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 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. We also propose an improvement of VagueGeometry to deal efficiently with the processing of vague topological predicates. 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* that 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* [Bennett 2010]. Several real-world phenomena are characterized by spatial vagueness. An example refers to species habitats that have uncertain locations. In this case, it could be impossible to determine the exact location of species due to their constant positional movement, such as feeding and reproduction activities in different locations. Another example is related to rivers and lakes that have different volumes of water due to different precipitation levels. Further, air polluted areas are also phenomena characterized by spatial vagueness. In this case, pollution may by spread to different boundaries, according to different streams of air. In general, there are a spatial extent that *certainly* belongs to the real-world phenomena (that is, the *kernel*) and a spatial extent that *maybe* or *possibly* belongs to the real-world phenomena (that is, the *conjecture*) [Siqueira et al. 2014].

Several approaches 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* [Cohn and Gotts 1996; 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 for *vague points*, *vague lines*, and *vague regions*. They also introduce vague spatial operations to handle them, such as, *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 species habitats and air polluted areas. Species habitats and air polluted areas are represented by vague regions. By using such data, a user can ask the following query: "Find all polluted areas of rivers that *possibly* overlap with habitats of species". Here, *possibly overlap* means that the overlap occurs to some extent, that is, 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, a 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 propose VagueGeometry, a novel ADT 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 [Cohn and Gotts 1996; 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, has closure properties of vague spatial operations resulting in valid results, and introduces an expressive set of operations, such as vague geometric set operations, vague topological predicates, and vague numerical operations.

VagueGeometry greatly overcomes the aforementioned limitations. It has the following characteristics:

—It offers textual and binary representations for vague spatial objects, which allow users to insert and retrieve vague spatial objects. 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 directly access a spatial database con-

taining 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 improvement to process efficiently vague topological predicates in spatial queries, such as vague range queries and vague spatial joins.

In this article, we extend our previous work [Carniel et al. 2015b] by including several novel topics. First, we describe vague spatial objects by using graphical representations. Second, we include several examples of textual representations of VagueGeometry. Third, we introduce type-dependent vague spatial operations, such as common border between two vague region objects. Fourth, we define SQL signatures for all VagueGeometry operations. Fifth, we present a running example that shows how to use the VagueGeometry in SQL queries. Finally, we include an additional experimental evaluation that analyzes the performance of VagueGeometry in the query processing of vague spatial joins.

This article is organized as follows. Section 2 surveys related work. Section 3 summarizes the Vague Spatial Algebra. Section 4 introduces the proposed VagueGeometry ADT, while Section 5 shows how to use VagueGeometry in SQL queries, Section 6 describes performance tests. Finally, Section 7 concludes the article.

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]. They mainly differ from our proposed VagueGeometry in the practicable applicability of the user to handle vague spatial objects in spatial queries. Table I depicts a detailed comparison among the functionalities provided by related work and VagueGeometry (last column), considering: (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. As can be noted, the proposed VagueGeometry is the only approach that provides all functionalities. The main highlights about the comparison are described as follows.

Textual and binary representations are important functionalities that allow users to easily insert and retrieve vague spatial objects. They also permit interoperability between spatial applications. Although Pauly and Schneider [2008; 2010] do not include these representations, they allow the definition of vague spatial objects by using extensive terminal command lines. Kraipeerapun [2004] and Dilo et al. [2006] support options to represent vague spatial objects by using files in the format of the GRASS GIS[1], which is a widely used open source GIS. As can be noted, these files depend on a specific system. Further, users are not able to understand them.

Providing an expressive set of vague spatial operations composed by vague geometric set operations, vague topological predicates, and vague numerical operations, applications are able to improve the management of vague spatial objects. The vague geometric set operations refer to vague geometric union, vague geometric intersection, and vague geometric difference between vague points, lines, and regions. In this case, Kraipeerapun [2004] and Dilo et al. [2006] do not implement the vague geometric difference between vague lines.

Offering operators in the SQL language is an important functionality since it allows users to intuitively handle results of spatial operations in SQL queries. Although Pauly and Schneider [2008; 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).

[1]http://grass.osgeo.org/

Table I. Comparisons of VagueGeometry with related work.

Functionality	Approaches				
	Kraipeerapun [2004]; Dilo et al. [2006]	Zinn et al. [2007]	Pauly and Schneider [2008; 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

Finally, implementing vague spatial objects in a spatial DBMS allows users to apply them during the development of their applications. Oracle has license restrictions and it can not be properly compared and evaluated in our performance tests. On the other hand, VagueGeometry is an open source implementation based on the PostgreSQL DBMS with the PostGIS spatial extension that allows to be extended with additional functionalities.

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. A simple vague spatial object consists of a single component in both kernel and conjecture, while a complex vague spatial object consists of multiple connected components in both kernel and conjecture. 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. Formally, let $\alpha \in \{\textit{crisp point}, \textit{crisp line}, \textit{crisp region}\}$. 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 $\textit{disjoint}(o_k, o_c) \vee \textit{meet}(o_k, o_c)$ holds. The kernel and conjecture of o are symbolized by o_k and o_c, respectively.

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 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 by Pauly and Schneider [2010].

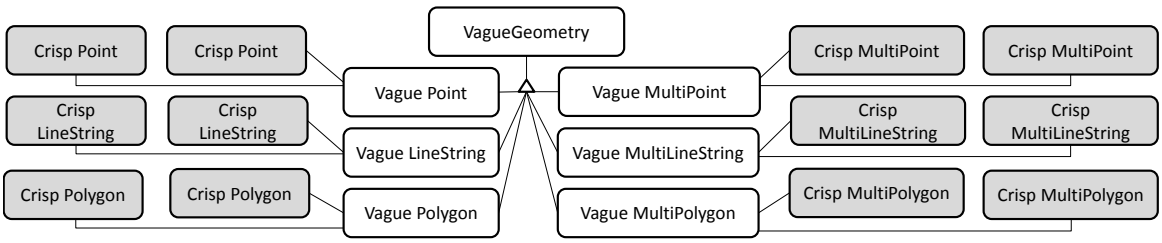


Fig. 1. The vague spatial data types of VagueGeometry.

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 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 OGC specifications[3]. A detailed documentation of VagueGeometry is available at `http://gbd.dc.ufscar.br/vaguegeometry/`.

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 presents an improvement of VagueGeometry to deal efficiently with the processing of vague topological predicates.

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 shown in gray in Figure 1. Figure 2 shows examples of VagueGeometry objects, that is, instances of vague spatial data types.

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 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

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

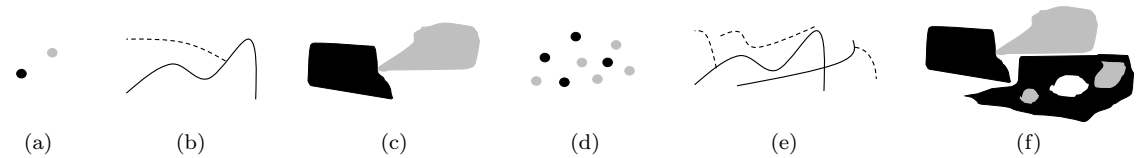


Fig. 2. Examples of VagueGeometry objects according to its data type (Figure 1): a vague point object (a), a vague linestring object (b), a vague polygon object (c), a vague multipoint object (d), a vague multilinestring object (e), and a vague multipolygon object (f). Black objects and solid lines denote kernel parts while gray objects and dashed lines denote conjecture parts.

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 a 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, a SRID is specified by adding the attribute *srsName* in the tag that indicates the VagueGeometry data type. This attribute is the same attribute *srsName* of the GML representation. Finally, for the vGeoJSON representation, a SRID is specified by adding the attribute *crs*, which corresponds to the attribute *crs* of the GeoJSON format.

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 union between 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 a unique representation. For each VagueGeometry data type (Figure 1), Tables II and III show examples of VWKT and vGeoJSON

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]]]]}}

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 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 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 that return VWKT, VGML, VKML, vGeoJSON, and VWKB representations of a vague spatial object are defined as follows, respectively:

(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*. Note that we are able to use the vague geometric union as an aggregate function (operation (ii) below), which computes the union of a set of objects stored in rows of data. These operations have the following SQL signatures:

(i) *VG_ Union*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueGeometry
(ii) *VG_ Union*(VagueGeometry set *vg_field*) → VagueGeometry
(iii) *VG_ Intersection*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueGeometry
(iv) *VG_ Difference*(VagueGeometry *vg1*, VagueGeometry *vg2*) → VagueGeometry

Regarding the 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 does not* occur, and *maybe* if a relationship *possibly* occurs. To deal with this three-valued logic, 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

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

(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, VagueGeometry supports the following type-dependent vague spatial operations: *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 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 shown 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.

(a) (b)

Fig. 3. 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.

By using logical and boolean operators, we are able to restrict the returning values of vague topological predicates in SQL queries. For instance, we can evaluate if two VagueGeometry objects *possibly* overlap by specifying the condition "$\sim\sim$*VG_ Overlap*(*vg1*, *vg2*)". We can even combine logical and boolean operators in a 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 Efficient Processing of Vague Topological Predicates

In this section, we propose an improvement of VagueGeometry to deal efficiently with the processing of vague topological predicates. The proposed improvement, called *MBRs for Vague Topological Predicates* (or *MBRVP* for short), 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 3a. The first case is whether MBRs of the union between the kernel and the conjecture of two vague spatial objects are disjoint. The second case is whether MBRs of the kernel and the conjecture of two vague spatial objects are disjoint. Note that the second case can happen even if the first case holds. Further, since we cannot guarantee that the conjecture parts intersect due to the dead spaces of MBRs, 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:

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

By using this definition, we are able to return *true* for vague disjoint if $S_d(A, B) = \mathit{true}$ holds, and return *false* for vague meets, vague intersects, vague overlap, and vague equals if $S_d(A, B) = \mathit{false}$ 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 3b). The first case is whether 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 is whether the MBR of the kernel of the first vague spatial object and 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.

5. EMPLOYING VAGUEGEOMETRY IN A SPATIAL APPLICATION

In this section, we show how to employ the VagueGeometry ADT in a spatial application. Figure 4 depicts our example of application, 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 these tables, we are able to insert VagueGeometry objects by using textual or binary

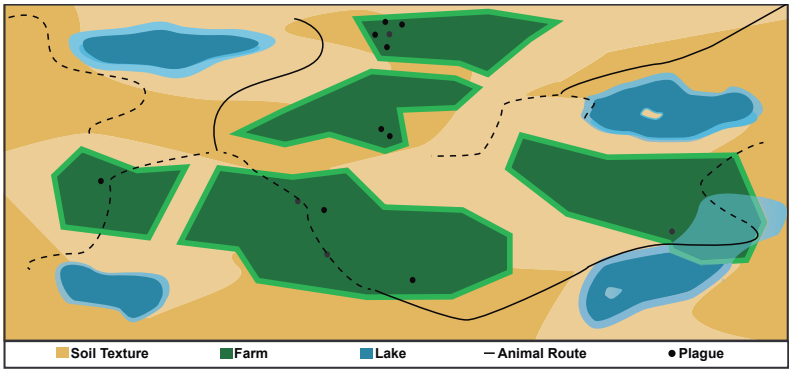


Fig. 4. Example based on an agriculture environment. Darker colors and solid lines denote kernel parts while lighter colors and dashed lines denote conjecture parts.

representations (Section 4.1) in SQL INSERT INTO commands. For instance, in the following SQL command, we insert the vague multipoint object from Table II in the *plague* table.

```
INSERT INTO plague VALUES(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 all plagues. Then, we perform the intersection between this result and each farm. Finally, we show the result by using VWKT representation.

```
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 farms since the water of a lake can invade them.

```
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 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 possibly or certainly intersect a window query QW. The window query QW is a VagueGeometry object 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. This means that these farms and animals need to be carefully examined. For this query, we employ the boolean operator &.

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

6. PERFORMANCE EVALUATION

The advantages of VagueGeometry were analyzed through experimental tests that processed spatial queries with vague topological predicates. We analyzed 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 for processing vague spatial range queries and vague spatial joins, respectively.

6.1 Experimental Setup

To the best of our knowledge, there is no a consensus in the spatial database community in how to extract vague spatial objects from real-world phenomena. Due to this and aiming to explore the geometric complexity of vague regions, we considered two synthetic datasets D_1 and D_2. D_1 was composed of 100,000 vague region objects generated as follows. First, we constructed a Voronoi diagram of 200,000 crisp points randomly generated, which produced the same number of crisp regions. Second, for each crisp region, we added 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. To create a pair, we randomly selected a crisp region and then we selected the nearest crisp region that was disjoint or adjacent to the first one. We randomly assigned a crisp region as kernel and the other crisp region as conjecture in order to form 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 to be stored in D_1. The same process was used to create the dataset D_2; but, instead of 100,000 vague region objects, two sets of 10,000 vague region objects were created.

We computed two types 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) *VGwithoutMBRVP* that used VagueGeometry without the MBRVP improvement described in Section 4.4; and (iii) *VGwithMBRVP* that used VagueGeometry improved with MBRVP. For *baseline*, 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. We defined two configurations to our proposal due to the following reasons. First, we are able to show that VagueGeometry improves query processing even if its implementation does not encompass MBRVP. Second, we are also able to show that VagueGeometry extended with MBRVP always produces better performance results. Third, we can isolate the improvement introduced by MBRVP in the analysis of query processing, which represents an important research result.

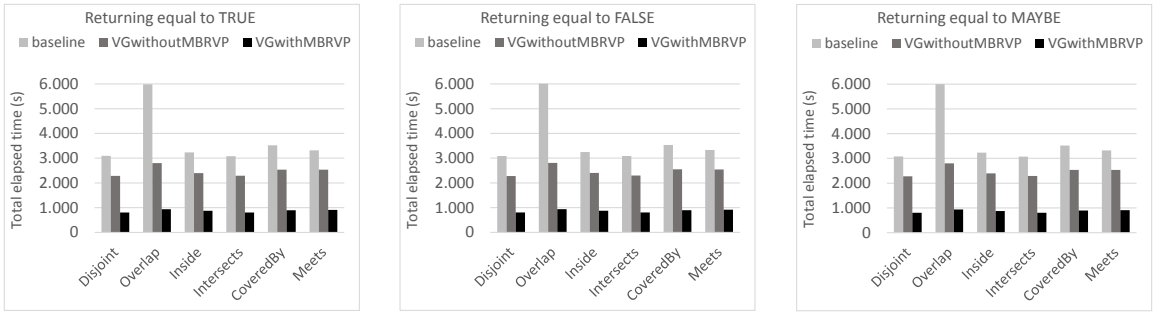
Note that we did not employ the approaches surveyed in Section 2 here due to the following limitations. While the approaches proposed by Zinn et al. [2007], Kraipeerapun [2004], Dilo et al. [2006], and Carniel et al. [2015a] *do not provide support for vague topological predicates*, the approaches described by Pauly and Schneider [2008; 2010][4] 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 *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 "TRUE", "FALSE", or "MAYBE" and P is the vague topological predicate. For vague spatial range queries, QW is the query window.

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

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, QW_k, QW_c)`	`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)`
VGwithoutMBRVP *VGwithMBRVP*	`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)`



(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).

Regarding *VGwithoutMBRVP* and *VGwithMBRVP*, *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 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. It is worth noticing that the query optimizer generated the same query execution plans for each type of query since they are simple queries and spatial indexes were not employed in *baseline*.

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 dataset 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.

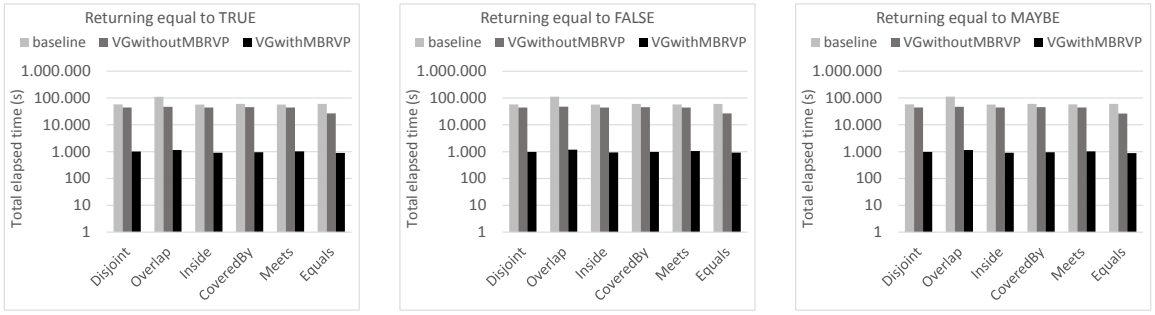


Fig. 6. Performance results of the execution of vague spatial joins for each vague topological predicate considering the returning values of *true* (a), *false* (b), and *maybe* (c).

Clearly, the performance of *VGwithoutMBRVP* overcame *baseline*. These results demonstrate that implementing VagueGeometry by using the extensibility provided by PostgreSQL guaranteed more efficiency than the implementation of vague topological predicates by using the PL/pgSQL functions together with current functionalities of the PostGIS spatial extension. The performance gain imposed by *VGwithoutMBRVP* over *baseline* 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 *VGwithoutMBRVP*, we gathered yet better results with the improvement proposed in Section 4.4. *VGwithMBRVP* leaded to a performance gain that ranged from 72% up to 84% if compared with *baseline*. Further, *VGwithMBRVP* provided a performance gain against *VGwithoutMBRVP* that ranged from 63% up to 66%. The results demonstrate that the MBRVP improvement drastically reduced the time spent to process crisp 9-intersection matrices in vague topological predicates.

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

6.3 Execution of Vague Spatial Joins

This experimental evaluation was conducted on dataset D_2 that stored two sets of 10,000 vague region objects. Figure 6 depicts the performance results. We also obtained here similar elapsed times for processing the spatial queries for each return value (that is, *true*, *false*, and *maybe*).

Again, *VGwithoutMBRVP* greatly overcame *baseline*, providing performance gains that ranged from 21% up to 58%. Further, the improvement proposed in Section 4.4 guaranteed to *VGwithMBRVP* a performance gain of at least 98% over *baseline*. Comparing *VGwithoutMBRVP* and *VGwithMBRVP*, the latter provided a performance gain that ranged from 96% up to 97% over the former. Moreover, each configuration required approximately 194 MB to store the data set D_2, which indicates that *VGwithMBRVP* did not introduce overhead in the storage and it guaranteed the best obtained results.

Due to the performance results to execute vague spatial range queries and vague spatial joins, the implementation of VagueGeometry available at `http://gbd.dc.ufscar.br/vaguegeometry/` includes the MBRVP improvement by default.

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%. Improved with MBRVP, VagueGeometry provided even better results, which varied from 72% up to 98%.

Future work will analyze the performance of VagueGeometry varying query selectivity and dataset sizes. Further, we aim to extend VagueGeometry in order to provide index structures that minimize the processing of crisp 9-intersection matrices since such functionality can negatively affect the time processing of vague spatial queries. Finally, we will also store statistical data about vague spatial objects to be used by the PostgreSQL query optimizer.

Acknowledgments. This work has been supported by the Brazilian federal research agencies CAPES and CNPq as well as by the São Paulo Research Foundation (FAPESP). A. C. Carniel has been supported by the grants #2012/12299-8 and #2015/26687-8, FAPESP. R. R. Ciferri has been supported by the grant #311868/2015-0, CNPq. C. D. A. Ciferri has been supported by the grant #2016/04990-3, FAPESP.

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.

Bennett, B. Spatial Vagueness. In R. Jeansoulin, O. Papini, H. Prade, and S. Schockaert (Eds.), *Methods for Handling Imperfect Spatial Information*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 15–47, 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, 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, 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 SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, USA, pp. 529–532, 2014.

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

Cohn, A. G. and Gotts, N. M. The 'Egg-Yolk' Representation of Regions with Indeterminate Boundaries. In P. A. Burrough and A. U. Frank (Eds.), *Geographic Objects with Indeterminate Boundaries*. Taylor & Francis, Great Britain, pp. 171–187, 1996.

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, The Netherlands, 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.