Open GIS Consortium, Inc.

OpenGIS® Simple Features Specification

For SQL

Revision 1.1

OpenGIS Project Document 99-049
Release Date: May 5, 1999

WARNING: The Open GIS Consortium (OGC) releases this specification to the public without warranty. It is subject to change without notice. This specification is currently under active revision by the OGC Technical Committee

Requests for clarification and/or revision can be made by contacting the OGC at revisions@opengis.org.
Table of Contents

0  PREFACE ......................................................................................................................... 0-1
  0.1  SUBMITTING COMPANIES ....................................................................................... 0-1
  0.2  SUBMISSION CONTACT POINTS .............................................................................. 0-1
  0.3  DOCUMENT CONVENTIONS .................................................................................... 0-2
  0.4  REVISION HISTORY .................................................................................................. 0-2
  0.5  EDITORIAL NOTES ..................................................................................................... 0-3
1  OVERVIEW .................................................................................................................... 1-1
  1.1  APPROACH ............................................................................................................... 1-1
2  ARCHITECTURE .............................................................................................................. 2-1
  2.1  GEOMETRY OBJECT MODEL ................................................................................. 2-1
     2.1.1  Geometry ........................................................................................................... 2-2
     2.1.2  Geometry Collection ......................................................................................... 2-4
     2.1.3  Point ................................................................................................................... 2-4
     2.1.4  MultiPoint ......................................................................................................... 2-4
     2.1.5  Curve .................................................................................................................. 2-5
     2.1.6  LineString, Line, LinearRing .............................................................................. 2-5
     2.1.7  MultiCurve ........................................................................................................ 2-6
     2.1.8  MultiLineString .................................................................................................. 2-7
     2.1.9  Surface ............................................................................................................... 2-7
     2.1.10 Polygon .............................................................................................................. 2-8
     2.1.11 MultiSurface ..................................................................................................... 2-10
     2.1.12 MultiPolygon ................................................................................................... 2-10
     2.1.13 Relational Operators ......................................................................................... 2-12
  2.2  ARCHITECTURE—SQL92 IMPLEMENTATION OF FEATURE TABLES ................. 2-20
     2.2.1  Feature Table Metadata Views ......................................................................... 2-21
     2.2.2  Geometry Columns Metadata Views ............................................................. 2-21
     2.2.3  Spatial Reference System Information Views .................................................... 2-21
     2.2.4  Feature Tables and Views .................................................................................. 2-22
     2.2.5  Geometry and Geometric Element Views ........................................................ 2-22
     2.2.6  Notes on SQL92 data types ............................................................................ 2-23
     2.2.7  Notes on ODBC Access to Geometry Values stored in Binary form .................. 2-24
  2.3  ARCHITECTURE—SQL92 WITH GEOMETRY TYPES IMPLEMENTATION OF FEATURE TABLES ... 2-24
     2.3.1  Feature Table Metadata Views ......................................................................... 2-24
     2.3.2  Geometry Columns Metadata Views ............................................................. 2-24
     2.3.3  Spatial Reference System Information Views .................................................... 2-24
     2.3.4  Feature Tables and Views .................................................................................. 2-25
0 Preface

0.1 Submitting Companies

The following companies submitted this implementation specification in response to the OGC Request 1, Open Geodata Model Working Group, A Request for Proposals: OpenGIS Features (OpenGIS Project Document Number 96-021):

- Environmental Systems Research Institute, Inc.
- IBM Corporation.
- Informix Software, Inc.
- MapInfo Corporation.
- Oracle Corporation.

0.2 Submission Contact Points

All questions about the joint submission should be directed to:

David Beddoe
ESRI–Washington DC.
2070 Chain Bridge Road, Suite 180
Vienna, VA 22182
Phone: (703) 506-9515
Email: dbeddoe@esri.com

Paul Cotton
IBM Corporation
1150 Eglinton Ave.
Toronto, Ontario M3C 1H7
Canada
cotton@vnet.ibm.com

Robert Uleman
Informix Software, Inc.
300 Lakeside Drive, Suite 2700
Oakland, CA 94612
uleman@informix.com
0.3 Document Conventions

The Courier New font has been used to indicate SQL or other code segments.

0.4 Revision History

Revision 1.0 includes the following changes from Revision 0:

- Replaced the term ‘byte stream’ with ‘representation’. The source for this change was proposal #1 from Revision Request 97-402.

- Made several minor corrections concerning typographical errors, fixed the definition of the GEOMETRY_COLUMNS table to remove foreign key constraints that accessed INFORMATION_SCHEMA, fixed several functions to replace the Boolean return values with integer returns, and made a clarification on the example in section 3.1.3. The source for these changes was Revision Request 97-403.

Revision 1.1 includes the following changes from Revision 1.0:

- Function name consistency

- Consistent use of UML notation for section 2 (Architecture)

- 18 character function name limits

- Explicit specification of ETYPe codes for SQL numeric representation

- Clarify handling of mixed spatial references in SQL functions

- Fix errors in diagrams

- Misc. typographical errors

- Remove Spatial Reference Data not present in EPSG 1.3 specification

When problems were identified, such as inconsistent function names or function names that exceed 18 characters, the correction was made to conform to the SQL/MM specification.
1 Overview

The purpose of this specification is to define a standard SQL schema that supports storage, retrieval, query and update of simple geospatial feature collections via the ODBC API. A simple feature is defined by the OpenGIS Abstract specification to have both spatial and non-spatial attributes. Spatial attributes are geometry valued, and simple features are based on 2D geometry with linear interpolation between vertices.

1.1 Approach

Simple geospatial feature collections will conceptually be stored as tables with geometry valued columns in a Relational DBMS (RDBMS), each feature will be stored as a row in a table. The non-spatial attributes of features will be mapped onto columns whose types are drawn from the set of standard ODBC/SQL92 data types. The spatial attributes of features will be mapped onto columns whose SQL data types are based on the underlying concept of additional geometric data types for SQL. A table whose rows represent OpenGIS features shall be referred to as a feature table. Such a table shall contain one or more geometry valued columns. Feature table implementations are described for two target SQL environments: SQL92 and SQL92 with Geometry Types.

In the SQL92 environment, a geometry-valued column is implemented as a Foreign Key reference into a geometry table. A geometry value is stored using one or more rows in the geometry table. The geometry table may be implemented using either standard SQL numeric types or SQL binary types, schemas for both alternatives are described.

The term SQL92 with Geometry Types is used to refer to a SQL92 environment that has been extended with a set of Geometry Types. In this environment a geometry-valued column is implemented as a column whose SQL type is drawn from the set of Geometry Types. This specification describes a standard set of SQL Geometry Types based on the OpenGIS Geometry Model, together with the SQL functions on those types. This specification does not attempt to standardize any part of the mechanism by which the Geometry Types are added to and maintained in the SQL environment: The standard SQL3 mechanism for extending the type system of a SQL database is through the definition of user defined Abstract Data Types. Commercial implementations of SQL92 environments with user defined type support are available as of mid 1997. The SQL3 standard should be ratified in 1998.

Both the SQL92 and the SQL92 with Geometry Types implementations extend the SQL92 Information Schema in a uniform manner so as to support standard Metadata Queries that return:

1. The list of feature tables in a database.
2. The list of geometry columns for any feature table in the database.
3. The Spatial Reference System for any geometry column in the database.

Both the SQL92 and the SQL92 with Geometry Types implementations are accessed from ODBC using the support already built into ODBC for fetching and storing standard integer, character and binary ODBC SQL types.

In order to be compliant with this OpenGIS ODBC/SQL specification for geospatial feature collections, implementers shall choose to implement any one of three alternatives (1a, 1b or 2) described in this specification:

1. SQL92 implementation of feature tables
   a) using numeric SQL types for geometry storage and ODBC access.
   b) using binary SQL types for geometry storage and ODBC access.

2. SQL92 with Geometry Types implementation of feature tables supporting both textual and binary ODBC access to geometry.

The remainder of this specification is structured as follows:

- Chapter 2 describes the architecture of the system for both the SQL92 environment and for the SQL92 with Geometry Types environment. It begins with a Distributed Computing Platform neutral conceptual object model for Geometry. Upon this object model, the detailed specification for geometry values, geometry types and the SQL functions that operate upon geometry types is based.

- Chapter 3 specifies the architectural components of the system for the SQL92 environment and for the SQL92 with Geometry Types environment.

- Chapter 4 details supported spatial reference system data for use with this specification.

- Chapter 5 contains the references utilized by the specification.
2 Architecture

2.1 Geometry Object Model

This section describes the object model for geometry. It is Distributed Computing Platform neutral and uses OMT notation. The object model for geometry is shown in Figure 2.1. The base Geometry class has subclasses for Point, Curve, Surface and Geometry Collection. Each geometric object is associated with a Spatial Reference System, which describes the coordinate space in which the geometric object is defined.

Figure 2.1 is based on extending the Geometry Model specified in the OpenGIS Abstract Specification with specialized 0, 1 and two-dimensional collection classes named MultiPoint, MultiLineString and MultiPolygon for modelling geometries corresponding to collections of Points, LineStrings and Polygons respectively. MultiCurve and MultiSurface are introduced as abstract superclasses at this RFP that generalize the collection interfaces to handle Curves and Surfaces. The figure shows aggregation lines between the leaf collection classes and their element classes, the aggregation lines for non-leaf collection classes are described in the text.

The attributes, methods and assertions for each geometry class are described below. In describing methods, this is used to refer to the receiver of the method (the object being messaged). The scope of the methods and attributes is based on the scope of RFP1 (SimpleFeatures).
2.1.1 Geometry

Geometry is the root class of the hierarchy. Geometry is an abstract (non-instantiable) class.

The instantiable subclasses of Geometry defined in this specification are restricted to 0, 1 and two-dimensional geometric objects that exist in two-dimensional coordinate space ($\mathbb{R}^2$).

All instantiable geometry classes described in this specification are defined so that valid instances of a geometry class are topologically closed (i.e. all defined geometries include their boundary).

2.1.1.1 Basic Methods on Geometry

**Dimension**: Integer—The inherent dimension of this Geometry object, which must be less than or equal to the coordinate dimension. This specification is restricted to geometries in two-dimensional coordinate space.

**GeometryType**: String—Returns the name of the instantiable subtype of Geometry of which this Geometry instance is a member. The name of the instantiable subtype of Geometry is returned as a string.

**SRID**: Integer—Returns the Spatial Reference System ID for this Geometry.

**Envelope**: Geometry—The minimum bounding box for this Geometry, returned as a Geometry. The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MAXX, MINY), (MAXX, MAXY), (MINX, MAXY), (MINX, MINY)).

**AsText**: String—Exports this Geometry to a specific well-known text representation of Geometry.
AsBinary( ):Binary—Exports this Geometry to a specific well-known binary representation of Geometry.

IsEmpty( ):Integer —Returns 1 (TRUE) if this Geometry is the empty geometry. If true, then this Geometry represents the empty point set, \( \emptyset \), for the coordinate space.

IsSimple( ):Integer —Returns 1 (TRUE) if this Geometry has no anomalous geometric points, such as self intersection or self tangency. The description of each instantiable geometric class will include the specific conditions that cause an instance of that class to be classified as not simple.

Boundary( ):Geometry —Returns the closure of the combinatorial boundary of this Geometry. The combinatorial boundary is defined as described in section 3.12.3.2 of [1]. Because the result of this function is a closure, and hence topologically closed, the resulting boundary can be represented using representational geometry primitives as discussed in [1], section 3.12.2.

2.1.1.2 Methods for testing Spatial Relations between geometric objects:

The methods in this section are defined and described in more detail following the description of the sub types of Geometry.

Equals(anotherGeometry:Geometry):Integer — Returns 1 (TRUE) if this Geometry is ‘spatially equal’ to anotherGeometry.

Disjoint(anotherGeometry:Geometry):Integer— Returns 1 (TRUE) if this Geometry is ‘spatially disjoint’ from anotherGeometry.

Intersects(anotherGeometry:Geometry):Integer— Returns 1 (TRUE) if this Geometry ‘spatially intersects’ anotherGeometry.

Touches(anotherGeometry:Geometry):Integer— Returns 1 (TRUE) if this Geometry ‘spatially touches’ anotherGeometry.

Crosses(anotherGeometry:Geometry):Integer— Returns 1 (TRUE) if this Geometry ‘spatially crosses’ anotherGeometry.

Within(anotherGeometry:Geometry):Integer — Returns 1 (TRUE) if this Geometry is ‘spatially within’ anotherGeometry.

Contains(anotherGeometry:Geometry):Integer — Returns 1 (TRUE) if this Geometry ‘spatially contains’ anotherGeometry.

Overlaps(anotherGeometry:Geometry):Integer — Returns 1 (TRUE) if this Geometry ‘spatially overlaps’ anotherGeometry.

Relate(anotherGeometry:Geometry, intersectionPatternMatrix:String):Integer— Returns 1 (TRUE) if this Geometry is spatially related to anotherGeometry, by testing for intersections between the Interior, Boundary and Exterior of the two geometries as specified by the values in the intersectionPatternMatrix.

2.1.1.3 Methods that support Spatial Analysis

Distance(anotherGeometry:Geometry):Double—Returns the shortest distance between any two points in the two geometries as calculated in the spatial reference system of this Geometry.
Buffer(distance:Double):Geometry—Returns a geometry that represents all points whose distance from this Geometry is less than or equal to distance. Calculations are in the Spatial Reference System of this Geometry.

ConvexHull():Geometry—Returns a geometry that represents the convex hull of this Geometry.

Intersection(anotherGeometry:Geometry):Geometry—Returns a geometry that represents the point set intersection of this Geometry with anotherGeometry.

Union(anotherGeometry:Geometry):Geometry—Returns a geometry that represents the point set union of this Geometry with anotherGeometry.

Difference(anotherGeometry:Geometry):Geometry—Returns a geometry that represents the point set difference of this Geometry with anotherGeometry.

SymDifference(anotherGeometry:Geometry):Geometry—Returns a geometry that represents the point set symmetric difference of this Geometry with anotherGeometry.

2.1.2 Geometry Collection

A GeometryCollection is a geometry that is a collection of 1 or more geometries.

All the elements in a GeometryCollection must be in the same Spatial Reference. This is also the Spatial Reference for the GeometryCollection.

GeometryCollection places no other constraints on its elements. Subclasses of GeometryCollection may restrict membership based on dimension and may also place other constraints on the degree of spatial overlap between elements.

2.1.2.1 Methods

NumGeometries():Integer—Returns the number of geometries in this GeometryCollection.


2.1.3 Point

A Point is a 0-dimensional geometry and represents a single location in coordinate space. A Point has a x-coordinate value and a y-coordinate value.

The boundary of a Point is the empty set.

2.1.3.1 Methods

X():Double—The x-coordinate value for this Point.

Y():Double—The y-coordinate value for this Point.

2.1.4 MultiPoint

A MultiPoint is a 0 dimensional geometric collection. The elements of a MultiPoint are restricted to Points. The points are not connected or ordered.
A MultiPoint is simple if no two Points in the MultiPoint are equal (have identical coordinate values).

The boundary of a MultiPoint is the empty set.

2.1.5 Curve

A Curve is a one-dimensional geometric object usually stored as a sequence of points, with the subtype of Curve specifying the form of the interpolation between points. This specification defines only one subclass of Curve, LineString, which uses linear interpolation between points.

Topologically a Curve is a one-dimensional geometric object that is the homeomorphic image of a real, closed, interval $D = \{ x \in \mathbb{R} \mid a \leq x \leq b \}$ under a mapping $f : [a, b] \rightarrow \mathbb{R}^2$ as defined in [1], section 3.12.7.2.

A Curve is simple if it does not pass through the same point twice ([1], section 3.12.7.3)

$$\forall c \in Curve, [a, b] = c.\text{Domain},$$

$$c.\text{IsSimple} \iff (\forall x_1, x_2 \in (a, b) x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)) \wedge (\forall x_1, x_2 \in [a, b) x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2))$$

A Curve is closed if its start point is equal to its end point. ([1], section 3.12.7.3)

The boundary of a closed Curve is empty.

A Curve that is simple and closed is a Ring.

The boundary of a non-closed Curve consists of its two end points. ([1], section 3.12.3.2).

A Curve is defined as topologically closed.

2.1.5.1 Methods

\textbf{Length( ):Double}—The length of \textit{this} Curve in its associated spatial reference.

\textbf{StartPoint( ):Point}—The start point of \textit{this} Curve.

\textbf{EndPoint( ):Point}—The end point of \textit{this} Curve.

\textbf{IsClosed( ):Integer}—Returns 1 (TRUE) if \textit{this} Curve is closed (StartPoint( ) = EndPoint( )).

\textbf{IsRing( ):Integer}—Returns 1 (TRUE) if \textit{this} Curve is closed (StartPoint( ) = EndPoint( )) and \textit{this} Curve is simple (does not pass through the same point more than once).

2.1.6 LineString, Line, LinearRing

A LineString is a Curve with linear interpolation between points. Each consecutive pair of points defines a line segment.

A Line is a LineString with exactly 2 points.

A LinearRing is a LineString that is both closed and simple. The curve in Figure 2.2—(3) is a closed LineString that is a LinearRing. The curve in Figure 2.2—(4) is a closed LineString that is not a LinearRing.
2.1.6.1 Methods

**NumPoints( )**:Integer—The number of points in this LineString.

**PointN(N:Integer)**:Point—Returns the specified point N in this LineString.

2.1.7 MultiCurve

A MultiCurve is a one-dimensional GeometryCollection whose elements are Curves (Figure 2.3).

MultiCurve is a non-instantiable class in this specification, it defines a set of methods for its subclasses and is included for reasons of extensibility.

A MultiCurve is simple if and only if all of its elements are simple, the only intersections between any two elements occur at points that are on the boundaries of both elements.

The boundary of a MultiCurve is obtained by applying the ‘mod 2’ union rule: A point is in the boundary of a MultiCurve if it is in the boundaries of an odd number of elements of the MultiCurve. ([1], section 3.12.3.2).

A MultiCurve is closed if all of its elements are closed. The boundary of a closed MultiCurve is always empty.

A MultiCurve is defined as topologically closed.
2.1.7.1 Methods

IsClosed( ): Integer—Returns 1 (TRUE) if this MultiCurve is closed (StartPoint ( ) = EndPoint ( ) for each curve in this MultiCurve)

Length( ): Double—The Length of this MultiCurve which is equal to the sum of the lengths of the element Curves.

2.1.8 MultiLineString

A MultiLineString is a MultiCurve whose elements are LineStrings.

![Diagram of MultiLineStrings]

Figure 2.3—(1) a simple MultiLineString, (2) a non-simple MultiLineString with 2 elements, (3) a non-simple, closed MultiLineString with 2 elements

The boundaries for the MultiLineStrings in Figure 2.3 are (1)—{s1, e2}, (2)—{s1, e1}, (3)—∅

2.1.9 Surface

A Surface is a two-dimensional geometric object.

The OpenGIS Abstract Specification defines a simple Surface as consisting of a single ‘patch’ that is associated with one ‘exterior boundary’ and 0 or more ‘interior’ boundaries. Simple surfaces in three-dimensional space are isomorphic to planar surfaces. Polyhedral surfaces are formed by ‘stitching’ together simple surfaces along their boundaries, polyhedral surfaces in three-dimensional space may not be planar as a whole ([1], sections 3.12.9.1, 3.12.9.3).

The boundary of a simple Surface is the set of closed curves corresponding to its ‘exterior’ and ‘interior’ boundaries. ([1], section 3.12.9.4).
The only instantiable subclass of Surface defined in this specification, Polygon, is a simple Surface that is planar.

2.1.9.1 Methods

Area():Double—The area of this Surface, as measured in the spatial reference system of this Surface.

Centroid():Point—The mathematical centroid for this Surface as a Point. The result is not guaranteed to be on this Surface.

PointOnSurface():Point—A point guaranteed to be on this Surface.

2.1.10 Polygon

A Polygon is a planar Surface, defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon.

The assertions for polygons (the rules that define valid polygons) are:

1. Polygons are topologically closed.
2. The boundary of a Polygon consists of a set of LinearRings that make up its exterior and interior boundaries.
3. No two rings in the boundary cross, the rings in the boundary of a Polygon may intersect at a Point but only as a tangent:

   \[ \forall P \in \text{Polygon}, \forall c1, c2 \in P.\text{Boundary()}, c1 \neq c2, \forall p, q \in \text{Point}, p, q \in c1, p \neq q, \{ p \in c2 \Rightarrow q \notin c2 \} \]
4. A Polygon may not have cut lines, spikes or punctures:

   \[ \forall P \in \text{Polygon}, P = \text{Closure(Interior}(P)) \]
5. The Interior of every Polygon is a connected point set.
6. The Exterior of a Polygon with 1 or more holes is not connected. Each hole defines a connected component of the Exterior.

In the above assertions, Interior, Closure and Exterior have the standard topological definitions. The combination of 1 and 3 make a Polygon a Regular Closed point set.

Polygons are simple geometries.

Figure 2.4 shows some examples of Polygons. Figure 2.5 shows some examples of geometric objects that violate the above assertions and are not representable as single instances of Polygon. The objects shown in Figure 2.5—(1) and 2.5—(4) can be represented as 2 separate Polygons.
Figure 2.4—Examples of Polygons with 1, 2 and 3 rings respectively.

Figure 2.5—Examples of objects not representable as a single instance of Polygon. (1) and (4) can be represented as 2 separate Polygons.
2.1.10.1 Methods

ExteriorRing():LineString— Returns the exterior ring of this Polygon.

NumInteriorRing():Integer— Returns the number of interior rings in this Polygon.

InteriorRingN(N:Integer):LineString— Returns the Nth interior ring for this Polygon as a LineString.

2.1.11 MultiSurface

A MultiSurface is a two-dimensional geometric collection whose elements are surfaces. The interiors of any two surfaces in a MultiSurface may not intersect. The boundaries of any two elements in a MultiSurface may intersect at most at a finite number of points.

MultiSurface is a non-instantiable class in this specification, it defines a set of methods for its subclasses and is included for reasons of extensibility. The instantiable subclass of MultiSurface is MultiPolygon, corresponding to a collection of Polygons.

2.1.11.1 Methods

Area():Double— The area of this MultiSurface, as measured in the spatial reference system of this MultiSurface.

Centroid():Point— The mathematical centroid for this MultiSurface. The result is not guaranteed to be on this MultiSurface.

PointOnSurface():Point— A Point guaranteed to be on this MultiSurface.

2.1.12 MultiPolygon

A MultiPolygon is a MultiSurface whose elements are Polygons.

The assertions for MultiPolygons are:

1. The interiors of 2 Polygons that are elements of a MultiPolygon may not intersect.
   \[ \forall M \in MultiPolygon, \forall Pi, Pj \in M.Geometries(), \forall i, j, Interior(Pi) \cap Interior(Pj) = \emptyset \]

2. The Boundaries of any 2 Polygons that are elements of a MultiPolygon may not ‘cross’ and may touch at only a finite number of points. (Note that crossing is prevented by assertion 1 above).
   \[ \forall M \in MultiPolygon, \forall Pi, Pj \in M.Geometries(), \forall ci \in Pi.Boundaries(), cj \in Pj.Boundaries() \]
   \[ ci \cap cj = \{pi, ..., pk | pi \in Point, 1 <= i <= k\} \]

3. A MultiPolygon is defined as topologically closed.

4. A MultiPolygon may not have cut lines, spikes or punctures, a MultiPolygon is a Regular, Closed point set:
   \[ \forall M \in MultiPolygon, M = Closure(Interior(M)) \]

5. The interior of a MultiPolygon with more than 1 Polygon is not connected, the number of connected components of the interior of a MultiPolygon is equal to the number of Polygons in the MultiPolygon.
The boundary of a MultiPolygon is a set of closed curves (LineStrings) corresponding to the boundaries of its element Polygons. Each Curve in the boundary of the MultiPolygon is in the boundary of exactly 1 element Polygon, and every Curve in the boundary of an element Polygon is in the boundary of the MultiPolygon.

The reader is referred to work by Worboys, et. al (7, 8) and Clementini, et. al (5, 6) for work on the definition and specification of MultiPolygons.

Figure 2.6 shows 4 examples of valid MultiPolygons with 1, 3, 2 and 2 polygon elements respectively.

![Figure 2.6—Examples of MultiPolygons](image)

Figure 2.7 shows examples of geometric objects not representable as single instances of MultiPolygons.

Note that the subclass of Surface named Polyhedral Surface described in the [1], is a faceted surface whose facets are Polygons. A Polyhedral Surface is not a MultiPolygon because it violates the rule for MultiPolygons that the boundaries of the element Polygons intersect only at a finite number of points.
2.1.13 Relational Operators

This section provides a more detailed specification of the relational operators on geometries.

2.1.13.1 Background

The Relational Operators are Boolean methods that are used to test for the existence of a specified topological spatial relationship between two geometries. Topological spatial relationships between two geometric objects have been a topic of extensive study in the literature [4,5,6,7,8,9,10]. The basic approach to comparing two geometries is to make pair-wise tests of the intersections between the Interiors, Boundaries and Exteriors of the two geometries and to classify the relationship between the two geometries based on the entries in the resulting ‘intersection’ matrix.

The concepts of Interior, Boundary and Exterior are well defined in general topology. For a review of these concepts the user is referred to Egenhofer, et al [4]. These concepts can be applied in defining spatial relationships between two-dimensional objects in two-dimensional space ($\mathbb{R}^2$). In order to apply the concepts of Interior, Boundary and Exterior to 1 and 0 dimensional objects in $\mathbb{R}^2$, a combinatorial topology approach must be applied. ([1], section 3.12.3.2). This approach is based on the accepted definitions of the boundaries, interiors and exteriors for simplicial complexes [12] and yields the following results:

The boundary of a geometry is a set of geometries of the next lower dimension. The boundary of a Point or a MultiPoint is the empty set. The boundary of a non-closed Curve consists of its two end Points, the boundary of a closed Curve is empty. The boundary of a MultiCurve consists of those Points that are in the boundaries of an odd number of its element Curves. The boundary of a Polygon consists of its set of Rings. The boundary of a MultiPolygon consists of the set of Rings of its Polygons. The boundary of an arbitrary collection of geometries whose interiors are disjoint consists of geometries drawn from the boundaries of the element geometries by application of the ‘mod 2’ union rule ([1], section 3.12.3.2).
The domain of geometric objects considered is those that are topologically closed. The interior of a geometry consists of those points that are left when the boundary points are removed. The exterior of a geometry consists of points not in the interior or boundary.

Studies on the relationships between two geometries both of maximal dimension in \( \mathbb{R}^1 \) and \( \mathbb{R}^2 \) considered pair-wise intersections between the Interior and Boundary sets and led to the definition of a 4 Intersection Model [8]. The model was extended to consider the exterior of the input geometries, resulting in a nine intersection model [11] and further extended to include information on the dimension of the results of the pair-wise intersections resulting in a dimensionally extended nine intersection model [5]. These extensions allow the model to express spatial relationships between points, lines and areas, including areas with holes and multi component lines and areas [6].

### 2.1.13.2 The Dimensionally Extended Nine-Intersection Model

Given a geometry a, let \( I(a) \), \( B(a) \) and \( E(a) \) represent the Interior, Boundary and Exterior of a respectively. The intersection of any two of \( I(a) \), \( B(a) \) and \( E(a) \) can result in a set of geometries, \( x \), of mixed dimension. For example, the intersection of the boundaries of two polygons may consist of a point and a line. Let \( \text{dim}(x) \) return the maximum dimension (-1, 0, 1, or 2) of the geometries in \( x \), with a numeric value of -1 corresponding to \( \text{dim}(\emptyset) \). A dimensionally extended nine-intersection matrix (DE-9IM) then has the form:

<table>
<thead>
<tr>
<th>Interior</th>
<th>Boundary</th>
<th>Exterior</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{dim}(I(a) \cap I(b)) )</td>
<td>( \text{dim}(I(a) \cap B(b)) )</td>
<td>( \text{dim}(I(a) \cap E(b)) )</td>
</tr>
<tr>
<td>( \text{dim}(B(a) \cap I(b)) )</td>
<td>( \text{dim}(B(a) \cap B(b)) )</td>
<td>( \text{dim}(B(a) \cap E(b)) )</td>
</tr>
<tr>
<td>( \text{dim}(E(a) \cap I(b)) )</td>
<td>( \text{dim}(E(a) \cap B(b)) )</td>
<td>( \text{dim}(E(a) \cap E(b)) )</td>
</tr>
</tbody>
</table>

**Table 2.1—The DE-9IM**

For regular, topologically closed input geometries, computing the dimension of the intersection of the Interior, Boundary and Exterior sets does not have as a prerequisite the explicit computation and representation of these sets. For example to compute if the interiors of two regular closed polygons intersect, and to ascertain the dimension of this intersection, it is not necessary to explicitly represent the interior of the two polygons (which are topologically open sets) as separate geometries. In most cases the dimension of the intersection value at a cell is highly constrained given the type of the two geometries. For example, in the Line-Area case the only possible values for the Interior-Interior cell are drawn from \{-1, 1\} and in the Area-Area case the only possible values for the Interior-Interior cell are drawn from \{-1, 2\}. In such cases no work beyond detecting the intersection is required.

Figure 2.8 shows an example DE-9IM for the case where \( a \) and \( b \) are two polygons that overlap.
A spatial relationship predicate can be formulated on two geometries that takes as input a pattern matrix representing the set of acceptable values for the DE-9IM for the two geometries. If the spatial relationship between the two geometries corresponds to one of the acceptable values as represented by the pattern matrix, then the predicate returns TRUE.

The pattern matrix consists of a set of 9 pattern-values, one for each cell in the matrix. The possible pattern-values $p$ are \{T, F, *, 0, 1, 2\} and their meanings for any cell where $x$ is the intersection set for the cell are as follows:

\[
\begin{align*}
    p = T & \Rightarrow \dim(x) \in \{0, 1, 2\}, \text{ i.e. } x \neq \emptyset \\
    p = F & \Rightarrow \dim(x) = -1, \text{ i.e. } x = \emptyset \\
    p = * & \Rightarrow \dim(x) \in \{-1, 0, 1, 2\}, \text{ i.e. Don’t Care} \\
    p = 0 & \Rightarrow \dim(x) = 0 \\
    p = 1 & \Rightarrow \dim(x) = 1 \\
    p = 2 & \Rightarrow \dim(x) = 2
\end{align*}
\]

The pattern matrix can be represented as an array or list of nine characters in row major order. As an example the following code fragment could be used to test for ‘Overlap’ between two areas:

```c
char * overlapMatrix = 'T*T***T**';
Geometry* a, b;
Boolean b = a->Relate(b, overlapMatrix);
```
2.1.13.3 Named Spatial Relationship predicates based on the DE-9IM

The Relate predicate based on the pattern matrix has the advantage that clients can test for a large number of spatial relationships and fine tune the particular relationship being tested. It has the disadvantage that it is a lower level building block and does not have a corresponding natural language equivalent. Users of the proposed system include IT developers using the COM API from a language such as Visual Basic, and interactive SQL users who may wish, for example, to select all features ‘spatially within’ a query polygon, in addition to more spatially ‘sophisticated’ GIS developers.

To address the needs of such users a set of named spatial relationship predicates have been defined in [5,6] for the DE-9IM. The five predicates are named Disjoint, Touches, Crosses, Within and Overlaps. The definition of these predicates [5,6] is given below. In these definitions the term P is used to refer to 0 dimensional geometries (Points and MultiPoints), L is used to refer to one-dimensional geometries (LineStrings and MultiLineStrings) and A is used to refer to two-dimensional geometries (Polygons and MultiPolygons).

**Disjoint**

Given two (topologically closed) geometries \(a\) and \(b\),

\[
a.\text{Disjoint}(b) \iff a \cap b = \emptyset
\]

Expressed in terms of the DE-9IM:

\[
a.\text{Disjoint}(b) \iff (I(a) \cap I(b) = \emptyset) \land (I(a) \cap B(b) = \emptyset) \land (B(a) \cap I(b) = \emptyset) \land (B(a) \cap B(b) = \emptyset) \\
\iff a.\text{Relate}(b, "FF*FF****")
\]

**Touches**

The Touches relation between two geometries \(a\) and \(b\) applies to the A/A, L/L, L/A, P/A and P/L groups of relationships but not to the P/P group. It is defined as:

\[
a.\text{Touches}(b) \iff (I(a) \cap I(b) = \emptyset) \land (a \cap b) \neq \emptyset
\]

Expressed in terms of the DE-9IM:

\[
a.\text{Touches}(b) \iff (I(a) \cap I(b) = \emptyset) \land (B(a) \cap B(b) \neq \emptyset) \lor (I(a) \cap B(b) \neq \emptyset) \lor (B(a) \cap B(b) \neq \emptyset) \\
\iff a.\text{Relate}(b, "FT*******") \lor a.\text{Relate}(b, "F**T*****") \lor a.\text{Relate}(b, "F***T****")
\]

Figure 2.9 shows some examples of the Touches relation.
Crosses

The Crosses relation applies to P/L, P/A, L/L and L/A situations. It is defined as:

\[
a.\text{Crosses}(b) \Leftrightarrow (\text{dim}(I(a) \cap I(b)) < \max(\text{dim}(I(a)), \text{dim}(I(b)))) \land (a \cap b \neq a) \land (a \cap b \neq b)
\]

Expressed in terms of the DE-9IM:

Case \( a \in P, b \in L \) or Case \( a \in P, b \in A \) or Case \( a \in L, b \in A \):

\[
a.\text{Crosses}(b) \Leftrightarrow (I(a) \cap I(b) \neq \emptyset) \land (I(a) \cap E(b) \neq \emptyset) \Leftrightarrow a.\text{Relate}(b, \text{`T*T******'})
\]

Case \( a \in L, b \in L \):

\[
a.\text{Crosses}(b) \Leftrightarrow \text{dim}(I(a) \cap I(b)) = 0 \Leftrightarrow a.\text{Relate}(b, \text{`0********'})
\]

Figure 2.10 shows some examples of the Crosses relation.
Within

The Within relation is defined as:

\[ a.Within(b) \iff (a \cap b = a) \land (I(a) \cap I(b) \neq \emptyset) \]

Expressed in terms of the DE-9IM:

\[ a.Within(b) \iff (I(a) \cap I(b) \neq \emptyset) \land (I(a) \cap E(b) = \emptyset) \land (B(a) \cap E(b) = \emptyset) \iff a.Relate(b, 'T**F***') \]

Figure 2.11 shows some examples of the Within relation.
Overlaps

The Overlaps relation is defined for A/A, L/L and P/P situations.

It is defined as:

\[ a.\text{Overlaps}(b) \Leftrightarrow (\dim(I(a)) = \dim(I(b)) = \dim(I(a) \cap I(b))) \land (a \cap b \neq a) \land (a \cap b \neq b) \]

Expressed in terms of the DE-9IM:

Case \( a \in P, b \in P \) or Case \( a \in A, b \in A \):

\[ a.\text{Overlaps}(b) \Leftrightarrow (\dim(I(a) \cap I(b) \neq \emptyset)) \land (\dim(I(a) \cap E(b) \neq \emptyset)) \land (\dim(E(a) \cap I(b) \neq \emptyset)) \Leftrightarrow a.\text{Relate}(b, 'T**T*1**T**') \]

Case \( a \in L, b \in L \):

\[ a.\text{Overlaps}(b) \Leftrightarrow (\dim(I(a) \cap I(b) = 1)) \land (\dim(I(a) \cap E(b) \neq \emptyset)) \land (\dim(E(a) \cap I(b) \neq \emptyset)) \Leftrightarrow a.\text{Relate}(b, '1**T*1**T**') \]

Figure 2.12 shows some examples of the Overlaps relation.
The following additional named predicates are also defined for user convenience:

**Contains**

\[ a.\text{Contains}(b) \Leftrightarrow b.\text{Within}(a) \]

**Intersects**

\[ a.\text{Intersects}(b) \Leftrightarrow \neg a.\text{Disjoint}(b) \]

Based on the above operators the following methods are defined on Geometry:

**Equals**\(\text{anotherGeometry:Geometry}:\text{Integer}\)—Returns 1 (TRUE) if *this* Geometry is ‘spatially equal’ to anotherGeometry.

**Disjoint**\(\text{anotherGeometry:Geometry}:\text{Integer}\)—Returns 1 (TRUE) if *this* Geometry is ‘spatially disjoint’ from anotherGeometry.

**Intersects**\(\text{anotherGeometry:Geometry}:\text{Integer}\)—Returns 1 (TRUE) if *this* Geometry ‘spatially intersects’ anotherGeometry.

**Touches**\(\text{anotherGeometry:Geometry}:\text{Integer}\)—Returns 1 (TRUE) if *this* Geometry ‘spatially touches’ anotherGeometry.

**Crosses**\(\text{anotherGeometry:Geometry}:\text{Integer}\)—Returns 1 (TRUE) if *this* Geometry ‘spatially crosses’ anotherGeometry.
Within(anotherGeometry:Geometry):Integer—Returns 1 (TRUE) if this Geometry is ‘spatially within’ anotherGeometry.

Contains(anotherGeometry:Geometry):Integer—Returns 1 (TRUE) if this Geometry ‘spatially contains’ anotherGeometry.

Overlaps(anotherGeometry:Geometry):Integer—Returns 1 (TRUE) if this Geometry ‘spatially overlaps’ anotherGeometry.

Relate(anotherGeometry:Geometry, intersectionPatternMatrix:String):Integer—Returns 1 (TRUE) if this Geometry is spatially related to anotherGeometry, by testing for intersections between the Interior, Boundary and Exterior of the two geometries.

2.2 Architecture—SQL92 Implementation of Feature Tables

A SQL92 implementation of OpenGIS simple geospatial feature collections defines a schema for storage of feature table, geometry and spatial reference system information. The SQL92 implementation does not define SQL functions for access, maintenance, or indexing of geometry, as these functions cannot be uniformly implemented across database systems using the SQL92 standard.

The figure below describes the database schema necessary to support the OpenGIS simple feature data model. A feature table or view corresponds to an OpenGIS feature class. Each feature view contains some number of features represented as rows in the view. Each feature contains some number of geometric attribute values represented as columns in the feature view. Each geometric column in a feature view is associated with a particular geometric view or table that contains geometry instances in a single spatial reference system. The correspondence between the feature instances and the geometry instances shall be accomplished through a foreign key that is stored in the geometry column of the feature table. This foreign key references the GID primary key of the geometry table.

![Figure 2.13—Schema for feature tables under SQL92](image-url)
Depending upon the type of storage specified by the geometry metadata, Geometry instances shall be stored as either arrays of coordinate values or as binary values using an OpenGIS defined Well-Known Binary Representation for Geometry. In the former case, SQL numeric types are used for the coordinates and client side functions are needed to build OpenGIS geometry objects from the retrieved coordinate numeric values. In the latter case clients may feed the retrieved well-known binary representation directly into the Geometry factory of the client side computing environment (e.g., an OLE/COM or CORBA geometry factory) or choose to access the individual coordinate values by unpacking the well-known representation.

2.2.1 Feature Table Metadata Views

A feature table is any table having 1 or more foreign key reference to a geometry table or view. The set of feature tables in a database can be determined using the above rule from the TABLES, REFERENTIAL_CONSTRAINTS and COLUMNS metadata views in the SQL92 INFORMATION_SCHEMA. The set of feature tables can also be determined by issuing a query over the GEOMETRY_COLUMNS metadata view described below.

2.2.2 Geometry Columns Metadata Views

Each geometry column will be represented as a row in the standard COLUMNS metadata view in the SQL92 INFORMATION_SCHEMA. Spatial Reference System Identity is however not a standard part of the SQL92 INFORMATION_SCHEMA. To represent this information we introduce an additional metadata view named GEOMETRY_COLUMNS.

The GEOMETRY_COLUMNS table or view consists of a row for each geometry column in the database. The data stored for each geometry column includes:

- the identity of the feature table of which it is a member,
- the spatial reference system ID,
- the type of geometry for the column,
- the coordinate dimension for the column,
- the identity of the geometry table that stores its instances, and
- the information necessary to navigate the geometry tables in the case of normalized geometry storage.

2.2.3 Spatial Reference System Information Views

Every geometry column is associated with a Spatial Reference System. The Spatial Reference System identifies the coordinate system for all geometries stored in the column, and gives meaning to the numeric coordinate values for any geometry instance stored in the column. Examples of commonly used Spatial Reference Systems include ‘Latitude Longitude’, and ‘UTM Zone 10’.

The SPATIAL_REFERENCE_SYSTEMS table stores information on each Spatial Reference System in the database. The columns of this table are the Spatial Reference System Identifier (SRID), the Spatial Reference System Authority Name (AUTH_NAME), the Authority Specific Spatial Reference System Identifier (AUTH_SRID) and the Well-known Text description of the Spatial Reference System (SRTEXT). The Spatial Reference System Identifier (SRID) constitutes a unique integer key for a Spatial Reference System within a database.
Interoperability between clients is achieved via the SRTEXT column which stores the Well-known Text representation for a Spatial Reference System as described in Section 3.4.

2.2.4 Feature Tables and Views

A Feature is an object with geometric attributes [1]. Features are stored as rows in tables, each geometric attribute is a foreign key reference to a geometry table or view. Relationships between Features are defined as FOREIGN KEY references between feature tables.

2.2.5 Geometry and Geometric Element Views

There are two implementations for storing geometries in SQL92: using a normalized geometry SQL92 schema, and using a binary geometry SQL92 schema. The binary geometry schema uses the Well-known Binary Representation for Geometry (WKBGeometry) described in section 3.3. The normalized geometry implementation defines fixed width SQL92 tables such as the example in Figure 2.14. Each primitive element in the geometry is distributed over some number of adjacent rows in the table ordered by a sequence number (SEQ), and identified by a primitive type (ETYPE). Each geometry identified by a key (GID), consists of a collection of elements numbered by an element sequence (ESEQ).

The rules for geometric entity representation in the normalized SQL92 schema are defined as follows:

- ETYPE designates the geometry type.
- Geometries may have multiple elements. The ESEQ value identifies the individual elements.
- An element may be built up from multiple parts (rows). The rows and their proper sequence are identified by the SEQ value.
- Polygons may contain holes, as described in the geometry object model.
- Polygon rings must close when assembled from an ordered list of parts. The SEQ value designates the part order.
- Coordinate pairs that are not used must be set to Nil in complete sets (both X and Y). This is the only way to identify the end of list of coordinates.
- For geometries that continue onto an additional row (as defined by an constant element sequence number or ESEQ) the last point of one row is equal to the first point of the next.
- There is no limit on the number of elements in the geometry, or the number of rows in a element.
The binary geometry implementation is illustrated in Table 2.2, and uses the same GID as a key, but stores the geometry using the Well-known Binary Representation for Geometry (WKBGeometry) described in section 3.3. The geometry table includes the minimum bounding rectangle for the geometry as well as the WKBGeometry for the geometry. This permits construction of spatial indexes without accessing the actual geometry structure, if desired.

<table>
<thead>
<tr>
<th>GID</th>
<th>XMIN</th>
<th>YMIN</th>
<th>XMAX</th>
<th>YMAX</th>
<th>GEOMETRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>30</td>
<td>&lt; WKBGeometry &gt;</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>0</td>
<td>60</td>
<td>30</td>
<td>&lt; WKBGeometry &gt;</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>30</td>
<td>30</td>
<td>60</td>
<td>&lt; WKBGeometry &gt;</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>60</td>
<td>60</td>
<td>&lt; WKBGeometry &gt;</td>
</tr>
</tbody>
</table>

Table 2.2—Example of geometry table for above Polygon Geometry using the Well-known Binary Representation for Geometry.

2.2.6 Notes on SQL92 data types

There are various ways to store the same values in a relational database. For example, there are usually several ways to store numbers. In this specification, the use of a storage alternative is not meant to be binding. Since the storage type of any column is available in the data dictionary, and such casting operators between similar types are available, any particular implementation may use alternative storage formats as long as casting operations would not lead to difficulties.
2.2.7 Notes on ODBC Access to Geometry Values stored in Binary form.

ODBC provides standard mechanisms to bind character, numeric and binary data values.

This section describes the process of retrieving geometry values for the case where the binary storage alternative is chosen.

The WKB_GEOMETRY column in the geometry table for a geometry column surfaces in ODBC as one of the ODBC binary SQL data types (SQL_BINARY, SQL_VARBINARY, or SQL_LONGVARBINARY). An application binds to this column using the ODBC 2.0 C datatype SQL_C_BINARY.

For example, the application would use the SQL_C_BINARY value for the fCType parameter of SQLBindCol (or SQLGetData) in order to describe the application data buffer that will receive the fetched Geometry data value. Similarly, a dynamic parameter whose value is a Geometry would be described using the SQL_C_BINARY value for the fCType parameter of SQLBindParameter.

This allows binary values to be both retrieved from and inserted into the geometry tables.

2.3 Architecture—SQL92 with Geometry Types Implementation of Feature Tables

2.3.1 Feature Table Metadata Views

A feature table is any table having one or more columns whose SQL Type is drawn from the set of Geometry SQL Types defined in section 3.2.3. The set of feature tables in a database can be determined from the TABLES and COLUMNS metadata views in the SQL92 INFORMATION_SCHEMA. The set of feature tables can also be determined by querying the GEOMETRY_COLUMNS metadata view as described below.

2.3.2 Geometry Columns Metadata Views

Each geometry column will be represented as a row in the standard COLUMNS metadata view in the SQL92 INFORMATION_SCHEMA. Spatial Reference System Identity is however not a standard part of the SQL92 INFORMATION_SCHEMA. To represent this information we introduce an additional metadata view named GEOMETRY_COLUMNS.

The GEOMETRY_COLUMNS table or view consists of a row for each geometry column in the database. The data stored for each geometry column includes the identity of the feature table of which it is a member, the spatial reference system ID, the type of geometry for the column, and the coordinate dimension.

The columns in the GEOMETRY_COLUMNS metadata view for the SQL92 with Geometry Types environment are a subset of the columns in the GEOMETRY_COLUMNS view defined for the SQL92 environment.

2.3.3 Spatial Reference System Information Views

Every geometry column is associated with a Spatial Reference System. The Spatial Reference System identifies the coordinate system for all geometries stored in the column, and gives meaning to the numeric coordinate values for any geometry instance stored in the column. Examples of commonly used Spatial Reference Systems include ‘Latitude Longitude’, and ‘UTM Zone 10’.

The SPATIAL_REFERENCE_SYSTEMS table stores information on each Spatial Reference System in the database. The columns of this table are the Spatial Reference System Identifier (SRID), the Spatial
Reference System Authority Name (AUTH_NAME), the Authority Specific Spatial Reference System Identifier (AUTH_SRID) and the Well-known Text description of the Spatial Reference System (SRTEXT). The Spatial Reference System Identifier (SRID) constitutes a unique integer key for a Spatial Reference System within a database.

Interoperability between clients is achieved via the SRTEXT column which stores the Well-known Text representation for a Spatial Reference System as described in section 3.4.

The Spatial Reference System Information View for the SQL92 with Geometry Types implementation is identical to the Spatial Reference System Information View for the SQL92 implementation.

2.3.4 Feature Tables and Views

A Feature is an object with geometric attributes [1]. Feature are stored in tables, each geometric attribute is stored in a geometric column whose type is drawn from the set of SQL Geometry Types described in section 3.2.3. Relationships between Features are defined as FOREIGN KEY references between feature tables.

2.3.5 Background Information on SQL Abstract Data Types

The term Abstract Data Type (ADT) refers to a data type that extends the SQL type system.

ADT types can be used to define the column types for tables, this allows values stored in the columns of a table to be instances of ADTs.

SQL functions may be declared to take ADT values as arguments, and return ADT values as results.

An ADT may be defined as a subtype of another ADT, referred to as its supertype. This allows an instance of the subtype to be stored in any column where an instance of the supertype is expected and allows an instance of the subtype to be used as an argument or return value in any SQL function that is declared to use the super type as an argument or return value.

The above definition of ADTs is value based, and value based ADTs with the above properties are defined as part of the current draft SQL3 standard.

SQL implementations that support Abstract Data Types may also support the concept of References to Abstract Data Type instances that are stored as rows in a table whose type corresponds to the type of the Abstract Data Type. The terms RowType and Reference to RowType are also used to describe such types. The above concepts of Types that support tables whose rows are instances of the Type and that support References to Type instances are also part of the current draft SQL3 standard.

This specification allows Geometry Types to be implemented as either pure value based Types or as Types that support persistent References.

2.3.6 Scope of this OpenGIS Geometry Types specification

This specification does not attempt to standardize and does not depend upon any part of the mechanism by which Types are added and maintained in the SQL environment including

- The syntax and functionality provided for defining types
- The syntax and functionality provided for defining SQL functions
- The physical storage of type instances in the database
• The specific terminology used to refer to types, for example, ADT.

This specification does standardize:

• The names and geometric definitions of the OpenGIS SQL Types for Geometry.
• The names, signatures and geometric definitions of the OpenGIS SQL Functions for Geometry.

The types for geometry are defined in black box terms, i.e. all access to information about a geometry type instance is through SQL functions. No attempt is made to distinguish functions that may access type instance attributes (such as the dimension of a geometry instance) from functions that may compute values given a type instance (such as the centroid of a polygon). In particular, a SQL3 implementation of this specification would be free to nominate any set of functions as observer methods on attributes of an Abstract Data Type in SQL3 as long as the signatures of the SQL functions described in this specification are preserved.

This specification does not place any requirements on when or how or who defines the Geometry Types. In particular, a compliant system may be shipped to the database user with the set of Geometry Types and Functions already built into the RDBMS server, or with the set of Geometry Types and Functions supplied to the database user as a dynamically loaded extension to the RDBMS server or in any other manner not mentioned in this specification.

2.3.7 SQL Geometry Type Hierarchy

The SQL Geometry Types are organized into a type hierarchy based on the Open GIS Geometry Model and are shown in the figure below.

![SQL Geometry Type Hierarchy Diagram](image-url)

**Figure 2.15—SQL Geometry Type Hierarchy**
The root type, named Geometry, has subtypes for Point, Curve, Area and GeometryCollection. A GeometryCollection is a Geometry that is a collection of possibly heterogeneous Geometries. MultiPoint, MultiCurve and MultiSurface are specific subtypes of GeometryCollection used to manage homogenous collections of Points, Curves and Surfaces. The 0 dimensional geometric Types are Point and MultiPoint. The one-dimensional geometric Types are Curve and MultiCurve together with their subclasses. The two-dimensional geometric Types are Surface and MultiSurface together with their subclasses.

SQL functions are defined to construct instances of the above types given well-known textual or binary representations of the types. SQL functions defined on the types implement the methods described in the Geometry Model of section 2.1.

2.3.8 Geometry Values and Spatial Reference Systems

In order to model Spatial Reference System information each geometry value in the SQL92 with Geometry Types implementation is associated with a Spatial Reference System. Capturing this association at the level of the individual geometry value allows literal geometry values that are not yet part of a column in the database, to be associated with a Spatial Reference System. Examples of such geometry values are geometry values that are used as a parameter to a spatial query or a geometry value that is part of an insert statement. Capturing this association at the level of the individual geometry value also allows functions that take two geometry values to check for compatible spatial reference systems.

A geometry value is associated with a Spatial Reference System by storing the Spatial Reference System Identity (SRID) for the Spatial Reference System as a part of the geometry value. As explained in the Spatial Reference System Metadata views, each Spatial Reference System in the database is identified by a unique value of SRID.

The SRID for a geometry is assigned to it at construction time. This allows the SQL92 with Geometry Types implementation to ensure that

1. the geometry values being inserted into a geometry column match the Spatial Reference System declared for the geometry column
2. queries that spatially join columns from different tables operate on geometry columns with compatible Spatial Reference Systems.

If either of these conditions are violated, a run time SQL error is generated. These compatible spatial reference system checks are not possible in the SQL92 implementation.

The SRID function, defined on the Geometry type, returns the integer SRID of a geometry value.

In all operations on the Geometry type, geometric calculations shall be done in the spatial reference system of the first geometric object. Returned objects shall be in the spatial reference system of the first geometric object unless explicitly stated otherwise.

Before a geometry can be constructed and inserted into a table, the corresponding row for its SRID must exist in the SPATIAL_REFERENCE_SYSTEMS table, else construction of the geometry will fail. When defining a table, a SQL check constraint can be used to enforce the rule that all geometries in a geometry column have the same SRID as that defined for the column in the GEOMETRY_COLUMNS table. The following example shows the definition of a table, named Countries, with two columns named Name and Geometry of type VARCHAR and POLYGON respectively.

```
CREATE TABLE Countries (
    Name VARCHAR(200) NOT NULL PRIMARY KEY,
    Location Polygon NOT NULL,
)```
CONSTRANIT spatial_reference
CHECK (SRID(Geometry) in (SELECT SRID from GEOMETRY_COLUMNS where
F_TABLE_CATALOG = <catalog> and F_TABLE_SCHEMA = <schema> and
F_TABLE_NAME = 'Countries' and F_GEOMETRY_COLUMN = 'Location'))

We expect that most implementations will use Stored Procedures similar to those shown below for the purpose of adding and dropping geometry columns to and from a feature table.

The `AddGeometryColumn` procedure will:

1. ensure that an entry for the SRID exists in the SPATIAL_REFERENCE_SYSTEMS table.
2. add an entry to the GEOMETRY_COLUMNS table that stores the SRID for the geometry column.
3. add the geometry column to the feature table using a SQL ALTER TABLE statement
4. add the Spatial Reference Check Constraint to the feature table

The `DropGeometryColumn` stored procedure will:

1. drop the spatial reference Check Constraint on the feature table
2. drop the entry from the GEOMETRY_COLUMNS table
3. drop the geometry column from the feature table

2.3.9 ODBC Access to Geometry Values in the SQL with Geometry Types case

Spatial data are accessed using the SQL query language extended with SQL functions on Geometry Types as described in section 3.2.3. The SQL pass through capabilities of ODBC allow a client to pass these or any extended SQL statements containing RDBMS specific SQL extensions to a server. (Applications are free to send any SQL statements to an RDBMS even if the statement is not described within the ODBC conformance levels).

Geometry columns are implemented using the Geometry data types described above.

GIS applications will be able to determine the existence of a Geometry column based on the Geometry data type or one of its subtypes using one or more of the following ODBC programming techniques:

The SQLTypeInfo function can be used to determine both the TYPE_NAME and the underlying SQL_DATA_TYPE of an ODBC SQL Type.

The SQLColumns catalog function can be used to determine the TYPE_NAME and the underlying SQL_DATA_TYPE of a column in a table.

The SQLDescribeCol and SQLColAttributes functions can be used to determine a column’s data type and description.

An ODBC client application uses either one of two SQL functions

GeomFromText ([in] String, [in] Integer) : Geometry, or
GeomFromWKB([in] Binary,[ in] Integer) : Geometry

or their type specific versions (for example, PolygonFromText and PolygonFromWKB) to pass geometry values into the database from a client application that represents them using either the well-known text or the well-known binary representations.

The input arguments to the above functions are ODBC standard character, binary and integer data types (SQL_C_CHAR, SQL_C_BINARY, SQL_C_INTEGER) and clients bind to these parameters using standard ODBC binding methods.

An ODBC client application uses either one of two SQL functions

AsText([in]Geometry) : String, or

AsBinary([in]Geometry) : Binary

to extract geometry values from the database as either text or well-known binary values.

The output arguments to the above functions are ODBC standard character and binary data types (SQL_C_CHAR, SQL_C_BINARY) and clients bind to these parameters using standard ODBC binding methods.

The above SQL functions are described in sections 3.2.8 and 3.2.9.
In order to be compliant with this OpenGIS ODBC/SQL specification for geospatial feature collections an implementer shall choose to implement the components described in this section for **any one of three** alternatives (1a, 1b or 2) listed below and described in this specification:

1. **SQL92** implementation of feature tables
   a) using numeric SQL types for geometry storage and ODBC access.
   b) using binary SQL types for geometry storage and ODBC access.

2. **SQL92 with Geometry Types** implementation of feature tables supporting both textual and binary ODBC access to geometry.

The components for the **SQL92** implementation of feature tables are described in section 3.1. Alternatives 1a) and 1b) listed above differ only in the implementation of the geometry table component as described in section 3.1.4.

The components for the **SQL92 with Geometry Types** implementation of feature tables are described in section 3.2.

### 3.1 Components—SQL92 Implementation of Feature Tables

The components of the ODBC OpenGIS specification for feature table implementation in a SQL92 environment consists of the tables or views discussed in this section. Since the existence of some unknown table is prerequisite for a view, most of the definitions below are stated as `CREATE TABLE` statements. Views that create the same logical structure are equally compliant. *Table names and column names have been restricted to 18 characters in length to allow for the widest possible implementation.*

#### 3.1.1 Spatial Reference System Information

**3.1.1.1 Component Overview**

The Spatial Reference Systems table, which is named `SPATIAL_REF_SYS`, stores information on each spatial reference system used in the database.
3.1.1.2 Table or View Constructs

The following CREATE TABLE statement creates an appropriately structured Spatial Reference Systems table.

```
CREATE TABLE SPATIAL_REF_SYS
(  
  SRID INTEGER NOT NULL PRIMARY KEY,
  AUTH_NAME VARCHAR (256),
  AUTH_SRID INTEGER,
  SRTEXT VARCHAR (2048)
)
```

3.1.1.3 Field Description

The meanings of the attributes in the view are as follows:

- **SRID**—an integer value that uniquely identifies each Spatial Reference System within a database.
- **AUTH_NAME**—the name of the standard or standards body that is being cited for this reference system. EPSG would be a valid AUTH_NAME.
- **AUTH_SRID**—the ID of the Spatial Reference System as defined by the Authority cited in AUTH_NAME.
- **SRTEXT**—The Well-known Text representation of the Spatial Reference System.

3.1.1.4 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.1.2 Geometry Columns Metadata View

3.1.2.1 Component Overview

The Geometric Columns Metadata view provides metadata information on the spatial reference for each geometry column in the database.

3.1.2.2 Table or View Constructs

The following CREATE TABLE statement creates an appropriately structured table. This should be either an actual table or an updateable view so that insertion of reference system information can be done directly with SQL.

```
CREATE TABLE GEOMETRY_COLUMNS (  
  F_TABLE_CATALOG VARCHAR(256) NOT NULL,
  F_TABLE_SCHEMA VARCHAR(256) NOT NULL,
  F_TABLE_NAME VARCHAR(256) NOT NULL,
  F_GEOMETRY_COLUMN VARCHAR(256) NOT NULL,
  G_TABLE_CATALOG VARCHAR(256) NOT NULL,
  G_TABLE_SCHEMA VARCHAR(256) NOT NULL,
  G_TABLE_NAME VARCHAR(256) NOT NULL,
  STORAGE_TYPE INTEGER,
)
3.1.2.3 Field Description

The fields in the Geometric Complex Information view are:

- **F_TABLE_CATALOG, F_TABLE_SCHEMA, F_TABLE_NAME**—the fully qualified name of the feature table containing the geometry column.

- **F_GEOMETRY_COLUMN**—the name of the column in the feature table that is the geometry column. This column will contain a foreign key reference into the geometry table for a SQL92 implementation.

- **G_TABLE_CATALOG, G_TABLE_SCHEMA, G_TABLE_NAME**—the name of the geometry table and its schema and catalog. The geometry table implements the geometry column.

- **STORAGE_TYPE**—the type of storage being used for this geometry column.
  
  0 = normalized geometry SQL92 implementation.
  
  1 = binary geometry SQL92 implementation (Well-known Binary Representation for Geometry).

- **GEOMETRY_TYPE**—the type of geometry values stored in this column. The use of a non-leaf geometry class name from the Geometry Object Model described in section 3.1 for a geometry column implies that domain of the column corresponds to instances of the class and all of its subclasses.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GEOMETRY</td>
</tr>
<tr>
<td>1</td>
<td>POINT</td>
</tr>
<tr>
<td>2</td>
<td>CURVE</td>
</tr>
<tr>
<td>3</td>
<td>LINESTRING</td>
</tr>
<tr>
<td>4</td>
<td>SURFACE</td>
</tr>
<tr>
<td>5</td>
<td>POLYGON</td>
</tr>
<tr>
<td>6</td>
<td>COLLECTION</td>
</tr>
<tr>
<td>7</td>
<td>MULTIPoint</td>
</tr>
<tr>
<td>8</td>
<td>MULTICURVE</td>
</tr>
<tr>
<td>9</td>
<td>MULTILINESTRING</td>
</tr>
<tr>
<td>10</td>
<td>MULTISURFACE</td>
</tr>
<tr>
<td>11</td>
<td>MULTIPOLYGON</td>
</tr>
</tbody>
</table>

- **COORD_DIMENSION**—the number of ordinates used in the complex, usually corresponds to the number of dimensions in the spatial reference system.

- **MAX_PPR**—(This value contains data for the normalized SQL92 geometry implementation only) points per row, the number of points stored as ordinate columns in the geometry table.

- **SRID**—the ID of the spatial reference system used for the coordinate geometry in this table. It is a foreign key reference to the SPATIAL_REF_SYS table.
3.1.2.4 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns for ODBC.

3.1.3 Feature Tables and Views

The basic restriction in this specification for feature tables is that for each geometric attribute they include geometry via a FOREIGN KEY reference to a geometry table. Feature-to-feature relations would similarly be defined as FOREIGN KEY references. By [1], features are simply objects that have geometric attributes. In SQL92, these geometric attributes are stored in the geometry tables.

The general format of a feature table would be as follows:

```
CREATE TABLE <feature-name>  (
   <FID name> <FID type>,
   <feature attributes> <other FID type> REFERENCES <other feature view>,
   ... (other FID based attributes for feature relations)
   ... (other attributes for feature)
   <geometry attribute 1> <GID type>,
   ... (other geometric attributes for feature)
   PRIMARY KEY <FID name>,
   ... (other geometric attributes foreign key statements)
   FOREIGN KEY <geometric attribute 1> REFERENCES <geometry-table-name-1>,
   FOREIGN KEY <FID relation name> REFERENCES <FEATURE table> <other FID name>,
   ... (other geometric attributes foreign key statements)
)
```

The geometric attribute Foreign Key reference applies only for the case where the geometry table stores geometry in binary form. In the case where geometry is stored in normalized form there may be multiple rows in the geometry table corresponding to a single geometry value. In this case the geometry attribute reference may be captured by a constraint that checks that the geometry column value stored in the Feature Table corresponds to the GID value for some row in the Geometry Table.

The foreign key reference to the geometry table name creates an entry in the data dictionary that ties this table to that geometry table. This is sufficient to identify this table as a feature table. Foreign keys also define feature-to-feature relations. Alternatively, applications may check the GEOMETRY_COLUMNS view, where all geometry columns and their associated feature tables and geometry tables are listed.

3.1.4 Geometry Tables or Views

3.1.4.1 Component Overview

Each Geometry View stores geometry instances corresponding to a geometry column in a feature table. Geometries may be stored as individual ordinate values, using SQL types, or as binary objects, using the OpenGIS Well-known Binary Representation for Geometry. Table schemas for both implementations are provided.
3.1.4.2 Geometry stored using ODBC/SQL numeric types

3.1.4.3 Table or View Constructs

The following CREATE TABLE statement creates an appropriately structured table for geometry stored as individual ordinate values using SQL types. Implementations should either use this table format or provide stored procedures to create, populate and maintain this table.

```
CREATE TABLE <table name> (  
    GID NUMBER NOT NULL,  
    ESEQ INTEGER NOT NULL,  
    ETYPE INTEGER NOT NULL,  
    SEQ INTEGER NOT NULL,  
    X1 <ordinate type>,  
    Y1 <ordinate type>,  
    ... <repeated for each ordinate, repeated for each point>  
    X<max_ppr> <ordinate type>,  
    Y<max_ppr> <ordinate type>,  
    ...,  
    <attribute name> <attribute type>  
)  
CONSTRANT GID_PK PRIMARY KEY (GID, ESEQ, SEQ)
```

3.1.4.4 Field Descriptions:

The fields of a geometric view are:

- **GID**—identity of this geometry
- **ESEQ**—identifies multiple components within a geometry
- **ETYPE**—element type of this primitive element for the geometry. The following values are defined for ETYPE:
  
  1 = Point  
  2 = LineString  
  3 = Polygon

- **SEQ**—identifies the sequence of rows to define a geometry component
- **X1**—first ordinate of first point
- **Y1**—second ordinate of first point
- ...—(repeated for each ordinate, for this point)
- ...—(repeated for each coordinate, for this row)
• \(X_{\text{MAX PPR}}\)--first ordinate of last point. The maximum number of points per row \(\text{MAX PPR}\) is consistent with the information in the \texttt{GEOMETRY_COLUMNS} table.

• \(Y_{\text{MAX PPR}}\)--second ordinate of last point

• ...—(repeated for each ordinate, for this last point)

• \(<\text{ATTRIBUTE}>\)—other attributes can be carried in the geometry view for specific feature schema

### 3.1.4.5 Geometry stored using ODBC/SQL binary types

### 3.1.4.6 Table or View Constructs

The following \texttt{CREATE TABLE} statement creates an appropriately defined table for geometry stored using the OpenGIS Well-known Binary Representation for Geometry defined in section 4.3. Implementations should either use this table format or provide stored procedures to create, populate and maintain this table.

```sql
CREATE TABLE <table name> (  
  GID NUMBER NOT NULL PRIMARY KEY,
  XMIN <ordinate type>,
  YMIN <ordinate type>,
  XMAX <ordinate type>,
  YMAX <ordinate type>,
  WKB_GEOMETRY VARBINARY,
  <attribute name> <attribute type>
)
```

### 3.1.4.7 Field Descriptions

The fields of a geometric view are:

• \(\text{GID}\)—identity of this geometry

• \(\text{XMIN}\)—the minimum x-coordinate of the geometry bounding box

• \(\text{YMIN}\)—the minimum y-coordinate of the geometry bounding box

• \(\text{XMAX}\)—the maximum x-coordinate of the geometry bounding box

• \(\text{YMAX}\)—the maximum y-coordinate of the geometry bounding box

• \(\text{WKB_GEOMETRY}\)—the well-known binary representation of the geometry

• \(<\text{ATTRIBUTE}>\)—other attributes can be carried in the geometry view for specific feature schema

### 3.1.4.8 Exceptions, Errors, and Error Codes

Error handling will use the standard SQL status returns for ODBC.
3.1.5 Operators

No SQL92 spatial operators are defined as part of this specification.

3.2 Components—SQL92 with Geometry Types Implementation of Feature Tables

The components of the ODBC OpenGIS specification for feature table implementation in a SQL92 with Geometry Types environment consists of the tables or views, SQL types and SQL functions discussed in this section.

Since the existence of some unknown table is prerequisite for a view, most of the definitions below are stated as \texttt{CREATE TABLE} statements. Views that create the same logical structure are equally compliant.

3.2.1 Spatial Reference System Information View

3.2.1.1 Component Overview

This component is identical to the corresponding Component described for the SQL92 implementation:

3.2.1.2 Table or View Constructs

The following \texttt{CREATE TABLE} statement creates an appropriately structured Spatial Reference Systems table.

\begin{verbatim}
CREATE TABLE SPATIAL_REF_SYS

  ( SRID INTEGER NOT NULL PRIMARY KEY,
    AUTH_NAME VARCHAR (256),
    AUTH_SRID INTEGER,
    SRTEXT VARCHAR (2048)
  )
\end{verbatim}

3.2.1.3 Field Description

The meanings of the attributes in the view are as follows:

- \texttt{SRID}—an integer value that uniquely identifies each Spatial Reference System within a database.
- \texttt{AUTH_NAME}—the name of the standard or standards body that is being cited for this reference system. EPSG would be a valid \texttt{AUTH_NAME}
- \texttt{AUTH_SRID}—the ID of the Spatial Reference System as defined by the Authority cited in \texttt{AUTH_NAME}.
- \texttt{SRTEXT}—The Well-known Text representation of the Spatial Reference System.

3.2.1.4 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.
3.2.2 Geometry Columns Metadata View

3.2.2.1 Component Overview

The Geometric Columns Information view provides metadata information on the spatial reference for each geometry column in the database. The columns for this view in the SQL92 with Geometry Types implementation are a subset of the columns in the SQL92 implementation.

3.2.2.2 Table or View Constructs

The following CREATE TABLE statement creates an appropriately structured table. This should be either an actual table or an updateable view so that insertion of reference system information can be done directly with SQL.

```sql
CREATE TABLE GEOMETRY_COLUMNS (  
    F_TABLE_CATALOG VARCHAR(256) NOT NULL,  
    F_TABLE_SCHEMA VARCHAR(256) NOT NULL,  
    F_TABLE_NAME VARCHAR(256) NOT NULL,  
    F_GEOMETRY_COLUMN VARCHAR(256) NOT NULL,  
    COORD_DIMENSION INTEGER,  
    SRID INTEGER REFERENCES SPATIAL_REF_SYS,  
    CONSTRAINT GC_PK PRIMARY KEY  
        (F_TABLE_CATALOG, F_TABLE_SCHEMA, F_TABLE_NAME, F_GEOMETRY_COLUMN)  
)
```

3.2.2.3 Field Description

The fields in the Geometric Complex Information view are:

- **F_TABLE_CATALOG, F_TABLE_SCHEMA, F_TABLE_NAME**—the fully qualified name of the feature table containing the geometry column.

- **F_GEOMETRY_COLUMN**—the name of the geometry column in the feature table.

- **COORD_DIMENSION**—the coordinate dimension for the geometry values in this column, which will be equal to the number of dimensions in the spatial reference system.

- **SRID**—the ID of the spatial reference system used for the coordinate geometry in this table. It is a foreign key reference to the SPATIAL_REFERENCES table.

3.2.2.4 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns for ODBC.

3.2.3 SQL Geometry Types

3.2.3.1 Component Overview

The SQL Geometry Types extend the set of available SQL92 types to include Geometry Types.
3.2.3.2 Language Constructs

The SQL language will support a subset of the following set of SQL Geometry Types: \{Geometry, Point, Curve, LineString, Surface, Polygon, GeometryCollection, MultiCurve, MultiLineString, MultiSurface, MultiPolygon, MultiPoint\}. The permissible type subsets that an implementer may choose to implement are described in Table 3.1 below.

An implementation must preserve the subtype relationships between geometry types shown in Figure 3.1 below for the types that are implemented. An implementation that implements 2 types A and B where B is an immediate subtype of A in Figure 3.1 is free to introduce additional types C, outside the scope of this specification, between A and B as long as A continues to be a supertype of B.

![Figure 3.1—Subtype relationships between Types](image)

Geometry, Curve, Surface, MultiCurve and MultiSurface are defined to be non-instantiable types. No constructors are defined for these types.

The remaining seven types are defined to be instantiable. An implementation may support only a subset of these seven types as instantiable as defined in the table below.

<table>
<thead>
<tr>
<th>Type Level</th>
<th>Available Types</th>
<th>Instantiable Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Geometry, Point, Curve, LineString, Surface, Polygon, GeometryCollection</td>
<td>Point, LineString, Polygon, GeometryCollection</td>
</tr>
<tr>
<td>2</td>
<td>Geometry, Point, Curve, LineString, Surface, Polygon, GeometryCollection, MultiPoint, MultiCurve, MultiLineString, MultiSurface, MultiPolygon</td>
<td>Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon</td>
</tr>
</tbody>
</table>
Any implemented SQL geometry type may be used as the type for a column. Declaring a column to be of a particular type implies that any instance of the type or of any of its subtypes may be stored in the column.

3.2.4 Feature Tables and Views

3.2.4.1 Component Overview

The basic restriction in this specification for feature tables is that each geometric attribute is modeled using a column whose type corresponds to a SQL Geometry Type as defined in section 3.2.3. Feature-to-feature relations are defined as FOREIGN KEY references.

3.2.4.2 Table or View Constructs

The general format of a feature table in the SQL92 with Geometry Types implementation shall be as follows:

```
CREATE TABLE <feature-name>  (
   <FID name> <FID type>,
   <feature attributes> <other FID type> REFERENCES <other feature view>,
   ... (other FID based attributes for feature relations)
   ... (other attributes for feature)
   <geometry attribute 1> <Geometry type>,
   ... (other geometric attributes for feature)
   PRIMARY KEY <FID name>,
   FOREIGN KEY <FID relation name> REFERENCES <FEATURE table> <other FID name>
   CONSTRAINT SRS_1 CHECK (SRID(<geometry attribute 1>) in (SELECT SRID from GEOMETRY_COLUMNS where F_TABLE_CATALOG = <catalog> and F_TABLE_SCHEMA = <schema> and F_TABLE_NAME = <feature-name> and F_GEOMETRY_COLUMN = <geometry attribute 1>))
   ... (spatial reference constraints for other geometric attributes)
)
```

The use of a SQL Geometry Type for one of the columns in the table identifies this table as a feature table. Alternatively, applications may check the GEOMETRY_COLUMNS view, where all geometry columns and their associated feature tables and geometry tables are listed.

3.2.4.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.
3.2.5 SQL Textual Representation of Geometry

3.2.5.1 Component Overview

Each Geometry Type has a Well-known Text representation that may be used both to construct new instances of the type and to convert existing instances to textual form for alphanumeric display.

3.2.5.2 Language Constructs

The Well-known Text representation of Geometry is defined below; the notation \{\} denotes 0 or more repetitions of the tokens within the braces, the braces do not appear in the output token list. The text representation of the instantiable geometric types implemented shall conform to this grammar.

```
<Geometry Tagged Text> :=
    <Point Tagged Text>
    | <LineString Tagged Text>
    | <Polygon Tagged Text>
    | <MultiPoint Tagged Text>
    | <MultiLineString Tagged Text>
    | <MultiPolygon Tagged Text>
    | <GeometryCollection Tagged Text>

<Point Tagged Text> :=
    POINT <Point Text>

<LineString Tagged Text> :=
    LINESTRING <LineString Text>

<Polygon Tagged Text> :=
    POLYGON <Polygon Text>

<MultiPoint Tagged Text> :=
    MULTIPOINT <Multipoint Text>

<MultiLineString Tagged Text> :=
    MULTILINESTRING <MultiLineString Text>

<MultiPolygon Tagged Text> :=
    MULTIPOLYGON <MultiPolygon Text>

<GeometryCollection Tagged Text> :=
    GEOMETRYCOLLECTION <GeometryCollection Text>

<Point Text> := EMPTY | { <Point> }

<Point> ::= <x> <y>

<x> ::= double precision literal

<y> ::= double precision literal

<LineString Text> := EMPTY
    | { <Point> }, <Point> |

<Polygon Text> := EMPTY
    | { <LineString Text> }, <LineString Text> |

<Multipoint Text> := EMPTY
    | { <Point Text> }, <Point Text> |

<MultiLineString Text> := EMPTY
```
The above grammar has been designed to support a compact and readable textual representation of geometric instances. The representation of a geometry that consists of a set of homogeneous components does not include the tags for each embedded component.

### 3.2.5.3 Examples

Examples of SQL textual representations of Geometry Types are shown below. The coordinates are shown as integer values; coordinates may be any double precision value.

<table>
<thead>
<tr>
<th>Geometry Type</th>
<th>SQL Text Literal Representation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>'POINT (10 10)'</td>
<td>aPoint</td>
</tr>
<tr>
<td>LineString</td>
<td>'LINESTRING (10 10, 20 20, 30 40)'</td>
<td>aLineString with 3 points</td>
</tr>
<tr>
<td>Polygon</td>
<td>'POLYGON ((10 10, 10 20, 20 20, 20 15, 10 10))'</td>
<td>aPolygon with 1 exterior ring and 0 interior rings</td>
</tr>
<tr>
<td>Multipoint</td>
<td>'MULTIPOINT (10 10, 20 20)'</td>
<td>aMultiPoint with 2 point</td>
</tr>
<tr>
<td>MultiLineString</td>
<td>'MULTILINESTRING ((10 10, 20 20), (15 15, 30 15))'</td>
<td>aMultiLineString with 2 linestrings</td>
</tr>
<tr>
<td>MultiPolygon</td>
<td>'MULTIPOLYGON (((10 10, 10 20, 20 20, 20 15, 10 10)), ((60 60, 70 70, 80 60, 60 60)))'</td>
<td>aMultiPolygon with 2 polygons</td>
</tr>
<tr>
<td>GeomCollection</td>
<td>'GEOMETRYCOLLECTION (POINT (10 10), POINT (30 30), LINESTRING (15 15, 20 20))'</td>
<td>aGeometryCollection consisting of 2 Point values and a LineString value</td>
</tr>
</tbody>
</table>

### 3.2.6 SQL Functions for Constructing a Geometry Value given its Well-known Text Representation

#### 3.2.6.1 Component Overview

The functions are used to construct Geometry instances from their text representations.

#### 3.2.6.2 Language Constructs

The `GeomFromText` function, takes a geometry textual representation (a `<Geometry Tagged Text>` as described in the grammar above), and a Spatial Reference System ID (SRID) and creates an instance of the appropriate geometry type. This function plays the role of the Geometry factory in SQL.

An implementation shall substitute an SQL type suitable for representing text data (e.g., `VARCHAR`) for the type `String` below.
Chapter 3  Component Specifications

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeomFromText(geometryTaggedText String, SRID Integer) : Geometry</td>
<td>Construct a Geometry value given its well-known textual representation.</td>
</tr>
</tbody>
</table>

The return type of the Geometry function is the Geometry supertype. For construction of Geometry values to be stored in columns restricted to a particular subtype, an implementation shall also provide a type specific construction function for each instantiable subtype as described in the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointFromText (pointTaggedText String, SRID Integer) : Point</td>
<td>Construct a Point</td>
</tr>
<tr>
<td>LineFromText (lineStringTaggedText String, SRID Integer) : LineString</td>
<td>Construct a LineString</td>
</tr>
<tr>
<td>PolyFromText (polygonTaggedText String, SRID Integer) : Polygon</td>
<td>Construct a Polygon</td>
</tr>
<tr>
<td>MPointFromText (multiPointTaggedText String, SRID Integer) : MultiPoint</td>
<td>Construct a MultiPoint</td>
</tr>
<tr>
<td>MLineFromText (multiLineStringTaggedText String, SRID Integer) : MultiLineString</td>
<td>Construct a MultiLineString</td>
</tr>
<tr>
<td>MPolyFromText (multiPolygonTaggedText String, SRID Integer) : MultiPolygon</td>
<td>Construct a MultiPolygon</td>
</tr>
<tr>
<td>GeomCollFromTxt (geometryCollectionTaggedText String, SRID Integer) : GeomCollection</td>
<td>Construct a GeometryCollection</td>
</tr>
</tbody>
</table>

As an optional feature, an implementation may also support ‘building’ of Polygon or MultiPolygon values given an arbitrary collection of possibly intersecting rings or closed LineString values. Implementations that support this feature should include the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BdPolyFromText (multiLineStringTaggedText String, SRID Integer) : Polygon</td>
<td>Construct a Polygon given an arbitrary collection of closed linestrings as a MultiLineString text representation.</td>
</tr>
<tr>
<td>BdMPolyFromText (multiLineStringTaggedText String, SRID Integer) : MultiPolygon</td>
<td>Construct a MultiPolygon given an arbitrary collection of closed linestrings as a MultiLineString text representation.</td>
</tr>
</tbody>
</table>

3.2.6.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.6.4 Example

The following example shows the use of the Polygon type specific constructor:

```sql
INSERT INTO Countries (Name, Location)
VALUES ('Kenya', PolygonFromText('POLYGON ((x y, x y, x y, ..., x y)'), 14))
```
3.2.7 SQL Functions for Constructing a Geometry Value given its Well-known Binary Representation

3.2.7.1 Component Overview

The functions are used to construct geometry instances from their well-known binary representations.

3.2.7.2 Language Constructs

The `GeomFromWKB` function, takes a well-known binary representation of geometry (WKBGeometry as described in section 3.3) and a Spatial Reference System ID (SRID) and creates an instance of the appropriate geometry type. This function plays the role of the Geometry Factory in SQL. An implementation shall substitute an SQL type used to represent binary values for the type `Binary` in the definitions below.

```
GeomFromWKB (WKBGeometry Binary, SRID Integer) : Geometry
```

Construct a Geometry value given its well-known binary representation.

The return type of the `Geometry` function is the `Geometry` supertype. For construction of `Geometry` values to be stored in columns restricted to a particular subtype, an implementation shall also provide a type specific construction function for each instantiable subtype as described in the table below (the well-known binary representations for each `Geometry` type are as described in section 3.3).

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PointFromWKB</code> (WKBPoint Binary, SRID Integer): Point</td>
<td>Point</td>
<td>Construct a Point</td>
</tr>
<tr>
<td><code>LineFromWKB</code> (WKBLinestring Binary, SRID Integer): LineString</td>
<td>LineString</td>
<td>Construct a LineString</td>
</tr>
<tr>
<td><code>PolyFromWKB</code> (WKBPolygon Binary, SRID Integer): Polygon</td>
<td>Polygon</td>
<td>Construct a Polygon</td>
</tr>
<tr>
<td><code>MPointFromWKB</code> (WKBMultipoint Binary, SRID Integer): Multipoint</td>
<td>Multipoint</td>
<td>Construct a Multipoint</td>
</tr>
<tr>
<td><code>MLineFromWKB</code> (WKBMultilinestring Binary, SRID Integer): Multilinestring</td>
<td>Multilinestring</td>
<td>Construct a Multilinestring</td>
</tr>
<tr>
<td><code>MPolyFromWKB</code> (WKBMultipolygon Binary, SRID Integer): Multipolygon</td>
<td>Multipolygon</td>
<td>Construct a Multipolygon</td>
</tr>
<tr>
<td><code>GeomCollFromWKB</code> (WKBGeometryCollection Binary, SRID Integer): GeomCollection</td>
<td>GeometryCollection</td>
<td>Construct a GeometryCollection</td>
</tr>
</tbody>
</table>

As an optional feature, an implementation may also support the ‘building’ of `Polygon` or `MultiPolygon` values given an arbitrary collection of possibly intersecting rings or closed `LineString` values. Implementations that support this feature should include the following functions:

```
BdPolyFromWKB (WKBMultilinestring Binary, SRID Integer): Polygon
```

Construct a Polygon given an arbitrary collection of closed linestrings as a `Multilinestring` binary representation.
### 3.2.7.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

#### 3.2.7.4 Examples

The following example shows the use of the binary Polygon type specific constructor in Dynamic SQL, the :wkb and :srid parameters are bound to application program variables containing the binary representation of a Polygon and of the SRID respectively:

```sql
INSERT INTO Countries (Name, Location)
VALUES ('Kenya', PolygonFromWKB(:wkb, :srid))
```

### 3.2.8 SQL functions for obtaining the Well-known Text Representation of a Geometry

#### 3.2.8.1 Component Overview

This function returns the well-known textual representation for a Geometry.

#### 3.2.8.2 Language Constructs

The `AsText` function takes a single argument of type `Geometry` and returns its well-known textual representation. This function applies to all subtypes of `Geometry`.

```sql
AsText (g Geometry) : String
```

**Returns the well-known textual representation**

#### 3.2.8.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

#### 3.2.8.4 Examples

The following example shows the use of the `AsText` function to extract the name and textual representation of geometry of all countries whose names begin with the letter K.

```sql
SELECT Name, AsText(Location) FROM Countries WHERE Name LIKE 'K%'
```

### 3.2.9 SQL functions for obtaining the Well-known Binary Representation of a Geometry

#### 3.2.9.1 Component Overview

This function returns the well-known binary representation for a Geometry.
3.2.9.2 Language Constructs

The `AsBinary` function takes a single argument of type `Geometry` and returns its well-known binary representation. This function applies to all subtypes of `Geometry`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AsBinary (g Geometry)</code></td>
<td>Returns the well-known binary representation</td>
</tr>
</tbody>
</table>

3.2.9.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.9.4 Example

The following example shows the use of the `AsBinary` function to extract the name and well-known binary representation of geometry for all countries whose names begin with the letter K.

```sql
SELECT Name, AsBinary(Location) FROM Countries WHERE Name LIKE 'K%'
```

3.2.10 SQL Functions on Type Geometry

3.2.10.1 Component Description

In all operations on the `Geometry` type, geometric calculations shall be done in the spatial reference system of the first geometric object. Returned objects shall be in the spatial reference system of the first geometric object unless explicitly stated otherwise.

The following SQL functions apply to all subtypes of `Geometry`.

3.2.10.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Dimension(g Geometry)</code></td>
<td>Returns the dimension of the <code>Geometry</code>, which is less than or equal to the dimension of the coordinate space.</td>
</tr>
<tr>
<td><code>GeometryType(g Geometry)</code></td>
<td>Returns the name of the instantiable subtype of <code>Geometry</code> of which this instance is a member, as a <code>String</code>.</td>
</tr>
<tr>
<td><code>AsText(g Geometry)</code></td>
<td>Returns the well-known textual representation</td>
</tr>
<tr>
<td><code>AsBinary(g Geometry)</code></td>
<td>Returns the well-known binary representation</td>
</tr>
<tr>
<td><code>SRID(g Geometry)</code></td>
<td>Returns the Spatial Reference System ID for this <code>Geometry</code>.</td>
</tr>
<tr>
<td><code>IsEmpty(g Geometry)</code></td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments.</td>
</tr>
<tr>
<td></td>
<td>TRUE if this <code>Geometry</code> corresponds to the empty set.</td>
</tr>
</tbody>
</table>
IsSimple(g Geometry): Integer
The return type is Integer, with a return value of 1 for TRUE, 0 for FALSE, and –1 for UNKNOWN corresponding to a function invocation on NULL arguments.
TRUE if this Geometry is simple, as defined in the Geometry Model.

Boundary(g Geometry) : Geometry
Returns a Geometry that is the combinatorial boundary of g as defined in the Geometry Model.

Envelope(g Geometry) : Geometry
Returns the rectangle bounding g as a Polygon. The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MAXX, MINY), (MAXX, MAXY), (MINX, MAXY), (MINX, MINY)).

3.2.10.3 Exceptions, Errors, and Error Codes
Error handling will be accomplished by using the standard SQL status returns.

3.2.11 SQL Functions on Type Point

3.2.11.1 Component Description
The following SQL functions are defined on Point.

3.2.11.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(p Point)</td>
<td>Return the x-coordinate of Point p as a Double Precision number</td>
</tr>
<tr>
<td>Y(p Point)</td>
<td>Return the y-coordinate of Point p as a Double Precision number</td>
</tr>
</tbody>
</table>

3.2.11.3 Exceptions, Errors, and Error Codes
Error handling will be accomplished by using the standard SQL status returns.

3.2.12 SQL Functions on Type Curve

3.2.12.1 Component Overview
The following SQL functions apply to all subtypes of Curve.

3.2.12.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartPoint(c Curve) : Point</td>
<td>Return a Point containing the first point of c</td>
</tr>
<tr>
<td>EndPoint(c Curve) : Point</td>
<td>Return a Point containing the last point of c</td>
</tr>
</tbody>
</table>
IsClosed(c Curve) : Integer

The return type is Integer, with a return value of 1 for TRUE, 0 for FALSE, and –1 for UNKNOWN corresponding to a function invocation on NULL arguments.

Return TRUE if c is closed, i.e., if StartPoint(c) = EndPoint(c)

IsRing(c Curve) : Integer

The return type is Integer, with a return value of 1 for TRUE, 0 for FALSE, and –1 for UNKNOWN corresponding to a function invocation on NULL arguments.

Return TRUE if c is a Ring, i.e., if c is closed and simple. A simple curve does not pass through the same point more than once.

Length(c Curve) : Double Precision

Return the length of c

3.2.12.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.13 SQL Functions on Type LineString

3.2.13.1 Component Overview:

The following SQL functions apply to LineString.

3.2.13.2 Language Constructs:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPoints(l LineString) : Integer</td>
<td>Return the number of points in the LineString.</td>
</tr>
<tr>
<td>PointN(l LineString, n Integer) : Point</td>
<td>Return a Point containing point n of l</td>
</tr>
</tbody>
</table>

3.2.13.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.14 SQL Functions on Type Surface

3.2.14.1 Component Overview

The following SQL functions apply to all subtypes of Surface.

3.2.14.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid(s Surface) : Point</td>
<td>Return the centroid of s, which may lie outside s</td>
</tr>
<tr>
<td>PointOnSurface(s Surface) : Point</td>
<td>Return a Point guaranteed to lie on the surface</td>
</tr>
<tr>
<td>Area(s Surface) : Double Precision</td>
<td>Return the area of s</td>
</tr>
</tbody>
</table>
3.2.14.3  Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.15 SQL Functions on Type Polygon

3.2.15.1  Component Overview

The following SQL functions apply to Polygon.

3.2.15.2  Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExteriorRing(p Polygon) : LineString</td>
<td>Return the exterior ring of p.</td>
</tr>
<tr>
<td>NumInteriorRing(p Polygon) : Integer</td>
<td>Return the number of interior rings.</td>
</tr>
<tr>
<td>InteriorRingN(p Polygon, n Integer) : LineString</td>
<td>Return the nth interior ring. The order of rings is not geometrically significant.</td>
</tr>
</tbody>
</table>

3.2.15.3  Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.16 SQL Functions on Type GeomCollection

3.2.16.1  Component Overview

The following SQL functions apply to GeomCollection and all of its subtypes.

3.2.16.2  Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumGeometries(g GeomCollection) : Integer</td>
<td>Return the number of geometries in the collection.</td>
</tr>
<tr>
<td>GeometryN(g GeomCollection, n Integer) : Geometry</td>
<td>Return the nth geometry in the collection. The order of the elements in the collection is not geometrically significant.</td>
</tr>
</tbody>
</table>

3.2.16.3  Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.17 SQL Functions on Type MultiCurve

3.2.17.1  Component Overview

The following SQL functions apply to all subtypes of MultiCurve.
3.2.17.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsClosed(mc MultiCurve) : Integer</td>
<td>The return type is Integer, with a return value of 1 for TRUE, 0 for FALSE, and –1 for UNKNOWN corresponding to a function invocation on NULL arguments. Return TRUE if mc is closed.</td>
</tr>
<tr>
<td>Length(mc MultiCurve) : Double Precision</td>
<td>Return the length of mc.</td>
</tr>
</tbody>
</table>

3.2.17.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.18 SQL Functions on Type MultiSurface

3.2.18.1 Component Overview

The following SQL functions apply to all subtypes of MultiSurface.

3.2.18.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid(ms MultiSurface) : Point</td>
<td>Return the centroid of ms, which may lie outside ms</td>
</tr>
<tr>
<td>PointOnSurface(ms MultiSurface) : Point</td>
<td>Return a Point guaranteed to lie on the multi surface</td>
</tr>
<tr>
<td>Area(ms MultiSurface) : Double Precision</td>
<td>Return the area of ms</td>
</tr>
</tbody>
</table>

3.2.18.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.19 SQL functions that test Spatial Relationships

3.2.19.1 Component Overview

The following functions test named spatial relationships between two geometries. The specific definitions of these spatial relationships in terms of the DE-9IM may be found in section 2.1.13.2.

3.2.19.2 Language Constructs:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals(g1 Geometry, g2 Geometry) : Integer</td>
<td>The return type is Integer, with a return value of 1 for TRUE, 0 for FALSE, and –1 for UNKNOWN corresponding to a function invocation on NULL arguments. TRUE if g1 and g2 are equal.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Disjoint</strong> ( (g_1, g_2) )</td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments. <code>TRUE</code> if the intersection of ( g_1 ) and ( g_2 ) is the empty set.</td>
</tr>
<tr>
<td><strong>Touches</strong> ( (g_1, g_2) )</td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments. <code>TRUE</code> if the only points in common between ( g_1 ) and ( g_2 ) lie in the union of the boundaries of ( g_1 ) and ( g_2 ).</td>
</tr>
<tr>
<td><strong>Within</strong> ( (g_1, g_2) )</td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments. <code>TRUE</code> if ( g_1 ) is completely contained in ( g_2 ).</td>
</tr>
<tr>
<td><strong>Overlaps</strong> ( (g_1, g_2) )</td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments. <code>TRUE</code> if the intersection of ( g_1 ) and ( g_2 ) results in a value of the same dimension as ( g_1 ) and ( g_2 ) that is different from both ( g_1 ) and ( g_2 ).</td>
</tr>
<tr>
<td><strong>Crosses</strong> ( (g_1, g_2) )</td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments. <code>TRUE</code> if the intersection of ( g_1 ) and ( g_2 ) results in a value whose dimension is less than the maximum dimension of ( g_1 ) and ( g_2 ) and the intersection value includes points interior to both ( g_1 ) and ( g_2 ), and the intersection value is not equal to either ( g_1 ) or ( g_2 ).</td>
</tr>
<tr>
<td><strong>Intersects</strong> ( (g_1, g_2) )</td>
<td>The return type is <code>Integer</code>, with a return value of 1 for <code>TRUE</code>, 0 for <code>FALSE</code>, and –1 for <code>UNKNOWN</code> corresponding to a function invocation on <code>NULL</code> arguments. Convenience predicate: <code>TRUE</code> if the intersection of ( g_1 ) and ( g_2 ) is not empty.</td>
</tr>
</tbody>
</table>

\[
\text{Intersects}(g_1, g_2) \Leftrightarrow \neg \text{Disjoint}(g_1, g_2)
\]
The following function tests if the specified spatial relationship between two geometry values exists, where the spatial relationship is expressed as a string encoding the acceptable values for the DE-9IM between the two geometries, as described in the Geometry Object Model.

<table>
<thead>
<tr>
<th>Contains(g1 Geometry, g2 Geometry) : Integer</th>
<th>The return type is Integer, with a return value of 1 for TRUE, 0 for FALSE, and –1 for UNKNOWN corresponding to a function invocation on NULL arguments. Convenience predicate: TRUE if g2 is completely contained in g1. Contains(g1, g2) ⇔ Within(g2, g1)</th>
</tr>
</thead>
</table>

3.2.19.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

3.2.19.4 Example Queries

The functions and predicates in this section allow the expression of detailed spatial relationship queries.

Return all parcels that intersect a specified polygon:

```
SELECT Parcel.Name, Parcel.Id FROM Parcels
WHERE Intersects(Parcels.Location, PolygonFromWKB(:wkb, :srid)) = 1
```

Return all parcels completely contained in a specified polygon:

```
SELECT Parcel.Name, Parcel.Id FROM Parcels
WHERE Within(Parcels.Location, PolygonFromWKB(:wkb, :srid)) = 1
```

The following adjacency query may be used to select all parcels that are ‘adjacent’ to a query parcel and share one or more boundary lines with a query parcel while excluding parcels that share only corner points.

```
SELECT Parcel.Name, Parcel.Id FROM Parcels
WHERE Touches(Parcels.Location, PolygonFromWKB(:wkb, :srid)) = 1 and Overlaps(Boundary(Parcels.Location), Boundary(PolygonFromWKB(:wkb, :srid))) = 1
```

3.2.20 SQL Functions for Distance Relationships

3.2.20.1 Component Overview

The distance function can be used to calculate the distance between two values of type Geometry.
### 3.2.20.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Distance(g1 Geometry, g2 Geometry) : Double Precision</code></td>
<td>Return the distance between g1 and g2.</td>
</tr>
</tbody>
</table>

### 3.2.20.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

### 3.2.20.4 Example Query

```
SELECT Airport.Name FROM Airports
WHERE Distance(PointFromText(:pointTaggedText, :srid), Airport.Location) < 2000
```

### 3.2.21 SQL Functions that implement Spatial Operators

#### 3.2.21.1 Component Overview

These functions implement set-theoretic and constructive geometry operations on geometry values. These operations are defined for all types of `Geometry`.

#### 3.2.21.2 Language Constructs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Intersection (g1 Geometry, g2 Geometry) : Geometry</code></td>
<td>Return a <code>Geometry</code> that is the set intersection of geometries g1 and g2.</td>
</tr>
<tr>
<td><code>Difference (g1 Geometry, g2 Geometry) : Geometry</code></td>
<td>Return a <code>Geometry</code> that is the closure of the set difference of g1 and g2.</td>
</tr>
<tr>
<td><code>Union (g1 Geometry, g2 Geometry) : Geometry</code></td>
<td>Return a <code>Geometry</code> that is the set union of g1 and g2.</td>
</tr>
<tr>
<td><code>SymDifference(g1 Geometry, g2 Geometry) : Geometry</code></td>
<td>Return a <code>Geometry</code> that is the closure of the set symmetric difference of g1 and g2 (logical XOR of space).</td>
</tr>
<tr>
<td><code>Buffer (g1 Geometry, d Double Precision) : Geometry</code></td>
<td>Return as <code>Geometry</code> defined by buffering a distance <code>d</code> around g1, where <code>d</code> is in the distance units for the Spatial Reference of g1.</td>
</tr>
<tr>
<td><code>ConvexHull(g1 Geometry) : Geometry</code></td>
<td>Return a <code>Geometry</code> that is the convex hull of g1.</td>
</tr>
</tbody>
</table>

### 3.2.21.3 Exceptions, Errors, and Error Codes

Error handling will be accomplished by using the standard SQL status returns.

### 3.2.21.4 Example Query

The following query returns the name of the state and the fragment(s) of the state that fall within the query polygon for each state that intersects the query polygon.

```
SELECT States.Name, Intersection(PolygonFromWKB(:wkb, :srid), States.Location)
FROM States
```
WHERE Intersects(PolygonFromWKB(:wkb, :srid), States.Location)

3.2.22 SQL Function usage and References to Geometry

The SQL Functions that operate on Geometry Types have been defined above to take geometry values as arguments. This conforms to the model for value based ADTs under SQL3.

As described in section 2.3.5, a SQL Type may also support the concept of persistent references to instances of the Type. To support the latter type of implementation, a reference to a geometry type instance, REF(Geometry), may be used in place of a Geometry value in the SQL functions defined in this section.

3.3 The Well-known Binary Representation for Geometry (WKBGeometry)

3.3.1 Component Overview

The Well-known Binary Representation for Geometry (WKBGeometry), provides a portable representation of a Geometry value as a contiguous stream of bytes. It permits Geometry values to be exchanged between an ODBC client and an SQL database in binary form.

3.3.2 Component Description

The Well-known Binary Representation for Geometry is obtained by serializing a geometry instance as a sequence of numeric types drawn from the set {Unsigned Integer, Double} and then serializing each numeric type as a sequence of bytes using one of two well defined, standard, binary representations for numeric types (NDR, XDR). The specific binary encoding (NDR or XDR) used for a geometry representation is described by a one byte tag that precedes the serialized bytes. The only difference between the two encodings of geometry is one of byte order, the XDR encoding is Big Endian, the NDR encoding is Little Endian.

3.3.2.1 Numeric Type Definitions

An Unsigned Integer is a 32-bit (4-byte) data type that encodes a nonnegative integer in the range [0, 4294967295].

A Double is a 64-bit (8-byte) double precision data type that encodes a double precision number using the IEEE 754 double precision format.

The above definitions are common to both XDR and NDR.

3.3.2.2 XDR (Big Endian) Encoding of Numeric Types

The XDR representation of an Unsigned Integer is Big Endian (most significant byte first).

The XDR representation of a Double is Big Endian (sign bit is first byte).

3.3.2.3 NDR (Little Endian) Encoding of Numeric Types

The NDR representation of an Unsigned Integer is Little Endian (least significant byte first).

The NDR representation of a Double is Little Endian (sign bit is last byte).
3.3.2.4 Conversion between the NDR and XDR representations of WKBGeometry

Conversion between the NDR and XDR data types for Unsigned Integer and Double numbers is a simple operation involving reversing the order of bytes within each Unsigned Integer or Double number in the representation.

3.3.2.5 Relationship to other COM and CORBA data transfer protocols

The XDR representation for Unsigned Integer and Double numbers described above is also the standard representation for Unsigned Integer and for Double number in the CORBA Standard Stream Format for Externalized Object Data that is described as part of the CORBA Externalization Service Specification [15].

The NDR representation for Unsigned Integer and Double number described above is also the standard representation for Unsigned Integer and for Double number in the DCOM protocols that is based on DCE RPC and NDR [16].

3.3.2.6 Description of WKBGeometry Representations

The Well-known Binary Representation for Geometry is described below. The basic building block is the representation for a Point, which consists of two Double numbers. The representations for other geometries are built using the representations for geometries that have already been defined.

```c
// Basic Type definitions
// byte : 1 byte
// uint32 : 32 bit unsigned integer (4 bytes)
// double : double precision number (8 bytes)

// Building Blocks : Point, LinearRing

Point {
    double x;
    double y;
};

LinearRing {
    uint32 numPoints;
    Point points[numPoints];
}

enum wkbGeometryType {
    wkbPoint = 1,
    wkbLineString = 2,
    wkbPolygon = 3,
    wkbMultiPoint = 4,
    wkbMultiLineString = 5,
    wkbMultiPolygon = 6,
    wkbGeometryCollection = 7
};

enum wkbByteOrder {
```
wkbXDR = 0, // Big Endian
wkbNDR = 1 // Little Endian
);

WKBPoint {
    byte byteOrder;
    uint32 wkbType; // 1
    Point point;
}

WKBLinestring {
    byte byteOrder;
    uint32 wkbType; // 2
    uint32 numPoints;
    Point points[numPoints];
}

WKBPolygon {
    byte byteOrder;
    uint32 wkbType; // 3
    uint32 numRings;
    LinearRing rings[numRings];
}

WKBMultiPoint {
    byte byteOrder;
    uint32 wkbType; // 4
    uint32 num_wkbPoints;
    WKBPoint WKBPoints[num_wkbPoints];
}

WKBMultiLineString {
    byte byteOrder;
    uint32 wkbType; // 5
    uint32 num_wkbLineStrings;
    WKBLinestring WKBLinestrings[num_wkbLineStrings];
}

wkbMultiPolygon {
    byte byteOrder;
    uint32 wkbType; // 6
    uint32 num_wkbPolygons;
    WKBPolygon wkbPolygons[num_wkbPolygons];
}

WKBGeometry {
    union {
        WKBPoint point;
        WKBLinestring linestring;
        WKBPolygon polygon;
        WKBGeometryCollection collection;
        WKBMultiPoint mpoint;
        WKBMultiLineString mlinestring;
    }
}
Figure 3.2 shows a pictorial representation of the Well-known Representation for a Polygon with one outer ring and one inner ring.

![Well-known Binary Representation for a Geometry](image)

**Figure 3.2**—Well-known Binary Representation for a Geometry value in NDR format (B=1) of type Polygon (T=3) with 2 linear rings (NR = 2) each ring having 3 points (NP = 3).

### 3.3.2.7 Assertions for Well-known Binary Representation for Geometry

The Well-known Binary Representation for Geometry is designed to represent instances of the geometry types described in the Geometry Object Model and in the OpenGIS Abstract Specification. Any **WKBGeometry** instance must satisfy the assertions for the type of Geometry that it describes. These assertions may be found in the section 2.1.

These assertions imply the following for Rings, Polygons and MultiPolygons:

#### 3.3.2.8 Linear Rings

Rings are simple and closed, which means that Linear Rings may **not** self-touch.
3.3.2.9 Polygons

No two Linear Rings in the boundary of a Polygon may cross each other, the Linear Rings in the boundary of a polygon may intersect at most at a single point but only as a tangent.

3.3.2.10 MultiPolygons

1. The interiors of 2 Polygons that are elements of a MultiPolygon may not intersect.

2. The Boundaries of any 2 Polygons that are elements of a MultiPolygon may touch at only a finite number of points.

For more details on the above assertions and for the assertions for each geometry type the reader is referred to the Geometry Object Model section of this specification.

3.4 Well-known Text Representation of Spatial Reference Systems

3.4.1 Component Overview

The Well-known Text Representation of Spatial Reference Systems provides a standard textual representation for spatial reference system information.

3.4.2 Component Description

The definitions of the well-known text representation are modeled after the POSC/EPSG coordinate system data model.

A spatial reference system, also referred to as a coordinate system, is a geographic (latitude-longitude), a projected (X,Y), or a geocentric (X,Y,Z) coordinate system.

The coordinate system is composed of several objects. Each object has a keyword in upper case (for example, DATUM or UNIT) followed by the defining, comma-delimited, parameters of the object in brackets. Some objects are composed of objects so the result is a nested structure. Implementations are free to substitute standard brackets ( ) for square brackets [ ] and should be prepared to read both forms of brackets.

The EBNF (Extended Backus Naur Form) definition for the string representation of a coordinate system is as follows, using square brackets, see note above:

```
<coordinate_system> = <projected_cs> | <geographic_cs> | <geocentric_cs>
<projected_cs> = PROJCS['<name>', <geographic_cs>, <projection>, {<parameter>,}* <linear_unit>]
<prejection> = PROJECTION['<name>']
<parameter> = PARAMETER['<name>', <value>]
:value> = <number>
```

A data set's coordinate system is identified by the PROJCS keyword if the data are in projected coordinates, by GEOGCS if in geographic coordinates, or by GEOCCS if in geocentric coordinates.

The PROJCS keyword is followed by all of the 'pieces' which define the projected coordinate system. The first piece of any object is always the name. Several objects follow the projected coordinate system name: the geographic coordinate system, the map projection, 0 or more parameters, and the linear unit of measure. All projected coordinate systems are based upon a geographic coordinate system so we will describe the
pieces specific to a projected coordinate system first. As an example, UTM zone 10N on the NAD83 datum is defined as:

```xml
PROJCS['NAD_1983_UTM_Zone_10N',
    <geographic cs>,
    PROJECTION['Transverse_Mercator'],
    PARAMETER['False_Easting',500000.0],
    PARAMETER['False_Northing',0.0],
    PARAMETER['Central_Meridian',-123.0],
    PARAMETER['Scale_Factor',0.9996],
    PARAMETER['Latitude_of_Origin',0.0],
    UNIT['Meter',1.0]]
```

The name and several objects define the geographic coordinate system object in turn: the datum, the prime meridian, and the angular unit of measure.

```xml
<geographic cs> = GEOGCS['<name>', <datum>, <prime meridian>, <angular unit>]
<datum> = DATUM['<name>', <spheroid>]
<spheroid> = SPHEROID['<name>', <semi-major axis>, <inverse flattening>]
<semi-major axis> = <number>  NOTE: semi-major axis is measured in meters and must be > 0.
<inverse flattening> = <number>
<prime meridian> = PRIMEM['<name>', <longitude>]
<longitude> = <number>
```

The geographic coordinate system string for UTM zone 10 on NAD83 is

```xml
GEOGCS['GCS_North_American_1983',
    DATUM['D_North_American_1983',
        SPHEROID['GRS_1980',6378137,298.257222101]],
    PRIMEM['Greenwich',0],
    UNIT['Degree',0.0174532925199433]]
```

The UNIT object can represent angular or linear unit of measures.

```xml
<angular unit> = <unit>
#linear unit> = <unit>
/unit> = UNIT['<name>', <conversion factor>]
<conversion factor> = <number>
```

<conversion factor> specifies number of meters (for a linear unit) or number of radians (for an angular unit) per unit and must be greater than zero.

So the full string representation of UTM Zone 10N is

```xml
PROJCS['NAD_1983_UTM_Zone_10N',
    GEOGCS['GCS_North_American_1983',
        DATUM['D_North_American_1983',
            SPHEROID['GRS_1980',6378137,298.257222101]],
        PRIMEM['Greenwich',0],UNIT['Degree',0.0174532925199433]],
    PROJECTION['Transverse_Mercator'],PARAMETER['False_Easting',500000.0],
    PARAMETER['False_Northing',0.0],PARAMETER['Central_Meridian',-123.0],
    PARAMETER['Scale_Factor',0.9996],PARAMETER['Latitude_of_Origin',0.0],
    UNIT['Meter',1.0]]
```

A geocentric coordinate system is quite similar to a geographic coordinate system. It is represented by

```xml
<geocentric cs> = GEOCCS['<name>', <datum>, <prime meridian>, <linear unit>]
4 Supported Spatial Reference Data

4.1 Supported Linear Units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meter</td>
<td>1.0</td>
</tr>
<tr>
<td>Foot (International)</td>
<td>0.3048</td>
</tr>
<tr>
<td>U.S. Foot</td>
<td>12/39.37</td>
</tr>
<tr>
<td>Modified American Foot</td>
<td>12.0004584/39.37</td>
</tr>
<tr>
<td>Clarke's Foot</td>
<td>12/39.370432</td>
</tr>
<tr>
<td>Indian Foot</td>
<td>12/39.370141</td>
</tr>
<tr>
<td>Link</td>
<td>7.92/39.370432</td>
</tr>
<tr>
<td>Link (Benoit)</td>
<td>7.92/39.370113</td>
</tr>
<tr>
<td>Link (Sears)</td>
<td>7.92/39.370147</td>
</tr>
<tr>
<td>Chain (Benoit)</td>
<td>792/39.370113</td>
</tr>
<tr>
<td>Chain (Sears)</td>
<td>792/39.370147</td>
</tr>
<tr>
<td>Yard (Indian)</td>
<td>36/39.370141</td>
</tr>
<tr>
<td>Yard (Sears)</td>
<td>36/39.370147</td>
</tr>
<tr>
<td>Fathom</td>
<td>1.8288</td>
</tr>
<tr>
<td>Nautical Mile</td>
<td>1852.0</td>
</tr>
</tbody>
</table>

4.2 Supported Angular Units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radian</td>
<td>1.0</td>
</tr>
<tr>
<td>Decimal Degree</td>
<td>π/180</td>
</tr>
<tr>
<td>Decimal Minute</td>
<td>(π/180)/60</td>
</tr>
<tr>
<td>Decimal Second</td>
<td>(π/180)/3600</td>
</tr>
<tr>
<td>Gon</td>
<td>π/200</td>
</tr>
<tr>
<td>Grad</td>
<td>π/200</td>
</tr>
</tbody>
</table>

4.3 Supported Spheroids

<table>
<thead>
<tr>
<th>Name</th>
<th>Semi-major Axis</th>
<th>Inverse Flattening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airy</td>
<td>6377563.396</td>
<td>299.3249646</td>
</tr>
<tr>
<td>Modified Airy</td>
<td>6377340.189</td>
<td>299.3249646</td>
</tr>
<tr>
<td>Australian</td>
<td>6378160</td>
<td>298.25</td>
</tr>
<tr>
<td>Bessel</td>
<td>6377397.155</td>
<td>299.1528128</td>
</tr>
<tr>
<td>Modified Bessel</td>
<td>6377492.018</td>
<td>299.1528128</td>
</tr>
<tr>
<td>Bessel (Namibia)</td>
<td>6377483.865</td>
<td>299.1528128</td>
</tr>
<tr>
<td>Clarke 1866</td>
<td>6378206.4</td>
<td>294.9786982</td>
</tr>
<tr>
<td>Clarke 1866 (Michigan)</td>
<td>6378693.704</td>
<td>294.978684677</td>
</tr>
<tr>
<td>Clarke 1880 (Arc)</td>
<td>6378249.145</td>
<td>293.466307656</td>
</tr>
<tr>
<td>Clarke 1880 (Benoit)</td>
<td>6378300.79</td>
<td>293.466234571</td>
</tr>
<tr>
<td>Clarke 1880 (IGN)</td>
<td>6378249.2</td>
<td>293.46602</td>
</tr>
<tr>
<td>Clarke 1880 (RGS)</td>
<td>6378249.145</td>
<td>293.465</td>
</tr>
<tr>
<td>Clarke 1880 (SGA)</td>
<td>6378249.2</td>
<td>293.46598</td>
</tr>
</tbody>
</table>
4.4 Supported Geodetic Datums

<table>
<thead>
<tr>
<th>Datums</th>
<th>Semimajor axis (m)</th>
<th>Semiminor axis (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everest 1830</td>
<td>6377276.345</td>
<td>300.8017</td>
</tr>
<tr>
<td>Everest 1975</td>
<td>6377301.243</td>
<td>300.8017</td>
</tr>
<tr>
<td>Everest (Sarawak and Sabah)</td>
<td>6377298.556</td>
<td>300.8017</td>
</tr>
<tr>
<td>Modified Everest 1948</td>
<td>6377304.063</td>
<td>300.8017</td>
</tr>
<tr>
<td>GEM10C</td>
<td>6378137</td>
<td>298.257222101</td>
</tr>
<tr>
<td>GRS 1980</td>
<td>6378137</td>
<td>298.257222101</td>
</tr>
<tr>
<td>Helmert 1906</td>
<td>6378200</td>
<td>298.3</td>
</tr>
<tr>
<td>International 1924</td>
<td>6378388</td>
<td>297.0</td>
</tr>
<tr>
<td>Krassovky</td>
<td>6378245</td>
<td>298.3</td>
</tr>
<tr>
<td>NWL9D</td>
<td>6378145</td>
<td>298.25</td>
</tr>
<tr>
<td>OSU_86F</td>
<td>6378136.2</td>
<td>298.25722</td>
</tr>
<tr>
<td>OSU_91A</td>
<td>6378136.3</td>
<td>298.25722</td>
</tr>
<tr>
<td>Plessis 1817</td>
<td>6376523</td>
<td>308.64</td>
</tr>
<tr>
<td>Sphere (radius = 1.0)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sphere (radius = 6371000 m)</td>
<td>6371000</td>
<td>0</td>
</tr>
<tr>
<td>Struve 1860</td>
<td>6378297</td>
<td>294.73</td>
</tr>
<tr>
<td>War Office</td>
<td>6378300.583</td>
<td>296</td>
</tr>
<tr>
<td>WGS 1984</td>
<td>6378137</td>
<td>298.257223563</td>
</tr>
</tbody>
</table>

4.4 Supported Geodetic Datums

Adindan                         | Lisbon                      |
Afgooye                         | Loma Quintana               |
Agadez                          | Lome                        |
Australian Geodetic Datum 1966  | Luzon 1911                  |
Australian Geodetic Datum 1984  | Mahe 1971                   |
Ain el Abd 1970                 | Makassar                    |
Amersfoort                      | Malongo 1987                |
Aratu                           | Manoça                      |
Arc 1950                        | Massawa                     |
Arc 1960                        | Merchich                    |
Ancienne Triangulation Francaise| Militär-Geographische Institute |
Barbados                        | Mhast                       |
Batavia                         | Minna                       |
Beduaram                        | Monte Mario                 |
Beijing 1954                    | M'poraloko                  |
Reseau National Belge 1950      | NAD Michigan                |
Reseau National Belge 1972      | North American Datum 1927   |
Bermuda 1957                    | North American Datum 1983   |
Bern 1898                       | Nahrwan 1967                |
Bern 1938                       | Naparima 1972               |
Bogota                          | Nord de Guerre              |
Bukit Rimpah                    | NGO 1948                    |
Camacupa                        | Nord Sahara 1959            |
Campo Inchauspe                 | NSWC 9Z-2                   |
Cape                            | Nouvelle Triangulation Francaise |
Carthage                        | New Zealand Geodetic Datum 1949 |
Chua                            | OS (SN) 1980                |
Conakry 1905                    | OSGB 1936                   |
Corrego Alegre                  | OSGB 1970 (SN)              |
Côte d'Ivoire                   | Padang 1884                 |
Datum 73                        | Palestine 1923              |
Deir ez Zor                     | Pointe Noire                |
Deutsche Hauptdreiecknetz       | Provisional South American Datum 1956 |
Douala                          | Pulkovo 1942                |
European Datum 1950             | Qatar                       |
European Datum 1987             | Qatar 1948                  |
Egypt 1907                      | Qornoq                       |
European Reference System 1989  | RT38                        |
Fahud                           | South American Datum 1969   |
Gandajika 1970                  | Sapper Hill 1943            |
Garoua                          | Schwarzeck                  |
4.5 Supported Prime Meridians

<table>
<thead>
<tr>
<th>Prime Meridian</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greenwich</td>
<td>0° 0' 0&quot;</td>
</tr>
<tr>
<td>Bern</td>
<td>7° 26' 22.5&quot; E</td>
</tr>
<tr>
<td>Bogota</td>
<td>74° 4' 51.3&quot; W</td>
</tr>
<tr>
<td>Brussels</td>
<td>4° 22' 4.71&quot; E</td>
</tr>
<tr>
<td>Ferro</td>
<td>17° 40' 0&quot; W</td>
</tr>
<tr>
<td>Jakarta</td>
<td>106° 48' 27.79&quot; E</td>
</tr>
<tr>
<td>Lisbon</td>
<td>9° 7' 54.862&quot; W</td>
</tr>
<tr>
<td>Madrid</td>
<td>3° 41' 16.58&quot; W</td>
</tr>
<tr>
<td>Paris</td>
<td>2° 20' 14.025&quot; E</td>
</tr>
<tr>
<td>Rome</td>
<td>12° 27' 8.4&quot; E</td>
</tr>
<tr>
<td>Stockholm</td>
<td>18° 3' 29&quot; E</td>
</tr>
</tbody>
</table>

4.6 Supported Map Projections

<table>
<thead>
<tr>
<th>Cylindrical Projections</th>
<th>Conic Projections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassini</td>
<td>Albers conic equal-area</td>
</tr>
<tr>
<td>Gauss-Kruger</td>
<td>Lambert conformal conic</td>
</tr>
<tr>
<td>Mercator</td>
<td>Azimuthal or Planar Projections</td>
</tr>
<tr>
<td>Oblique Mercator (Hotine)</td>
<td>Polar Stereographic</td>
</tr>
<tr>
<td>Transverse Mercator</td>
<td>Stereographic</td>
</tr>
</tbody>
</table>

4.7 Map Projection Parameters

- **central_meridian**: the line of longitude chosen as the origin of x-coordinates.
- **scale_factor**: used generally to reduce the amount of distortion in a map projection.
- **standard_parallel_1**: a line of latitude that has no distortion generally. Also used for ‘latitude of true scale.’
- **standard_parallel_2**: a line of latitude that has no distortion generally.
- **longitude_of_center**: the longitude which defines the center point of the map projection.
- **latitude_of_center**: the latitude which defines the center point of the map projection.
- **latitude_of_origin**: the latitude chosen as the origin of y-coordinates.
- **false_easting**: added to x-coordinates. Used to give positive values.
- **false_northing**: added to y-coordinates. Used to give positive values.
- **azimuth**: the angle east of north which defines the center line of an oblique projection.
- **longitude_of_point_1**: the longitude of the first point needed for a map projection.
- **latitude_of_point_1**: the latitude of the first point needed for a map projection.
- **longitude_of_point_2**: the longitude of the second point needed for a map projection.
- **latitude_of_point_2**: the latitude of the second point needed for a map projection.
5 References


