# The VagueGeometry Abstract Data Type

Anderson Chaves Carniel[1], Ricardo Rodrigues Ciferri[2], Cristina Dutra de Aguiar Ciferri[1]

[1] University of São Paulo, Brazil
accarniel@gmail.com, cdac@icmc.usp.br
[2] Federal University of São Carlos, Brazil
ricardo@dc.ufscar.br

**Abstract.** *Spatial vagueness* has been increasingly required by geoscientists to handle *vague spatial objects*, that is, spatial objects found in real-world phenomena that do not have exact locations, strict boundaries, or sharp interiors. However, there is a gap in the literature in how to handle spatial vagueness in spatial database management systems and Geographical Information Systems (GIS) since they mainly provide support to *crisp spatial objects*, that is, objects that have well-defined locations, boundaries, and interiors. In this article, we fill this gap. We propose VagueGeometry, a novel *abstract data type* that allows users to manipulate vague spatial objects in spatial applications and GIS. The main advantages of our VagueGeometry are that (i) it offers textual and binary representations for vague spatial objects, (ii) it includes an expressive set of vague spatial operations, (iii) it supports SQL operators, and (iv) its implementation is open source. Experimental results show that VagueGeometry improved the performance of spatial queries with vague topological predicates from 21% up to 98% if compared with functionalities available in current spatial databases.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Spatial databases and GIS

Keywords: abstract data types, spatial databases, vague spatial objects, vague topological predicates

## 1. INTRODUCTION

Spatial database management systems (spatial DBMS) and Geographical Information Systems (GIS) mainly provide support to handle *crisp spatial objects* to represent real-world phenomena by using crisp points, crisp lines, and crisp regions. Crisp spatial objects characterize spatial phenomena with exact locations and whose shape and boundary are precisely defined [Schneider and Behr 2006]. Examples are cities with their political boundaries. For their handling, spatial operations like geometric set operations (for instance, union), topological predicates (for instance, overlap), and numerical operations (for instance, distance) are defined and used in spatial queries.

However, geoscientists are increasingly interested in modeling spatial real-world phenomena that do not have exact locations, strict boundaries, or sharp interiors. This characterization leads to *spatial vagueness*. There are several real-world phenomena that are characterized by spatial vagueness, such as habitats of migratory species, rivers with different levels of water in different time periods, and air polluted areas that can vary due to climatic changes. In general, it has a spatial extent that *certainly* belongs to the real-world phenomena (that is, the *kernel*) and a spatial extent that *maybe* belongs to the real-world phenomena (that is, the *conjecture*) [Siqueira et al. 2014].

There are a number of approaches proposed in the literature that define models to represent spatial vagueness, which can be classified as *probabilistic models* [Li et al. 2007; Zinn et al. 2007], *fuzzy models* [Kraipeerapun 2004; Dilo et al. 2006; Dilo et al. 2007; Carniel et al. 2014], and *exact models* [Clementini and Di Felice 1997; Pauly and Schneider 2008; 2010; Bejaoui et al. 2010]. These models introduce concepts and notions of *vague spatial objects* by formally defining spatial data types

The authors have been supported by the Brazilian research agencies FAPESP, CAPES, and CNPq.

for *vague points*, *vague lines*, and *vague regions*. They also introduce vague spatial operations to handle them, that is, *vague geometric set operations* (for instance, vague geometric union), *vague topological predicates* (for instance, vague overlap), and *vague numerical operations* (for instance, vague distance).

There are several advantages of incorporating vague spatial objects and their operations into spatial databases, such as to provide a more realistic representation of application environments, to allow users to manipulate vague spatial objects found in real-world phenomena, and to provide an efficient processing of operations on vague spatial objects. For instance, in an ecological application, users aim to manage habitats of species and polluted areas of rivers. The habitats and the polluted areas of rivers are represented by vague regions. This means that habitats have locations where species certainly appear and locations where species maybe appear. Further, rivers have unpolluted areas and areas where there is some kind of pollution. By using such data, a user can ask the following query: "Find all polluted areas of rivers that *maybe overlap* with habitats of species". In this query, spatial vagueness is represented by the *maybe overlap* predicate, which will return true when the overlap occurs to some extent, that is, occurs with some degree of uncertainty.

To handle spatial objects in spatial applications, *abstract data types* (ADT) have been used in spatial DBMS and GIS. An ADT aids the use of spatial operations in spatial queries by hiding their complexities from the user. While ADTs for crisp spatial data are deeply implemented in the literature, this is not the case for vague spatial data. Although there are approaches that provide ADTs for vague spatial data [Kraipeerapun 2004; Dilo et al. 2006; Zinn et al. 2007; Pauly and Schneider 2008; 2010; Carniel et al. 2015a], they face several limitations. First, they only provide support for a small subset of vague spatial operations. Second, they do not support textual and binary representations of vague spatial objects. Third, they do not support SQL operators to handle results of vague spatial operations. Finally, the majority of the approaches does not implement vague spatial objects in a spatial DBMS, or are specifically designed to run on a proprietary DBMS.

In this article, we fill this gap in the literature. We propose a novel ADT named VagueGeometry to incorporate vague spatial objects into a spatial DBMS. VagueGeometry is based on the *exact model* since this model reuses existing concepts and implementations of crisp spatial data and formally defines a complete set of vague spatial operations. The main advantage to use implementation of crisp spatial data is that they are well defined and their efficiency is largely explored in the literature. Among the exact models [Clementini and Di Felice 1997; Pauly and Schneider 2008; 2010; Bejaoui et al. 2010], VagueGeometry is based on the *Vague Spatial Algebra* (VASA) [Pauly and Schneider 2008; 2010] since it formally defines simple and complex vague spatial data types. Further, VASA introduces a more expressive algebra than the models described in Bejaoui et al. [2010] and Clementini and Di Felice [1997] by including a huge set of operations, such as vague geometric set operations, vague topological predicates, and vague numerical operations.

VagueGeometry greatly overcomes the aforementioned limitations. This article extends our previous work in Carniel et al. [2015b], which presents a first version of the VagueGeometry. We highlight several unpublished subjects addressed in this article, such as (i) graphical representations of vague spatial objects, (ii) several examples of instance of textual representations of VagueGeometry, (iii) type-dependent vague spatial operations (for instance, common border between two vague region objects), (iv) SQL signatures for all VagueGeometry operations, (v) a running example that shows how to use the VagueGeometry in SQL queries, and (vi) an additional experimental evaluation that analyzes the performance of VagueGeometry in the query processing for vague spatial joins. Hence, our proposed VagueGeometry has the following characteristics.

—It offers textual and binary representations for vague spatial objects, which make possible their use to define, insert, and retrieve vague spatial objects by using textual or binary representations. Further, these representations can be used as a way of communication and interoperability between different spatial applications.

—It implements an expressive set of spatial operations for vague spatial objects. To comply with this goal, VagueGeometry includes the specification of vague geometric set operations, vague topological predicates, vague numerical operations, and type-dependent operations. As a result, the use of VagueGeometry empowers the management of vague spatial objects in spatial applications by users.

—It supports SQL operators that allow users to handle results of vague topological predicates and vague numerical operations.

—It is open source and implemented in the open source PostgreSQL DBMS with the PostGIS spatial extension. This means that spatial applications are able to access directly a spatial database containing vague spatial objects and handle these objects by using vague spatial operations accordingly. A running example is employed in order to show how to use VagueGeometry in SQL queries.

—It includes an efficient improvement to process vague topological predicates in spatial queries, such as vague range queries and vague spatial joins.

This article is organized as follows. Section 2 surveys related work and shows a comparison table between the VagueGeometry and the current related work. Section 3 summarizes the Vague Spatial Algebra. Section 4 introduces the proposed VagueGeometry by specifying its operations and shows how to use VagueGeometry in SQL queries by using a running example. Section 5 details the improvement in the processing of vague topological predicates. Section 6 describes performance tests. Section 7 concludes the article and presents future work.

## 2. RELATED WORK

Few approaches implement ADTs for manipulating vague spatial objects in spatial DBMS and GIS [Kraipeerapun 2004; Dilo et al. 2006; Zinn et al. 2007; Pauly and Schneider 2008; 2010; Carniel et al. 2015a]. These approaches mainly differ from our proposed VagueGeometry in the practicable applicability of the user to handle vague spatial objects in spatial queries. We compare these approaches with VagueGeometry by considering the following functionalities related to the support provided by them: (i) textual representation, (ii) binary representation, (iii) spatial operations that return spatial data types values, that is, vague geometric set operations, (iv) spatial relationships, that is, vague topological predicates, (v) spatial operations that return numbers, that is, vague numerical operations, (vi) SQL operators, and (vii) coupling with a spatial DBMS.

Table I shows a comparison among the functionalities provided by related work and VagueGeometry (last column). The majority of the approaches does not provide textual and binary representations (functionalities (i) and (ii)) to allow the user to insert and retrieve vague spatial objects. While Zinn et al. [2007] provides input and output functions for vague spatial objects by means of the binary format, this is not the case for the textual representation. The approaches described in Pauly and Schneider [2008] and Pauly and Schneider [2010] define vague spatial objects by using extensive terminal command lines without any textual or binary representations. The approaches described in Kraipeerapun [2004] and Dilo et al. [2006] support options to represent vague spatial objects by using files in the format of the GRASS GIS. However, the binary and textual formats are not understandable for users and depend on a specific system, thus limiting interoperability between spatial applications. Carniel et al. [2015a] provides a textual representation for manipulating vague spatial objects; but, this is not the case for the binary representation. On the other hand, VagueGeometry define textual and binary representations for vague spatial objects.

Regarding functionality (iii), the approaches described in Zinn et al. [2007], Pauly and Schneider [2008], Pauly and Schneider [2010], and Carniel et al. [2015a] implement vague geometric union, vague geometric intersection, and vague geometric difference between vague points, vague lines, and vague regions. The approaches described in Kraipeerapun [2004] and Dilo et al. [2006] do not implement the vague geometric difference between vague lines. Regarding functionalities (iv) and (v), the approaches described in Pauly and Schneider [2008] and Pauly and Schneider [2010] provide support to vague

Table I. Comparisons of VagueGeometry with related work.

| Functionality | Approaches | | | | |
|---|---|---|---|---|---|
| | Kraipeerapun [2004]; Dilo et al. [2006] | Zinn et al. [2007] | Pauly and Schneider [2008]; Pauly and Schneider [2010] | Carniel et al. [2015a] | Proposed VagueGeometry |
| (i) Textual Representation | Yes | No | No | Yes | Yes |
| (ii) Binary Representation | Yes | Yes | No | No | Yes |
| (iii) Vague Geometric Set Operations | Yes, but not all | Yes | Yes | Yes | Yes |
| (iv) Vague Topological Predicates | No | No | Yes | No | Yes |
| (v) Vague Numerical Operations | No | No | Yes | No | Yes |
| (vi) SQL Operators | No | No | No | No | Yes |
| (vii) Coupling with a spatial DBMS | No | Yes, using PostgreSQL | Yes, using Oracle | Yes, using PostgreSQL | Yes, using PostgreSQL |

topological predicates and vague numerical operations, while the remaining approaches do not provide support for these operations, and therefore, limit the management of vague spatial objects and the type of queries that can be processed. Our proposed VagueGeometry implements an expressive set of spatial operations for vague spatial objects, which includes the specification of vague geometric set operations, vague topological predicates, and vague numerical operations.

Furthermore, there is no related work that support SQL operators (functionality (vi)). Although the approaches described in Pauly and Schneider [2008] and Pauly and Schneider [2010] propose some operators, they do not implement them. As a result, we fill Table I with "No" with regard to these approaches and functionality (vi). However, offering operators in the SQL language is an important functionality since it allows users to intuitively handle results of spatial operations in SQL queries. Therefore, differently from related work, VagueGeometry supports SQL operators.

Finally, regarding functionality (vii), the approaches described in Kraipeerapun [2004] and Dilo et al. [2006] do not implement vague spatial objects in a spatial DBMS, while the approaches described in Zinn et al. [2007], Pauly and Schneider [2008], Pauly and Schneider [2010], and Carniel et al. [2015a] offer this functionality. However, the implementation of Pauly and Schneider [2008] and Pauly and Schneider [2010][1] is developed in Oracle, which has license restrictions. On the other hand, VagueGeometry is an open source implementation based on the PostgreSQL DBMS with the PostGIS spatial extension.

## 3. VAGUE SPATIAL ALGEBRA

Vague Spatial Algebra (VASA) [Pauly and Schneider 2008; 2010] is an exact model that defines vague spatial objects, which can be simple or complex. In addition, VASA defines a huge set of operations to handle these objects, and thus, this model is more extensive than other exact models. A vague spatial object in VASA is defined as a pair of crisp spatial objects of the same spatial data type, which must be disjoint or adjacent. The first object represents the *kernel* part, while the second object represents the *conjecture* part. Further, VASA is characterized by separate the portions of space of a spatial object by considering minimum and maximum approximations. As a result, a spatial object can comprise or expand according to a minimum limit (that is, the kernel part) and a maximum limit (that is, the conjecture part). Formally, let $\alpha \in \{crisp\ point, crisp\ line, crisp\ region\}$. Then, a vague spatial data type in VASA is defined by $v(\alpha) = \alpha \times \alpha$, such that for an object $o = (o_k, o_c) \in v(\alpha)$, the property $disjoint(o_k, o_c) \vee meet(o_k, o_c)$ holds. The kernel and conjecture of $o$ are symbolized by $o_k$ and $o_c$, respectively.

[1] http://www.cise.ufl.edu/research/SpaceTimeUncertainty/

VASA defines the following operations to handle vague spatial objects: vague geometric set operations, vague topological predicates, type-dependent vague spatial operations, and vague numerical operations. Vague geometric set operations are defined by reusing the crisp geometric set operations, that is, crisp geometric union, crisp geometric intersection, and crisp geometric difference. Vague topological predicates are based on the three-valued logic, and thus, can return *true*, *false*, or *maybe*. Let $A$ and $B$ be two vague spatial objects. A vague topological predicate is evaluated by using the well-known crisp 9-intersection matrix [Schneider and Behr 2006] for the following combinations: $A_k \times B_k$, $A_k \times (B_c \oplus B_k)$, $(A_k \oplus A_c) \times B_k$, and $(A_k \oplus A_c) \times (B_c \oplus B_k)$, where $\oplus$ denotes the crisp geometric union. Type-dependent vague spatial operations deal with a limited subset of possible types of input. For instance, the common border operation computes the shared boundary between two vague region objects and yields a vague line object. Finally, vague numerical operations return a pair of numeric values corresponding to a minimum and a maximum value. For instance, the area of a vague region object has a minimum value that corresponds to the area of its kernel and a maximum value that corresponds to the area of the union between its kernel and its conjecture. Details of the formal definitions for vague spatial operations of VASA are given in Pauly and Schneider [2010].

## 4. THE VAGUEGEOMETRY ABSTRACT DATA TYPE

In this section, we propose VagueGeometry, an ADT to handle vague spatial objects in a spatial DBMS. VagueGeometry was implemented by using the C language and the extensibility provided by the PostgreSQL internal library. It is based on VASA, and thus, we make use of the spatial operations provided by PostGIS and GEOS to implement the vague spatial data types and their operations. GEOS[2] is a C/C++ library that implements crisp spatial data types and their crisp spatial operations according to the OGC specifications[3]. A detailed documentation of VagueGeometry is available at `http://gbd.dc.ufscar.br/vaguegeometry/`.

This section introduces VagueGeometry as follows. Section 4.1 details the VagueGeometry data types and their textual and binary representations, while Section 4.2 summarizes the main vague spatial operations and Section 4.3 introduces the SQL operators. Finally, Section 4.4 shows examples of using VagueGeometry in a spatial application.

### 4.1 Representation of Vague Spatial Objects

Figure 1 depicts vague spatial data types of VagueGeometry, which can be simple or complex. Simple vague spatial data types named *vague point*, *vague linestring*, and *vague polygon* denote simple vague points, simple vague lines, and simple vague regions, respectively. Complex vague spatial data types named *vague multipoint*, *vague multilinestring*, and *vague multipolygon* denote complex vague points, complex vague lines, and complex vague regions, respectively. We employ this notation to follow the same notation used by the OGC specification for crisp spatial objects. Note that a vague spatial object of VagueGeometry is composed of a pair of disjoint or adjacent crisp spatial objects of the same spatial data type, which are showed in gray in Figure 1. Figure 2 shows examples of VagueGeometry objects, that is, instances of vague spatial data types. For vague points and vague regions, black objects denote the kernel part, while gray objects denote the conjecture part. For vague lines, solid lines denote the kernel part, while dashed lines denote the conjecture part.

In order to insert and retrieve VagueGeometry objects, we propose textual and binary representations for vague spatial objects. We present our proposed representations by first detailing the *textual representations*. They are: (i) *Vague Well-Known Text* (VWKT), (ii) *Vague Geography Markup Language* (VGML), (iii) *Vague Keyhole Markup Language* (VKML), and (iv) *Vague Geographic JavaScript Object Notation* (vGeoJSON). These textual representations are based on the OGC specifications that

[2] http://trac.osgeo.org/geos/
[3] http://www.opengeospatial.org/

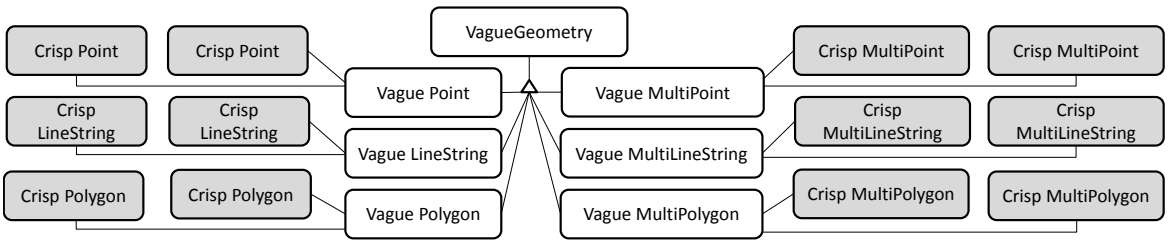



Fig. 1. The vague spatial data types of VagueGeometry.

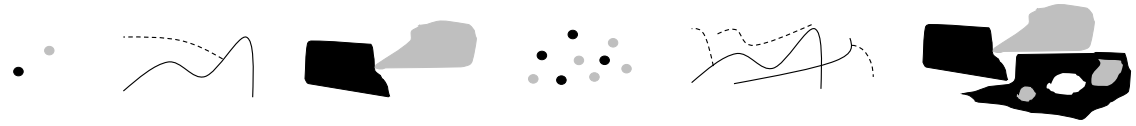



Fig. 2. Examples of VagueGeometry objects according to its data type (Figure 1): a vague point object (a), a vague linestring object (b), a vague polygon object (c), a vague multipoint object (d), a vague multilinestring object (e), and a vague multipolygon object (f).

use the following textual representations for crisp spatial objects: Well-Known Text (WKT), Geography Markup Language (GML), Keyhole Markup Language (KML), and Geographic JavaScript Object Notation (GeoJSON).

The VWKT, VGML, VKML, and vGeoJSON representations are defined as follows. Let $A$ be a VagueGeometry object, which can assume different data types (Figure 1), formed by the kernel $A_k$ and the conjecture $A_c$. Let *name* be a function that returns a keyword representing the data type of $A$. For instance, $name(A)$ returns the keyword `VAGUEPOINT` if $A$ is a vague point object. Finally, let *WKT*, *GML*, *KML*, and *GeoJSON* be functions that get a crisp spatial object as input and return its respective textual representation. The textual representations for a VagueGeometry object $A$ are defined as follows.

(i) $VWKT(A) = name(A)(WKT(A_k); WKT(A_c))$

(ii) $VGML(A) =$ <vgml:$name(A)$><vgml:Kernel>$GML(A_k)$</vgml:Kernel>
<vgml:Conjecture>$GML(A_c)$</vgml:Conjecture></vgml:$name(A)$>

(iii) $VKML(A) =$ <vkml:$name(A)$><vkml:Kernel>$KML(A_k)$</vkml:Kernel>
<vkml:Conjecture>$KML(A_c)$</vkml:Conjecture></vkml:$name(A)$>

(iv) $vGeoJSON(A) =$ {"type": "$name(A)$", "kernel": $GeoJSON(A_k)$, "conjecture": $GeoJSON(A_c)$}

Additionally, textual representations can contain the Spatial Reference System Identifier (SRID), which is an unique numerical value that identifies the spatial coordinate system definitions. For the VWKT representation, the SRID is specified by adding its number at the beginning of its representation, and thus, an *Extended*-VWKT format is obtained. For the VGML representation, the SRID is specified by adding the attribute *srsName* in the tag that indicates the VagueGeometry data type. Finally, for the vGeoJSON representation, the SRID is specified by adding the attribute *crs*.

We now move our discussion to the proposed *binary representation*, called *Vague Well-Known Binary* (VWKB). It is based on the Well-Known Binary (WKB) representation for crisp spatial objects documented in the OGC specification. Our VWKB representation is defined as follows. Let *id* be a function that returns an integer in the binary format symbolizing the data type of $A$. For instance, $id(A)$ returns 1, in the binary format, if $A$ is a vague point object. Let *WKB* be a function that gets a crisp spatial object as input and returns its respective WKB representation. Let *endianess*

be an flag that indicates the way in which the bytes are organized in main memory (that is, either *big-endian* or *little-endian*). The VWKB representation for a VagueGeometry object $A$ is defined as follows.

$$VWKB(A) = endianess + id(A) + WKB(A_k) + WKB(A_c),$$

where + denotes the union between the serialized data.

VagueGeometry supports textual and binary representations to allow its use in different spatial applications. Hence, spatial applications based on XML or web services that use XML as communication are able to use the VGML and VKML representations. Web applications that utilize JavaScript as main language are able to use the vGeoJSON representation. Applications that manage binary files are able to use the VWKB representation. Finally, for general purpose, applications can make use of the VWKT representation. It is important to note that these representations also provide interoperability between applications since a vague spatial object has an unique representation. For each VagueGeometry data type (Figure 1), Tables II and III show examples of VWKT and vGeoJSON representations, respectively. Note that the same VagueGeometry objects are used in these tables in order to visualize them in different representations. For instance, the same vague point object is used in the tables (first line of each table).

### 4.2 Vague Spatial Operations

VagueGeometry provides support to *input and output operations*, *vague geometric set operations*, *vague topological predicates*, *vague numerical operations*, and *type-dependent vague spatial operations*. While *input operations* transform textual or binary representations into a VagueGeometry object, *output operations* transform a VagueGeometry object into a textual or binary representation. The SQL signatures for input operations, that consider the VWKT, VGML, VKML, vGeoJSON, and VWKB representations as input, are defined as follows, respectively.

(i) *VG_VagueGeomFromText*(text *VWKT*, integer *SRID*) → VagueGeometry
(ii) *VG_VagueGeomFromVGML*(text *VGML*) → VagueGeometry
(iii) *VG_VagueGeomFromVKML*(text *VKML*) → VagueGeometry
(iv) *VG_VagueGeomFromvGeoJSON*(text *vGeoJSON*) → VagueGeometry
(v) *VG_VagueGeomFromVWKB*(bytea *VWKB*, integer *SRID*) → VagueGeometry

The SQL signatures for output operations of the VWKT, VGML, VKML, vGeoJSON, and VWKB representations are defined as follows, respectively.

Table II. Examples of VWKT representations.

| **VagueGeometry Data Type** | **VWKT Representation of a Vague Spatial Object** |
|---|---|
| Vague Point | VAGUEPOINT(POINT(0 0); POINT(1 1)) |
| Vague LineString | VAGUELINESTRING(LINESTRING(0 0, 1 1, 2 2); LINESTRING(2 2, 3 3)) |
| Vague Polygon | VAGUEPOLYGON(POLYGON((1 1, 2 1, 2 2, 1 2, 1 1)); POLYGON((1 1, 0 1, 0 2, 1 2, 1 1))) |
| Vague MultiPoint | VAGUEMULTIPOINT(MULTIPOINT(1 1, 3 3); MULTIPOINT(2 2, 5 5)) |
| Vague MultiLineString | VAGUEMULTILINESTRING(MULTILINESTRING((1 1, 2 2), (3 3, 4 4)); MULTILINESTRING((2 2, 3 3), (5 5, 6 6))) |
| Vague MultiPolygon | VAGUEMULTIPOLYGON(MULTIPOLYGON(((1 1, 0 1, 0 2, 1 2, 1 1)), ((10 10, 0 10, 0 20, 10 20, 10 10))); MULTIPOLYGON(((1 1, 2 1, 2 2, 1 2, 1 1)), ((10 10, 20 10, 20 20, 10 20, 10 10)))) |

Table III. Examples of vGeoJSON representations.

| VagueGeometry Data Type | vGeoJSON Representation of a Vague Spatial Object |
|---|---|
| Vague Point | {"type":"VaguePoint", "crs":{"type":"name", "properties":{"name":"EPSG:4326"}}, "kernel": {"type": "Point", "coordinates": [0,0]}, "conjecture": {"type": "Point", "coordinates": [1,1]}} |
| Vague LineString | {"type":"VagueLineString", "crs":{"type":"name", "properties":{"name":"EPSG:4326"}}, "kernel": {"type": "LineString", "coordinates": [[0,0],[1,1],[2,2]]}, "conjecture": {"type": "LineString", "coordinates": [[2,2],[3,3]]}} |
| Vague Polygon | {"type":"VaguePolygon", "crs":{"type":"name", "properties":{"name":"EPSG:4326"}}, "kernel": {"type": "Polygon", "coordinates": [[[1,1],[2,1],[2,2],[1,2],[1,1]]]}, "conjecture": {"type": "Polygon", "coordinates": [[[1,1],[0,1],[0,2],[1,2],[1,1]]]}} |
| Vague MultiPoint | {"type":"VagueMultiPoint", "crs":{"type":"name", "properties":{"name":"EPSG:4326"}}, "kernel": {"type": "MultiPoint", "coordinates": [[1,1],[3,3]]}, "conjecture": {"type": "MultiPoint", "coordinates": [[2,2],[5,5]]}} |
| Vague MultiLineString | {"type":"VagueMultiLineString", "crs":{"type":"name", "properties":{"name":"EPSG:4326"}}, "kernel": {"type": "MultiLineString", "coordinates": [[[1,1],[2,2]],[[3,3],[4,4]]]}, "conjecture": {"type": "MultiLineString", "coordinates": [[[2,2],[3,3]],[[5,5],[6,6]]]}} |
| Vague MultiPolygon | {"type":"VagueMultiPolygon", "crs":{"type":"name", "properties":{"name":"EPSG:4326"}}, "kernel": {"type": "MultiPolygon", "coordinates": [[[[1,1],[0,1],[0,2],[1,2],[1,1]]],[[[10,10],[0,10],[0,20],[10,20],[10,10]]]]}, "conjecture": {"type": "MultiPolygon", "coordinates": [[[[1,1],[2,1],[2,2],[1,2],[1,1]]],[[[10,10],[20,10],[20,20],[10,20],[10,10]]]]}} |

(i) *VG_AsText*(VagueGeometry *vg*) → text
(ii) *VG_AsVGML*(VagueGeometry *vg*) → text
(iii) *VG_AsVKML*(VagueGeometry *vg*) → text
(iv) *VG_AsVGeoJSON*(VagueGeometry *vg*) → text
(v) *VG_AsVWKB*(VagueGeometry *vg*) → bytea

Vague geometric set operations get two VagueGeometry objects as input and yield another VagueGeometry object. VagueGeometry implements *vague geometric union*, *vague geometric intersection*, and *vague geometric difference*. These operations have the following SQL signatures. Note that we are able to use the vague geometric union operation in (ii) as an aggregation function.

(i) *VG_Union*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueGeometry
(ii) *VG_Union*(VagueGeometry *vg*) → VagueGeometry
(iii) *VG_Intersection*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueGeometry
(iv) *VG_Difference*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueGeometry

Regarding vague topological predicates, VagueGeometry supports *vague contains*, *vague coveredBy*, *vague covers*, *vague crosses*, *vague disjoint*, *vague equals*, *vague inside*, *vague intersects*, *vague meets*, and *vague overlap*. These predicates are based on the three-valued logic, and can return *true*, *false*, or *maybe*. A predicate returns *true* if a relationship *certainly* occurs, *false* if a relationship *certainly not* occurs, and *maybe* if a relationship *occurs to some extent*. To deal with it, VagueGeometry also includes the VagueBool data type. As a result, a VagueBool object can assume *true*, *false*, or *maybe* as value, which correspond to the possible return values of vague topological predicates. In addition, it is possible to use crisp spatial objects as input, which is handled as a vague spatial object containing only the kernel part. The vague topological predicates have the following SQL signatures.

(i) *VG_Contains*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(ii) *VG_CoveredBy*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool

(iii) *VG_Covers*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(iv) *VG_Crosses*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(v) *VG_Disjoint*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(vi) *VG_Equals*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(vii) *VG_Inside*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(viii) *VG_Intersects*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(ix) *VG_Meets*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool
(x) *VG_Overlap*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueBool

Vague numerical operations supported by VagueGeometry are: *vague area of a vague region object*, *vague length of a vague line object*, and *farthest* and *nearest distance between two vague spatial objects*. These operations return two numeric values, which symbolize the minimum and the maximum values of an operation. To deal with it, VagueGeometry also includes the VagueNumeric data type. As a result, a VagueNumeric object is composed of a pair of double values, which correspond to the minimum and the maximum values returned by vague numerical operations. The vague numerical operations have the following SQL signatures.

(i) *VG_Area*(VagueGeometry *r*) → VagueNumeric
(ii) *VG_Length*(VagueGeometry *l*) → VagueNumeric
(iii) *VG_NearestDistance*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueNumeric
(iv) *VG_FarthestDistance*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueNumeric

Finally, regarding type-dependent vague spatial operations, VagueGeometry supports *common border between two vague region objects* and *common points between two vague line objects*. They have the following SQL signatures.

(i) *VG_CommonBorder*(VagueGeometry *r*, VagueGeometry *r*) → VagueGeometry
(ii) *VG_CommonPoints*(VagueGeometry *l*, VagueGeometry *l*) → VagueGeometry

As can be noted, our proposed VagueGeometry implements an expressive set of vague spatial operations, which includes the specification of vague geometric set operations, vague topological predicates, vague numerical operations, and type-dependent vague spatial operations.

### 4.3 SQL Operators

We propose SQL operators to handle VagueBool and VagueNumeric objects, that is, the result of vague topological predicates and vague numerical operations, respectively. For the vague topological predicates, we propose the *logical operators and* (&&), *or* (||), and *not* (!), and the *boolean operators* ∼, ∼∼, and &. Logical operators employ the three-valued logic, which is showed in Table IV. They get VagueBool objects as input and yield another VagueBool object.

Boolean operators are unary operators that get a VagueBool object as input and have true or false as possible return values. Hence, a boolean operator transforms a vague topological predicate into a boolean condition. Let $a$ be a VagueBool object. The operator ∼ yields true if the $a$ is *true* or *maybe*, and false otherwise. The operator ∼∼ yields true if $a$ is *maybe*, and false otherwise. The operator & yields true if $a$ is *true*, and false otherwise.

By using logical and boolean operators, we are able to use them in SQL queries. For instance, we can evaluate if two VagueGeometry objects *maybe* overlap by specifying the condition

Table IV. Truth tables of the three-valued logic. The logical operators *and* (&&) and *or* (||) are commutative and showed in (a) while the operator *not* (!) is showed in (b).

| **VagueBool *a*** | **VagueBool *b*** | ***a* && *b*** | ***a* \|\| *b*** |
|---|---|---|---|
| true | true | true | true |
| true | false | false | true |
| true | maybe | maybe | true |
| false | false | false | false |
| maybe | maybe | maybe | maybe |

(a)

| **VagueBool *a*** | **!*a*** |
|---|---|
| true | false |
| false | true |
| maybe | maybe |

(b)

"$\sim\sim$*VG_Overlap*(*vg1*, *vg2*)". We can even combine logical and boolean operators in an unique condition. For instance, the condition "$\sim$(*VG_Meets*(*vg1*, *vg2*) || *VG_Overlap*(*vg1*, *vg2*))" indicates that the VagueGeometry objects $vg1$ and $vg2$ possibly meet *or* overlap.

Regarding the vague numerical operations, we propose the binary operators $=$ and $\sim$, which get a VagueNumeric object and a numeric value as inputs and yield true or false. Let $v$ be a VagueNumeric object. Let $n$ be a numeric value. The operator $=$ yields true if $n$ is equal to the minimum value of $v$, and false otherwise. The operator $\sim$ yields true if $n$ is between the minimum and the maximum value of $v$, and false otherwise. For instance, users can use this operator to specify the condition "*VG_Area*($r$) $\sim$ 800" in a SQL query to restrict vague region objects in the attribute $r$ that have approximately 800 of area.

### 4.4 Running Example

In this section, we introduce a running example in order to show how to use the VagueGeometry ADT in a spatial application. Figure 3 depicts our running example, which is based on an agriculture environment. This application manages soil textures, farms, lakes, animal routes, and plagues that are represented by the relational table schemas *texture*, *farm*, *lake*, *animal*, and *plague*, respectively. Soil textures, farms, and lakes are represented by complex vague region objects, while animal routes are represented by complex vague line objects and plagues are represented by complex vague point objects. Each table has an attribute *id* as primary key and an attribute *geo* to store its VagueGeometry object. In the following, we show the SQL command to create the table *plague*. Note that the data type of the attribute *geo* is vague multipoint since a plague is represented by a complex vague point object. We specify this by using the name of the VagueGeometry data type between parentheses.

```
CREATE TABLE plague (
  id INTEGER PRIMARY KEY,
  geo VAGUEGEOMETRY(VAGUEMULTIPOINT))
```

We are able to similarly define the other tables, that is, *texture*, *farm*, *lake*, and *animal*. Once we have defined all the tables, we are able to insert VagueGeometry objects by using textual or binary representations (Section 4.1) in SQL insert commands. For instance, in the following SQL command, we insert the vague multipoint object from Table II in the table *plague*.

```
INSERT INTO plague(1, VG_VagueGeomFromText(
        'VAGUEMULTIPOINT(MULTIPOINT(1 1, 3 3); MULTIPOINT(2 2, 5 5))', 4326))
```

Once we have inserted VagueGeometry objects, we are able to handle them in SQL queries. The first query asks for all points in each farm threatened by plagues. This query employs the vague geometric union as an aggregate function on the all plagues. Then, we perform the intersection between this result and each farm. Finally, we show the result by using the VWKT representation.

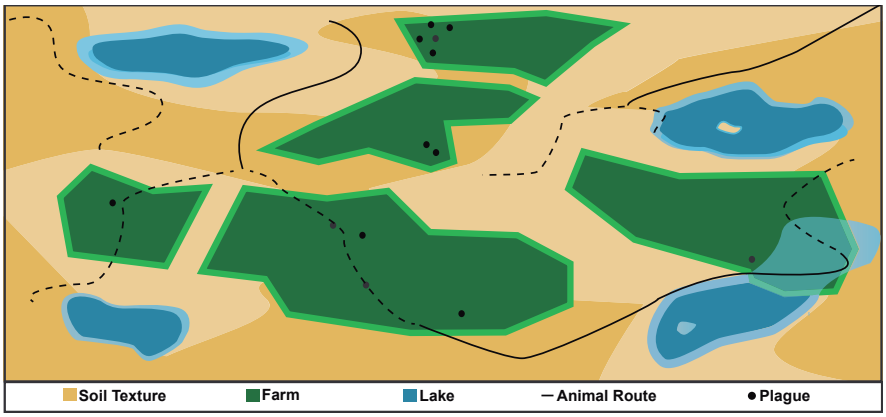



Fig. 3. Running example based on an agriculture environment. Darker colors and solid lines denote kernel parts while lighter colors and dashed lines denote conjecture parts.

```
SELECT VG_AsText(VG_Intersection(U.mp, F.geo)), F.id
FROM (SELECT VG_Union(geo) as mp FROM plague) as U, farm as F
```

The following query returns the common borders and intersection parts between farms and lakes. This means that the common border operation yields lines, while the vague geometric intersection yields regions. The result of these operations are represented by the vGeoJSON representation and indicate critical places for the farms since the water of a lake can invade a farm.

```
SELECT VG_AsVGeoJSON(VG_CommonBorder(F.geo, L.geo)),
       VG_AsVGeoJSON(VG_Intersection(F.geo, L.geo))
FROM farm as F, lake as L
```

The next query asks for the all farms that could be threatened by animal routes since the nearest distance between them is approximately 500. To restrict this, we make use of the operator $\sim$.

```
SELECT F.id, A.id
FROM farm as F, animal as A
WHERE VG_NearestDistance(F.geo, A.geo) ~ 500
```

The next query is a *vague spatial range query* that returns the soil textures that maybe or certainly intersects with a window query $QW$. The window query $QW$ is a VagueGeometry object, which can be composed of one or more objects in the kernel part and one or more objects in the conjecture part. Further, they are rectangular shaped. Aiming to restrict the returning value of the predicate, we employ the boolean operator $\sim$.

```
SELECT T.id
FROM texture as T
WHERE ~VG_Intersects(T.texture, QW)
```

The final query is a *vague spatial join* that returns all pairs of farms and animal routes that certainly overlap, which means that these farms and animals need to be carefully examined. For this query, we employ the boolean operator & to specify that the result be equal to true.

```
SELECT F.id, A.id
FROM farm as F, plague as P, animal as A
WHERE &VG_Contains(F.geo, A.route)
```

## 5. EFFICIENT PROCESSING OF VAGUE TOPOLOGICAL PREDICATES

The implementation of VagueGeometry includes an improvement for the processing of vague topological predicates. The proposed improvement, called *MBRs for Vague Topological Predicates* (*MBRVP*), makes use of *Minimum Boundary Rectangles* (MBR) of the kernel and conjecture parts of vague spatial objects to return the results of vague topological predicates *in some situations*. In these situations, MBRVP can avoid the computation of crisp 9-intersection matrices of a vague topological predicate. As a result, the time to process spatial queries is reduced.

We consider two situations, named *disjointness between MBRs* and *set containment between MBRs*. The disjointness between MBRs encompasses two specific cases, as depicted in Figure 4a. The first case occurs if the MBRs of the union between the kernel and the conjecture of two vague spatial objects are disjoint. The second case occurs if the MBRs of the kernel and the conjecture of two vague spatial objects are disjoint. Note that the second case can happen even when the first case holds. Further, we cannot guarantee that the conjecture parts intersects due to the dead spaces of MBRs, and thus, we cannot return maybe. Let $A$ and $B$ two vague spatial objects. Let also $MBR_o$ be a MBR of a crisp spatial object $o$. We define the *disjointness between MBRs* $S_d(A, B)$ as follows.

$$S_d(A,B) = \begin{cases} true & \text{if } ((MBR_{A_k} \cup MBR_{A_c}) \cap (MBR_{B_k} \cup MBR_{B_c}) = \varnothing) \vee \\ & \quad (MBR_{A_k} \cap MBR_{B_k} = \varnothing \wedge MBR_{A_k} \cap MBR_{B_c} = \varnothing \wedge \\ & \quad MBR_{A_c} \cap MBR_{B_k} = \varnothing \wedge MBR_{A_c} \cap MBR_{B_c} = \varnothing) \\ false & \text{otherwise} \end{cases}$$

By using this definition, we are able to return *true* for vague disjoint if $S_d(A, B) = true$ holds, and return *false* for vague meets, vague intersects, vague overlap, and vague equals if $S_d(A, B) = false$ holds. Otherwise, the predicate is evaluated with the computation of crisp 9-intersection matrices.

Regarding the set containment between MBRs, it also encompasses two specific cases (Figure 4b). The first case occurs if the MBR of the kernel of the first vague spatial object is not inside the MBR of the union between the kernel and the conjecture of the second vague spatial object. The second case occurs if the MBR of the kernel of the first vague spatial object and the MBRs of the kernel and the conjecture of the second vague spatial object are disjoint. Let $A$ and $B$ be two vague spatial objects. Let also $MBR_o$ be a MBR of a crisp spatial object $o$. We define the *set containment between MBRs* $S_{sc}(A, B)$ as follows.

$$S_{sc}(A,B) = \begin{cases} true & \text{if } (MBR_{A_k} \not\subseteq (MBR_{B_k} \cup MBR_{B_c})) \vee \\ & \quad (MBR_{A_k} \cap MBR_{B_k} = \varnothing \wedge MBR_{A_k} \cap MBR_{B_c} = \varnothing) \\ false & \text{otherwise} \end{cases}$$

By using this definition, we are able to return *false* for vague inside and vague coveredBy if $S_{sc}(A, B) = true$ holds. Otherwise, the respective predicate is evaluated. Similarly, if $S_{sc}(B, A) = true$ holds, then we can return *false* for vague contains and vague covers, and evaluate the respective predicates otherwise.

## 6. PERFORMANCE EVALUATION

The advantages of our proposed solutions (that is, the VagueGeometry ADT and the MBRVP improvement) were analyzed through experimental tests that process spatial queries with vague topological predicates. We analyze topological predicates since they incur high costs of processing and they are very common in spatial applications. Section 6.1 introduces the experimental setup used in the performance evaluation. Section 6.2 and Section 6.3 discusses the performance results of execution of vague spatial range queries and vague spatial joins, respectively.


Fig. 4. Examples of the situations where the MBRVP improvement is applied: (a) the disjointness between MBRs and (b) the set containment between MBRs. Black regions represent the kernel, while gray regions represent the conjecture.

### 6.1 Experimental Setup

We used the data sets $D_1$ and $D_2$. $D_1$ was composed of 100,000 vague region objects synthetically generated as follows. First, we constructed a Voronoi diagram on of 200,000 crisp points randomly generated, which produced the same number of crisp regions. Second, for each crisp region, we added more points to increase its complexity. As a result, each crisp region was formed by averagely 313 points. Third, we created pairs of crisp regions that were disjoint or adjacent. Each pair was created by selecting randomly a crisp region and then by selecting the nearest crisp region that was disjoint or adjacent to the first one. We randomly assigned a crisp region as the kernel and the other crisp region as the conjecture, which formed a vague region object. After creating a pair, we discarded the used regions, such that these regions were not used to create another pair. In the total, we generated 100,000 vague region objects in order to compute vague spatial range queries. The same process was used to create the data set $D_2$; but, instead of 100,000 vague region objects, two sets of 10,000 vague region objects were created in order to compute vague spatial joins between them.

We computed two type of spatial queries, vague spatial range queries and vague spatial joins. For the vague spatial range queries, we considered the following predicates: *vague disjoint*, *vague overlap*, *vague inside*, *vague intersects*, *vague coveredBy*, and *vague meets*. The workload was composed of 100 vague spatial range queries for each vague topological predicate. We also defined a query window for each vague spatial range query, which was composed of a vague region object that had the rectangular format for the kernel and the conjecture. Therefore, we randomly generated 100 different query windows. For the vague spatial joins, we considered the following predicates: *vague disjoint*, *vague overlap*, *vague inside*, *vague coveredBy*, and *vague meets*.

We defined the following configurations: (i) *baseline* that used current functionalities provided by the PostgreSQL with the PostGIS spatial extension; (ii) *VagueGeometry* that used the proposed VagueGeometry ADT without improvements; and (iii) *VagueGeometry+* that used VagueGeometry empowered with the proposed MBRVP improvement. For the *baseline* configuration, we implemented vague topological predicates by using the Procedural Language/PostgreSQL (PL/pgSQL), which had "TRUE", "FALSE", or "MAYBE" as possible return textual values. For their use, we stored the kernel and the conjecture of each vague spatial object in separated columns in a relational table.

Note that we did not employ the approaches surveyed in Section 2 here due to the following limitations. While the approaches proposed in Zinn et al. [2007], Kraipeerapun [2004], Dilo et al. [2006], and Carniel et al. [2015a] *do not provide support for vague topological predicates*, the approaches described in Pauly and Schneider [2008] and Pauly and Schneider [2010] are *specifically implemented in Oracle*, which has license restrictions. Further, we used PostgreSQL in the performance tests to isolate the effects of the DBMS, and thus providing a fair comparison.

Table V depicts the SQL templates of the vague spatial range queries and vague spatial joins used in the three configurations. Consider the template for the *baseline*. *baselineTable*, *baselineTable1*, and *baselineTable2* are tables composed of three attributes: (i) *id* that is the primary key, (ii) *kernel_geo* that represents the kernel of a vague region object, and (iii) *conjecture_geo* that represents the conjecture of a vague region object. Further, $R$ is the textual return value that may contain

Table V. SQL templates of the vague spatial range queries and vague spatial joins.

| Configuration | SQL Template | |
|---|---|---|
| | **Vague Spatial Range Query** | **Vague Spatial Join** |
| *baseline* | `SELECT id FROM baselineTable WHERE R = P(kernel_geo, conjecture_geo, QWk, QWc)` | `SELECT A.id, B.id FROM baselineTable1 as A, baselineTable2 as B WHERE R = P(A.kernel_geo, A.conjecture_geo, B.kernel_geo, B.conjecture_geo)` |
| *VagueGeometry* *VagueGeometry+* | `SELECT id FROM vaguegeom WHERE O P(vg, QW)` | `SELECT A.id, B.id FROM vaguegeom1 as A, vaguegeom2 as B WHERE O P(A.vg, B.vg)` |

"TRUE", "FALSE", or "MAYBE" and $P$ is the vague topological predicate. For vague spatial range queries, $QW$ is the query window. Regarding the *VagueGeometry* and the *VagueGeometry+* configurations, *vaguegeom*, *vaguegeom1*, and *vaguegeom2* are tables that stored vague region objects in the attribute *vg* by using our proposed VagueGeometry. In addition, $O$ corresponds to the use of the SQL operators introduced in Section 4.3. This means that the operator $\sim\sim$ was used to specify that $P$ returned maybe, the operator & was used to specify that $P$ returned true, and the combination of the operator & with the operator ! (that is, &!) was used to specify that $P$ returned false. Note that the SQL templates are equivalent for each type of query, that is, they generate the same result, but using the specific functionalities provided by the corresponding configurations.

The experiments were conducted on a computer with an Intel® Core™ i7-4770 processor with frequency of 3.40GHz, 2 TB SATA hard drive with 7200 RPM, and 32 GB of main memory. The operating system was CentOS 6.5 with Kernel Version 2.6.32-431.el6.x86 64. We employed PostgreSQL 9.3.3, PostGIS 2.2.0, and GEOS 3.4.2.

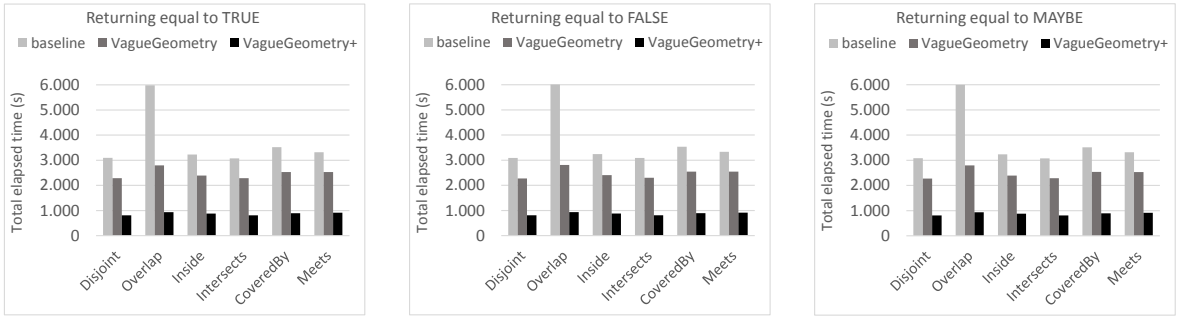
We collected the elapsed time in seconds. In detail, we executed 100 vague spatial range queries for each vague topological predicate and each value of return. Further, we executed each vague spatial range query and vague spatial join 10 times and calculated their average elapsed time. Furthermore, we performed the tests locally to avoid network latency and we flushed the system cache after the execution of each query.

## 6.2 Execution of Vague Spatial Range Queries

This experimental evaluation was conducted on data set $D_1$ and its results are reported in Figure 5. For each configuration and each return value of the three-valued logic (that is, true, false, and maybe), we gathered similar elapsed times for processing the spatial queries. This means that the performance results showed a similar complexity for each return value.

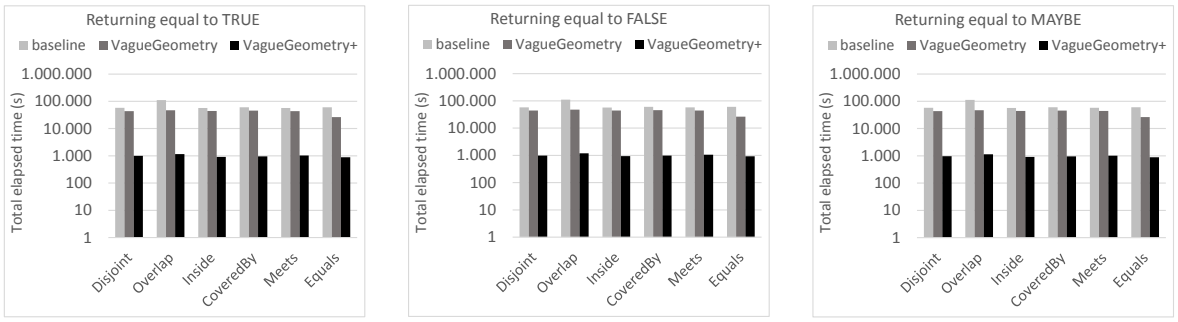
Clearly, the performance of the *VagueGeometry* configuration overcame the *baseline* configuration. In fact, the internal structures of the VagueGeometry were more efficient than the definition of vague topological predicates by using current functionalities of the spatial DBMS. The performance gain imposed by the *VagueGeometry* configuration over the *baseline* configuration ranged from 23% up to 53%, where the performance gain is calculated as the percentage that determines how much more efficient one configuration was than another configuration.

Despite the expressive performance gains obtained by the *VagueGeometry* configuration, we gathered yet better results with the improvement proposed in Section 5. The *VagueGeometry+* configuration leaded to a performance gain that ranged from 72% up to 84% if compared with the *baseline* configuration. Further, compared to the *VagueGeometry* configuration, the *VagueGeometry+* configuration provided a performance gain that ranged from 63% up to 66%. This means that the use of the MBRVP improvement drastically reduced the necessity of processing crisp 9-intersection matrices in vague topological predicates.




(a) (b) (c)

Fig. 5. Performance results of the execution of vague spatial range queries for each vague topological predicate considering the returning values of true (a), false (b), and maybe (c).




(a) (b) (c)

Fig. 6. Performance results of the execution of vague spatial joins for each vague topological predicate considering the returning values of true (a), false (b), and maybe (c).

Regarding storage space, the *baseline* configuration required 961 MB, the *VagueGeometry* configuration required 960 MB, and the *VagueGeometry+* configuration required 963 MB. We can conclude that the storage cost were almost the same. In addition, the storage of the MBRs of the kernel and the conjecture of each VagueGeometry object in the *VagueGeometry+* configuration did not introduce overhead in the execution of the spatial queries.

### 6.3 Execution of Vague Spatial Joins

This experimental evaluation was conducted on data set $D_2$ that stored two sets of 10,000 vague region objects. Figure 6 depicts the performance results. Similarly to the discussed results in Section 6.2, we obtained similar elapsed times for processing the spatial queries for each return value (that is, true, false, and maybe).

Again, the performance of the *VagueGeometry* configuration greatly overcame the *baseline* configuration. The performance gain imposed by the *VagueGeometry* configuration over the *baseline* configuration ranged from 21% up to 58%. The performance gain is increased when *VagueGeometry+* is considered. By using the improvement proposed in Section 5, we obtained an expressive performance gain of at least 98% if compared with the *baseline* configuration. If we compare the *VagueGeometry* and *VagueGeometry+*, the *VagueGeometry+* configuration provided a performance gain that ranged from 96% up to 97%. This means that the use of the MBRVP improvement is highly recommended in the vague spatial join processing since it drastically avoided the computation of crisp 9-intersection matrices in vague topological predicates. Moreover, each configuration required approx-

imately 194 MB to store the data set $D_2$, which indicates that *VagueGeometry+* did not introduce overhead in the storage and it guaranteed the best obtained results.

## 7. CONCLUSIONS AND FUTURE WORK

In this article, we propose VagueGeometry, a novel abstract data type to handle vague spatial objects in the PostgreSQL with the PostGIS spatial extension. VagueGeometry empowers the management of spatial applications by offering textual and binary representations for vague spatial objects and by providing an expressive set of spatial operations, including vague geometric set operations, vague topological predicates, vague numerical operations, and type-dependent vague spatial operations. As facilities, VagueGeometry introduces SQL operators to handle results of vague topological predicates and vague numerical operations. We also introduce MBRVP, an improvement to VagueGeometry to speed up the performance of spatial queries to process vague topological predicates.

Comparisons of VagueGeometry with current functionalities available on PostgreSQL showed that VagueGeometry provided better performance results for spatial queries with vague topological predicates. The performance gain of VagueGeometry varied from 21% up to 58%. Empowered with MBRVP, VagueGeometry provided even better results, which varied from 72% up to 98%. Future work will deal with the extension of VagueGeometry to allow the use of index structures and the storage of statistical data about vague spatial objects to be used by the PostgreSQL query optimizer.

## REFERENCES


Bejaoui, L., Pinet, F., Schneider, M., and Bédard, Y. OCL for formal modelling of topological constraints involving regions with broad boundaries. *GeoInformatica* 14 (3): 353–378, 2010.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. An abstract data type to handle vague spatial objects based on the fuzzy model. In *Proceedings of the Brazilian Symposium on GeoInformatics*. Campos do Jordão, SP, Brazil, pp. 210–221, 2015a.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. Embedding vague spatial objects into spatial databases using the vaguegeometry abstract data type. In *Proceedings of the Brazilian Symposium on GeoInformatics*. Campos do Jordão, SP, Brazil, pp. 233–244, 2015b.

Carniel, A. C., Schneider, M., Ciferri, R. R., and Ciferri, C. D. A. Modeling fuzzy topological predicates for fuzzy regions. In *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*. New York, NY, USA, pp. 529–532, 2014.

Clementini, E. and Di Felice, P. Approximate topological relations. *International Journal of Approximate Reasoning* 16 (2): 173–204, 1997.

Dilo, A., Bos, P., Kraipeerapun, P., and de By, R. Storage and manipulation of vague spatial objects using existing GIS functionality. In *Flexible Databases Supporting Imprecision and Uncertainty*, G. Bordogna and G. Psaila (Eds.). Vol. 203. Springer Berlin Heidelberg, pp. 293–321, 2006.

Dilo, A., de By, R. A., and Stein, A. A system of types and operators for handling vague spatial objects. *International Journal of Geographical Information Science* 21 (4): 397–426, 2007.

Kraipeerapun, P. *Implementation of vague spatial objects*. M.S. thesis, International Institute for Geo-Information Science and Earth Observation, 2004.

Li, R., Bhanu, B., Ravishankar, C., Kurth, M., and Ni, J. Uncertain spatial data handling: Modeling, indexing and query. *Computers & Geosciences* 33 (1): 42–61, 2007.

Pauly, A. and Schneider, M. Querying vague spatial objects in databases with VASA. In A. Stein, W. Shi, and W. Bijker (Eds.), *Quality Aspects in Spatial Data Mining*. CRC Press, USA, pp. 3–14, 2008.

Pauly, A. and Schneider, M. VASA: An algebra for vague spatial data in databases. *Information Systems* 35 (1): 111–138, 2010.

Schneider, M. and Behr, T. Topological relationships between complex spatial objects. *ACM Transactions on Database Systems* 31 (1): 39–81, 2006.

Siqueira, T. L., Ciferri, C. D. A., Times, V. C., and Ciferri, R. R. Modeling vague spatial data warehouses using the VSCube conceptual model. *Geoinformatica* 18 (2): 313–356, 2014.

Zinn, D., Bosch, J., and Gertz, M. Modeling and querying vague spatial objects using shapelets. In *Proceedings of the International Conference on Very Large Data Bases*. Vienna, Austria, pp. 567–578, 2007.