# Piet: a GIS-OLAP Implementation

**Ariel Escribano**
Universidad de Buenos Aires
aescribano@dc.uba.ar

**Leticia I. Gomez**
Instituto Tecnólogico de de Buenos Aires
lgómez@itba.edu.ar

**Bart Kuijpers**
Limburgs Universitair Centrum
bart.kuijpers@luc.ac.be

**Alejandro A. Vaisman**
Universidad de Buenos Aires
av2n@dc.uba.ar

## ABSTRACT

Data aggregation in Geographic Information Systems (GIS) is a desirable feature, although only marginally present in commercial systems nowadays, mostly through ad-hoc solutions. Integration between GIS and OLAP systems is still needed in commercial systems. With this in mind, we have developed Piet, a system that that makes use of a novel query processing technique: first, a process called *subpolygonization* decomposes each thematic layer into open convex polygons; then, another process computes and stores in a database the overlay of those layers for later use by the query processor. We describe in detail the implementation of Piet, and provide evidence, through experimentation over real-world maps, that *overlay precomputation* for spatial aggregate queries can be competitive with GIS systems that employ indexing schemes based on R-trees. Given that the data model and implementation do not prevent the use of traditional indexing techniques as R-tree and aR-trees, we our proposal against these techniques in our experiments.

## 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 [16]. In GIS, geometric objects within a map are organized in *thematic layers* (e.g., provinces in one layer, rivers in another one). These spatial objects may also be annotated with some numerical o categorical complementary information. Typical queries in GIS ask for geometric objects that satisfy some condition, or, even 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], or novel techniques like aR-trees [12]. Although 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) [7] comprises a set of tools and algorithms for querying multidimensional databases containing large amounts of data, usually called data warehouses. In OLAP, data is usually perceived as a *data cube.* Each cell in this data cube contains a measure or set of measures representing facts and the contextual information which conform *dimensions,* organized, typically in hierarchies. OLAP tools allow us to aggregate the measures along the dimensions. Efficient evaluation if OLAP queries requires, more often than not, the use of precalculation techniques [4].

Our proposal is aimed at integrating these two different worlds in a single framework [2, 9], allowing, for instance, to evaluate queries like "total income in provinces crossed by at least one river", where income information is stored in the data warehouse and provinces and rivers information is stored in a GIS. Moreover, the results will be navigated in the usual OLAP way, through operations like roll-up and drill-down. In this paper, after providing a quick overview of the data model(Section 2), we describe the implementation of a system, denoted *Piet* (after the Dutch painter Piet Mondrian), that accomplishes the goals mentioned above (Section 3). This system integrates open source GIS and OLAP technologies, and overlay precomputation. For the latter, we introduce a novel technique denoted *common subpolygonization* which decomposes each thematic layer into open convex polygons. We also discuss the key topics of *scalability* and *error management* (Section 4), and report experimental results over real-world maps (Section 5 showing that overlay precomputation can compete with other 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, districts and cities. Additionally, the maps contain demo-


*DOLAP* '2007 Lisboa, Portugal

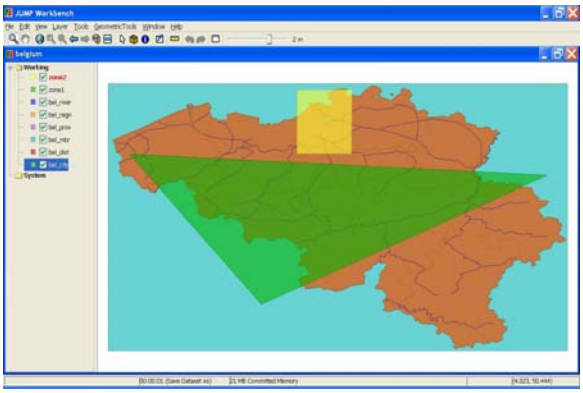

**Figure 1: Running example: a map of Belgium**

graphic 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 from the spatial library of the GIS Center [1]. There is also a data warehouse with information about stores and sales for different regions of Belgium (this information has been generated for the experiments, and is not actual information).

## 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.* [17] 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 [14] 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" are not supported. Also, no experimental results are provided . Extending this model, Jensen *et al.* [6] proposed a multidimensional data model for mobile services. With a different approach, Rao *et al* [15] 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, 12] 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.

[1] 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 model where GIS, OLAP, and moving objects 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 the schema of a GIS dimension: the bottom level of each hierarchy, denoted the *Algebraic part* of the dimension, contains the infinite points in a layer, and could be described by means of linear algebraic equalities and inequalities [13]. 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 $L$, 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 as 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* of the dimension for storing non-spatial data. This part contains the conventional OLAP structures, as defined in [5]. 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 a line identifier in the geometric part, and that in the same portion of the dimension, this line aggregates on a polyline identifier. In the OLAP part we have two dimensions, representing districts and rivers, associated to the corresponding graphs, as*

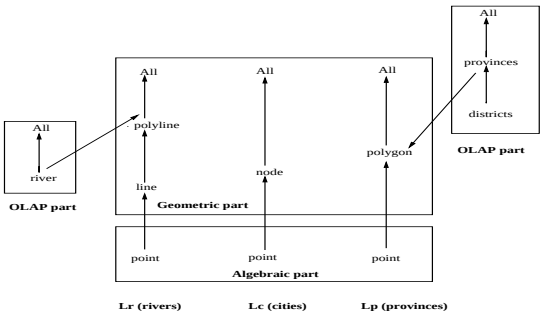


**Figure 2: An example of a *GIS* dimension Schema**

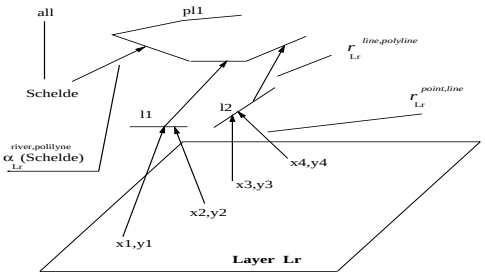


**Figure 3: A *GIS* dimension instance for Figure 2.**

*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 $\alpha$ 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 $(state, Year, Population)$.

*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 satisfies a condition. For example, the query "total population of the cities crossed by the river "Schelde" reads:

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

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

The meaning of the query is the following: $\alpha^{\mathrm{Ri}\to\mathrm{Pl}}_{L_r,Rivers}$('schelde') maps the identifier of the Schelde river to a polyline in layer $L_r$ (representing rivers). The relation $r^{\mathrm{Pt}\to\mathrm{Pl}}_{L_r}(x, y, L_r, pl_1)$ 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 geometries that satisfy both conditions can be retrieved, and the function $ft_{pop}$ is applied to them.

*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, an 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). Therefore, Piet implements a different strategy, consisting in three steps: (a) partitioning each layer in subgeoemetries, according to the carrier lines defined by these geometries (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 that this strategy can be an efficient alternative for evaluating queries of this kind. In Section 5 we compare overlay pre-computation against other well-known indexing techniques like R-Trees [1], and aR-Trees [11, 12].

To conclude this section, we will give some definitions. We will work within a bounding box B × 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 $C_{\mathrm{pl}}$ 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 $C_{\mathrm{pg}}$ 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 $C_{\mathrm{p}}$ of a point p *consists of the horizontal and the vertical lines intersecting in the point. The* carrier set $C_{\mathrm{L}}$ of a layer L *is the union of the carrier sets of the points, polylines and polygons appearing in the layer.* □

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 roullup relations $r$ will turn into functions (given that no two points can map to the same open convex polygon). Given $C_{\mathrm{L}}$, the carrier set of a layer L, and a bounding box, the set of open convex polygons, open line segments and points, induced by $C_{\mathrm{L}}$, that are strictly

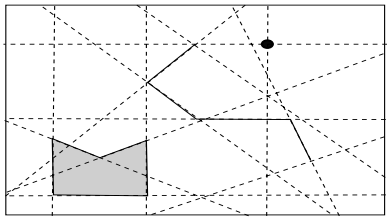

**Figure 4: The carrier sets of a point, a polyline and a polygon are the dotted lines.**

inside the bounding box, is called the *convex polygonization of* L. As we are interested in queries involving multiple overlapped layers, we need to be capable of compute the *common sub-polygonization* operation, that further subdivides the bounding box according to the carrier sets of the layers involved. Given two layers $L_1$ 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, of course, be generalized for more than two layers.

## 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 preformed using a data loader component. The second module (denoted *precalculated data generator*) allows the storage and execution of precomputed data. Its main component is a so-called "data processor" that processes raw data and generates (off-line) the following information: (a) Precomputed data: containing the subgeometries generated in the subpoligonization step, fact values associated to those subgeometries (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 and the maps, and their association; (c) GIS-OLAP relations: containing the information needed to associate geometric components and data warehouse information (v.g., a point in the map can be related with an Store concept 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[2] with Postgis 1.2 spatial extensions[3]. 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[4]. For OLAP

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

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

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

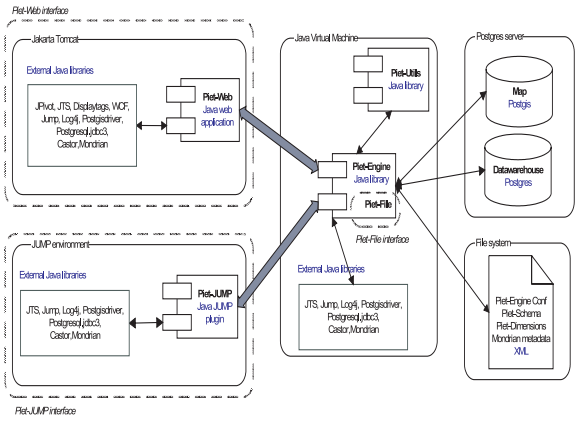


**Figure 5: The Piet Components.**

queries we used Mondrian[5] and the MDX query language, an industry OLAP standard[6]. This technological architecture is shown in Figure 5. We will explain some of the components in this figure as we progress in the paper.

Our Piet implementation consists of two main modules: *Piet-Jump,* which provides a Graphical User Interface (GUI) for displaying maps, and *Piet-Web,* which allows running queries over the framework. Figure 5 shows a component called PIET-Schema, which consists in a set of metadata definitions, used by different modules of the system. The definitions include: the storage location of the geometric components and their associated measures, the subgeometries corresponding to the subpolygonization 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. PIET uses this information to answer the GISOLAP-QL queries described below. Metadata are stored in XML documents containing three kinds of tags: `Subpoligonization`, `Layer`, and `Measure`. An example of a `Subpoligonization`. We omit the description of these documents due to space limitations.

*Subpolygonization.* As we explained above, the common subpoligonization 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* is used to generate the convex subpolygons 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. As a remark, this subpolygonization turns the rollup *relations* into rollup *functions*, given that the process produces *open convex polygons.*

*Overlay Precomputation.* The original geometries in different layers overlap if they have in common one of the fol-

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

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

lowing: points, sub-lines of the carrierset (generated by the intersection of the carrier lines), or subpolygons (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), and its storage in the `Preoverlay` table, for its during query processing:

listLayers = determineListOfLayersInvolvedInTheQuery()
geoComponents = determineListOfGeometries(listLayers)
carrierlines = generateCarrierLines(geoComponents)
The next step generates points, subsegments and polygons in which the original *geoComponents* are divided due the overlay. While the subgeometries are being produced, 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 `Subpoligonization`.
subgeometries = generateSubgeometriesAndPropagateFacts(carrierlines)
The next step computes the original geometries that have subgeometries in common, and stores their identifiers in a table called `Preoverlay`. This table also stores the identifiers of the common subgeometries.
calculatePreoverlay(subgeometries)

Note that pure geometric queries (like "total of regions crossed by rivers"), and geometric queries that do not require fact aggregation (e.g.,"*total number of regions crossed by rivers*") can be answered using just the `Subpoligonization` table (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 Belgium, in regions crossed by rivers*". In this case, the `Preoverlay` table is used first to find the common subgeometries for the layers containing regions and rivers; after this, table `Subpoligonization` is used to find the values of the proportional facts previously computed. Finally, for aggregation queries in the OLAP environment with geometric constraints, the `Preoverlay` table is used to find the geometry identifiers of the original layers; this value is later used to access the layer table and the corresponding OLAP dimension values.

***Query Language.*** We implemented a query language that combines, in a single expression, queries about geometric and OLAP content, without losing the ability to express pure GIS or OLAP queries. Queries can be submitted in two ways: using the Piet-Jump interface, or through a query language, denoted GISOLAP-QL, that we briefly describe below. 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` *geometric operations*;

This 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 `SELECT` clause contains a list of layers and/or measures, which must be defined in the corresponding PIET-Schema of the `FROM` clause. The `WHERE` clause in the GIS-Query part, consists in conjunctions and/or disjunctions of geometric operations applied over a the elements of the layers involved. The expression also includes the kind of subgeometry used to perform the operation. The syntax for an operation is of the form *operation name(list of layer members, subgeometry)*. The accepted values for *subgeometry* are "Point", "LineString" and "Polygon"[7].

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 Products])
ON rows from [Sales] where [Time].[2005]
```

*The GIS-Query returns the provinces of the GIS dimension 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 return the requested information. 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 the ids in the OLAP part of the model, stored in a PIET mapping table. Then an MDX sub-expression is produced for each region, traversing the different dimension levels (starting from* All *down to* province*).* □

# 4. SCALABILITY AND ERROR HANDLING

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 subpolygonization instead of dividing the map into arbitrary rectangles in order to approximate the original shape of a geometry (an idea similar to Riemann Integral Approximation), increases the number of subgeometries (i.e., rectangles) required to minimize errors. In this section we describe how we addressed in our implementation, two key issues: scalability and error management.

***Scalability.*** Usually, the layers of real-world maps contain a large number of geometric objects, and their geometries contain 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. Thus, the points that they generate produce 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

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

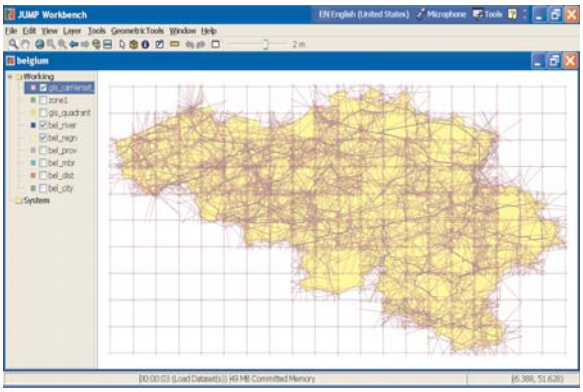


**Figure 6: Running example: carrierlines in each partition generated by regions and rivers layers.**

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); the original geometries in the different layers are also 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 technique reduces in several orders of magnitude the number of carrier lines generated. Figure 6 shows, for our running example, the result of dividing the original map (and the geometries in each layer) in 20x10 rectangles. We can see the different density of carrier lines in each rectangle. Also note that regions within the bounding box, but not in the country, contain no lines. With the naive approach, these regions would have been affected, producing carrier lines in zones that are outside the area of interest. 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 subpoligonization of the affected rectangles.

*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 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 $L_1$). This problem arises because of the lack of robustness of the intersection algorithm provided by JTS (see Figure 5). To solve this problem we extended the JTS library, and created a carrier line representation, which stores the carrier line vector and a list of intersection points (cuts) with other carrier lines generated using the current map. When calculating the intersection between $L_1$ 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 the second case, 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 points, because of finite representation problems or inaccuracies in the map definition). If we use these in the polygonization algorithm using the functions provided by JTS, the algorithm will fail to create subpolygons related with those points, due to robustness problems. 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. To verify the similarity between a pair of points we use an error margin, as shown below.

```
boolean isSimilarPoint(Point p1, Point p2) {
return result = ((-1.0) * ERROR<p1.getX() - p2.getX() && p1.getX() -
p2.getX()<ERROR
&& (-1.0) * ERROR<p1.getY() - p2.getY() && p1.getY() - p2.getY()<ERROR);
}
```

The "ERROR" variable represents the error margin used to consider a pair of points similar. A standard value for this variable should be in the order of $10^{-12}$.

Finally, to accelerate the search of similar points in the cuts list and the subsequent step in the subpolygonization process (sub-lines generation), the points are stored ordered according to their distance to the coordinate origin. With this idea, the search of similar cuts finishes as soon as the distance of the searched point in the cuts list is exceeded. Additionally, keeping the points ordered facilitates the generation of sub-lines. In Section 5 we will give additional remarks to the error management topic.

## 5. EXPERIMENTAL EVALUATION

In this section we present the results of our experimental evaluation of Piet over different sets of four kinds of queries. The main goal of these experiments is to determine under which conditions one strategy behaves better that 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 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 running example 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 subpolygonization. Five kinds of experiments were performed, measuring average execution time: (a) 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.

*Subpolygonization.* Table I shows the execution times for the subpolygonization process for the 200 rectangles, from the generation of carrier lines to the generation of the precomputed overlayed layers. Table II reports the maximum, minimum, and average number of subgeometries in the grid squares (for the overlay of the five layers). Finally, we compared the sizes of the database before and after computing

| | |
|---|---|
| Generation of CarrierLines | 26 minutes 46.4420 seconds |
| Generation of Points | 36 minutes 7.41700 seconds |
| Generation of Segment Lines | 25 minutes 41.8650 seconds |
| Generation of Polygons | 2 hours 59 minutes 15.8900 seconds |
| Polygon Associations | 7 minutes 52.3340 seconds |
| Line Associations | 18 minutes 39.7870 seconds |
| Point Associations | 33 minutes 8.94200 seconds |
| Generation of Preoverlays | 2.03600 seconds |

**Table 1: Execution times for the subpolygonization.**

| 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 2: Number of subgeometries in the grid.**

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 Mega Bytes.*

*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 [8] - Generalized index search tree). All the layers are indexed. Finally, we ran the postGIS queries without indexing for the postGIS queries. 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. We do not show the actual query expressions for the sake of space. Figure 7 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 ninty 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 subgeometries in common (linestrings) with rivers; thus, only the pre-computed table gis_pre_linestring_26, which contains subgeometries 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 subgeometries in common (points) with cities and which contains subgeometries in common (linestrings) with rivers. Therefore, it only queries the pre-computed tables gis_pre_point_19 and gis_pre_linestring_26. Q4 behaves in a similar way.

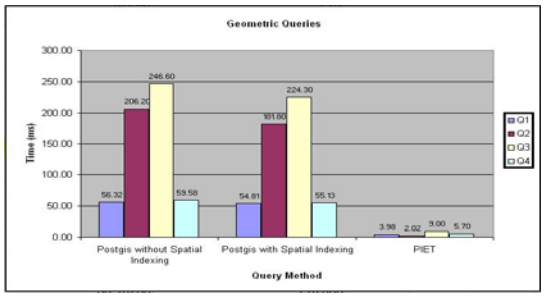


**Figure 7: Execution time for geometric queries.**

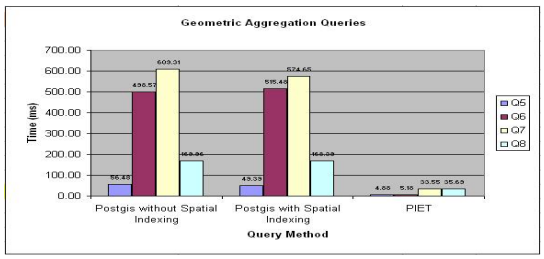


**Figure 8: Execution time for geometric aggregation.**

*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 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.* Note that query Q8 does not only require the pre-overlay table which binds together the layers involved, but also the subpologonization table. Figure 8 shows the results. We can see that Piet clearly outperforms postGIS with or without R-tree indexing. The differences range from approximately five times (for Q8) to ninty times (for Q6), in favor of Piet. The sizes of the query results are 3, 2, 40, and 2 tuples, for Q5 through Q8, respectively.

*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 9 and 10. 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, only for regions containing at least area under cereal cultivation equal or higher than 1000, considering only the part of the river that lies within the query region.*

When a query is restricted to a region, like it is the case of queries Q9 to Q11, Piet performance is still similar than

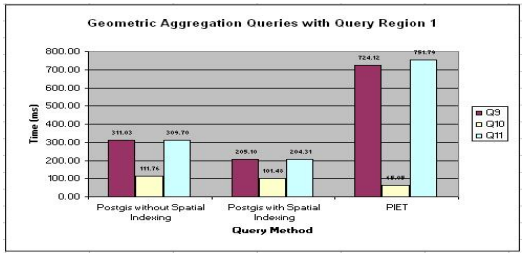


**Figure 9: Geometric aggregation - query region 1.**

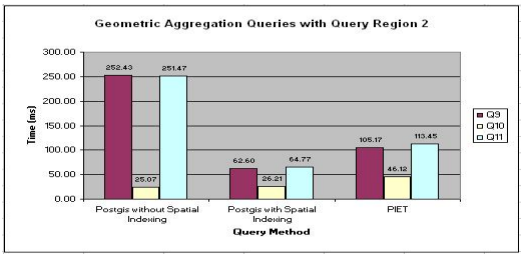


**Figure 10: Geometric aggregation - query region 2.**

the other two methods, although in this case,the latter performs slightly better than Piet. Obviously, here the problem is the on-the-fly computation of the intersection between the query region and the precomputed overlay. Indexing the latter with an R-Tree does not yield significant improvements.

*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, only for regions crossed by rivers and for regions within a query region in Wallonne.* Table 5 shows the results.

*Precision in Piet.* In the special case of queries using query regions, as the query region is not affected by the subpoligonization process, the subgeometries shared by the layers may not lie completely within the query region. Thus, the ones that that are within the query region are discarded, affecting the results. To illustrate the situation we show in Figure 11, a zoomed portion of the subgeometries of rivers that do not lie inside the query region and which are not been considered in the answer to the query *list each district with its length of rivers, considering only parts of rivers within a query region.* This problem can be fixed in Piet at the cost of computing the exact length (inside the query region) of the segments that are intersected by the region boundaries, as other indexing techniques, like R-Tree variation do.

*GISOLAP-QL Queries.* For the experiment (e), we ran GISOLAP-QL queries (Section 3), that let us express integrated GIS and OLAP queries in a very simple way. We ran SQL queries and full GISOLAP-QL queries. First the system computes the identifiers of the geometries that verify the geometric queries (i.e., the part before the '—'), and

| Query | PostGIS with spatial indexing (ms) | aR-tree (ms) | Piet (ms) |
|---|---|---|---|
| Q12 | 47.997 | 5 | 14.669 |
| Q13 | 39.541 | 4.2 | 2955.060 |

**Table 3: Piet vs. *aRtree*.**

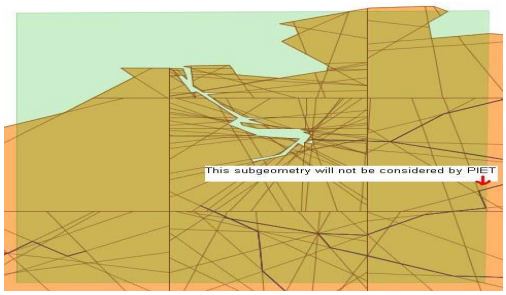


**Figure 11: Precision in Piet.**

then pass this information on to the MDX expression (the part after the '—'). The MDX expression is then merged with the geometric information, producing the final MDX query. The times for computing the SQL-like part were similar to the ones already reported, and the time for generating the complete MDX expression is negligible. Due to space limitations, we limit ourselves to show the result for the query Q14: "*Unit Sales, Store Cost and Store Sales for the products and promotion media offered by stores only in provinces crossed by rivers*". Figure 12 shows the result for Q14, which includes the three dimensions: Store (obtained through the geometric query), Promotion Media, and Product. A Piet user can navigate this result (drilling-down or rolling-up along the dimensions). Figure 13 shows an example of a drill-down operation.

## 6. FUTURE WORK

Our implementation showed that integrating OLAP and spatial data can be efficiently performed using the overlay precomputation techniques. We are currently working to provide Piet with spatio-temporal capabilities, specifically in the field of moving object data, that can be naturally added to this framework.

## 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*, 2006.

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

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

[6] 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.

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

| | | | Measures | | |
|---|---|---|---|---|---|
| Store | Promotion Media | Product | Unit Sales | Store Cost | Store Sales |
| Topeka | All Media | All Products | | | |
| Wichita | All Media | All Products | | | |
| Lincoln | All Media | All Products | 67,659 | 56,772.50 | 142,277.07 |
| Salem | All Media | All Products | | | |

**Figure 12: Result for the full GISOLAP-QL query.**

| | | | Measures | | |
|---|---|---|---|---|---|
| Store | Promotion Media | Product | Unit Sales | Store Cost | Store Sales |
| Topeka | All Media | All Products | | | |
| Wichita | All Media | All Products | | | |
| -Lincoln | All Media | All Products | 67,659 | 56,772.50 | 142,277.07 |
| Store 11 | -All Media | All Products | 26,079 | 21,948.94 | 55,058.79 |
| | Bulk Mail | All Products | | | |
| | Cash Register Handout | All Products | 1,695 | 1,476.69 | 3,699.69 |
| | Daily Paper | All Products | | | |
| | Daily Paper, Radio | All Products | 1,446 | 1,225.38 | 3,058.22 |
| | Daily Paper, Radio, TV | All Products | 400 | 340.58 | 838.67 |
| | In-Store Coupon | All Products | 385 | 342.59 | 852.65 |
| | No Media | All Products | 17,709 | 14,838.85 | 37,212.87 |
| | Product Attachment | All Products | 2,160 | 1,779.88 | 4,514.04 |
| | Radio | All Products | | | |
| | Street Handout | All Products | | | |
| | Sunday Paper | All Products | 417 | 357.56 | 892.44 |
| | Sunday Paper, Radio | All Products | 1,011 | 841.73 | 2,130.39 |
| | Sunday Paper, Radio, TV | All Products | | | |
| | TV | All Products | 856 | 745.69 | 1,859.82 |
| Store 13 | All Media | All Products | 41,580 | 34,823.56 | 87,218.28 |

**Figure 13: Drilling down the result of Figure 12.**


[8] M. Kornacker. Access methods for next-generation database systems. *Ph.D Thesis, UC at Berkeley*, 2000.

[9] B. Kuijpers and Alejandro Vaisman. A data model for moving objects supporting aggregation. In *Proceedings of the First International Workshop on Spatio-Temporal Data Mining (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] D. Papadias, Y. Taoy, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *Proceedings of ICDE'02*, pages 166–175, 2002.

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

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

[15] 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.

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

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

[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.