Physical Design and Implementation of Spatial Data Warehouses Supporting Continuous Fields

Leticia Gómez[1], Alejandro Vaisman[2], and Esteban Zimányi[3]

[1] Instituto Tecnológico de Buenos Aires
lgomez@itba.edu.ar
[2] Universidad de Buenos Aires
avaisman@dc.uba.ar
[3] Université Libre de Bruxelles
ezimanyi@ulb.ac.be

Abstract. Although many proposals exist for extending Geographic Information Systems (GIS) with OLAP and data warehousing capabilities (a topic denoted SOLAP), only recently the importance of supporting continuous fields (i.e., phenomena that are perceived as having a value at each point in space and/or time) has been acknowledged. Examples of such phenomena include temperature, altitude, or land use. In this paper we discuss physical design issues arising when a spatial data warehouse includes a combination of spatial and non-spatial dimensions and measures, and spatio-temporal dimensions representing continuous fields. We give the syntax and semantics of the data types (and their operators) needed to support fields in SOLAP environments, and present an implementation of these types, on top of spatial-SQL. We also show how queries using the spatio-temporal operators for fields are written, parsed, and executed.

1 Introduction

In the last few years, efforts have been carried out to integrate Geographic Information Systems (GIS) [1] and OLAP (On-Line Analytical Processing) [2]. This integration, called SOLAP (standing for Spatial OLAP), aims at exploring spatial data by drilling on maps, in the same way as OLAP operates over tables and charts. This concept was introduced by Rivest *et al.* [3], who also describe the desirable features and operators a SOLAP system should have. A survey on the topic can be found in [4]. The need for sophisticated GIS-based decision support systems, for the analysis of organizational data with respect to geographic information, is encouraging OLAP and GIS vendors to integrate their products.

Advances in data analysis technologies raise new challenges. One of them is the need to handle *continuous fields*, which describe physical phenomena that change continuously in time and/or space. Examples of such phenomena are temperature, pressure, and land elevation. Besides physical geography, continuous fields (from now on, fields), like land use and population density, are used in human geography as an aid in spatial decision-making process. Formally, a field

T.B. Pedersen, M.K. Mohania, and A M. Tjoa (Eds.): DaWaK 2010, LNCS 6263, pp. 25–39, 2010.

is defined as composed of [5]: (a) a domain $\mathcal{D}$ which is a continuous set; (b) a range of values $\mathcal{R}$; and (c) a mapping function f from $\mathcal{D}$ to $\mathcal{R}$.

Although some work has been done to support querying fields in GIS, spatial multidimensional analysis of continuous data is still in its infancy. Existing multidimensional models dealing with discrete data are not adequate for the analysis of continuous phenomena. Multidimensional models and associated query languages are thus needed, to support continuous data. Recently, Vaisman and Zimányi [6] presented a conceptual model for SOLAP that supports dimensions and measures representing continuous fields, and characterized multidimensional queries over fields. They defined a *field* data type, a set of associated operations, and a multidimensional calculus supporting this data type. In this paper we go a step further, and study the translation of this conceptual data model to physical structures based on the well-known star-schema [2]. We also introduce two new data types, field and tempfield, define a semantics for the operators associated to these types, and present an implementation for them. Finally, we define an SQL-like query language over the physical structures and operators mentioned above, and provide a preliminary implementation of the language.

This paper is organized as follows. Section 2 provides an overview of related work dealing with fields. Section 3 presents the conceptual model, and introduces the field data type and its associated operators. In Section 4 we discuss the physical warehouse design to implement the conceptual model and we introduce the SQL-like language to support fields. Section 5 presents the operators of the field data type, whose implementation is shown in Section 7. Section 6 presents the query language, and Section 8 sketches how a query in this language is implemented. We conclude in Section 9.

2 Related Work

In his pioneering work on defining *algebra for fields*, Tomlin [7] proposed a so-called map algebra, based on the notion that a map is used to represent a continuous variable (e.g., temperature). There are three types of functions in Map algebra: *local*, *focal*, and *zonal*. Local functions compute a value at a certain location as a function of the value(s) at this location in other map layer(s). Focal functions compute each location's value as a function of existing values in the neighboring locations of existing layers (i.e., they are characterized by the topological predicate *touches*). Zonal functions (characterized by the topological predicate *inside*), compute a location's new value from one layer (containing the values for a variable), associated to the zone (in another map) containing the location. Câmara *et al.* [8] and Cordeiro *et al.* [9] formalized and extended these functions, supporting more topological predicates. We base our proposal on this work, and on the proposal of Mennis *et al.* [10], where map algebra operators are extended to query time-varying fields. The model and query language we present here cover those proposals, and extend them to the multidimensional setting.

Paolino *et al.* [5] introduced *Phenomena*, a visual language for querying continuous fields, based on a conceptual model where users view the world as consisting of both continuous fields and discrete objects, and are able to manipulate

them in a uniform manner. Phenomena uses an extension to Spatial SQL that supports continuous fields, proposed by Laurini *et al.* [11]. GeoRaster[1] is a feature of Oracle Spatial that allows storing, indexing, querying, analyzing, and delivering raster data, and its associated metadata. GeoRaster provides specialized data types and associated operators, as well as an object relational schema, which can be used to store and manipulate multidimensional raster layers. None of these tools and languages were devised for a SOLAP setting.

Regarding *fields and multidimensional models*, the joint contribution of the GIS and OLAP communities to this problem has been limited. Shanmugasundaram *et al.* [12] proposed a data cube representation that deals with continuous dimensions. This works focuses on using the known data density to calculate aggregate queries without accessing the data. The representation reduces the storage requirements, but continuity is addressed in a limited way. Ahmed *et al.* use interpolation methods to estimate (continuous) values for dimension levels and measures, based on existing sample data values [13]. Continuous cube cells are computed on-the-fly, producing a continuous representation of the discrete cube. These proposals are based on a data model devised for OLAP, not for *spatial* OLAP, which goes against a comprehensive representation of spatial dimensions and measures. Opposite to this, our approach is based on a conceptual multidimensional model designed with spatial data in mind. Thus, continuous fields are introduced as a natural extension to this model. In order to support fields, Vaisman and Zimányi presented a conceptual model for spatio-temporal OLAP supporting fields, and a calculus to query such data. In this paper, we build on that work to propose a user-friendly SQL-like version of the calculus making use of two new data types that support spatio-temporal fields.

3 Preliminaries

We now briefly describe the conceptual model proposed in [6], extending the *MultiDim* model [14] to support fields. For this we use the example in Figure 1, which represents information about crops produced at land plots. We use this model also as our running example. There is information in vector format describing the location of land plots in provinces. Further, there are raster maps of elevation, soil type, temperature, and precipitation.

A *multidimensional schema* is a finite set of dimensions and fact relationships. A *dimension* comprises at least one *hierarchy*, which contains at least one level. A hierarchy with only one level is called a *basic hierarchy*. Levels in a hierarchy (e.g., the one formed by LandPlot and Province) are related to each other through a binary relationship that defines a partial order $\preceq$ between them. Given two consecutive related levels l_i, l_j, if $l_i \preceq l_j$ then l_i is called *child* and l_j is called *parent*. When levels in a hierarchy are spatial, they are related by a topological relationship. For example, the ⬭ pictogram in the LandPlot hierarchy indicates that a land plot is covered by its parent (a province).

[1] `http://download.oracle.com/docs/html/B10827_01/geor_intro.htm`

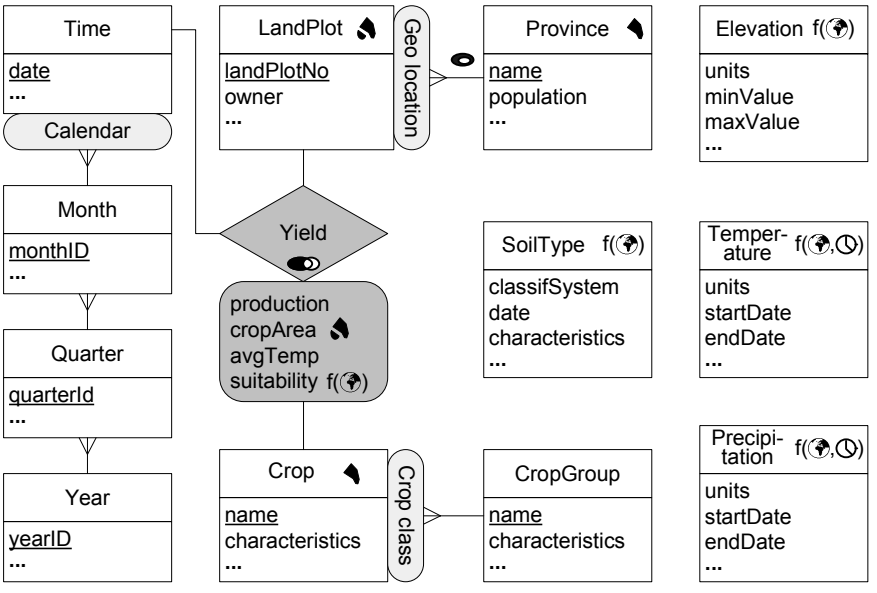

Fig. 1. An example of a spatial data warehouse with continuous fields

A level representing the least detailed data in a hierarchy is called a *leaf level* (e.g., LandPlot), and is related to at least one *fact relationship* (e.g., Yield). The latter represents an n-ary relationship between two or more leaf levels. If these levels are spatial, the relationship may also be topological and requires a spatial predicate. For example, the ⬬ pictogram in Yields indicates an intersection between the spatial dimensions LandPlot and Crop. A fact relationship contains measures, which may be thematic or spatial. The former (e.g., production) are the usual alphanumeric measures in standard OLAP, and may be calculated using spatial operators, such as distance or area. The latter are represented by a geometry or a field. An example is the cropArea measure, which is computed as the intersection of land plots and crop areas. Dimension levels are composed of key attributes and property attributes. A *key attribute* of a parent level (e.g., name in Province) determines how child members are grouped for applying aggregation functions to measures. A *property attribute* contains additional features of a level; it can be spatial (represented by a geometry or field) or thematic (alphanumeric data types).

To support fields, we include the notion of field dimensions and field measures. *Non-temporal* field levels and measures are identified by the f(🌍) pictogram, while *temporal* ones are identified by the f(🌍,🕒) pictogram. A *field dimension* is a dimension containing at least one level that is a field. In our example, the field dimensions are Elevation, SoilType, Temperature, and Precipitation where the latter two are temporal field dimensions. A *field measure* is a measure represented by a continuous field. For example, suitability is a field measure computed in terms of elements in the model, e.g., the suitability at a certain point can be a

function of the kind of crop, temperature, precipitation, and elevation, at that point or its vicinity. Finally, a *field hierarchy* is a set of related field levels; it allows a field to be seen at different granularities. Although not shown in our example, SoilType can define a hierarchy for soil classification, e.g., the USDA Soil Taxonomy.

Notice that in our approach field dimensions deserve particular treatment. In traditional multidimensional models, every dimension is connected to at least one fact relationship. The same approach has been followed in models introducing fields in spatial data warehouses (e.g., [13]), where dimension instances are values in the underlying domain (that may be obtained through on-the-fly interpolation). Due to the nature of continuous fields, there may be an infinite number of instances, each one corresponding to one possible value of the domain. We chose a different approach: we define a field dimension containing only one instance (corresponding to the function), and the attributes of the field dimension correspond to metadata describing it, like the units at which the values are recorded (e.g., Celsius or Farenheit for temperature). Consequently, field dimensions are part of the model, but are not tied to any particular fact table. The physical model we propose below shows the viability of this approach, and reveals the drawbacks of on-the-fly interpolation.

4 A Physical Model for DWs with Continuous Fields

We discuss next a physical data model supporting the conceptual model introduced in Section 3. Most conceptual models for spatial data warehouses proposed so far are based on the star/snowflake schema, but do not follow such schema when it comes to implementation issues. In these models it is not clear how a field (e.g., implemented as a raster grid) can fit into the standard star/snowflake schema. As mentioned before, we propose a different approach where field dimensions are not linked to the fact table. Although field dimensions are part of the model, and are considered as elements in the query language, there is no natural key to tie fields to a fact table. However, aggregations over fields can be included as pre-computed field measures in the fact table.

We represent fields using raster structures describing regular square grids. Our implementation is based on the OpenGIS specification for coverage geometry and interpolation functions of the OGC [15]. Other implementations of fields using irregular tesselations of space, such as triangulated irregular networks (TIN) or Voronoi diagrams, are possible and are left for future work. Nontemporal fields are stored in a table containing a spatial attribute (denoted geom) representing the geometry of the cell, and an alphanumeric attribute that stores its value. Temporal field tables have two additional attributes representing, respectively, the start and end instants of the value's validity interval.

Dimension tables represent both spatial and nonspatial dimensions. There can be either one table per dimension level or one table per dimension, depending on whether the dimension is normalized or not (i.e., either a star or a snowflake schema is used). *Spatial dimension tables* have an additional spatial attribute

(denoted by geom) containing the geometry of the object. *Field dimension tables* have, in addition to the attributes of field tables, an attribute containing the metadata of the field (which can be implemented, for example, as an XML document). Non-field dimension tables have a surrogate identifier denoted id. This identifier allows dimension tables (spatial or not) to be linked to fact tables through a foreign key relationship.

Fact tables include references to spatial and nonspatial dimension tables. We consider four kinds of measures: (1) numeric, as in standard OLAP; (2) spatial measures; (3) field measures; and (4) field aggregations. An example of a measure of type (2) is cropArea in Figure 1. This measure, of spatial type region, represents the intersection (or any other valid spatial operation) of *all* the spatial dimensions, while taking into account the other nonspatial dimensions. In our example, at a given day, a member of the LandPlot level (say, L1) may intersect a member of the Crop level (e.g., wheat). The intersection of the geometries of both members (e.g., multipolygons according to the OGC data types) results in another geometry that is recorded in the fact table as the spatial measure cropArea. An example of measure of type (3) is suitability in Figure 1. This measure is actually a field, which can be precomputed, at each point, as a function of geographic characteristics of related spatial dimensions (LandPlot and Crop), other related dimensions (Time), other fields (Elevation, ...), and other parameters. One possible way of implementing such a measure is to have a field value associated to each instance of the fact table, in our case, to each combination of land plot, crop, and time. Measures of type (4) are aggregations of measures of type (3), in a way that resembles map algebra operations. In Figure 1, measure avgTemp indicates the average value of the temperature field at the finest granularity of the fact table, that is, at the combination of land plot, crop, and day. In other words, a tuple in the fact table Yield will have an attribute that represents this temperature.

5 The Field and Temporal Field Data Types

We define next two new data types, denoted field and tempfield, and their corresponding operations, along the lines of Güting *et al.* [16].

Field types capture the variation in space of base types. They are obtained by applying a constructor field(·). Hence, a value of type field(real) (e.g., representing altitude) is a continuous function $f : \mathsf{point} \rightarrow \mathsf{real}$. We describe next some of the operations of field types.

A set of operations realize the *projection into the domain and range*. The defspace operator receives a field, and returns the geometry defining it; rangevalues receives a field, and returns the set of values that the function takes.

Another set of operators allow the *interaction with domain and range*. Operations atpoint, atpoints, atline, and atregion restrict the function to a given subset of the space defined by a spatial value. That means that the operators receive a field and a geometry, and return a field restricted to such geometry. Operations at and atrange restrict the function to a point or to a point set in the range of

the function. Predicates atmin and atmax reduce the function to the points in space when its value is minimal or maximal.

Rate of change operators compute how a field changes across space. Functions partialder_x and partialder_y give, respectively, the partial derivative of the function defining the field with respect to the one of the axis x and y.

Aggregation operators take a field as argument and produce a scalar value. Operations fmin and fmax give, respectively, the minimum and maximum value taken by the function. Three field aggregation operators take as argument a field over numeric values (int or real) and return a real value. These are volume, area, surface with their standard meaning. From these basic operators, other derived operators are defined, namely favg, fvariance, and fstdev.

All operations on base or spatial types are generalized for field types. An operation is *lifted* (following [16]) to allow any of the argument types to be replaced by the respective field type and also return a corresponding field type. Intuitively, the semantics of such lifted operations is that the result is computed at each point using the non-lifted operation. *Aggregation* operators are also uplifted in the same way. For instance, an uplifted avg operator combines several fields, yielding a new field where the average is computed at each point in space. These uplifted aggregation operations correspond to Tomlin's *local functions* [7].

Focal, zonal, and global operators can be derived from the above operators. *Focal* (or *neighborhood*) operators compute a new field in which the output value at a point is a function of the values of the input field in the neighborhood "around" that point. Neighborhoods can be defined by different sizes and geometries. Different arithmetic and statistical functions can be applied to summarize neighborhood values. For example, a focalmax that computes at each point p the maximum value of the neighborhood around that point at a distance d can be defined as follows

$$\mathsf{focalmax}(f, p, d) \stackrel{\text{def}}{=} \mathsf{fmax}(\mathsf{atregion}(f, \mathsf{buffer}(p, d))).$$

Here, the buffer operator creates a surface of radius d around point p, the atregion operator restricts the field f to that surface, and the fmax operator takes the maximum value among all the values of the resulting field.

Zonal operators take as input two fields, f_1 defining the input values and f_2 defining a set of zones, and compute an output field where the value at each point is computed from all values of the input field that belong to the zone associated with that point. For example, a zonalmax that computes at each point p the maximum value of the zone to which p belongs can be defined as follows

$$\mathsf{zonalmax}(f_1, f_2, p) \stackrel{\text{def}}{=} \mathsf{fmax}(\mathsf{atregion}(f_1, \mathsf{defspace}(\mathsf{at}(f_2, \mathsf{val}(\mathsf{atpoint}(f_2, p)))))).$$

Here, atpoint restricts the field f_2 defining the zones to the point p, val takes the value v of the field at that point, at restricts f_2 to the points that have value v, defspace obtains the underlying space where f_2 takes value v, atregion restricts the input field f_1 to that space, and the fmax operator takes the maximum value among all the values of the resulting field.

Finally, *global functions* compute a field in which the value at a point is computed from potentially all the points of the underlying space. An example is

the Euclidean distance which, given a set of "sources" defining objects of interest such as schools, hospitals, or roads, computes for each point p of the underlying space the distance to the closest source. If the sources are defined by a geometry g (of one of the four spatial types) such a function can be defined as follows

$$\mathsf{globaldistance}(p, g) \stackrel{\text{def}}{=} \mathsf{distance}(p, g),$$

where the distance function [16] determines the minimum Euclidean distance between the closest pair of points from the first and second arguments.

Temporal fields model phenomena whose value change along time and space. (e.g., temperature). The work in [6] defines temporal fields based on the moving types in [16]. Moving (or temporal) types are obtained by applying the constructor $\mathsf{moving}(\cdot)$. Hence, $\mathsf{moving}(\mathsf{real})$ (e.g., representing the temperature at a specific point) is a continuous function $f : \mathsf{instant} \rightarrow \mathsf{real}$. Temporal fields are obtained by applying a constructor $\mathsf{tempfield}(\cdot)$ which is an abbreviation of $\mathsf{moving}(\mathsf{field}(\cdot))$. We describe next some of the operators of moving types.

A set of operations realize the *projection into the domain and range*. Operations deftime and rangevalues return, respectively, the projection of a moving type into its domain and range. In other words, given a temporal field, deftime returns the intervals in which it is defined, and rangevalues returns a set with the values in its range.

Another set of operators allow the *interaction with domain and range*. Operations atinstant and atperiod restrict the function to a given time or set of time intervals. That means, given a field and a time instant (period), returns the field(s) valid at that time(s). Operations initial and final return, respectively, the (instant,value) pairs for the first and last instant of the definition time. Operation at restricts the function to a point or to a point set (a range) in the range of the function. Predicates atmin and atmax reduce the function to the times when it was minimal or maximal, respectively. The present predicate allows checking whether the temporal function is defined at an instant of time, or is ever defined during a given set of intervals. Analogously, predicate passes checks whether the function ever assumed one of the values from the range given as second argument.

Finally, as was the case for field types, all operations on nontemporal types are generalized (or lifted) for moving types. As an example, the = operator has lifted versions where one or both of its arguments can be moving types and the result is a moving Boolean. Intuitively, the semantics of such lifted operations is that the result is computed at each time instant using the non-lifted operation.

6 A SOLAP Language That Supports Fields

We now present an SQL-like query language for the model of Section 3. This model requires a language that supports different kinds of objects, namely dimensions, fact tables (spatial and non-spatial), and fields. Vaisman and Zimányi [6] proposed a query language based on the tuple relational calculus extended with aggregate functions and variable definitions proposed by Klug [17]), and

showed that extending this calculus with *field* types is enough to express multidimensional queries over fields. We base our language on this calculus.

We start with a simple example that does not include fields: "For land plots located in the province of Limburg and crops in the cereals group give the maximum production by month".

```
SELECT l.landPlotNo, t.month, max(y.production)
FROM LandPlot l, Crop c, Time t, Yield y
WHERE l.province.name="Limburg" AND c.group.name="Cereal"
GROUP BY l.landPlotNo, t.month
```

Like in typical OLAP languages, we hide the structure of the dimensions, which is stored as metadata. Also, metadata allows determining which type of objects are the ones in the FROM clause (e.g., dimension tables – spatial or not –, fact tables, or fields). This query can be trivially translated to SQL as:

```
SELECT l.landPlotNo, m.month, max(y.production)
FROM LandPlot l, Province p, Crop c, Group g, Time t, Month m, Yield y
WHERE y.landPlot=l.id AND l.province= p.id AND p.name="Limburg"
AND y.crop=c.id AND c.group=g.id AND g.name="Cereal"
AND y.time=t.id AND t.month=m.id
GROUP BY l.landPlotNo, m.month
```

We next introduce fields in the language. Let us start with a simple query, not involving a fact table: "Total area at sea level in the province of Antwerp".

```
SELECT area(intersection(defspace(at(e.geom,0)),l.province.geom))
FROM Elevation e, LandPlot l
WHERE l.province.name="Antwerp"
```

Function at restricts the elevation field to the points in space that have the value 0, and defspace yields the region containing such points, which is then intersected with the province of Antwerp. The area operator is finally applied.

The next query includes a fact table: "For land plots having at least 30% of their surface at the sea level, give the average suitability value for wheat on February 1st, 2009."

```
SELECT l.LandPlotNo, favg(y.suitability)
FROM Elevation e, LandPlot l, Yield y, Crop c, Time t
WHERE area(defspace(atregion(at(e,0),l.geom)))/area(l.geom) > 0.3
AND c.name="Wheat" AND t.date="02/01/2009"
```

Here, the elevation field is restricted to the value 0 by means of function at, and the resulting field is restricted to the geometry of the land plot with function atregion. The operator defspace obtains the geometry of the restricted field, the area of this geometry is computed, and this is finally divided by the total area of the land plot. Then, the average suitability is computed using the field aggregation operation favg applied to the field measure suitability.

We now show a spatio-temporal query including fields: "Land plots at the sea level in Limburg with average temperature greater than 10 °C in March 2009 and suitability (at every point of the land plot) for a wheat crop at June 1st, 2009 greater than 1.4."

```
SELECT l.landPlotNo
FROM LandPlot l, Crop c, Time t, Temperature temp, Yield y
WHERE l.province.name="Limburg" AND
favg(avg(atperiods(atregion(temp,l.geom),["03/01/09","03/31/09"])))>10
AND intersects(defspace(at(e,0)),l.geom)
AND t.date= "1/6/2009" AND c.name="Wheat"
AND defspace(atrange(y.suitability,[1.4,-]))=l.geom
```

The temperature field, restricted to the geometry of the land plot and to March 2009, is aggregated with the avg operator (a local cubic operation). Then, favg is applied to obtain the average at the land plot, which is then compared to 10. The topological predicate intersects verifies that the land plot overlaps the region defined by the elevation field restricted to the sea level. After obtaining the instance of the fact relationship relating the land plot, the date, and the wheat crop, the suitability field for this instance is restricted to the points that have a value greater than 1.4, the region containing those points is obtained with function defspace, and it is verified that this region equals the geometry of the land plot, ensuring that every point satisfies the condition.

Finally, we show an example of a query returning a field: "Restrict the precipitation field to December, 2009, to the areas with an altitude greater than 150m, and an average production of wheat greater than one thousand tons."

```
SELECT atregion(atregion(atperiod(p,["12/1/2009","12/31/2009"]),
      defspace(atrange(e,[150,-]))),
    (SELECT l.geom
    FROM Yield y, LandPlot l, Crop c
    WHERE c.name="Wheat"
    GROUP BY l.geom
    HAVING AVG(y.production) > 1000))
FROM Elevation e, Precipitation p
```

The atperiod function restricts the precipitation field to December, 2009 and the result is restricted (inner atregion) to the space defined (defspace) by the restriction of the elevation field to values greater than 150 (atrange). The outer atregion function restricts this resulting field to the result of the inner query which returns the set of geometries for land plots having an average production of wheat greater than one thousand.

7 Implementing the Operators

We show now how the operators over fields are implemented. We designed the following experimental scenario, according to the conceptual model of Figure 1.

We downloaded field data from the WorldClim site[2], which provides layers with raster information at different resolutions. For our region of interest (a portion of Belgium), we used elevation data with a resolution of 5 arc-minutes, obtaining 655 cells, and temperature and precipitation data with a resolution of 10 arc-minutes, obtaining 185 cells. Raster data was downloaded in a generic grid format exported to ESRI Shape file format[3], an later imported to a PostgreSQL database with the PostGIS plugin[4] . This generates polygons with associated values. The units for elevation, precipitation, and temperature are meters, milimeters, and Celsius * 10, respectively. Both, precipitation and temperature data correspond to monthly values. We created synthetically dimension and fact data (e.g., land plots, crops). As we explained in Section 4, fields are stored in tables with attributes 'geom' and 'value'. In addition, temporal fields have attributes 'startDate' and 'endDate' representing the validity interval of the field.

We now show how the defspace and atregion operators are implemented. The other ones are implemented analogously. Since the actual Java code is self-descriptive, we have chosen to show this code instead of pseudo-code listings.

```
(1) Geometry defspace(String tempFieldTable) throws SQLException {
(2) String sqlDML;
(3) sqlDML= String.format("SELECT geom FROM %s", tempFieldTable);
(4) PreparedStatement pstmt = dbConn.prepareStatement(sqlDML);
(5) ResultSet rs = pstmt.executeQuery();
(6) Collection⟨Geometry⟩ geomCollection = new ArrayList⟨Geometry⟩();
(7) while (rs.next()){
(8)    Geometry aGeom = GeometryReader.getGeometry(rs.getObject(1));
(9)    geomCollection.add(aGeom);}
(10) pstmt.close();
(11) return unionAll(geomCollection);}
```

Fig. 2. A Java function to compute defspace

Figure 2 shows a Java function implementing the defspace operator. It receives as parameter the name of the table representing a field and returns the geometry over which the field is defined (i.e., the union of all the polygons that the field contains). The SQL statement in Line (3) retrieves the spatial element in the field table. The loop in Line (7) creates a collection of these geometries.

Note that Figure 3 shows two Java functions that implement the atregion operator. For implementation reasons we need to define two different functions that differ in the type of second parameter. The first atregion function receives a field and a geometry as parameters, and returns a field restricted to the boundaries of the geometry. If the geometry is empty, the field is not updated. The SQL statement in Line (5) deletes the tuples of the field that have no intersection with the geometry. The statement in Line (9) updates the spatial attribute

[2] http://www.worldclim.org/current

[3] http://www.esri.com/

[4] http://www.postgresql.org/; http://www.postgis.org/.

```
void atregion(String tempFieldTable, Geometry geom) throws SQLException {
(1) if (geom.isEmpty())
(2)   return;
(3) String sqlDML;
(4) PreparedStatement pstmt;
(5) sqlDML= String.format("DELETE FROM %s WHERE NOT
                           INTERSECTS(geom, %s)", tempFieldTable, geom);
(6) pstmt = dbConn.prepareStatement(sqlDML);
(7) pstmt.execute();
(8) pstmt.close();
(9) sqlDML= String.format("UPDATE %s SET geom=INTERSECTION(geom, %s)",
                           tempFieldTable, geom);
(10) pstmt = dbConn.prepareStatement(sqlDML);
(11) pstmt.execute();
(12) pstmt.close();}

void atregion(String tempFieldTable, Collection⟨Geometry⟩ geomCollection)
       throws SQLException {
(1) if (geomCollection.isEmpty())
(2)   return;
(3) atregion(tempFieldTable, unionAll(geomCollection)); }

Geometry unionAll(Collection⟨Geometry⟩ geomCollection){
(1) Geometry[] geomArray= new Geometry[geomCollection.size()];
(2) int i=0;
(3) for(Iterator⟨Geometry⟩ iter = geomCollection.iterator(); iter.hasNext(); i++) {
(4)   geomArray[i]= iter.next(); }
(5) GeometryFactory geometryFactory = new GeometryFactory();
(6) GeometryCollection polygonCollection=
      geometryFactory.createGeometryCollection(geomArray);
(7) Geometry union = polygonCollection.union();
(8) return union;}
```

Fig. 3. Java functions implementing the atregion operator

of the remaining tuples with the intersection between the field and the geometry. Since the underlying language does not provide a 'Union' operator that recursively computes the union of a set of geometries, we implemented a second version of atregion. which first computes the union of all geometries in the second parameter by invoking function unionAll. Its result is used in a call to the first atregion function explained above. Line (7) in function unionAll computes a union of geometries. Lines (5) and (6) are only for type conversion.

8 Implementing the Language

In this section we show how the last query in Section 6 is translated and executed. Figure 4 shows part of the computation of this query. The upper part of Figure 4 shows the sequence of function calls starting from the inner operator of

```
(1) String fieldTempTableNameElev= initField("Elevation");
(2) atrange(fieldTempTableNameElev, 150, Double.MAX_VALUE);
(3) Geometry unionField = defspace(fieldTempTableNameElev);
(4) String fieldTempTableNamePrec = initField("Precipitation");
(5) atperiod(fieldTempTableNamePrec, "12/1/2009", "12/31/2009");
(6) atregion(fieldTempTableNamePrec, unionField);
(7) lastPhase(fieldTempTableNamePrec);

public void lastPhase(String fieldTempTableName) throws SQLException {
(1) String sqlDML=
(2) "SELECT l.geom" +
(3) "FROM Yield y, LandPlot l, Crop c" +
(4) "WHERE c.name=\"Wheat\" " +
(5) "AND y.landPlot=l.landplotNo AND y.cropId =c.id " +
(6) "GROUP BY l.geom" +
(7) "HAVING AVG(y.production) > 1000";
(8) PreparedStatement pstmt = dbConn.prepareStatement(sqlDML);
(9) ResultSet rs = pstmt.executeQuery();
(10) Collection⟨Geometry⟩ geomCollection = new ArrayList⟨Geometry⟩();
(11) while (rs.next()){
(12)    Geometry aGeom = GeometryReader.getGeometry(rs.getObject(1));
(13)    geomCollection.add(aGeom); }
(14) pstmt.close();
(15) atregion(fieldTempTableName, geomCollection);
(16) spatialDump(fieldTempTableName, "_A");}
```

Fig. 4. Query evaluation

the SELECT clause of the query (which, remember, returns a field). Since we do not assume that field data fit in main memory, we use a temporary table that is updated by sequentially applying the functions explained in Section 7. Let us be more concrete. We have shown in Section 7 that the atregion operator updates geometries and deletes tuples. Thus, the function initField(nameOfFieldTable) (Line (1) in Figure 4) generates a temporary table containing the data in the original field table (in this case, Elevation). This table is the one that changes during the execution of the query, preserving the original field. Then, in Line (2) atrange is applied over the field returned in the previous step to delete the tuples that do not satisfy the condition (elevation > 150). A unique geometry is then generated over the result from the previous step using defspace (Line (3)). Then, a temporary table is created for the precipitation temporal field (Line (4)), atperiod is applied to the precipitation table for restricting the time frame of the field in Line (5), and atregion is applied to the field obtained in the previous step for restricting it with the geometry returned in Line (3). Finally, the function lastPhase is called. This function computes the collection of geometries corresponding to the inner query in the FROM clause. In Line (9) of lastPhase (shown in the lower part of Figure 4), the *translated* inner query is executed (where all the implicit joins are written in Line (5)), returning the land plots

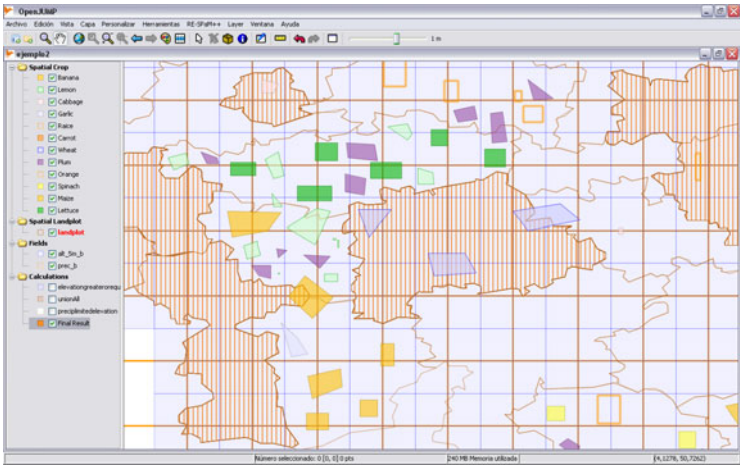

Fig. 5. The field resulting from the query

satisfying the query as a set of geometries collected in the loop in Line (11). In Line (15) atregion is invoked, and in Line (16) the result is returned.

Figure 5 shows the result of the query execution. There are two grids of different precision, one for elevation and one for precipitation. The zones with vertical bars indicate the resultant field, i.e, a precipitation field in regions with the desired altitude, and only one kind of crop.

9 Conclusion and Future Work

We have presented a physical model for spatio-temporal data warehouses that supports continuous fields. This model is based on two new data types, namely field and tempfield. These data types have a collection of operators, which we discussed. A relevant contribution of the present paper is the implementation of these operators and an associated SQL-like language that allows expressing SOLAP queries over continuous fields. The main goal of this implementation consists in showing the viability of our approach.

As future work, we will perform extensive testing of the operators and the language proposed here. Since spatio-temporal data warehouses contain huge amounts of data, optimization issues are extremely important. They include issues such as appropriate index structures, pre-aggregation, and efficient query optimization, among others. With respect to the latter, our example queries can be expressed in several ways, exploiting either the fact relationship or the geometries of the dimension levels with spatial and topological operators. Although these alternative queries yield the same result, the evaluation time of them may vary significatively, depending on the actual population of the data warehouse. Finally, we will consider other possible implementations of fields such as triangulated irregular networks (or TINs) and Voronoi diagrams.

References

1. Worboys, M.F., Duckham, M.: GIS: A Computing Perspective, 2nd edn. CRC Press, Second edn (2004)
2. Kimball, R.: The Data Warehouse Toolkit. J. Wiley and Sons, Inc., Chichester (1996)
3. Rivest, S., Bédard, Y., Marchand, P.: Toward better suppport for spatial decision making: Defining the characteristics of spatial on-line analytical processing (SOLAP). Geomatica 55, 539–555 (2001)
4. Bédard, Y., Rivest, S., Proulx, M.: Spatial online analytical processing (SOLAP): Concepts, architectures, and solutions from a geomatics engineering perspective. In: Wrembel-Koncilia (ed.) Data Warehouses and OLAP: Concepts, Architectures and Solutions, pp. 298–319. IRM Press (2007)
5. Paolino, L., Tortora, G., Sebillo, M., Vitiello, G., Laurini, R.: Phenomena: a visual query language for continuous fields. In: Proc. of ACM-GIS, pp. 147–153 (2003)
6. Vaisman, A.A., Zimányi, E.: A multidimensional model representing continuous fields in spatial data warehouses. In: Proc. of ACM-GIS, pp. 168–177 (2009)
7. Tomlin, D.: Geographic Information Systems and Cartographic Modelling. Prentice-Hall, Englewood Cliffs (1990)
8. Câmara, G., Palomo, D., de Souza, R.C.M., de Oliveira, D.: Towards a generalized map algebra: Principles and data types. In: Proc. of GeoInfo, pp. 66–81 (2005)
9. Cordeiro, J.P., Câmara, G., Moura, U.F., Barbosa, C.C., Almeida, F.: Algebraic formalism over maps. In: Proc. of GeoInfo., pp. 49–65 (2005)
10. Mennis, J., Viger, R., Tomlin, C.: Cubic map algebra functions for spatio-temporal analysis. Cartography and Geographic Information Science 32, 17–32 (2005)
11. Laurini, R., Paolino, L., Sebillo, M., Tortora, G., Vitiello, G.: A spatial SQL extension for continuous field querying. In: Proc. of COMPSAC Workshops, pp. 78–81 (2004)
12. Shanmugasundaram, J., Fayyad, U.M., Bradley, P.S.: Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In: Proc. of KDD, pp. 223–232 (1999)
13. Ahmed, T.O., Miquel, M.: Multidimensional structures dedicated to continuous spatiotemporal phenomena. In: Jackson, M., Nelson, D., Stirk, S. (eds.) BNCOD 2005. LNCS, vol. 3567, pp. 29–40. Springer, Heidelberg (2005)
14. Malinowski, E., Zimányi, E.: Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications. Springer, Heidelberg (2008)
15. Open Geospatial Consortium Inc.: OpenGIS Abstract Specification: Topic 6: The Coverage Type and its Subtypes. OGC 07-011, Version 4 (2007)
16. Güting, R.H., Schneider, M.: Moving Objects Databases. Morgan Kaufmann, San Francisco (2005)
17. Klug, A.: Equivalence of relational algebra and relational calculus query languages having aggregate functions. Journal of the ACM 29, 699–717 (1982)