Information Systems 34 (2009) 551–576

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Spatial aggregation: Data model and implementation

Leticia Gómez [c], Sophie Haesevoets [a], Bart Kuijpers [b], Alejandro A. Vaisman [b,d,*]

[a] Luciad NV, Belgium
[b] Hasselt University and Transnational University of Limburg, Belgium
[c] Instituto Tecnológico de Buenos Aires, Argentina
[d] Universidad de Buenos Aires, Argentina

ARTICLE INFO

Article history:
Received 25 May 2008
Received in revised form
13 December 2008
Accepted 11 March 2009
Recommended by: L. Wong

Keywords:
Data warehousing
OLAP
GIS
Aggregation

ABSTRACT

Data aggregation in Geographic Information Systems (GIS) is a desirable feature, only marginally present in commercial systems nowadays, mostly through ad hoc solutions. We address this problem introducing a formal model that integrates, in a natural way, geographic data and non-spatial information contained in a data warehouse external to the GIS. This approach allows both aggregation of geometric components and aggregation of measures associated to those components, defined in GIS fact tables. We define the notion of *geometric aggregation*, a general framework for aggregate queries in a GIS setting. Although general enough to express a wide range of (aggregate) queries, some of these queries can be hard to compute in a real-world GIS environment because they involve computing an integral over a certain area. Thus, we identify the class of *summable* queries, which can be efficiently evaluated replacing this integral with a sum of functions of geometric objects. Integration of GIS and OLAP (On Line Analytical Processing) is supported also through a language, GISOLAP-QL. We present an implementation, denoted *Piet*, which supports four kinds of queries: standard GIS, standard OLAP, geometric aggregation (like "total population in states with more than three airports"), and integrated GIS-OLAP queries ("total sales by product in cities crossed by a river", also allowing navigation of the results). Further, Piet implements a novel query processing technique: first, a process called *subpolygonization* decomposes each thematic layer in a GIS, into open convex polygons; then, another process (the *overlay precomputation*) computes and stores in a database the overlay of those layers for later use by a query processor. Experimental evaluation showed that for a wide class of geometric queries, overlay precomputation outperforms R-tree-based techniques, suggesting that it can be an alternative for GIS query processing.

1. Introduction

Geographic Information Systems (GIS) have been extensively used in various application domains, ranging from economical, ecological and demographic analysis, to city and route planning [41,49]. Spatial information in a GIS is typically stored in different so-called *thematic layers* (or *themes*). Information in themes is generally stored in Object-Relational databases, where spatial data (i.e., geometric objects) are associated to attribute information, of numeric or string type. Spatial data in the different thematic layers of a GIS system can be mapped univocally to each other using a common frame of reference, like a coordinate system. These layers can be overlapped or overlayed to obtain an integrated spatial view.

OLAP (On Line Analytical Processing) [24] comprises a set of tools and algorithms that allow efficiently querying multidimensional databases, containing large amounts of

* Corresponding author at: Universidad de Buenos Aires, Argentina. Tel./fax: +54 11 4902 0421.
E-mail addresses: lgomez@itba.edu.ar (L. Gómez), sofie.haesevoets@luciad.com (S. Haesevoets), bart.kuijpers@uhasselt.be (B. Kuijpers), avaisman@dc.uba.ar (A.A. Vaisman).

0306-4379/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2009.03.002

data, usually called data warehouses. In OLAP, data are organized as a set of *dimensions* and *fact tables*. In this multidimensional model, data can be perceived as a *data cube*, where each cell contains a measure or set of (probably aggregated) measures of interest. OLAP dimensions are further organized in hierarchies that favor the data aggregation process [4]. Several techniques and algorithms have been developed for query processing, most of them involving some kind of aggregate precomputation [19] (an idea we will use later in this paper).

1.1. Problem statement

Nowadays, organizations need sophisticated GIS-based decision support system (DSS) to analyze their data with respect to geographic information, represented not only as attribute data, but also in maps, probably in different thematic layers. In this sense, OLAP and GIS vendors are increasingly integrating their products.[1] Aggregate queries are central to DSSs. Thus, classical aggregate queries (like "total sales of cars in California"), and aggregation combined with complex queries involving geometric components ("total sales in all villages crossed by the Mississippi river within a radius of 100 km around New Orleans") must be efficiently supported. Moreover, navigation of the results using typical OLAP operations like roll-up or drill-down is also required. These operations are not supported by commercial GIS in a straightforward way. *First,* GIS data models were developed with "transactional" queries in mind. Thus, the databases storing non-spatial attributes or objects are designed to support those (non-aggregate) kinds of queries. DSSs need a different data model, where non-spatial data, consolidated from different sectors in an organization, is stored in a data warehouse. Here, numerical data are stored in fact tables built along several dimensions. For instance, if we are interested in the sales of certain products in stores in a given region, we may consider the sales amounts in a fact table over the dimensions Store, Time and Product. Moreover, in order to guarantee summarizability [28], dimensions are organized into aggregation hierarchies. For example, stores can aggregate over cities which in turn can aggregate into regions and countries. Each of these aggregation levels can also hold descriptive attributes like city population, the area of a region, etc. To fulfill the requirements of integrated GIS-DSS, warehouse data must be linked to geographic data. For instance, a polygon representing a region must be associated to the region identifier in the warehouse. *Second,* system integration in commercial GIS is not an easy task. The GIS and OLAP worlds must be integrated in an ad hoc fashion, in a different way each time an implementation is required. In this paper we address this problem, and introduce a framework which naturally integrates the GIS and OLAP worlds. We also describe an implementation of this proposal, denoted Piet, and show the advantages of this approach.

[1] See Microstrategy and MapInfo integration in http://www.microstrategy.com/, http://www.mapinfo.com/.

1.2. An introductory example

Consider the following real-world example, which we also use in our experiments. We selected four layers with geographic and geological features obtained from the National Atlas Website.[2] These layers contain states, cities, and rivers in North America, and volcanoes in the northern hemisphere (published by the Global Volcanism Program—GVP). Cities and volcanoes are represented as points, rivers as polylines, and states as polygons. There is also non-spatial information stored in a conventional data warehouse, where dimension tables contain customer, stores and product information, and a fact table contains stores sales across time. Also numerical and textual information on the geographic components exist (e.g., population, area), stored as usual in a GIS. In this scenario, conventional GIS and organizational data can be integrated for decision support analysis. Sales information could be analyzed in the light of geographical features, conveniently displayed in maps. We show that this analysis could benefit from the integration of both worlds in a single framework. Even though this integration could be possible with existing technologies, ad hoc solutions are expensive because, besides requiring lots of complex coding, their are hardly portable. To make things more difficult, ad hoc solutions require data exchange between GIS and OLAP applications to be performed. This implies that the output of a GIS query must be exported as a dimension of a data cube, and merged for further analysis. For example, suppose that a business analyst is interested in studying the sales of nautical goods in stores located in cities crossed by rivers. She has a map that comprises two layers, one for rivers (represented as polylines) and other for cities (represented as polygons). She also has stored sales in a data cube, containing a dimension *Store* or *Geography* with *city* as a dimension level. She would first query the GIS, to obtain the cities of interest. Then, she would need to 'manually' select in the data cube the cities of interest, and export them to a spreadsheet, to be able to go on with the analysis (in the best case, an ad hoc customized middleware could help her). Of course, she must repeat this for each query involving a (geographic) dimension in the data cube. On the contrary, using *Piet* (the system we propose in this paper), she only needs to bind geographic elements in the maps, to the existing data cube(s) that integrate organizational information, and the system is ready to receive her queries. The results are displayed for further drilling down/rolling up (see Figs. 9 and 10 in Section 6 which discusses these kinds of queries in detail). Further, ad hoc improvements could be also performed during the Extraction, Transformation, and Loading (ETL) stage, like, for instance, adding an attribute indicating if the city is crossed by a river, avoiding computing the intersection each time a query is posed.

[2] http://www.nationalatlas.gov

1.3. Contributions and paper organization

After providing a brief background on previous approaches to the interaction between GIS and OLAP (Section 2), we propose a formal model for spatial aggregation that supports efficient evaluation of aggregate queries in spatial databases based on the OLAP paradigm (Section 3). This model is aimed at integrating GIS and OLAP in a unique framework for decision support. We assume that non-spatial data are stored in data warehouses [24], created and maintained separately from the GIS, a so-called loosely coupled approach. We formally define the notion of *geometric aggregation* that characterizes a wide range of aggregate queries over regions defined as semi-algebraic sets. We show that our proposal supports aggregation of geometric components, aggregation of measures associated with these components, and aggregation of measures defined in data warehouses, external to the GIS. Although this theoretical framework is general enough to express many interesting queries, practical problems can be solved without the need of dealing with geometries and semi-algebraic sets, which can be difficult and computationally expensive. Thus, as our second contribution, we identify a class of queries that we denote *summable* (Section 4), allowing geometric aggregation to be performed without resorting to coordinates and geometric union algorithms. We formally study summable queries, and define when a geometric aggregate query is or is not summable.[3] Usually, summable queries involve overlapping thematic layers. We show that summable queries can be efficiently evaluated over a new layer that contains the pre-computation of the overlay (i.e., the spatial join) of the individual thematic layers that are involved in a query, following the paradigm of view materialization in OLAP [19] (Section 5). We call this subdivision the *common subpolygonization* of the plane. Our ultimate idea is to provide a working alternative to standard R-tree-based query processing that could be included in a query processor as another strategy for performing spatial joins with or without aggregation. As a particular application of these ideas, we discuss *topological* aggregation queries, and sketch how they can be efficiently evaluated using a topological invariant instead of geometric elements.

We also present *Piet*, an open source implementation of our proposal (named after the Dutch painter Piet Mondrian), along with experimental results showing that, contrary to the usual belief [18], precomputing the *common subpolygonization* can successfully compete with typical R-tree-based solutions used in most commercial GIS (Section 7). The *Piet* software architecture is prepared to support not only overlay precomputation for query processing, but R-Trees and aR-Trees [33] as well. Our implementation provides a smooth integration between OLAP and GIS applications. Of course, this integration cannot prevent that queries are executed by two engines: a GIS engine, and an OLAP one (there is no way to avoid this with existing technology). Integration is thus provided at a higher abstraction level. Piet supports four kinds of queries: (a) standard GIS queries; (b) standard OLAP queries; (c) geometric aggregation queries (“total population in states with more than three airports”); (d) integrated GIS-OLAP queries (“total sales by product in cities crossed by a river”). OLAP-style navigation is also supported in the latter case. Queries can be submitted from a graphical interface, or written in GISOLAP-QL, a language sketched in Section 6. We finally report experimental results (Section 8). Piet software, query demonstrations, and experiments over other maps, not shown here, are available on the project’s Web site.[4]

Remark 1. An extended abstract describing the implementation of Piet appeared in [10]. The present paper substantially extends the former one, by adding a detailed description of the formal model (only an overview was provided in [10]) and of the implementation, a more in depth analysis related work, and a different set of experiments. The latter is relevant to the validation of the approach, given the different characteristics of both sets of maps.

2. Background and related work

In the last five years, the topic of spatial OLAP and spatio-temporal OLAP, has been attracting the attention of the database and GIS communities. In this section we provide a short background on the notion of Spatial OLAP (SOLAP), and review previous work on conceptual modeling and implementation, comparing this existing approaches to our proposal.

The concept of SOLAP. Vega López et al. [48] present a comprehensive survey on spatiotemporal aggregation. Also, Bédard et al. [2] present a review of the efforts for integrating OLAP and GIS. Rivest et al. [42] introduced the concept of Spatial OLAP, a paradigm aimed at being able to explore spatial data by drilling on maps, as it is performed in OLAP with tables and charts. They describe the desirable features and operators a SOLAP system should have. Although they do not present a formal model for this, SOLAP concepts and operators have been implemented in a commercial tool called JMAP.[5]Related to the concept of SOLAP, Shekhar et al. [43] introduced MapCube, a visualization tool for spatial data cubes. MapCube is an operator that, given a so-called base map, cartographic preferences and an aggregation hierarchy, produces an album of maps that can be navigated via roll-up and drill-down operations.

Conceptual modeling. Stefanovic et al. [45] and Bédard et al. classify spatial dimension hierarchies according to their spatial references in: (a) non-geometric; (b) geometric to non-geometric; and (c) fully geometric. Dimensions of type (a) can be treated as any descriptive

[3] It will become clear that the notion of summability differs from the concept of summarizability studied in [28,44]. Summability allows replacing an integral by a sum. However, summarizability must be preserved for a query to be summable.

[4] http://piet.exp.dc.uba.ar/piet/index.jsp.

[5] http://www.kheops-tech.com/en/jmap/solap.jsp.

dimension. In dimensions of types (b) and (c), a geometry is associated to members of the hierarchies. Malinowski and Zimányi [30], in the *Multidim* model, extend this classification to consider that even in the absence of several related spatial levels, a dimension can be considered spatial. Here, a dimension level is spatial if it is represented as a spatial data type (e.g., point, region), allowing them to link spatial levels through topological relationships (e.g., contains, overlaps). Thus, a spatial dimension is a dimension that contains at least one spatial hierarchy. This model is an extension of previous conceptual model for OLAP introduced by the same authors, based on the well-known Entity-Relationship model [29]. In the models above, spatial measures are characterized in two ways, namely: (a) measures representing a geometry, which can be aggregated along the dimensions; (b) a numerical value, using a topological or metric operator. Most proposals support option (a), either as a set of coordinates [1,3,30,42], or a set of pointers to geometric objects [45]. In particular, in [30], the authors define measures as attributes of an n-ary fact relationship between dimensions. Further on, the same authors present a method to transform a conceptual schema to a logical one, expressed in the Object-Relational paradigm [31]. Fidalgo et al. [11] and da Silva et al. [46] introduced GeoDWFrame, a framework for spatial OLAP, which classifies dimensions as geographic and hybrid, if they represent only geographic data, or geographic and non-spatial data, respectively. Over this framework, da Silva et al. [47] propose GeoMDQL, a query language based on MDX and OGC[6] simple features, for querying spatial data cubes. It is worth noting that all these conceptual models follow what we can denote a *tightly coupled* approach between the GIS and OLAP components, given that the spatial objects are included in the data warehouse. On the contrary, we follow a *loosely coupled* approach, where GIS maps and data warehouses are maintained in a separate fashion, and bound by a matching function. We believe that this approach favors autonomy, updating and maintenance of the databases. Pourabas [39] introduced a conceptual model that uses binding attributes to bridge the gap between spatial databases and a data cube. No implementation of the proposal is discussed. Besides, this approach relies on the assumption that all the cells in the cube contain a value, which is not the usual case in practice, as the author expresses. Moreover, the approach also requires modifying the structure of the spatial data. This is not the case in our proposal, which does not require to modify existing data structures.

Implementation. Han et al. [17] use OLAP techniques for materializing selected spatial objects, and proposed a so-called *spatial data cube*. This model only supports aggregation of such spatial objects. Pedersen and Tryfona [38] propose pre-aggregation of spatial facts. First, they pre-process these facts, computing their disjoint parts in order to be able to aggregate them later, given that pre-aggregation works if the spatial properties of the objects are distributive over some aggregate function. This proposal ignores the geometry. Thus, queries like "Give me the total population of cities crossed by a river" are not supported. The paper does not address forms other than polygons, although the authors claim that other more complex forms are supported by the method. No experimental results are reported. Extending this model with the ability to represent partial containment hierarchies (useful for a location-based services environment), Jensen et al. [23] proposed a multidimensional data model for mobile services, i.e., services that deliver content to users, depending on their location. This model also omits considering the geometry, limiting the set of queries that can be addressed. With a different approach, Rao et al. [40], and Zang et al. [50] combine OLAP and GIS for querying so-called spatial data warehouses, using R-trees for accessing data in fact tables. The data warehouse is then evaluated in the usual OLAP way. Thus, they take advantage of OLAP hierarchies for locating information in the R-tree which indexes the fact table. Here, although the measures are not spatial objects, they also ignore the geometric part, limiting the scope of the queries they can address. It is assumed that some fact table, containing the ids of spatial objects exists. Moreover, these objects happen to be just points, which is quite unrealistic in a GIS environment, where different types of objects appear in the different layers. Other proposals in the area of indexing spatial and spatio-temporal data warehouses [33,34] combine indexing with pre-aggregation, resulting in a structure denoted *aggregation R-tree* (aR-tree), an R-tree that annotates each MBR (minimum bounding rectangle) with the value of the aggregate function for all the objects that are enclosed by it. We implemented an aR-tree for experimentation (see Section 8). This is a very efficient solution for some particular cases, specially when a query is posed over a query region whose intersection with the objects in a map must be computed on-the-fly. However, problems may appear when leaf entries partially overlap the query window. In this case, the result must be estimated, or the actual results computed using the base tables. Kuper and Scholl [27] suggest the possible contribution of constraint database techniques to GIS. Nevertheless, they do not consider spatial aggregation, nor OLAP techniques.

In summary, although the proposals above address particular problems, as far as we are aware of this isthe first work to address the problem as a whole, from the formal study of the problem of integrating spatial and warehousing information in a single framework that allows to obtain the full potential of this integration, discussed in the first part of this paper, to the implementation of the proposal, presented in the second part, along with the discussion of practical and implementation issues, together with experimental results that validate our approach.

3. Spatial aggregation

Our proposal is aimed at integrating, in a single model, spatial and non-spatial information, probably produced independently from each other. We assume, without loss

[6] Open Geospatial Consortium, http://www.opengeospatial.org.

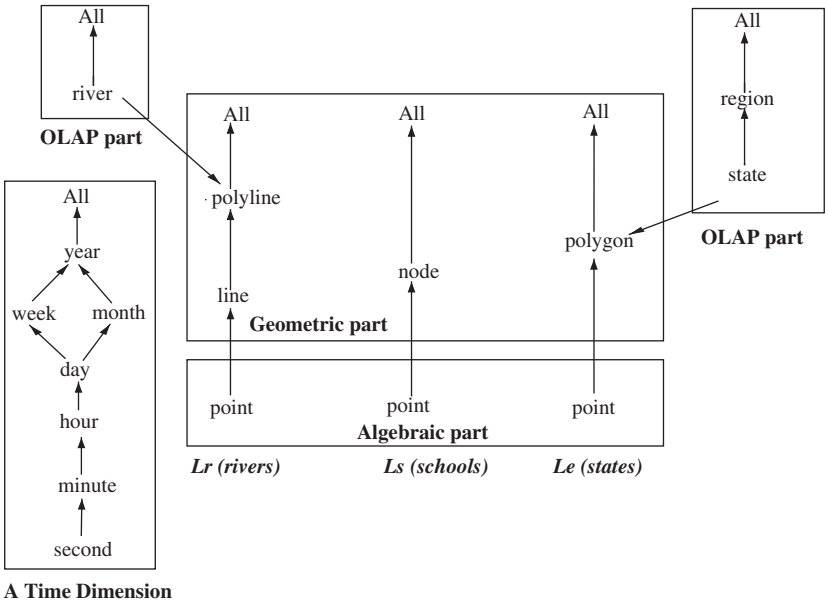

Fig. 1. An example of a *GIS* dimension schema.

of generality, that non-spatial data are stored in a data warehouse following the standard OLAP notion of dimension hierarchies and fact tables [4,22]. We denote this approach *loosely coupled*, given that warehouse and spatial data are maintained independently from each other. The main component of the model is denoted a *GIS dimension*. A GIS dimension consists, as usual in databases, in a dimension schema that describes the dimension's structure, and dimension instances. Each dimension is composed of a set of graphs, each one describing a set of geometries in a thematic layer. Fig. 1 shows a GIS dimension schema, with three graphs representing three different layers, following our running example: rivers (L_r), volcanoes (L_v), and states (L_e). We define three sectors, denoted the *algebraic part*, the *geometric part*, and the *classical OLAP part*. Typically, each layer contains a set of binary relations between geometries of a single kind (although the latter is not mandatory). For example, an instance of the relationship (*line*, *polyline*) stores the identifiers of the lines belonging to a polyline. There is always a finest level in the dimension schema, represented by a node with no incoming edges. We assume that this level, called "point", represents points in space. The level "point" belongs to the *algebraic part* of the model. Here, data in each layer are represented as infinite sets of points (x, y), finitely described by means of linear algebraic equalities and inequalities. In the *geometric part*, data consist of a finite number of elements of certain geometries. This part is used for solving the geometric part of a query, for instance to find all polygons that compose the shape of a country. Each point in the *algebraic part* corresponds to one or more elements in the *geometric part*. Note that, for example, *a point may correspond to two adjacent polygons*, or to the *intersection of two or more roads*. Moreover, a line may correspond to more than one polygon. There is also a distinguished level, denoted "All", with no outgoing edges.

Non-spatial information is represented in the *OLAP part*, and is associated to levels in the geometric part. For example, information about states, stored in a data warehouse, can be associated to polygons, or information about rivers, to polylines. Typically, these concepts are represented as a set of dimension levels, which are part of a hierarchy in the usual OLAP sense.

Example 1. In Fig. 1, the level *polygon* in layer L_e is associated with two dimension levels, *state* and *region*, such that *state* $\rightarrow$ *region* ("$A \rightarrow B$" means that there is a functional dependency from level A to level B in the OLAP part[4]). Each level may have attributes associated, like population or number of schools. Thus, a geometrically represented component is associated with a dimension level in the OLAP part. There is also an OLAP hierarchy associated to the layer L_r at the level of *polyline*. Notice that since dimension levels are associated to geometries, it is straightforward to associate facts stored in a data warehouse in the OLAP part, in order to aggregate these facts along geometric dimensions. Finally, in the algebraic part, the relationship $\langle point, polygon \rangle$ associates infinite point sets with polygons.

Formally, assume there is a set of layer names **L**, and a set **G** of geometry names, which contains at least the following geometries: point, node, line, polyline, polygon and the distinguished element "All". Each geometry G of **G** has an associated domain $dom(G)$. The domain of Point, dom(Point), for example, is the set of all pairs in $\mathbb{R}^2$. The domain of All $=$ {all}. The domain of the elements G of **G**, except Point and All, is a set of geometry identifiers, g_{id}, i.e., g_{id} are identifiers of geometry instances, like polylines or polygons.

Definition 1 (*GIS dimension schema*). A *GIS dimension schema* is tuple $\text{dGIS}_{sch} = \langle \mathscr{H}, \mathscr{F}_{att}, \mathscr{D} \rangle$ where $\mathscr{H}$ is a finite

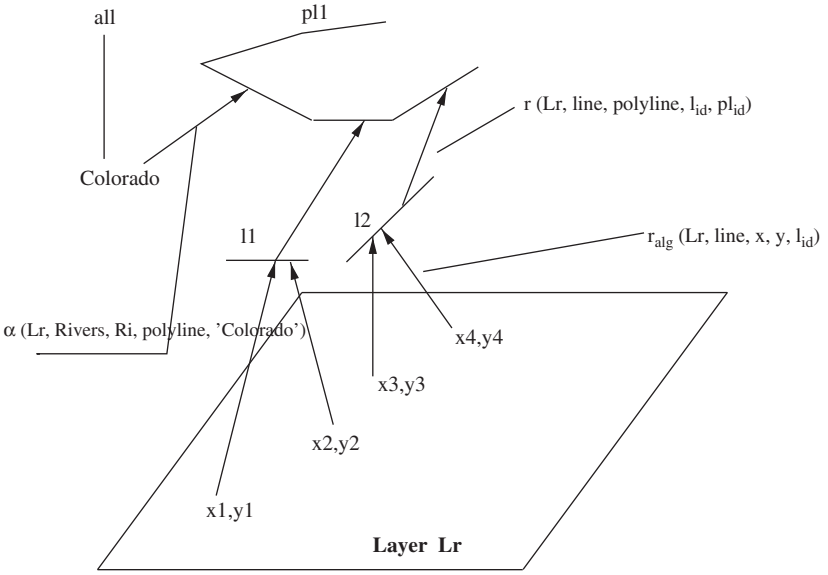

Fig. 2. A portion of a *GIS* dimension instance in Fig. 1.

set of graphs, $\mathscr{F}_{att}$ a set of functions, and $\mathscr{D}$ a set of OLAP dimension schemas. We define these sets below.

Given a layer $L \in \mathbf{L}$, H(L), is a graph where: (a) there is a node for each kind of geometry $G \in \mathbf{G}$ in L; (b) there is an edge between two nodes G_i and G_j if G_j is composed of geometries of type G_i (i.e., the granularity of G_j is coarser than that of G_i); (c) there is a distinguished member *All* that has no outgoing edges; (d) there is exactly one node representing the geometry $\langle point \rangle$ with no incoming edges. The dimension schemas $D \in \mathscr{D}$ are tuples of the form $\langle dname, Levels, \preceq \rangle$, such that *dname* is the name of the dimension, *Levels* is a set of dimension level names, and $\preceq$ is a partial order between levels (see [22]). Finally, $\mathscr{F}_{att}$ contains *partial* functions (denoted *Att*) mapping attributes in OLAP dimensions to geometries in the layers.

Example 2. The GIS dimension depicted in Fig. 1 has the schema: $\text{dGIS}_{sch} = \langle \{(\text{H}_1(L_r), (\text{H}_2(L_v), \text{H}_3(L_e)\}, \{Att(\text{state}), Att(\text{river}), \{Rivers, States\}\rangle$. In this schema, for example, $\text{H}_1(L_r) = (\{\text{point}, \text{line}, \text{polyline}, \text{All}\}, \{(\text{point}, \text{line}), (\text{line}, \text{polyline}), (\text{polyline}, \text{All})\})$. For the OLAP dimensions *Rivers* and *States*, the *Att* functions in $\mathscr{F}_{att}$ are: $Att(\text{state}, \text{States}) = (\text{polygon}, L_e)$ (meaning that the attribute *states* maps to polygons in layer L_e), and $Att(\text{river}, \text{Rivers}) = (\text{polyline}, L_r)$ (i.e., the attribute *river* maps to polylines in the layer L_r).

Definition 2 (*GIS dimension instance*). Let $\text{dGIS}_{sch} = \langle \mathscr{H}, \mathscr{F}_{att}, \mathscr{D} \rangle$ be a GIS dimension schema. A *GIS dimension instance* is a tuple $\langle \text{dGIS}_{sch}, r_{alg}, r, \alpha, \mathscr{D}_{inst} \rangle$, where (a) r_{alg} is a 5-ary relation representing the rollup between the algebraic and geometric parts. Thus, it is of the form $\langle L_i, G_j, x, y, g_{id} \rangle$, where L_i is a layer name, G_j is the geometry in the geometric part to which *point* rolls up, x, y are the coordinates of a point, and g_{id} is the identifier of the object associated to x, y in the geometric part; (b) r is a 5-ary relation representing the rollup between objects in the geometric part. It is of the form $\langle L_i, G_i, G_j, g_{id_i}, g_{id_j} \rangle$, where L_i is a layer name, G_i and G_j are geometries in the geometric part such that the former rolls up to the latter (i.e., there is an edge $G_i \to G_j$ in $\text{H}(L_i)$ in dGIS_{sch}), and g_{id_i} and g_{id_j} are instances of these geometries. We denote r_{alg} and r *rollup* relations. The function α maps a *member* in a level *level* in an OLAP dimension D, to an object g_{id} in a geometry G in a layer L. This mapping corresponds to the functions in $\mathscr{F}_{att}$. Intuitively, α provides a link between a data warehouse instance and an instance of the hierarchy graph. Finally, for each dimension schema $D \in \mathscr{D}$ there is a dimension instance in $\mathscr{D}_{inst}$, composed of a set of rollup functions [22] that relate elements in the different dimension levels (intuitively, these functions indicate how dimension level members are aggregated).

Example 3. Fig. 2 shows a portion of an instance of the GIS dimension schema of Fig. 1. A member ('Colorado') of the level *rivers* in the OLAP dimension rivers, is mapped (through the function α) to the polyline pl_1, in layer L_r. We show four points at the *point* level $\{(x_1, y_1), \ldots, (x_4, y_4)\}$ (recall that the points at this level are actually infinite and described by algebraic expressions). We also show the relations r_{alg} and r, containing the association of points to lines, and lines to polylines, respectively. For example, r_{alg} contains the tuple $\langle L_r, line, x_4, y_4, l_2 \rangle$.

Elements in the geometric part in Definition 1 can be associated with *facts*, each fact being quantified by one or more *measures*, in the usual OLAP sense.

Definition 3 (*GIS fact table*). Given a geometry G in a graph H(L) of a GIS dimension schema, and a list M of measures $(M_1, \ldots, M_k)$, a *GIS fact table schema* is a tuple $FT = (G, L, M)$. A *base GIS fact table schema* is a tuple $BFT = (point, L, M)$ that means, a fact table with the finest geometric granularity. A *GIS fact table instance* is a function

ft mapping values in $dom(G) \times \mathbf{L}$ to values in $dom(M_1) \times \cdots \times dom(M_k)$. A *base GIS fact table instance* maps values in $\mathbb{R}^2 \times L$ to values in $dom(M_1) \times \cdots \times dom(M_k)$.

Example 4. Consider a fact table containing *state* populations in our running example, stored at the polygon level. The fact table schema would be $(polyId, L_e, population)$, where *population* is the measure. If information about, for example, temperature data, is stored at the *point* level, we would have a base fact table with schema $(point, L_e, temperature)$, with instances like $(x_1, y_1, L_e, 25)$. Note that temporal information could be also stored in these fact tables, by simply adding the *time* dimension to the fact table. This would allow to store temperature information across time.

Basically, a GIS fact table is a standard OLAP fact table where one of the dimensions is composed of geometric objects in a layer. Classical fact tables in the OLAP part, defined in terms of the OLAP dimension schemas can also exist. For instance, instead of storing the population associated to a polygon identifier, as in Example 4, this information may reside in a data warehouse.

3.1. Geometric aggregation

We now give a precise definition of the kinds of aggregate queries we deal with.

Definition 4 (*Geometric aggregation*). Given a GIS dimension as introduced in Definitions 1 and 2, a *geometric aggregation* is the expression

$$\iint_{\mathbb{R}^2} \delta_C(x,y) h(x,y)\, dx\, dy,$$

where $C = \{(x,y) \in \mathbb{R}^2 \mid \varphi(x,y)\}$,[7] and δ_C is defined as follows:

$\delta_C(x,y) = 1$ on the two-dimensional parts of C is a Dirac delta function [8] on the *zero-dimensional* parts of C; and it is the product of a Dirac delta function with a combination of Heaviside step functions [21] for the *one-dimensional* parts of C (see Appendix A for details). Also, φ is a first-order (FO) formula in a multi-sorted logic $\mathscr{L}$ over the reals, geometric objects, and dimension level members. The vocabulary of $\mathscr{L}$ contains the relations r, r_{alg}, and the function α, together with the binary functions $+$ and $\times$ on real numbers, the binary predicate $<$ on real numbers and the real constants 0 and 1.[8] Also constants for layers, dimension names, dimension level names, and geometry names may appear in $\mathscr{L}$.[9] Atomic formulas in $\mathscr{L}$ are combined with the standard logical operators $\wedge$, $\vee$ and $\neg$ and existential and universal quantifiers over real variables, variables for geometric objects identifiers, and variables for dimension level members. Finally, h is an integrable function constructed from elements of $\{1, ft\}$ (*ft* stands for fact table), using arithmetic operations.

Note that this definition gives the basic construct for geometric aggregation queries. More involved queries can be written as combinations of this construct (e.g., "total number of airports per square kilometer" would require dividing the geometric aggregation that computes the number of airports in the query region, by the aggregation computing the area of such region).[10]

Example 5. The following queries refer to the example introduced in Section 1. The layers containing cities and rivers are labeled L_c and L_r, respectively, although in order to make the queries more interesting, we define cities as polygons instead of points. The population density for each coordinate in L_c is stored in a *base fact table* ft_{pop} (we assume it is stored in some finite way, i.e., using polynomial equations over the real numbers). In what follows, we abbreviate Point, Polygon and PolyLine by Pt, Pg and Pl, respectively. Also, Ci and Ri stand for the attributes city and river, respectively. Further, all constants are capitalized, to distinguish them from variables in our expressions. Finally, note that in the queries below, the Dirac delta function is such that $\delta_C(x,y) = 1$, inside the region C, and $\delta_C(x,y) = 0$, outside this region.

Q_1: *Total population of cities within 100 km from San Francisco*:

$$Q_1 \equiv \iint_{C_1} ft_{pop}(x,y,L_c)\, dx\, dy,$$

where C_1 is defined by the expression:

$$\begin{aligned} C_1 &= \{(x,y) \in \mathbb{R}^2 \mid (\exists x')(\exists y')(\exists x'')(\exists y'')(\exists pg_1)(\exists pg_2)(\exists c) \\ &\qquad (\alpha(L_c, \text{Cities}, \text{Ci}, \text{Pg}, \text{San Francisco}) \\ &= pg_1 \wedge r_{alg}(L_c, \text{Pg}, x', y', pg_1) \wedge \alpha(L_c, \text{Cities}, \text{Ci}, \text{Pg}, c) \\ &= pg_2 \wedge r_{alg}(L_c, \text{Pg}, x'', y'', pg_2) \wedge pg_2 \neq pg_1 \wedge ((x'' - x')^2 \\ &\quad + (y'' - y')^2 \leqslant 100^2) \wedge r_{alg}(L_c, \text{Pg}, x, y, pg_2))\}. \end{aligned}$$

Here, $\alpha(L_c, \text{Cities}, \text{Ci}, \text{Pg}, \text{San Francisco})$ maps the city of San Francisco (an instance of the level Ci in dimension Cities), to a polygon pg_1 in layer L_c. The third and fourth lines find the cities within 100 km of San Francisco, and the relation $r_{alg}(L_c, \text{Pg}, x, y, pg_2)$, with the mapping between the points and the polygons that satisfy the condition. We are interested in the points that belong to pg_2.

Q_2: *Total population of the cities crossed by the Colorado river*:

$$Q_2 \equiv \iint_{C_2} ft_{pop}(x,y,L_c)\, dx\, dy,$$

[7] The sets C in Definition 4 are known in mathematics as *semi-algebraic sets*. In the GIS practice, only linear sets (points, polylines and polygons) are used. Therefore, it could suffice to work with addition over the reals only, leaving out multiplication.

[8] The FO logic over the structure $(\mathbb{R}, +, \times, <, 0, 1)$ is well-known as the FO logic with polynomial constraints over the reals. This logic is well-studied as a data model and query language in the field of constraint databases [37].

[9] We may also quantify over layer variables, dimension level variables, etc., but we have chosen not to do this, for the sake of clarity.

[10] For the language $\mathscr{L}$, as usual in relational database theory, we assume set semantics, while in Section 8, set, bag, and mixed bag-set semantics [6] are supported through the SQL-like query languages.

$$C_2 = \{(x,y) \in \mathbb{R}^2 | (\exists x')(\exists y')(\exists pg_1)(\exists c) \\ (r_{alg}(L_c, \mathrm{Pl}, x', y', \alpha(L_r, \mathrm{Rivers}, \mathrm{Ri}, \mathrm{Pl}, \mathrm{Colorado})) \\ \wedge \alpha(L_c, \mathrm{Cities}, \mathrm{Ci}, \mathrm{Pg}, c) = pg_1 \wedge r_{alg}(L_c, \mathrm{Pg}, x', y', pg_1) \\ \wedge r_{alg}(L_c, \mathrm{Pg}, x, y, pg_1))\}.$$

4. Summable queries

The framework we presented in the previous section is general enough to allow expressing complex geometric aggregation queries (Definition 4) over a GIS in a formal way. However, computing these queries within this framework can be extremely costly, as the following discussion shows. Let us consider again Example 5. Here ft_{pop} is a density function. This could be a constant function over cities, e.g., the density in all points of San Francisco, say, 1000 people per square kilometer. But ft_{pop} is allowed to be more complex too, like for instance a piecewise constant density function or even a very precise function describing the true density at any point. Moreover, just computing the expression "C" of Definition 4 could be practically infeasible. In Example 5, query Q_2, computing on-the-fly the intersection (overlay) of the cities and rivers is likely to be very expensive. Therefore, we identify a subclass of geometric aggregate queries that simplifies the computation of the integral of the functions $h(x,y)$ of Definition 4. Specifically, we show that storing less precise information (for instance, having a simpler function ft_{pop} in Example 5) results in a more efficient computation of the integral. There are queries where even if the function ft_{pop} is piecewise constant over the cities, there is no other way of computing the population over the region defined by φ than taking the integral, as φ can define any semi-algebraic set. An example of a query of this kind is: "Total population endangered by a poisonous cloud described by φ, a formula in FO logic over $(\mathbb{R}, +, \times, <, 0, 1)$)". Further, just computing the population within an arbitrarily given region cannot be performed. However, for queries Q_1 and Q_2 the situation is different. Indeed, the sets C_1 and C_2 return a finite set of polygons, representing cities. If the function ft_{pop} is constant for each city, it suffices to compute ft_{pop} once for each polygon, and then multiply this value with the area of the polygon. Summing up the products would yield the correct result, without the need of integrating ft_{pop} over the area C_1 or C_2. This is exactly the subclass of queries we want to propose, those that can be rewritten as sums of functions of geometric objects returned by condition "C". We denote these queries *summable*.

Definition 5 (*Summable query*). A geometric aggregation query $Q = \iint_{\mathbb{R}^2} \delta_C(x,y) h(x,y)\, dx\, dy$ is *summable* if and only if:

(1) $C = \bigcup_{g \in G} ext(g)$, where G is a set of geometric objects, and $ext(g)$ means the *geometric extension* of g, that is, the subset of $\mathbb{R}^2$ that g occupies (e.g., a polygon or a polyline, as a subset of $\mathbb{R}^2$).

(2) There exists h', constructed using $\{1, f_t\}$ and arithmetic operators, such that

$$Q = \sum_{g \in S} h'(g) \quad \text{with } h'(g) = \iint_{\mathbb{R}^2} \delta_{ext(g)}(x,y) h(x,y)\, dx\, dy.$$

Working with less accurate functions for this type of queries means that the base GIS fact table instances of Definition 3 will be defined as mappings from values in $dom(G) \times \mathbf{L}$ to values in $dom(M_1) \times \cdots \times dom(M_k)$ (where the elements in $dom(G)$ are the geometric objects $g \in G$ such that $r_{alg}(L, G, x, y, g)$), instead of mappings from values in $\mathbb{R}^2 \times \mathbf{L}$ to values in $dom(M_1) \times \cdots \times dom(M_k)$.

Note that Definition 5 implies that the three summarizability conditions defined by Lenz et al. [28] must be preserved. The first two conditions (disjointness of categorical attributes, and completeness, respectively) are satisfied by our definition of a GIS dimension. The third condition requires function $h'(g)$ to be summarizable over g. The functions we use in this paper (typically sums, averages and maximum/minimum) satisfy the third condition in [28]. We remark that in this section (like in the previous one) we work with set semantics. Since the integration region is replaced by geometric identifiers, set semantics expresses correctly the most usual summable queries of interest.[11]

Example 6. Let us reconsider query Q_2 from Example 5. The function ft_{pop} (i.e., the fact table) now maps elements of $dom(\mathrm{Polygon})$ to populations. Note that C_2'returns a finite set of polygons, indicated by their ids (denoted g_{id}).

Q_2: *Total population of the cities crossed by the Colorado river*:

$$\mathrm{Q}_2 \equiv \sum_{g_{id} \in C_2'} ft_{pop}(g_{id}, L_c).$$

$$C_2' = \{g_{id} | (\exists x)(\exists y)(\exists c) \\ (r_{alg}(L_r, \mathrm{Pl}, x, y, \alpha(L_r, \mathrm{Rivers}, \mathrm{Ri}, \mathrm{Pl}, \mathrm{Colorado})) \\ \wedge \alpha(L_c, \mathrm{Cities}, \mathrm{Ci}, \mathrm{Pg}, c) = g_{id} \wedge r_{alg}(L_c, \mathrm{Pg}, x, y, g_{id}))\}.$$

Queries aggregating over zero or one-dimensional regions (like, for instance, queries requiring counting the number of occurrences of some phenomena) can also be summable. For example, counting the number of airports over a certain region, can be expressed as $Q \equiv \sum_{g_{id} \in C} 1$. Moreover, the aggregation can also be expressed over a fact table in the application part of the model, like in the query "Total number of students in cities crossed by the Colorado river". This query is expressed as $\sum_{\mathrm{Ci} \in C'} ft_{cities}^{\#students}(\mathrm{Ci})$, where the sum is performed over a set of city identifiers (the integration region C', which we

[11] This can be extended to support bag semantics [13,25], at the expense of increasing the presentation formal overload, and we chose to avoid this. Moreover, queries requiring bag semantics, like "Total number of rivers in California and Nevada" (where, if a river crosses both states must be counted twice), can be written using combinations of the basic construct (splitting the query and adding the results), as we commented in Section 3.1.

boolean DecideSummability(C)

Input: A query region "C".
Output: "True", if "C" is a finite set of elements of a geometry representing the query region for Q. "False" otherwise.

1: **for** each layer L and each geometry G in L **do**
2: $S = \phi$;
3: **for** each $g \in G$ **do**
4: **if** $ext(g) \subseteq C$ **then**
5: $S = S \cup \{g\}$;
6: **if** $C = \bigcup_{g \in S} ext(g)$ **then**
7: Return "True";
8: Return "False".

Fig. 3. Deciding summability.

omit here, since it is defined in the way we have already explained), and the function $ft^{\#students}_{cities}$ maps cities to the number of students in them (a fact table containing the city identifiers and, as a measure, the number of students). This fact table is outside the geometry of the GIS, showing that summable queries integrate GIS and OLAP in an elegant way.

Summable queries are useful in practice because, most of the time, we do not have information about parts of an object, like, for instance, the population of a part of a city. On the contrary, populations are often given by totals per city or province, etc. In this case, we may divide the city, for example, in a set of sub-polygons such that each sub-polygon represents a neighborhood. Thus, queries asking for information on such neighborhoods become summable.[12]

Algorithm in Fig. 3 decides if the integration region C is of the form $\bigcup_{g \in G} ext(g)$. If C is of this form, then the second condition of Definition 5 is automatically satisfied. The complexity of this algorithm is $O(\gamma \cdot (\sigma + \psi))$, where γ is the total number of all elements of all geometries in all layers, σ is the cost of subset testing between geometric objects involved in the computation, and ψ is the cost of the union in line 5. The latter two depend on the choice of data models and data structures used to encode geometric objects.

Once we have established that C is a finite union of elements g of some geometry G, it is easy to see how h' can be obtained from h. Indeed, for each $g \in G$, we can define $h'(g)$ as $\iint_{\mathbb{R}^2} \delta_{ext(g)}(x,y)h(x,y)\,dx\,dy$. Since h is built from the constant 1, fact table values and arithmetic operations, also h' can be seen to be constructible from 1, fact table values and arithmetic operations. The decision algorithm can be easily turned into an algorithm that produces, for a given "C", an equivalent description as a union of elements of some geometry. Once this description is found it is straightforward to find the function h'. This is illustrated by the aggregate queries Q_1 and Q_2 that are given in both forms in Sections 3 and 4, respectively.

5. Overlay precomputation

Many interesting queries in GIS boil down to computing intersections, unions, etc., of objects that are in different layers. Hereto, their overlay has to be computed. Query Q_2 in Section 4 is a typical example where cities crossed by rivers have to be returned. The on-the-fly computation of the set "C" containing all those cities, is costly because most of the time we need to go down to the algebraic part of the system, and compute the intersection between the geometries (e.g., states and rivers, cities and airports, and so on). Therefore, we study the possibilities and consequences of precomputing the overlay operation and show that this can be an efficient alternative for evaluating queries of this kind. R-trees [14], and aR-trees [33,34] can also be used to efficiently compute these intersections on-the-fly. In Section 8 we discuss this issue.

Haesevoets et al. [15,16] define affine-invariant triangulations of the plane based on the fact that any set of (both ways unbounded) lines, partitions the plane in a set of convex polygons and show that this partition is invariant under affine transformations of the plane. Definition 6 extends these notions to all geometric components in a thematic layer.

Definition 6 (*The carrier set of a layer*). The *carrier set* C_{pl} *of a polyline* $\mathrm{pl} = (p_0, p_1, \ldots, p_{(l-1)}, p_l)$ consists of all lines that share infinitely many points with the polyline, together with the two lines through p_0 and p_l, and perpendicular to the segments (p_0, p_1) and $(p_{(l-1)}, p_l)$, respectively. Analogously, the *carrier set* C_{pg} *of a polygon*

[12] Actually, we could get rid of the algebraic part of the queries if, instead of working with the topological relations r and r_{alg}, we (a) store in a relation r the geometric extension e of the objects in a layer (e.g., as a sequence of coordinates of the vertices of a polygon, or polyline, like in postGIS); (b) allow quantification over a second-order variable for these extensions; (c) add topological operations to the language, like `intersect`, or `contains`. Thus, a layer like L_r (assuming it only contains polylines) could be described in a relation r by tuples of the form $\langle L_r, Pl, e_{pl}, pl_{id}\rangle$, where e_{pl} is a complex data type containing the extension of the rivers (polylines) in L_r, and pl_{id} is the id of the polyline. Then, C'_2 would read

$$C'_2 = \{g_{id} \mid (\exists x)(\exists y)(\exists c)(\exists e_{pl})(\exists e_{pg}) (r(L_r, \mathrm{Pl}, e_{pl}, \alpha(L_r, \mathrm{Rivers}, \mathrm{Ri}, \mathrm{Pl}, \mathrm{Colorado})) \wedge \alpha(L_c, \mathrm{Cities}, \mathrm{Ci}, \mathrm{Pg}, c) = g_{id} \wedge r(L_c, \mathrm{Pg}, e_{pg}, g_{id}) \wedge intersects(e_{pl}, e_{pg}))\}.$$

In this expression, $\exists e_{pg}$ is a kind of second-order quantification given that it is a variable of complex type. This provides the intuition of how summable queries could be translated into practical languages supporting topological operations. At this stage of the paper we prefer the more general approach, i.e., to work with the algebraic part of the model. In Section 5 we describe another way of avoiding working with coordinates, using the notion of *subpolygonization*.

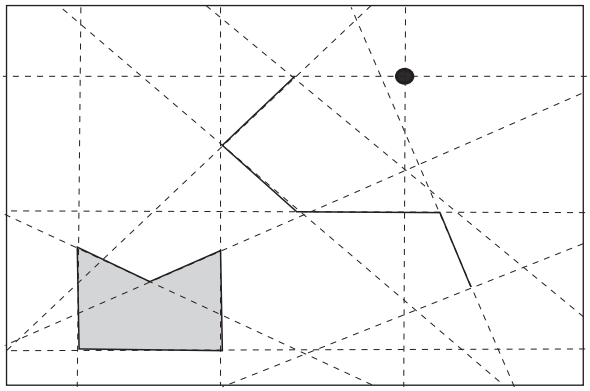

Fig. 4. The carrier sets of a point, a polyline and a polygon are the dotted lines.

pg is the set of all lines that share infinitely many points with the boundary of the polygon. Finally, the *carrier set* C_p *of a point* p consists of the horizontal and the vertical lines intersecting in the point. The *carrier set* C_L *of a layer* L is the union of the carrier sets of the points, polylines and polygons appearing in the layer. Fig. 4 illustrates the carrier sets of a point, a polyline and a polygon.

The carrier set of a layer induces a partition of the plane into open convex polygons, open line segments and points. In Section 5.2 we give more detail about this subdivision, and discuss complexity issues.

As some of the open convex polygons are unbounded, we consider a bounding box $B \times B$ in $\mathbb{R}^2$, containing all intersection points between lines (in practice, we consider the bounding box as an additional layer). Also, in what follows, a line segment is given as a pair of points, and a polyline as a tuple of points.

Definition 7. Let C_L be the carrier set of a layer L, and let $B \times B$ in $\mathbb{R}^2$ be a bounding box. The set of open convex polygons, open line segments and points, induced by C_L, that are strictly inside the bounding box, is called the *convex polygonization of* L, denoted $CP(L)$.

In our proposal, the thematic layers in a GIS application are overlayed by means of the *common subpolygonization* operation that further subdivides the bounding box $B \times B$ according to the carrier sets of the layers involved.

Definition 8 (*Subpolygonization*). Given two layers L_1 and L_2, and their carrier sets C_{L_1} and C_{L_2}, the *common subpolygonization* of L_1 according to L_2, denoted $\mathrm{CSP}(L_1, L_2)$ is a refinement of the convex polygonization of L_1, computed by partitioning each open convex polygon and each open line segment in it along the carriers of C_{L_2}.

Definition 8 can be generalized for more than two layers, denoted $\mathrm{CSP}(L_1, L_2, \ldots, L_k)$. It can be shown that the overlay-operation on planar subdivision induced by a set of carriers is commutative and associative. The proof is straightforward, and we omit it.

Example 7. Fig. 5 shows thecommon subpolygonization of a layer L_c containing one city (the pentagon with corner points a, b, c, d and e), and another layer, L_r, containing one river (the polyline pqr). Note that the subpolygonization generates many empty polygons containing no information whatsoever. We take advantage of this fact to optimize the process (see Section 7).

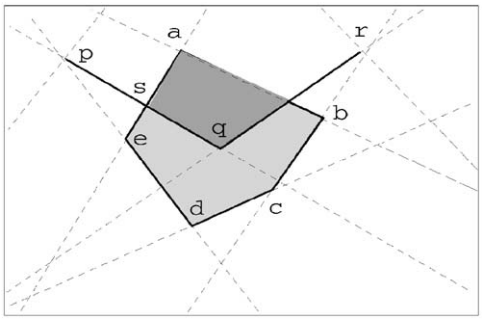

Fig. 5. The common subpolygonization of a layer.

The question that naturally arises is: why do we use the carriers of geometric objects in the computation of the overlay operation, instead of just the points and line segments that bound those objects? There are several reasons for this. First, consider the situation in the left frame of Fig. 6. A river *rqp* originates somewhere in a city, and then leaves it. The standard map overlay operation divides the river into two parts: one part, *rq*, inside the city, and the other one, *qp*, outside the city. Nevertheless, the city layer is not affected. On the one hand, we cannot leave the city unaffected, as our goal is in fact to precompute the overlay. On the other hand, partitioning the city into the line segment *rq* and the polygon *abcd* without the line segment *rq* results in an object which is not a polygon anymore. Such a shape is not only very unnatural, but, for example, computing its area may cause difficulties. With the common subpolygonization we guarantee convex polygons. Many *useful operations on polygons become very simple if the polygons are convex* (e.g., triangulation, computing the area, etc.). Also, many interesting affine invariants can be defined for convex polygons [12]. Another reason for the common subpolygonization is that it gives more precise information. The right frame of Fig. 6 shows the polygonization of the left frame. The partition of the city into more parts, depending on where the river originates, allows, for instance, asking for parts of the city with fertile and dry soil, depending on the presence of the river in those parts. As a more concrete example, consider the query:

Q_3: *Total length of the part of the Colorado river that flows through the state of Nevada. The following expression may solve the problem*:

$$Q_3 \equiv \sum_{g_{id} \in C'_4} ft_{length}(g_{id}, L_r),$$

where C'_3 is the set:

$$\begin{aligned} C'_3 = \{ & g_{id} | (\exists x)(\exists y)(\exists g_1) \\ & \alpha(L_r, \mathrm{Rivers}, \mathrm{Ri}, \mathrm{Pl}, \mathrm{Colorado}) = g_1 \wedge r_{alg} \\ & (L_r, \mathrm{Pl}, x, y, g_1) \wedge r_{alg}(L_e, \mathrm{Pg}, x, y, \alpha(L_e, \mathrm{States}, \mathrm{St}, \mathrm{Pg}, \mathrm{Nevada})) \\ & \wedge r_{alg}(L_r, \mathrm{Pl}, x, y, g_{id}) \wedge r(L_r, \mathrm{Pl}, \mathrm{Pg}, g_{id}, g_1)) \} \end{aligned}$$

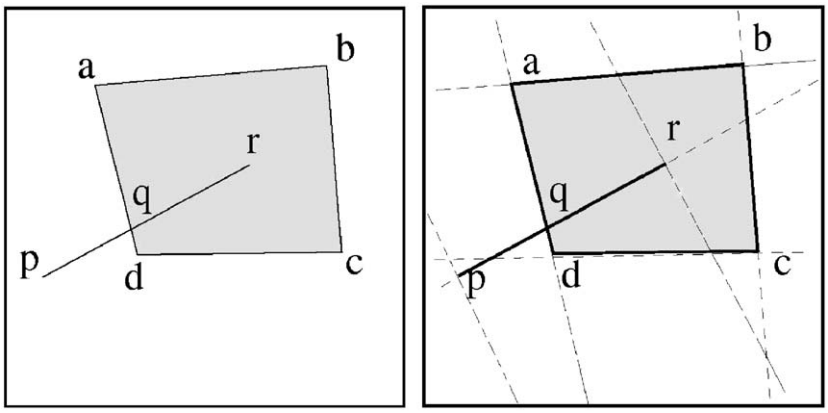

Fig. 6. Common subpolygonization vs. map overlay.

The expression Q_3 above gives the correct answer to the query when the river is such that the polyline representing it lies within the state boundaries. For instance, it would not work if the river is represented as polyline with a straight line passing through Nevada, because we would not know the length *inside* that state. When this is not the case, a common subpolygonization would solve the problem. Otherwise, the river must be partitioned in advance.

5.1. Using the common subpolygonization

From a *conceptual* point of view, we characterize the common subpolygonization of a set of layers as a schema transformation of the GIS dimensions involved. Basically, this operation reduces to updating the graphs of Definition 1. For this, we base ourselves on the notion of dimension updates introduced by Hurtado et al. [22], who also provide efficient algorithms for such updates. These updates allow, for instance, inserting a new level into a dimension and its corresponding rollup functions, or splitting/merging levels. The difference here is that in the original graph we have relations instead of rollup functions.

Consider the hierarchy graphs $H_1(L_1)$ and $H_2(L_2)$ depicted on the left-hand side of Fig. 7. After computing the common subpolygonization, the hierarchy graph is updated as follows: there is a unique hierarchy (remember that $CSP(L_1, L_2) = CSP(L_2, L_1)$) with bottom level Point, and three levels of the type Node (a geometry containing single points in $\mathbb{R}^2$), OPolyline (open polyline), and OPolygon (open polygon). We do not explain the update procedure here, we limit ourselves to show the final result. Note that now we have *all the geometries in a common layer* (in the example below we show the impact of this fact). The right-hand side of Fig. 7 shows the updated dimension graph. As a consequence of the subdivision in open polygons, open polylines and points, besides the relations r and r_{alg}, we also have functions, which we call *rollup functions*, of the form $f(G_j, G_k, g_{id})$, where G_k is a geometry (e.g., a polygon) that originated G_j

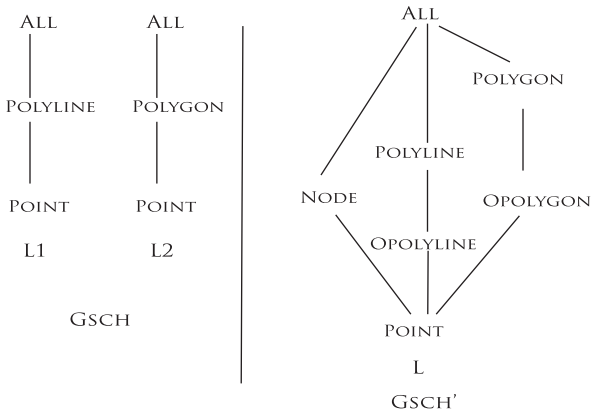

Fig. 7. Updated dimension schema.

(e.g., an open polygon), and g_{id} is an instance of G_j. Also, a point will belong only to an open polygon or open polyline. Note that, since the subpolygonization lies in a single layer, we do not need layer as an argument of the rollup functions.

We now investigate the effects of the common subpolygonization over the evaluation of summable queries. Specifically, we propose (a) to evaluate summable queries using the common subpolygonization; and (b) to precompute the common subpolygonization. Precomputation is a well-known technique in query evaluation, particularly in the OLAP setting. As in common practice, the user can choose to precompute all possible overlays, or only the combinations most likely to be required. The implementation we show in the next section supports both policies. So far, we have expressed the integration regions C in terms of theelements of the *algebraic* part of the GIS schema. However, the common subpolygonization, along with its precomputation, allows us to get rid of this part, and only refer to the ids of the geometries involved, also for computing the query region, that can be now expressed in terms of open polygons (OPg), open polylines (OPl) and points. Thus, query Q_3, "Total length of the part of the *Colorado* river that flows through the state

of *Nevada*", can be computed in a precise way as follows:

$$C_3' = \{g_{id} | (\alpha(\text{Rivers}, \text{Ri}, \text{Pl}, \text{Colorado}) = f(\text{OPl}, \text{Pl}, g_{id}) \wedge r(\text{OPl}, \text{Pg}, g_{id}, \alpha(\text{States}, \text{St}, \text{Pg}, \text{Nevada})))\}.$$

5.2. Complexity

Let G_{Sch} be the GIS dimension schema on the left-hand side of Fig. 7. Let G_{Inst} be an instance containing a set of polygons R, a set of points P and a set of polylines L. Moreover, let the maximum number of corner points of a polygon and the maximum number of line segments composing a polyline be denoted n_R and n_L, respectively. The carrier set of all layers, i.e., the union of the carrier sets for each layer separately, (see Definition 6) then contains at most $N = 2|P| + |L|(n_L + 2) + |R|n_R$ elements. These carriers represent a so-called *planar subdivision*, i.e., a partition of the plane into points, open line segments and open polygons. Planar subdivisions are studied in computational geometry [7]. It is a well-known fact that the complexity of a planar subdivision induced by N carriers is $O(N^2)$.

Property 1 (*Complexity of planar subdivision*). *Given a planar subdivision induced by N carriers*: (i) *the number of points is at most*[13] $N(N-1)/2$; (ii) *the number of open line segments is at most* N^2; (iii) *the number of open convex polygons is at most* $N^2/2 + N/2 + 1$. *The* complexity of the planar subdivision *is defined as the sum of the three expressions above*.

It follows that, if we precompute the overlay operation, in the worst case, the instance G'_{Inst} of the updated schema becomes quadratic in the size of the original instance G_{Inst}. However, as different layers typically store different types of information, the intersection will be only a small part of G'_{Inst}. Also, several elements of G'_{Inst} will not be of interest to any layer and can be discarded.

We now addressthe complexity of computing the planar subdivision, given a bounding box. Such a bounding box can be constructed in time quadratic in the number of carriers [7]. Typically, during its construction, a planar subdivision is stored in a *doubly connected edge list*. In a planar subdivision, each line segment or edge is adjacent to at most two open polygons. Accordingly, each edge is split into two directed half-edges, denoted *twins*, each one adjacent to a different polygon. We assume half-edges adjacent to the same polygon are given, and directed in counterclockwise order. As a half-edge is directed, we can say it has an origin and a destination. Finally, a doubly connected edge list consists of three collections of records, one for the points, one for the open polygons, and one for the half-edges. These records store the following geometrical and topological information: (i) The point record for a point p stores the coordinates of p, and a pointer to an arbitrary half-edge that has p as its origin. (ii) The record for an open polygon o stores a pointer to some half-edge adjacent to it. (iii) The half-edge record of a half-edge $\overrightarrow{e}$ stores a pointer to its origin, its twin, and the face it is adjacent to. It also stores pointers to the previous and next edges on the boundary of the incident open polygon, when traversing this boundary in counterclockwise order. Given a set of N carriers, a doubly connected edge list can be constructed in $O(N^2)$ time [7]. Furthermore, for each element of the planar subdivision, in order to compute the new rollup functions, we need to check which objects of G_{Inst} it belongs to. Checking one element in the subdivision against G_{Inst} can be done in logarithmic time by preprocessing G_{Inst}, as described in [9]. When a layer is added to the GIS, for example, the common subpolygonization needs to be updated. The following result holds [7].

Lemma 1. *Let S_1 and S_2 be planar subdivisions of complexity m_1 and m_2, respectively. The overlay of S_1 and S_2 can be constructed in time $O(m \log m + k \log m)$, where $m = m_1 + m_2$ and k is the complexity of the overlay.*

5.3. An application: topological geometric aggregation

In Section 4 we introduced *summable* queries to avoid integrals that may not be efficiently computable. At this point, the question that arises is: "What information do we store at the lowest level of the geometric part?" Should we store all the information about coordinates of nodes and corner points defining open convex polygons, or do we completely discard all coordinate information? Depending on the purpose of the system, there are several possibilities. A straightforward way of getting rid of the algebraic part of a GIS dimension schema is to store the coordinates of nodes, end points of line segments and corner points of convex polygons. A closer analysis reveals that this information might not be necessary for all applications. For example, queries about intersections of rivers and cities, or cities adjacent to rivers, do not require coordinate information, but rather *topological information*. To formalize the "*depending on the purpose of the system*" statement above, we need the notion of *genericity* of queries, first introduced by Chandra and Harel [5], and later applied to spatial databases [26,36]. In a nutshell, a query is *generic* if its result is independent of the internal representation of the data.

Topological or *isotopy-generic queries* are useful genericity classes for geometric aggregation queries. For instance, query Q_2 (Example 5) or "Number of states adjacent to *Nevada*", are topological geometric aggregation queries. If the purpose of the system is to answer topological queries, topological invariants [35,37] provide an efficient way of storing the information of one layer or the common subpolygonization of several layers. This invariant is based on a maximal topological cell decomposition, which is, in general, hard to compute. Thus, in order to compute the topological invariant we use the common subpolygonization, which happens to be a refinement of the decomposition.

Fig. 8 shows the topological information on the subpolygonization of two layers, one containing a city and the other containing a (straight) river. A topological invariant can be constructed from the subpolygonization

[13] Equality holds in case the lines are in general position, meaning that at each intersection point, only two lines intersect.

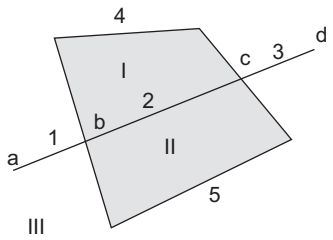

Fig. 8. Topological information. Nodes are indicated by a, b, …, open curves by 1, 2, …. Faces are labelled I, II, ….

as follows. Cells of dimension 0, 1 and 2 are called vertices, edges (these are different from line segments), and faces, respectively. The topological invariant is a finite structure consisting of the following relations: (1) unary relations *Vertex*, *Edge*, *Face* and *Exterior Face*. The latter is a distinguished face of dimension 2, i.e., the unbounded part of the complement of the figure; (2) a binary relation *Regions* providing, for each region name r in the GIS instance the set of cells making up r; (3) a ternary relation *Endpoints* providing endpoints for edges; (4) a binary relation *Face–Edge* providing, for each face (including the exterior cell) the edges on its boundary; (5) a binary relation *Face–Vertex* providing, for each face (including the exterior cell) the vertices adjacent to it; (6) a 5-ary relation *Between* providing the clockwise and counter-clockwise orientation of edges incident to each vertex. For example, the relation *Face–Edge* includes the tuples $(I,2),(I,4)$; *Face–Vertex* includes $(I,b),(I,c)$; and relation *Between*, the tuples $(\leftarrow,1,5,2)$ and $(\leftarrow,5,2,4)$, indicating the edges adjacent to vertex b in counter-clockwise direction. Thus, for answering summable queries, we can use the relations representing the topological invariant and the rollup functions, instead of the coordinates of the points in the algebraic part, provided that we add the relations described above, as extra information attached to the hierarchy instance. For instance, the query: "Number of states adjacent to *Nevada*" requires topological information about adjacency between polygons and lines given by the Face–Edge relation.

6. GIS-OLAP integration

The framework introduced in Section 3 allows a seamless integration between the GIS and OLAP worlds. In our proposal, denoted Piet (after Piet Mondrian, the painter whose name was adopted for the open source OLAP system we also use in the implementation), we achieve this integration by means of two mechanisms: (a) a metadata model, denoted *Piet-Schema*; and (b) a query language, denoted GISOLAP-QL, where a query is composed of two sections: a GIS section, denoted GIS query, with a specific syntax, and an OLAP section, OLAP-Query, with MDX syntax.[14] The idea underlying the choice of combining the two languages was to keep the GIS and OLAP standards as much as possible, even at the cost of losing semantic abstraction.

6.1. Piet-Schema

Piet-Schema is a set of metadata definitions that include: storage location of the geometric components and their associated measures, the sub-geometries corresponding to the subpolygonization of the layers in a map, and the relationships between the geometric components and the OLAP information. Piet uses this information to answer the queries written in the language we describe in Section 6.2. Metadata are stored in XML documents containing three kinds of elements: `Subpolygonization`, `Layer`, and `Measure`.[15]

The `Subpolygonization` element (identified with a tag of the same name) includes the location of the instances of each sub-geometry (sub-node, sub-polygon or sub-line) in the data repository (in our implementation, the PostGIS database where the map is stored). It also includes the name of the table containing the instance of each sub-geometry, the names of the key fields, and the identifiers allowing to associate elements of the original geometries with the elements of the corresponding sub-geometries.

The XML element `Layer` contains information of each of the layers that compose a map, and their relationship with the subgeometries and the data warehouse. This includes the name of the layer, the name of the table storing the actual data, the name of the key fields, the geometry and the description. There is also a list `Properties` which details the facts associated to geometric components of the layer. An element `SubpolygonizationLevel` indicates the subpolygonization levels that can be used (for instance, if it is a layer representing rivers, only *point* and *line* could be used). Finally, the relationship (if it exists) between the layer and the data warehouse is defined in the element `OLAPRelation`, that contains, for this layer, the name of the OLAP dimension table where the associated objects are stored, the hierarchy level for these objects, and the table associating the geometric objects and the OLAP dimension levels (in other words, the function α). Finally, an element `OLAPTable` contains the MDX statement that inserts a dimension in the original GISOLAP-QL expression. The last component of Piet-Schema definition is a list of `measure` elements where the measures associated to geometric components in the GIS dimension are specified, together with the aggregate functions that they support.

6.2. The GISOLAP-QL query language

GISOLAP-QL has a very simple syntax, allowing to express integrated GIS and OLAP queries. For the OLAP

[14] MDX is a query language initially proposed by Microsoft as part of the OLEDB for OLAP specification, later adopted as a de facto standard by most OLAP vendors. See http://msdn2.microsoft.com/en-us/library/ms145506.aspx.

[15] In what follows we denote sub-geometries the geometries that are originated by the subpolygonization process (e.g., a polyline originates sub-lines, or a polygon originates sub-polygons). Also, we denote indistinctly geometric elements or geometric objects the instances of a geometry.

part of the query we keep the syntax and semantics of MDX. A GISOLAP-QL query is of the form:

GIS-Query | *OLAP-Query*

A pipe ("|") separates two query sections: a GIS query and an OLAP query. The OLAP section of the query applies to the OLAP part of the data model (namely, the data warehouse) and is written in MDX. The GIS part of the query has the typical SELECT FROM WHERE SQL form, except for a separator (";") at the end of each clause:

SELECT *list of layers and/or measures*;
FROM *Piet-Schema*;
WHERE *geometric operations*;

The SELECT clause is composed of a list of layers and/or measures, which must be defined in the Piet-Schema of the FROM clause. The query returns the geometric components (or their associated measures) that belong to the layers in the SELECT clause, and verify the conditions in the WHERE clause. The FROM clause just contains the name of the schema used in the query. The WHERE clause in the GIS-query part consists in conjunctions and/or disjunctions of geometric operations applied over all the elements of the layers involved. The expression also includes the kind of sub-geometry used to perform the operation (this is only used if the subpolygonization technique is selected to solve the query). Although any typical geometric operation can be supported, our current implementation supports the "intersection" and "contains" operations. Accepted values for *subgeometry* are "Point", "LineString" and "Polygon".[16] For example, the expression contains (layer.usa_states, layer.usa_rivers, subplevel.Linestring) computes the states which contain at least one river, using the geometric objects of type *linestring* generated and associated during the overlay precomputation. Finally, the WHERE clause can also mention a query region (the region where the query must be evaluated).

Example 8. The query "description of rivers, cities and store branches, for branches in cites crossed by a river" reads in GISOLAP-QL:

```
SELECT layer.usa_rivers, layer.usa_cities, layer.usa_
stores;
FROM Piet-Schema;
WHERE intersection(layer.usa_rivers, layer.usa_
cities,subplevel.Linestring)
and contains(layer.usa_cities, layer.usa_stores,
subplevel.Point);
```

This query returns the components r, s, and c in the layers usa_rivers, usa_stores and usa_cities, respectively, such that r and c intersect, and s is contained in c (i.e., the coordinates of the point that represents s in layer usa_stores are included in the region determined by the polygon that represents c in layer usa_cities). In other words, if L is a list of attributes (geometric components) in the SELECT clause, $I = \{(r_1, c_1), (r_2, c_2), (r_3, c_3)\}$ is the result of the intersection operation, and $C = \{(c_1, s_1), (c_2, s_2)\}$ is the result of the contains operation, the semantics of the query above is given, operationally, by the expression $\mathbf{\Pi}_L(I \bowtie C)$.

The query "number of branches by city" uses a geometric measure defined in Piet-Schema. The query reads:

```
SELECT layer.usa_cities,measure.StoresQuantity;
FROM Piet-Schema;
WHERE intersection(layer.usa_cities,layer.usa_stores,
subplevel.Point);
```

6.3. Spatial OLAP with GISOLAP-QL

Users of GIS-based DSSs usually need to perform OLAP operations involving a data warehouse associated to geographic objects in maps. Thus, they would write "full" GISOLAP-QL queries, i.e., queries composed of the GIS and OLAP parts (obviously, a real-world system should provide her with report-writing tools, based on the query language). A GISOLAP query is simply an MDX query that receives as input the result returned by the GIS portion of the query. Consider the dimensions Store, Promotion Media, and Product, and assume that the following hierarchy (specified in Piet-Schema) defines the *Store* dimension: store → city → state → country → All. The query "total number of units sold and their cost, by product, promotion media (e.g., radio, TV) and store" reads:

```
SELECT layer.usa_states; FROM Piet-Schema;
WHERE intersection(layer.usa_states,layer.usa_stores,
subplevel.point);

                |
select [Measures].[Unit Sales], [Measures].[Store
Cost],[Measures].[Store Sales]
ON columns
{([Promotion Media].[All Media],[Product].[All
Products])} ON rows
from [Sales]
```

The GIS-query returns the states which intersect store branches at the point level. The OLAP section of the query uses the measures in the data warehouse, in order to return the requested information. In this example, first, the GIS section of the query returns three identifiers, 1, 2, and 3, corresponding, respectively, to the states of California, Oregon and Washington. These identifiers correspond to three ids in the OLAP part of the model, stored in a Piet mapping table. Then, an MDX subexpression is built for each state, traversing the different

[16] For instance, when computing store branches close to rivers, we would use *linestring* and *point*.

			Measures		
Store	Promotion Media	Product	Unit Sales	Store Cost	Store Sales
Los Angeles	All Media	All Products			
Salem	All Media	All Products			
Seattle	All Media	All Products	46,996	40,037.98	100,295.52

Fig. 9. Query result for the full GISOLAP-QL example query.

dimension levels (starting from *All* down to *state*). The information is obtained from the XML element `OLAPTable` in the Piet-Schema. Finally, the MDX clause `Children`[17] is added, allowing to obtain the children of each state in the hierarchy (in this case, the cities). The sub-expressions for the three states in this query are linked to each other using the MDX clauses `Union` and `Hierarchize`.[18] The final MDX is:

```
Hierarchize(Union(Union({[Store].[All
Stores].[USA].[CA].Children},
{[Store].[All Stores].[USA].[OR].Children)},
{[Store].[All Stores].[USA].[WA].Children})))
```

The MDX subexpression is finally added to the OLAP-query section of the GISOLAP-QL statement. The resulting expression is:

```
select {[Measures].[Unit Sales], [Measures].[Store
Cost],[Measures].[Store Sales]}
ON columns
Crossjoin(Hierarchize(Union(Union({[Store].[All
Stores].[USA].[CA].Children},
{[Store].[All Stores].[USA].[OR].Children}),
{[Store].[All Stores].[USA].[WA].Children})),
{([Promotion Media].[All Media], [Product].[All
Products])})
ON rows
from [Sales]
```

Our Piet implementation allows the resulting MDX statement to be executed over a Mondrian engine (see Section 7 for details) in a single framework. Fig. 9 shows the result for our example. We can see three dimensions: Store (obtained through the geometric query), Promotion Media, and Product. The results are shown starting from the "city" level. A user can then navigate this result (drilling-down or rolling-up along the dimensions) to reach the desired granularity. Fig. 10 shows an example, drilling down starting from Seattle.

7. Implementation

In this section we describe our implementation. We first present the software architecture and components, and then discuss the algorithmic solutions for two key aspects of the problem: accuracy and scalability. The general system architecture is depicted in Fig. 11. A *Piet Administrator* defines the data warehouse schema, loads the GIS (maps) and OLAP (facts and hierarchies) information into a data repository, and creates a relation between both worlds (maps and facts), also defining the information to be included in each layer. The repository is implemented over a PostgreSQL database.[19] GIS data are stored and managed using PostGIS,[20] which adds support for geographic objects to the PostgreSQL database, and implements all of the Open Geospatial Consortium specification except some 'hard' spatial operations (the system was developed with the requirement of being OpenGIS-compliant[21]).

A graphic interface is used for loading GIS and OLAP information into the system and defining the relations between both kinds of data. The GIS part of this component is based on JUMP,[22] an open source software for drawing maps and exporting them to standard formats. Facts and dimension information are loaded using a customized interface. For managing OLAP data, we extended Mondrian,[23] in order to allow processing queries involving geometric components. The OLAP navigation tool was developed using Jpivot.[24]

A *Data Manager* processes data in basically two ways: (a) performs GIS and OLAP data association; (b) precomputes the overlay of a set of geographic layers, adapts the affected GIS dimensions, and stores the information in the database. The *query processor* delivers a query to the module solving one of the four kinds of queries supported by our implementation (shown at the bottom of Fig. 11). Of course, new kinds of queries can be easily added. For example, the topological queries explained in Section 5.3 are not supported by our current implementation. However, geo-ontologies [32] can be used to implement the topological relations for query processing. These kinds of queries, introduced in Section 1, are: (a) standard GIS queries; (b) standard OLAP queries; (c) geometric aggregation queries; (d) integrated GIS-OLAP queries. We give examples of each one of them in Section 8.

7.1. Piet components and functionality

The Piet implementation consists of two main modules: (a) Piet-JUMP, which includes (among other utilities) a graphic interface for drawing and displaying maps, and a back-end allowing overlay precomputation via the common subpolygonization and geometric queries; (b) Piet-Web, which allows executing GISOLAP-QL and pure OLAP queries. The result of these queries can be navigated in standard OLAP fashion.

Piet-JUMP Module. This module consists of a series of "plug-ins" added to the JUMP platform: the *Precalculate*

[17] `Children` returns a set containing the children of a member in a dimension level.

[18] `Union` returns the union of two sets, `Hierarchize` sorts the elements in a set according to an OLAP hierarchy.

[19] http://www.postgresql.org/.

[20] http://postgis.refractions.net.

[21] OpenGIS is a OGC specification aimed at allowing GIS users to freely exchange heterogeneous geodata and geoprocessing resources in a networked environment.

[22] http://www.jump-project.org/.

[23] http://mondrian.sourceforge.net.

[24] http://jpivot.sourceforge.net.

			Measures		
Store	Promotion Media	Product	Unit Sales	Store Cost	Store Sales
+Los Angeles	+All Media	+All Products			
+Salem	+All Media	+All Products			
−Seattle	+All Media	+All Products	46,996	40,037.98	100,295.52
HQ	+All Media	+All Products			
Store 6	−All Media	+All Products	21,333	18,266.44	45,750.24
	Bulk Mail	−All Products	1,512	1,323.63	3,295.13
		+Drink	139	126.06	297.48
		+Food	1,105	963.90	2,410.03
		+Non-Consumable	268	233.67	587.62
	Cash Register Handout	+All Products	514	452.56	1,132.35
	Daily Paper	+All Products	1,279	1,098.90	2,735.05
	Daily Paper, Radio	+All Products	511	429.27	1,103.81
	Daily Paper, Radio, TV	+All Products	899	760.03	1,906.56
	In-Store Coupon	+All Products			
	No Media	+All Products	14,051	12,035.61	30,154.93
	Product Attachment	+All Products	1,013	866.55	2,184.40
	Radio	+All Products			
	Street Handout	+All Products	258	211.20	528.76
	Sunday Paper	+All Products	853	721.97	1,785.55
	Sunday Paper, Radio	+All Products			
	Sunday Paper, Radio, TV	+All Products			
	TV	+All Products	443	366.73	923.70
Store 7	+All Media	+All Products	25,663	21,771.54	54,545.28

Fig. 10. Drilling down starting from the result of Fig. 9.

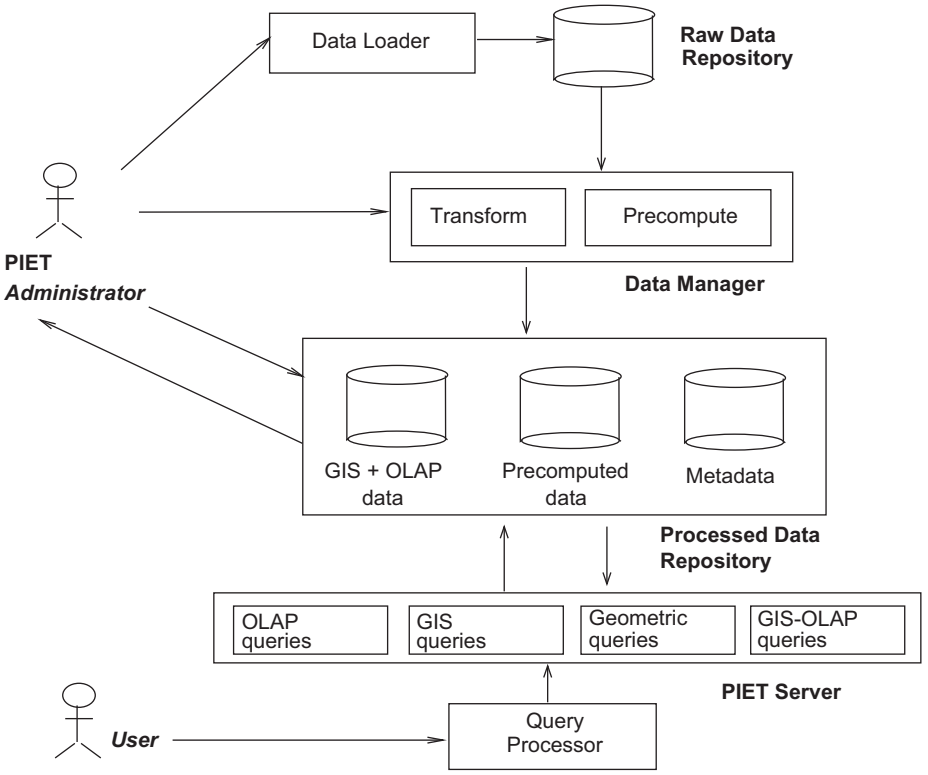

Fig. 11. The Piet Architecture.

Overlay, *Function Execution*, *GIS-OLAP association*, and *OLAP query* plug-ins. In what follows we briefly describe the modules in order to give the reader a clear idea of the functionality of the system, and how the techniques described in Section 5 are implemented.

The *Precalculate Overlay* plug-in computes the overlay of a set of selected thematic layers. The information generated is used by the other plug-ins. Besides the set of layers to overlay, the user must create a layer containing only the "bounding box". For all possible combinations of the selected layers, the plugin performs the following tasks:

(a) Generates the carrier sets corresponding to the geometric objects in the layers. This process creates,

for each possible combination, a table containing the generated carrier lines.

(b) Computes the common subpolygonization of the *layer combination*. In this step, the new geometry levels in the schema are obtained, namely: nodes, open polylines, and open polygons. This information is stored in a *different table for each geometry*, and each element is assigned a unique identifier.

(c) Associates the original geometric objects to the newly generated ones. This is the most computationally expensive process. The JTS Topology Suite[25] was extended and improved (see below) for this task. The information obtained is stored in the database (in one table for each level, for each layer combination) in the form ⟨*id of an element of the subpolygonization, id of the original geometric element*⟩ pairs.

(d) Propagates the values of the density functions to the objects in the subpolygonization. This is performed in parallel with the association process explained above.

Finally, for each combination of layers, we find all the elements in the subpolygonization that are common to more than one original geometric object. In the database, a table is generated for each layer combination, and for each geometry in the subpolygonization (i.e., node, open line, open polygon). A tuple in each table contains the unique identifier of a geometric object, and the unique identifier of the object in the subpolygonization.

The *Function Execution* plug-in computes a density function defined in a thematic layer, *within a query region* (or the entire bounding box if the query region is not defined). The result is a new layer with the geometric objects in the subpolygonization and their corresponding function values. Along with the selected query region, a density function is also defined. Two kinds of subpolygonizations can be used: *full* subpolygonization (when *all* the layers are overlayed) or *partial* subpolygonization (when the overlay involves only a subset of the layers). In the second case the process is faster, but precision may be unacceptable, depending on how well the polygons fit the query region.

The *GIS-OLAP association* plug-in associates spatial information to information in a data warehouse. This information is used by the "OLAP query" plugin and the Piet-Web module. A table contains the unique identifier of the geometric object, the unique identifier of the corresponding object in the data warehouse, and, optionally, a description of such object.

The *OLAP query* plug-in joins the two modules that compose the implementation. For a given spatial query and an OLAP model, the plugin generates and executes an MDX query that merges both kinds of data, and is then passed on to an OLAP tool.

Piet-Web Module. This module handles GISOLAP-QL queries, spatial aggregation queries, and even pure OLAP queries. In all cases, the result is a dataset that can be navigated using any OLAP tool. This module includes: (a) the GISOLAP-QL parser; (b) a translator to SQL; (c) a module for merging spatial and MDX queries through query re-writing, as explained in Section 6.

[25] JTS is an API providing fundamental geometric functions, supporting the OCG model. See http://www.vividsolutions.com/jcs/.

7.2. Robustness and scalability issues

As with all numerical computation using finite-precision numbers, geometric algorithms included in Piet may present problems of robustness, i.e., incorrect results due to round-off errors. Many basic operations in the JTS library used in the Piet implementation have not yet been optimized and tuned. Thus, we extended and improved this library, resulting in the so-called Piet-Utils library. Additionally, the subpolygonization of the overlayed layers generates a huge number of geometric elements. In this setting, scalability issues must be addressed, to guarantee performance in practical real-world situations. Thus, we propose a partition of the map using a grid, which optimizes the computation of the subpolygonization while preserving its geometric properties.

7.2.1. Robustness

We address separately the computation of the carrier lines and the subpolygonization process.

Computation of carrier lines. In a Piet environment, geometric elements are internally represented using the vector model. The JTS library is composed of objects of type *geometry*. Some examples of instances of these objects are: POINT (378 145), LINESTRING (191 300, 280 319, 350 272, 367 300), and POLYGON (83 215, 298 213, 204 74, 120 113, 83 215). Each geometric component includes the name and a list of vertices, as pairs of (X, Y) coordinates.

The first step of the subpolygonization process is the generation of a list containing the carrier lines produced by the carrier sets of the geometric components of each layer. The original JTS functions may produce duplicated carrier lines, arising from the incorrect overlay of (apparently) similar geometric objects. For instance, if a river in one layer coincides with a state boundary in another layer, duplicated carrier lines may appear due to mathematical errors, and propagate to the polygonization step. The algorithm used in Piet eliminates these duplicated carrier lines after the carrier set is generated.

We also address the problem of minimizing the mathematical errors that may appear in the computation of the intersection between carrier lines in different layers. First, given a set of carrier lines $L_1, L_2, \ldots, L_n$, the intersection between them is computed one line at a time, picking a line L_i, $i = 1, n-1$, and computing its intersection with L_{i+j}, $j \geqslant 1$. Thus, the intersection between two lines L_k, L_s is always computed only once. However, it is still possible that three or more lines intersect in points very close to each other. In this case, we use a boolean function called `isSimilarPoint`, which, given two points and an error bound (set by the user), decides if the points are or are not the same (if the points are different they will generate new polygons). There is also a function `addCutPoint` which receives a point p and a list P of points associated to a carrier line L. This function is

used while computing the intersection of L with the rest of the carrier lines. If there is a point in P, "similar" to p, then p is not added to P (i.e., no new cut point is generated). The points are stored *sorted* according to their distance to the origin, in order to speed-up the similarity search.

As an example, consider three carrier lines: L_1, L_2 and L_3. P_1 is the point where L_1 intersects L_2 and L_3. Also assume that the algorithm that generates the sub-nodes is currently usingL_2 as pivot line (i.e., L_1 was already used, and L_3 is still waiting). The algorithm computes the intersection between L_2 and L_3, which happens to be a point P_3 very close to P_1. If the difference is less than a given threshold, P_3 will not be added to the list of cutpoints for L_2. The same will happen for L_3.

Subpolygonization: The points where the carrier sets intersect each other generate objects denoted *sub-lines*. From these sub-lines, *sub-polygons* are computed. This information is stored in the postGIS database. The sub-polygons are produced using a JTS class called *Polygonizer*. Our improvements to the JTS library ensure that no duplicated sub-lines will be used to generate the sub-polygons. Further, the sub-lines that the Polygonizer receives do not include the lines generated by the bounding box.

The most costly process is the association of the sub-geometries to the original geometries. For this computation we also devised some techniques to improve the functions provided by the JTS library. For instance, due to mathematical errors, two adjacent sub-polygons may appear as overlapping geometries. As a consequence, the JTS intersection function provided by JTS would, erroneously, return *True*. We replaced this function with a new one, a boolean function denoted `OverlapPg` (again, "`error`" is defined by the user), which receives two objects and a given error threshold, and returns *True* if their overlapping area is larger than such threshold.

7.2.2. Scalability

The subpolygonization process is a huge CPU and memory consumer. Even though Property 1 shows that the planar subdivision is quadratic in the worst case, for large maps, the number of sub-geometry elements produced may be unacceptable for some hardware architectures. This becomes worse for a high number of layers involved in the subpolygonization. In order to address this issue, we do not compute the common subpolygonization over an entire map. Instead, we further divide the map into a grid, and compute separately the subpolygonization within each rectangle in the grid. This scheme produces sub-polygons only where they are needed. For instance, in our running example, we have a layer with volcanoes in the northern hemisphere. The density of the volcanoes is higher in the western region, and decreases toward the east. Therefore, a huge portion of the map, without any object of interest, would be affected by the carrier lines generated by the subpolygonization. Besides, a carrier line generated by a volcano in the west would impact a region in the east. It would be more natural that the influence of a carrier line remains within the "neighborhood" of the geometric object that generates it. The grid subdivision solves these problems. Reducing the number of geometric objects generated by the subpolygonization, of course, also reduces the size of the final database. Also note that the squares in the grid could be of different sizes. In addition, it would be possible to compute the polygonization of the squares in the grid in parallel, provided the necessary hardware is available. As a remark, note that the grid partition also allows the refinement of a particular rectangle, if, for instance, this rectangle is overloaded with geometric objects. Finally, note that geographies can be updated. For instance, a city may grow, a country may be split or even integrated with another one. Since these changes, in general, only affect a portion of a map, only the common subpolygonization for the affected rectangles must be recomputed, and the rest of the map would remain unchanged.

Implementing and using the grid. The grid partition is implemented as a relational table, denoted gis_quadrant, containing the geometry and identifier of each rectangle (i.e., the table contains as many tuples as rectangles are in the grid). In addition, a table is generated for each layer, containing only the non-empty grid partitions. In our running example, we have a table volcanoes_grid, where only the squares containing a volcano are stored (about 10% of the total grid). This dramatically reduces the subpolygonization time (which uses these tables, instead of the original ones), as we report in Section 8. The grid partitioning improves, in many cases, query processing. Since the map overlays are precomputed, query evaluation reduces to compute aggregations *over the tables containing these overlays.* When attribute information is needed, a join with the tables storing the result of the subpolygonization process is also needed. Due to grid partitioning, these tables *only contain the subpolygons of interest,* speeding up query execution time. When a query must be executed constrained to a query region defined at running time (Fig. 12), the intersection between the grid and the query region must becomputed on-the-fly, which impacts in the query execution times, as we show in Section 8. Nevertheless, the grid partitioning allows that to just compute the intersection for the affected rectangles, obtaining an important improvement in the performance of these queries.

The size of the grid is defined in a trial-and-error iterative process. This process starts with a size that is chosen according to the kinds of maps at hand, until we obtain a grid that results in an acceptable subpolygonization time (of course, the available hardware plays an important role here). As an example, in our experiments of Section 8, we used a grid has one thousand squares (20×50), having started from a grid of four squares (2×2).

8. Experimental evaluation

We discuss the results of a set of tests, aimed at providing evidence that the overlay precomputation method, for a wide class of geometric queries (with or without aggregation), outperforms other well-established methods like R-tree [14] and aggregation R-tree (aR-tree)

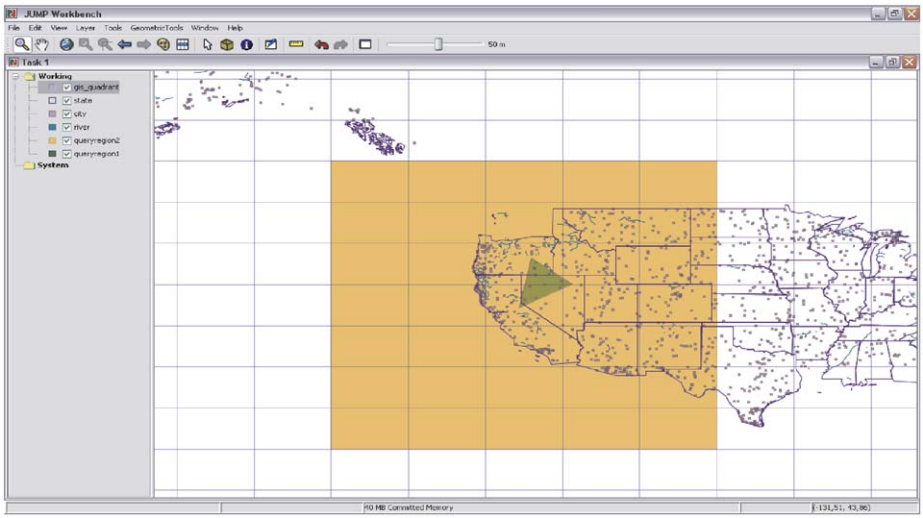

Fig. 12. Query regions for geometric aggregation.

[33] indexing. The main goal of our experiments is to study under which conditions one strategy behaves better than the other ones. This can be a first step toward a query optimizer that can choose the better strategy for any given GIS query. *We are aimed at showing that our approach is a valid and promising one, and not to compare Piet in the current implementation stage with a state-of-the-art commercial GIS.*

We ran our tests on a dedicated IBM 3400× server equipped with a dual-core Intel-Xeon processor, at a clock speed of 1.66 GHz. The total free RAM memory was 4.0 GB, and we used a 250 GB disk drive. The tests were run over the real-world maps. There are four layers, containing rivers, cities and states in United States and Alaska, and volcanoes in the northern hemisphere, containing, respectively, the following number of objects: 353 (of type multilinestring), 1767 (points), 151 (polygons), and 719 (points), (in the original map, many of the states where divided in subpolygons, resulting in the 151 objects reported). Non-spatial information is stored in a data warehouse, with dimensions customer, stores and product, and a fact table contains store sales across time (we introduced this warehouse in Section 1). The size of the warehouse in 60 MB. Also *numerical and textual information on the geographic components exist* (e.g., population, area), stored as usual in the GIS. We defined a grid for computing the subpolygonization, dividing the bounding box in rectangles, as shown in Fig. 12. The size of the grid is 20×50 squares (i.e., 1000 squares in total). We remark that we also tested Piet using other kinds of maps, and in all cases the results we obtained were similar to the ones reported here.[26]

We performed five kinds of experiments: (a) subpolygonization; (b) geometric queries without aggregation (GIS queries); (c) geometric aggregation queries; (d) geometric aggregation queries including a query region; (e) full GISOLAP-QL queries.

Table 1
Average subpolygonization times.

Number of layers	Average execution time
4	4 h 54 min 55.83 s
3	3 h 4 min 1.03 s
2	1 h 20 min 45.08 s

Table 2
Total subpolygonization times.

Number of layers	Total execution time
4	4 h 54 min 55 s
3	12 h 16 min 4 s
2	8 h 4 min 30 s

Tables 1 and 2 show the execution times for the subpolygonization process for the 1000 squares, from the generation of carrier lines to the generation of the precomputed overlayed layers. The grid strategy achieves an important performance improvement in computing the common subpolygonization, against the alternative of using the full map (i.e., without grid partitioning), close to an order of magnitude. Table 1 shows the average execution times for a combination of 2, 3 and 4 layers. For example, the third line means that a combination of two layers takes an average of 1 h and 20 min to compute. Table 2 is interpreted as follows. The second line in this table tells that computing *all* possible three-layer combinations, takes about 12 h. Analogously, the third line informs that computing *all possible* two-layer combinations takes more than 8 h. Note that the first line of Tables 1 and 2 is the same: they report the total time for computing the overlay of the four layers (i.e., there is only one layer). In spite of the time required by the process,

[26] See http://piet.exp.dc.uba.ar/piet.

note that precomputing the subpolygonization is performed off-line, in the same way view materialization is computed usually in OLAP. Also, since we use the grid partitioning strategy, updates are likely to be performed within a reasonable time window (see discussion in Section 7). Table 3 reports the maximum, minimum, and average number of elements in the sub-geometries in the grid rectangles, for the combination of the four layers. We also compared the sizes of the database before and after computing the subpolygonization: the initial size of the database is 166 MB. After the precomputation of the overlay of the four layers, the database occupies 621 MB. Although this seems to be an important increase in databases size, all possible layer overlay combinations were materialized, with no compression of any kind (i.e., the worst case). If needed, a partial materialization strategy (along the lines of [19]) can be applied. Note that even though the size of the database is moderate, we believe that the maximum number of sub-geometries per rectangle is high enough for making the experiments significant, also considering that we work with real-world maps.

Table 3
Number of sub-geometries in the grid for the 4-layers overlay.

Sub-geometry	Max.	Min.	Avg.
# of carrier lines per rectangle	616	4	15
# of points per rectangle (carrier lines intersection in a rectangle)	107 880	4	452
# of segment lines per rectangle (segments of carrier lines in a rectangle)	212 256	4	868
# of polygons per rectangle	104 210	1	396

For tests of type (b), we selected four geometric queries that compute the intersection between different combinations of layers, without aggregation. The queries were evaluated over the entire map (i.e., no query region was specified). Table 4 shows the queries and their expressions in the three query languages. Note that we have chosen to show, in Tables 4 and 5, the SQL translations of the Piet queries. In this way, the tables containing the overlay precomputation (with prefix gis_pre_), and the subpolygonization tables (with prefix gis_subp_) are displayed. We first ran the queries generated by Piet against the PostgreSQL database. We then ran equivalent queries with PostGIS, which uses an R-tree implemented using GiST—Generalized index search tree—[20]. Finally, we ran the postGIS queries without indexing for the postGIS queries.

Remark 2. Although we cannot claim that PostGIS is the best implementation of R-trees, we believe that it is appropriate to compare it against our implementation. It is a well-established and recognized product, and professional developers have participated in its implementation. On the contrary, Piet is still at a proof-of-concept stage. Thus, we believe this is a fair comparison.

The following are the conditions under which the tests were run: (a) all the layers were indexed (see Fig. 17 for index sizes); (b) all PostGIS queries have been optimized analyzing the generated query plans in order to obtain the best possible performance; (c) all Piet tables have been indexed over attributes that participate in a join or in a

Table 4
Geometric queries.

Query	Method	Code
Q1: List the states that contain at least one volcano	PostGIS without spatial indexing	SELECT DISTINCT state.piet_id FROM state, volcano WHERE contains(state.geometry, volcano.geometry)
	PostGIS with spatial indexing	SELECT DISTINCT state.piet_id FROM state, volcano WHERE state.geometry && volcano.geometry AND contains(state.geometry, volcano.geometry)
	Piet	SELECT DISTINCT p1.state FROM gis_pre_point_9 p1
Q2: List the states and the cities within them	PostGIS without spatial indexing	SELECT state.id, city.id FROM state, city WHERE contains(state.geometry, city.geometry)
	PostGIS with spatial indexing	SELECT state.piet_id, city.piet_id FROM state, city WHERE state.geometry && city.geometry AND contains(state.geometry, city.geometry)
	Piet	SELECT p1.state, p1.city FROM gis_pre_point_11 p1
Q3: List states and the cities within them, only for states crossed by at least one river	PostGIS without spatial indexing	SELECT DISTINCT state.piet_id , city.piet_id FROM state, city WHERE contains(state.geometry, city.geometry) AND state.id in (SELECT state.id FROM state, river WHERE intersects(state.geometry, river.geometry))
	PostGIS with spatial indexing	SELECT DISTINCT state.piet_id , city.piet_id FROM state, city WHERE state.geometry && city.geometry AND contains(state.geometry, city.geometry) AND state.piet_id in (SELECT state.piet_id FROM state, river WHERE state.geometry && river.geometry AND intersects(state.geometry, river.geometry))
	Piet	SELECT DISTINCT p1.state, p1.city FROM gis_pre_point_11 p1 WHERE p1.state IN (SELECT p2.state FROM gis_pre_linestring_7 p2)
Q4: List states crossed by al least ten rivers	PostGIS without spatial indexing	SELECT p1.piet_id FROM state p1, river p2 WHERE intersects(p1.geometry, p2.geometry) GROUP BY p1.piet_id HAVING count(p2.piet_id) > = 10
	PostGIS with spatial indexing	SELECT p1.piet_id FROM state p1, river p2 WHERE p1.geometry && p2.geometry AND intersects(p1.geometry, p2.geometry) GROUP BY p1.piet_id HAVING count(p2.piet_id) > = 10
	Piet	SELECT p1.state FROM gis_pre_linestring_7 p1 GROUP BY p1.state HAVING count(distinct p1.river) > = 10

Table 5
Geometric aggregation queries.

Query	Method	Code
Q5: Total number of rivers along with the total number of volcanoes in California	PostGIS without spatial indexing	SELECT count(DISTINCT river.piet_id), count(DISTINCT volcano.piet_id) FROM volcano, river, state WHERE state='California' AND contains(state.geometry, river.geometry) AND contains(state.geometry, volcano.geometry)
	PostGIS with spatial indexing	SELECT count(DISTINCT river.piet_id), count(DISTINCT volcano.piet_id) FROM volcano, river, state WHERE state='California' AND river.geometry && state.geometry AND volcano.geometry && state.geometry AND contains(state.geometry, river.geometry) AND contains(state.geometry, volcano.geometry)
	Piet	SELECT count(DISTINCT p1.river), count(DISTINCT p2.volcano) FROM gis_pre_linestring_3 p1, gis_pre_point_4 p2, state s WHERE p1.state = p2.state AND s.state= 'California' AND p2.state = s.piet_id
Q6: Average elevation of volcanoes by state	PostGIS without spatial indexing	SELECT avg(elev), state.piet_id FROM volcano, state WHERE contains(state.geometry, volcano.geometry) GROUP BY state.piet_id
	PostGIS with spatial indexing	SELECT avg(elev), state.ID FROM volcano, state WHERE volcano.geometry && state.geometry AND contains(state.geometry, volcano.geometry) GROUP BY state.piet_id
	Piet	SELECT avg(p1.elev), p2.state FROM gis_subp_point_4 p1, gis_pre_point_4 p2 WHERE p1.originalgeometryID = p2.volcano GROUP BY p2.state order by p2.state
Q7: Average elevation of volcanoes by state, only for states crossed by at least one river.	PostGIS without spatial indexing	SELECT avg(elev), state.Piet_id FROM volcano, state WHERE contains(state.geometry, volcano.geometry) AND state.PIET_id in (SELECT state.Piet_id FROM state, river WHERE intersects(state.geometry, river.geometry) GROUP BY state.Piet_ID
	PostGIS with spatial indexing	SELECT avg(elev), state.Piet_id FROM volcano, state WHERE contains(state.geometry, volcano.geometry) AND state.geometry && volcano.geometry AND state.Piet_id in (SELECT state.Piet_id FROM state, river WHERE intersects(state.geometry, river.geometry) AND state.geometry && river.geometry) GROUP BY state.piet_id
	Piet	SELECT avg(p1.elev), p2.state FROM gis_subp_point_4 p1, gis_pre_point_4 p2 WHERE p1.originalgeometryID = p2.volcano AND p2.state IN (SELECT state FROMgis_pre_linestring_3) GROUP BY p2.state
Q8: Total length of the part of each river which intersects states containing at least one volcano with elevation higher than 4300	PostGIS without spatial indexing	SELECT length(intersection(state.geometry, river.geometry)), river.Piet_id FROM river, state WHERE intersects(state.geometry, river.geometry) AND state.Piet_id in (SELECT state.Piet_id from state, volcano WHERE contains(state.geometry, volcano.geometry) AND volcano.elev >4300)
	PostGIS with spatial indexing	SELECT length(intersection(state.geometry, river.geometry)), river.Piet_id FROM river, state WHERE river.geometry && state.geometry AND intersects(state.geometry, river.geometry) AND state.Piet_id in (SELECT state.Piet_id from state, volcano WHERE state.geometry && volcano.geometry AND contains(state.geometry, volcano.geometry) AND volcano.elev >4300)
	Piet	SELECT SUM(length(p1.geometry)), p2.river FROM gis_subp_linestring_3 p1, gis_pre_linestring_3 p2 WHERE p1.uniqueID = p2.uniqueID and p1.originalgeometryID IN (SELECT p4.state FROM gis_subp_point_1 p3, gis_pre_point_4 p4 WHERE p3.originalgeometryID = p4.volcano AND p3.elev >4300) group by p2.river

selection; the size of the indexes totals 400 Mb, for all combinations of subpolygonization tables; (d) in all cases, queries were executed without the overhead of the graphic interface; (e) the queries were ran 10 times for each method, and we report the average execution times; (f) full subpolygonization was used for Piet queries.

Fig. 13 shows the execution times for the set of geometric queries. We can see that Piet clearly outperforms postGIS with or without R-tree indexing. The differences between Piet and R-tree indexing range between seven and eight times in favor of Piet; for PostGIS without indexing, these differences go from ten to fifty times.

For tests of type (c), we selected four geometric aggregation queries that compute aggregations over the result of some geometric condition which involves the intersection between different combinations of layers. Table 5 depicts the expressions for these queries. Fig. 14 showsthe results. Piet outperformed postGIS in queries Q5 through Q7 (ranging between four and five times faster with respect to indexed PostGIS), but was outperformed in query Q8. This has to do, probably, with the complicated shape of the rivers combined with the number of carrier lines generated in regions with high density of volcanoes. Note, however, that, even in this case, execution times remain compatible with user needs. Piet's performance could be improved reducing the size of the grid rectangles *only* for high density regions, taking advantage of the flexibility of the grid partition strategy. We did not use this feature, given that the improvement would only apply to this particular case, and thus, results would not be comparable.

For the experiments of type (d), we ran the following three queries adding a query region. We worked with the query regions shown in Fig. 12.

Q9: *Average elevation of volcanoes by state, for volcanoes within the query region.*

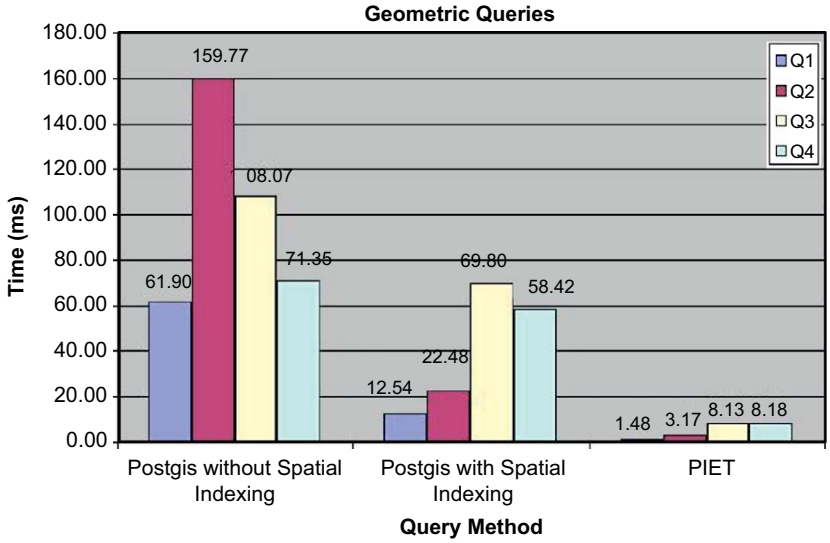

Fig. 13. Execution time for geometric queries.

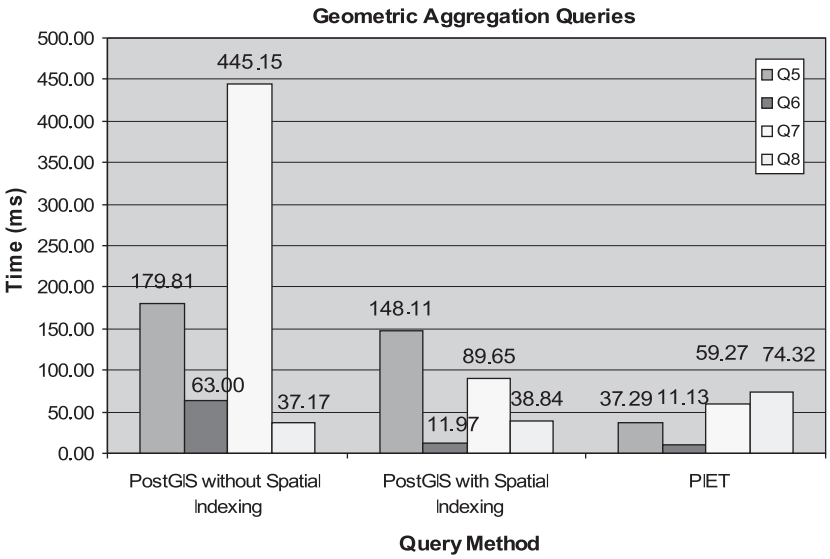

Fig. 14. Execution time for geometric aggregation queries.

Q10: *Average elevation of volcanoes by state only for the states crossed by at least one river, considering only volcanoes within the query region.*

Q11: *For each state show the total length of the part of each river which intersects it, only for states within the query regions, containing at least one volcano with elevation greater than 4300 m.*

The query expressions are of the kind of the ones given in Tables 4 and 5, and we omit them for the sake of space. The results are shown in Figs. 15 and 16. We denote query regions #1 and #2 the smaller and larger regions in Fig. 12, respectively.

Figs. 15 and 16 show the results. We can see that for the small query region, Piet still performs better than PostGIS (indexed or not) for queries Q10 and Q11. For Q9, performances were similar for the three methods. On the contrary, for the larger region, Piet delivered better performance for Q9 and Q10, but for Q11 PostGIS with R-tree outperformed Piet (since this query is similar to Q8 above, the reasons of this result are likely to be the same). We note that, in the presence of query regions, Piet pays the price of the on-the-fly computation of the intersection between the query region and the subpolygonization. We indexed the overlayed subpolygonization with an R-tree, with the intention of speeding-up the computation of the intersection between the query region and the subpolygons, but the results were not satisfactory and we discarded this strategy. As a final remark, we implemented an optimization in Piet: taking advantage of the grid partition, only the rectangles that intersect were the

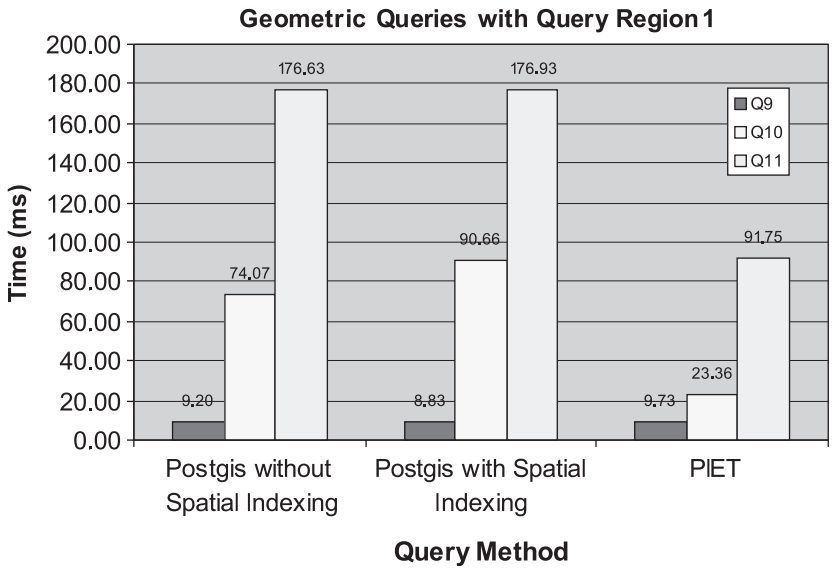

Fig. 15. Geometric aggregation within query region # 1.

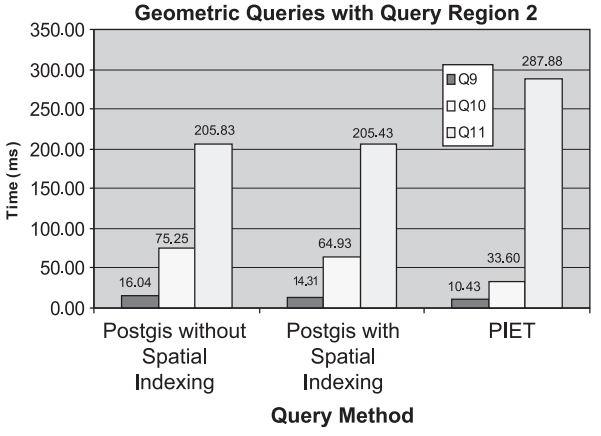

Fig. 16. Geometric aggregation within query region # 2.

region boundaries were considered (i.e., the algorithm that computes the intersection only analyzes the *relevant* rectangles).

Precision of Piet Aggregation. We have commented above that, in some cases, we may lose precision in Piet when we aggregate measures defined over geometric objects. This problem appears when the object associated to measure to be aggregated does not lie within the query region (this also occurs in aR-trees, as we comment below). We ran a variation of query Q8: "*length of rivers within a query region*". Here, the boundary of the region is crossed by some rivers, and we measured the difference between the lengths computed by Piet and by postGIS (the latter always gives the exact result). Table 6 shows the results. The object ID represents the river being measured.

For all the rivers, except the ones with IDs 250 and 258 (crossed by the query region), the result returned by Piet is exact. When this is not the case, the error is less than

Table 6
Precision in Piet.

Object ID	Exact length	Computed by Piet	Diff. (%)
55	0.594427175	0.59442717	0
250	1.33177252	1.272456	4.7
251	0.2424391242	0.24243912	0
252	0.67318281	0.6731828	0
253	0.5103286611	0.510328661	0
254	0.0955072453	0.09550724	0
258	0.636150619	0.59679889	6.7

index on geometry	# tuples	# pages	disk space (KB)
City	1767	12	98
Volcano	719	5	40
River	353	3	24
State	151	1	8.2

Fig. 17. Disk space used by the R-tree or aR-tree indexing techniques.

7%. Notice that this error could be fixed in Piet assuming the overhead of computing the exact length (inside the query region) of the segments that are intersected by the region boundaries. Thus, we think precision is acceptable given that, on the one hand, errors can appear only when a query region is intersected by an object that is being measured, and, on the other hand, the error can always be fixed at the expense of an overhead, that varies in each situation.

Aggregation R-Trees. The aR-tree stores, for each minimum bounding rectangle (MBR), the value of the aggregation function for all the objects enclosed by the MBR. For our experimental evaluation, we implemented the aR-tree and ran two geometric aggregation queries,

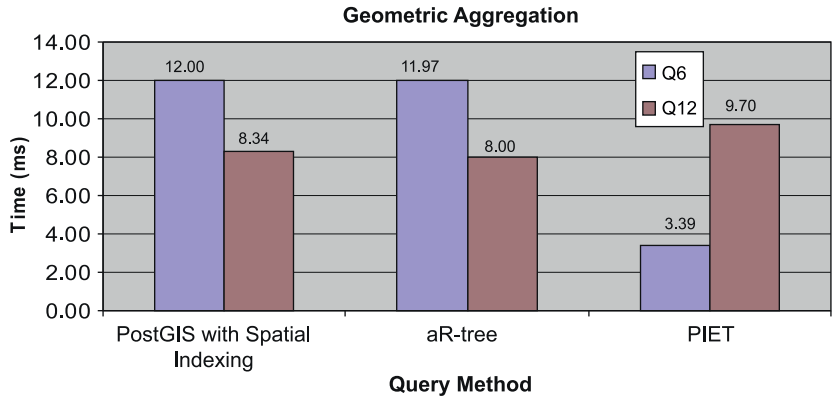

Fig. 18. Piet vs. aR-tree and R-tree.

with or without a query region. Fig. 17 shows the space used by the R-tree and the aR-tree indexes (the space is approximately the same in both cases), for each map layer. We can see that, as expected, the space required by indexing is considerably smaller than for storing the overlay precomputation. We report the results obtained running two queries:

Q6: "*Average elevation of volcanoes by state*" (a geometric aggregation, with no query region defined); and

Q12: "*Maximum elevation of volcanoes within a query region in California*".

The height of the aR-tree was $h = 2$, with six minimum bounding rectangles (MBR) in the first level. Fig. 18 shows the results. We can see that for Q6, Piet is still much better than the other two methods. However, in the presence of a query region, aR-tree and R-tree are between 15% and 20% better than Piet. Again, on-the-fly computation of the query region makes the difference. However, we remark that the reported results were obtained in situations that favor aR-trees, since the queries deal with *points*. Aggregation over other kinds of objects that do overlap the query region may not be so favorable to aR-trees, given that base tables must be accessed, or otherwise, precision may be poor. The main benefit of aR-trees with respect to R-trees comes from pruning tree traversal when a region is completely included in an MBR, because, in this case, they do not need to reach the leaves of the index tree (because the values associated to all the geometries enclosed by the MBR have been aggregated). However, if this is not the case, the aR-tree must reach the leaves, as standard R-trees do, and the aR-tree advantages are lost.

Finally, we ran several tests of type (e). We already explained that GISOLAP-QL queries are composed of a GIS part and an OLAP part, expressed in MDX. The times for computing the GIS part (the SQL-like expression) were similar to the ones reported above. Note that in this case, there is no tool to compare Piet against to. We measured the total time for running a query, composed of the time needed for building the query, and the query execution time. As an example, we report the result of the following query:

Q13: Unit Sales, Store Cost and Store Sales for the products and promotion media offered by stores only in states containing at least one volcano. The GISOLAP-QL query reads:

```
SELECT layer.state; FROM PietSchema; WHERE
contains
  (layer.state,layer.volcano,subplevel.
point);

                |

SELECT [Measures].[Unit Sales],
[Measures].[Store Cost], [Measures].[Store
Sales]
ON columns, ([Promotion Media].[All Media],
[Product].[All Products])
ON rows
FROM [Sales]
```

The query assembly and execution times are:

Assembly (ms)	Execution (ms)	Total (ms)
2023	60	2083

8.1. Discussion

Our results showed that the overlay precomputation of the common subpolygonization appears as an interesting alternative to other more traditional methods, contrary to what has been believed so far [18]. In addition, the class of queries that clearly benefit from overlay precomputation can be identified by a query processor, and added to any existing GIS in a straightforward way. We summarize our results as follows:

- For pure geometric queries, Piet (i.e. overlay precomputation) clearly outperformed R-trees.
- For geometric aggregation over the entire map (i.e., when no query region must be intersected at run time with the precomputed sub-polygons), Piet clearly outperformed the other methods in almost all the experiments.
- When a query region is present, indexing methods and overlay precomputation deliver similar performance.

As a general rule, the performance of overlay precomputation improves as the query region turns smaller.
- Piet always delivered execution times compatible with user needs.
- The cost of integrating GIS results and OLAP navigation capabilities through the GISOLAP-QL query language (i.e., merging the GIS part results with the MDX expression) goes from low to negligible.
- For very large and complicated maps, with large query regions, aR-trees have the potential to outperform the other two techniques.

9. Conclusion

In this paper we introduced a formal model that integrates GIS and OLAP applications in an elegant way. We formalized the notion *geometric aggregation*, and identified a class of queries, denoted *summable*, which can be evaluated without accessing the algebraic part of the GIS dimensions.

We proposed to precompute the *common subpolygonization* of the overlay of thematic layers as an alternative optimization method for evaluation of summable queries. We sketched a query language for GIS and OLAP integration, and described a tool, denoted Piet, that implements our proposal. The results of our experimental evaluation (carried out over real-world maps) show that precomputing the *common subpolygonization* can, for several kinds of geometric queries, outperform traditional query evaluation methods. We believe this is a relevant result, given that, up till now it has been thought that overlay materialization was not competitive against traditional search methods for GIS queries [18]. In addition, it is straightforward to add overlay precomputation to existing query optimizers, as an alternative query processing strategy.

Our future work has two main directions: on the one hand, we think there is still room to improve the performance of query processing using overlay precomputation. On the other hand, we are looking forward to apply the concepts presented in this paper in the Moving Objects Databases setting.

Appendix A

A simple example of a one-dimensional Dirac delta function [8] (or impulse function) $\delta_a(x)$ for a real number a can be $\lim_{\varepsilon\to\infty} f_a(\varepsilon,x)$, where $f_a(\varepsilon,x)=\varepsilon$ if $a-1/2\varepsilon\leqslant x\leqslant a+1/2\varepsilon$ and $f_a(\varepsilon,x)=0$ elsewhere. For a two-dimensional point (a,b) in $\mathbb{R}^2$, we can define the two-dimensional Dirac delta function $\delta_{(a,b)}(x,y)$ as $\lim_{\varepsilon\to\infty} f_{(a,b)}(\varepsilon,x,y)$, with $f_{(a,b)}(\varepsilon,x,y)=\varepsilon^2$ if $a-1/2\varepsilon\leqslant x\leqslant a+1/2\varepsilon$ and $b-1/2\varepsilon\leqslant y\leqslant b+1/2\varepsilon$ and $f_{(a,b)}(\varepsilon,x,y)=0$ elsewhere.

If C is a *finite set of points in the plane*, then the *delta function of* C, $\delta_C(x,y)$, is defined as $\sum_{(a,b)\in C}\delta_{(a,b)}(x,y)$. It has the property that $\int\int_{\mathbb{R}^2}\delta_C(x,y)\,dx\,dy$ is equal to the cardinality of C. Intuitively, including a Dirac delta function in geometric aggregation, allows to express geometric aggregate queries like "number of airports in a region C". If C is a *one-dimensional curve*, then the definition of $\delta_C(x,y)$ is more complicated. Perpendicular to C we can use a one-dimensional Dirac delta function, and along C, we multiply it with a combination of Heaviside step functions[21]. The one-dimensional Heaviside step function is defined as $H(x)=1$ if $x\geqslant 0$ and $H(x)=0$ if $x<0$. For C, we can define a Heaviside function $H_C(x,y)=1$ if $(x,y)\in C$ and $H_C(x,y)=0$ outside C. As a simple example, let us consider the curve C given by the equation $y=0\wedge 0\leqslant x\leqslant S$. The function $\delta_C(x,y)$, in this case, can be defined as $\delta_0(y)\cdot H(x)\cdot H(S-x)$. The one-dimensional Dirac delta function $\delta_0(y)$ takes care of the fact that perpendicular to C, an impulse is created. The factors $H(x)$ and $H(S-x)$ take care of the fact that this impulse is limited to C. In this case, it is easy to see that $\int\int_{\mathbb{R}^2}\delta_C(x,y)\,dx\,dy$ is the length of C and in fact this is true for arbitrary C. For arbitrary C, the definition of δ_C is rather complicated and involves the use of $H_C(x,y)$. We omit the details. Intuitively, this combination of functions allows to express geometric aggregate queries like "Give me the length of the Colorado river".

References

[1] Y. Bédard, T. Merret, J. Han, Fundamentals of spatial data warehousing for geographic knowledge discovery, Geographic Data Mining and Knowledge Discovery, 2001, pp. 53–73.
[2] Y. Bédard, S. Rivest, M. Proulx, Spatial online analytical processing (SOLAP): concepts, architectures, and solutions from a geomatics engineering perspective, in: Wrembel-Koncilia (Ed.), Data Warehouses and OLAP: Concepts, Architectures and Solutions, IRM Press, 2007, pp. 298–319 (Chapter 13).
[3] S. Bimonte, A. Tchounikine, M. Miquel, Towards a spatial multidimensional model, in: DOLAP'05, 2005, pp. 39–46.
[4] L. Cabibbo, R. Torlone, Querying multidimensional databases, in: Proceedings of the DBPL'97, East Park, Colorado, USA, 1997, pp. 253–269.
[5] A.K. Chandra, D. Harel, Computable queries for relational data bases, J. Comput. Syst. Sci. 21 (2) (1980) 156–178.
[6] S. Cohen, Equivalence of queries combining set and bag-set semantics, in: Proceedings of the PODS'06, ACM Press, New York, 2006, pp. 70–79.
[7] M. De Berg, M. Van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, second ed., Springer, Berlin, 2000.
[8] P. Dirac, The Principle of Quantum Mechanics, Oxford University Press, Oxford, 1958.
[9] H. Edelsbrunner, L.J. Guibas, J. Stolfi, Optimal point location in a monotone subdivision, SIAM J. Comput. 15 (2) (1986) 317–340.
[10] A. Escribano, L. Gómez, B. Kuijpers, A. Vaisman, Piet: a GIS-OLAP implementation, in: DOLAP'07, 2007.
[11] R. Fidalgo, V. Cesário Times, J. Da Silva, F. Fonseca, Geodwframe: a framework for guiding the design of geographical dimensional schemas, in: DaWaK, 2004, pp. 26–37.
[12] J. Flusser, Affine invariants of convex polygons, IEEE Trans. Image Process. 11 (9) (2002) 1117–1118.
[13] S. Grumbach, T. Milo, Towards tractable algebras for bags, J. Comput. Syst. Sci. 52 (3) (1996) 570–588.
[14] A. Gutman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the SIGMOD'84, 1984, pp. 47–57.
[15] S. Haesevoets, B. Kuijpers, Time-dependent affine triangulation of spatio-temporal data, in: Proceedings of the ACM-GIS'04, Washington, DC, USA, 2004, pp. 57–66.
[16] S. Haesevoets, A triangle-based logic for affine-invariant querying of two-dimensional data, in: Proceedings of the CDB'04, Paris, France, 2004, pp. 53–74.
[17] J. Han, N. Stefanovic, K. Koperski, Selective materialization: an efficient method for spatial data cube construction, in: Proceedings of PAKDD'98, 1998, pp. 144–158.
[18] J. Han, M. Kamber, Data Mining, Concepts and Techniques, Morgan Kaufmann Publishers, Los Altos, CA, 2001.

[19] V. Harinarayan, A. Rajaraman, J. Ullman, Implementing data cubes efficiently, in: Proceedings of the SIGMOD'96, Montreal, Canada, 1996, pp. 205–216.
[20] J. Hellerstein, J. Naughton, A. Pfeffer, Generalized search trees for database systems, in: VLDB, 1995, pp. 562–573.
[21] R. Hoskins, Generalised Functions, Ellis Horwood Series: Mathematics and its Applications, Wiley, New York, 1979.
[22] C. Hurtado, A. Mendelzon, A. Vaisman, Maintaining data cubes under dimension updates, in: Proceedings of the IEEE/ICDE'99, 1999, pp. 346–355.
[23] C. Jensen, A. Kligys, T. Pedersen, I. Timko, Multidimensional data modeling for location-based services, VLDB J. 13 (1) (2004) 1–21.
[24] R. Kimball, The Data Warehouse Toolkit, Wiley, New York, 1996.
[25] A. Klug, Equivalence of relational algebra and relational calculus query languages having aggregate functions, J. ACM 29 (3) (1982) 699–717.
[26] B. Kuijpers, D.V. Gucht, Genericity in spatial databases, in: J. Paredaens, G. Kuper, L. Libkin (Eds.), Constraint Databases, Springer, Berlin, 2000, pp. 293–304 (Chapter 12).
[27] G. Kuper, M. Scholl, Geographic information systems, in: J. Paredaens, G. Kuper, L. Libkin (Eds.), Constraint Databases, Springer, Berlin, 2000, pp. 175–198 (Chapter 12).
[28] H. Lenz, A. Shoshani, Summarizability in OLAP and statistical data bases, in: Proceedings of the Ninth International Conference on Scientific and Statistical Database Management, August 11–13, 1997, Olympia, Washington, USA, IEEE Computer Society, 1997, pp. 132–143.
[29] E. Malinowski, E. Zimányi, OLAP hierarchies: a conceptual perspective, in: CAiSE, 2004, pp. 477–491.
[30] E. Malinowski, E. Zimányi, Representing spatiality in a conceptual multidimensional model, in: GIS, 2004, pp. 12–22.
[31] E. Malinowski, E. Zimányi, Logical representation of a conceptual model for spatial data warehouses, Geoinformatica 11 (4) (2007) 431–457.
[32] O. Nigro, S.G. Cisaro, D. Xodo, Data Mining with Ontologies: Implementations, Findings and Frameworks, Idea Group, 2007.
[33] D. Papadias, P. Kalnis, J. Zhang, Y. Tao, Efficient OLAP operations in spatial data warehouses, in: Proceedings of the SSTD'01, 2001, pp. 443–459.
[34] D. Papadias, Y. Tao, P. Kalnis, J. Zhang, Indexing spatio-temporal data warehouses, in: Proceedings of the ICDE'02, 2002, pp. 166–175.
[35] C.H. Papadimitriou, D. Suciu, V. Vianu, Topological queries in spatial databases, in: Proceedings of the PODS'96, ACM Press, New York, 1996, pp. 81–92.
[36] J. Paredaens, J.V. den Bussche, D.V. Gucht, Towards a theory of spatial database queries, in: Proceedings of the PODS'94, New York, 1994, pp. 279–288.
[37] J. Paredaens, G. Kuper, L. Libkin (Eds.), Constraint databases, Springer, Berlin, 2000.
[38] T. Pedersen, N. Tryfona, Pre-aggregation in spatial data warehouses, in: Proceedings of the SSTD'01, 2001, pp. 460–480.
[39] E. Pourabas, Cooperation with geographic databases, in: M. Raffanelli (Ed.), Multidimensional Databases, Idea Group, 2003, pp. 166–199 (Chapter 13).
[40] F. Rao, L. Zang, X. Yu, Y. Li, Y. Chen, Spatial hierarchy and OLAP-favored search in spatial data warehouse, in: Proceedings of the DOLAP'03, Louisiana, USA, 2003, pp. 48–55.
[41] P. Rigaux, M. Scholl, A. Voisard, Spatial Databases, Morgan Kaufmann, Los Altos, CA, 2002.
[42] S. Rivest, Y. Bédard, P. Marchand, Toward better support for spatial decision making: defining the characteristics of spatial, on-line analytical processing (SOLAP), Geomatica 55 (4) (2001) 539–555.
[43] S. Shekhar, C. Lu, X. Tan, S. Chawla, R. Vatsavai, Mapcube: a visualization tool for spatial data warehouses, in: H. Miller, J. Han (Eds.), Geographic Data Mining and Knowledge Discovery (GKD), Taylor & Francis, London, 2001, pp. 74–109.
[44] A. Shoshani, OLAP and statistical databases: similarities and differences, in: Proceedings of the ACM TODS, 1997.
[45] N. Stefanovic, J. Han, K. Koperski, Object-based selective materialization for efficient implementation of spatial data cubes, IEEE Trans. Knowl. Data Eng. 12 (6) (2000) 938–958.
[46] J.d. Silva, A. Cesário Times, V. Salgado, An open source and web based framework for geographic and multidimensional processing, in: SAC, 2006, pp. 63–67.
[47] J.d. Silva, A.S. Castro Vera, A. Oliveira, R. Fidalgo, A. Salgado, V. Times, Querying geographical data warehouses with GEOMDQL, in: SBBD, 2007, pp. 223–237.
[48] I. Vega López, R. Snodgrass, B. Moon, Spatiotemporal aggregate computation: a survey, IEEE Trans. Knowl. Data Eng. 17 (2) (2005) 271–286.
[49] M.F. Worboys, GIS: A Computing Perspective, Taylor & Francis, London, 1995.
[50] L. Zang, Y. Li, F. Rao, X. Yu, Y. Chen, An approach to enabling spatial OLAP by aggregating on spatial hierarchy, in: Proceedings of the DaWak'03, Prague, Czech Republic, 2003, pp. 35–44.