational values of a measure μ and let us denote the value of this measure in the cell $(a_1,a_2,\ldots,a_n)$ by $\mu(a_1,a_2,\ldots,a_n)$. If we have a full prime product labeling on $A_1\times A_2\times\cdots\times A_n$, then we can consider the sum over this Cartesian product of the product of the prime product labels with the value of μ:

$$\sum_{(a_1,a_2,\ldots,a_n)\in A_1\times A_2\times\cdots\times A_n} \mu(a_1,a_2,\ldots,a_n)\cdot w_1(a_1)\cdot w_2(a_2)\cdots w_n(a_n). \qquad (1)$$

Since each cell of $A_1 \times A_2 \times \cdots \times A_n$ has a unique prime product label, and since these labels are rationally independent (as we show in Property 2 below), this sum enables us to retrieve the values $\mu(a_1,a_2,\ldots,a_n)$.

If we have a partial prime product labeling on $A_1\times A_2\times\cdots\times A_n$, determined by an index set I, then, again, we can consider the sum over this Cartesian product of the product of the partial prime product labels with the value of μ:

$$\sum_{(a_1,a_2,\ldots,a_n)\in A_1\times A_2\times\cdots\times A_n} \mu(a_1,a_2,\ldots,a_n)\cdot \prod_{i\in I} w_i(a_i). \qquad (2)$$

Now, all cells in $A_1 \times A_2 \times \cdots \times A_n$ above a cell in the projection of $A_1 \times A_2 \times \cdots \times A_n$ on its components with indices in I, receive the same prime label. This means that these cells are "grouped" together and the above sum allows us to retrieve the part of the sum that belongs to each group. To make this clearer, we can write (†2) as

$$\sum_{\times_{i\in I} A_i} \left(\sum_{\times_{i\in I^c} A_i} \mu(a_1,a_2,\ldots,a_n) \right) \cdot \prod_{i\in I} w_i(a_i) \quad , \qquad (2a)$$

where the outer sum ranges over the components of $A_1 \times A_2 \times \cdots \times A_n$ whose index belongs to I and where the inner sum ranges over the components of $A_1 \times A_2 \times \cdots \times A_n$ whose index belongs to $I^c := \{1,2,\ldots,n\} \setminus I$. The above statement says that (†2) allows us to uniquely determine the sums

$$\sum_{\times_{i\in I^c} A_i} \mu(a_1,a_2,\ldots,a_n).$$

We remark that this last sum is the same for all cells above a cell in the projection of $A_1\times A_2\times\cdots\times A_n$ on the components whose index is in I.

The following definition gives a name to the above sums.

Definition 17 (Prime sums). We call sums of type ($\dagger_1$) *full prime sums* and sums of type ($\dagger_2$) *partial prime sums (over I)*. □

√ − √−

...,The ~~fol~~lowing property can be derived from the well-known fact that the field extension$\sqrt{p_n}$) = {a_0 + $a_1\sqrt{2}$ +− $a_2\sqrt{3}$+− ... + $a_n\sqrt{p_n}$ | **Q**n(over2, **Q**3, $a_0,a_1,a_2,\ldots,a_n \in$ **Q**} has degree 2

and corollaries of this property (see Chapter 8 in [4]). No square root of a prime number is a rational combination of square roots of other primes.

Property 2. Let $n > 1$ and let $A_1 \times A_2 \times \cdots \times A_n$ be a Cartesian product of finite sets. We assume that the cells $(a_1,a_2,\ldots,a_n)$ of this set contain rational values $\mu(a_1,a_2,\ldots,a_n)$ of a measure μ. Let I be a subset of $\{1,2,\ldots,n\}$ and let w_i be prime labelings of the sets A_i, for $i \in I$, that form a prime product I-labeling (see Definition 16). Then we have that the prime sum ($\dagger_2$) uniquely determines the values $\text{P}_{\times i \in Ic A_i}$ □ $\mu(a_1,a_2,\ldots,a_n)$ for all cells of $A_1 \times A_2 \times \cdots \times A_n$. □

Proof. We give the proof of this property in Appendix ??.

Remark 2. We remark that we use these prime (product) labels in a purely *symbolic* way without actually calculating the square root values in them. The square roots are treated as symbolic entities in the computations. □

Given these facts about prime (product) labels, we are ready to define atomic OLAP operations that allow us to implement grouping. In what follows, we apply these prime labels to the case where the sets A_i in $A_1 \times A_2 \times \cdots \times A_n$ are domains of dimensions (e.g., at the bottom level), or domains of dimensions at some level.

Definition 18 (Grouping transformations). The following creations of a new measure τ_{l+1} are *grouping transformations*:

1. (Prime labels for groups in one dimension) Let D_i be a dimension and ` a level in the dimension schema $\sigma(D_i)$ of a dimension $D_i$. Let $dom(D_{i.}`) = \{b_1,b_2,\ldots,b_m\}$ with induced order $b_1 < b_2 < \cdots < b_m$ (see Property 1). If the prime labels $w_1,w_2,\ldots,w_k$ have been used by previous transformations, then for all j, with j 6= i, and all $x_j \in dom(D_j)$, we have $\tau_{l+1}(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d) = w_{k+l}$ if $\rho(x_i,b_l)$. We denote this transformation by $\gamma_{D_i.`}(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d)$ or $\gamma_{D_i.`}$, for short, and call the result of such a transformation a *prime labeling*.
2. (Projection of a prime sum) If the result of some previous transformation τ_m is a (full or partial) prime sum $\sum_{i=k}^{k+l} a_i \cdot w_i$ (over the complete matrix M(D)) in which prime (product) labels $w_k,w_{k+1},\ldots,w_{k+l}$ (computed in a previous transformation τ_n) are used, then τ_{l+1} is a new measure that "projects" on the appropriate component from the prime sum, that is, $\tau_{l+1}(x_1,x_2\ldots,x_d) = a_{k+l}$ if the prime (product) label $\tau_n(x_1,x_2\ldots,x_d) = w_{k+l}$. We denote this projection transformation by $\tau_m|_{\tau_n}$.

   □

In OLAP practice we often need to compute the number of elements in a dimension level. That means, we perform an aggregation without operating on measures, but on dimension members. We use the concepts explained above for doing this. As always, we use *sales* information for certain products, at certain locations, at certain time moments.

Example 16. Consider the query "total number of cities", which asks to count the number of cities appearing at the *Bottom* level of the dimension *Location*. We can implement this query using CountDistinct and prime labels, given $\mu_1 = sales$, as follows:

- $\tau_1 = \gamma_{Location.City}$ (gives each city a different prime label);
- $\tau_2 = \#_{6=}(\tau_1)$ (counts the number of different prime labels, e.g., the number of cities, and stores this number in every cell);
- $\phi^{(1)} = \sigma_{Location.Bottom}$ (this flag creation selects all cells of the matrix).

The output measures are $\underline{sales};\tau_2;\phi^{(1)}$, which are renamed $\underline{sales};\tau_1;\phi$. This means that the total number of cities is now available in every cell of the matrix $M(\mathrm{D})$. □

Example 17. We look at the query "total number of countries", which asks to count the number of countries appearing at the *Country* level of the dimension $Location$. We can implement this query using Count-Distinct and prime labels, given $\mu_1 = sales$, as follows:

- $\tau_1 = \gamma_{Location.Country}$ (gives each country a different prime label);
- $\tau_2 = \#_{6=}(\tau_1)$ (counts the number of different prime labels and thus the number of countries); – $\phi^{(1)} = \sigma_{Location.Bottom}$ (this flag creation selects all cells of the matrix).

The output measures are $\underline{sales};\tau_2;\phi^{(1)}$, which are renamed $\underline{sales};\tau_1;\phi$. This means that the total number of countries is now available in every cell of the matrix. □

Note that, like in the previous example, we are operating on the dimensions, and we are not aggregating measures, which shows the generality of our approach. The next example goes further into this issue, since it uses grouping within the same dimension, in order to compute the number of elements in a child dimension level rolling-up to their corresponding parent element. Thus, we can express queries like "number of cities per country", "number of products per category", and so on.

Example 18 below, uses two prime labels on the dimension *Location*. One prime labeling is at the *Country* level; the second prime labeling is at the *City* (*Bottom*) level. This construction fits in the given prime product labeling concept if we consider A_1 = *dom(Location.Country)* and A_2 = *dom(Location.City)*.

Example 18. Consider the query "for each country, give the total number of cities". This query can be implemented as follows (we explain the details below):

- $\tau_1 = \gamma_{Location.Country}$ (this gives each country a prime label);
- $\tau_2 = \gamma_{Location.City}$ (this gives each city a (fresh) prime label);
- $\tau_3 = \tau_1 \cdot \tau_2$ (this gives each city a product of prime labels);
- $\tau_4 = \mathrm{SUM}_3(\tau_3)$;
- $\tau_5 = \gamma_{Product.Bottom}$ (gives each product a different prime label);
- $\tau_6 = \#_{6=}(\tau_5)$ (counts the number of products – see Example 16); – $\tau_7 = \gamma_{Time.Bottom}$ (gives each time moment a different prime label);
- $\tau_8 = \#_{6=}(\tau_7)$ (counts the number of moments in time – see Example 16);
- $\tau_9 = \tau_6 \cdot \tau_8$ (is the number of products times the number of time moments);
- $\tau_{10} = \tau_4/\tau_9$ (normalization of the sum);
- $\tau_{11} = \tau_{10}\,|_{\tau_2}$; (projection over the prime labels of city);
- $\tau_{12} = \mathrm{SUM}_3(\tau_{11})$ (3-dimensional sum);
- $\tau_{13} = \tau_{12}/\tau_9$ (normalization of the sum);

– $\tau_{14} = \tau_{13}\,|_{\tau_1}$ (projection over the prime labels of country);

– $\phi^{(1)} = \sigma_{Location.Bottom}$ (this flag creation selects all cells of the matrix).

We now discuss this example, using the data given in Example 4. Transformation τ_1 gives each country a next available prime label. Since no labels have been used yet,√ *belgium* gets label√1 and *france* gets

label 2. Transformation√τ_2 gives each city a next available prime label. Since√√ 1 and 2 have been used,√

antwerp gets label 3, *brussels* gets label 5, √*paris* gets label√7, and *marseille*√gets label√ 11.

Transformation√√ √τ_3 gives *antwerp* the value √3 (i.e.,√1√. 3, *brussels* the value 5(1. 5), *paris* the value 14 (2. 7), and *marseille* the value√ 22√ (2√. 11). If there are 10 products and 100 time√ moments, then τ_4 puts the value $10 \cdot 100 \cdot (3 + 5 + 14 + 22)$ in each cell of the matrix $M(\mathrm{D})$.

Transformations τ_6 and τ_8 count the number of products and the number of time moments (using fresh prime labels), and the product of these quantities is computed in√ √ √ √ τ_9. In τ_{10}, τ_3 is divided by this product, putting 3 + 5 + 14 + 22 in every cell of the matrix.

√_ √_ √_ √__

Transformation τ_{11} is a projection on the prime labels of√ √ √ √ *City*. Since√ 3√, 5, √7, and√ √11 are the√ prime labels for the cities, and since 3+ 5+ 14+√ 22 = 1· 3+1· 5+ 2· 7+ 2· 11, this will put 1 in the cells of *antwerp* and *brussels* and 2 in the cells of *paris* and *marseille*. These are the coefficients of the sum, that are associated with each of the prime labels for the cities (e.g., 1 is the√ √

coefficient for 3 and 5, respectively√*antwerp* and *brussels*). √

Next, τ_{12} puts $10 \cdot 100 \cdot (2 \cdot 1 + 2 \cdot 2)$ in every cell of the cube and τ_{13} puts $2 \cdot {}^{1\,+\,2}\!\sqrt{\cdot}\,2$ in every cell of the cube. Finally, τ_{14} projects on the prime labels of countries, which are 1 and 2. This puts a 2 in every cell of a Belgian city and a 2 in every cell in a French city. This is the result of the query, as the flag indicates, that is returned in every cell. Now every cell of a city in *belgium* has the count of 2 cities, as has every city in *france*. □

4.4.6. Counting and min-max revisited

Now that we know prime (product) labelings, we can give extensions of the counting and min-max transformations of Definition 14. Here, the counting, the minimum, and the maximum are taken over cells which share a common prime (product) label.

Definition 19. The following creations of a new measure τ_{l+1} are generalizations of the *counting and min-max* transformations:

1. (Count-Distinct) If the result of some previous transformation τ_m is a prime (product) labeling of the cells of $M(\mathrm{D})$, then $\tau_{l+1}(x_1, x_2 \ldots, x_d) = \#_{6=}\,|_{\tau_m}(\alpha)$, with $\alpha \in \{\mu_1, \mu_2, \ldots, \mu_k, \tau_1, \tau_2, \ldots, \tau_l\}$ counts the

number of different values of the measure α in cells of the matrix $M(\mathrm{D})$ that have the same prime product label as $\tau_m(x_1,x_2 \dots,x_d)$.

2. (Min-Max) If the result of some previous transformation τ_m is a prime (product) labeling of the cells of $M(\mathrm{D})$, then $\tau_{l+1}(x_1,x_2 \dots,x_d) = \min |_{\tau_m} (\alpha)$, with $\alpha \in \{\mu_1,\mu_2,\dots,\mu_k,\tau_1,\tau_2,\dots,\tau_l\}$, gives the the smallest value of the measure α in cells of the matrix $M(\mathrm{D})$ that have the same prime product label as $\tau_m(x_1,x_2 \dots,x_d)$. And $\tau_{l+1}(x_1,x_2 \dots,x_d) = \max |_{\tau_m} (\alpha)$ is defined similarly.□

We remark that when there is only one prime label (for instance, 1) throughout the matrix $M(\mathrm{D})$, then the above generalization of the counting and min-max transformations correspond to the version of Definition 14.

5. The classical OLAP operations

In this section, we prove that the classical OLAP operations can be expressed using the OLAP transformations from Section 4. These classic operations can be combined to express complex analytical queries. The classical OLAP operations are

- Dice (see Section 5.2);
- Slice (see Section 5.3);
- Slice-and-Dice (see Section 5.4); – Roll-Up (see Section 5.5); and – Drill-Down (see Section 5.5).

Throughout this section, we assume that the input data cube D_{in} has k given measures $\mu_1,\mu_2,\dots,\mu_k$ (as in Definition 7) and that at some point in the OLAP process this cube is transformed to a cube D, having measures $\underline{\mu_1},\underline{\mu_2},\dots,\underline{\mu_k};\tau_1,\tau_2,\dots,\tau_l;\phi$,

where $\tau_1,\tau_2,\dots,\tau_l$, with $l > 0$, are created measures and ϕ is an input/output flag.

5.1. Boolean cell-selection condition

Before we present our study on the OLAP operations, we need to define the notion of a Boolean cellselection condition. We also give a lemma about its expressiveness that is used throughout Section 5.

Definition 20 (Boolean condition on cells). Let $M(\mathrm{D}) = dom(D_1) \times dom(D_2) \times \cdots \times dom(D_d)$ be the matrix of D. A *Boolean condition on the cells of* $M(\mathrm{D})$ is a function φ from $M(\mathrm{D})$ to $\{0,1\}$. We say that the cells of $M(\mathrm{D})$ in the set $\varphi^{-1}(\{1\})$ are *selected* by φ.

We say that a Boolean condition φ is *transformation-expressible* if there is a sequence of OLAP transformations $\tau_1,\tau_2,\dots,\tau_k$ such that $\varphi(x_1,x_2,\dots,x_d) = \tau_k(x_1,x_2,\dots,x_d)$ for all $(x_1,x_2,\dots,x_d) \in M(\mathrm{D})$. □

Lemma 1. If $\varphi,\varphi_1,\varphi_2$ are transformation-expressible Boolean conditions on cells, then NOT φ, φ_1 AND φ_2 and φ_1 OR φ_2 are transformation-expressible Boolean conditions on cells. □

Proof. We give the proof in Appendix ??. □

5.2. Dice

Intuitively, the *Dice* operation selects the cells in a cube D that satisfy a Boolean condition φ on the cells. The syntax for this operation is

DICE(D,φ), where φ is a Boolean condition over level values and measures. The resulting cube has the same dimensionality as the original cube. This operation is analogous to a selection in the relational algebra. In a data cube, it selects the cells that satisfy the condition φ by flagging them with a 1 in the output cube. The operation has been already illustrated in Examples 8, 9, 10 and 11, with queries such as DICE(D,$Location.City = antwerp\ OR\ Location.City = brussels$).

We also allow equality and order constraints on objects at certain levels and in different dimensions, as illustrated by the example

DICE(D,*Location.Country* = *belgium AND Time.Day* > 15/1/2014).

We also consider equality and order constraints over measures, like in the query

DICE(D,*sales* > 50) of Example 8. Therefore, our approach covers all typical cases in real-world OLAP [?]. We next formalize the operator's definition in terms of our transformation language. In the remainder, we use the term *OLAP operation* to express a sequence of OLAP transformations.

Definition 21 (Dice). Given a data cube D, the operation DICE(D,φ), selects all cells of the matrix $M(\mathrm{D})$ that satisfy the Boolean condition φ by giving them a 1 flag in the output. The Boolean condition φ on the cells of $M(\mathrm{D})$ is a Boolean combination of conditions of the form:

- A selector on a value b at a certain level ℓ of some dimension D_i;
- A comparison condition at some level ℓ from a dimension schema $\sigma(D_i)$ of a dimension D_i of the cube of the form $\ell < c$ or $c < \ell$, where c is a constant (at that level ℓ);
- An equality or comparison condition on some measure α of the form $\alpha = c$, $\alpha < c$ or $c < \alpha$, where c is a (rational) constant. □

Property 3. Let D be a data cube en let φ be a Boolean condition on the cells of $M(\mathrm{D})$ (as in Definition 21). The operation DICE(D,φ) is expressible as an OLAP operation. □

Proof. The proof is given in Appendix ??. □

5.3. Slice

Intuitively, the *Slice* operation takes as input a d-dimensional, k-ary data cube D and a dimension D_i and returns as output SLICE(D,D_i), which is a "$(d-1)$-dimensional" data cube in which the original measures $\mu_1,\ldots,\mu_k$ are replaced by their aggregation (sum) over different values of elements in $dom(D_i)$. In other words, dimension D_i is removed from the data cube, and, if this operation is part of a sequence of OLAP ones, D_i will not be visible in the next operations. That means, for instance, that we will not be able to dice on the levels of the removed dimension. As we will see, the "removal" of dimensions is, in our approach, implemented by means of the destroyer measure δ. We remark that the aggregation above is due to the fact that, in order to eliminate a dimension D_i, this dimension should have exactly one element [1], therefore a roll-up (which we explain later in Section 5.5) to the level *All* in $D-i$ is performed.

For example, if (D_1,D_2,D_3) = (*Product, Location, Time*), and we consider

SLICE(D,$Location$),

then we obtain a cube whose (*Product,Time*)-cells contain the sums of the given measures for certain products and times, but summed over all locations (for each time-product combination).

Obviously, in our philosophy, we keep the d-dimensional data cube and store identical aggregate values for all locations, in the cells above some product-time combination. Next, we destroy all locations, except the representative for All in the *Location* dimension. As explained in Example 6, *antwerp* represents All and we only keep the cells for *antwerp*, where we keep the aggregate values. We formalize this next.

Definition 22 (Slice). Given a data cube D, and one of its dimensions D_i, the operation SLICE(D,D_i) "replaces" the measures $\mu_1,\mu_2,\ldots,\mu_k$ by their aggregation (sum) $\mu_n^{\Sigma_i}$ (for 1 6 n 6 k) as follows:

$$\mu_n^{\Sigma_i}(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d) = \mathrm{X} \mu_n(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d),$$
$$x_i \in dom(D_i)$$

for all $(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d) \in M(\mathrm{D})$. Further, the operation SLICE(D,D_i) destroys all cells except those of the representative of All for dimension D_i. We abbreviate the above 1-dimensional sum as SUM$_{D_i}(\mu_n)$. □

Property 4. Let D be a data cube and let D_i be one of its dimensions. The operation SLICE(D,D_i) is expressible as an OLAP operation. □

Proof. The formal proof of of Property 4 can be found in Appendix ??. Example 19 gives the intuition of such proof. □

Example 19. Consider our running example with dimensions (D_1,D_2,D_3) = (*Product, Location, Time*) and measure μ_1 = *sales,* and consider the query

SLICE$(\mathrm{D},$*Location*$)$.

This query returns a cube with (*product,time*)-cells which contain the sums of μ_1 for each product-time combination, over all locations (for that product and that time). At the end, all cells not belonging to the representative of All in the dimension *Location*, that is, *antwerp*, are destroyed.

The query SLICE$(\mathrm{D},$*Location*$)$ is the result of the following transformations

- $\tau_{l+1} = \gamma_{Product.Bottom}$ (prime labels on products);
- $\tau_{l+2} = \gamma_{Time.Bottom}$ (fresh prime labels on time moments);
- $\tau_{l+3} = \tau_{l+1} \cdot \tau_{l+2}$ (product of the two previous prime labels);
- $\tau_{l+4} = \mu_1 \cdot \tau_{l+3}$ (product);
- τ_{l+5} = SUM$_3(\tau_{l+4})$ (3-dimensional sum);
- $\tau_{l+6} = \tau_{l+5} |_{\tau_{l+3}}$ (projection on prime product labels);
- $\tau_{l+7} = \sigma_{Location.All}$ (selects the representative of All in the dimension *Location*); – $\delta = \tau_{l+7}$ (destroys all cells except the representative of All in dimension *Location*); – $\phi^{(1)} = \sigma_{Location.All}$ (this flag creation selects the relevant cells of the matrix).

The transformation τ_{l+4} gives each (*product,time*)-combination a unique prime product label. This label is multiplied by the *sales* in each cell. We then make the global sum over $M(\mathrm{D})$ in τ_{l+5}. The transformation $\tau_{l+6} = \tau_{l+5} |_{\tau_{l+3}}$ is the projection over the prime product labels for (*product,time*)combinations. This gives each cell above some fixed (*product,time*)-combination, the sum of the *sales*, over all locations, for that (*product,time*)-combination. All cells of $M(\mathrm{D})$ that do not belong to *antwerp* (selected in τ_{l+7}), which represents all, are destroyed by δ.

□

5.4. Slice and dice

A particular case of the *Slice* operation occurs when the dimension to be removed already contains a unique value at the bottom level. Then, we can avoid the roll-up to *All*, and define a new operation, called *Slice-and-Dice*. Although this can be seen as a *Dice* operation followed by a *Slice* one, in practice, both operations are usually applied together.

Definition 23. Given a data cube D, one of its dimensions D_i and some value a in the domain $dom(D_i)$, the operation SLICE-DICE(D, D_i, a) contains all the cells in the matrix $M(\mathrm{D})$ such that the value of the dimension D_i equals a. All other cells are destroyed. □

Property 5. Let D be a data cube, D_i on of its dimensions en let $a \in dom(D_i)$. The operation SLICE-DICE(D, D_i, a) is expressible as an OLAP operation. □

Proof. The proof is given in Appendix ??. □

Example 20. In our running example, the operation SLICE-DICE$(\mathrm{D}, \mathit{Location}, \mathit{antwerp})$ is implemented by the output flag $\sigma_{Location.City=antwerp}$. □

5.5. Roll-Up and Drill-Down

We now address two key operations in typical OLAP practice, namely *Roll-Up* and *Drill-Down*. Intuitively, the former aggregates measure values along a dimension up to a certain level. The latter, disagregates measure values along a dimension, down to a certain level. However, as we already commented, although at first sight it may appear that *Drill-Down* is the inverse of *Roll-Up*, like stated in [1], this is not necessarily the case, particularly when we are composing several OLAP operations, and, for example, a *Roll-Up* is followed by a *SLICE* or a *DICE.* In these cases, we cannot just undo the *Roll-Up*, but we need to account for the cells that have been eliminated on the way.

More precisely, the *Roll-Up* operation takes as input a data cube D, a dimension D_i and a subpath h of a hierarchy H over D_i, starting in a node ℓ_0 and ending in a node ℓ, and returns the aggregation of the original cube along D_i up to level ℓ for some of the input measures $\alpha_1,\alpha_2,\ldots,\alpha_r$. *Roll-Up* uses one of the following classic SQL aggregation functions, applied to the indicated protected and computed measures $\alpha_1,\alpha_2,\ldots,\alpha_r$ (selected from $\mu_1,\mu_2,\ldots,\mu_k;\tau_1,\ldots,\tau_l;\phi$):

- sum (SUM);
- average (AVG);
- minimum and maximum (MIN and MAX);
- count and count-distinct (COUNT and COUNT-DISTINCT).

We remark that, usually, measures have an associated *default* aggregation function. The typical aggregation function for the measure *sales*, for instance, is SUM.

We denote the above roll-up operation as

$$\text{ROLL-UP}(\mathcal{D},D_i,H(\ell' \to \ell),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\}),$$

where f_i is one of the above aggregation functions that is associated to α_i, for $i = 1,2,\ldots,r$. Since we are mainly interested in the expressiveness of this operation as a sequence of atomic transformations, we remark that only the destination node ℓ in the path h is relevant. Indeed, the result of this roll-up remains the same if the subpath h is extended to start from the *Bottom* node of dimension D_i. So, we can abbreviate the above notation to

$$\text{ROLL-UP}(\mathcal{D},D_i,H(\ell),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\}),$$

and assume that the roll-up starts at the *Bottom* level.

The *Drill-down* operation takes as input a data cube $\mathcal{D}$, a dimension D_i and a subpath h of a hierarchy H over D_i, starting in a node ℓ and ending in a node ℓ' (at a lower level in the hierarchy), and returns the aggregation of the original cube along D_i from the bottom level up to level ℓ'. The drill-down uses the same type of aggregation functions as the roll-up. Again, since we are only interested in the expressiveness of this operation, we remark that the drill-down operation

$$\text{DRILL-DOWN}(\mathcal{D},D_i,H(\ell' \leftarrow \ell),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\}),$$

has the same output as $\text{ROLL-UP}(\mathcal{D},D_i,H(\ell'),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\})$. Therefore, we can limit the further discussion in this section to the roll-up.

We remark that, since we assume, by definition, that dimension graphs are *sound*, we can also omit reference to the hierarchy H in the above notation and simply write $\text{ROLL-UP}(\mathcal{D},D_i,\ell,\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\})$ and $\text{DRILL-DOWN}(\mathcal{D}, D_i, \ell', \{(\alpha_i, f_i) \mid i = 1, 2, \ldots, r\})$, for these OLAP operations.

Definition 24 (ROLLUP). Given a data cube $\mathcal{D}$, one of its dimensions D_i, and a hierarchy H over D_i, ending in a node ℓ, the operation

$$\text{ROLL-UP}(\mathcal{D},D_i,H(\ell),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\})$$

□

computes the aggregation of the measures α_i by their aggregation functions f_i, for $i = 1,2,\ldots,r$, as follows:

$$\alpha_i^{f_i}(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d) = f_i(\{\alpha_i((x_1,\ldots,x_{i-1},y_i,x_{i+1},\ldots,x_d) \mid y_i \in dom(D_i) \text{ and }$$

$$\rho_H(y_i,b)\}),$$

for all $(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_d) \in M(\mathrm{D})$, for which $\rho_H(y_i,b)$, for some $b \in dom(D_{i.}`)$. This roll-up flags all representative *Bottom*-level objects for elements of $dom(D_{i.}`)$ as active.

Property 6. Let D be a data cube, let D_i be one of its dimensions, and let H be a hierarchy over D_i ending in a node `. Let $\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\}$ be a set of selected measures (taken from the protected measures $\mu_1,\mu_2,\ldots,\mu_k$ and the computed measures $\tau_1,\ldots,\tau_k$ of D), with their associated aggregation functions. The operation ROLL-UP$(\mathrm{D},D_i,H(`),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\})$ is expressible as an OLAP operation. □

Proof. We give the proof of this property in Appendix ??. □

We next illustrate the roll-up implementation, using our running example.

Example 21. In this example we simulate the *Roll-Up* operation, using prime (product) labels, sums and projections together with the 3-dimensional sum. We look at the query “total sales per country”. We use the simplified syntax, only indicating the level to which we roll-up on the *Location* dimension (i.e.,
Country). The query

ROLL-UP$(\mathrm{D},Location,Country,\{(sales,\mathrm{SUM})\})$

is the result of the following transformations, given the measure $\mu_1 = sales$:

1. $\tau_{`+1} = \gamma_{Product.Bottom}$ (prime labels on products);
2. $\tau_{`+2} = \gamma_{Time.Bottom}$ (prime labels on time moments);
3. $\tau_{`+3} = \gamma_{Location.Country}$ (prime labels on countries);
4. $\tau_{`+4} = \tau_{`+1} \cdot \tau_{`+2} \cdot \tau_{`+3}$; (prime product label – in one step);
5. $\tau_{`+5} = \mu_1 \cdot \tau_{`+4}$ (product of labels with *sales*);
6. $\tau_{`+6} = \mathrm{SUM}_3(\tau_{`+5})$ (3-dimensional sum);
7. $\tau_{`+7} = \tau_{`+5}|_{\tau_{`+4}}$ (projection on prime product labels); 8. $\phi^{(1)} = \sigma_{Location.Country}$ (output flag on country-representatives).

Transformation $\tau_{`+4}$ gives every product-date-country combination a unique prime product label. Normally this product takes more steps. Above, we have abbreviated it to one transformation. The transformation $\tau_{`+7}$ gives the aggregation result, and $\phi^{(1)}$ is the flag that says that only the cities *antwerp* and *paris*, which represent the level *Country*, are active in the output (and nothing else of the original cube). □

□

We continue with another example of a roll-up operation. To avoid redundancy, we only give highlevel descriptions of its implementation as a sequence of atomic OLAP transformations.

Example 22. Let us consider a rather complex, although usual query in data analysis in real-world situations: "Average sales for cities that are above the country average". The query can be answered using our OLAP transformations as follows:

- Compute the total sale per country (like in Example 13);
- Compute the number of sales per country (see the proof of Property 6);
- Take the quotient of these two values;
- Flag $\sigma_{location.Country}$;
- Compute the total sales over all products and all dates per city;
- Compute the total (non-zero) sales per city;
- Take the quotient of the two previous values;
- Select the cities for which this quotient exceeds the "average sale per country". – Use this last Boolean as an output flag.

5.6. The composition of classical OLAP operations

To conclude this section, we remark that the main result of this paper is the proof of the completeness of an OLAP algebra, composed of the most classical OLAP operations Dice (Section 5.2, Slice (Section 5.3), Slice-and-Dice (Section 5.4), Roll-Up, and Drill-Down (Section 5.5). This result is summarized by the following theorem.

Theorem 1. The classical OLAP operations and their composition are expressible by OLAP operations (that is, as sequences of atomic OLAP transformations).□

Proof. The proof of this theorem follows immediately from the properties in this section, where we have proved that all of these operations can be expressed as a sequence of transformations, whose correctness we had also proved (see Section 4).□

We conclude this section with an example that illustrates the power and generality of our approach, combining a sequence of OLAP operations, and expressing them as a sequence of OLAP transformations.

Example 23. Let us consider an OLAP user, who is analyzing sales in different countries and regions. She wants to analyze and compare sales in the north of Belgium (the Flanders region), and in the south of France (which we, generically, have denoted *south* in our running example). She starts navigating the cube (as we said, indistinctly this can be done through a query language or with a graphic tool), and first filters the cube, keeping just the cells of the two desired regions. This is done with the following expression:

□

DICE(D,*Location.Region = flanders OR Location.Region = south*).

As we showed, this can be implemented as a sequence of atomic OLAP transformations. Now the user has a cube with the cells that do not have been destroyed. Next, within the same navigation process, she obtains the total sales, in France and Belgium, only considering the desired regions, by means of:

ROLL-UP(D,*Location*,*Country*,{(*sales*,SUM)}).

This will only consider the valid cells for rolling up. After this, our user only wants to keep the sales in France (since she is within the same process, she will obviously obtain the sales in the south of France). Thus, she writes (or "clicks"):

DICE(D,*Location.Country = france*).

Finally, she wants to go back to the details, one level below in the hierarchy (that is, the sales in the south of France, the latter being the country she is at, at this stage of her navigation). For this, she does:

DRILL-DOWN(D,*Location*,*Region*,{(*sales*,SUM)})

In our approach, this will be a roll-up from the bottom level to the *Region* level, but only considering the cells that have not been destroyed.

$\square$

6. Conclusion and discussion

We have presented a formal, mathematical approach, to solve a practical problem, which is, to provide a formal semantics to a collection of the OLAP operations most frequently used in real-world practice. Although OLAP is a very popular field in data analytics, this is the first time a formalization like this is given. The need for this formalization is clear: in a world being flooded by data of different kinds, users must be provided with tools allowing them to have an abstract "cube view" and cube manipulation capabilities, regardless of the underlying data types. Without a solid basis and unambiguous definition of cube operations, the former could not be achieved. We claim that our work is the first one of this kind, and will serve as a basis to build more robust practical tools to address the forthcoming challenges in this field.

We have addressed the four core OLAP operations: slice, dice, roll-up, and drill-down. This does not harm the value of the work. On the contrary, this approach allows us to focus on our main interest, that is, to study the formal basis of the problem. Our line of work can be extended to address other kinds of OLAP queries, like queries involving more complex aggregate functions like moving averages, rankings, and the like. Further, cube combination operations, like drill-across, must be included in the picture. We believe that our contribution provides a solid basis upon which, a complete OLAP theory can be built.

Acknowledgments

Alejandro Vaisman was supported by a travel grant from Hasselt University (Korte verblijven–inkomende mobiliteit, BOF15KV13). He was also partially supported by PICT-2014 Project 0787.

References

[1] R. Agrawal, A. Gupta and S. Sarawagi, Modeling multidimensional databases, In *Proceedings of the 15th International Conference on Data Engineering, (ICDE)*, Birmingham, UK, 1997, IEEE Computer Society, pp. 232–243.

[2] C. Ciferri, R. Ciferri, L. Gómez, M. Schneider, A. Vaisman and E. Zimányi, Cube algebra: A generic user-centric model and query language for OLAP cubes, *International Journal of Data Warehousing and Mining* 9(2) (2013), 39–65.

[3] F. Dehne, Q. Kong, A. Rau-Chaplin, H. Zaboli and R. Zhou, Scalable real-time OLAP on cloud architectures, *Journal of Parallel and Distributed Computing* 7980 (2015), 31– 41.

[4] J.-P. Escofier, *Galois Theory, volume 204 of Graduate Texts in Mathematics*, Springer-Verlag, 2001.

[5] F. Ravat, O. Teste, R. Tournier and G. Zurfluh, Algebraic and graphic languages for OLAP manipulations, *International Journal of Data Warehousing and Mining* 4(1) (2008), 17–46.

[6] L. Gómez, S. Gómez and A. Vaisman, A generic data model and query language for spatiotemporal OLAP cube analysis, In *Proceedings of the 15th International Conference on Extending Database Technology, EDBT 2012*, Berlin, Germany, 2012, pp. 300–311.

[7] M. Gyssens and L. Lakshmanan, A foundation for multi-dimensional databases, In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB*, Athens, Greece, 1997, pp. 106–115.

[8] S. Harinath, R. Pihlgren, D.-Y. Lee, J. Sirmon and R. Bruckner, *Professional Microsoft SQL Server 2012 Analysis Services with MDX and DAX*, Wrox, 2012.

[9] R. Kimball, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouse*, Wiley, 1996.

[10] H.D. Macedo and J.N. Oliveira, A linear algebra approach to OLAP, *Formal Aspects of Computing* 27(2) (2015), 283–307.

[11] O. Romero and A. Abelló, On the need of a reference algebra for OLAP, In *Proceedings of the 9th International Conference on Data Warehousing and Knowledge Discovery, DaWaK'07*, Regensburg, Germany, 2007, pp. 99–110.

[12] C. Stolte, D. Tang and P. Hanrahan, Polaris: a system for query, analysis, and visualization of multidimensional databases, *Communications of ACM* 51(11) (2008), 75–84.

[13] A. Vaisman and E. Zimányi, *Data Warehouse Systems: Design and Implementation*, Springer, 2014.

[14] J. Varga, L. Etcheverry, A. Vaisman, O. Romero, T.B. Pedersen and C. Thomsen, Enabling OLAP on statistical linked open data, In *Proceedings of the 32nd International Conference of Data Engineering, (ICDE)*, Helsinki, Finland, 2016, pp. 1346–1349.

[15] P. Vassiliadis, Modeling multidimesional databases, cubes and cube operations, In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management, (SSDBM)*, Capri, Italy, 1998, pp. 53–62.

[16] G. Viswanathan and M. Schneider, Bigcube: A metamodel for managing multidimensional data, In *ISCA 19th International Conference on Software Engineeringand Data Engineering (SEDE)*, San Francisco, CA, USA, 2010, pp. 237–242.

[17] G. Viswanathan and M. Schneider, On the requirements for user-centric spatial data warehousing and SOLAP, in: *Proceedings of the DASFAA 2011 Workshops, volume 6637 of Lecture Notes in Computer Science*, J. Xu, G. Yu, S. Zhou and R. Unland, eds, Hong Kong, China, Springer, 2011, pp. 144–155.

Appendix A Proof of property 2

We now give the proof of Property 2. We repeat the property here, to facilitate reading.

Property 2. Let $n > 1$ and let $A_1 \times A_2 \times \cdots \times A_n$ be a cartesian product of finite sets. We assume that the cells $(a_1,a_2,\dots,a_n)$ of this set contain rational values $\mu(a_1,a_2,\dots,a_n)$ of a measure μ. Let I be a subset of $\{1,2,\dots,n\}$ and let w_i be prime labelings of the sets A_i, for $i \in I$, that form a prime product I-labelling (see Definition 16). Then we have that the prime sum (†2) uniquely determines the values $\sum_{\times_{i \in I^c} A_i} \mu(a_1,a_2,\dots,a_n)$ for all cells of $A_1 \times A_2 \times \cdots \times A_n$. □

Proof. First, we assume I contains one element. Without loss of generality, we may assume that $I = \{1\}$. Then the prime sum (over I) is

$$\sum_{(a_1,a_2,\dots,a_n)\in A_1\times A_2\times\cdots\times A_n} \mu(a_1,a_2,\dots,a_n)\cdot w_1(a_1) =$$

$$\sum_{a_1\in A_1}\left(\sum_{(a_2,\dots,a_n)\in A_2\times\cdots\times A_n} \mu(a_1,a_2,\dots,a_n)\right)\cdot w_1(a_1).$$

Let us assume this sum is equal to

$$\sum_{a_1\in A_1}\left(\sum_{(a_2,\dots,a_n)\in A_2\times\cdots\times A_n} \mu'(a_1,a_2,\dots,a_n)\right)\cdot w_1(a_1)$$

for some measure μ^0 and that there exists a $a_0 \in A_1$ such that

$$\text{X}_{(a_2,\ldots,a_n)\in A_2\times\cdots\times A_n} \quad 6= \quad \text{X}_{(a_2,\ldots,a_n)\in A_2\times\cdots\times A_n} \quad \mu_0(a_0,a_2,\ldots,a_n).\ \mu(a_0,a_2,\ldots,a_n)$$

Since all $\mu(a_1,a_2,\ldots,a_n)$ and $\mu^0(a_1,a_2,\ldots,a_n)$ are assumed to be rational numbers, this implies that $w_1(a_0)$ is a rational combination of the other labels $w_1(a_1)$, with $a_1 \in A_1 \setminus \{a_0\}$. Since the labels

w the field extension $_1(a_1\sqrt{})$, with$\sqrt{\ a_1}\in\sqrt{}\mathbf{Q}A_1(\sqrt{}$, are square roots of different prime numbers, this leads to a contradiction, since$2,\sqrt{3},\ldots,\sqrt{p_n})$, for any n, has degree $2n$ over $\mathbf{Q}$. In other words, the square

roots 2, 3,..., p_n (together with 1) are linearly independent over $\mathbf{Q}$ (see Chapter 8 in [4]).

When the cardinality of I is strictly larger than 1, we can use a similar argumentation. Then we work with prime product labels of the form $\text{Q}_{i\in I}\ w_i(a_i)$. Because of the restrictions on these products, imposed by Definition 16 (injectivity of the labelling function per dimension and disjointness of labels between dimensions), we see that these product labels differ one from the other by at least one prime factor (under the square root). Therefore, these labels are also linearly independent over $\mathbf{Q}$ [4]. This completes the proof. □

Appendix B Proof of lemma 1

Lemma 1. If $\varphi,\varphi_1,\varphi_2$ are transformation-expressible Boolean conditions on cells, then NOT φ, φ_1 AND φ_2 and φ_1 OR φ_2 are transformation-expressible Boolean conditions on cells. □

Proof. Obviously, a Boolean combination of Boolean conditions is a Boolean condition. Let us assume that φ, φ_1 and φ_2 are transformation-expressible by sequences of OLAP transformations that end in τ_k, τ_{k1} and τ_{k2}, respectively. Then φ_1 AND φ_2 can be expressed by the transformation $\tau_m = \tau_{k1}\cdot\tau_{k2}$, which is 1 on cells if and only if both φ_1 and φ_2 give 1 on those cells.

For the negation, we have the following sequence of additional transformations:

– $\tau_m = 1$ (rational constant);

– $\tau_{m+1} = -1$ (rational constant); – $\tau_{m+2} = \tau_{m+1}\cdot\tau_k$ (product); and – $\tau_{m+3} = \tau_m + \tau_{m+2}$ (sum).

Here, we simulate substraction using the sum. The transformation τ_{m+3} equals $\tau_m - \tau_k$ and turns $\tau_k = 0$ into 1 and a $\tau_k = 1$ into 0. So, the transformation τ_{m+3} expresses NOT φ.

Via de Morgan's law, we can express φ_1 OR φ_2 using conjunction and negation. An alternative implementation of the OR is given by $\tau_m = \tau_{k1}+\tau_{k2}$ (this sum gives 0, 1 or 2); and $\tau_{m+1} = \tau_m/\tau_m$. This last transformation maps 1 and 2 on 1 and 0 on 0 (in Definition 11, we defined $0/0$ to be 0). □

Appendix C Proof of property 3

Property 3. Let D be a data cube en let φ be a Boolean condition on the cells of $M(\mathrm{D})$ (as in Definition 21). The operation DICE(D,φ) is expressible as an OLAP operation. □

Proof. Since DICE(D,φ) is a cell-selecting operation, it suffices, by Lemma 1, to show that DICE(D,φ) is expressible by an atomic Boolean cell-selection condition φ (without logical connectives).

We have to consider the three cases of Definition 21.

For the first case, DICE(D,φ) is simply expressed by the selector $\tau_{l+1} = \sigma_{D_i.`=b}$, which is the output flag that indicates the appropriate cells of $M(\mathrm{D})$.

For the second case, if `is a level from a dimension schema $\sigma(D_i)$ of a dimension $D_i$ and $c \in dom(D_i.`)$ and φ is of the form $` < c$ or $c <$ `, then the comparison test on levels $\tau_{l+1} = (` <_` c)$ (or $\tau_{l+1} = (c <_` `)$), expresses DICE(D,φ). Again, τ_{l+1} specifies the output flag.

For the third case, if α is some measure, then $\tau_{l+1} = c$ (rational constant), followed by $\tau_{l+2} = (\alpha = c)$, $\tau_{l+2} = (\alpha < c)$ or $\tau_{l+2} = (\alpha > c)$ (equality or comparison test on a measure), respectively, express DICE(D,φ). Once again, τ_{l+2} can serve as the output flag. This concludes the proof. □

Appendix D Proof of property 4

Property 4. Let D be a data cube and let D_i be one of its dimensions. The operation SLICE(D,D_i) is expressible as an OLAP operation. □

Proof. Let D be a data cube, and D_i be one of its dimensions. The operation SLICE(D,D_i) is expressible in the OLAP algebra by the following sequence of transformations:

- $\tau_{l+1} = \gamma_{D_1.Bottom}$ (prime labels on dimension D_1); – ...
- $\tau_{l+1+i-2} = \gamma_{D_{i-1}.Bottom}$ (prime labels on dimension D_i-1);
- $\tau_{l+1+i-1} = \gamma_{D_{i+1}.Bottom}$ (prime labels on dimension D_i+1); – ...
- $\tau l+1+d-2 = \gamma_{D_d.Bottom}$ (prime labels on dimension Dd);
- $\tau l+1+d-1 = \tau l+1 \cdot \tau l+1+1$;
- $\tau l+1+d = \tau l+1+d-1 \cdot \tau l+1+2$; – ...
- $\tau_{l+1+2d-4} = \tau_{l+1+d-1} \cdot \tau_{l+1+d-2}$ (product of all prime labels);
- $\tau_{l+1+2d-3} = \mu_1 \cdot \tau_{l+1+2d-4}$ (product of measure with product of all prime labels); – ...
- $\tau_{l+1+2d+k-4} = \mu_k \cdot \tau_{l+1+2d-4}$ (product of measure with product of all prime labels);
- $\tau l+1+2d+k-3 = \mathrm{SUM}d(\tau l+1+2d-3)$ (d-dimensional sum); – ...
- $\tau_{l+1+2d+2k-4} = \mathrm{SUM}_d(\tau_{l+1+2d+k-4})$ (d-dimensional sum);
- $\tau l+1+2d+2k-3 = \tau l+1+2d+k-3 \,|_{\tau_{l+1+2d-4}}$ (projection on product labels); – ...
- $\tau l+1+2d+3k-4 = \tau l+1+2d+2k-4 \,|_{\tau_{l+1+2d-4}}$ (projection on product labels);
- $\tau_{l+1+2d+3k-3} = \sigma_{D_i.All}$ (selects the representative of All for dimension D_i);

– $\delta = \tau_{l+1+2d+3k-3}$ (destroyer);
– $\phi^{(k)} = \tau_{l+1+2d+3k-3}$ (output flag).

Transformations $\tau_{l+1},\ldots,\tau_{l+1+d-2}$ create (fresh) prime labels for each of the dimensions $D_1,\ldots, D_{i-1},D_{i+1},\ldots,D_d$. Transformation $\tau_{l+1+2d-4}$ gives the product of all these prime labels. This means that every $(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_d) \in dom(D_1) \times \cdots \times dom(D_{i-1}) \times dom(D_{i+1}) \times \cdots \times dom(D_d)$ has a unique prime product label, that is shared by all cells above the projected cell $(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_d)$ in the direction of the dimension D_i. Transformations $\tau_{l+1+2d-3},\ldots, \tau_{l+1+2d+k-4}$ multiply the measures $\mu_1,\mu_2,\ldots,\mu_k$ with the prime product label. Transformations $\tau_{l+1+2d+k-3},\ldots,\tau_{l+1+2d+2k-4}$ make partial prime sums of the measures $\mu_1,\mu_2,\ldots,\mu_k$ over the complete matrix $M(\mathrm{D})$. The last k transformations $\tau_{l+1+2d+2k-3},\ldots,\tau_{l+1+2d+3k-4}$ project on the primeproduct-labels giving each cell above $(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_d)$ the sum of the k measures above it. Finally, the destroyer δ and the output flag $\phi^{(k)}$ select the representative of All for dimension D_i and make sure that the other cells of $M(\mathrm{D})$ are destroyed. The output, for cells that are not destroyed, is

$\underline{\mu_1},\underline{\mu_2},\ldots,\underline{\mu_k};\tau_{l+1+2d+2k-3},\ldots,\tau_{l+1+2d+3k-4};\phi^{(k)}$,

which is renamed to

$\underline{\mu_1},\underline{\mu_2},\ldots,\underline{\mu_k};\tau_1,\ldots,\tau_k;\phi$.

For 1 6 n 6 k, $\tau_n = \mu_n^{\Sigma_i}$ is the desired aggregate value. This concludes the proof. □

Appendix E Proof of property 5

Property 5. Let D be a data cube, D_i on of its dimensions en let $a \in dom(D_i)$. The operation SLICE-DICE(D,D_i,a) is expressible as an OLAP operation. □

Proof. Let D be a data cube, D_i on of its dimensions en let $a \in dom(D_i)$. The selector $\sigma_{D_i.Bottom=a}$ is the transformation that serves as destroyer and output flag and that expresses SLICE-DICE(D,D_i,a). This concludes the proof. □

Appendix F Proof of Property 6

Property 6. Let D be a data cube, let D_i be one of its dimensions, and let H be a hierarchy over D_i ending in a node `. Let $\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\}$ be a set of selected measures (taken from the protected measures $\mu_1,\mu_2,\ldots,\mu_k$ and the computed measures $\tau_1,\ldots,\tau_k$ of D), with their associated aggregation functions. The operation ROLL-UP$(\mathrm{D},D_i,H(`),\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\})$ is expressible as an OLAP operation.

Proof. Let D be a data cube, let D_i be one of its dimensions, and let H be a hierarchy over D_i ending in a node `. Let $\{(\alpha_i,f_i) \mid i = 1,2,\ldots,r\}$ be a set of selected measures with their associated aggregation functions.

We start by remarking that the aggregations of the measures α_i by the functions f_i, can be computed consecutively for $i = 1,2,...,r$. At the end their results are copied as the last r computed measures, and an output flag of type $\phi^{(r)}$, which is a selector $\sigma_{D_i.`}$, returns these r aggregation results as output.

Now, it remains to be shown how the SQL aggregation functions SUM, AVG, MIN, MAX, COUNT and COUNT-DISTINCT can be implemented as sequences of atomic OLAP transformations for an arbitrary measure α.

(1) SUM: We give a description of the implementation of the SUM by a sequence of atomic OLAP transformations. Since a detailed description of a similar procedure was given in the proof of Property 4, we refer to that proof for details.
Here, we first create prime labels $\gamma_{D_j.Bottom}$ for all j 6= i and, for dimension D_i, prime labels $\gamma_{D_i.`}$ at the level `. Next, we create a measure that is the product of all these prime labels. This prime product label gives each cell $(x_1,...,x_{i-1},y_i,x_{i+1},...,x_d)$ of the matrix a unique label, modulo rolling-up to the same object at level ` for the dimension $D_i$. This implies that $(x_1,...,x_{i-1},y_i,x_{i+1},...,x_d)$ and $(x_1,...,x_{i-1},y_i^0,x_{i+1},...,x_d)$, for which there is a $b \in dom(D_i.`)$ such that $\rho_H(y_i,b)$ and $\rho_H(y_i^0,b)$, get the same prime product label. Then we take the d-dimensional sum of the product of this prime product label with α. The projection on the prime product label, gives the desired result. That is, the cells $(x_1,...,x_{i-1},y_i,x_{i+1},...,x_d)$ and $(x_1,...,x_{i-1},y_i^0,x_{i+1},...,x_d)$, for which there is a $b \in dom(D_i.`)$ such that $\rho_H(y_i,b)$ and $\rho_H(y_i^0,b)$, get the same aggregation (sum) value of α over all objects that roll-up to b. For a detailed description, we refer to the proof of Property 4 and for an illustration, we refer to Example 21.

(2) COUNT: Here, we proceed as in the case of SUM, except that, before taking the sum, we do not multiply the prime product labels with α, but with 1. We can count the cells for which α is nonzero, by multiplying the prime product labels by the quotient α/α, rather than by 1. We remark that by the definition of quotient, we know that $0/0 = 0$, which implies that the cells with a zero value for α are not counted.

(3) AVG: The aggregation function AVG can be implemented using the implementation of SUM, followed by the implementation of COUNT (counting all or all non-zeroes) and then computing the quotient of these two values.

(4) MIN and MAX: As in the case of SUM, we create prime product labels for all cells of $M(\mathrm{D})$. Let us call this prime product labels τ_m. Then we multiply this prime product labels by α, resulting in the measure τ_{m+1}. Next, we apply the generalised form of the maximum (or minimum) transformation $\max|_{\tau_m}(\tau_{m+1})$ to obtain the maximum value of α per prime product label. Similarly, $\min|_{\tau_m}(\tau_{m+1})$ gives the desired minimal values.

(5) COUNT-DISTINCT: We proceed as in the case of MIN and MAX, but now we obtain the result by the transformation $\#_{6=}|_{\tau_m}(\tau_{m+1})$, which is the generalized form of the Count-Distinct. This concludes the proof.

□