Piet: a GIS-OLAP Implementation

Ariel Escribano
Universidad de Buenos Aires
Ciudad Universitaria PI,
Bs.As., Argentina
aescribano@dc.uba.ar

Leticia I. Gomez
Instituto Tecnólogico de
Buenos Aires
Av. E. Madero 399
Bs.As., Argentina
lgomez@itba.edu.ar

Bart Kuijpers
University of Hasselt,
Departement WNI
Gebouw D, B-3590
Diepenbeek, Belgium
bart.kuijpers@uhasselt.be

Alejandro A. Vaisman
Universidad de Buenos Aires
Ciudad Universitaria PI,
Bs.As., Argentina
avaisman@dc.uba.ar

ABSTRACT

Data aggregation in Geographic Information Systems (GIS) is a desirable feature, although only marginally present in commercial systems, which also fail to provide integration between GIS and OLAP (On Line Analytical Processing). With this in mind, we have developed Piet, a system that makes use of a novel query processing technique: first, a process called *sub-polygonization* decomposes each thematic layer in a GIS, into open convex polygons; then, another process computes and stores in a database the overlay of those layers for later use by a query processor. We describe the implementation of Piet, and provide experimental evidence that *overlay precomputation* can outperform GIS systems that employ indexing schemes based on R-trees.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial Databases and GIS; H.4.2 [**Information Systems Applications**]: Decision Support

General Terms

Experimentation, Performance

Keywords

GIS, OLAP, View Materialization

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 [15]. In GIS, geometric objects within a map are organized in *thematic layers* where spatial objects may also be annotated with numerical or categorical information. Typical queries in GIS ask for geometric objects that satisfy some condition, or, involve the aggregation of geographic measures (i.e. area, length). To evaluate queries efficiently, index structures for geometric objects are used, like some variation of R-tree [1]. Novel techniques like aR-trees [11] have also been proposed for dealing with aggregate data. Although it is usual in GIS practice to store non-spatial data in the thematic layers, when aggregation becomes important, it would be advisable to organize the non-spatial GIS data in a data warehouse. OLAP (On Line Analytical Processing) [8] comprises a set of tools and algorithms for querying multidimensional databases containing large amounts of data, usually called data warehouses, organized as sets of facts and dimension hierarchies. Efficient evaluation of OLAP queries requires, more often than not, the use of some kind of precomputation [5].

Our work is aimed at integrating these two different worlds in a single framework [2, 9]. In this way, we will be able to evaluate a query like "total income in provinces crossed by at least one river", and navigate the results in the usual OLAP way. After an overview of the data model(Section 2), we describe the implementation of *Piet*, a system (named after the Dutch painter Piet Mondrian) that accomplishes the goals mentioned above (Section 3) [1]. Piet integrates open source GIS and OLAP technologies, introducing the concept of overlay precomputation, that makes use of a novel technique denoted *common sub-polygonization*, which decomposes each thematic layer into open convex polygons. We discuss *scalability* and *error management* (Section 4). Finally, we report experimental results over real-world maps (Section 5) which show that overlay precomputation can outperform other well-known query optimization techniques.

Running Example. Throughout the paper we will be using a real-world map of Belgium, consisting of five layers, containing geographic information about rivers, regions, provinces,

[1] Documentation and a demo can be found at http://piet.exp.dc.uba.ar/piet/

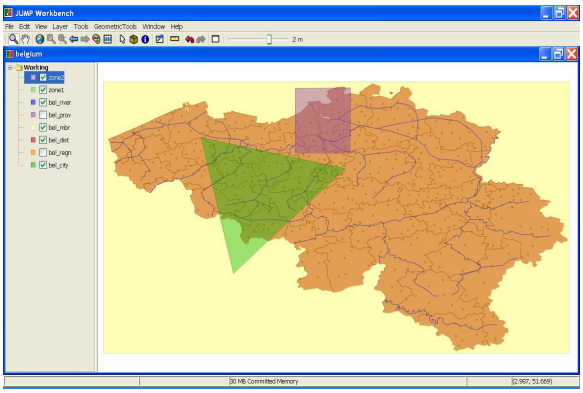

Figure 1: Running example: a map of Belgium

districts and cities. Additionally, the maps contain demographic and economic information. The rivers are represented as polylines, the cities as points, and the other layers as polygons. Figure 1 shows the map including a layer containing the definition of two regions of interest for a query (see Section 5). The maps were obtained from the spatial library of the GIS Center [2]. There is also a data warehouse with information about stores and sales for different regions of Belgium.

1.1 Related Work

In general, the information in a GIS application is divided over several *thematic layers*. The information in each layer consists of purely spatial data on the one hand that is combined with classical alpha-numeric attribute data on the other hand (usually stored in a relational database). For spatial data representation we will use (like most current GIS do) the *vector model* [10]. Here, infinite sets of points in space are represented as finite geometric structures, or *geometries*. More concretely, vector data within a layer consists of a finite number of tuples of the form (*geometry*, *attributes*), where a geometry can be a point, a polyline or a polygon. There are several possible data structures to actually store these geometries [19].

Although some authors have pointed out the benefits of combining GIS and OLAP, not much work has been done in this field. Vega López *et al* [18] presented a comprehensive survey on spatiotemporal aggregation that includes a section on spatial aggregation. Rivest *et al.* [16] introduced the concept of SOLAP (standing for Spatial OLAP), and described the desirable features and operators a SOLAP system should have, without giving a formal model for this. Han *et al.* [3] used 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 [13] proposed 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. However, this proposal ignores the geometry, and only addresses polygons. Thus, queries like "Give me the total population of cities crossed by a river" would not be supported. No experimental results are provided either. Extending this model, Jensen *et al.* [7] proposed a multidimensional data model for mobile services. This model omits considering the geometry, limiting the set of queries that can be addressed. With a different approach, Rao *et al* [14] 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. The only geometries studied in this proposal are points, a quite unrealistic assumption in a real GIS environment. Other proposals in the area of indexing spatial and spatio-temporal data warehouses [11] combine indexing with pre-aggregation, resulting in a structure denoted *Aggregation R-tree* (aR-tree), which annotates each MBR (Minimal 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 5).

In summary, the discussion above shows that the problem of integrating spatial and warehousing information in a single framework is still in its infancy.

[2] http://giscenter-sl.isu.edu

2. DATA MODEL OVERVIEW

The implementation we present in this work is based in the data model introduced in [2, 9]. There, the authors proposed a model where spatial and non-spatial data are integrated in a single framework. The model defines a GIS dimension composed of a set of graphs, each one describing a set of geometries in a thematic layer. A GIS dimension is considered, as usual, as composed of a schema and instances. Figure 2 shows the schema of a GIS dimension: the bottom level of each hierarchy, denoted the *Algebraic part*, contains the infinite points in a layer, and could be described by means of linear algebraic equalities and inequalities [12]. Above this part there is the *Geometric part,* that stores the identifiers of the geometric elements of GIS and is used to solve the geometric part of a query (e.g., find the polylines in a river representation). Each point in the Algebraic part may correspond to one or more elements in the Geometric part (e.g., if more than one polylines intersect with each other). Thus, at the *GIS dimension instance* level we will have rollup *relations* (denoted $r_L^{geom_1 \to geom_2}$). For instance, $r_{L_{city}}^{Point \to Pg}(x, y, pg_1)$ says that, in a layer L_{city}, a point (x, y) corresponds to a polygon identified by pg_1 in the Geometric part. We will see that the mechanism used for precomputation of the overlayed layers in the map, will take us back to the concept of rollup function, where a point (x,y) will correspond to *exactly* one geometry identifier. Finally, there is the *OLAP part* for storing non-spatial data. This part contains the conventional OLAP structures, as defined in [6]. The levels in the geometric part are associated to the OLAP part via a function, denoted $\alpha_{L,D}^{dimLevel \to geom}$. For instance, $\alpha_{L_r,Rivers}^{riverId \to g_r}$ associates information about a river in the OLAP part ($riverId$) in a dimension *Rivers*, to the identifier of a polyline (g_r) in a layer containing rivers (L_r) in the Geometric part.

EXAMPLE 1. *Figure 2 shows a GIS dimension schema, where we defined three layers, for rivers, cities, and provinces, respectively. The schema is composed of three graphs; the graph for rivers, for instance, contains edges saying that a point (x, y) in the algebraic part relates to line identifiers in the geometric part, and that in the same portion*

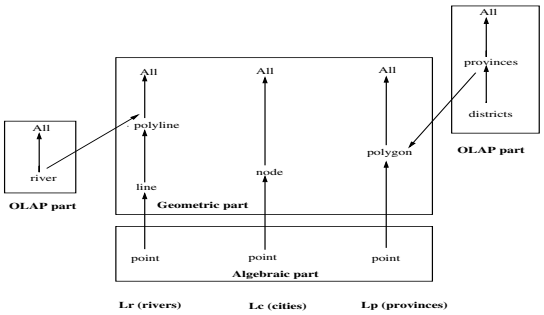

Figure 2: An example of a *GIS* dimension Schema

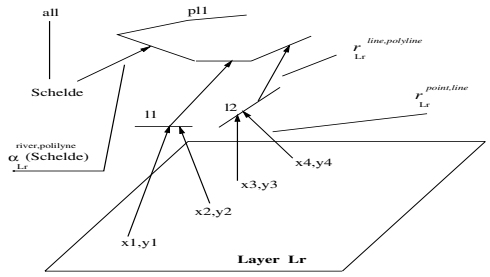

Figure 3: A *GIS* dimension instance for Figure 2.

of the dimension, lines relate to polyline identifiers. In the OLAP part we have two dimensions, representing districts and rivers, associated to the corresponding graphs, as the figure shows. For example, a river identifier at the bottom layer of the dimension representing rivers in the OLAP part, is mapped to the polyline level in the geometric part in the graph representing the structure of the rivers layer.

Figure 3 shows a portion of a GIS dimension instance for the rivers layer L_r in the dimension schema of Figure 2. We can see that an instance of a GIS dimension in the OLAP part is associated (via an α function) to the polyline pl_1, which corresponds to the Schelde river, in Antwerp. For clarity, we only show four different points at the point level $\{(x_1, y_1), \ldots, (x_4, y_4)\}$. There is a relation $r_{L_r}^{point,line}$ containing the association of points to the lines in the line *level. Analogously, there is also a relation $r_{L_r}^{line,polyline}$, between the line and polyline levels, in the same layer.* □

Elements in the geometric part can be associated with *facts*, each fact being quantified by one or more *measures*, not necessarily a numeric value. Besides the GIS fact tables, there may also be classical fact tables in the OLAP part, defined in terms of the dimension schemas. For instance, instead of storing the population associated to a polygon identifier, as in Example 2, the same information may reside in a data warehouse, with schema $(state, Population)$.

Geometric Aggregation. Based on the data model described above, the notion of *geometric aggregation* was defined. However, general geometric aggregation queries are hard to evaluate, because they require the computation of a double integral representing the area where some condition is satisfied. Thus, Piet addresses a class of queries denoted *summable*, of the form: $\sum_{g \in S} h(g)$, where h is a function (represented, for instance, by a fact table), and the sum is performed over all the identifiers of the objects that satisfy a condition. For example, the query "total population of the cities crossed by the river "Schelde" reads:

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

$$C = \{g_{id} \mid (\exists x)(\exists y)(\exists pl_1)(\exists c \in dom(Ci)) \\ (\alpha^{\mathrm{Ri}\to\mathrm{Pl}}_{L_r, Rivers}(\text{'Schelde'}) = (pl_1) \wedge r^{\mathrm{Pt}\to\mathrm{Pl}}_{L_r}(x, y, pl_1) \wedge \\ \alpha^{\mathrm{Ci}\to\mathrm{Pg}}_{L_c, Districts}(c) = g_{id} \ \wedge \ r^{\mathrm{Pt}\to\mathrm{Pg}}_{L_c}(x, y, g_{id}))\}.$$

The meaning of the query is: $\alpha^{\mathrm{Ri}\to\mathrm{Pl}}_{L_r, Rivers}$('Schelde') maps the identifier of the Schelde river to a polyline in layer L_r (representing rivers). The relation $r^{\mathrm{Pt}\to\mathrm{Pl}}_{L_r}(x, y, pl_1)$ contains the mapping between the points and the polylines representing the rivers that satisfy the condition. The other functions are analogous. Thus, the identifiers of the geometric elements that satisfy both conditions can be retrieved, and the sum of ft_{pop} (which represents the population associated to a polygon g_{id}) over these objects can be performed.

Overlay Precomputation. Many interesting queries in GIS require computing intersections, unions, etc., of objects that are in different layers. Hereto, their overlay has to be computed. For the summable queries defined above, on-the-fly computation of the sets "C" containing all those cities would be costly, mainly because most of the time we will 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). Piet implements a different strategy, consisting in three steps: (a) partitioning each layer in subgeoemetries, according to the carrier lines defined by the geometries in each layer (see below); this allows to detect which geographic regions are common to the layers involved; (b) precomputing the overlay operation; (c) evaluating the queries using the layer containing all the precomputed subgeometries . We will show in Section 5 that this strategy can be an efficient alternative for query evaluation.

To conclude this section, we will give some definitions. We will work within a bounding box $\mathrm{B} \times \mathrm{B}$ in $\mathbb{R}^2$, where B is a closed interval of $\mathbb{R}$, as it is usual in GIS practice.

Definition 1 (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 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 of a layer L *is defined as* $C_{\mathrm{L}} = C_{\mathrm{pl}} \bigcup C_{\mathrm{pg}} \bigcup C_{\mathrm{p}}$. □

Figure 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. Thus, the rollup relations r will turn into functions (given that no two points can map to the same open convex polygon). Given C_{L} and a bounding box, we denote the *convex polygonization of* L, the set of open convex polygons, open line segments and points, induced by C_{L}, that are strictly inside the bounding box. Given two layers L_1

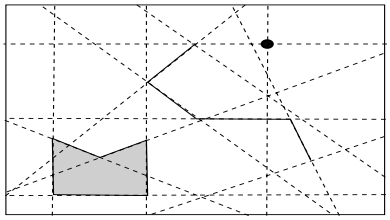

Figure 4: The carrier sets of a point, a polyline and a polygon are the dotted lines.

and L_2, and their carrier sets C_{L_1} and C_{L_2}, the *common sub-polygonization* of L_1 according to L_2, denoted $CSP(L_1, L_2)$ 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}. This can be generalized for more than two layers.

The reader may wonder, at this point, why do we use the carrier lines to divide the map in zones using the boundaries of the geometries involved. The reason is that using the sub-polygonization instead of dividing the map into arbitrary rectangles in order to approximate the original shape of a geometry increases the number of sub-geometries (i.e., rectangles) required to minimize errors. This idea is similar to Riemann's Integral Approximation [17].

3. PIET IMPLEMENTATION

System Architecture. The architecture is divided into three modules. The first module (called *raw data setup*) gathers information and stores the acquired data in a data warehouse and a map (geometric data, with possibly some relationship with the OLAP part). The whole process of gathering and storing information is performed using a data loader component. The second module (denoted *precalculated data generator*) allows the storage and execution of precomputed data. Its main component processes raw data and generates (off-line) the following information: (a) Precomputed data: containing the sub-geometries generated in the sub-poligonization step, fact values associated to those sub-geometries (conforming a GIS dimension), and overlay precalculated information for the original geometric components of the map; (b) Metadata: describing the data structures of the data warehouse, the maps, and their association; (c) GIS-OLAP relations: containing the information needed to associate geometric components and warehouse data (e.g., a point in the map can be related with a store identifier in the data warehouse). The third module (*query processor*) allows running four kinds of queries (see Section 5): geometric, geometric aggregation, OLAP and GIS-OLAP queries, based on the raw and the precalculated data generated in the previous steps.

Implementation Details. Our framework was developed using PostgreSQL 8.2.3 database[3] with Postgis 1.2 spatial extensions[4]. The source code was developed with Java 1.5. The geometric functions used belong to the JTS library. The Web Plug-in run under Tomcat-Apache 5.5 WebServer. The stand-alone plug-in runs under Jump 1.2[5]. For OLAP queries we used Mondrian[6] and the MDX query language, an industry OLAP standard[7]. We will explain some of these components as we progress in the paper.

There are two interfaces for running queries in Piet: *Piet-Jump*, which provides a Graphical User Interface (GUI) for displaying maps and running geometric queries, and *Piet-Web*, which allows running GISOLAP-QL queries (described below). A component called Piet-Schema contains a set of metadata definitions, used by different modules of the system. These definitions include: the storage location of the geometric components and their associated measures, the sub-geometries corresponding to the sub-polygonization of all the layers in a map, and the relationships between the geometric components and the OLAP information used to answer integrated GIS and OLAP queries. Metadata are stored in XML documents containing three kinds of elements: `Sub/polygonization`, `Layer`, and `Measure`. We omit the description of these documents due to space limitations.

Sub-polygonization. As we explained above, the common sub-poligonization of a layer requires the computation of the overlay of the thematic layers, using the carrier lines of Definition 1. These carrier lines induce points, linestrings, and polygons, common to the layers involved. After producing the carrier lines, the procedure continues by intersecting pairs of carrier lines, and obtains the set of sub-nodes associated to each of these lines, using each pair of sub-nodes on a carrier line to create a so-called sub-line. Finally, a method called *Polygonizer* generates the convex sub-polygons using these sub-lines. The procedure used for creating sub-lines prevents either duplicates or lines too similar to each other, as well as lines not belonging to a polygon.

Overlay Precomputation. The original geometries in different layers overlap if they have in common one of the following: points, sub-lines of the carrierset (generated by the intersection of the carrier lines), or sub-polygons (the open convex polygons generated by the sub-lines). The following algorithm sketches the overlay computation (we used self-descriptive function names for the sake of brevity).

1. listLayers = determineListOfLayersInvolvedInTheQuery()
2. geoComponents = determineListOfGeometries(listLayers)
3. carrierlines = generateCarrierLines(geoComponents)
4. subgeometries = generateSubgeometriesPropagateFacts(carrierlines)
5. calculatePreoverlay(subgeometries)

In step 4, the original geometric components are divided into points, sub-lines and sub-polygons due to the overlay. At the same time, the associated numeric fact values are propagated proportionally to the area or length of the original one. This information is stored in a table called `Subpolygonization`. Step 5 computes the original geometric objects that have sub-parts in common, and stores their identifiers in a table called `Preoverlay`. This table also stores the identifiers of the common sub-geometries.

Note that geometric queries that do not require fact aggregation (e.g.,"*total number of regions crossed by rivers*") can be answered using just the `Subpolygonization` table

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

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

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

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

[7] http://msdn.microsoft.com

(this fact is reflected in our experimental results). More complex kinds of queries also require the information stored in the `Preoverlay`. For example, "*total number of employees working in agricultural activities in regions crossed by rivers*". In this case, the `Preoverlay` table is used first to find the common sub-geometries for the layers containing regions and rivers; after this, `Sub-polygonization` is used to find the values of the proportional facts previously computed. Finally, for aggregate queries with geometric constraints in the OLAP environment, `Preoverlay` is used to find the geometry id's of the original layers, which are later used to access the layer table and the corresponding OLAP dimension values.

Query Language. We devised a query language that combines, in a single expression, GIS and OLAP statements. Queries can be submitted to Piet in two ways: using the Piet-Jump interface, or through a query language, denoted GISOLAP-QL. A GISOLAP-QL query is of the form: *GIS-Query | OLAP-Query*. The 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 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` *conjunctions/disjuntions of geometric operations*;

The `SELECT` clause contains a list of layers and/or measures, defined in the corresponding Piet-Schema of the `FROM` clause. The `WHERE` clause consists of conjunctions and/or disjunctions of geometric operations applied over elements in the layers involved. The expression also includes the kind of sub-geometry used to perform the operation. The syntax for an operation is of the form *operation name(list of layer members, sub-geometry)*. The accepted values for *sub-geometry* are "Point", "LineString" and "Polygon"[8]. Note that this syntax actually implements the first-order language commented in Section 2: the GIS part obtains the objects that satisfy the condition "C", and the OLAP part performs the aggregation over these objects.

EXAMPLE 2. *Consider the query* "Unit Sales, Store Cost and Store Sales for products and promotion media offered by stores only in provinces crossed by rivers*".*

```
SELECT layer.bel_prov;
FROM PietSchema;
WHERE
intersection(layer.bel_river,layer.bel_prov,subplevel.linestring); |
select [Measures].[Unit Sales], [Measures].[Store Cost],
[Measures].[Store Sales]
ON columns, ([Promotion Media].[All Media], [Product].[All Prod-
ucts]) ON rows from [Sales]
```

The GIS-Query returns the provinces intersected by rivers. The OLAP section of the query uses the measures in the data warehouse in the OLAP part (Unit Sales, Store Cost, Store Sales), in order to answer the query. The dimensions are Promotion Media and Product. The hierarchy for the Store *dimension defined in the Piet-Schema is: store → city → province → Country → All. Let us suppose, for simplicity, that the GIS part of the query, returns three identifiers from 82 through 84, corresponding to the provinces of Antwerpen, Liege and Luxembourg. These identifiers correspond to their ids in the OLAP part of the model, stored in a Piet mapping table. Then, an MDX sub-expression is produced for each region, by means of traversing the different dimension levels (starting from* All *down to* province*).* □

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

4. SCALABILITY AND ERROR HANDLING

Scalability. Usually, the layers of real-world maps contain a large number of geometric objects with irregularities like holes, bays or gulfs. In this scenario, the amount of lines generated may be huge. As long as the complexity of objects increases, the number of carrier lines and the interaction between them (i.e., the intersection points), will increase accordingly, introducing several problems mainly because the carrier lines will usually go beyond the geographic area of influence of the object that generated them, producing irrelevant partitions that increase the computational cost of the algorithms presented in Section 3. For example, a line generated in Brussels is unlikely to have any relevant impact on Liege. As a consequence, we improved the naive approach, and implemented a technique, denoted "grid partition". This technique divides the map in $N \times M$ rectangles (where N and M are two integer parameters) and the original geometric objects in the different layers are allocated to these rectangles. Each rectangle is treated individually (i.e., the algorithm described above will be applied to each partition), and the carrier lines generated by the objects in each rectangle do not go beyond such rectangle. This reduces in several orders of magnitude the number of carrier lines generated, and the size of the database. In our running example we have divided the original map (and the geometries in each layer) in 200 rectangles (a grid of 10 x 20 horizontal and vertical subdivisions, respectively). Now consider, for example, Figure 1. We can see that there is a large part of the map that is within the bounding box but not in the country. Using the grid subdivision, this zone will contain no carrier lines (i.e., the grid rectangles in this region will be empty). With the naive approach, this zone of the map would have been affected, producing carrier lines outside the area of interest. Also note that the different density of carrier lines within each rectangle will be different. Therefore, less dense rectangles can be computed very efficiently. Moreover, if available, the algorithm could be run in a parallelized environment. Finally, in the particular case where a few partitions generate a number of carrier lines significantly higher than the rest, they could be further partitioned, like in the well-known divide and conquer technique. Thus, in the case that some zone of the map would change over time (v.g., the surge of new countries or provinces), we can take advantage of the rectangle sub-division, and only recompute the sub-poligonization of the affected rectangles. The former means that selecting the appropriate size of the grid is an iterative task, and that the number of rectangles depends on the complexity of the map.

Error Handling. During data precalculation, finite numeric representation problems affect the calculation of intersection points between carrier lines. Two cases arise: (a) intersection of a pair of carrier lines; (b) intersection of more than two carrier lines. Let us denote these carrierlines $L_i, i =$

$1, ..n$. In the first case, for carrier lines L_1 and L_2, it could be the case (due to the lack of robustness of the intersection algorithm provided by JTS) that P_1 (generated by intersecting L_1 and L_2,) is different (with a very small difference) from P2 (generated by intersecting L_2 and L_1). Thus, we extended the JTS library using two data structures: a vector containing the carrier lines and a list of intersection points (cuts) with other carrier lines that are generated during the process. When calculating the intersection between two lines L_1 and L_2, a point P_1 is generated and stored both in the L_1 and L_2 cut lists. Therefore, it is not necessary to calculate the intersection of L_2 with L_1, as this intersection will be already in the list of L_1, saving processing time and solving the robustness problem. In case (b), it may occur that carrier lines L_1, L_2 and L_3 intersect in points P_1, P_2 and P_3, very close to each other (but not exactly the same point, because of finite representation problems or inaccuracies in the map definition). If we use these points in the polygonization algorithm, the functions provided by JTS will fail to create sub-polygons related with those points. To solve this problem our carrier line representation checks, before adding a new cut in the cut list, if there is already a similar (very near) cut as the one it is trying to add to the list. If that is the case, the existing point is also added to the other carrier line that generates the cut. The function below verifies the similarity between a pair of points. The variable `error` must be defined by the analyst.

boolean isSimilarPoint(Point p1, Point p2,`error`) {
if $|p1.x - p2.x| <$ `error` and $|p1.y - p2.y| <$ `error` **then** result=true
otherwise result = false.

5. EXPERIMENTAL EVALUATION

In this section we present the results of our experimental evaluation of Piet over different sets of four kinds of queries. We show different kind of queries, and we ran all of them using the sub-polygonization strategy. We compare the results against other indexing techniques, and show that the common sub-polygonization appears as a competitive alternative to other well-established methods, contrary to what has been believed so far [4]. We ran our tests on a dedicated IBM 3400x 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 there was a 250Gb disk drive. We used the maps described in Section 1, i.e., five layers of a Belgium map containing information of rivers, cities, districts, provinces and regions. We defined a grid that partitions the bounding box in 20 x 10 rectangles for computing the sub-polygonization. Five kinds of experiments were performed, measuring average execution time: (a) sub-polygonization; (b) geometric queries without aggregation (GIS queries); (c) geometric aggregation queries; (d) geometric aggregation queries including a query region; (e) full GISOLAP-QL queries.

Sub-polygonization. Table 1 shows average execution times for the sub-polygonization process for the 200 rectangles. We report the average execution times for *any combination* of layers (i.e., average time for overlaying any two layers, any three layers, etc.). Table 2 shows the total execution times for *all combinations* of layers (i.e., all 2-layer combinations, all 3-layer combinations, etc.). This corresponds to a full materialization. *Note that all these processes are supposed to be run off-line.* Table 3 reports the maximum, minimum, and average number of sub-geometries in the grid squares (for the overlay of the five layers). We compared the sizes of the database before and after computing the sub-polygonization: the initial size of the database is *80 Mb.* After the precomputation of the overlay of the four layers, the database occupies *950 Mega Bytes.* Notice that this includes the 31 layer combinations with no compression of any kind, i.e., the worst possible case. We also remark that half of this size corresponds to 6 overlay combinations. Thus, a partial materialization strategy (along the lines of [5]) can be applied, if needed, with limited cost in performance.

Number of Layers	Average Execution Time
5	5 hours 48 minutes 25.1910 seconds
4	1 hour 55 minutes 20.6170 seconds
3	0 hour 43 minutes 54.9380 seconds
2	0 hour 13 minutes 55.0870 seconds

Table 1: Average sub-polygonization times

Number of Layers	Total Execution Time
5	5 hours 48 minutes 25.1910 seconds
4	9 hours 36 minutes 43.0850 seconds
3	7 hours 19 minutes 9.37500 seconds
2	2 hours 19 minutes 10.8690 seconds

Table 2: Total sub-polygonization times

Pure Geometric Queries. 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). The queries were: (a) Q1: *Districts crossed by at least one river*; (b) Q2: *Districts and the cities within them*; (c) Q3: *Districts and the cities within them only for districts crossed by at least one river*; (d) Q4: *Districts crossed by al least five rivers.* 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). All the layers are indexed. Finally, we ran the postGIS queries without indexing. PostGIS queries have been optimized analyzing the generated query plans. All Piet tables have been indexed over attributes that participate in a join or in a selection. In all cases, queries were executed without the overhead of the graphic interface. All the queries were ran 10 times, and we report the average execution time.

Figure 5 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 range from approximately ten times (for Q4) to ninety times (for Q2), in favor of Piet. The sizes of the query results are 40, 583, 562, and 5 tuples, for Q1 through Q4, respectively. We can see that query Q1 only needs to find districts that have sub-geometries in common (linestrings) with rivers; thus, only one pre-computed table, which contains sub-geometries shared by this two layers, needs to be queried. A similar situation occurs in the case of Q2. Analogously, Q3 only needs to know about districts that have sub-geometries in common (points) with cities and which contains sub-geometries in common (linestrings) with rivers. Therefore, it only queries two pre-computed tables. Q4 behaves in a similar way.

Subgeometry	Max	Min	Average
# of Carrier Lines per rectangle	99	4	28
# of Points per rectangle (carrier lines intersection within a rectangle)	2617	4	361
# of Segment Lines per rectangle (segment of carrier lines within a rectangle)	5136	4	694
# of Polygons per rectangle	2518	1	335

Table 3: Number of sub-geometries in the grid.

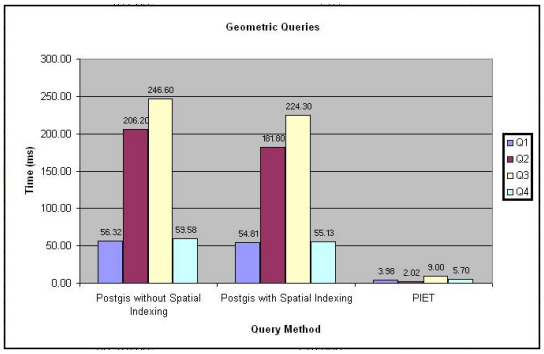

Figure 5: Execution time for geometric queries.

Geometric Aggregation Queries. 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. These queries are: (a) Q5: *List each region with the total number of rivers that crossed it*; (b) Q6: *List each region with the total number of rivers that crossed it, only for regions that contains at least 20 cities*; (c) Q7: *List each district with the total number of rivers that crossed it and the total number of cities that it contains*; (d) Q8: *For each region show the total length of the part of rivers which intersects it, only for regions with at least an area under cereal cultivation equal or higher than 1000 Km^2.* Note that query Q8 does not only require the pre-overlay table which binds together the layers involved, but also the sub-pologonization table. Figure 6 shows the results. We can see that, again, Piet clearly outperforms postGIS with or without R-tree indexing. The differences range from approximately five times (for Q8) to ninety times (for Q6), in favor of Piet.

Geometric Aggregation Queries including a query region. For the experiment (d), we ran the following three queries, and added two different query regions (shown in Figure 1). The results are depicted in Figures 7 and 8. We denote query regions #1 and #2 the large and small regions in Figure 1, respectively. The queries are: (a) *Q9: List each region with the total number of rivers that crossed it, considering only the part of the river that lies within the query region*; (b) Q10: *For each district show the total number of cities, for cities within the query region*; Q11: *For each region show the total length of the part of each river which intersects it, for regions with at least an area under cereal cultivation $\geq 1000Km^2$, considering only the part of the river that lies within the query region.*

Figures 7 and 8 show that, when a query is restricted to a region Piet still performs better than R-tree spatial indexing, although the differences are not significant. Note that, except for $Q10$, the difference in performance increases for

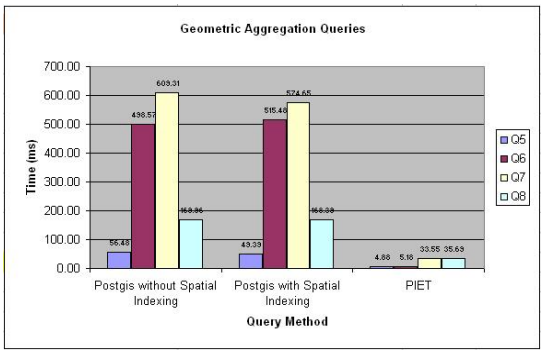

Figure 6: Execution time for geometric aggregation.

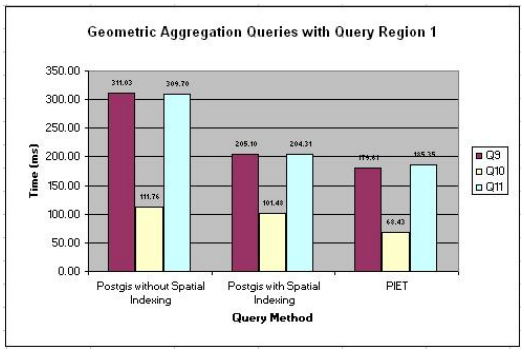

Figure 7: Geometric aggregation - query region 1.

smaller query regions. Obviously, here Piet is affected by the on-the-fly computation of the intersection between the query region and the precomputed overlay. An optimization algorithm was used in Piet: only the rectangles in the grid affected by the query region were considered for computing the intersection (indexing the latter with an R-Tree did not yield a significant improvement). As the region turns smaller, there is a lower number of tuples involved in the computation, and the performance of Piet turns better.

Aggregation R-Trees. We implemented the aggregation R-tree (aR-tree) [11], an R-tree variation that stores not only the MBRs of different geometries but also the value of some aggregation function for all objects that are enclosed by the MBR. We ran geometric aggregation queries, with or without a query region. We report the results obtained running geometric aggregation queries: (a) Q12: *Maximum area under cereal cultivation, only for regions crossed by rivers*, and (b) Q13: *Maximum area under cereal cultivation, for regions crossed by rivers*(using query region #1). Table 4 shows the results. We can see that Piet still performs better that PostGIS using R-tree, and also better than aR-tree.

GISOLAP-QL Queries. We ran GISOLAP-QL queries, that integrate GIS and OLAP in a very simple way. First the system computes the identifiers of the geometries that verify the geometric queries (i.e., the part before the '|'), and then passes this information on to the MDX expression, which is then merged with the geometric information, producing the final MDX query. Times for computing the SQL-like part were similar to the already reported ones. Time of generating the complete MDX expression is negligible.

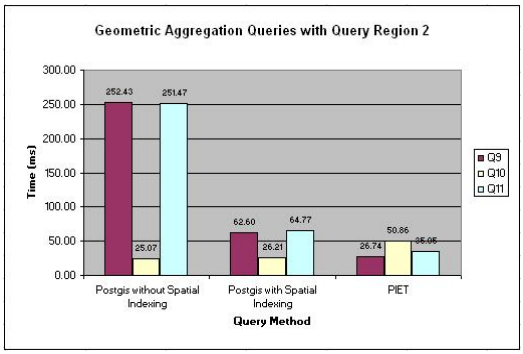

Figure 8: Geometric aggregation - query region 2.

Query	PostGIS with spatial indexing (ms)	aR-tree (ms)	Piet (ms)
Q12	47.997	45.25	14.669
Q13	474.35	470.02	345.26

Table 4: Piet vs. aR-tree and R-tree.

Discussion. We can conclude that (a) Piet clearly outperforms R-Trees for queries performed over an entire map (i.e., that do not require on-the fly intersection between a query region and the sub-polygonization), either for queries involving aggregation or not; (b) When a query region is present, indexing methods and overlay precomputation deliver similar performance; (c) As a general rule, the performance of overlay precomputation improves as the query region turns smaller. (d) Piet always delivered execution times compatible with user needs; (e) The cost of integrating GIS results and OLAP navigation capabilities through the GISOLAP-QL query language is negligible; (f) The main benefit of aR-trees come from pruning tree traversal when a region is completely included in a MBR. 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. Otherwise, the aR-tree must reach the leaves, as standard R-trees do. In spite of this, for very large maps and large query regions, aR-trees have the potential to outperform the other techniques. (g) The class of geometric queries that clearly benefits from overlay precomputation can be easily identified by a query processor, and added to any existing GIS system straightforwardly.

6. FUTURE WORK

The implementation presented in this paper provides an efficient and smooth integration between OLAP and spatial data, not present in commercial GIS systems. In addition, our experiments showed that there is an important class of geometric queries (aggregate or not) that can benefit from precomputing the overlay of the thematic layers in a GIS.

We are currently working to augment Piet with spatio-temporal capabilities, specifically in the field of moving object data, that can be naturally added to this framework.

Acknowledgements This work has been partially funded by the European Union under the FP6-IST-FET programme, Project n. FP6-14915, the Research Foundation Flanders (FWO-Vlaanderen), ProjectG.0344.05., and the Scientific Agency of Argentina, Project PICT n. 21350.

7. REFERENCES

[1] A. Gutman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD'84*, pages 47–57, 1984.

[2] S. Haesevoets, B. Kuijpers, and A. Vaisman. Spatial aggregation: Data model and implementation. In *Submitted for revew*, 2007.

[3] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Proceedings of PAKDD'98*, pages 144–158, 1998.

[4] J. Han and M. Kamber. *Data Mining, Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[5] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of SIGMOD'96*, pages 205 – 216, Montreal, Canada, 1996.

[6] C. Hurtado, A.O. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. In *Proceedings of IEEE/ICDE'99*, pages 346–355, 1999.

[7] C.S. Jensen, A. Kligys, T.B Pedersen, and I. Timko. Multidimensional data modeling for location-based services. *VLDB Journal 13(1)*, pages 1–21, 2004.

[8] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd. Ed.* J.Wiley and Sons, Inc, 2002.

[9] B. Kuijpers and Alejandro Vaisman. A data model for moving objects supporting aggregation. In *Proceedings of STDM'07*, Istambul, Turkey, 2007.

[10] G. Kuper and M. Scholl. Geographic information systems. In J. Paredaens, G. Kuper, and L. Libkin, editors, *Constraint databases*, chapter 12, pages 175–198. Springer-Verlag, 2000.

[11] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proceedings of SSTD'01*, pages 443 – 459, 2001.

[12] J. Paredaens, G. Kuper, and L. Libkin, editors. *Constraint databases*. Springer-Verlag, 2000.

[13] T.B Pedersen and N. Tryfona. Pre-aggregation in spatial data warehouses. *Proceedings of SSTD'01*, pages 460–480, 2001.

[14] F. Rao, L. Zang, X. Yu, Y. Li, and Y. Chen. Spatial hierarchy and OLAP-favored search in spatial data warehouse. In *Proceedings of DOLAP'03*, pages 48–55, Louisiana, USA, 2003.

[15] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases*. Morgan Kaufmann, 2002.

[16] S. Rivest, Y. Bédard, and P. Marchand. Modeling multidimensional spatio-temporal data warehouses in a context of evolving specifications. *Geomatica, 55 (4)*, 2001.

[17] G. Shilov, B. Gurevich. *Integral, Measure, and Derivative: A Unified Approach*. Richard A. Silverman, trans. Dover Publications, 1978.

[18] I. Vega López, R. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Transactions on Knowledge and Data Engineering 17(2)*, 2005.

[19] M. F. Worboys. *GIS: A Computing Perspective*. Taylor&Francis, 1995.