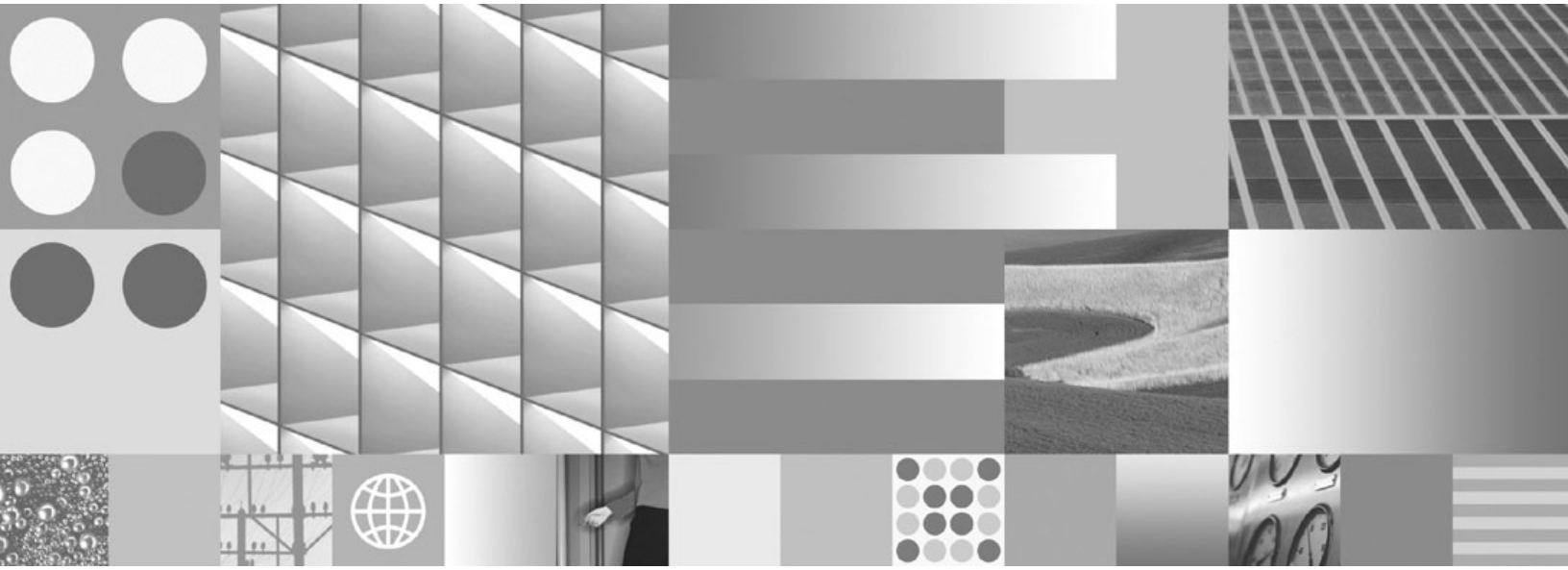
IBM Informix

Version 8.21

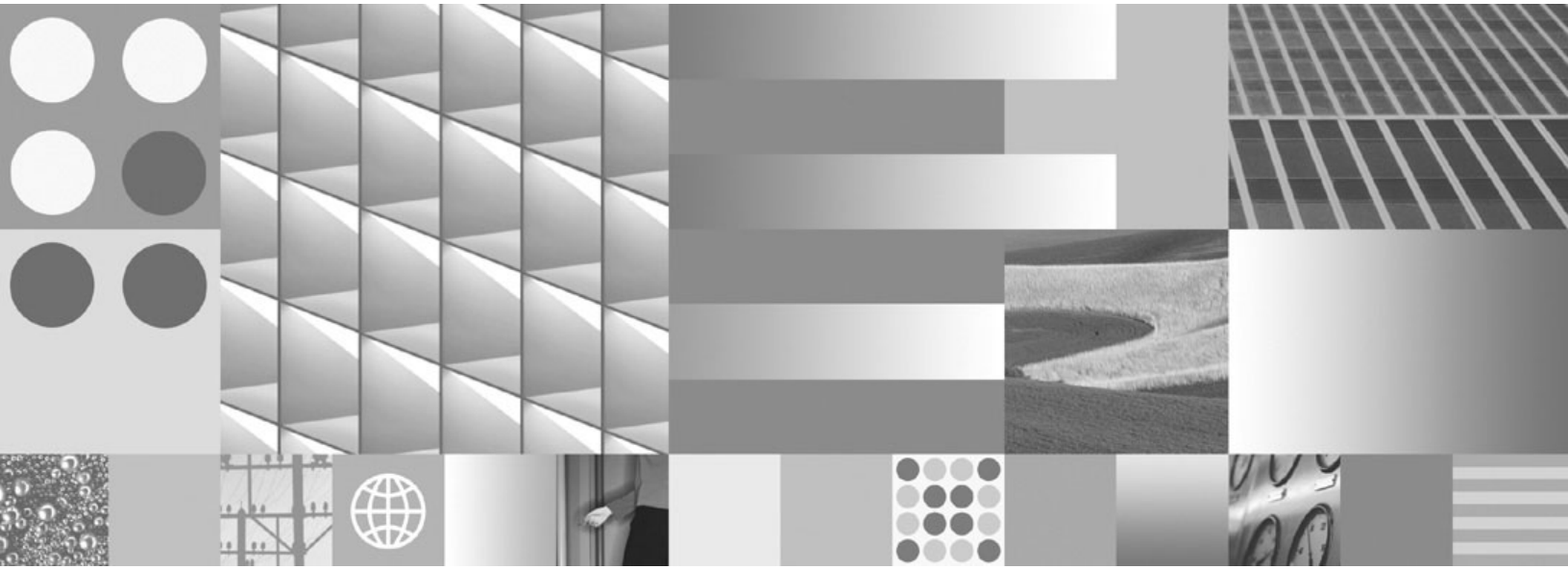


IBM Informix Spatial DataBlade Module User's Guide

G229-6405-03

IBM Informix

Version 8.21



IBM Informix Spatial DataBlade Module User's Guide

G229-6405-03

Note:
Before using this information and the product it supports, read the information in “Notices” on page I-1.

This edition replaces G229-6405-02.

Contents

Introduction

In This Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About This Publication

This publication contains information to assist you in using the IBM Informix Spatial DataBlade Module with IBM Informix Dynamic Server (IDS). A DataBlade® module adds custom data types and supporting routines to the server.

This section discusses the organization of the publication, the intended audience, and the associated software products that you must have to develop and use the IBM Informix Spatial DataBlade Module. General information about the Informix® Spatial DataBlade module is available at http://www.ibm.com/software/data/informix/blades/spatial.

Types of Users

This publication is written for the following audience:

- Database administrators who install the IBM Informix Spatial DataBlade Module
- Developers who write applications to access spatial information stored in IBM Informix Dynamic Server (IDS) databases

Software Dependencies

To use the IBM Informix Spatial DataBlade Module, you must use IBM Informix Dynamic Server (IDS), Version 9.40 or later as your database server. Check the release notes for specific version compatibility.

Assumptions About Your Locale

IBM® Informix products can support many languages, cultures, and code sets. All the information related to character set, collation and representation of numeric data, currency, date, and time is brought together in a single environment, called a GLS (Global Language Support) locale.

The examples in this publication are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

What's New in Spatial DataBlade Module for IDS, Version 8.21

For a comprehensive list of new features for this release, see the *IBM Informix Dynamic Server Getting Started Guide*. The following changes and enhancements are relevant to this publication.

Table 1. What's New in Spatial DataBlade Module 8.21.xC3

Overview	Reference
Enhanced Support for Keyhole Markup Language You can now use the Keyhole Markup Language (KML) as input and output with the Spatial DataBlade module. You can now include shape attributes with KML geometric string expressions and provide KML bounding boxes to your application from the Spatial DataBlade module. You can also send geometric data in KML format to the Spatial DataBlade module.	"KML Representation" on page 3-4

Table 2. What's New in Spatial DataBlade Module 8.21.xC1

Overview	Reference
The function **SE_AsKML()** returns the Keyhole Markup Language (KML) representation of an ST_Geometry.	"KML Representation" on page 3-4 and "SE_AsKML()" on page 8-14

Documentation Conventions

Special conventions are used in the product documentation for IBM Informix Dynamic Server.

Technical Changes

Technical changes to the text are indicated by special characters depending on the format of the documentation.

HTML documentation
New or changed information is surrounded by blue >> and << characters.

PDF documentation
A plus sign (+) is shown to the left of the current changes. A vertical bar (|) is shown to the left of changes made in earlier shipments.

Feature, Product, and Platform Markup

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information.

Some examples of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Windows Only

Identifies information that is specific to the Windows® operating system

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Windows)

Example Code Conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional Documentation

Documentation about IBM Informix products is available in various formats.

You can view, search, and print all of the product documentation from the IBM Informix Dynamic Server information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

For additional documentation about IBM Informix Dynamic Server and related products, including release notes, machine notes, and documentation notes, go to the online product library page at http://www.ibm.com/software/data/informix/techdocs.html. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

Compliance with Industry Standards

IBM Informix products are compliant with various standards.

The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

Syntax Diagrams

Syntax diagrams use special components to describe the syntax for statements and commands.

Table 3. Syntax Diagram Components

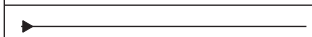
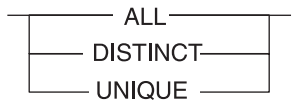
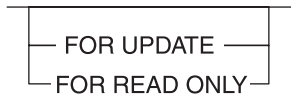
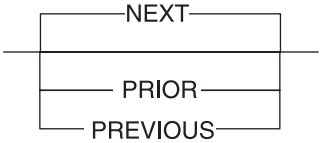
Component represented in PDF	Component represented in HTML	Meaning
	`>>----------`	Statement begins.
	`---------->`	Statement continues on next line.
	`>----------`	Statement continues from previous line.
	`----------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+----------+---` `  '------LOCAL------'`	Optional item.

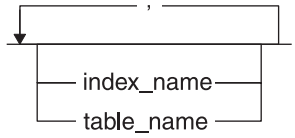
Table 3. Syntax Diagram Components (continued)

Component represented in PDF	Component represented in HTML	Meaning

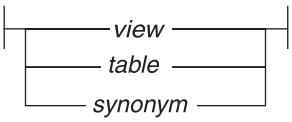
	```---+-----ALL-------+---
	  +--DISTINCT-----+
	  '---UNIQUE------'```
		Required item with choice. One and only one item must be present.

	```---+------------------+---
	  +--FOR UPDATE-----+
	  '--FOR READ ONLY--'```
		Optional items with choice are shown below the main line, one of which you might specify.

	``` .---NEXT---------.
	----+----------------+---
	  +---PRIOR--------+
	  '---PREVIOUS-----'```
		The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.

	``` .-------,-----------.
	  V |
	---+------------------+---
	  +---index_name---+
	  '---table_name---'```
		Optional items. Several items are allowed; a comma must precede each repetition.

	```>>-| Table Reference |-><```	Reference to a syntax segment.

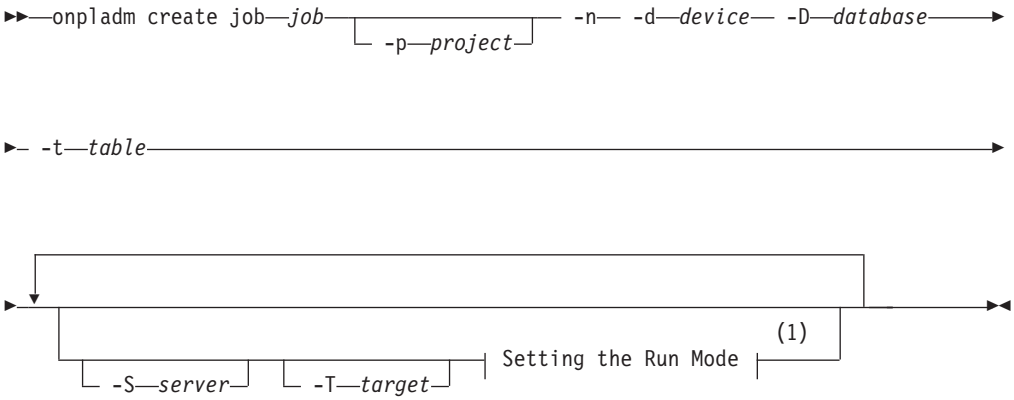
Table Reference
 
	``Table Reference
	|--+-----view--------+--|
	  +------table------+
	  '----synonym------'```
		Syntax segment.

How to Read a Command-Line Syntax Diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a No-Conversion Job

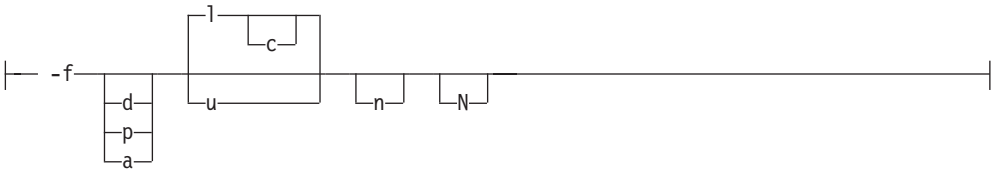


Notes:

1 See page Z-1

This diagram has a segment named "Setting the Run Mode," which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you would find this segment in on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the Run Mode:



To see how to construct a command correctly, start at the top left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and Punctuation

Keywords are words reserved for statements and all commands except system-level commands.

When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

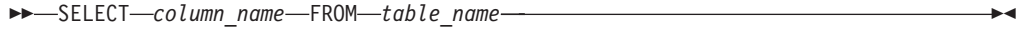
You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:

- Send e-mail to docinf@us.ibm.com.
- Go to the information center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp and open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.
- Add comments to topics directly in the IDS information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more! Find out more at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp?topic=/com.ibm.start.doc/contributing.htm.

Feedback from all methods is monitored by those who maintain the user documentation. The feedback methods are reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support Web site at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. Introducing the IBM Informix Spatial DataBlade Module

In This Chapter

This chapter introduces the IBM Informix Spatial DataBlade Module, describes how to use the **spatial_references** table, and explains how to create a table with a spatial column and insert data into it.

Introducing the IBM Informix Spatial DataBlade Module

The IBM Informix Spatial DataBlade Module embeds a geographic information system (GIS) into your IBM Informix Dynamic Server (IDS) kernel. The IBM Informix Spatial DataBlade Module implements the Open GIS Consortium, Inc. (OpenGIS®, or OGC) SQL3 specification of abstract data types (ADTs). These data types can store spatial data such as the location of a landmark, a street, or a parcel of land. The IBM Informix Spatial DataBlade Module also conforms to the OpenGIS Simple Features Specification for SQL Revision 1.1.

The GIS of the past were spatially centric and focused on gathering spatial data and attaching nonspatial attribute data to it. The IBM Informix Spatial DataBlade Module integrates spatial and nonspatial data, providing a seamless point of access using SQL (Structured Query Language).

The IBM Informix Spatial DataBlade Module provides SQL functions capable of comparing the values in spatial columns to determine if they intersect, overlap, or are related in many different ways. These functions can answer questions like, "Is this school within five miles of a hazardous waste site?" An application programmer can use an SQL query to join a table storing sensitive sites such as schools, playgrounds, and hospitals to another table containing the locations of hazardous sites and return a list of sensitive areas at risk. For example, Figure 1-1 on page 1-2 shows that a school and hospital lie within the five-mile radius of two hazardous site locations and the nursing home lies safely outside both radii.

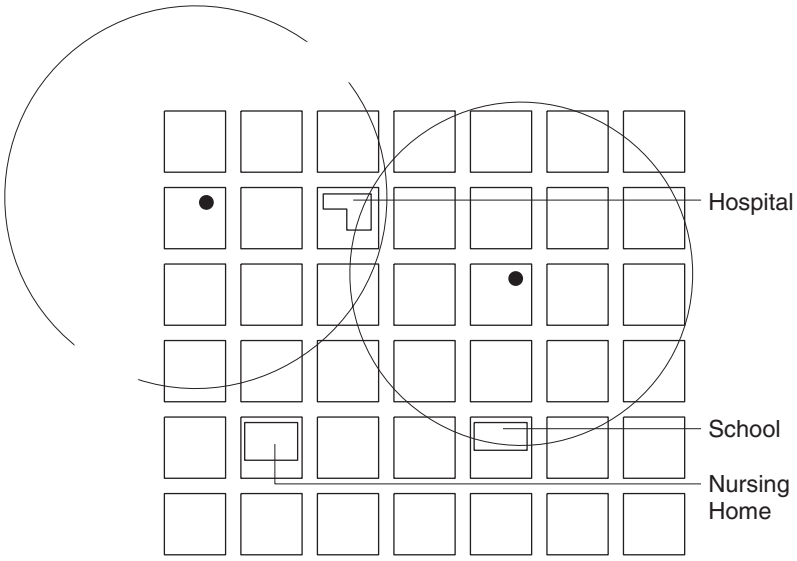


Figure 1-1. Determining Whether Sensitive Sites Are Within Dangerous Areas

With the IBM Informix Spatial DataBlade Module, you use the **ST_Overlaps()** function to evaluate whether the polygon representing the building footprint of the school overlaps the circular polygon that represents the five-mile radius of a hazardous waste site. The building footprints of the school, hospital, and nursing home are stored in the ST_Polygon data type and the location of each hazardous waste site is stored in an ST_Point type.

When to Use the Geodetic or the Spatial DataBlade Module

The IBM Informix Spatial DataBlade module and the IBM Informix Geodetic DataBlade module both manage GIS data in an Informix database. The DataBlade modules use different core technologies that solve different problems and complement each other:

- The IBM Informix Geodetic DataBlade module treats the Earth as a globe. It uses a latitude and longitude coordinate system on an ellipsoidal Earth model. Geometric operations are precise regardless of location.

  The Geodetic DataBlade module is best used for global datasets and applications, such as satellite imagery repositories.
- The IBM Informix Spatial DataBlade Module treats the Earth as a flat map. It uses planimetric (flat-plane) geometry, which means that it approximates the round surface of the Earth by projecting it onto flat planes using various transformations. This leads to some distortion around the edges of the map

  The Spatial DataBlade module is best used for regional datasets and applications, such as those shown in the examples in this publication.

Using the IBM Informix Spatial DataBlade Module

With the IBM Informix Spatial DataBlade Module installed and registered in your database, you can create tables that contain spatial columns. You can insert and store geographic features in the spatial columns.

Figure 1-2 shows the architecture for the IBM Informix Spatial DataBlade Module. The Informix server with the IBM Informix Spatial DataBlade Module can communicate to Java™ applications through the Informix JDBC driver, C applications through the Informix ODBC Driver, directly with ESQL/C applications and DB-Access, and to ESRI application servers through the Informix ODBC Driver. ESRI application servers communicate to license managers and ESRI clients.

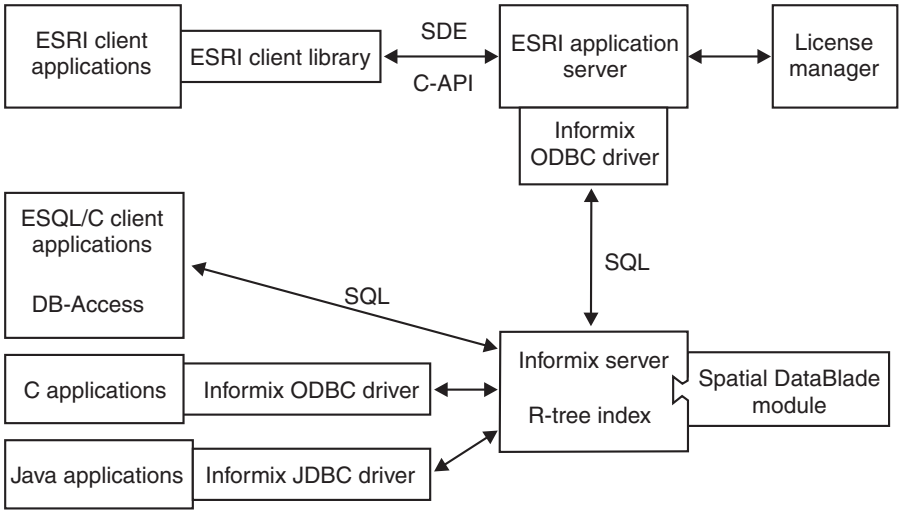


Figure 1-2. Architecture for the IBM Informix Spatial DataBlade Module

To help you use the IBM Informix Spatial DataBlade Module, you can find FAQs, examples, tips, documentation, tuning information and more at the following URL:http://www.ibm.com/software/data/informix/blades/spatial/

Loading Spatial Data into Your Database

For each data type provided by the IBM Informix Spatial DataBlade Module, there is a text file import and a text file export routine. Whenever you execute a load or unload statement in the DB-Access utility, these import and export routines are automatically called (the **dbimport** and **dbexport** utilities also use these routines).

"Inserting Data into a Spatial Column" on page 1-9 provides a detailed description of how to insert data into your IBM Informix database.

Using the ArcSDE Service and GIS Software

The ESRI ArcSDE service complements the IBM Informix Spatial DataBlade Module by providing immediate access to the spatial data stored in your IBM Informix database for the powerful ESRI GIS software programs: ArcView® GIS, MapObjects®, and ArcInfo® software.

The ArcSDE service automatically converts spatial column data into ESRI shape representation, making it available to all ESRI-supported applications and other applications capable of reading this format.

Accessing spatially enabled tables through the ArcSDE service allows you to write applications using the existing tools offered by ESRI GIS software or create applications using the SDE C application programming interface (API). An experienced open database connectivity (ODBC) programmer can also make calls to the IBM Informix Spatial DataBlade Module SQL functions.

Writing Applications and Using Queries

You can use the IBM Informix interactive query tool, DB-Access, to run SQL queries against your spatial data. You can also write applications that access the database:

- **ESQL/C applications**. Use the IBM Informix Client SDK, Version 2.5 or later, to connect to the IBM Informix server. (You may also use IBM Informix Client SDK, Version 2.1.)
- **C applications**. Use the IBM Informix ODBC Driver to connect to the IBM Informix server.
- **Java applications**. Use the IBM Informix JDBC Driver to connect to the IBM Informix server.

Querying the spatial columns directly requires converting the data to one of the three supported external formats. The **ST_AsText()** function converts a spatial column value to the OGC Well-Known Text (WKT) representation. The **ST_AsBinary()** and **SE_AsShape()** functions convert the spatial column values to OGC Well-Known Binary (WKB) and ESRI shape formats. Once converted, applications can display or manipulate the data. Chapter 3, "Data Exchange Formats," on page 3-1,describes the external data formats and how to use them.

Using Indexes

The IBM Informix Spatial DataBlade Module provides the R-tree index to allow indexing of spatial data. R-tree indexes are specifically designed to provide fast, efficient access to spatial data. "Creating a Spatial Index" on page 1-10 introduces R-tree indexes and Chapter 4, "Using R-Tree Indexes," on page 4-1, describes in detail how to use them.

Using the spatial_references Table

In order to use the IBM Informix Spatial DataBlade Module, you must set up appropriate entries for your spatial data in the **spatial_references** table, described next.

The spatial_references Table

Important: Before you register the IBM Informix Spatial DataBlade Module in a database, you must drop any triggers you have added to the **spatial_references** table. See "Using Triggers with the spatial_references Table" on page 1-7 for further information.

The **spatial_references** table stores data about each map projection that you use to store the spherical geometry of the Earth. For example, your data may use the Mercator projection. The **spatial_references** table holds information that enables the IBM Informix Spatial DataBlade Module to translate the data into a flat, X-Y coordinate system.

This mapping between spheroidal and flat data is called a *spatial reference system*. The spatial reference ID (**srid**) is the unique key for the record in the

spatial_references table describing a particular spatial reference system. All spatial reference systems that you use in your database must have a record in the **spatial_references** table. All geometries in a spatial column must use the same spatial reference system. (*Geometry* is the term adopted by the OpenGIS Consortium to refer to two-dimensional spatial data).

Internal functions of the IBM Informix Spatial DataBlade Module use the parameters of a spatial reference system to translate and scale each floating point coordinate of the geometry into 54-bit positive integers prior to storage. When retrieved, the coordinates are restored to their external floating point format.

The Structure of the spatial_references Table

Before you can create geometry and insert it into a table you must enter the SRID of that geometry into the **spatial_references** table. The columns of the **spatial_references** table are described in the following table.

Column Name	Type	Example Value	Description
srid	INTEGER NOT NULL	`12345`	Primary key: the unique key for the record describing a particular spatial reference system
description	VARCHAR(64)	`WGS 1984`	A text description of the spatial reference system
auth_name	VARCHAR(255)	`EPSG`	The name of the standard or standards body cited for the reference system
auth_srid	INTEGER	`4326`	The ID of the spatial reference system as defined by the authority cited in **auth_name**
falsex	FLOAT NOT NULL	`-180`	The external floating point X coordinates are converted to integers for storage in the database by subtracting the **falsex** values.
falsey	FLOAT NOT NULL	`-90`	The external floating point Y coordinates are converted to integers for storage in the database by subtracting the **falsey** values.
xyunits	FLOAT NOT NULL	`1000000`	Before being inserted into the database, the external floating point X and Y coordinates are scaled by the value in **xyunits**, adding a half unit, and truncating the remainder.
falsez	FLOAT NOT NULL	`-1000`	The external floating point Z coordinates are converted to integers for storage in the database by subtracting the **falsez** values.
zunits	FLOAT NOT NULL	`1000`	A factor used to scale the Z-coordinate
falsem	FLOAT NOT NULL	`-1000`	The external floating point M coordinates are converted to integers for storage in the database by subtracting the **falsem** values.
munits	FLOAT NOT NULL	`1000`	A factor used to scale the measure values

Column Name	Type	Example Value	Description
srtext	CHAR(2048)	`GEOGCS["GCS_WGS_1984", DATUM["D_WGS_1984", SPHEROID["WGS_1984", 6378137, 298.257223563]], PRIMEM["Greenwich",0], UNIT["Degree", 0.0174532925199433]]`	The **srtext** column contains the WKT representation of the spatial reference system. Appendix B, "OGC Well-Known Text Representation of Spatial Reference Systems," on page B-1, contains a complete description of this format.

The following example shows how to insert a spatial reference system into the **spatial_references** table:

```
INSERT INTO spatial_references
     (srid, description, auth_name, auth_srid, falsex, falsey,
      xyunits, falsez, zunits, falsem, munits, srtext)
   VALUES (1, NULL, NULL, NULL, 0, 0, 100, 0, 1, 0, 1, 'UNKNOWN');
```

In this example, the spatial reference system has an SRID value of 1, a false X, Y of (0,0), and its system units are 100. The Z coordinate and measure offsets are 0, while the Z coordinate and measure units are 1.

The next section describes how to choose appropriate values for the **falsex**, **falsey**, **falsez**, **falsem**, **xyunits**, **zunits**, and **munits** columns.

Selecting a False Origin and System Units

Select a false origin and system unit to store all of your coordinate values at an acceptable scale. To do so, you must know the range of your data and the scale to maintain. Because coordinates are stored as positive 54-bit integers, the maximum range of values allowed is between 0 and 9,007,199,254,740,991, but the actual range is dependent on the false origin and system units of the spatial reference system.

A negative false origin shifts the range of values in the negative direction. A positive false origin shifts the range of values in the positive direction. For example, a false origin of –1000, with a system unit of one, stores a range of values between –1000 to 2^{53} –1000.

The system unit scales the data and cannot be less than one. The larger the system unit the greater the scale that can be stored, but the smaller the range of values. For example, given a system unit of 1000 and a false origin of zero, data with three digits to the right of the decimal point are supported; the range of possible values is reduced to 0.001 to 2^{50}.

If you want to maintain a scale of three digits to the right of the decimal point, set your system units to 1,000. Set the false origin to less than the minimum coordinate value in your data set. The false origin must be small enough to account for any buffering of the data. If the minimum coordinate value is –10,000 and your application includes functions that buffer the data by 5,000, the false origin must be at least –15,000. Finally, make sure that the maximum ordinate value is not greater than 2^{53} by applying the following formula to the maximum value. This formula converts floating point coordinates into system units:

```
stored value = truncate(((ordinate - false origin) * system unit) + 0.5)
```

"SE_CreateSRID()" on page 8-31 describes the **SE_CreateSRID()** utility function that, given the X and Y extents of a spatial data set, computes the false origin and system units.

Important: Do not change a spatial reference system's false origin and system units once you have inserted spatial data into a table using that SRID. The IBM Informix Spatial DataBlade Module uses these parameters to translate and scale your data prior to storage. If you change these values, you will be unable to retrieve your original floating point coordinate data.

Using Triggers with the spatial_references Table

If you have added your own triggers to the **spatial_references** table, you must drop them before you register the IBM Informix Spatial DataBlade Module in a database. After registering the DataBlade module using BladeManager, follow these steps:

1. Edit the script, **combine_triggers.sql**, in the **$INFORMIXDIR/extend/spatial.version** directory to combine your triggers with those added by the Spatial DataBlade module.
2. Run the **combine_triggers.sql** script using the DB-Access utility.

See the *IBM Informix DataBlade Module Installation and Registration Guide* for information about registering DataBlade modules.

Working with Spatial Tables

This section describes how to create tables with spatial columns, how to insert and update spatial data, and how to create indexes on spatial columns.

Creating a Spatial Table

A spatial table is a table that includes one or more spatial columns. To create a spatial table, include a spatial column in the column clause of the CREATE TABLE statement. A spatial column can only accept data of the type required by the spatial column. For example, a column of ST_Polygon type rejects integers, characters, and even other types of geometry. In addition to the geometry column, ESRI client software also requires a unique key, the **se_row_id** INTEGER column.

Consider the sensitive areas and hazardous waste sites example. Stored in the **sensitive_areas** table are schools, hospitals, and playgrounds, while the **hazardous_sites** table holds locations of hazardous waste sites. The ST_Polygon data type is used to store the sensitive areas, while hazardous sites are stored as points using the ST_Point type:

```
CREATE TABLE sensitive_areas (se_row_id integer NOT NULL,
                              area_id   integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      ST_Polygon);

CREATE TABLE hazardous_sites (se_row_id integer NOT NULL,
                              site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

Establishing a Spatial Reference System

Before you populate a spatial table, you must define a spatial reference system for all of the geometry data that will be loaded into a spatial column. Spatial reference system information is stored in the **spatial_references** table, described in "The

spatial_references Table" on page 1-4. The SRID of a geometry's spatial reference system is stored with the geometry itself, so it is important that you choose an SRID carefully.

If the coordinates of your data are in latitude and longitude, you may be able to use one of several predefined SRIDs that are inserted into the **spatial_references** table when the IBM Informix Spatial DataBlade Module is registered in a database. If the coordinates are in a different system (for example, UTM) you must define a new spatial reference system.

Tip: If your spatial data is supplied in shapefiles, you can use the **infoshp** utility to help you define a spatial reference system. The **SE_CreateSRID()** and **SE_CreateSrtext()** SQL functions (described in Chapter 8, "SQL Functions," on page 8-1) may also be helpful.

For our hazardous sites and sensitive areas example, the coordinates are in a local countywide XY coordinate system, so we need to create a new spatial reference system. The X and Y coordinates range from 0 to 250000. We can use the **SE_CreateSRID()** function to do the math for us:

```
EXECUTE FUNCTION SE_CreateSRID(0, 0, 250000, 250000,
                               "Springfield county XY coord system");

(expression)

          5
```

The return value of the function is the SRID, which we can then use when we insert data.

Using the geometry_columns Table

Whenever you create a table with a spatial column, or add a spatial column to an existing table, ESRI client software also requires you to add an entry to the **geometry_columns** table. This is a metadata table, like **spatial_references**, which stores data about data. For our hazardous sites and sensitive areas example, the INSERT statements look like this:

```
INSERT INTO geometry_columns
          (f_table_catalog, f_table_schema, f_table_name,
           f_geometry_column, geometry_type, srid)
   VALUES ("mydatabase",         -- database name
           "ralph",              -- user name
           "sensitive_areas",    -- table name
           "zone",               -- spatial column name
            5,                   -- column type (5 = polygon)
            5);                  -- srid

INSERT INTO geometry_columns
          (f_table_catalog, f_table_schema, f_table_name,
           f_geometry_column, geometry_type, srid)
   VALUES ("mydatabase",         -- database name
           "ralph",              -- user name
           "hazardous_sites",    -- table name
           "location",           -- spatial column name
            1,                   -- column type (1 = point)
            5);                  -- srid
```

Appendix F, "Values for the geometry_type Column," on page F-1, shows valid entries for the **geometry_type** column and the shapes they represent.

Tip: The **loadshp** utility, described in Appendix A, "Loading and Unloading Shapefile Data," on page A-1, automatically creates these entries in the **geometry_columns** table. The ESRI **shp2sde** command also does this.

Inserting Data into a Spatial Column

You can copy data into or out of the database using the traditional external geometry formats or latitude and longitude based geometry formats:

- IBM Informix load format
- OGC well-known text representation (WKT)
- OGC well-known binary representation (WKB)
- ESRI shapefile format
- Geography Markup Language (GML)
- Keyhole Markup Language (KML)

Except for the IBM Informix load format, all of these formats require the use of input and output conversion functions to insert spatial data into, and retrieve data from, a database. The IBM Informix Spatial DataBlade Module has functions to convert data into its stored data types for each of these formats. Chapter 3, "Data Exchange Formats," on page 3-1 describes how the input and output conversion functions of the external geometry formats operate.

For example, in the SQL statements below, records are inserted into the **sensitive_areas** and **hazardous_sites** table. The **ST_PolyFromText()** function converts the well-known text representation of a polygon into an ST_Polygon type before inserting it into the **ZONE** column of the **sensitive_areas** table. Likewise, the **ST_PointFromText()** function converts the well-known text representation of a point into an ST_Point type before inserting it into the **LOCATION** column of the **hazardous_sites** table. You can also enter data in ESRI shapefile format using the **SE_PolyFromShape()** and **ST_PointFromShape()** functions, or in well-known binary format using the **ST_PolyFromWKB()** and **ST_PointFromWKB()** functions.

```
INSERT INTO sensitive_areas VALUES (
    1, 408, 'Summerhill Elementary School', 67920.64, 'school',
    ST_PolyFromText('polygon ((52000 28000,58000 28000,58000 23000,52000 23000,52000 28000))',5)
);

INSERT INTO hazardous_sites VALUES (
    1, 102, 'W. H. Kleenare Chemical Repository',
    ST_PointFromText('point (17000 57000)',5)
);
```

Alternatively, you can use the IBM Informix load format, in which case a conversion function is not required:

```
INSERT INTO sensitive_areas VALUES (
    2, 129, 'Johnson County Hospital', 102281.91, 'hospital',
    '5 polygon ((32000 55000,32000 68000,38000 68000,38000 52000,35000 52000,35000 55000,32000 55000))'
);

INSERT INTO hazardous_sites VALUES (
    2, 59, 'Landmark Industrial',
    '5 point (58000 49000)'
);
```

In the simple example above, a few records are inserted into database tables. However, the actual amount of data loaded into a GIS system usually ranges between ten thousand records for smaller systems and one hundred million records for larger systems. The design of the IBM Informix Spatial DataBlade Module can handle the entire range.

One option for loading data is to use the **loadshp** utility described in Appendix A, "Loading and Unloading Shapefile Data," on page A-1. This utility allows you to copy data from ESRI shapefiles to an IBM Informix database. A companion utility, **unloadshp**, allows you to unload data from a database to shapefiles.

Important: The **loadshp** utility does not add information to ESRI system tables, such as the **layers** table. Therefore, data loaded with the **loadshp** utility is not accessible to ArcSDE and other ESRI client tools. Use the ESRI **shp2sde** command to load data if you want to access it using ESRI client tools. Data loaded using the **loadshp** utility is accessible to client programs that do not depend on ESRI system tables other than the OGC-standard **geometry_columns** and **spatial_references** tables.

Another option is to develop your own loader application. The source code for two sample programs, **load_wkb** and **load_shapes**, is supplied with the IBM Informix Spatial DataBlade Module. These programs illustrate how to convert data into OGC well-known binary and ESRI shapefile formats. The programs can be modified and linked into existing applications. They are located under **$INFORMIXDIR/extend/spatial.version/examples**. ESQL/C and ODBC versions of both programs are provided.

Alternatively, you can acquire data from a vendor and load the data through the ESRI SDE server. The **shp2sde** command loads shapefiles into an existing table. Accessing IBM Informix Spatial DataBlade Module tables through SDE software also provides immediate access to SDE client software such as ArcView GIS, MapObjects, ARC/INFO and ArcExplorer. MicroStation and AutoCAD are also accessible through SDE CAD client software.

Creating a Spatial Index

Since spatial columns contain multidimensional geographic data, applications querying those columns require an index strategy that quickly identifies all geometries that satisfy a specified spatial relationship. For this reason the IBM Informix Spatial DataBlade Module provides support for building a spatial index, called the R-tree index.

The R-tree Index

The two-dimensional R-tree index differs from the traditional hierarchical (one-dimensional) B-tree index. Spatial data is two-dimensional, so you cannot use the B-tree index for spatial data. Similarly, you cannot use an R-tree index with nonspatial data.

Chapter 4, "Using R-Tree Indexes," on page 4-1, describes how to create and use R-tree indexes.

To create an R-tree index on the **location** column of the **hazardous_sites** table in our example, use the CREATE INDEX statement:

```
CREATE INDEX location_ix
   ON hazardous_sites (location ST_Geometry_ops)
   USING RTREE;
```

Tip: The **loadshp** utility, described in Appendix A, "Loading and Unloading Shapefile Data," on page A-1, automatically creates an R-tree index for you after loading data. The ESRI **shp2sde** command can also be used to create an R-tree index.

Restriction: You cannot rename a database if the database contains a table that has an R-tree index defined on it, because R-tree indexes are implemented with secondary access method. Databases that use primary access method (also called virtual table interface) or secondary access method (also called virtual index interface) cannot be renamed.

Updating Statistics

IBM Informix Dynamic Server's query optimizer will not use the R-tree index unless the statistics on the table are up-to-date. If the R-tree index is created after the data has been loaded, the statistics are current and the optimizer will use the index. However, if the index is created before the data is loaded, the optimizer will not use the R-tree index, because the statistics are out of date. To update statistics, use the UPDATE STATISTICS SQL statement:

```
UPDATE STATISTICS FOR TABLE hazardous_sites;
```

Updating Values in a Spatial Column

The SQL UPDATE statement alters the values in a spatial column just as it does any other type of column. Typically, spatial column data must be retrieved from the table, altered in a client application, and then returned to the server. The following pair of SQL statements illustrates how to fetch the spatial data from one row in the **hazardous_sites** table and then update the same item:

```
SELECT ST_AsText(location) FROM hazardous_sites
   WHERE site_id = 102;

UPDATE hazardous_sites
   SET location = ST_PointFromText('point(18000 57000)', 5)
   WHERE site_id = 102;
```

Performing Spatial Queries

A common task in a GIS application is to retrieve the visible subset of spatial data for display in a window. The easiest way to do this is to define a polygon representing the boundary of the window and then use the **SE_EnvelopesIntersect()** function to find all spatial objects that overlap this window:

```
SELECT name, type, zone FROM sensitive_areas
   WHERE SE_EnvelopesIntersect(zone,
      ST_PolyFromText('polygon((20000 20000,60000 20000,60000 60000,20000 60000,20000 20000))', 5));
```

Queries can also use spatial columns in the SQL WHERE clause to qualify the result set; the spatial column need not be in the result set at all. For example, the following SQL statement retrieves each sensitive area with its nearby hazardous waste site if the sensitive area is within five miles of a hazardous site. The **ST_Buffer()** function generates a circular polygon representing the five-mile radius around each hazardous location. The ST_Polygon geometry returned by the **ST_Buffer()** function becomes the argument of the **ST_Overlaps()** function, which returns `t` (TRUE) if the **zone** ST_Polygon of the **sensitive_areas** table overlaps the ST_Polygon generated by the **ST_Buffer()** function:

```
SELECT sa.name sensitive_area, hs.name hazardous_site
   FROM sensitive_areas sa, hazardous_sites hs
   WHERE ST_Overlaps(sa.zone, ST_Buffer(hs.location, 26400));


sensitive_area  Summerhill Elementary School
hazardous_site  Landmark Industrial

sensitive_area  Johnson County Hospital
hazardous_site  Landmark Industrial
```

Chapter 5, "Running Parallel Queries," on page 5-1, describes how to run queries using parallel UDRs.

Optimizing Spatial Queries

Two environment variables are available for altering the default behavior of the query optimizer:

- ST_MAXLEVELS can solve some UPDATE STATISTICS failures if your system runs low on shared memory.
- ST_COSTMULTIPLIER lets you adjust the cost multiplier to more accurately reflect your system.

Running UPDATE STATISTICS HIGH on a very large table might require large amounts of shared memory (tens of megabytes). If sufficient shared memory is not available, UPDATE STATISTICS will fail. If it fails, you can use the ST_MAXLEVELS environment variable to reduce the memory requirements. The default value is 16. Decrease it to reduce the amount of memory that is needed to build a histogram. Decreasing the value of ST_MAXLEVELS might result in a less accurate histogram than would otherwise be possible. The minimum recommended value is 12.

Set the ST_COSTMULTIPLER environment variable to a non-zero positive value to adjust the cost for each row that is computed by the Spatial DataBlade module. The DataBlade module multiplies its cost estimate by this value.

If you set either of these environment variables, you must stop and restart the database server for the change to take effect.

Important: IBM reserves the right to drop support for any of these environment variables in future releases.

Spatial Memory Reuse

To improve performance, the Spatial DataBlade module manages its own pool of temporary memory buffers for processing spatial data. You can change the behavior of this memory management system, if necessary, in two ways:

- By setting the ST_MEMMODE environment variable as described in "ST_MEMMODE Environment Variable."
- By changing the value of the MemMode user-settable parameter as described in "MemMode Parameter" on page 1-13.

ST_MEMMODE Environment Variable

This environment variable can have three values:

- **0** Disables memory buffer reuse. All Spatial DataBlade module memory allocation requests are sent directly to the server. Temporary buffers, used for processing spatial data, are allocated from the per_routine memory pool and are not reused between UDR invocations. Several memory buffers are typically allocated and freed for every row in a table that is being processed.
- **1** Enables memory buffer reuse. This is the default value. Temporary buffers are allocated from the per_command memory pool. As they are freed, they are returned to a pool that is maintained by the Spatial DataBlade module and are reused for subsequent memory requests. This pool is drained when the UDR sequence completes, which is after all rows in a table are processed.
- **2** Disables memory buffer reuse, but allocate all temporary buffers from the server per_command memory pool. This mode, in conjunction with the

DONTDRAINPOOLS server environment variable, has an effect similar to mode=1, but lets the server manage the memory.

Buffers that hold UDR return values are allocated from the per_command memory pool and are reused between UDR invocations. The ST_MEMMODE environment variable must be set before the server starts. If you change the value of the environment variable, you must restart the server. To override an existing value without restarting the server, you can also change the value of a parameter called MemMode as described in "MemMode Parameter."

MemMode Parameter

MemMode is a parameter that takes the same values as the ST_MEMMODE environment variable. Unlike the environment variable, you can set the value of MemMode at any time using the **SE_ParamSet()** function as described in "SE_ParamSet() function" on page 8-117. The value of MemMode takes precedence over the value of the ST_MEMMODE environment variable. MemMode remains set until the server is shut down. When the server is restarted, it sets the value to that of ST_MEMMODE.

Use the following functions to view and change the value of the MemMode parameter:

- "SE_ParamGet() function" on page 8-116
- "SE_ParamSet() function" on page 8-117

Data Replication

You can use spatial data types with Enterprise Replication and high-availability clusters. The necessary support functions are automatically created in your database when you register the Spatial DataBlade module.

The following conditions must be met to replicate spatial data:

- You must ensure that all copies of the **spatial_references** table are synchronized at all times.
- Spatial data type columns in tables that you include in your data replication system must be nullable.

For information on Enterprise Replication, see the *IBM Informix Dynamic Server Enterprise Replication Guide*. For information on high-availability clusters, see the *IBM Informix Administrator's Guide*.

The Web Feature Service DataBlade Module

The Web Feature Service (WFS) DataBlade Module is a transaction service that supports the following operations:

- Creating new feature instances
- Deleting feature instances
- Updating feature instances
- Querying features based on spatial and non-spatial constraints

The WFS DataBlade Module includes a CGI client program and a Dynamic Server server-side function to let web programs send requests to Dynamic Server for geographical features. These geographical features are encoded in the platform-independent, XML-based geography markup language (GML). You can use the Spatial DataBlade or Geodetic DataBlade modules to enable Dynamic

Server to manage geographical features. The WFS DataBlade Module is written to the Open Geospatial Consortium (OGC) standard in document 04-094, Web Feature Service Implementation Specification Version 1.1.0. For more information, see the *IBM Informix Database Extensions User's Guide* and the OGC web site at http://www.opengeospatial.org.

Chapter 2. Using Spatial Data Types

In This Chapter

This chapter describes the data types that the IBM Informix Spatial DataBlade Module provides to store spatial data and also describes how to use them.

The functions referred to in this chapter are documented in detail in Chapter 8, "SQL Functions," on page 8-1.

Properties of the Spatial Data Types

The OGC, in its publication of *OpenGIS Features for ODBC (SQL) Implementation Specification*, selected the term *geometry* to represent spatial features such as point locations and polygons. Typically, points represent an object at a single location, linestrings represent a linear characteristic, and polygons represent a spatial extent. An abstract definition of the OpenGIS noun *geometry* might be, "a point or aggregate of points symbolizing a feature on the ground."

The ST_Geometry data type provided by the IBM Informix Spatial DataBlade Module is an abstract noninstantiable superclass. Its subclasses provide instantiable data types. This means you can define table columns to be of such types. Figure 2-1 on page 2-2 shows the class hierarchy for the IBM Informix Spatial

DataBlade Module data types.

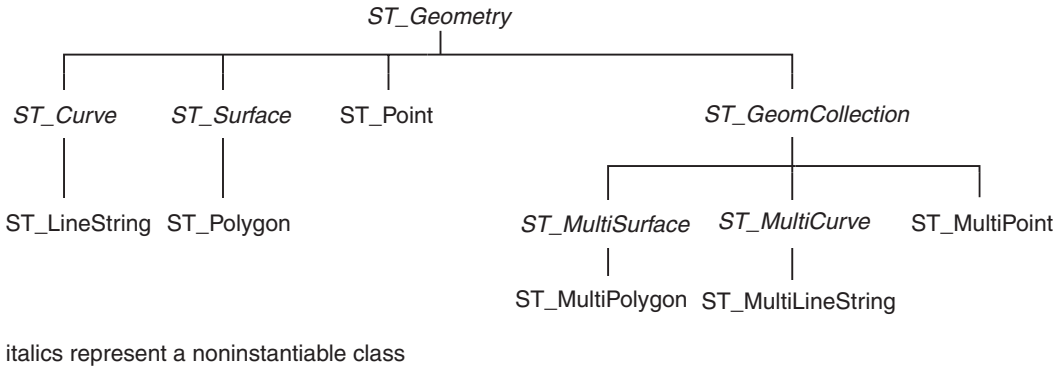


Figure 2-1. Data Types Class Diagram

Throughout the remainder of this publication, the terms *geometry* or *geometries* collectively refer to the superclass ST_Geometry data type and all of its subclass data types. Whenever it is necessary to specify the geometry superclass directly, it will be referred to as the ST_Geometry superclass or the ST_Geometry data type.

Tip: You can define a column as type ST_Geometry, but ST_Geometry values cannot be inserted into it since they cannot be instantiated. However, any of the ST_Geometry subclass data type values can be inserted into this column.

The rest of this section describes the properties of the spatial data types. Each subclass data type inherits the properties of the ST_Geometry superclass and adds properties of its own. Functions that operate on the ST_Geometry data type also operate on any of the subclass data types. However, functions that are defined at the subclass level only operate on that data type and its subclasses data types.

"Spatial Data Types" on page 2-4 describes each of the subclass data types.

Interior, Boundary, and Exterior

All geometries occupy a position in space defined by their interior, boundary, and exterior.

- The exterior of a geometry is all space not occupied by the geometry.
- The boundary of a geometry serves as the interface between its interior and exterior.
- The interior is the space occupied by the geometry.

The subclass inherits the interior and exterior properties from **ST_Geometry**; the boundary property differs for each data type.

The **ST_Boundary()** function takes an ST_Geometry type and returns an ST_Geometry that represents the source ST_Geometry boundary.

Simple or Nonsimple

Some subclasses of ST_Geometry (ST_LineStrings, ST_MultiPoints, and ST_MultiLineStrings) are either simple or nonsimple. They are simple if they obey all topological rules imposed on the subclass and nonsimple if they bend a few rules. ST_LineString is simple if it does not intersect its interior. ST_MultiPoint is simple if none of its elements occupy the same coordinate space. ST_MultiLineString is simple if none of its element's interiors intersect.

The **ST_IsSimple()** function takes an ST_Geometry and returns `t` (TRUE) if the ST_Geometry is simple and `f` (FALSE), otherwise.

Empty or Not Empty

A geometry is empty if it does not have any points. An empty geometry has a NULL envelope, boundary, interior, and exterior. An empty geometry is always simple and can have Z coordinates or measures. Empty linestrings and multilinestrings have a 0 length. Empty polygons and multipolygons have 0 area.

The **ST_IsEmpty()** function takes an ST_Geometry and returns `t` (TRUE) if the ST_Geometry is empty and `f` (FALSE) otherwise.

Number of Points

A geometry can have zero or more points. A geometry is considered empty if it has zero points. The ST_Point subclass is the only geometry that is restricted to zero or one point; all other subclasses can have zero or more.

Envelope

The envelope of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates. The envelopes of most geometries form a boundary rectangle; however, the envelope of a point is the point itself, since its minimum and maximum coordinates are the same, and the envelope of a horizontal or vertical linestring is a linestring represented by the endpoints of the source linestring.

The **ST_Envelope()** function takes an ST_Geometry and returns a ST_Geometry that represents the source ST_Geometry envelope.

Dimension

A geometry can have a dimension of 0, 1, or 2.

The dimensions are:

- **0.** The geometry has neither length or area.
- **1.** The geometry has a length.
- **2.** The geometry contains area.

The point and multipoint subclasses have a dimension of 0. Points represent zero-dimensional features that can be modeled with a single coordinate, while multipoints represent data that must be modeled as a cluster of unconnected coordinates.

The subclasses linestring and multilinestring have a dimension of 1. They store road segments, branching river systems, and any other features that are linear in nature.

Polygon and multipolygon subclasses have a dimension of 2. Forest stands, parcels, water bodies, and other features whose perimeter encloses a definable area can be rendered by either the polygon or multipolygon data type.

Dimension is important not only as a property of the subclass, but also plays a part in determining the spatial relationship of two features. The dimension of the

resulting feature or features determines whether or not the operation was successful. The dimension of the features is examined to determine how they should be compared.

The **ST_Dimension()** function takes an ST_Geometry and returns its dimension as an integer.

Z Coordinates

Some geometries have an associated altitude or depth. Each of the points that form the geometry of a feature can include an optional Z coordinate that represents an altitude or depth normal to the earth's surface.

The **SE_Is3D()** predicate function takes an ST_Geometry and returns `t` (TRUE) if the function has Z coordinates and `f` (FALSE), otherwise.

Measures

Measures are values assigned to each coordinate. The value represents anything that can be stored as a double-precision number.

The **SE_IsMeasured()** function takes an ST_Geometry and returns `t` (TRUE) if it contains measures and `f` (FALSE), otherwise.

Spatial Reference System

The spatial reference system identifies the coordinate transformation matrix for each geometry. All spatial reference systems known to the database are stored in the **spatial_references** table. For information about entering records and maintaining the **spatial_references** table, refer to "The spatial_references Table" on page 1-4.

The **ST_SRID()** function takes an ST_Geometry and returns its spatial reference identifier as an integer.

Spatial Data Types

The data types are divided into two categories: the base geometry subclasses and the homogeneous collection subclasses.

- The base geometries are:
  - ST_Point
  - ST_LineString
  - ST_Polygon
- The homogeneous collections are:
  - ST_MultiPoint
  - ST_MultiLineString
  - ST_MultiPolygon

Homogeneous collections are collections of base geometries; in addition to sharing base geometry properties, homogeneous collections also have some properties of their own.

The **ST_GeometryType()** function takes an ST_Geometry and returns the instantiable subclass in the form of a character string. The **ST_NumGeometries()** function takes a homogeneous collection and returns the number of base geometry

elements it contains. The **ST_GeometryN()** function takes a homogeneous collection and an index and returns the *n*th base geometry.

ST_Point

ST_Point is a zero-dimensional geometry that occupies a single location in coordinate space. ST_Point is used to define features such as oil wells, landmarks, and elevations.

Properties of ST_Point

An ST_Point has a single X,Y coordinate value, is always simple, and has a NULL boundary. An **ST_Point** may include a Z coordinate and an M value.

Functions That Operate on ST_Point

Functions that operate solely on the ST_Point data type include **ST_X()**, **ST_Y()**, **SE_Z()**, and **SE_M()**.

The **ST_X()** function returns a point data type's X coordinate value as a double-precision number.

The **ST_Y()** function returns a point data type's Y coordinate value as a double-precision number.

The **SE_Z()** function returns a point data type's Z coordinate value as a double-precision number.

The **SE_M()** function returns a point data type's M coordinate value as a double-precision number.

ST_LineString

ST_LineString is a one-dimensional object stored as a sequence of points defining a linear interpolated path. ST_LineString types are often used to define linear features such as roads, rivers, and power lines.

Properties of ST_LineString

An ST_LineString is simple if it does not intersect its interior. The endpoints (the boundary) of a closed ST_LineString occupy the same point in space. An ST_LineString is a ring if it is both closed and simple. In addition to properties inherited from the superclass ST_Geometry, ST_LineString values have length.

The endpoints normally form the boundary of a ST_LineString unless the ST_LineString is closed, in which case the boundary is NULL. The interior of a ST_LineString is the connected path that lies between the endpoints, unless it is closed, in which case the interior is continuous. Figure 2-2 shows examples of ST_LineString objects: (1) is a simple nonclosed ST_LineString; (2) is a nonsimple nonclosed ST_LineString; (3) is a closed simple ST_LineString and therefore is a ring; (4) is a closed nonsimple ST_LineString—it is not a ring.

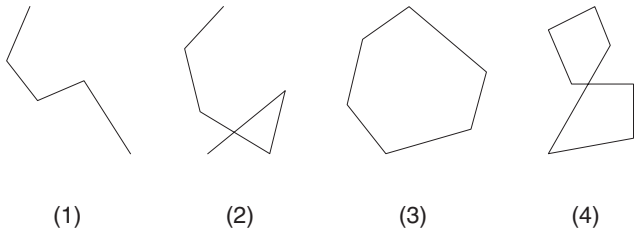


Figure 2-2. Examples of ST_LineString Objects

Functions That Operate on ST_LineString

Functions that operate on ST_LineString include **ST_StartPoint()**, **ST_EndPoint()**, **ST_PointN()**, **ST_Length()**, **ST_NumPoints()**, **ST_IsRing()**, and **ST_IsClosed()**.

The **ST_StartPoint()** function returns the linestring's first point.

The **ST_EndPoint()** function returns the linestring's last point.

The **ST_PointN()** function takes an ST_LineString and an index to *n*th point and returns that point.

The **ST_Length()** function returns the linestring's length as a double-precision number.

The **ST_NumPoints()** function returns the number of points in the linestring's sequence as an integer.

The **ST_IsRing()** function returns `t` (TRUE) if the ST_LineString is a ring and `f` (FALSE) otherwise.

The **ST_IsClosed()** function returns `t` (TRUE) if the ST_LineString is closed and `f` (FALSE) otherwise.

The **ST_Polygon()** function creates a polygon from an ST_LineString that is a ring.

ST_Polygon

ST_Polygon is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. Most often, ST_Polygon defines parcels of land, water bodies, and other features having spatial extent.

Properties of ST_Polygon

ST_Polygon is always simple. The exterior and any interior rings define the boundary of an ST_Polygon, and the space enclosed between the rings defines the ST_Polygon's interior. The rings of an ST_Polygon can intersect at a tangent point, but never cross. In addition to the other properties inherited from the superclass ST_Geometry, ST_Polygon has area.

Figure 2-3 shows examples of ST_Polygon objects: (1) is an ST_Polygon whose boundary is defined by an exterior ring; (2) is an ST_Polygon whose boundary is defined by an exterior ring and two interior rings. The area inside the interior rings is part of the ST_Polygon's exterior; (3) is a legal ST_Polygon because the rings intersect at a single tangent point.

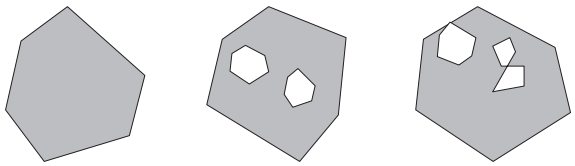


Figure 2-3. Examples of ST_Polygon Objects

Functions That Operate on ST_Polygon

Functions that operate on ST_Polygon include **ST_Area()**, **ST_ExteriorRing()**, **ST_NumInteriorRing()**, **ST_InteriorRingN()**, **ST_Centroid()**, and **ST_PointOnSurface()**.

The **ST_Area()** function returns the polygon's area as a double-precision number.

The **ST_ExteriorRing()** function returns the polygon's exterior ring as an ST_LineString.

The **ST_NumInteriorRing()** function returns the number of interior rings that the polygon contains.

The **ST_InteriorRingN()** function takes an ST_Polygon and an index and returns the *n*th interior ring as an ST_LineString.

The **ST_Centroid()** function returns an ST_Point that is the center of the ST_Polygon's envelope.

The **ST_PointOnSurface()** function returns an ST_Point that is guaranteed to be on the surface of the ST_Polygon.

The **ST_Perimeter()** function returns the perimeter of an ST_Polygon or ST_MultiPolygon.

ST_MultiPoint

ST_MultiPoint is a collection of ST_Points. ST_MultiPoint can define aerial broadcast patterns and incidents of a disease outbreak.

An ST_MultiPoint is simple if none of its elements occupy the same coordinate space. Just like its elements, it has a dimension of `0`. The boundary of a ST_MultiPoint is NULL.

ST_MultiLineString

ST_MultiLineString is a collection of ST_LineStrings. ST_MultiLineStrings are used to define streams or road networks.

Properties of ST_MultiLineStrings

ST_MultiLineStrings are simple if they only intersect at the endpoints of the ST_LineString elements. ST_MultiLineStrings are nonsimple if the interiors of the ST_LineString elements intersect.

The boundary of an ST_MultiLineString is the non-intersected endpoints of the ST_LineString elements. The ST_MultiLineString is closed if all its ST_LineString elements are closed. The boundary of a ST_MultiLineString is NULL if all the endpoints of all the elements are intersected. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiLineStrings have length.

Figure 2-4 on page 2-8 shows examples of ST_MultiLineStrings:

- (1) is a simple ST_MultiLineString whose boundary is the four endpoints of its two ST_LineString elements.
- (2) is a simple ST_MultiLineString because only the endpoints of the ST_LineString elements intersect. The boundary is two non-intersected endpoints.
- (3) is a non-simple ST_MultiLineString because the interior of one of its ST_LineString elements is intersected. The boundary of this ST_MultiLineString is the three non-intersected endpoints.
- (4) is a simple non-closed ST_MultiLineString. It is not closed because its element ST_LineStrings are not closed. It is simple because none of the interiors of any of the element ST_LineStrings intersect.
- (5) is a simple closed ST_MultiLineString. It is closed because all its elements are closed. It is simple because none of its elements intersect at the interiors.

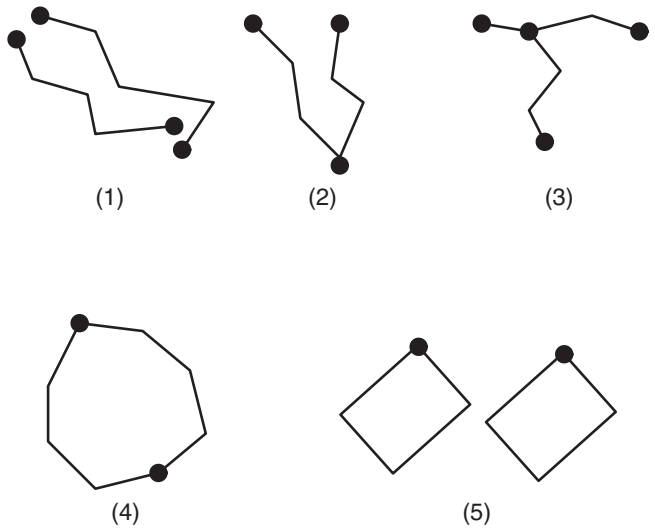


Figure 2-4. Examples of ST_MultiLineString Objects

Functions That Operate on ST_MultiLineStrings

Functions that operate on ST_MultiLineStrings include **ST_Length()** and **ST_IsClosed()**.

The **ST_Length()** function returns the cumulative length of all its ST_LineString elements as a double-precision number.

The **ST_IsClosed()** function returns `t` (TRUE) if the ST_MultiLineString is closed and `f` (FALSE), otherwise.

ST_MultiPolygon

ST_MultiPolygons define features such as a forest stratum or a non-contiguous parcel of land such as an island chain.

Properties of ST_MultiPolygon

The boundary of an ST_MultiPolygon is the cumulative length of its elements' exterior and interior rings. The interior of an ST_MultiPolygon is defined as the cumulative interiors of its element ST_Polygons. The boundary of an ST_MultiPolygon's elements can only intersect at a tangent point. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiPolygons have area.

Figure 2-5 shows examples of ST_MultiPolygon: (1) is ST_MultiPolygon with two ST_Polygon elements. The boundary is defined by the two exterior rings and the three interior rings; (2) is an ST_MultiPolygon with two ST_Polygon elements. The boundary is defined by the two exterior rings and the two interior rings. The two ST_Polygon elements intersect at a tangent point.

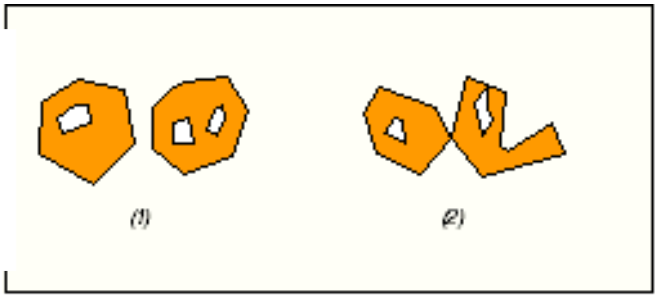


Figure 2-5. Examples of ST_MultiPolygon Objects

Functions That Operate on ST_MultiPolygons

Functions that operate on ST_MultiPolygons include **ST_Area()**, **ST_Centroid()**, and **ST_PointOnSurface()**.

The **ST_Area()** function returns the cumulative area of its ST_Polygon elements as a double-precision number.

The **ST_Centroid()** function returns an ST_Point that is the center of an ST_MultiPolygon's envelope.

The **ST_PointOnSurface()** function returns an ST_Point that is guaranteed to be on the surface of one of its ST_Polygon elements.

Locale Override

The external text representation of double-precision numbers in IBM Informix Spatial DataBlade Module types follows the U.S. English locale (**en_us.8859-1**). In this standard, all text input must use dots (.) as decimal separators and must be single-byte ASCII, regardless of the locale. (Internally, the IBM Informix Spatial DataBlade Module overrides the current locale with the U.S. English locale.)

For example, in many European locales, the decimal separator is a comma. You can keep using a non-English locale, but you must use dots in all text input, as follows:

```
ST_PointFromText('point zm (10.01 20.04 3.2 9.5)', 1)
```

This is true even if in your locale you normally use `3,2` instead of `3.2`. The IBM Informix Spatial DataBlade Module always produces dots when it generates external text representations, regardless of the locale.

Using Spatial Data Types with SPL

If you are working with SPL, there are two restrictions you need to be aware of:

- LVARCHAR variables in an SPL routine are restricted to 2K.
- SPL does not yet support a global variable defined on a UDT or complex type.

Chapter 3. Data Exchange Formats

In This Chapter

The IBM Informix Spatial DataBlade Module supports the following GIS data exchange formats.

- Well-known text representation
- Well-known binary representation
- ESRI shape representation
- Geography markup language (GML)
- Keyhole markup language (KML)

This chapter describes each of these formats.

Well-Known Text Representation

The IBM Informix Spatial DataBlade Module provides several functions that generate a geometry from the OGC well-known text (WKT) representation. The WKT is an ASCII text-formatted string that allows geometry to be exchanged in ASCII text form. You can use these functions in a third- or fourth-generation language (3GL or 4GL) program because they do not require the definition of any special program structures.

The functions are listed below. Chapter 8, "SQL Functions," on page 8-1, includes sample code fragments demonstrating how to use each of these functions.

The **ST_GeomFromText()** function creates an ST_Geometry from a text representation of any geometry type.

The **ST_PointFromText()** function creates an ST_Point from a point text representation.

The **ST_LineFromText()** function creates an ST_LineString from a linestring text representation.

The **ST_PolyFromText()** function creates an ST_Polygon from a polygon text representation.

The **ST_MPointFromText()** function creates an ST_MultiPoint from a multipoint representation.

The **ST_MLineFromText()** function creates an ST_MultiLineString from a multilinestring representation.

The **ST_MPolyFromText()** function creates an ST_MultiPolygon from a multipolygon representation.

The **ST_AsText()** function converts an existing geometry into a text representation.

Appendix C, "OGC Well-Known Text Representation of Geometry," on page C-1, gives a detailed description of the text representation.

Well-Known Binary Representation

The IBM Informix Spatial DataBlade Module provides several functions that generate a geometry from the OGC well-known binary (WKB) representation. The WKB representation is a contiguous stream of bytes. It permits geometry to be exchanged between a client application and an SQL database in binary form.

The functions are listed below. Chapter 8, "SQL Functions," on page 8-1, includes sample code fragments demonstrating how to use each of these functions.

The **ST_GeomFromWKB()** function creates an ST_Geometry from a WKB representation of any geometry type.

The **ST_PointFromWKB()** function creates an ST_Point from a point WKB representation.

The **ST_LineFromWKB()** function creates an ST_LineString from a linestring WKB representation.

The **ST_PolyFromWKB()** function creates an ST_Polygon from a polygon WKB representation.

The **ST_MPointFromWKB()** function creates an ST_MultiPoint from a multipoint WKB representation.

The **ST_MLineFromWKB()** function creates an ST_MultiLineString from a multilinestring WKB representation.

The **ST_MPolyFromWKB()** function creates an ST_MultiPolygon from a multipolygon WKB representation.

The **ST_AsBinary()** function converts an existing geometry value into well-known binary representation.

Appendix D, "OGC Well-Known Binary Representation of Geometry," on page D-1, provides a detailed description of WKB format.

ESRI Shape Representation

The IBM Informix Spatial DataBlade Module provides several functions that generate a geometry from an *ESRI shape representation*. In addition to the two-dimensional representation supported by the Open GIS well-known binary representation, the ESRI shape representation also supports optional Z coordinates and measures.

The following functions generate geometry from an ESRI shape. Chapter 8, "SQL Functions," on page 8-1, includes sample code fragments demonstrating how to use each of these functions.

The **SE_GeomFromShape()** function creates an ST_Geometry from a shape of any geometry type.

The **SE_PointFromShape()** function creates an ST_Point from a point shape.

The **SE_LineFromShape()** function creates an ST_LineString from a polyline shape.

The **SE_PolyFromShape()** function creates an ST_Polygon from a polygon shape.

The **SE_MPointFromShape()** function creates an ST_MultiPoint from a multipoint shape.

The **SE_MLineFromShape()** function creates an ST_MultiLineString from a multipart polyline shape.

The **SE_MPolyFromShape()** function creates an ST_MultiPolygon from a multipart polygon shape

For all of these functions, the first argument is the shape representation and the second argument is the spatial reference identifier to assign to the **ST_Geometry**. For example, the **SE_GeomFromShape** function has the following syntax:

```
SE_GeomFromShape(shapegeometry, SRID)
```

The **SE_AsShape** function converts the geometry value into an ESRI shape representation.

Appendix E, "ESRI Shape Representation," on page E-1, provides a detailed description of the ESRI shape representation.

GML Representation

The IBM Informix Spatial DataBlade Module provides several functions that generate a geometry from a Geography Markup Language (GML) representation. The GML can be represented as either GML2 (OGC GML standard 2.1.2) or GML3 (OGC GML standard 3.1.1). In addition to the two-dimensional representation supported by the Open GIS well-known binary representation, the GML representation also supports optional Z coordinates and measures. The following functions generate a geometry from a GML string. "SQL Functions" on page 8-8 includes sample code fragments that demonstrate how to use each of these functions.

The **ST_GeomFromGML()** function creates an ST_Geometry from a string of any geometry type.

The **ST_PointFromGML()** function creates an ST_Point from a point string.

The **ST_LineFromGML()** function creates an ST_LineString from a polyline string.

The **ST_PolyFromShape()** function creates an ST_Polygon from a polygon string.

The **ST_MPointFromGML()** function creates an ST_MultiPoint from a multipoint string.

The **ST_MLineFromGML()** function creates an ST_MultiLineString from a multipart polyline string.

The **ST_MPolyFromGML()** function creates an ST_MultiPolygon from a multipart polygon string.

For all of these functions, the first argument is the GML representation and the second optional argument is the spatial reference identifier to assign to the ST_Geometry. For example, the **ST_GeomFromGML()** function has the following syntax:

```
ST_GeomFromGML(gml_string, [SRID])
```

```
 ST_GeomFromGML('<gml:Point srsName="ESPG:1234" srsDimension="2">
     <gml:pos>10.02 20.01</gml:pos></gml:Point>', 1000)
```

The **SE_AsGML()** and **ST_AsGML()** functions convert the geometry value into a GML representation.

The **ST_EnvelopeAsGML()** function converts an ST_Polygon into a GML3 Envelope element.

KML Representation

The IBM Informix Spatial DataBlade Module provides functions to generate a geometry from a Keyhole Markup Language (KML) representation. KML is an XML-based schema for expressing geographic annotation and visualization on online maps and earth browsers. The following functions generate a geometry from a KML string. "SQL Functions" on page 8-8 includes sample code fragments that demonstrate how to use each of these functions.

The **SE_EnvelopeFromKML()** function creates an ST_Polygon from a KML LatLonBox string.

The **ST_GeomFromKML()** function creates an ST_Geometry from a KML fragment.

The **ST_LineFromKML()** function creates an ST_LineString from a KML LineString string.

The **ST_MLineFromKML()** function creates an ST_MultiLineString from a KML MultiGeometry string.

The **ST_MPointFromKML()** function creates an ST_MultiPoint from a KML MultiGeometry and Point combination string.

The **ST_MPolyFromKML()** function creates an ST_MultiPolygon from a KML MultiGeometry and Polygon combination string.

The **ST_PointFromKML()** function creates an ST_Point from a KML Point string.

The **ST_PolyFromKML()** function creates an ST_Polygon from a KML Polygon string.

For most of these functions, the first argument is the KML representation and the second optional argument is the spatial reference identifier to assign to the ST_Geometry. For example, the **ST_LineFromKML()** function has the following syntax:

```
ST_LineFromKML(kmlstring lvarchar, SRID integer)
```

The **SE_AsKML()** and **ST_AsKML()** functions convert the geometry value into a GML representation.

The **ST_EnvelopeAsKML()** function converts an ST_Envelope bounding box into a KML LatLonBox string.

Chapter 4. Using R-Tree Indexes

In This Chapter

This chapter explains how to create indexes on data stored with the IBM Informix Spatial DataBlade Module in an IBM Informix Dynamic Server (IDS) database. An index organizes access to data so that entries can be found quickly, without searching every row.

The IBM Informix Spatial DataBlade Module supports the R-tree access method, which enables you to index multidimensional objects.

The syntax for creating an index is described in detail in the CREATE INDEX statement in *IBM Informix Guide to SQL: Syntax*. This chapter describes how to create an index using the data types and functions provided in the IBM Informix Spatial DataBlade Module. It also explains how queries use indexes, and it includes special considerations to keep in mind when writing queries involving spatial data.

About Indexes

Queries that use an index execute more quickly and provide a significant performance improvement. An R-tree index is a secondary data structure that organizes access to spatial data. For a detailed description of R-tree indexes, refer to the *IBM Informix R-Tree Index User's Guide*.

Tip: An index does not change the results of your query in any way. You do not need to use an index to execute a query; however, it does improve performance.

When you create an index, you can specify an *access method* and an *operator class* (if you do not specify an access method, the default B-tree access method is used). The access method organizes the data in a way that speeds up access. The operator class is used by the optimizer to determine whether to use an index in a query.

Access Methods

Two access methods are available to IBM Informix Spatial DataBlade Module users:

- B-tree
- R-tree

The B-Tree Access Method

The IBM Informix Dynamic Server (IDS) default access method is B-tree. If you do not specify an access method in the CREATE INDEX statement, B-tree is used. The B-tree access method creates a one-dimensional ordering that speeds access to traditional numeric or character data.

You can use B-tree to index a column of nonspatial data or to create a functional index on the results of a spatial function that returns numeric or character data. For example, you could create a functional B-tree index on the results of the **ST_NumPoints()** function because **ST_NumPoints()** returns an integer value.

Important: The B-tree access method indexes numeric and character data only. You cannot use the B-tree access method to index spatial data.

The R-Tree Access Method

The R-tree access method speeds access to multidimensional data. It organizes data in a tree-shaped structure, with bounding boxes at the nodes. Bounding boxes indicate the farthest extent of the data that is connected to the subtree below.

A search using an R-tree index can quickly descend the tree to find objects in the general area of interest and then perform more exact tests on the objects themselves. An R-tree index can improve performance because it eliminates the need to examine objects outside the area of interest. Without an R-tree index, a query would need to evaluate every object to find those that match the query criteria.

To create an R-tree index, you must specify an operator class that supports an R-tree index on the data type you want to index. The operator class you use with the IBM Informix Spatial DataBlade Module is **ST_Geometry_Ops**.

Creating an R-Tree Index

To use the R-tree access method, you create an index on a column of a spatial type using the following syntax:

```
CREATE INDEX index_name
ON table_name (column_name ST_Geometry_Ops)
USING RTREE (parameters)
index_options;
```

You must provide:

- *index_name*. The name to give your index
- *table_name*. The name of the table that contains the spatial column to index
- *column_name*. The name of the spatial column
- *parameters* (optional). The parameters available for R-tree indexes are bottom_up_build (described in "Bottom-up Versus Top-down Index Builds" on page 4-4), BOUNDING_BOX_INDEX, NO_SORT, sort_memory, and fill_factor.
- *index_options* (optional). The index options are FRAGMENT BY and IN.

You cannot rename databases that contain R-tree indexes.

Refer to *IBM Informix R-Tree Index User's Guide* for explanations of parameters and index options other than BOTTOM_UP_BUILD, BOUNDING_BOX_INDEX, and NO_SORT.

The BOTTOM_UP_BUILD, BOUNDING_BOX_INDEX, and NO_SORT parameters affect the size of the index and the speed at which it is built. The following table shows the valid combinations of these parameters.

Parameters Clause of CREATE INDEX Statement	Description
BOTTOM_UP_BUILD='no', BOUNDING_BOX_INDEX='no', NO_SORT='no'	Creates an index by inserting spatial objects into the R-tree one at a time A copy of each object's in-row data is stored at the leaf level of the R-tree. This is the default if the DBSPACETEMP parameter in your **onconfig** file is not defined.
BOTTOM_UP_BUILD='no', BOUNDING_BOX_INDEX='yes', NO_SORT='no'	Creates a more compact index from the top down Only the bounding boxes of each object are stored at the leaf level of the R-tree. • No temporary dbspace is required. • Requires IBM Informix Dynamic Server Version 9.21 or later.
BOTTOM_UP_BUILD='yes', BOUNDING_BOX_INDEX='no', NO_SORT='no'	Creates an index by sorting the spatial data and then building the R-tree from the bottom up This is generally faster than building an index from the top down. This is the default if you have a temporary dbspace *and* it is specified by the DBSPACETEMP parameter in your **onconfig** file.
BOTTOM_UP_BUILD='yes', BOUNDING_BOX_INDEX='no', NO_SORT='yes'	Creates an index in less time • Does not require a temporary dbspace • Spatial data must be presorted, either by loading the data in a predetermined order or by creating a clustered functional B-tree index (see information about the **SE_SpatialKey()** function in "SE_SpatialKey()" on page 8-138). • Requires IBM Informix Dynamic Server Version 9.21 or later
BOTTOM_UP_BUILD='yes', BOUNDING_BOX_INDEX='yes', NO_SORT='no'	Creates a more compact index from the bottom up, which is faster than building from the top down • This is the default if you have a temporary dbspace *and* it is specified by the DBSPACETEMP parameter in your **onconfig** file. • Spatial data need not be presorted. • Requires IBM Informix Dynamic Server Version 9.21 or later
BOTTOM_UP_BUILD='yes', BOUNDING_BOX_INDEX='yes', NO_SORT='yes'	Creates a more compact index in less time • Does not require a temporary dbspace • Spatial data must be presorted. • Requires IBM Informix Dynamic Server Version 9.21 or later

Important: If you are using the Version 9.21 IBM Informix server or a later version, *BOUNDING_BOX_INDEX*='yes' is the default.

Bottom-up Versus Top-down Index Builds

R-tree indexes are created using one of two options: *top-down* build or *bottom-up* build. When an index is built from the top down, spatial objects are inserted into the R-tree one at a time. Objects are grouped together in the R-tree according to their spatial proximity to one another.

When an index is built from the bottom up, objects are first sorted according to a numeric value which is generated by the **SE_SpatialKey()** function. Then the R-tree is built by inserting all objects into the leaf pages (the bottom level) of the tree in sorted order and then building a hierarchy of bounding boxes above the leaf pages.

An R-tree index can be built much more quickly from the bottom up than from the top down, typically 10 to 20 times faster. This increased speed comes at a price: you must create a temporary dbspace of sufficient size to sort all the spatial data in the table. To determine the size, use the following formula:

Size (in bytes) = Number of rows in table * 1000

If you are using IBM Informix Dynamic Server Version 9.21 or later, the BOUNDING_BOX_INDEX option substantially reduces the temporary dbspace size requirement. The formula becomes:

Size (in bytes) = Number of rows in table * 100

Refer to your *Administrator's Guide for IBM Informix Dynamic Server* for more information about dbspaces.

The NO_SORT option eliminates the need for a temporary dbspace, but your spatial data *must* be presorted in the table. This means that it must be inserted into the table in sorted order, or sorted by means of a clustered B-tree index after it is loaded. For an example of how to sort spatial data, refer to "SE_SpatialKey()" on page 8-138.

You should also note that the bottom-up build option only makes sense if you load all of your spatial data into a table before you create an R-tree index. There is no performance advantage to either build option if you create an index on an empty table and then insert data.

Depending on certain characteristics of your spatial data, an index built from the top down may be more effective than one built from the bottom up. Properties that can adversely affect a bottom-up built index include many objects that overlap each other, or many objects that are very close together, such that the sort key produced by the **SE_SpatialKey()** function is the same value for a large number of objects.

Functional R-Tree Indexes

Under certain conditions, you can create a functional R-tree index, using one of several functions provided by the IBM Informix Spatial DataBlade Module. A functional index is built using the value returned by a function rather than the value of a column in a table.

For example, you can create an R-tree index on the centroids of the objects in a table, rather than on the objects themselves:

```
CREATE TABLE poly_tab (poly_col ST_Geometry);

CREATE INDEX centroid_idx ON poly_tab
   (ST_Centroid(poly_col) ST_Geometry_ops) USING RTREE;
```

You can use the following Spatial DataBlade functions in a functional index:

- **ST_Centroid()**
- **ST_EndPoint()**
- **ST_Envelope()**
- **SE_Midpoint()**
- **ST_Point()**
- **ST_PointN()**
- **ST_PointOnSurface()**
- **ST_StartPoint()**

Verifying That the Index Is Correctly Built

To verify whether an index was successfully built, use the **oncheck** utility. The output from **oncheck** for a successfully built index starts as shown:

```
WARNING: index check requires a s-lock on tables whose lock level is page.
Information for Partition num: 1049006
Node: Level 0, Pagenum 32, Usage 55.7%, No. of Children 11, right -1
```

Refer to your *Administrator's Guide for IBM Informix Dynamic Server* for information about using the **oncheck** utility.

The Spatial Operator Class ST_Geometry_Ops

An operator class defines operators, called *strategy functions*, that may be able to use an index. The operator class that the IBM Informix Spatial DataBlade Module uses is **ST_Geometry_Ops**.

Warning: Do not use **rtree_ops**, the default R-tree operator class for the secondary access method.

The strategy functions for the **ST_Geometry_Ops** operator classes are:

- **ST_Contains()**
- **ST_Crosses()**
- **ST_Equals()**
- **SE_EnvelopesIntersect()**
- **ST_Intersects()**
- **SE_Nearest()**
- **SE_NearestBbox()**
- **ST_Overlaps()**
- **ST_Touches()**
- **ST_Within()**

These functions are described in detail in Chapter 8, "SQL Functions," on page 8-1.

How Spatial Operators Use R-Tree Indexes

Once you have created an index, it will be considered by the query optimizer for any supported combination of argument types. The index can be used regardless of whether the indexed column is the first or second argument. An operator with two parameters—also known as a *binary operator*—can take advantage of an index defined on a column that serves as one of its arguments. Chapter 8, "SQL Functions," on page 8-1, discusses spatial operators in detail.

Chapter 5. Running Parallel Queries

In This Chapter

This chapter explains how to run parallel queries using Version 8.11, or later, of the IBM Informix Spatial DataBlade Module. Running queries in parallel distributes the work for one aspect of a query among several processors and can dramatically improve performance.

You can use any of the SQL functions that Chapter 8, "SQL Functions," on page 8-1, describes in a parallel query, except the **SE_BoundingBox()**, **SE_CreateSRID()**, **SE_MetadataInit()**, and **SE_Trace()** functions.

Parallel Query Execution Infrastructure

The IBM Informix Spatial DataBlade Module creates and maintains various database objects to manage the execution of queries in parallel. Do not alter or delete any of the following items of infrastructure:

- The **SE_MetadataTable** table.
- The SE_Metadata opaque type.
- The metadata smart large object, which is stored in the default sbspace.
- The metadata lohandle file.
- The metadata memory cache.
- Triggers on the **spatial_references** table (see "Using Triggers with the spatial_references Table" on page 1-7 if you want to add your own triggers to the **spatial_references** table).

The DataBlade module also provides a repair function, **SE_MetadataInit()** (described next), that you can use to re-initialize this infrastructure in case of difficulty.

The SE_MetadataInit() Function

In some rare circumstances, you may need to run the **SE_MetadataInit()** function to resolve the following problems:

- The metadata lohandle file is corrupted or missing, or cannot be created.
- The metadata smart large object is corrupted or missing.
- The metadata memory cache is corrupted or locked.

The **SE_MetadataInit()** function cannot be run in parallel.

The **SE_MetadataInit()** function is described in "SE_MetadataInit()" on page 8-89.

Executing Parallel Queries

To run queries in parallel, you must use the IBM Informix Dynamic Server (IDS) parallel database query (PDQ) feature. To take full advantage of the performance enhancement PDQ provides, you need to use fragmented tables; table fragmentation allows you to store parts of a table on different disks.

Please refer to your *Performance Guide for IBM Informix Dynamic Server* for information about these topics. The *IBM Informix Performance Guide* contains a chapter that describes how the database server executes queries in parallel and how you can manage running queries in parallel; it also includes a chapter that describes how to work with fragmented tables.

Chapter 6. Sizing Your Spatial Data

In This Chapter

This chapter explains how to estimate the amount of space required for tables that contain spatial columns.

Further tuning information is available at the following URL:
http://www.ibm.com/software/data/informix/blades/spatial/

Estimating the Size of Spatial Tables

This section describes how to estimate the amount of space required by tables containing spatial data columns. The table size approximations discussed here should be within 100 MB accuracy for large tables; to make estimates of greater precision, refer to the *Performance Guide for IBM Informix Dynamic Server*.

To approximate the size of a spatial table:

1. Estimate the size of the spatial column.
2. Estimate the size of the non-spatial columns.
3. Estimate dbspace overhead requirements.
4. Combine the estimates from Steps 1, 2, and 3 to approximate the storage space for the spatial table.
5. Estimate the smart large object storage space.

The rest of this section describes how to estimate each of these quantities.

Estimating the Size of the Spatial Column

Estimate the size of the spatial column using the formula:

```
spatial column size = (average points per feature * coordinate factor) + annotation size
```

The *average points per feature* is the sum of all coordinate points required to render the features stored in a spatial table divided by the number of rows in the table. If the sum of all coordinates is difficult to obtain, use the approximations of the average number of points per feature for each data type in the table.

The collection data types (MultiPoints, MultiLineStrings, and MultiPolygons) are difficult to estimate. The numbers shown in the table are based on the types of data sets that these data types are most often applied to: broadcast patterns for MultiPoints, stream networks for MultiLineStrings, and island topology for MultiPolygons.

Data Type	Average Points per Feature
Point	1
LineString (urban)	5
LineString (rural)	50
Polygon (urban)	7
Polygon (rural)	150
MultiPoint	50
MultiLineString	250
MultiPolygon	1000

The *coordinate factor* is based on the dimensions of the coordinates stored by the spatial column. Select the coordinate factor from the table below.

Coordinate Type	Coordinate Factor
XY	4.8
XYZ	7.2
XY and measures	7.2
XYZ and measures	9.6

If your layer includes annotation, set the *annotation size* to 300 bytes (this is the average space required to store most annotation). This includes the space required to store text, placement geometry, lead line geometry, and various attributes describing the annotation's size and font.

Estimating the Size of Non-Spatial Columns

To determine the size of the remaining columns of a spatial table, create the table without the spatial column and query the **rowsize** column of the **systables** table. In this example, a table called **lots** is created with three columns:

```
create table lots (lot_id integer,
                   owner_name varchar(128),
                   owner_address varchar(128))
```

Selecting the row size for the **lots** table from **systables** returns a value of 262 bytes:

```
select rowsize from systables where tabname = 'lots';

262
```

Tables containing variable length columns of type VARCHAR or NVARCHAR require the row size to be reduced to reflect the actual length of the data stored. In this example, the **owner_name** and **owner_address** columns are variable length VARCHAR columns and may occupy up to 129 bytes each (128 bytes for data plus an extra byte for the null terminator). The average size of the owner name is actually 68 bytes, and the average size of the address is 102 bytes. Therefore, the estimated row size should be reduced to 174 bytes.

Estimating Dbspace Overhead Requirements

Spatial tables with more than 10,000 rows require about 200 bytes of overhead per row.

Spatial tables with fewer than 10,000 rows but more than 1,000 rows require 300 bytes per row.

Spatial tables with fewer than 1,000 rows require 400 bytes per row.

Deriving the Storage Space for the Table

To make a rough estimate of the storage space required by the spatial table:

1. Estimate the number of rows in the table.
2. Add the spatial column size, the non-spatial column size, and the dbspace overhead size together.
3. Multiply the sum by the estimated number of rows.

Estimating the Smart Large Object Storage Space

A spatial data value is stored in row if the value is less than or equal to 930 bytes. However, if the value is greater than 930 bytes, only a pointer of 64 bytes is stored in row. This pointer refers to the smart large object in which the actual data is stored.

To estimate the amount of smart large object storage space:

1. Determine the smart large object ratio using the following formula:

   ```
   smart large object ratio = (spatial column size / 1920)
   ```

   The smart large object ratio cannot be greater than 1. So if the smart large object ratio you calculate is greater than one, set it to 1.
2. Multiply your smart large object ratio by the number of rows in your table to obtain the amount of smart large object space used by your table:

   ```
   smart large object space = ( ( smart large object ratio ) *
          number of rows)
   ```
3. Determine the amount of in-line table space required using the following formula:

   ```
   in-line space = (size of spatial table) - (smart large object space)
   ```

Important: Smart large object sbspaces should be stored on a disk separate from both the table and the indexes.

Estimating the Size of Spatial Indexes

This section provides information relevant to ArcSDE service users.

The ArcSDE service creates and maintains two indexes whenever you add a spatial column to one of your tables. The ArcSDE service creates an R-tree index on the spatial column and a B-tree index on the SE_ROW_ID INTEGER column. The R-tree index is named **a***n*_**ix1** and the B-tree index is named **a**_*n*_**ix2**, where *n* is the spatial column's layer number assigned by the ArcSDE service.

The indexes are three percent greater than the dbspace overhead space and spatial column size of the table.

To calculate the index space requirements:

1. Combine the spatial column size and the dbspace overhead space size.
2. Multiply this sum by the number of rows in the table.
3. Multiply the result of Step 2 by 1.03.

Chapter 7. Determining Spatial Relationships and Transforming Geometries

In This Chapter

This chapter describes SQL functions that:

- Determine spatial relationships
- Produce a new geometry
- Transform geometries

For information about all the SQL functions provided by the IBM Informix Spatial DataBlade Module, see Chapter 8, "SQL Functions," on page 8-1.

Functions That Determine Spatial Relationships

The IBM Informix Spatial DataBlade Module provides functions to determine whether a specific relationship exists between a pair of geometries. The distance separating a hazardous waste disposal site and a school is an example of a spatial relationship.

This section describes:

- **ST_Equals()**

- **ST_Disjoint()**
- **ST_Intersects()**
- **ST_Touches()**
- **ST_Overlaps()**
- **ST_Crosses()**
- **ST_Within()**
- **ST_Contains()**

The Dimensionally Extended 9 Intersection Model (DE-9IM) is a mathematical approach that defines the pair-wise spatial relationship between geometries of different types and dimensions. If you are interested in using this model to understand how the functions in this section operate, refer to "The DE-9IM Model" on page 7-20.

ST_Equals()

ST_Equals() returns `t` (TRUE) if two geometries of the same type have identical X,Y coordinate values.

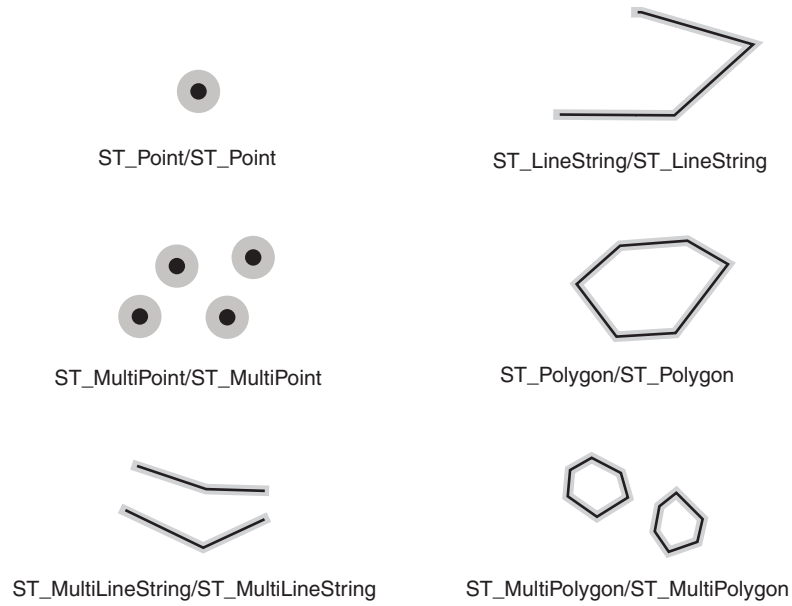


Figure 7-1. Equal Geometries

ST_Disjoint()

ST_Disjoint() returns t (TRUE) if two geometries do not intersect, overlap, or touch each other.

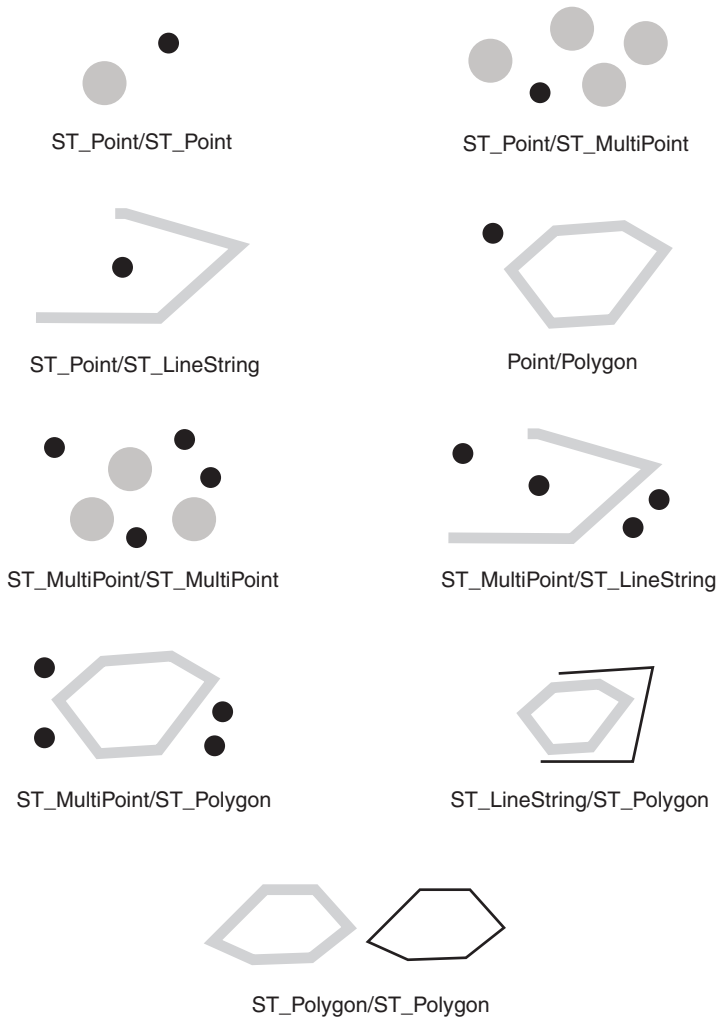


Figure 7-2. Disjoint Geometries

ST_Intersects()

ST_Intersects() returns `t` (TRUE) if two geometries intersect. **ST_Intersects()** returns the opposite result of **ST_Disjoint()**.

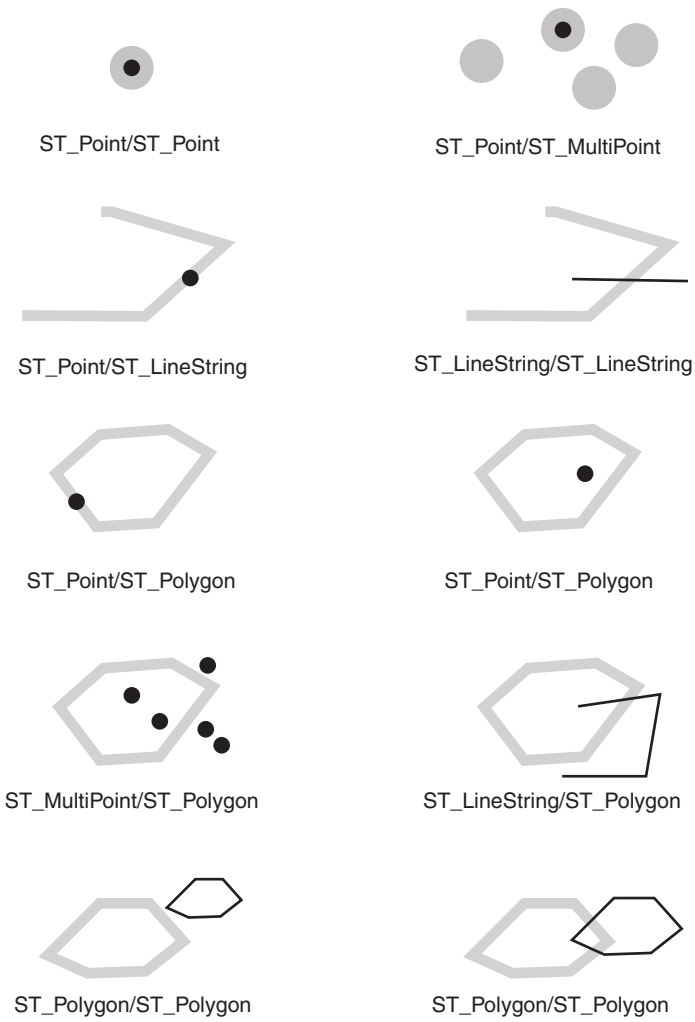


Figure 7-3. A Selection of Geometries that Intersect

ST_Touches()

ST_Touches() returns `t` (TRUE) when the interiors of two geometries do not intersect and the boundary of either geometry intersects the other's interior or boundary. At least one geometry must be an ST_LineString, ST_Polygon, ST_MultiLineString, or ST_MultiPolygon type.

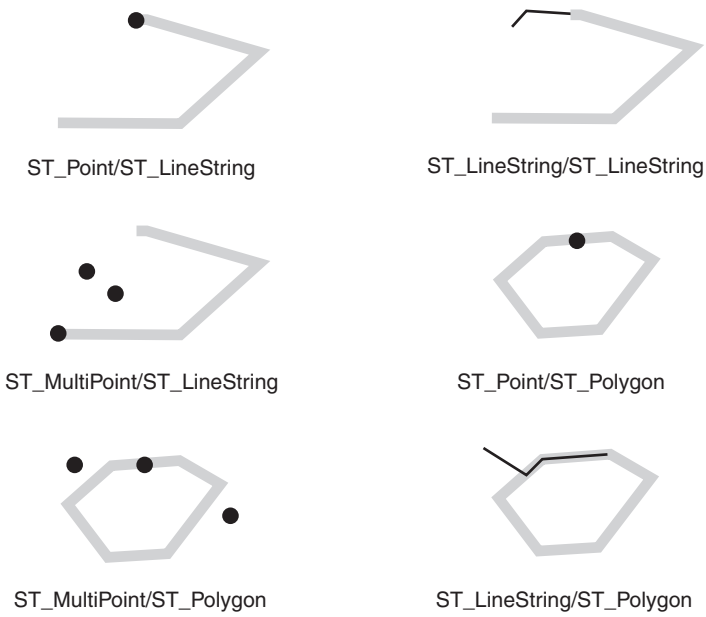


Figure 7-4. Touching Geometries

ST_Overlaps()

ST_Overlaps() determines whether two geometries overlap. **ST_Overlaps()** returns `t` (TRUE) only for geometries of the same dimension and only when their intersection set results in a geometry of the same dimension. In other words, if the intersection of two ST_Polygon types results in an ST_Polygon, then **ST_Overlaps()** returns `t` (TRUE).

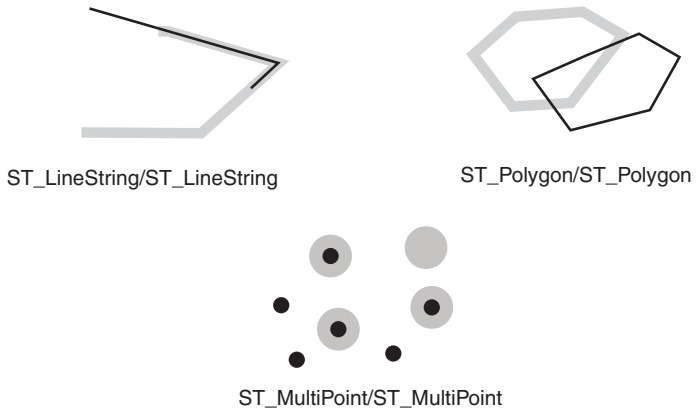


Figure 7-5. Overlapping Geometries

ST_Crosses()

ST_Crosses() determines whether the following pairs of geometries cross each other: ST_MultiPoint and ST_Polygon; ST_MultiPoint and ST_LineString; ST_LineString and ST_LineString; ST_LineString and ST_Polygon; and ST_LineString and ST_MultiPolygon.

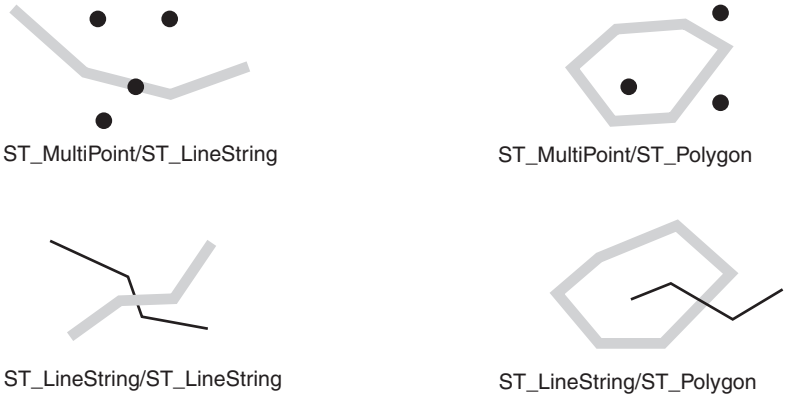


Figure 7-6. Crossing Geometries

ST_Within()

ST_Within() returns `t` (TRUE) if the first geometry is completely within the second geometry. The boundary and interior of the first geometry are not allowed to intersect the exterior of the second geometry. **ST_Within()** tests for the exact opposite result of **ST_Contains()**.

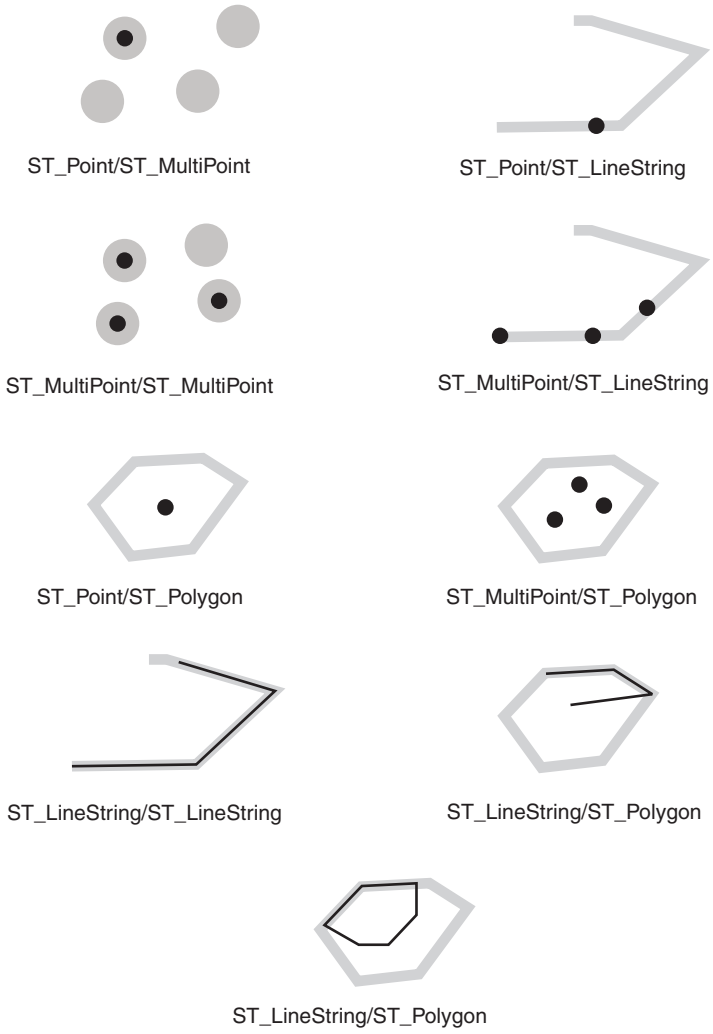


Figure 7-7. The ST_Within Function

ST_Contains()

ST_Contains() returns t (TRUE) if the second geometry is completely contained by the first geometry. The boundary and interior of the second geometry are not allowed to intersect the exterior of the first geometry. **ST_Contains()** returns the opposite result of **ST_Within()**.

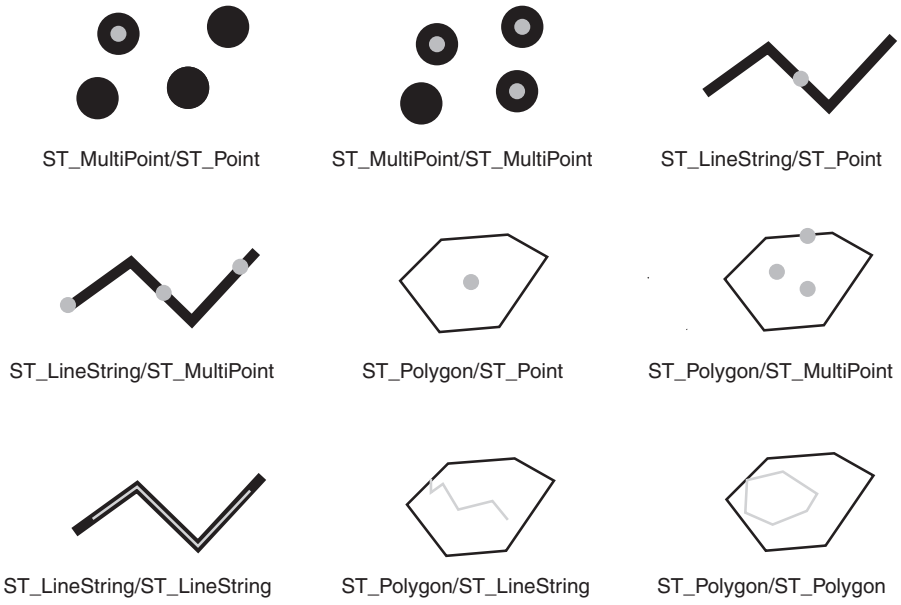


Figure 7-8. Contained Geometries

Functions That Produce a New Geometry

Some functions compare two existing geometries and return a new geometry based on how the two geometries are related. For example, the **ST_Difference()** function returns that portion of the first geometry that is not intersected or overlapped by the second.

This section describes:

- **ST_Intersection()**
- **ST_Difference()**
- **ST_Union()**
- **ST_SymDifference()**
- **SE_Dissolve()**

ST_Intersection()

The **ST_Intersection()** function returns the intersection set of two geometries. The intersection set is returned as a collection that is the minimum dimension of the source geometries. For example, for an ST_LineString that intersects an ST_Polygon, the **ST_Intersection()** function returns that portion of the ST_LineString common to the interior and boundary of the ST_Polygon as an ST_MultiLineString. The ST_MultiLineString contains more than one ST_LineString if the source ST_LineString intersects the ST_Polygon with two or more discontinuous segments.

If the geometries do not intersect or if the intersection results in a dimension less than both source geometries, an empty geometry is returned. The following figure illustrates some examples of the **ST_Intersection()** function.

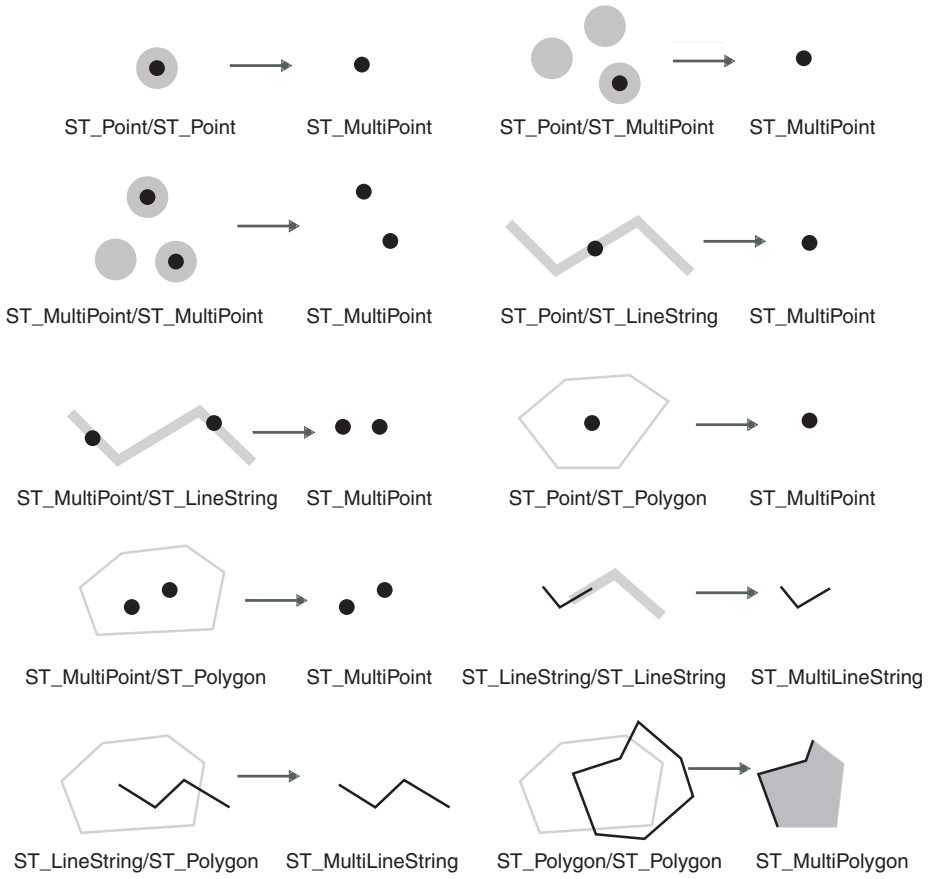


Figure 7-9. Intersection Sets of Geometries

ST_Difference()

The **ST_Difference()** function returns the portion of the primary geometry that is not intersected by the secondary geometry—the logical AND NOT of space. The **ST_Difference()** function only operates on geometries of like dimension and returns an ST_GeomCollection (ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon) that has the same dimension as the source geometries. In the event that the source geometries are equal, an empty geometry is returned.

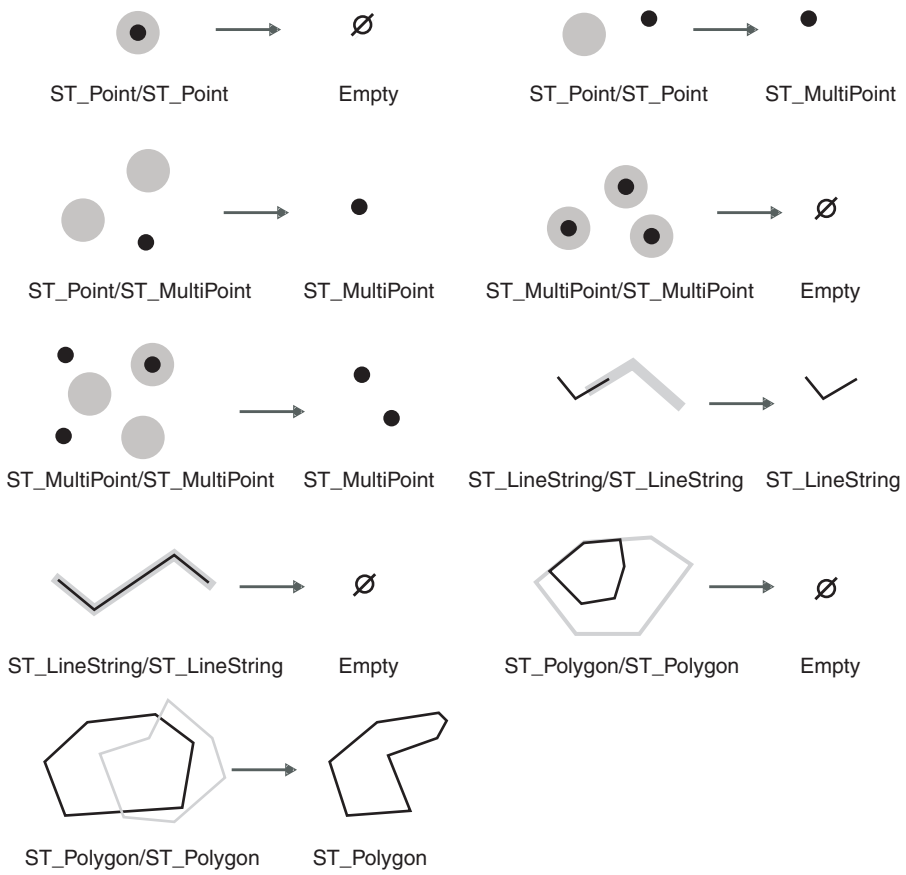


Figure 7-10. The ST_Difference() Function

ST_Union()

The **ST_Union()** function returns the union set of two geometries—the Boolean logical OR of space. The source geometries must have the same dimension. **ST_Union()** returns the result as an ST_GeomCollection (ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon).

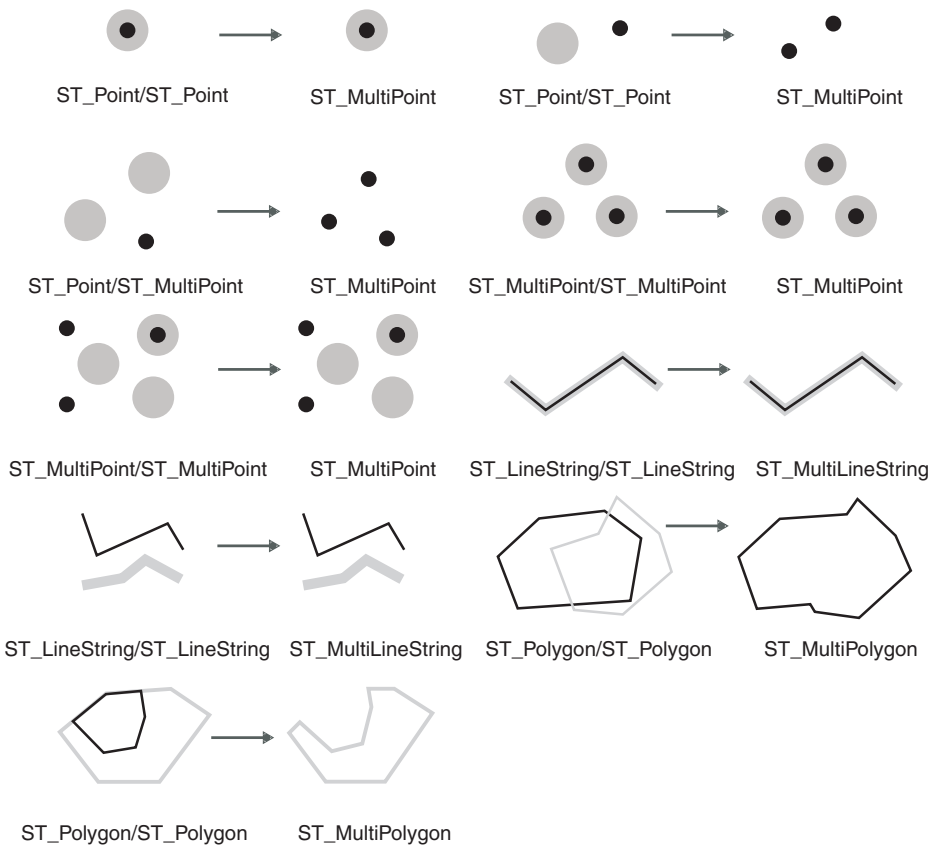


Figure 7-11. The Union Set of Two Geometries

ST_SymDifference()

The **ST_SymDifference()** function returns the symmetric difference of two geometries—the logical XOR of space (the portions of the source geometries that are not part of the intersection set). The source geometries must have the same dimension. If the geometries are equal, the **ST_SymDifference()** function returns an empty geometry; otherwise, the function returns the result as an ST_GeomCollection (ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon).

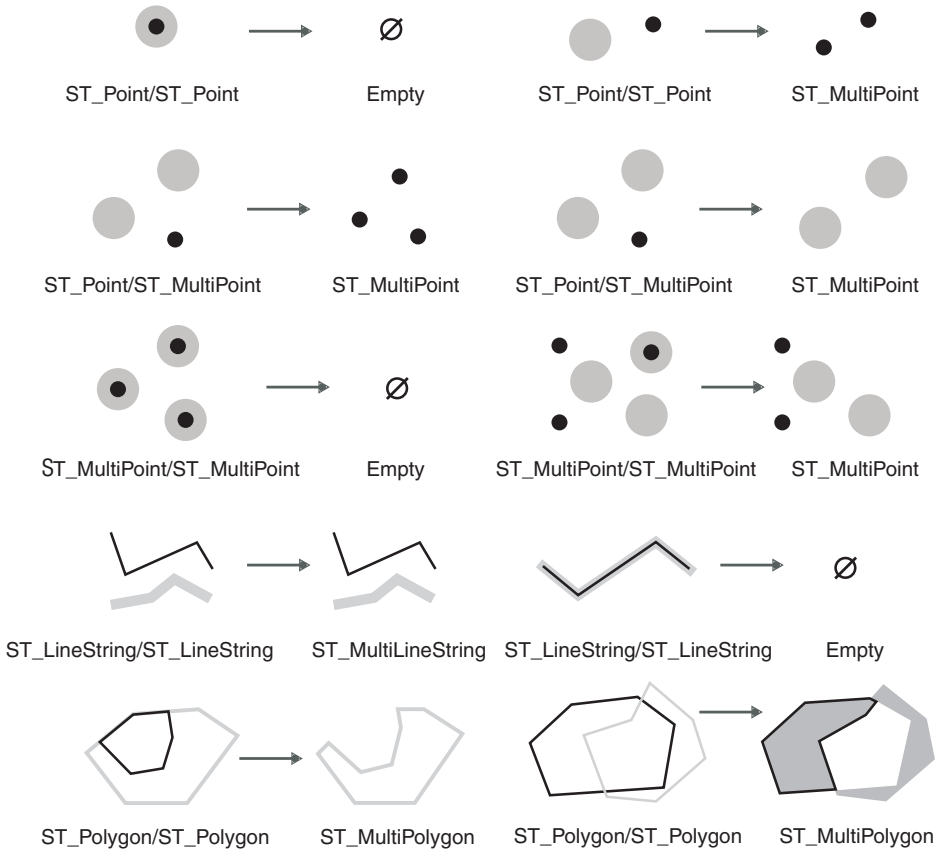


Figure 7-12. The Symmetric Difference of Geometries

SE_Dissolve()

SE_Dissolve() is an aggregate function that computes the union of geometries of the same dimension (if there is just one geometry that satisfies your query, it is returned unaltered).

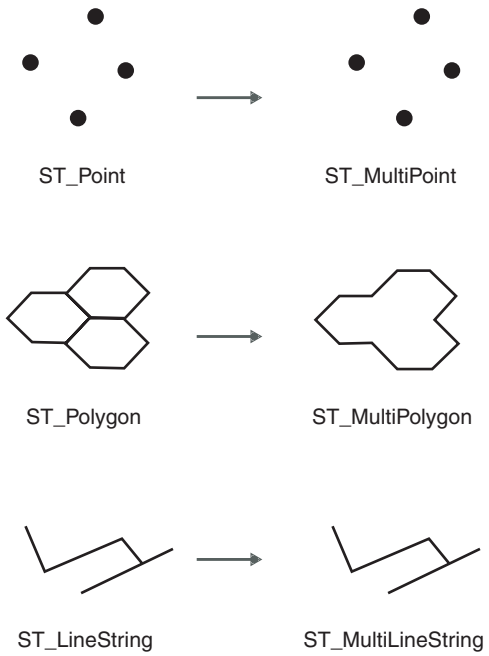


Figure 7-13. Geometries Resulting from Use of SE_Dissolve()

Functions That Transform Geometries

The IBM Informix Spatial DataBlade Module provides transformation functions that generate a new geometry from an existing geometry and a formula.

This section describes:

- **ST_Buffer()**
- **SE_LocateAlong()**
- **SE_LocateBetween()**
- **ST_ConvexHull()**
- **SE_Generalize()**

ST_Buffer

The **ST_Buffer()** function generates a geometry by encircling a geometry at a specified distance. A single polygon results when a primary geometry is buffered or when the buffer polygons of a collection are close enough to overlap. When enough separation exists between the elements of a buffered collection, individual buffer ST_Polygons result in an ST_MultiPolygon.

The **ST_Buffer()** function accepts both positive and negative distances, but only geometries with a dimension of 2 (ST_Polygon and ST_MultiPolygon) can apply a negative buffer. The absolute value of the buffer distance is used when the dimension of the source geometry is less than 2 (all geometries that are neither ST_Polygon nor ST_MultiPolygon). Generally speaking, positive buffer distances generate polygon rings that are away from the center of the source geometry, and for the exterior ring of a ST_Polygon or ST_MultiPolygon, toward the center when the distance is negative. For interior rings of an ST_Polygon or ST_MultiPolygon, the buffer ring is toward the center when the buffer distance is positive and away from the center when it is negative.

The buffering process merges buffer polygons that overlap. Negative distances greater than one-half the maximum interior width of a polygon result in an empty geometry.

As shown in Figure 7-14, **ST_Buffer()** generates a polygon or a multipolygon surrounding a given geometry at a specified radius.

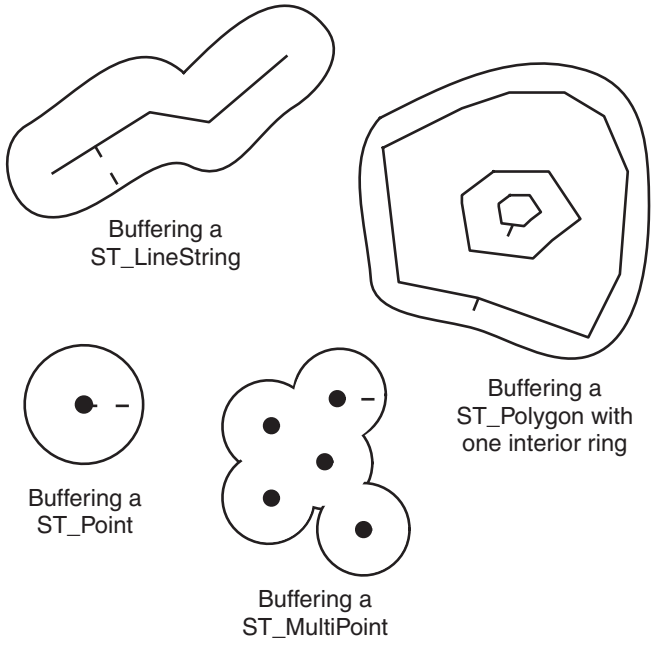


Figure 7-14. Buffers Generated by the ST_Buffer() Function

SE_LocateAlong()

For geometries that have measures, the location of a particular measure can be found with the **SE_LocateAlong()** function. **SE_LocateAlong()** returns the location as an ST_MultiPoint.

If the source geometry's dimension is 0 (for ST_Point and ST_MultiPoint), only points with a matching measure value are returned as an ST_MultiPoint. However, for source geometries whose dimension is greater than 0, the location is interpolated. For example, if the requested measure value is 5.5 and the ST_LineString vertices have measures of 3, 4, 5, 6, and 7, an interpolated point that falls exactly halfway between the vertices with measure values 5 and 6 is returned.

SE_LocateAlong() locates a point on a linestring by interpolating the given measure value, if necessary. Figure 7-15 shows a case where a point with measure 5.5 is interpolated halfway between the vertices of the ST_LineString with measures 5 and 6. For ST_MultiPoints, an exact match is required. In the case of the above ST_MultiPoint, **SE_LocateAlong()** returns the point that has measure 5.5.

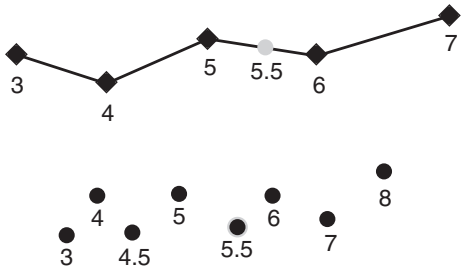


Figure 7-15. The SE_LocateAlong Function

SE_LocateBetween()

The **SE_LocateBetween()** function returns the set of paths or locations that lie between two measure values from a source geometry.

If the source geometry dimension is 0, **SE_LocateBetween()** returns an ST_MultiPoint consisting of all points whose measures lie between the two source measures.

For source geometries whose dimension is greater than 0, **SE_LocateBetween()** returns an ST_MultiLineString if a path can be interpolated; otherwise, **SE_LocateBetween()** returns an ST_MultiPoint containing the point locations.

An empty point is returned whenever **SE_LocateBetween()** cannot interpolate a path or find a location between the measures.

SE_LocateBetween() performs an inclusive search of the geometries; therefore, the geometry measures must be greater than or equal to the *from* measure and less than or equal to the *to* measure.

In Figure 7-16, **SE_LocateBetween()** returns an ST_MultiLineString that is between measures 4.3 and 6.9.

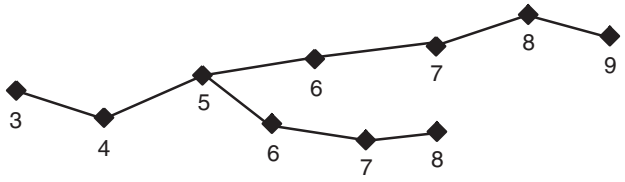


Figure 7-16. The SE_LocateBetween Function

ST_ConvexHull()

The **ST_ConvexHull()** function returns the convex hull polygon of any geometry that has at least three vertices forming a convex. Creating a convex hull is often the first step when tessellating a set of points to create a triangulated irregular network (TIN). If vertices of the geometry do not form a convex, **ST_ConvexHull()** returns a null.

ST_ConvexHull() generates an ST_Polygon value from the convex hull of three of the geometries pictured in Figure 7-17. **ST_ConvexHull()** returns a null for the two-point ST_LineString because it does not form a convex hull

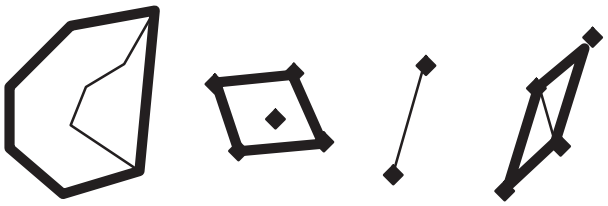


Figure 7-17. The ST_ConvexHull() Function.

SE_Generalize()

The **SE_Generalize()** function reduces the number of vertices in an ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon while preserving the general character of the geometric shape.

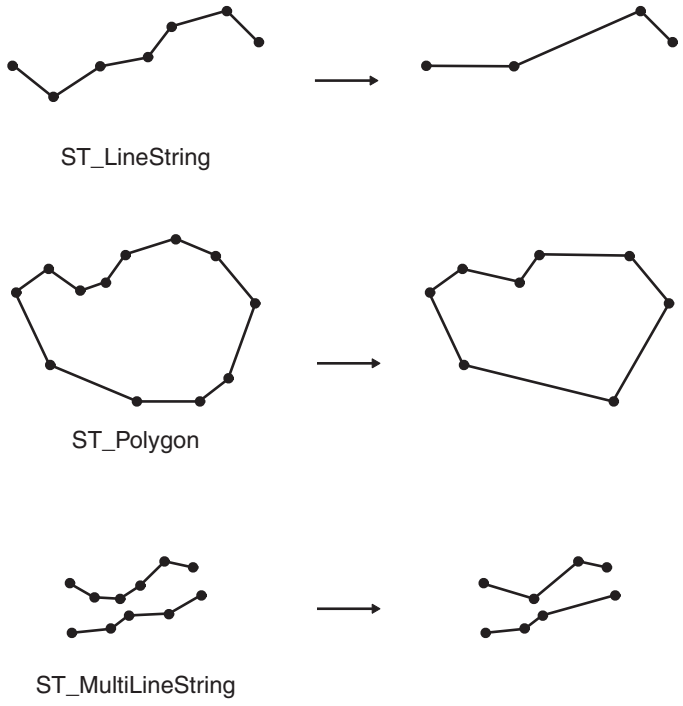


Figure 7-18. Geometries Resulting from Use of SE_Generalize()

The DE-9IM Model

The Dimensionally Extended 9 Intersection Model (DE-9IM) developed by Clementini and others, dimensionally extends the 9 Intersection Model of Egenhofer and Herring. DE-9IM is a mathematical approach that defines the pair-wise spatial relationship between geometries of different types and dimensions. This model expresses spatial relationships among all types of geometry as pair-wise intersections of their interior, boundary, and exterior with consideration for the dimension of the resulting intersections.

Given geometries *a* and *b*, I(*a*), B(*a*), and E(*a*) represent the interior, boundary, and exterior of *a*, and I(*b*), B(*b*), and E(*b*) represent the interior, boundary, and exterior of *b*. The intersections of I(*a*), B(*a*), and E(*a*) with I(*b*), B(*b*), and E(*b*) produce a 3-by-3 matrix. Each intersection can result in geometries of different dimensions. For example, the intersection of the boundaries of two polygons could consist of a point and a linestring, in which case the dim function would return the maximum dimension of 1.

The dim function returns a value of -1, 0, 1, or 2. The -1 corresponds to the null set that is returned when no intersection was found or dim(Æ).

	Interior	Boundary	Exterior
Interior	dim(I(a)ÇI(b))	dim(I(a)ÇB(b))	dim(I(a)ÇE(b))
Boundary	dim(B(a)ÇI(b))	dim(B(a)ÇB(b))	dim(B(a)ÇE(b))
Exterior	dim(E(a)ÇI(b))	dim(E(a)ÇB(b))	dim(E(a)ÇE(b))

The results of the spatial relationship for the functions described in "Functions That Determine Spatial Relationships" on page 7-1 can be understood or verified by comparing the results with a pattern matrix that represents the acceptable values for the DE-9IM.

The pattern matrix contains the acceptable values for each of the intersection matrix cells. The possible pattern values are:

T	An intersection must exist; dim = 0, 1, or 2.
F	An intersection must not exist; dim = -1.
*	It does not matter if an intersection exists or not; dim = -1, 0, 1, or 2.
0	An intersection must exist and its maximum dimension must be 0; dim = 0.
1	An intersection must exist and its maximum dimension must be 1; dim = 1.
2	An intersection must exist and its maximum dimension must be 2; dim = 2.For example, the pattern matrix of the **ST_Within()** function for geometry combinations has the following form:

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	F
a	Boundary	*	*	F
	Exterior	*	*	*

Simply put, the **ST_Within()** function returns TRUE when the interiors of both geometries intersect, and the interior and boundary of geometry *a* does not intersect the exterior of geometry *b*. All other conditions do not matter.

ST_Equals()

The DE-9IM pattern matrix for the **ST_Equals()** function ensures that the interiors intersect and that no interior or boundary of either geometry intersects the exterior of the other.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	F
a	Boundary	*	*	F
	Exterior	F	F	*

ST_Disjoint()

The **ST_Disjoint()** function pattern matrix simply states that neither the interiors nor the boundaries of either geometry intersect.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	F	F	*
a	Boundary	F	F	*
	Exterior	*	*	*

ST_Intersects()

The **ST_Intersects()** function returns TRUE if the conditions of any of the following pattern matrices returns TRUE.

The **ST_Intersects()** function returns TRUE if the interiors of both geometries intersect.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	*
a	Boundary	*	*	*
	Exterior	*	*	*

The **ST_Intersects()** function returns TRUE if the boundary of the first geometry intersects the boundary of the second geometry.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	*	T	*
a	Boundary	*	*	*
	Exterior	*	*	*

The **ST_Intersects()** function returns TRUE if the boundary of the first geometry intersects the interior of the second.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	*	*	*
a	Boundary	T	*	*
	Exterior	*	*	*

The **ST_Intersects()** function returns TRUE if the boundaries of either geometry intersect.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	*	*	*
a	Boundary	*	T	*
	Exterior	*	*	*

ST_Touches()

The pattern matrices show us that the **ST_Touches()** function returns TRUE when the interiors of the geometry do not intersect, and the boundary of either geometry intersects the other's interior or boundary.

The **ST_Touches()** function returns TRUE if the boundary of one geometry intersects the interior of the other but the interiors do not intersect.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	F	T	*
a	Boundary	*	*	*
	Exterior	*	*	*

The **ST_Touches()** function returns TRUE if the boundary of one geometry intersects the interior of the other but the interiors do not intersect.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	F	*	*
a	Boundary	T	*	*
	Exterior	*	*	*

The **ST_Touches()** function returns TRUE if the boundaries of both geometries intersect but the interiors do not.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	F	*	*
a	Boundary	*	T	*
	Exterior	*	*	*

ST_Overlaps()

This pattern matrix applies to ST_Polygon and ST_Polygon; ST_MultiPoint and ST_MultiPoint; and ST_MultiPolygon and ST_MultiPolygon overlaps. For these combinations, the **ST_Overlaps()** function returns TRUE if the interior of both geometries intersects the other's interior and exterior.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	T
a	Boundary	*	*	*
	Exterior	T	*	*

This pattern matrix applies to ST_LineString and ST_LineString; and to ST_MultiLineString and ST_MultiLineString overlaps. In this case the intersection of the geometries must result in a geometry that has a dimension of 1 (another ST_LineString or ST_MultiLineString). If the dimension of the intersection of the interiors had resulted in 0 (a point), the **ST_Overlaps()** function would return FALSE; however, the **ST_Crosses()** function would have returned TRUE.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	1	*	T
a	Boundary	*	*	*
	Exterior	T	*	*

ST_Crosses()

This **ST_Crosses()** function pattern matrix applies to ST_MultiPoint and ST_LineString; ST_MultiPoint and ST_MultiLineString; ST_MultiPoint and ST_Polygon; ST_MultiPoint and ST_MultiPolygon; ST_LineString and ST_Polygon; and ST_LineString and ST_MultiPolygon. The matrix states that the interiors must intersect and that at least the interior of the primary (geometry *a*) must intersect the exterior of the secondary (geometry *b*).

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	T
a	Boundary	*	*	*
	Exterior	*	*	*

This **ST_Crosses()** function matrix applies to ST_LineString and ST_LineString; ST_LineString and ST_MultiLineString; and ST_MultiLineString and ST_MultiLineString. The matrix states that the dimension of the intersection of the interiors must be 0 (intersect at a point). If the dimension of this intersection was 1 (intersect at a linestring), the **ST_Crosses()** function would return FALSE but the **ST_Overlaps()** function would return TRUE.

		b		
		Interior	**Boundary**	**Exterior**
	Interior	0	*	*
a	Boundary	*	*	*
	Exterior	*	*	*

ST_Within()

The **ST_Within()** function pattern matrix states that the interiors of both geometries must intersect and that the interior and boundary of the primary geometry (geometry *a*) must not intersect the exterior of the secondary (geometry *b*).

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	F
a	Boundary	*	*	F
	Exterior	*	*	*

ST_Contains()

The pattern matrix of the **ST_Contains()** function states that the interiors of both geometries must intersect, and that the interior and boundary of the secondary (geometry *b*) must not intersect the exterior of the primary (geometry *a*).

		b		
		Interior	**Boundary**	**Exterior**
	Interior	T	*	*
a	Boundary	*	*	*
	Exterior	F	F	*

Chapter 8. SQL Functions

In This Chapter

This chapter provides reference information for the SQL functions included with the IBM Informix Spatial DataBlade Module.

Summary of Functions by Task Type

The following table shows the functions available with the IBM Informix Spatial DataBlade Module, arranged by task type.

Function Task Type	Description	Function Name
Generate a geometry from a well-known text representation.	Takes a well-known text representation and returns an ST_Geometry using SRID 0	**ST_WKTToSQL()**
	Takes a well-known text representation and returns an ST_Geometry	**ST_GeomFromText()**
	Takes a well-known text representation and returns an ST_Point	**ST_PointFromText()**
	Takes a well-known text representation and returns an ST_LineString	**ST_LineFromText()**
	Takes a well-known text representation and returns an ST_Polygon	**ST_PolyFromText()**
	Takes a well-known text representation and returns an ST_MultiLine	**"ST_MLineFromText()" on page 8-95**
	Takes a well-known text representation and returns an ST_MultiPoint	**ST_MPointFromText()**
	Takes a well-known text representation and returns an ST_MultiPolygon	**ST_MPolyFromText()**

Function Task Type	Description	Function Name
Generate a geometry from a well-known binary representation.	Takes a well-known binary representation and returns an ST_Geometry using SRID 0	**ST_WKBToSQL()**
	Takes a well-known binary representation and returns an ST_Geometry	**ST_GeomFromWKB()**
	Takes a well-known binary representation and returns an ST_Point	**ST_PointFromWKB()**
	Takes a well-known binary representation and returns an ST_LineString	**ST_LineFromWKB()**
	Takes a well-known binary representation and returns an ST_Polygon	**ST_PolyFromWKB()**
	Takes a well-known binary representation and returns an ST_MultiLine	**"ST_MLineFromWKB()" on page 8-96**
	Takes a well-known binary representation and returns an ST_MultiPoint	**ST_MPointFromWKB()**
	Takes a well-known binary representation and returns an ST_MultiPolygon	**ST_MPolyFromWKB()**
Generate a geometry from an ESRI shape representation.	Takes an ESRI shape representation and returns an ST_Geometry using SRID 0	**SE_ShapeToSQL()**
	Takes an ESRI shape representation and returns an ST_Geometry	**SE_GeomFromShape()**
	Takes an ESRI shape representation and returns an ST_Point	**SE_PointFromShape()**
	Takes an ESRI shape representation and returns an ST_LineString	**SE_LineFromShape()**
	Takes an ESRI shape representation and returns an ST_Polygon	**SE_PolyFromShape()**
	Takes an ESRI shape representation and returns an ST_MultiLine	**"SE_MLineFromShape()" on page 8-94**
	Takes an ESRI shape representation and returns an ST_MultiPoint	**SE_MPointFromShape()**
	Takes an ESRI shape representation and returns an ST_MultiPolygon	**SE_MPolyFromShape()**
Generate a geometry from a GML representation.	Takes a GML2 or GML3 string representation of an envelope and returns an ST_Polygon	**"ST_EnvelopeFromGML()" on page 8-48**
	Takes a GML2 or GML3 string representation and returns an ST_Geometry	**"ST_GeomFromGML()" on page 8-58**
	Takes a GML2 or GML3 string representation and returns an ST_LineString	**"ST_LineFromGML()" on page 8-80**
	Takes a GML2 or GML3 string representation and returns an ST_MultiLineString	**"ST_MLineFromGML()" on page 8-91**
	Takes a GML2 or GML3 string representation and returns an ST_MultiPoint	**"ST_MPointFromGML()" on page 8-98**
	Takes a GML2 or GML3 string representation and returns an ST_MultiPolygon	**"ST_MPolyFromGML()" on page 8-103**
	Takes a GML2 or GML3 string representation and returns an ST_Point	**"ST_PointFromGML()" on page 8-121**
	Takes a GML2 or GML3 string representation and returns an ST_Polygon	**"ST_PolyFromGML()" on page 8-128**

Function Task Type	Description	Function Name
Generate a geometry from a KML representation.	Takes a KML LatLonBox string representation and returns an ST_Polygon	**"SE_EnvelopeFromKML()" on page 8-49**
	Takes a KML fragment string representation and returns an ST_Geometry	**"ST_GeomFromKML()" on page 8-60**
	Takes a KML LineString string representation and returns an ST_LineString	**"ST_LineFromKML()" on page 8-81**
	Takes a KML MultiGeometry string representation and returns an ST_MultiLineString	**"ST_MLineFromKML()" on page 8-93**
	Takes a KML MultiGeometry and Point combination string representation and returns an ST_MultiPoint	**"ST_MPointFromKML()" on page 8-99**
	Takes a KML MultiGeometry and Polygon combination string representation and returns an ST_MultiPolygon	**"ST_MPolyFromKML()" on page 8-104**
	Takes a KML Point string representation and returns an ST_Point	**"ST_PointFromKML()" on page 8-122**
	Takes a KML Polygon string representation and returns an ST_Polygon	**"ST_PolyFromKML()" on page 8-129**
Convert a geometry to an external format.	Returns the well-known text representation of a geometry object	**ST_AsText()**
	Returns the well-known binary representation of a geometry object	**ST_AsBinary()**
	Returns the ESRI shape representation of a geometry object	**SE_AsShape()**
	Returns the GML representation of a geometry object	**SE_AsGML()** or **"ST_AsGML()" on page 8-13**
	Returns the GML3 Envelope element of an ST_Envelope object	**"ST_EnvelopeAsGML()" on page 8-46**
	Returns the KML representation of a geometry object	**"ST_AsKML()" on page 8-16** or **"SE_AsKML()" on page 8-14**
	Returns the KML LatLonBox of an ST_Envelope object	**"SE_EnvelopeAsKML()" on page 8-47**
Manipulate the ST_Point data type.	Creates an ST_Point data type from an X and Y coordinate	**ST_Point()**
	Returns the X coordinate of a point	**ST_X()**
	Returns the Y coordinate of a point	**ST_Y()**
	Returns the Z coordinate of a point	**SE_Z()**
	Returns the measure value of a point	**SE_M()**
Manipulate the ST_LineString and ST_MultiLineString data types.	Returns the first point	**ST_StartPoint()**
	Returns the midpoint	**SE_Midpoint()**
	Returns the last point	**ST_EndPoint()**
	Returns the *n*th point	**ST_PointN()**
	Returns the length	**ST_Length()**
	Creates an ST_Polygon from a ring (closed linestring)	**ST_Polygon()**

Function Task Type	Description	Function Name
Manipulate the ST_Polygon and ST_MultiPolygon data types.	Calculates the area	**ST_Area()**
	Calculates the geometric center	**ST_Centroid()**
	Returns the exterior ring	**ST_ExteriorRing()**
	Counts the number of interior rings	**ST_NumInteriorRing()**
	Returns the *n*th interior ring	**ST_InteriorRingN()**
	Returns a point on the surface	**ST_PointOnSurface()**
	Returns the perimeter	**ST_Perimeter()**
Obtain parameters of a geometry.	Returns the dimensions of the coordinates	**ST_CoordDim()**
	Returns the dimension of the geometry	**ST_Dimension()**
	Returns the shortest distance to another geometry	**ST_Distance()**
	Returns the data type	**ST_GeometryType()**
	Returns the number of points	**ST_NumPoints()**
	Returns the spatial reference ID	**ST_SRID()**
	Returns the maximum and minimum X coordinate	**SE_Xmax() and SE_Xmin()**
	Returns the maximum and minimum Y coordinate	**SE_Ymax() and SE_Ymin()**
	Returns the maximum and minimum Z coordinate	**ST_Zmax() and ST_Zmin()**
	Returns the maximum and minimum measure value	**SE_Mmax() and SE_Mmin()**
Determine the properties of a geometry.	Determines whether a linestring or multilinestring is closed	**ST_IsClosed()**
	Determines whether a geometry is empty (it has no points)	**ST_IsEmpty()**
	Determines whether a geometry has measures	**SE_IsMeasured()**
	Determines whether a linestring is a ring (it is closed and simple)	**ST_IsRing()**
	Determines whether a geometry is simple	**ST_IsSimple()**
	Determines whether a geometry is topologically valid	**ST_IsValid()**
	Determines whether an object has Z coordinates	**SE_Is3D()**
	Tests whether two geometries meet the conditions specified by a specified DE-91M pattern matrix	**ST_Relate()**

Function Task Type	Description	Function Name
Determine if a certain relationship exists between two geometries.	Determines whether one geometry completely contains another	**ST_Contains()**
	Determines whether a geometry crosses another	**ST_Crosses()**
	Determines whether two geometries are completely non-intersecting	**ST_Disjoint()**
	Determines whether the envelopes of two geometries intersect	**SE_EnvelopesIntersect()**
	Determines whether two geometries are spatially equal	**ST_Equals()**
	Determines whether two geometries intersect	**ST_Intersects()**
	Determines whether two geometries overlap	**ST_Overlaps()**
	Determines whether two geometries touch	**ST_Touches()**
	Determines whether one object is completely inside another	**ST_Within()**
Compare geometries and return a new geometry based on how the geometries are related.	An aggregate function that computes the union of geometries of the same dimension	**SE_Dissolve()**
	Returns the portion of the primary geometry that is not intersected by the secondary geometry	**ST_Difference()**
	Computes the intersection of two geometries	**ST_Intersection()**
	Returns the perpendicular projection of a point on the nearest segment of a linestring	**SE_PerpendicularPoint()**
	Returns an ST_Geometry object that is composed of the parts of the two source geometries that are not common to both	**ST_SymDifference()**
	Computes the union of two geometries	**ST_Union()**
Generate a new geometry from an existing geometry.	Generates a geometry that is the buffer surrounding the source object	**ST_Buffer()**
	Generates the combined boundary of the geometry	**ST_Boundary()**
	Calculates the spatial extent of all geometries in a table column	**SE_BoundingBox()**
	Generates the convex hull of a geometry	**ST_ConvexHull()**
	Generates the bounding box of a geometry	**ST_Envelope()**
	Generates a geometry with fewer vertices but of the same general shape	**SE_Generalize()**
	Generates an ST_MultiPoint representing points that have the specified measure value	**SE_LocateAlong()**
	Generates an ST_MultiPoint or ST_MultiLineString representing points or paths that have measures in the specified range	**SE_LocateBetween()**
	Appends a vertex to a linestring	**SE_VertexAppend()**
	Deletes a vertex from a geometry	**SE_VertexDelete()**
	Updates a vertex in a geometry	**SE_VertexUpdate()**
Manage collections.	Returns the *n*th geometry	**ST_GeometryN()**
	Returns the number of geometries in the collection	**ST_NumGeometries()**

Function Task Type	Description	Function Name
Nearest-neighbor queries.	Retrieves nearby geometries in increasing distance order. **SE_NearestBbox()** is similar to **SE_Nearest()**, but measures distances between bounding boxes of geometries.	**SE_Nearest() and SE_NearestBbox()**
Assist in managing spatial reference systems.	Computes the false origin and system units for a data set and creates an entry in the **spatial_references** table	**SE_CreateSRID()**
	Returns the OGC well-known text representation of a spatial reference system	**SE_CreateSrtext()**
	Returns a geometry's spatial reference ID	**ST_SRID()**
	Returns the Authority Name and Authority SRID of a spatial reference ID	**"SE_SRID_Authority()" on page 8-141**
	Transforms a geometry from one spatial reference system to another	**ST_Transform()**
Administration	Returns the in-row, out-of-row and total size of a geometry in bytes	**SE_InRowSize() SE_OutOfRowSize() SE_TotalSize()**
	Reinitializes the spatial reference system memory cache	**SE_MetadataInit()**
	Returns the value of a specified parameter or all parameters if called with no parameters	**"SE_ParamGet() function" on page 8-116**
	Sets the specified parameter to a new value	**"SE_ParamSet() function" on page 8-117**
	Returns usage information (if called with no parameters)	**"SE_ParamSet() function" on page 8-117**
	Returns the version and release date of the IBM Informix Spatial DataBlade Module	**SE_Release()**
	Generates a sort key for geometries	**SE_SpatialKey()**
	Controls tracing to assist in debugging	**SE_Trace()**

SQL Functions

The rest of this chapter describes each of the SQL functions. All OpenGIS-ISO/SQLMM-compliant functions have the prefix *ST_*, while functions that extend the OpenGIS-ISO/SQLMM specification have the prefix *SE_*.

ST_Area()

ST_Area() returns the area of a polygon or multipolygon.

Syntax

```
ST_Area(pl1 ST_Polygon)
ST_Area(mpl1 ST_MultiPolygon)
```

Return Type

DOUBLE PRECISION

Example

The city engineer needs a list of building areas. To create the list, a GIS technician selects the building ID and area of each building footprint.

The building footprints are stored in the **buildingfootprints** table created with the following CREATE TABLE statement:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

To satisfy the city engineer's request, the technician selects the unique key, the **building_id**, and the **area** of each building footprint from the **buildingfootprints** table:

```
SELECT building_id, ST_Area(footprint) area
   FROM buildingfootprints;

building_id           area

        506 78.00000000000
        543 68.00000000000
       1208 123.0000000000
        178 87.00000000000
```

Figure 8-1 shows the four building footprints labeled with their building ID numbers and displayed alongside their adjacent street.

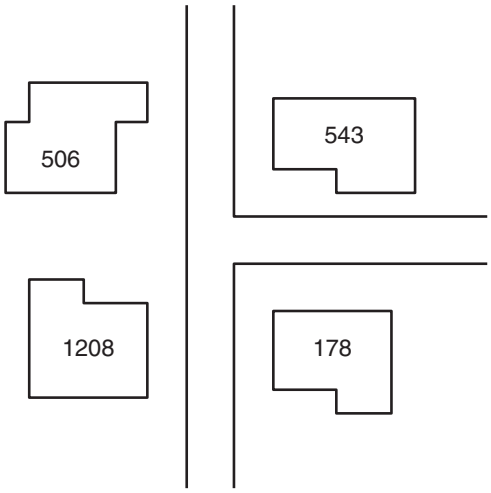


Figure 8-1. Building Footprints

ST_AsBinary()

ST_AsBinary() takes a geometry object and returns its well-known binary representation.

The return type of **ST_AsBinary()** is defined as ST_Geometry to allow spatial objects greater than 2 kilobytes to be retrieved by a client application. Typically, you use **ST_AsBinary()** to retrieve spatial data from the server and send it to a client, as in:

```
SELECT ST_AsBinary(geomcol) FROM mytable
```

IBM Informix Dynamic Server automatically casts the output of the **ST_AsBinary()** function to the proper data type for transmission to the client.

You can extend the functionality of the IBM Informix Spatial DataBlade Module by writing new user-defined routines (UDRs) in C or SPL. You can use **ST_AsBinary()** to convert an ST_Geometry to its well-known binary representation. If you pass the output of **ST_AsBinary()** to another UDR whose function signature requires an LVARCHAR input, you should explicitly cast the return type of **ST_AsBinary()** to LVARCHAR, as in:

```
EXECUTE FUNCTION MySpatialFunc(ST_AsBinary(geomcol)::lvarchar);
```

Syntax

```
ST_AsBinary(g1 ST_Geometry)
```

Return Type

ST_Geometry

Example

The code fragment below converts the footprint multipolygons of the **buildingfootprints** table into WKB multipolygons using the **ST_AsBinary()** function. The multipolygons are passed to the application's **draw_polygon()** function for display:

```
/* Create the SQL expression. */
    sprintf(sql_stmt,
            "SELECT ST_AsBinary(zone) "
            "FROM sensitive_areas WHERE "
            "SE_EnvelopesIntersect(zone,ST_PolyFromWKB(?,%d))", srid);

    /* Prepare the SQL statement. */
    SQLPrepare(hstmt, (UCHAR *) sql_stmt, SQL_NTS);

    /* Bind the query shape parameter. */
    pcbvalue1 = query_wkb_len;
    SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
                      SQL_INFX_UDT_LVARCHAR, query_wkb_len, 0,
                      query_wkb_buf, query_wkb_len, &pcbvalue1);

    /* Execute the query. */
    rc = SQLExecute(hstmt);

    /* Assign the results of the query (the buildingfootprint polygons)
     * to the fetched_binary variable. */
    SQLBindCol (hstmt, 1, SQL_C_BINARY,
                fetched_wkb_buf, 10000, &fetched_wkb_len);

    /* Fetch each polygon within the display window and display it. */
    while (1)
    {
        rc = SQLFetch(hstmt);
```

```
    if (rc == SQL_NO_DATA_FOUND)
        break;
    else
        returncode_check(NULL, hstmt, rc, "SQLFetch");

    draw_polygon(fetched_wkb_buf);
}

/* Close the result set cursor */
SQLCloseCursor(hstmt);
```

SE_AsGML()

The **SE_AsGML()** function returns the Geography Markup Language (GML) representation of an ST_Geometry spatial type. Typically, you use the **SE_AsGML()** function to retrieve the GML representation of a spatial primitive from the server and send it to a client, as in:

```
SELECT SE_AsGML(geomcol) FROM mytable
```

The return type of the **SE_AsGML()** function is defined as LVARCHAR. You can write new user-deinfed routines (UDRs) in C or SPL to extend the functionality of the Spatial DataBlade module. You can use the **SE_AsGML()** function to convert an ST_Geometry type to its GML representation.

Syntax

```
SE_AsGML(p ST_Geometry)
```

Return Type

ST_Geometry

Example

In this example, the **SE_AsGML()** function converts the **location** column of the table **mytable** into its GML description.

```
CREATE TABLE mytable (id integer, location ST_Point);

INSERT INTO mytable VALUES(
   1,
   ST_PointFromText('point (10.02 20.01)', 1000)
);

SELECT SE_AsGML(location) FROM mytable WHERE id = 1;

<gml:Point srsName="UNKNOWN">
<gml:coord><gml:X>10.02</gml:X><gml:Y>20.01</gml:Y></gml:coord>
</gml:Point>
```

The **$INFORMIXDIR/extend/spatial.***version***/examples/gml** directory contains more information about how to use an XML parser to validate the results of this function.

ST_AsGML()

The **ST_AsGML()** function returns the Geography Markup Language (GML) representation of an ST_Geometry spatial type that conforms to either the GML2 or GML3 standard. Typically, you use the **ST_AsGML()** function to retrieve the GML representation of a spatial primitive from the server and send it to a client, specifying the GML version to use. For example:

```
SELECT ST_AsGML(geomcol, 3) FROM mytable
```

Syntax

```
ST_AsGML(p ST_Geometry, int Version)
```

`Version` is the GML version that is used to encode the returned geometry. Specify `2` for GML2 and `3` for GML3. The default is `2`.

Return Type

ST_Geometry

Example

The first example returns a geometry from the **mypoints** table as a GML2 representation. The second example returns the geometry as a GML3 representation.

```
SELECT id, ST_AsGML(pt,2) FROM mypoints WHERE id=1

id 1
<gml:Point> <gml:coord><gml:X>-95.7</gml:X>
<gml:Y>38.1</gml:Y></gml:coord></gml:Point>

SELECT id, ST_AsGML(pt,3) AS gml_v32arg, ST_AsGML(pt)
  AS gml_v31arg FROM mypoints WHERE id=1

id 1
gml_v32arg <gml:Point><gml:pos>-95.7 38.1</gml:pos></gml:Point>
gml_v31arg <gml:Point><gml:pos>-95.7 38.1</gml:pos></gml:Point>
```

The **$INFORMIXDIR/extend/spatial.***version***/examples/gml** directory contains more information about how to use an XML parser to validate the results of this function.

SE_AsKML()

The **SE_AsKML()** function returns the Keyhole Markup Language (KML) representation of an ST_Geometry spatial type. Typically, you use the **SE_AsKML()** function to retrieve the KML representation of a spatial primitive from the server and send it to a client, as in:

```
SELECT SE_AsKML(geomcol) FROM mytable
```

The return type of the **SE_AsKML()** function is defined as LVARCHAR. You can write new user-defined routines (UDRs) in C or SPL to extend the functionality of the Spatial DataBlade module.

Syntax

```
SE_AsKML(p ST_Geometry)
```

Return Type

ST_Geometry

Example

In this example, the **SE_AsKML()** function converts the **location** column of the table **mytable** into its KML description.

```
CREATE TABLE mytable (id integer, location ST_Point);

INSERT INTO mytable VALUES(
   1, ST_PointFromText('point (10.02 20.01)', 1000)
);

SELECT id, SE_AsKML(p1) FROM point_t ORDER BY id asc;
id            100_xy
<Point><coordinates>10.02,20.01</coordinates></Point>

CREATE TABLE line_t (id integer, p1 ST_LineString);

INSERT INTO line_t VALUES(
  1,
  ST_LineFromText('linestring (0.0 0.0,0.0 1.0,1.0 0.0,1.0 1.0)',1000)
);
1 row(s) inserted.


INSERT INTO line_t VALUES(
  2,
  ST_LineFromText('linestring z (0.0 0.0 0,0.0 1.0 1,1.0 0.0 1,1.0 1.0 1)',1000)
);
1 row(s) inserted.

INSERT INTO line_t VALUES(
  3,
  ST_LineFromText('linestring m (0.0 0.0 0,0.0 1.0 1,1.0 0.0 1,1.0 1.0 1)',1000)
);
1 row(s) inserted.


SELECT id, SE_AsKML(p1) FROM line_t ORDER BY id ASC;
id            1
<LineString><coordinates>0,0 0,1 1,0 1,1</coordinates></LineString>
id            2
<LineString>
  <coordinates>0,0,0 0,1,1 1,0,1 1,1,1</coordinates>
```

```
</LineString>
id             3
<LineString><coordinates>0,0 0,1 1,0 1,1</coordinates></LineString>
```

ST_AsKML()

ST_AsKML() returns the Keyhole Markup Language (KML) representation of an ST_Geometry spatial type. **ST_AsKML()** can also take an optional parameter that describes KML shape attributes.

Syntax

```
ST_AsKML(p ST_Geometry)
ST_AsKML(p ST_Geometry, attributes lvarchar)
```

The attributes parameter can contain one or more KML shape attributes, in XML format. The following table describes common KML shape attributes.

Table 8-1. KML Shape Attributes

Attribute	Description
<extrude>	A Boolean value that specifies if the geometry is connected to the ground (1) or not (0). To extrude a geometry, the value of the <altitudeMode> attribute must be either **relativeToGround** or **absolute** and the Z coordinate within the <coordinates> element must be greater than zero (0). If this attribute is not specified, it is false (0).
<tessellate>	A Boolean value that specifies if the geometry should follow the terrain (1) or not (0). To enable tessellation for the geometry, the value for the <altitudeMode> attribute must be **clampToGround**. If this attribute is not specified, it is false (0).
<altitudeMode>	Specifies how Z coordinates in the <coordinates> element are interpreted. Possible values are: • **clampToGround** (default) = Indicates to ignore the altitude specification in the geometry and place it at ground level. • **relativeToGround** = Interprets the altitude specification in the geometry and places it at that altitude above the ground. • **absolute** = Interprets the altitude specification in the geometry and places it at that altitude above sea level.

The Spatial DataBlade module does not check whether the attributes are valid. If any of the attributes are not valid, the resulting KML fragment might not be valid when it is received by the application.

Return Type

LVARCHAR

Example

```
EXECUTE FUNCTION ST_AsKML(ST_PointFromText('POINT(-83.54354 34.23425)',4))
```

Output:

```
<Point>
        <coordinates>-83.5435399663,34.2342500677</coordinates>
</Point>
```

In this example, **ST_AsKML()** contains KML shape attributes to specify that the geometry is connected to the ground:

```
EXECUTE FUNCTION ST_AsKML(ST_PointFromText('POINT Z
                          (-83.54523 34.214312 100)',4),
                          '<extrude>1</extrude><altitudeMode>
                          clampToGround</altitudeMode>');
```

Output:

```
<Point>
       <extrude>1</extrude>
       <altitudeMode>clampToGround</altitudeMode>
       <coordinates>-83.545522957,34.2143120335,100</coordinates>
</Point>
```

SE_AsShape()

SE_AsShape() takes a geometry object and returns it in ESRI shapefile format.

The return type of **SE_AsShape()** is defined as ST_Geometry to allow spatial objects greater than 2 kilobytes in size to be retrieved by a client application.

Typically, you use **ST_AsShape()** to retrieve spatial data from the server and send it to a client, as in:

```
SELECT SE_AsShape(geomcol) FROM mytable
```

IBM Informix Dynamic Server automatically casts the output of the **SE_AsShape()** function to the proper data type for transmission to the client.

You can extend the functionality of the IBM Informix Spatial DataBlade Module by writing new user-defined routines (UDRs) in C or SPL. You can use **SE_AsShape()** to convert an ST_Geometry to ESRI shapefile format. If you pass the output of **SE_AsShape()** to another UDR whose function signature requires an LVARCHAR input, you should explicitly cast the return type of **SE_AsShape()** to LVARCHAR, as in:

```
EXECUTE FUNCTION MySpatialFunc(SE_AsShape(geomcol)::lvarchar);
```

Syntax

```
SE_AsShape(g1 ST_Geometry)
```

Return Type

ST_Geometry

Example

The code fragment below illustrates how the **SE_AsShape()** function converts the **zone** polygons of the **sensitive_areas** table into shape polygons. These shape polygons are passed to the application's **draw_polygon()** function for display:

```
/* Create the SQL expression. */
    sprintf(sql_stmt,
            "SELECT SE_AsShape(zone) "
            "FROM sensitive_areas WHERE "
            "SE_EnvelopesIntersect(zone,SE_PolyFromShape(?,%d))", srid);

    /* Prepare the SQL statement. */
    SQLPrepare(hstmt, (UCHAR *)sql_stmt, SQL_NTS);

    /* Bind the query geometry parameter. */
    pcbvalue1 = query_shape_len;
    SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
                      SQL_INFX_UDT_LVARCHAR, query_shape_len, 0,
                      query_shape_buf, query_shape_len, &pcbvalue1);

    /* Execute the query. */
    rc = SQLExecute(hstmt);

    /* Assign the results of the query (the Zone polygons) to the
       fetched_shape_buf variable. */
    SQLBindCol (hstmt, 1, SQL_C_BINARY,
                fetched_shape_buf, 10000, &fetched_shape_len);

    /* Fetch each polygon within the display window and display it. */
    while (SQL_SUCCESS == (rc = SQLFetch(hstmt)))
        draw_polygon(fetched_shape_buf);
```

ST_AsText()

ST_AsText() takes an ST_Geometry object and returns its well-known text representation.

The return type of **ST_AsText()** is defined as ST_Geometry to allow spatial objects greater than 2 kilobytes to be retrieved by a client application.

Typically, you use **ST_AsText()** to retrieve spatial data from the server and send it to a client, as in:

```
SELECT ST_AsText(geomcol) FROM mytable
```

IBM Informix Dynamic Server automatically casts the output of the **ST_AsText()** function to the proper data type for transmission to the client.

You can extend the functionality of the IBM Informix Spatial DataBlade Module by writing new user-defined routines (UDRs) in C or SPL. You can use **ST_AsText()** to convert an ST_Geometry to its well-known text representation. If you pass the output of **ST_AsText()** to another UDR whose function signature requires an LVARCHAR input, you should explicitly cast the return type of **ST_AsText()** to LVARCHAR, as in:

```
EXECUTE FUNCTION MySpatialFunc(ST_AsText(geomcol)::lvarchar)
```

Syntax

```
ST_AsText(g1 ST_Geometry)
```

Return Type

ST_Geometry

Example

The **ST_AsText()** function converts the **hazardous_sites** location point into its text description:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);



INSERT INTO hazardous_sites VALUES(
     102, 'W. H. Kleenare Chemical Repository',
     ST_PointFromText('point (1020.12 324.02)',1000)
);
SELECT site_id, name, ST_AsText(location) Location
   FROM hazardous_sites;


site_id   102
name      W. H. Kleenare Chemical Repository
location  POINT (1020.12 324.02)
```

ST_Boundary()

ST_Boundary() takes a geometry object and returns its combined boundary as a geometry object.

Syntax

```
ST_Boundary(g1 ST_Geometry)
```

Return Type

ST_Geometry

Example

In this example the **boundary_test** table is created with two columns: **geotype** defined as a VARCHAR, and **g1** defined as the superclass ST_Geometry. The INSERT statements that follow insert each one of the subclass geometries. The **ST_Boundary()** function retrieves the boundary of each subclass stored in the **g1** geometry column. Note that the dimension of the resulting geometry is always one less than the input geometry. Points and multipoints always result in a boundary that is an empty geometry, dimension -1. Linestrings and multilinestrings return a multipoint boundary, dimension 0. A polygon or multipolygon always returns a multilinestring boundary, dimension 1:

```
CREATE TABLE boundary_test (geotype varchar(20),
                            g1      ST_Geometry);

INSERT INTO boundary_test VALUES(
   'Point', ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO boundary_test VALUES(
   'Linestring',
   ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO boundary_test VALUES(
   'Polygon',
   ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,19.15
   33.94, 10.02 20.01))',1000)
);

INSERT INTO boundary_test VALUES(
   'Multipoint',
   ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO boundary_test VALUES(
   'Multilinestring',
   ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,11.92 25.64),
   (9.55 23.75,15.36 30.11))',1000)
);

INSERT INTO boundary_test VALUES(
   'Multipolygon',
   ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,25.02
   34.15,19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52 32.87,52.43
   31.90,51.71 21.73)))',1000)
);

SELECT geotype, ST_Boundary(g1)
   FROM boundary_test;


geotype       Point
(expression)  1000 POINT EMPTY
```

```
geotype       Linestring
(expression)  1000 MULTIPOINT (10.02 20.01, 11.92 25.64)

geotype       Polygon
(expression)  1000 MULTILINESTRING ((10.02 20.01, 19.15 33.94, 25.02
   34.15, 11.92 35.64, 10.02 20.01))

geotype       Multipoint
(expression)  1000 POINT EMPTY

geotype       Multilinestring
(expression)  1000 MULTIPOINT (9.55 23.75, 10.02 20.01, 11.92 25.64, 15.36 30.1
              1)

geotype       Multipolygon
(expression)  1000 MULTILINESTRING ((10.02 20.01, 19.15 33.94, 25.02 34.15,
   11.92 35.64, 10.02 20.01),(51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43
   31.9, 51.71 21.73))
```

SE_BoundingBox()

SE_BoundingBox() returns a polygon which represents the spatial extent (the minimum and maximum X and Y values) of all the geometries in a spatial column of a table.

Important: You must have an R-tree index on the spatial column.

Syntax

```
SE_BoundingBox (tablename varchar(128),
                columnname varchar(128))
```

Return Type

ST_Polygon

Example

The **buildingfootprints** table is created with the following statement. The **building_id** column uniquely identifies the buildings, the **lot_id** identifies the building's lot, and the footprint multipolygon stores the building's geometry:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

After the table is populated, create an R-tree index on the **footprint** column:

```
CREATE INDEX footprint_ix
   ON buildingfootprints (footprint ST_Geometry_ops)
   USING RTREE;
```

Use the **SE_BoundingBox()** function to obtain a polygon that defines the spatial extent of all the polygons in the table:

```
EXECUTE FUNCTION SE_BoundingBox ('buildingfootprints', 'footprint');


(expression)  1000 POLYGON ((7 22, 38 22, 38 55, 7 55, 7 22))
```

Obtaining the Spatial Extent of a Functional Index

You cannot use the **SE_BoundingBox()** function on a functional R-tree index. Instead, you must use the **rtreeRootBB()** function. The **rtreeRootBB()** function takes the index name and type name as arguments and returns the well-known text representation of an ST_Polygon, as shown in the following example:

```
CREATE TABLE xytab (x float, y float, srid int);

INSERT INTO xytab VALUES (1, 2, 0);
INSERT INTO xytab VALUES (3, 4, 0);

CREATE INDEX point_idx ON xytab
   (ST_Point(x,y,srid) ST_Geometry_ops) USING RTREE;

EXECUTE FUNCTION rtreeRootBB('point_idx', 'st_point');

(expression)  0 POLYGON ((1 2, 3 2, 3 4, 1 4, 1 2))
```

See Also

"ST_Envelope()" on page 8-44

ST_Buffer()

ST_Buffer() takes a geometry object and distance and returns a geometry object that is the buffer surrounding the source object.

Syntax

```
ST_Buffer(g1 ST_Geometry, distance double_precision)
```

Return Type

ST_Geometry

Example

The county supervisor needs a list of hazardous sites whose five-mile radius overlaps sensitive areas such as schools, hospitals, and nursing homes. The sensitive areas are stored in the table **sensitive_areas** that is created with the CREATE TABLE statement shown below. The **zone** column, defined as a **ST_Polygon**, stores the outline of each of the sensitive areas:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The hazardous sites are stored in the **hazardous_sites** table created below. The **location** column, defined as a point, stores the geographic center of each hazardous site:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The **sensitive_areas** and **hazardous_sites** tables are joined by the **ST_Overlaps()** function, which returns `t` (TRUE) for all **sensitive_areas** rows whose **zone** polygons overlap the buffered five-mile radius of the **hazardous_sites** location point.

```
SELECT sa.name, hs.name
   FROM sensitive_areas sa, hazardous_sites hs
   WHERE ST_Overlaps(sa.zone, ST_Buffer(hs.location,26400));


name  Johnson County Hospital
name  Landmark Industrial

name  Summerhill Elementary School
name  Landmark Industrial
```

Figure 8-2 shows that some of the sensitive areas in this administrative district lie within the five-mile buffer radius of the hazardous site locations. It is clear that both buffers intersect the hospital and one intersects the school. The nursing home lies safely outside both radii.

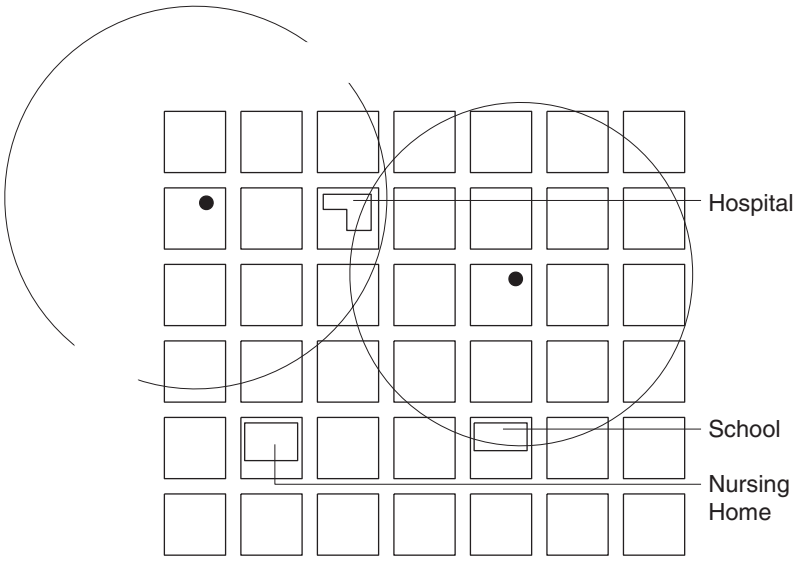


Figure 8-2. Sensitive Areas

ST_Centroid()

ST_Centroid() takes a polygon or multipolygon and returns its geometric center as a point.

Syntax

```
ST_Centroid(pl1 ST_Polygon)
ST_Centroid(mpl1 ST_MultiPolygon)
```

Return Type

ST_Point

Example

The city GIS technician wants to display the building footprint multipolygons as single points in a building density graphic.

The building footprints are stored in the **buildingfootprints** table that was created with the following CREATE TABLE statement:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

The **ST_Centroid()** function returns the centroid of each building footprint multipolygon:

```
SELECT building_id, ST_Centroid(footprint) Centroid
   FROM buildingfootprints;


building_id  506
centroid     1000 POINT (12.5 49.5)

building_id  543
centroid     1000 POINT (32 51.5)

building_id  1208
centroid     1000 POINT (12.5 30.5)

building_id  178
centroid     1000 POINT (32 28.5)
```

ST_Contains()

ST_Contains() takes two geometry objects and returns `t` (TRUE) if the first object completely contains the second; otherwise, it returns `f` (FALSE).

Syntax

```
ST_Contains(g1 geometry, g2 geometry)
```

Return Type

BOOLEAN

Example

In the example below, two tables are created: **buildingfootprints** contains a city's building footprints, and **lots** contains its lots. The city engineer wants to ensure that all building footprints are completely inside their lots.

In both tables, the ST_MultiPolygon data type stores the geometry of the building footprints and the lots. The database designer selected multipolygons for both features because lots can be separated by natural features such as a river, and building footprints can comprise several buildings:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);

CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The city engineer first selects the buildings that are not completely contained within one lot:

```
SELECT building_id
   FROM buildingfootprints, lots
   WHERE NOT ST_Contains(lot,footprint);
```

The city engineer realizes that although the first query provides a list of all building IDs that have footprints outside a lot polygon, it will not tell if the rest have the correct **lot_id** assigned to them. This second query performs a data integrity check on the **lot_id** column of the **buildingfootprints** table:

```
SELECT bf.building_id, bf.lot_id, lots.lot_id
   FROM buildingfootprints bf, lots
   WHERE NOT ST_Contains(lot,footprint)
   AND lots.lot_id <> bf.lot_id;
```

In Figure 8-3, the building footprints are labeled with their building IDs and lie inside their lot lines. The lot lines are illustrated with dotted lines and, although not shown, extend to the street centerline to completely encompass the lot lines and the building footprints within them

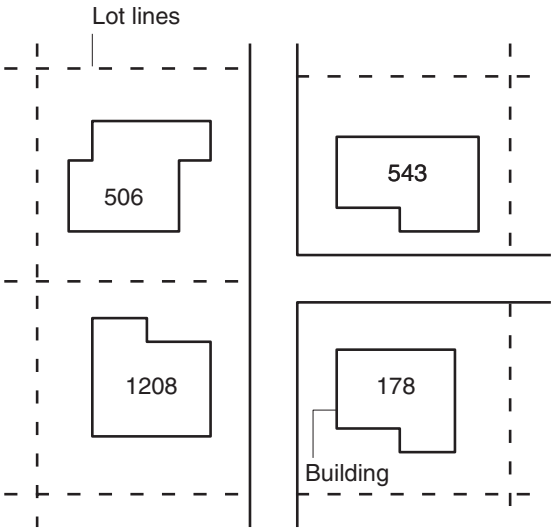


Figure 8-3. Building Footprints and Lot Lines.

ST_ConvexHull()

ST_ConvexHull() returns the convex hull of a geometry object.

Syntax

```
ST_ConvexHull (g1 ST_ Geometry)
```

Return Type

ST_Geometry

Example

The example creates the **convexhull_test** table that has two columns: **geotype** and **g1**. The **geotype** column is of VARCHAR(20) type and holds the name of the geometry subclass that is stored in **g1**, an ST_Geometry column:

```
CREATE TABLE convexhull_test (geotype varchar(20),
                              g1      ST_Geometry);
```

The following INSERT statements insert several geometry subclasses into the **convexhull_test** table:

```
INSERT INTO convexhull_test VALUES(
   'Point',
   ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO convexhull_test VALUES(
   'Linestring',
   ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO convexhull_test VALUES(
   'Polygon',
   ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
19.15 33.94,10.02 20.01))',1000)
);

INSERT INTO convexhull_test VALUES(
   'MultiPoint',
   ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO convexhull_test VALUES(
   'MultiLineString',
   ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,11.92
25.64),(9.55 23.75,15.36 30.11))',1000)
);

INSERT INTO convexhull_test VALUES(
   'Multipolygon',
   ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52
32.87,52.43 31.90,51.71 21.73)))',1000)
);
```

The SELECT statement lists the subclass name stored in the **geotype** column and the convex hull:

```
SELECT geotype, ST_ConvexHull(g1) convexhull
   FROM convexhull_test;
```

ST_CoordDim()

ST_CoordDim() returns the coordinate dimensions of the ST_Geometry value. For example, a point with a Z coordinate has three dimensions and a point with a Z coordinate and measures has four dimensions.

Syntax

```
ST_CoordDim(g ST_Geometry)
```

Return Type

INTEGER

Example

The **coorddim_test** table is created with the columns **geotype** and **g1**. The **geotype** column stores the name of the geometry subclass stored in the **g1** ST_Geometry column:

```
CREATE TABLE coorddim_test (geotype varchar(20),
                            g1      ST_Geometry);
```

The INSERT statements insert a sample subclass into the **coorddim_test** table:

```
INSERT INTO coorddim_test VALUES(
   'Point',
   ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO coorddim_test VALUES(
   'Point',
   ST_PointFromText('point z (10.02 20.01 3.21)',1000)
);

INSERT INTO coorddim_test VALUES(
   'LineString',
   ST_LineFromText('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)',1000)
);

INSERT INTO coorddim_test VALUES(
   'LineString',
   ST_LineFromText('linestring m (10.02 20.01 1.23, 10.32 23.98 4.56, 11.92
25.64 7.89)',1000)
);

INSERT INTO coorddim_test VALUES(
   'Polygon',
   ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94, 10.02 20.01))',1000)
);

INSERT INTO coorddim_test VALUES(
   'Polygon',
   ST_PolyFromText('polygon zm ((10.02 20.01 9.87 1.23, 11.92
35.64 7.65 2.34, 25.02 34.15 6.54 3.45, 19.15 33.94 5.43 4.56,
10.02 20.01 9.87 1.23))',1000)
);
```

The SELECT statement lists the subclass name stored in the **geotype** column with the dimension of that geometry:

```
SELECT geotype, ST_CoordDim(g1) coord_dimension
   FROM coorddim_test;

geotype              coord_dimension
```

```
Point                          2
Point                          3
LineString                     2
LineString                     3
Polygon                        2
Polygon                        4

6 row(s) retrieved.
```

SE_CreateSRID()

SE_CreateSRID() is a utility function that, given the X and Y extents of a spatial data set, computes the false origin and system units (as described in "Selecting a False Origin and System Units" on page 1-6) and creates a new entry in the **spatial_references** table. Appropriate offsets and scale factors for typical Z values and M values are also provided.

The **spatial_references** table holds data about spatial reference systems. A spatial reference system is a description of a coordinate system for a set of geometric objects; it gives meaning to the X and Y values that are stored in the database.

You need to specify an SRID of a spatial reference system when you load spatial data into a database, because the IBM Informix Spatial DataBlade Module translates and scales each floating point coordinate of the geometry into a 53-bit positive integer prior to storage.

When you register the IBM Informix Spatial DataBlade Module in a database, the **spatial_references** table is pre-loaded with an entry for SRID=0, which is suitable for worldwide spatial data in latitude/longitude format.

If SRID=0 is not suitable for your data, you can use **SE_CreateSRID()** to assist you in creating a more appropriate **spatial_references** table entry. However, because the **SE_CreateSRID()** function only considers the X and Y extents of your data, it may not create sufficiently refined parameters to describe the spatial reference system you need. For more information about spatial reference systems and how to compute false origins and scale factors, review "The spatial_references Table" on page 1-4.

Syntax

```
SE_CreateSrid (xmin float, ymin float,
               xmax float, ymax float,
               description varchar(64))
```

The *description* parameter is provided to encourage you to document any SRIDs that you create with this function.

Return Type

Returns the SRID of a newly created **spatial_references** table entry as an integer

If the **spatial_references** table already contains an identical entry, its SRID is returned.

Example

To create an entry in the **spatial_references** table that is suitable for spatial data in New Zealand:

```
EXECUTE FUNCTION SE_CreateSrid (166, -48, 180, -34,
                                "New Zealand: lat/lon coords");

(expression)

        1001
```

The following query shows the resulting **spatial_references** table entry:

```
SELECT * FROM spatial_references WHERE srid = 1001;


srid         1001
description  New Zealand: lat/lon coords
auth_name
auth_srid
falsex       164.6000000000
falsey       -49.4000000000
xyunits      127826407.5600
falsez       -1000.00000000
zunits       1000.000000000
falsem       -1000.00000000
munits       1000.000000000
srtext       UNKNOWN
```

See Also

"SE_CreateSrtext()" on page 8-33

SE_CreateSrtext()

SE_CreateSrtext() returns the OGC well-known text representation of a spatial reference system, given the ESRI Projection Engine ID number for a coordinate system. These ID numbers can be found in the file **pedef.h**, which is included with this DataBlade module in the directory **$INFORMIXDIR/extend/spatial.version/include**.

Syntax

```
SE_CreateSrtext (factory_id integer)
```

Return Type

LVARCHAR

Example

To obtain the spatial reference system text for the 1983 North American Datum:

```
EXECUTE FUNCTION SE_CreateSrtext(4269);


(expression)  GEOGCS["GCS_North_American_1983",DATUM["D_North_American_1983",SP
              HEROID["GRS_1980",6378137,298.257222101]],PRIMEM["Greenwich",0],U
              NIT["Degree",0.0174532925199432955]]
```

Tip: You can transfer the output of **SE_CreateSrtext()** directly into the **spatial_references** table with the following *SQL* statement:

```
UPDATE spatial_references
   SET srtext = SE_CreateSrtext(4269) WHERE srid = 1000;
```

ST_Crosses()

ST_Crosses() takes two geometry objects and returns `t` (TRUE) if their intersection results in an ST_Geometry object whose dimension is one less than the maximum dimension of the source objects. The intersection object must contain points that are interior to both source geometries, and the intersection object must not be equal to either of the source objects. Otherwise, it returns `f` (FALSE).

Syntax

```
ST_Crosses(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The county government is considering a new regulation stating that all hazardous waste storage facilities may not be within five miles of any waterway. The county GIS manager has an accurate representation of rivers and streams stored as multilinestrings in the **waterways** table, but has only a single point location for each of the hazardous waste storage facilities:

```
CREATE TABLE waterways (id     integer,
                        name   varchar(128),
                        water  ST_MultiLineString);

CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

To determine if it is necessary to alert the county supervisor to any existing facilities that would violate the proposed regulation, the GIS manager has to buffer the **hazardous_sites** locations to see if any rivers or streams cross the buffer polygons. The **ST_Crosses()** function compares the buffered **hazardous_sites** with **waterways**, returning only those records where the waterway crosses over the county's proposed regulated radius:

```
SELECT ww.name waterway, hs.name hazardous_site
   FROM waterways ww, hazardous_sites hs
   WHERE ST_Crosses(ST_Buffer(hs.location,(5 * 5280)),ww.water);


waterway        Fedders creek
hazardous_site  Landmark Industrial
```

Figure 8-4 shows that the five-mile buffered radius of the hazardous waste sites crosses the stream network that runs through the county's administrative district. Because the stream network is defined as an ST_MultiLineString, all linestring segments that are part of those segments that cross the radius are included in the result set

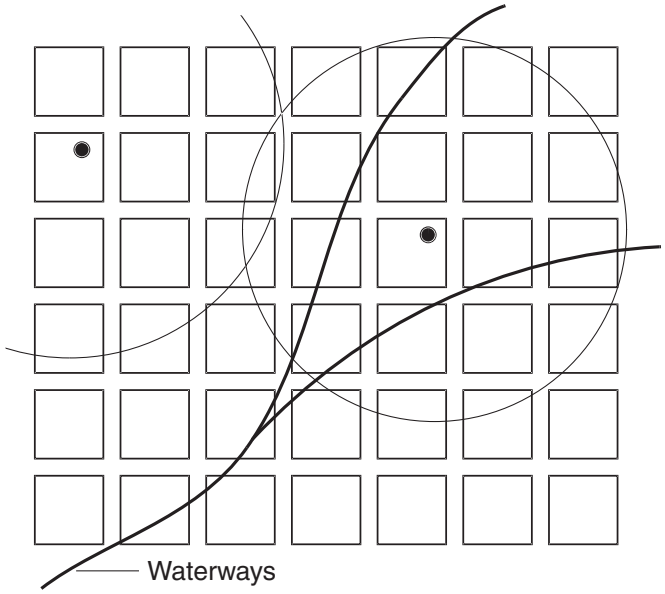


Figure 8-4. Hazardous Waste Sites and the Stream Network.

ST_Difference()

ST_Difference() takes two geometry objects and returns a geometry object that is the difference of the source objects.

Syntax

```
ST_Difference(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

ST_Geometry

Example

The city engineer needs to know the total city lot area not covered by buildings. In fact, the engineer wants the sum of the lot area after the building area has been removed:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);

CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The city engineer equijoins the **buildingfootprints** and **lots** table on the **lot_id** column and takes the sum of the area of the difference of the lots less the building footprints:

```
SELECT SUM(ST_Area(ST_Difference(lot,footprint)::ST_MultiPolygon))
   FROM buildingfootprints bf, lots
   WHERE bf.lot_id = lots.lot_id;
```

ST_Dimension()

ST_Dimension() returns the dimension of a geometry object. A geometry can have one of three dimensions:

- **0.** The geometry has neither length nor area.
- **1.** The geometry has length.
- **2.** The geometry has area.

Syntax

```
ST_Dimension(g1 ST_Geometry)
```

Return Type

INTEGER

Example

The **dimension_test** table is created with the columns **geotype** and **g1**. The **geotype** column stores the name of the subclass stored in the **g1** ST_Geometry column:

```
CREATE TABLE dimension_test (geotype varchar(20),
                             g1      ST_Geometry);
```

The following INSERT statements insert a sample subclass into the **dimension_test** table:

```
INSERT INTO dimension_test VALUES(
   'Point',
   ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO dimension_test VALUES(
   'Linestring',
   ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO dimension_test VALUES(
   'Polygon',
   ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
19.15 33.94,10.02 20.01))',1000)
);

INSERT INTO dimension_test VALUES(
   'Multipoint',
   ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO dimension_test VALUES(
   'Multilinestring',
   ST_MLineFromText('multilinestring ((10.02 20.01,10.32
23.98,11.92 25.64),(9.55 23.75,15.36 30.11))',1000)
);

INSERT INTO dimension_test VALUES(
   'Multipolygon',
   ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52
32.87,52.43 31.90,51.71 21.73)))',1000)
);
```

The SELECT statement lists the subclass name stored in the **geotype** column with the dimension of that geotype:

```
SELECT geotype, ST_Dimension(g1) Dimension
   FROM dimension_test;

geotype                dimension

Point                          0
Linestring                     1
Polygon                        2
Multipoint                     0
Multilinestring                1
Multipolygon                   2
```

ST_Disjoint()

ST_Disjoint() takes two geometries and returns `t` (TRUE) if the two geometries are completely non-intersecting; otherwise, it returns `f` (FALSE).

Syntax

```
ST_Disjoint(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

An insurance company wants to assess the insurance coverage for the town's hospital, nursing homes, and schools. Part of this process includes determining the threat the hazardous waste sites pose to each institution. At this time, the insurance company wants to consider only those institutions that are not at risk of contamination. The GIS consultant hired by the insurance company has been commissioned to locate all institutions that are outside a five-mile radius of a hazardous waste storage facility.

The **sensitive_areas** table contains several columns that describe the threatened institutions in addition to the **zone** column, which stores the institutions' polygon geometries:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The **hazardous_sites** table stores the identity of the sites in the **site_id** and **name** columns, while the actual geographic location of each site is stored in the **location** point column:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The SELECT statement lists the names of all sensitive areas that are outside the five-mile radius of a hazardous waste site:

```
SELECT sa.name
   FROM sensitive_areas sa, hazardous_sites hs
   WHERE ST_Disjoint(ST_Buffer(hs.location,(5 * 5280)), sa.zone);
```

You could also use the **ST_Intersects()** function to perform this query, because **ST_Intersects()** and **ST_Disjoint()** return the opposite results:

```
SELECT sa.name
   FROM sensitive_areas sa, hazardous_sites hs
   WHERE NOT ST_Intersects(ST_Buffer(hs.location,(5 * 5280)), sa.zone);
```

Figure 8-5 shows that the nursing home is the only sensitive area for which the **ST_Disjoint()** function will return `t` (TRUE) when comparing sensitive sites to the five-mile radius of the hazardous waste sites. The **ST_Disjoint()** function returns `t` whenever two geometries do not intersect in any way.

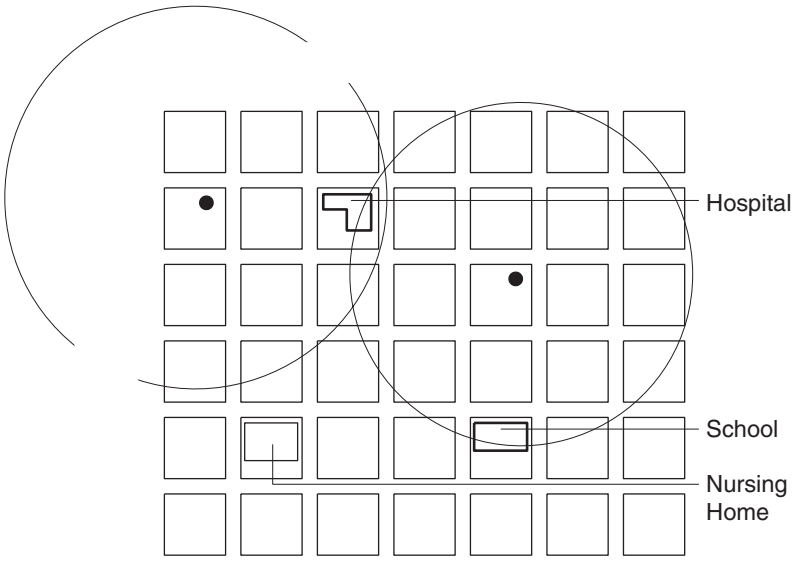


Figure 8-5. Sensitive Sites and Hazardous Waste Sites

SE_Dissolve()

SE_Dissolve() is an aggregate function that computes the union of geometries of the same dimension (if there is just one geometry that satisfies your query, it is returned unaltered).

Syntax

```
SE_Dissolve (g1 ST_Geometry)
```

Return Type

ST_Geometry

Example

The following example creates a single multipolygon from the individual hexagons inserted into the **honeycomb** table:

```
CREATE TABLE honeycomb (cell_id int, hex_cell ST_Polygon);

INSERT INTO honeycomb
   VALUES (1, '0 polygon((5 10,7 7,10 7,12 10,10 13,7 13,5 10))');
INSERT INTO honeycomb
   VALUES (2, '0 polygon((12 4,15 4,17 7,15 10,12 10,10 7,12 4))');
INSERT INTO honeycomb
   VALUES (3, '0 polygon((17 7,20 7,22 10,20 13,17 13,15 10,17 7))');
INSERT INTO honeycomb
   VALUES (4, '0 polygon((15 10,17 13,15 16,12 16,10 13,12 10,15 10))');


SELECT SE_Dissolve(hex_cell) FROM honeycomb;

se_dissolve  0 MULTIPOLYGON (((5 10, 7 7, 10 7, 12 4, 15 4, 17 7, 20 7, 22
10, 20 13, 17 13, 15 16, 12 16, 10 13, 7 13, 5 10)))
```

ST_Distance()

ST_Distance() returns the shortest distance separating two geometries.

Syntax

```
ST_Distance(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

DOUBLE PRECISION

Example

The city engineer needs a list of all buildings within one foot of any lot line.

The **building_id** column of the **buildingfootprints** table uniquely identifies each building. The **lot_id** column identifies the lot each building belongs to. The **footprints** multipolygon stores the geometry of each building's footprint:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

The **lots** table stores the **lot_id** that uniquely identifies each lot and the **lot** ST_MultiPolygon that contains the lot geometry:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The following query returns a list of building IDs that are within one foot of their lot lines. The **ST_Distance()** function performs a spatial join on the **footprints** and **lot** ST_MultiPolygon columns. However, the equijoin between **bf.lot_id** and **lots.lot_id** ensures that only the ST_MultiPolygons belonging to the same lot are compared by the **ST_Distance()** function:

```
SELECT bf.building_id
   FROM buildingfootprints bf, lots
   WHERE bf.lot_id = lots.lot_id
   AND ST_Distance(footprint,lot) <= 1.0;
```

ST_EndPoint()

ST_EndPoint() returns the last point of a linestring.

Syntax

```
ST_EndPoint(ln1 ST_LineString)
```

Return Type

ST_Point

Example

The **endpoint_test** table stores the **gid** INTEGER column, which uniquely identifies each row and the **ln1** ST_LineString column that stores linestrings:

```
CREATE TABLE endpoint_test (gid integer,
                            ln1 ST_LineString);
```

The following INSERT statements insert linestrings into the **endpoint_test** table. The first linestring does not have Z coordinates or measures, while the second one does:

```
INSERT INTO endpoint_test VALUES(
   1,
   ST_LineFromText('linestring (10.02 20.01,23.73 21.92,30.10 40.23)',1000)
);

INSERT INTO endpoint_test VALUES(
   2,
   ST_LineFromText('linestring zm (10.02 20.01 5.0 7.0,23.73 21.92
6.5 7.1,30.10 40.23 6.9 7.2)',1000)
);
```

The following query lists the **gid** column with the output of the **ST_EndPoint()** function. The **ST_EndPoint()** function generates an ST_Point geometry:

```
SELECT gid, ST_EndPoint(ln1) Endpoint
   FROM endpoint_test;


gid       1
endpoint  1000 POINT (30.1 40.23)

gid       2
endpoint  1000 POINT ZM (30.1 40.23 6.9 7.2)
```

See Also

"ST_StartPoint()" on page 8-142

ST_Envelope()

ST_Envelope() returns the bounding box of a geometry object. This is usually a rectangle, but the envelope of a point is the point itself, and the envelope of a horizontal or vertical linestring is a linestring represented by the endpoints of the source geometry.

Syntax

```
ST_Envelope(g1 ST_Geometry)
```

Return Type

ST_Geometry

Example

The **geotype** column of the **envelope_test** table stores the name of the geometry subclass stored in the **g1 ST_Geometry** column:

```
CREATE TABLE envelope_test (geotype varchar(20),
                            g1      ST_Geometry);
```

The following INSERT statements insert each geometry subclass into the **envelope_test** table:

```
INSERT INTO envelope_test VALUES(
   'Point',
   ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO envelope_test VALUES(
   'Linestring',
   ST_LineFromText('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 1000)
);

INSERT INTO envelope_test VALUES(
   'Linestring',
   ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)', 1000)
);

INSERT INTO envelope_test VALUES(
   'Polygon',
   ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94, 10.02 20.01))',1000)
);

INSERT INTO envelope_test VALUES(
   'Multipoint',
   ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)', 1000)
);

INSERT INTO envelope_test VALUES(
   'Multilinestring',
   ST_MLineFromText('multilinestring ((10.01 20.01,20.01
20.01,30.01 20.01), (30.01 20.01,40.01 20.01,50.01 20.01))',1000)
);

INSERT INTO envelope_test VALUES(
   'Multilinestring',
   ST_MLineFromText('multilinestring ((10.02 20.01,10.32
23.98,11.92 25.64),(9.55 23.75,15.36 30.11))',1000)
);
```

```
INSERT INTO envelope_test VALUES(
   'Multipolygon',
   ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52
32.87,52.43 31.90,51.71 21.73)))',1000)
);
```

The following query lists the subclass name and its envelope. The **ST_Envelope()** function returns a point, a linestring, or a polygon:

```
SELECT geotype, ST_Envelope(g1) Envelope
   FROM envelope_test;


geotype   Point
envelope  1000 POINT (10.02 20.01)

geotype   Linestring
envelope  1000 LINESTRING (10.01 20.01, 10.01 40.01)

geotype   Linestring
envelope  1000 POLYGON ((10.02 20.01, 11.92 20.01, 11.92 25.64, 10.02 25.64, 10.02 20.01))

geotype   Polygon
envelope  1000 POLYGON ((10.02 20.01, 25.02 20.01, 25.02 35.64, 10.02 35.64, 10.02 20.01))

geotype   Multipoint
envelope  1000 POLYGON ((10.02 20.01, 11.92 20.01, 11.92 25.64, 10.02 25.64, 10.02 20.01))

geotype   Multilinestring
envelope  1000 LINESTRING (10.01 20.01, 50.01 20.01)

geotype   Multilinestring
envelope  1000 POLYGON ((9.55 20.01, 15.36 20.01, 15.36 30.11, 9.55 30.11, 9.55 20.01))

geotype   Multipolygon
envelope  1000 POLYGON ((10.02 20.01, 73.36 20.01, 73.36 35.64, 10.02 35.64, 10.02 20.01))
```

See Also

"SE_BoundingBox()" on page 8-22

ST_EnvelopeAsGML()

ST_EnvelopeAsGML() takes a geometry value returned by ST_Envelope and generates a GML3 Envelope element.

Syntax

```
ST_EnvelopeAsGML(p ST_Geometry)
```

Return Type

ST_Geometry

Example

```
ST_EnvelopeFromGML(ST_Envelope(ST_LineFromText('LINESTRING(1 2, 3 4)', 1003)))
```

Output:

```
<gml:Envelope srsName="EPSG:1234">
    <gml:lowerCorner> 1 2</gml:lowerCorner>
    <gml:upperCorner> 3 4 </gml:upperCorner>
</gml:Envelope>
```

SE_EnvelopeAsKML()

SE_EnvelopeAsKML() takes a geometry value returned by ST_Envelope and returns it as a KML LatLonBox.

Syntax

```
SE_EnvelopeAsKML(p ST_Geometry)
```

Return Type

ST_Geometry

Example

```
EXECUTE FUNCTION SE_EnvelopeAsKML(ST_PolyFromText('POLYGON
                  ((-124.21160602 25.8373769872,
                  -67.1589579416 25.8373769872,
                  -67.1589579416 49.384359066,
                  -124.21160602 49.384359066,
                  -124.21160602 25.8373769872))',3));
```

Output:

```
<LatLonBox>
           <north>49.384359066</north>
           <south>25.8373769872</south>
           <east>-67.1589579416</east>
           <west>-124.21160602</west>
</LatLonBox>
```

ST_EnvelopeFromGML()

ST_EnvelopeFromGML() takes a GML2 or GML3 string representation of an envelope and an optional spatial reference ID and returns a geometry object. If the **srsName** attribute is specified in the GML string, then a corresponding entry in the **spatial_references** table must exist unless it is specified as UNKNOWN or DEFAULT.

Syntax

```
ST_EnvelopeFromGML(gml_string lvarchar)
ST_EnvelopeFromGML(gml_string lvarchar, SRID integer)
```

Return Type

A four sided ST_Polygon representing the envelope.

Example

```
ST_EnvelopeFromGML('<gml:Envelope>
    <gml:lowerCorner> -180.0 -90.0</gml:lowerCorner>
    <gml:upperCorner> 180.0 90.0 </gml:upperCorner>
</gml:Envelope>', 1003)
```

SE_EnvelopeFromKML()

SE_EnvelopeFromKML() takes a KML LatLonBox or LatLonAltBox and an optional spatial reference ID and returns a polygon. The LatLonBox and LatLonAltBox contain four coordinates: north, south, east, and west, that are used to form the traditional pair of SW, NE coordinates usually found with bounding boxes. LatLonAltBox also contains the elements minAltitude and maxAltitude, and while those will be accepted as valid tags in the KML fragment, they are not used to form a Z-polygon. Only 2-D polygons are returned.

Syntax

```
SE_EnvelopeFromKML(kml_string lvarchar)
SE_EnvelopeFromKML(kml_string lvarchar, SRID integer)
```

Return Type

A four sided ST_Polygon representing the envelope.

Example

In this example, the KML LatLonBox includes four coordinates:

```
EXECUTE FUNCTION SE_EnvelopeFromKML('<LatLonBox><north>34.54356</north>
                                    <south>33.543634</south><east>-83.21454</east>
                                    <west>-86.432536</west>',4);
```

Output:

```
4 POLYGON ((-86.3253600195 33.5436340112, -83.2145400212 33.543630112,
            -83.2145400212 34.5435600828, -86.3253600195 34.5435600828,
            -86.3253600195 33.5436340112))
```

In this example, the KML LatLonAltBox includes the four coordinates as well as the minAltitude, maxAltitude, and altitudeMode attributes:

```
EXECUTE FUNCTION SE_EnvelopeFromKML('<LatLonAltBox><north>45.0</north>
                <south>42.0</south><east>-80.0</east><west>-82.0</west>
                <minAltitude>0</minAltitude><maxAltitude>0</maxAltitude>
                <altitudeMode>clampToGround</altitudeMode>',4);
```

However, the output only includes the four coordinates:

```
4 POLYGON((-82.0 42.0, -80.0 42.0, -80.0 45.0, -82.0 45.0, -82.0 42.0))
```

SE_EnvelopesIntersect()

SE_EnvelopesIntersect() returns `t` (TRUE) if the envelopes of two geometries intersect; otherwise, it returns `f` (FALSE).

Syntax

```
SE_EnvelopesIntersect(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The **get_window()** function retrieves the display window coordinates from the application. The window parameter is actually a polygon shape structure containing a string of coordinates that represents the display polygon. The **SE_PolygonFromShape()** function converts the display window shape into a polygon that the **SE_EnvelopesIntersect()** function uses as its intersection envelope. All **sensitive_areas zone** polygons that intersect the interior or boundary of the display window are returned. Each polygon is fetched from the result set and passed to the **draw_polygon()** function:

```
/* Get the display window coordinates as a polygon shape. */
    get_window(&query_shape_buf, &query_shape_len);

    /* Create the SQL expression. The envelopesintersect function limits
     * the result set to only those zone polygons that intersect the
     * envelope of the display window. */
    sprintf(sql_stmt,
            "select SE_AsShape(zone) ",
            "from sensitive_areas where ",
            "SE_EnvelopesIntersect(zone,SE_PolyFromShape(?,1))");

    /* Prepare the SQL statement. */
    SQLPrepare(hstmt, (UCHAR *)sql_stmt, SQL_NTS);

    /* Bind the query geometry parameter. */
    pcbvalue1 = query_shape_len;
    SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
                      SQL_INFX_UDT_LVARCHAR, query_shape_len, 0,
                      query_shape_buf, query_shape_len, &pcbvalue1);

    /* Execute the query. */
    rc = SQLExecute(hstmt);

    /* Assign the results of the query (the Zone polygons) to the
       fetched_shape_buf variable. */
    SQLBindCol (hstmt, 1, SQL_C_BINARY, fetched_shape_buf, 100000,
                &fetched_shape_len);

    /* Fetch each polygon within the display window and display it. */
    while (SQL_SUCCESS == (rc = SQLFetch(hstmt)))
        draw_polygon(fetched_shape_buf);
```

ST_Equals()

ST_Equals() compares two geometries and returns `t` (TRUE) if the geometries are spatially equal; otherwise, it returns `f` (FALSE). Using the **ST_Equals()** function is functionally equivalent to using **ST_IsEmpty**(**ST_SymDifference**(*a*,*b*)).

The **ST_Equals()** function is computationally intensive and you should consider whether you could use the **Equal()** function instead, which does a byte-by-byte comparison of two objects. The **Equal()** function is a system function used by the IBM Informix Spatial DataBlade Module; it is called in SQL statements when you use the = operator, as shown in the second SELECT statement in the following example.

To illustrate the difference between **ST_Equals()** and **Equal()**, consider the following example:

```
CREATE TABLE equal_test (id   integer,
                         line ST_LineString);

INSERT INTO equal_test VALUES
   (1, ST_LineFromText('linestring(10 10, 20 20)', 1000));

INSERT INTO equal_test VALUES
   (2, ST_LineFromText('linestring(20 20, 10 10)', 1000));
```

The following query returns both rows, because **ST_Equals()** determines that both linestrings are spatially equivalent:

```
SELECT id FROM equal_test
   WHERE ST_Equals (line, ST_LineFromText('linestring(10 10, 20 20)', 1000));

        id

         1
         2
```

The following query only returns the first row, because **Equals()** only performs a memory comparison of the linestrings:

```
SELECT id FROM equal_test
   WHERE line = ST_LineFromText('linestring(10 10, 20 20)', 1000);

        id

         1
```

Syntax

```
ST_Equals(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The city GIS technician suspects that some of the data in the **buildingfootprints** table was somehow duplicated. To alleviate concern, the technician queries the table to determine if any of the footprints multipolygons are equal.

The **buildingfootprints** table is created with the following statement. The **building_id** column uniquely identifies the buildings, the **lot_id** identifies the building's lot, and the **footprint** multipolygon stores the building's geometry:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

The **buildingfootprints** table is spatially joined to itself by the **ST_Equals()** function, which returns 1 whenever it finds two multipolygons that are equal. The **bf1.building_id <> bf2.building_id** condition eliminates the comparison of a geometry to itself:

```
SELECT bf1.building_id, bf2.building_id
   FROM buildingfootprints bf1, buildingfootprints bf2
   WHERE ST_Equals(bf1.footprint,bf2.footprint)
   AND bf1.building_id <> bf2.building_id;
```

ST_ExteriorRing()

ST_ExteriorRing() returns the exterior ring of a polygon as a linestring.

Syntax

```
ST_ExteriorRing(pl1 ST_Polygon)
```

Return Type

ST_LineString

Example

An ornithologist studying the bird population on several South Sea islands knows that the feeding zone of the bird species of interest is restricted to the shoreline. As part of the calculation of the island's carrying capacity, the ornithologist requires the islands' perimeters. Some of the islands are so large, they have several ponds on them. However, the shorelines of the ponds are inhabited exclusively by another more aggressive bird species. Therefore, the ornithologist requires the perimeter of the exterior ring only of the islands.

The **ID** and **name** columns of the **islands** table identifies each island, while the **land** polygon column stores the island's geometry:

```
CREATE TABLE islands (id    integer,
                      name  varchar(32),
                      land  ST_Polygon);
```

The **ST_ExteriorRing()** function extracts the exterior ring of each island polygon as a linestring. The length of the linestring is calculated by the **ST_Length()** function. The linestring lengths are summarized by the SUM operator:

```
SELECT SUM(ST_Length(ST_ExteriorRing(land)))
  FROM islands;
```

As shown in Figure 8-6, the exterior rings of the islands represent the ecological interface each island shares with the sea. Some of the islands have lakes, which are represented by the interior rings of the polygons.

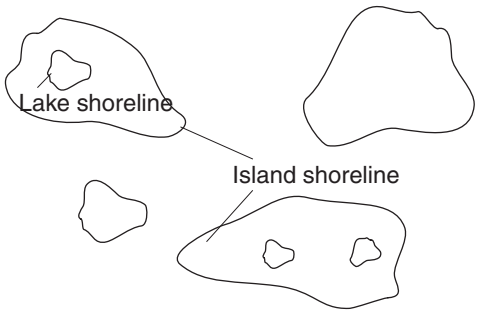


Figure 8-6. Islands and Lakes

SE_Generalize()

The **SE_Generalize()** function reduces the number of vertices in an ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon while preserving the general character of the geometric shape.

This function uses the Douglas-Peucker line-simplification algorithm. The vertex sequence of the input geometry is recursively subdivided until a run of vertices can be replaced by a straight line segment, with no vertex in that run deviating from the straight line by more than the threshold.

Z values, if present, are not considered when simplifying a set of vertices.

Syntax

```
SE_Generalize (g1 ST_Geometry, threshold float)
```

Return Type

ST_Geometry, unless the input geometry is an ST_Point, ST_MultiPoint, or an empty geometry of any subtype, in which case this function returns NULL

Example

```
CREATE TABLE jagged_lines(line ST_LineString);

INSERT INTO jagged_lines VALUES(
   "0 linestring(10 10, 20 20, 20 18, 30 30, 30 28, 40 40)"
);

-- Small threshold; no vertices removed:
SELECT SE_Generalize(line, 0.5) FROM jagged_lines;


(expression)  0 LINESTRING (10 10, 20 20, 20 18, 30 30, 30 28, 40 40)

-- Larger threshold; some vertices removed:
SELECT SE_Generalize(line, 2) FROM jagged_lines;


(expression)  0 LINESTRING (10 10, 40 40)
```

ST_GeometryN()

ST_GeometryN() takes a takes an ST_GeomCollection (ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon) and an INTEGER index and returns the *n*th **ST_Geometry** object in the collection.

Syntax

```
ST_GeometryN(mpt1 ST_MultiPoint, index integer)
ST_GeometryN(mln1 ST_MultiLineString, index integer)
ST_GeometryN(mpl1 ST_MultiPolygon, index integer)
```

Return Type

ST_Geometry

Example

The city engineer wants to know which building footprints are all inside the first polygon of the **lots** ST_MultiPolygon.

The **building_id** column uniquely identifies each row of the **buildingfootprints** table. The **lot_id** column identifies the building's lot. The **footprint** column stores the building geometries:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);

CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The query lists the **buildingfootprints** table values of **building_id** and **lot_id** for all building footprints that are all within the first lot polygon. The **ST_GeometryN()** function returns a first lot polygon element in the ST_MultiPolygon:

```
SELECT bf.building_id,bf.lot_id
   FROM buildingfootprints bf,lots
   WHERE ST_Within(footprint,ST_GeometryN(lot,1))
   AND bf.lot_id = lots.lot_id;
```

ST_GeometryType()

ST_GeometryType() takes an ST_Geometry object and returns its geometry type as a string.

Syntax

```
ST_GeometryType (g1 ST_Geometry)
```

Return Type

VARCHAR(32) containing one of the following text strings:

- st_point
- st_linestring
- st_polygon
- st_multipoint
- st_multilinestring
- st_multipolygon

Example

The **geometrytype_test** table contains the **g1** ST_Geometry column:

```
CREATE TABLE geometrytype_test(g1 ST_Geometry);
```

The following INSERT statements insert each geometry subclass into the **g1** column:

```
INSERT INTO geometrytype_test VALUES(
   ST_GeomFromText('point (10.02 20.01)',1000)
);

INSERT INTO geometryType_test VALUES(
   ST_GeomFromText('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 1000)
);

INSERT INTO geometrytype_test VALUES(
   ST_GeomFromText('polygon ((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94, 10.02 20.01))',1000)
);

INSERT INTO geometrytype_test VALUES(
   ST_GeomFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)', 1000)
);

INSERT INTO geometrytype_test VALUES(
   ST_GeomFromText('multilinestring ((10.02 20.01,10.32
23.98,11.92 25.64),(9.55 23.75,15.36 30.11))',1000)
);

INSERT INTO geometrytype_test VALUES(
   ST_GeomFromText('multipolygon (((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94 ,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52
32.87,52.43 31.90,51.71 21.73)))'
,1000)
);
```

The following query lists the geometry type of each subclass stored in the **g1** geometry column:

```
SELECT ST_GeometryType(g1) Geometry_type
   FROM geometrytype_test;
```

```
geometry_type  st_point

geometry_type  st_linestring

geometry_type  st_polygon

geometry_type  st_multipoint

geometry_type  st_multilinestring

geometry_type  st_multipolygon
```

ST_GeomFromGML()

ST_GeomFromGML() takes a GML2 or GML version 3 string representation and an optional spatial reference ID and returns a geometry object.

If the **srsName** attribute is specified in the GML string, a corresponding entry must exist in the **spatial_references** table, or it must be specified as UNKNOWN or DEFAULT. The GML representation can include Z and M measures, but must include the appropriate **srsDimension** attribute. The following table describes the corresponding geometry values for each **srsDimension** value.

Table 8-2. Geometry values for srsDimension values

srsDimension value	Geometry value
2	A geometry value with only X and Y coordinate values.
3	A geometry value with X, Y, and Z coordinate values in GML version 3 or using the <coordinates> tag in GML 2. A geometry value with X, Y and measure values if the <X>, <Y>, and <M> tags are used in the <coord> elements of the GML representation.
4	A geometry value with X, Y, Z, and measure coordinate values.

Syntax

```
ST_GeomFromGML(gml_string lvarchar)
ST_GeomFromGML(gml_string lvarchar, SRID integer)
```

For *SRID*, specify 2 if the GML conforms to GML version 2, or specify 3 for GML3. The default is 3.

Return Type

ST_Geometry

Example

The **geometry_test** table contains the INTEGER **gid** column, which uniquely identifies each row, and the **g1** column, which stores the geometry:

```
CREATE TABLE geometry_test (gid smallint, g1 ST_Geometry);
```

The following INSERT statements insert the data into the **gid** and **g1** columns of the **geometry_test** table. The **ST_GeomFromGML()** function converts the GML text representation of each geometry into its corresponding instantiable subclass:

```
INSERT INTO geometry_test VALUES (
1,
ST_GeomFromGML('<gml:Point srsName="DEFAULT" srsDimension="2">
  <gml:pos>10.02 20.01</gml:pos></gml:Point>',1000)) ;

INSERT INTO geometry_test VALUES (
2,
ST_GeomFromGML('<gml:LineString srsName="DEFAULT" srsDimension="2">
  <gml:posList dimension="2">10.01 20.01 10.01 30.01 10.01 40.01
  </gml:posList>
  </gml:LineString>',1000)) ;

INSERT INTO geometry_test VALUES (
3,
ST_GeomFromGML('<gml:Polygon srsName="DEFAULT" srsDimension="2">
  <gml:exterior>
```

```
  <gml:LinearRing>
  <gml:posList dimension="2">
    10.02 20.01 19.15 33.94 25.02 34.15 11.92 35.64 10.02 20.01
  </gml:posList>
  </gml:LinearRing>
  </gml:exterior>
  </gml:Polygon>',1000)) ;

INSERT INTO geometry_test VALUES (
4,
ST_GeomFromGML('<gml:MultiPoint srsName="DEFAULT" srsDimension="2">
  <gml:PointMember>
  <gml:Point srsName="DEFAULT" srsDimension="2">
  <gml:pos>10.02 20.01</gml:pos>
  </gml:Point>
  </gml:PointMember><gml:PointMember>
  <gml:Point srsName="DEFAULT" srsDimension="2">
  <gml:pos>10.32 23.98</gml:pos>
  </gml:Point>
  </gml:PointMember>
  <gml:PointMember>
  <gml:Point srsName="DEFAULT" srsDimension="2">
  <gml:pos>11.92 25.64</gml:pos>
  </gml:Point>
  </gml:PointMember>
  </gml:MultiPoint>',1000)) ;

INSERT INTO geometry_test VALUES (
5,
ST_GeomFromGML('<gml:MultiLineString srsName="DEFAULT" srsDimension="2">
  <gml:LineStringMember>
  <gml:LineString srsName="DEFAULT" srsDimension="2">
  <gml:posList dimension="2">10.02 20.01 10.32 23.98 11.92 25.64
  </gml:posList>
  </gml:LineString>
  </gml:LineStringMember>
  <gml:LineStringMember>
  <gml:LineString srsName="DEFAULT" srsDimension="2">
  <gml:posList dimension="2">9.55 23.75 15.36 30.11</gml:posList>
  </gml:LineString>
  </gml:LineStringMember>
  </gml:MultiLineString>',1000)) ;

INSERT INTO geometry_test VALUES(
6,
ST_GeomFromGML('<gml:MultiPolygon srsName="DEFAULT" srsDimension="2">
  <gml:PolygonMember><gml:Polygon srsName="DEFAULT" srsDimension="2">
  <gml:exterior>
  <gml:LinearRing>
  <gml:posList dimension="2">
    10.02 20.01 19.15 33.94 25.02 34.15 11.92 35.64 10.02 20.01
  </gml:posList>
  </gml:LinearRing>
  </gml:exterior>
  </gml:Polygon>
  </gml:PolygonMember>
  <gml:PolygonMember>
  <gml:Polygon srsName="DEFAULT" srsDimension="2">
  <gml:exterior><gml:LinearRing>
  <gml:posList dimension="2">
    51.71 21.73 73.36 27.04 71.52 32.87 52.43 31.9 51.71 21.73
  </gml:posList>
  </gml:LinearRing>
  </gml:exterior>
  </gml:Polygon>
  </gml:PolygonMember>
  </gml:MultiPolygon>',1000)) ;
```

ST_GeomFromKML()

ST_GeomFromKML() takes a KML fragment and returns an ST_Geometry corresponding to the fragment.

Syntax

```
ST_GeomFromKML(kml_string lvarchar)
```

Return Type

Depends on the KML fragment type, as shown in the following table.

Table 8-3. KML Fragment to Return Type Mapping

KML Fragment	Return Type
Point	ST_Point
LineString	ST_LineString
Polygon	ST_Polygon
MultiGeometry plus Point	ST_MultiPoint
MultiGeometry plus LineString	ST_MultiLineString
MultiGeometry plus Polygon	ST_MultiPolygon

Example

The **geometry_test** table contains the INTEGER **gid** column, which uniquely identifies each row, and the **geom** column, which stores the geometry:

```
CREATE TABLE geometry_test (gid INTEGER, geom ST_Geometry);
```

The following INSERT statements insert the data into the **gid** and **geom** columns of the **geometry_test** table. The **ST_GeomFromKML()** function converts the KML text representation of each geometry into its corresponding instantiable subclass:

```
INSERT INTO geometry_test VALUES(1,ST_GeomFromKML('<Point><coordinates>
                                 10.02,20.01</coordinates></Point>',4))

INSERT INTO geometry_test VALUES(2,ST_GeomFromKML('<LineString><coordinates>
                                 10.01,20.01 20.01,30.01 30.01,40.01</coordinates>
                                 </LineString>',4));
```

SE_GeomFromShape()

SE_GeomFromShape() takes a shape and a spatial reference ID and returns a geometry object.

Syntax

```
SE_GeomFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_Geometry

Example

The following C code fragment contains ODBC functions embedded with Spatial DataBlade SQL functions that insert data into the **lots** table.

The **lots** table was created with two columns: the **lot_id**, which uniquely identifies each lot, and the **lot** polygon column, which contains the geometry of each lot:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The **SE_GeomFromShape()** function converts shapes into Spatial DataBlade geometry. The entire INSERT statement is copied into **shp_sql**. The INSERT statement contains parameter markers to accept the **lot_id** and **lot** data, dynamically:

```
/* Create the SQL insert statement to populate the lots
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO lots (lot_id, lot) "
            "VALUES(?, SE_GeomFromShape(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the lot_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &lot_id, 0, &pcbvalue1);

    /* Bind the lot geometry to the second parameter. */
    pcbvalue2 = lot_shape_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, lot_shape_len, 0,
                           lot_shape_buf, lot_shape_len, &pcbvalue2);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_GeomFromText()

ST_GeomFromText() takes a well-known text representation and a spatial reference ID and returns a geometry object.

Syntax

```
ST_GeomFromText(wkt lvarchar, SRID integer)
```

Return Type

ST_Geometry

Example

The **geometry_test** table contains the INTEGER **gid** column, which uniquely identifies each row, and the **g1** column, which stores the geometry:

```
CREATE TABLE geometry_test (gid smallint,
                            g1  ST_Geometry);
```

The following INSERT statements insert the data into the **gid** and **g1** columns of the **geometry_test** table. The **ST_GeomFromText()** function converts the text representation of each geometry into its corresponding instantiable subclass:

```
INSERT INTO geometry_test VALUES(
   1,
   ST_GeomFromText('point (10.02 20.01)',1000)
);

INSERT INTO geometry_test VALUES(
   2,
   ST_GeomFromText('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)',1000)
);

INSERT INTO geometry_test VALUES(
   3,
   ST_GeomFromText('polygon ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01))',1000)
);

INSERT INTO geometry_test VALUES(
   4,
   ST_GeomFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO geometry_test VALUES(
   5,
   ST_GeomFromText('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64),(9.55 23.75,15.36 30.11))',1000)
);

INSERT INTO geometry_test VALUES(
   6,
   ST_GeomFromText('multipolygon (((10.02 20.01, 11.92 35.64,
25.02 34.15, 19.15 33.94, 10.02 20.01)),((51.71 21.73, 73.36
27.04, 71.52 32.87, 52.43 31.90, 51.71 21.73)))',1000)
);
```

ST_GeomFromWKB()

ST_GeomFromWKB() takes a well-known binary representation and a spatial reference ID to return a geometry object.

Syntax

```
ST_GeomFromWKB(WKB lvarchar, SRID integer)
```

Return Type

ST_Geometry

Example

The following C code fragment contains ODBC functions embedded with Spatial DataBlade SQL functions that insert data into the **lots** table.

The lots table was created with two columns: the **lot_id**, which uniquely identifies each lot, and the **lot** polygon column, which contains the geometry of each lot:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The **ST_GeomFromWKB()** function converts WKB representations into Spatial DataBlade geometry. The entire INSERT statement is copied into a **wkb_sql** CHAR string. The INSERT statement contains parameter markers to accept the **lot_id** and **lot** data, dynamically:

```
/* Create the SQL insert statement to populate the lots
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO lots (lot_id, lot) "
            "VALUES(?, ST_GeomFromWKB(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the lot_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &lot_id, 0, &pcbvalue1);

    /* Bind the lot geometry to the second parameter. */
    pcbvalue2 = lot_wkb_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, lot_wkb_len, 0,
                           lot_wkb_buf, lot_wkb_len, &pcbvalue2);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

SE_InRowSize()

SE_InRowSize() returns the size of the in-row portion of a geometry. Geometries which are less than 930 bytes are stored entirely in-row: that is, the entire value is stored in a table's dbspace.

You can use this function to obtain an estimate of the amount of diskspace consumed by one or more geometries. However, this function does not account for dbspace and sbspace overhead, so cannot be used to obtain an exact total.

Syntax

```
SE_InRowSize(ST_Geometry)
```

Return Type

INTEGER

See Also

"SE_OutOfRowSize()" on page 8-113

"SE_TotalSize()" on page 8-145

ST_InteriorRingN()

ST_InteriorRingN() returns the *n*th interior ring of a polygon as an ST_LineString. The order of the rings cannot be predefined since the rings are organized according to the rules defined by the internal geometry verification routines and not by geometric orientation.

Syntax

```
ST_InteriorRingN(pl1 ST_Polygon, index integer)
```

Return Type

ST_LineString

Example

An ornithologist studying the bird population on several South Sea islands knows that the feeding zone of this passive species is restricted to the seashore. Some of the islands are so large they have several lakes on them. The shorelines of the lakes are inhabited exclusively by another more aggressive species. The ornithologist knows that if the perimeter of the ponds on each island exceeds a certain threshold, the aggressive species will become so numerous that it will threaten the passive seashore species. Therefore, the ornithologist requires the aggregated perimeter of the interior rings of the islands.

Figure 8-7 shows the exterior rings of the islands that represent the ecological interface each island shares with the sea. Some of the islands have lakes, which are represented by the interior rings of the polygons

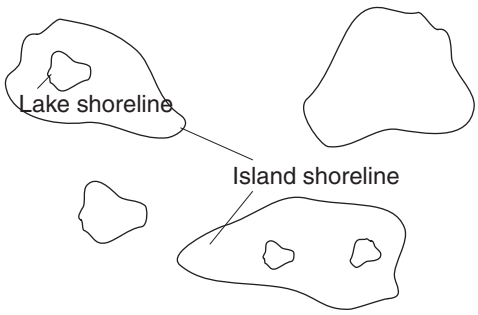


Figure 8-7. Islands and Lakes.

The **ID** and **name** columns of the **islands** table identifies each island, while the **land** ST_Polygon column stores the island geometry:

```
CREATE TABLE islands (id    integer,
                      name  varchar(32),
                      land  ST_Polygon);
```

This ODBC code fragment uses the **ST_InteriorRingN()** function to extract the interior ring (lake) from each island polygon as a linestring. The perimeter of the linestring returned by the **ST_Length()** function is totaled and displayed along with the island ID:

```
/* Prepare and execute the query to get the island IDs and number
       of lakes (interior rings); */
    sprintf(sql_stmt,
            "SELECT id, ST_NumInteriorRing(land) FROM islands");

    /* Allocate memory for the island cursor */
```

```
rc = SQLAllocHandle (SQL_HANDLE_STMT, hdbc, &island_cursor);

rc = SQLExecDirect (island_cursor, (UCHAR *)sql_stmt, SQL_NTS);

/* Bind the island table's id column to island_id. */
rc = SQLBindCol (island_cursor, 1, SQL_C_SLONG,
                 &island_id, 0, &id_ind);

/* Bind the result of ST_NumInteriorRing(land) to num_lakes. */
rc = SQLBindCol (island_cursor, 2, SQL_C_SLONG,
                 &num_lakes, 0, &lake_ind);

/* Allocate memory to the SQL statement handle lake_cursor. */
rc = SQLAllocHandle (SQL_HANDLE_STMT, hdbc, &lake_cursor);

/* Prepare the query to get the length of an interior ring. For
 * efficiency, we only prepare this query once. */
sprintf (sql_stmt,
         "SELECT ST_Length(ST_InteriorRingN(land, ?))"
         "FROM islands WHERE id = ?");
rc = SQLPrepare (lake_cursor, (UCHAR *)sql_stmt, SQL_NTS);

/* Bind the lake_number to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (lake_cursor, 1, SQL_PARAM_INPUT, SQL_C_LONG,
                       SQL_INTEGER, 0, 0,
                       &lake_number, 0, &pcbvalue1);

/* Bind the island_id to the second parameter. */
pcbvalue2 = 0;
rc = SQLBindParameter (lake_cursor, 2, SQL_PARAM_INPUT, SQL_C_LONG,
                       SQL_INTEGER, 0, 0,
                       &island_id, 0, &pcbvalue2);

/* Bind the result of the ST_Length function to lake_perimeter. */
rc = SQLBindCol (lake_cursor, 1, SQL_C_SLONG,
                 &lake_perimeter, 0, &length_ind);

/* Outer loop:
 * get the island ids and the number of lakes (interior rings).*/
while (1)
{
    /* Fetch an island.*/
    rc = SQLFetch (island_cursor);
    if (rc == SQL_NO_DATA)
        break;
    else
        returncode_check(NULL, hstmt, rc, "SQLFetch");

    /* Inner loop: for this island,
     * get the perimeter of all its lakes (interior rings). */
    for (total_perimeter = 0,lake_number = 1;
         lake_number <= num_lakes;
         lake_number++)
    {
        rc = SQLExecute (lake_cursor);
        rc = SQLFetch (lake_cursor);
        total_perimeter += lake_perimeter;
        SQLFreeStmt (lake_cursor, SQL_CLOSE);
    }

    /* Display the island ID and the total perimeter of its lakes.*/
    printf ("Island ID = %d, Total lake perimeter = %d\n",
            island_id,total_perimeter);
}

SQLFreeStmt (lake_cursor, SQL_DROP);
SQLFreeStmt (island_cursor, SQL_DROP);
```

ST_Intersection()

ST_Intersection() takes two ST_Geometry objects and returns the intersection set as an ST_Geometry object. If the two objects do not intersect, the return value is an empty geometry.

Syntax

```
ST_Intersection(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

ST_Geometry

Example

The fire marshal must obtain the areas of the hospitals, schools, and nursing homes intersected by the radius of a possible hazardous waste contamination.

The sensitive areas are stored in the **sensitive_areas** table that is created with the CREATE TABLE statement that follows. The **zone** column defined as an ST_Polygon type stores the outline of each of the sensitive areas:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The hazardous sites are stored in the **hazardous_sites** table created with the CREATE TABLE statement that follows. The **location** column, defined as an ST_Point type, stores a location that is the geographic center of each hazardous site:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The **ST_Buffer()** function generates a five-mile buffer surrounding the hazardous waste site locations. The **ST_Intersection()** function generates polygons from the intersection of the buffered hazardous waste sites and the sensitive areas. The **ST_Area()** function returns the intersection polygons' area, which is summarized for all hazardous sites by the SUM operator. The GROUP BY clause directs the query to aggregate the intersection areas by hazardous waste site ID:

```
SELECT hs.site_id, SUM(ST_Area(ST_Intersection(sa.zone,
       ST_Buffer(hs.location,(5 * 5280)))::ST_MultiPolygon))
   FROM sensitive_areas sa, hazardous_sites hs
   GROUP BY hs.site_id;

    site_id          (sum)

        102 87000000.00000
         59 77158581.63280
```

In Figure 8-8, the circles represent the five-mile buffer polygons surrounding the hazardous waste sites. The intersection of these buffer polygons with the sensitive area polygons produces three polygons: the hospital in the upper left-hand corner is intersected twice, while the school in the lower right-hand corner is intersected only once.

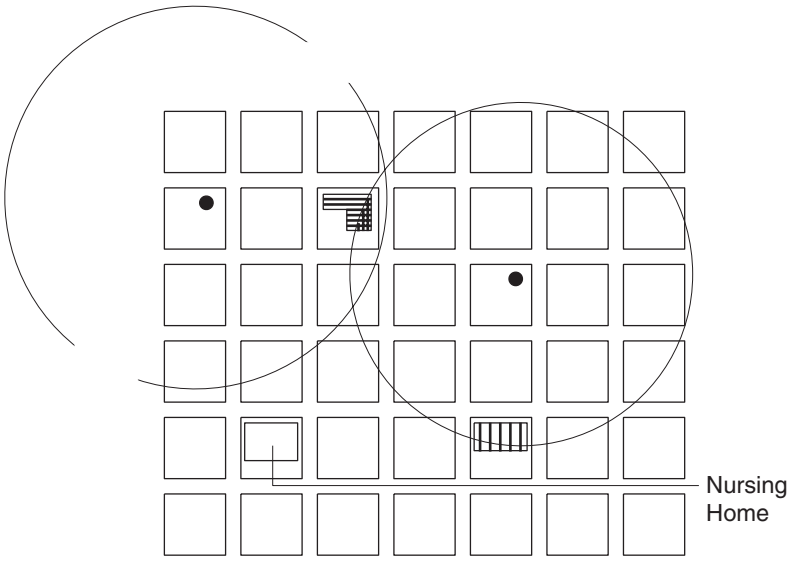


Figure 8-8. Using the ST_Intersection Function

ST_Intersects()

ST_Intersects() returns `t` (TRUE) if the intersection of two geometries does not result in an empty set; otherwise, returns `f` (FALSE).

Syntax

```
ST_Intersects (g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The fire marshal wants a list of sensitive areas within a five-mile radius of a hazardous waste site.

The sensitive areas are stored in the **sensitive_areas** table created below. The **zone** column is defined as an ST_Polygon type and stores the outline of the sensitive areas:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The hazardous sites are stored in the **hazardous_sites** table created with the CREATE TABLE statement that follows. The **location** column, defined as an ST_Point type, stores the geographic center of each hazardous site:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The query returns a list of sensitive-area and hazardous-site names for sensitive areas that intersect the five-mile buffer radius of the hazardous sites:

```
SELECT sa.name, hs.name
   FROM sensitive_areas sa, hazardous_sites hs
   WHERE ST_Intersects(ST_Buffer(hs.location,(5 * 5280)),sa.zone);


name  Johnson County Hospital
name  W. H. Kleenare Chemical Repository

name  Johnson County Hospital
name  Landmark Industrial

name  Summerhill Elementary School
name  Landmark Industrial
```

SE_Is3D()

SE_Is3d() returns t (TRUE) if the ST_Geometry object has three-dimensional coordinates; otherwise, returns f (FALSE). Properties of geometries are described in "Properties of the Spatial Data Types" on page 2-1.

Syntax

```
SE_Is3D(g1 ST_Geometry)
```

Return Type

BOOLEAN

Example

The **threed_test** table is created with INTEGER **gid** and **g1** ST_Geometry columns:

```
CREATE TABLE threed_test (gid   smallint,
                          g1    ST_Geometry);
```

The following INSERT statements insert two points into the **threed_test** table. The first point does not contain Z coordinates, while the second does:

```
INSERT INTO threed_test VALUES(
   1, ST_PointFromText('point (10 10)',1000)
);

INSERT INTO threed_test VALUES(
   1, ST_PointFromText('point z(10.92 10.12 5)',1000)
);
```

The query lists the contents of the **gid** column with the results of the **SE_Is3d** function. The function returns a 0 for the first row, which does not have a Z coordinate, and a 1 for the second row, which does:

```
SELECT gid, SE_Is3D (g1) is_it_3d from threed_test;

   gid is_it_3d

     1        f
     1        t
```

ST_IsClosed()

ST_IsClosed() takes an ST_LineString or ST_MultiLineString and returns `t` (TRUE) if it is closed; otherwise, it returns `f` (FALSE). Properties of geometries are described in "Properties of the Spatial Data Types" on page 2-1.

Syntax

```
ST_IsClosed(ln1 ST_LineString)
ST_IsClosed(mln1 ST_MultiLineString)
```

Return Type

BOOLEAN

Example

The **closed_linestring** table is created with a single ST_LineString column:

```
CREATE TABLE closed_linestring (ln1 ST_LineString);
```

The following INSERT statements insert two records into the **closed_linestring** table. The first record is not a closed linestring, while the second is:

```
INSERT INTO closed_linestring VALUES(
   ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)', 1000)
);

INSERT INTO closed_linestring VALUES(
   ST_LineFromText('linestring (10.02 20.01,11.92 35.64,25.02 34.15,
   19.15 33.94,10.02 20.01)',1000)
);
```

The query returns the results of the **ST_IsClosed()** function. The first row returns a `0` because the linestring is not closed, while the second row returns a 1 because the linestring is closed.

```
SELECT ST_IsClosed(ln1) Is_it_closed
   FROM closed_linestring;

is_it_closed

           f
           t
```

The **closed_mlinestring** table is created with a single ST_MultiLineString column:

```
CREATE TABLE closed_mlinestring (mln1 ST_MultiLineString);
```

The following INSERT statements insert an ST_MultiLineString record that is not closed and another that is:

```
INSERT INTO closed_mlinestring VALUES(
   ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,
   11.92 25.64),(9.55 23.75,15.36 30.11))',1000)
);

INSERT INTO closed_mlinestring VALUES(
   ST_MLineFromText('multilinestring ((10.02 20.01,11.92 35.64,
   25.02 34.15,19.15 33.94,10.02 20.01),(51.71 21.73,73.36 27.04,
   71.52 32.87,52.43 31.90,51.71 21.73))',1000)
);
```

The query lists the results of the **ST_IsClosed()** function. The first row returns `0` because the multilinestring is not closed. The second row returns `1` because the multilinestring stored in the **mln1** column is closed. A multilinestring is closed if all of its linestring elements are closed:

```
SELECT ST_IsClosed(mln1) Is_it_closed
   FROM closed_mlinestring;

is_it_closed

           f
           t
```

ST_IsEmpty()

ST_IsEmpty() returns `t` (TRUE) if the geometry is empty; otherwise, returns `f` (FALSE). See a description of properties of geometries in "Properties of the Spatial Data Types" on page 2-1.

Syntax

```
ST_IsEmpty(g1 ST_Geometry)
```

Return Type

BOOLEAN

Example

The CREATE TABLE statement below creates the **empty_test** table with geotype, which stores the data type of the subclasses that are stored in the **g1** ST_Geometry column:

```
CREATE TABLE empty_test (geotype varchar(20),
                         g1      ST_Geometry);
```

The following INSERT statements insert two records each for the geometry subclasses: point, linestring, and polygon; one record is empty and one is not:

```
INSERT INTO empty_test VALUES(
    'Point', ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO empty_test VALUES(
    'Point', ST_PointFromText('point empty',1000)
);

INSERT INTO empty_test VALUES(
    'Linestring',
    ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',1000)
);

INSERT INTO empty_test VALUES(
   'Linestring',
   ST_LineFromText('linestring empty',1000)
);

INSERT INTO empty_test VALUES(
    'Polygon',
    ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94,10.02 20.01))',1000)
);

INSERT INTO empty_test VALUES(
    'Polygon',
    ST_PolyFromText('polygon empty',1000)
);
```

The query returns the geometry type from the **geotype** column and the results of the **ST_IsEmpty()** function:

```
SELECT geotype, ST_IsEmpty(g1) Is_it_empty
   FROM empty_test


geotype              is_it_empty

Point                          f
```

```
Point                          t
Linestring                     f
Linestring                     t
Polygon                        f
Polygon                        t
```

SE_IsMeasured()

SE_IsMeasured() returns `t` (TRUE) if the ST_Geometry object has measures; otherwise, returns `f` (FALSE). Properties of geometries are described in "Properties of the Spatial Data Types" on page 2-1.

Syntax

```
SE_IsMeasured(g1 ST_Geometry)
```

Return Type

BOOLEAN

Example

The **measure_test** table is created with two columns: a SMALLINT column, **gid**, which uniquely identifies rows, and **g1**, an ST_Geometry column, which stores the ST_Point geometries:

```
CREATE TABLE measure_test (gid smallint,
                           g1  ST_Geometry);
```

The following INSERT statements insert two records into the **measure_test** table. The first record stores a point that does not have a measure, while the second record does have a measure value:

```
INSERT INTO measure_test VALUES(
    1,
    ST_PointFromText('point (10 10)',1000)
);

INSERT INTO measure_test VALUES(
    2,
    ST_PointFromText('point m (10.92 10.12 5)',1000)
);
```

The query lists the **gid** column and the results of the **SE_IsMeasured()** function. The **SE_IsMeasured()** function returns a `0` for the first row because the point does not have a measure; it returns a `1` for the second row because the point does have measures:

```
SELECT gid,SE_IsMeasured(g1) Has_measures
   FROM measure_test;

   gid has_measures

     1            f
     2            t
```

ST_IsRing()

ST_IsRing() takes an ST_LineString and returns `t` (TRUE) if it is a ring (that is, the ST_LineString is closed and simple); otherwise, it returns `f` (FALSE). Properties of geometries are described in "Properties of the Spatial Data Types" on page 2-1.

Syntax

```
ST_IsRing(ln1 ST_LineString)
```

Return Type

BOOLEAN

Example

The **ring_linestring** table is created with the SMALLINT column **gid** and the ST_LineString column **ln1**:

```
CREATE TABLE ring_linestring (gid smallint,
                              ln1 ST_LineString);
```

The following INSERT statements insert three linestrings into the **ln1** column. The first row contains a linestring that is not closed and is not a ring. The second row contains a closed and simple linestring that is a ring. The third row contains a linestring that is closed, but not simple, because it intersects its own interior. It is also not a ring:

```
INSERT INTO ring_linestring VALUES(
   1,
   ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)', 1000)
);

INSERT INTO ring_linestring VALUES(
   2,
   ST_LineFromText('linestring (10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94, 10.02 20.01)', 1000)
);

INSERT INTO ring_linestring VALUES(
   3,
   ST_LineFromText('linestring (15.47 30.12,20.73 22.12,10.83
14.13,16.45 17.24,21.56 13.37,11.23 22.56,19.11 26.78,15.47 30.12)', 1000)
);
```

The query returns the results of the **ST_IsRing()** function. The first and third rows return `0` because the linestrings are not rings, while the second row returns `1` because it is a ring:

```
SELECT gid, ST_IsRing(ln1) Is_it_a_ring
   FROM ring_linestring;

   gid is_it_a_ring

     1            f
     2            t
     3            f
```

ST_IsSimple()

ST_IsSimple() returns `t` (TRUE) if the geometry object is simple; otherwise, it returns `f` (FALSE). Properties of geometries are described in "Properties of the Spatial Data Types" on page 2-1.

Syntax

```
ST_IsSimple (g1 ST_Geometry)
```

Return Type

BOOLEAN

Example

The table **issimple_test** is created with two columns. The **pid** column is a SMALLINT containing the unique identifier for each row. The **g1** ST_Geometry column stores the simple and nonsimple geometry samples:

```
CREATE TABLE issimple_test (pid smallint,
                            g1  ST_Geometry);
```

The following INSERT statements insert two records into the **issimple_test** table. The first is a simple linestring because it does not intersect its interior. The second is nonsimple because it does intersect its interior:

```
INSERT INTO issimple_test VALUES(
   1,
   ST_LineFromText('linestring (10 10, 20 20, 30 30)',1000)
);

INSERT INTO issimple_test VALUES(
   2,
   ST_LineFromText('linestring (10 10,20 20,20 30,10 30,10 20,20 10)',1000)
);
```

The query returns the results of the **ST_IsSimple()** function. The first record returns `t` because the linestring is simple, while the second record returns `f` because the linestring is not simple:

```
SELECT pid, ST_IsSimple(g1) Is_it_simple
   FROM issimple_test;

   pid is_it_simple

     1           t
     2           f
```

ST_IsValid()

ST_IsValid() takes an ST_Geometry and returns `t` (TRUE) if it is topologically correct; otherwise it returns `f` (FALSE). Properties of geometries are described in "Properties of the Spatial Data Types" on page 2-1.

The IBM Informix Spatial DataBlade Module validates spatial data before accepting it, so **ST_IsValid()** always returns TRUE. This function may be used to validate spatial data supplied by other implementations of the OpenGIS spatial data specification.

Syntax

```
ST_IsValid(g ST_Geometry)
```

Return Type

BOOLEAN

ST_Length()

ST_Length() returns the length of an ST_LineString or ST_MultiLineString.

Syntax

```
ST_Length(ln1 ST_LineString)
ST_Length(mln1 ST_MultiLineString)
```

Return Type

DOUBLE PRECISION

Example

A local ecologist studying the migratory patterns of the salmon population in the county's waterways wants the length of all stream and river systems within the county.

The **waterways** table is created with the **ID** and **name** columns that identify each stream and river system stored in the table. The **water** column is a multilinestring, because the river and stream systems are often an aggregate of several linestrings:

```
CREATE TABLE waterways (id     integer,
                        name   varchar(128),
                        water  ST_MultiLineString);
```

The query returns the name of each system along with the length of the system generated by the length function:

```
SELECT name, ST_Length(water) Length
   FROM waterways;


name    Fedders creek
length  175853.9869703
```

Figure 8-9 displays the river and stream systems that lie within the county boundary.

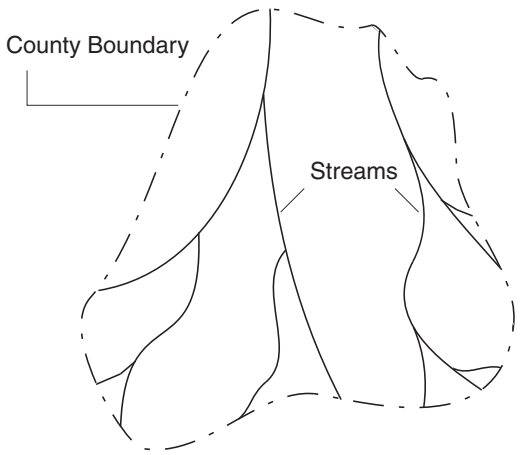


Figure 8-9. Stream and River Systems

ST_LineFromGML()

ST_LineFromGML() takes a GML2 or GML3 string representation of an ST_LineString and an optional spatial reference ID and returns a polyline object.

Syntax

```
ST_LineFromGML(gmlstring lvarchar)
ST_LineFromGML(gmlstring lvarchar, SRID integer)
```

Return Type

ST_LineString

Example

The **gml_linetest** table is created with the SMALLINT column **gid** and the ST_LineString column **ln1**:

```
CREATE TABLE gml_linetest (gid smallint, ln1 ST_LineString);

INSERT INTO gml_linetest VALUES (1, ST_LineFromGML('<gml:LineString>
               <gml:posList> -110.45 45.256 -109.48 46.46 -109.86 43.84
               </gml:posList></gml:LineString>',4));
INSERT INTO gml_linetest VALUES (2, ST_LineFromGML('<gml:LineString
               srsName="EPSG:4326" srsDimension="3"><gml:posList
               dimension="3">-110.449999933 45.2559999343 10
               -109.47999994 46.4600005499 10 -109.86000008
               43.8400000201 20</gml:posList></gml:LineString>'));
INSERT INTO gml_linetest VALUES(3,ST_LineFromGML('<gml:LineString
               srsName="EPSG:4326" srsDimension="4"><gml:posList
               dimension="4">-110.449999933 45.2559999343 10  54
               -109.47999994 46.4600005499 10  58
               -109.86000008 43.8400000201 20 64 </gml:posList>
               </gml:LineString>'));
```

The first record specifies a spatial reference ID of 4 (WGS84) and a default dimension of 2. The second and third records contain Z and M measures and pass the spatial reference ID through the srsName attribute.

Output:

```
SELECT * FROM gml_linetest;

gid  1
ln1  4 LINESTRING (-110.449999933 45.2559999343,
                   -109.47999994 46.4600000469,
                   -109.86000008 43.8400000201)

gid  2
ln1  4 LINESTRING Z (-110.449999933 45.2559999343 10,
                     -109.47999994 46.4600005499 10,
                     -109.86000008 43.8400000201 20)

gid  3
ln1  4 LINESTRING ZM (-110.449999933 45.2559999343 10 54,
                      -109.47999994 46.4600005499 10 58,
                      -109.86000008 43.8400000201 20 64)
```

ST_LineFromKML()

ST_LineFromKML() takes a KML LineString string and an optional spatial reference ID and returns a linestring object. A KML LineString string can contain the KML shape attributes <extrude>, <tessellate>, and <altitudeMode>, but they are ignored in the ST_LineString representation.

Syntax

```
ST_LineFromKML(kmlstring lvarchar)
ST_LineFromKML(kmlstring lvarchar, SRID integer)
```

Return Type

ST_LineString

Example

```
EXECUTE FUNCTION ST_LineFromKML('<LineString><coordinates>-130.597293,50.678292,
                            0 -129.733457,50.190606,0 -130.509877,49.387208,
                            0 -128.801553,48.669761,0 -129.156745,47.858658,
                           0 -128.717835,47.739997,0</coordinates></LineString>',4);
```

Output:

```
(expression)  4 LINESTRING Z (-130.597292947 50.6782919759 0, -129.733456972
50.1906059982 0, -130.509877068 49.3872080751 0, -128.801553066 48.669761042 0,
-129.156744951 47.8586579701 0, -128.717834948 47.7399970362 0)
```

SE_LineFromShape()

SE_LineFromShape() takes a shape of type polyline and a spatial reference ID to return an ST_LineString. A polyline with only one part is appropriate as an ST_LineString, and a polyline with multiple parts is appropriate as an ST_MultiLineString (see "SE_MLineFromShape()" on page 8-94).

Syntax

```
SE_LineFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_LineString

Example

The **sewerlines** table is created with three columns: **sewer_id**, which uniquely identifies each sewer line; the INTEGER **class** column, which identifies the type of sewer line (generally associated with the line's capacity); and the **sewer** ST_LineString column, which stores the sewer line geometry:

```
CREATE TABLE sewerlines (sewer_id  integer,
                         class     integer,
                         sewer     ST_LineString);
```

This code fragment populates the **sewerlines** table with the unique ID, class, and geometry of each sewer line:

```
/* Create the SQL insert statement to populate the sewerlines
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO sewerlines (sewer_id,class,sewer) "
            "VALUES(?, ?, SE_LineFromShape(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the sewer_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &sewer_id, 0, &pcbvalue1);

    /* Bind the sewer_class to the second parameter. */
    pcbvalue2 = 0;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &sewer_class, 0, &pcbvalue2);

    /* Bind the sewer geometry to the third parameter. */
    pcbvalue3 = sewer_shape_len;
    rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, sewer_shape_len, 0,
                           sewer_shape_buf, sewer_shape_len, &pcbvalue3);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_LineFromText()

ST_LineFromText() takes a well-known text representation of type ST_LineString and a spatial reference ID and returns an ST_LineString.

Syntax

```
ST_LineFromText(WKT lvarchar, SRID integer)
```

Return Type

ST_LineString

Example

The **linestring_test** table is created with a single **ln1** ST_LineString column:

```
CREATE TABLE linestring_test (ln1 ST_LineString);
```

The following INSERT statement inserts an ST_LineString into the **ln1** column using the **ST_LineFromText()** function:

```
INSERT INTO linestring_test VALUES(
   ST_LineFromText('linestring(10.01 20.03,20.94 21.34,35.93 19.04)',1000)
);
```

ST_LineFromWKB()

ST_LineFromWKB() takes a well-known binary representation of type ST_LineString and a spatial reference ID, returning an ST_LineString.

Syntax

```
ST_LineFromWKB(wkb lvarchar, SRID integer)
```

Return Type

ST_LineString

Example

The **sewerlines** table is created with three columns. The first column, **sewer_id**, uniquely identifies each sewer line. The INTEGER **class** column identifies the type of sewer line, generally associated with the line capacity. The **sewer ST_LineString** column stores the sewer line geometries:

```
CREATE TABLE sewerlines (sewer_id  integer,
                         class     integer,
                         sewer     ST_LineString);
```

This code fragment populates the **sewerlines** table with the unique ID, class, and geometry of each sewer line:

```
/* Create the SQL insert statement to populate the sewerlines
 * table. The question marks are parameter markers that indicate
 * the column values that will be inserted at run time. */
sprintf(sql_stmt,
        "INSERT INTO sewerlines (sewer_id,class,sewer) "
        "VALUES(?, ?, ST_LineFromWKB(?, %d))", srid);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

/* Bind the sewer_id to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                       SQL_INTEGER, 0, 0,
                       &sewer_id, 0, &pcbvalue1);

/* Bind the sewer_class to the second parameter. */
pcbvalue2 = 0;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_SLONG,
                       SQL_INTEGER, 0, 0,
                       &sewer_class, 0, &pcbvalue2);

/* Bind the sewer geometry to the third parameter. */
pcbvalue3 = sewer_wkb_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                       SQL_INFX_UDT_LVARCHAR, sewer_wkb_len, 0,
                       sewer_wkb_buf, sewer_wkb_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

SE_LocateAlong()

SE_LocateAlong() takes a geometry object and a measure to return as an ST_MultiPoint the set of points found having that measure.

Syntax

```
SE_LocateAlong(g1 ST_Geometry, m1 double precision)
```

Return Type

ST_Geometry

Example

The **locatealong_test** table is created with two columns: the **gid** column uniquely identifies each row, and the **g1** ST_Geometry column stores sample geometry:

```
CREATE TABLE locatealong_test (gid integer,
                               g1  ST_Geometry);
```

The following INSERT statements insert two rows. The first is a multilinestring, while the second is a multipoint:

```
INSERT INTO locatealong_test VALUES(
   1,
   ST_MLineFromText('multilinestring m ((10.29 19.23 5,23.82 20.29
6,30.19 18.47 7,45.98 20.74 8),(23.82 20.29 6,30.98 23.98 7,42.92
25.98 8))', 1000)
);

INSERT INTO locatealong_test VALUES(
   2,
   ST_MPointFromText('multipoint m (10.29 19.23 5,23.82 20.29
6,30.19 18.47 7,45.98 20.74 8,23.82 20.29 6,30.98 23.98 7,42.92
25.98 8)', 1000)
);
```

In this query, the **SE_LocateAlong()** function finds points whose measure is 6.5. The first row returns an ST_MultiPoint containing two points. However, the second row returns an empty point. For linear features (geometry with a dimension greater than 0), **SE_LocateAlong()** can interpolate the point, but for multipoints the target measure must match exactly:

```
SELECT gid, SE_locatealong(g1,6.5) Geometry
   FROM locatealong_test;


gid       1
geometry  1000 MULTIPOINT M (27.005 19.38 6.5, 27.4 22.135 6.5)

gid       2
geometry  1000 POINT M EMPTY
```

In this query, the **SE_LocateAlong()** function returns a multipoint for both rows. The target measure of 7 matches measures in the multilinestring and multipoint source data:

```
SELECT gid, SE_locatealong(g1,7) Geometry
   FROM locatealong_test;


gid       1
geometry  1000 MULTIPOINT M (30.19 18.47 7, 30.98 23.98 7)
```

```
gid       2
geometry  1000 MULTIPOINT M (30.19 18.47 7, 30.98 23.98 7)
```

SE_LocateBetween()

SE_LocateBetween() takes an ST_Geometry object and two measure locations and returns an ST_Geometry that represents the set of disconnected paths between the two measure locations.

Syntax

```
SE_LocateBetween(g1 ST_Geometry, fm double precision, tm double precision)
```

Return Type

ST_Geometry

Example

The **locatebetween_test** table is created with two columns: the **gid** INTEGER column uniquely identifies each row, while the **g1** ST_MultiLineString stores the sample geometry:

```
CREATE TABLE locatebetween_test (gid integer, g1 ST_Geometry);
```

The following INSERT statements insert two rows into the **locatebetween_test** table. The first row is an ST_MultiLineString and the second is an ST_MultiPoint:

```
INSERT INTO locatebetween_test VALUES(
   1,
   ST_MLineFromText('multilinestring m ((10.29 19.23 5,23.82 20.29
6, 30.19 18.47 7,45.98 20.74 8),(23.82 20.29 6,30.98 23.98 7,42.92
25.98 8))',1000)
);

INSERT INTO locatebetween_test VALUES(
   2,
   ST_MPointFromText('multipoint m (10.29 19.23 5,23.82 20.29
6,30.19 18.47 7,45.98 20.74 8,23.82 20.29 6,30.98 23.98 7,42.92
25.98 8)', 1000)
);
```

In the query, the **SE_LocateBetween** function locates measures lying between 6.5 and 7.5, inclusively. The first row returns an ST_MultiLineString containing several linestrings. The second row returns an ST_MultiPoint because the source data was ST_MultiPoint. When the source data has a dimension of 0 (point or multipoint), an exact match is required:

```
SELECT gid, SE_LocateBetween(g1,6.5,7.5) Geometry
   FROM locatebetween_test;


gid       1
geometry  1000 MULTILINESTRING M ((27.005 19.38 6.5, 30.19 18.47
7, 38.085 19.6
          05 7.5),(27.4 22.135 6.5, 30.98 23.98 7, 36.95 24.98 7.5))

gid       2
geometry  1000 MULTIPOINT M (30.19 18.47 7, 30.98 23.98 7)
```

SE_M()

SE_M() returns the measure value of a point.

Syntax

```
SE_M(p1 ST_Point)
```

Return Type

DOUBLE PRECISION

Example

The **m_test** table is created with the **gid** INTEGER column, which uniquely identifies the row, and the **pt1** ST_Point column that stores the sample geometry:

```
CREATE TABLE m_test (gid integer,
                     pt1 ST_Point);
```

The following INSERT statements insert a point with measures and a point without measures:

```
INSERT INTO m_test VALUES(
   1,
   ST_PointFromText('point (10.02 20.01)', 1000)
);

INSERT INTO m_test VALUES(
   2,
   ST_PointFromText('point zm (10.02 20.01 5.0 7.0)', 1000)
);
```

In this query, the **SE_M()** function lists the measure values of the points. Because the first point does not have measures, the **SE_M()** function returns NULL:

```
SELECT gid, SE_M(pt1) The_measure
   FROM m_test;

      gid    the_measure

        1
        2 7.000000000000
```

SE_MetadataInit()

For computational efficiency and to allow Spatial DataBlade functions to be executed in parallel, the contents of the **spatial_references** table are kept in both a smart large object and a memory cache. If these copies become corrupt or unreadable, the DataBlade module raises one of the following errors:

- `USE48 SE_Metadata lohandle file not found, unreadable, or corrupt.`
- `USE51 SE_Metadata smart blob is corrupt or unreadable.`
- `USE52 SE_Metadata memory cache is locked.`

Execute the **SE_MetadataInit()** function to reinitialize the spatial reference system smart large object and memory cache.

Syntax

```
SE_MetadataInit()
```

Return Type

The text string OK, if the function was successfully executed

Example

```
execute function SE_MetadataInit();
```

SE_Midpoint()

SE_Midpoint() determines the midpoint of a linestring. The midpoint is defined as that point which is equidistant from both endpoints of a linestring, measuring distance along the linestring.

If the input linestring has Z values or measures, the Z value or measure of the midpoint are computed by linear interpolation between the adjacent vertices.

Syntax

```
SE_Midpoint (ln1 ST_LineString)
```

Return Type

ST_Point

ST_MLineFromGML()

ST_MLineFromGML() takes a GML2 or GML3 string representation of an ST_MultiLineString and an optional spatial reference ID and returns a multipart polyline object.

Syntax

```
ST_MLineFromGML(gmlstring lvarchar)
ST_MLineFromGML(gmlstring lvarchar, SRID integer)
```

Return Type

ST_MultiLineString

Example

The **gml_linetest** table is created with the SMALLINT column **gid** and the ST_MultiLineString column **ln1**:

```
CREATE TABLE gml_linetest(gid smallint, ln1 ST_MultiLineString);

INSERT INTO gml_linetest VALUES (1, ST_MLineFromGML('<gml:MultiLineString>
             <gml:LineStringMember><gml:LineString><gml:posList>-110.45 45.256
             -109.48 46.46 -109.86 43.84</gml:posList></gml:LineString>
             </gml:LineStringMember><gml:LineStringMember><gml:LineString>
             <gml:posList>-99.45 33.256 -99.48 36.46 -99.86 33.84</gml:posList>
             </gml:LineString></gml:LineStringMember></gml:MultiLineString>',4));

INSERT INTO gml_linetest VALUES (2, ST_MLineFromGML('<gml:MultiLineString
            srsName="EPSG:4326" srsDimension="3"><gml:LineStringMember>
            <gml:LineString srsName="EPSG:4326" srsDimension="3"><gml:posList
            dimension="3">-110.449999933 45.2559999343 10 -109.47999994
            46.4600005499 10 -109.86000008 43.8400000201 20</gml:posList>
            </gml:LineString></gml:LineStringMember><gml:LineStringMember>
            <gml:LineString srsName="EPSG:4326" srsDimension="3"><gml:posList
            dimension="3">-99.45 33.256 10 -99.48 36.46 10 -99.86 33.84 20
            </gml:posList></gml:LineString></gml:LineStringMember>
            </gml:MultiLineString>'));

INSERT INTO gml_linetest VALUES (3, ST_MLineFromGML('<gml:MultiLineString
            srsName="EPSG:4326" srsDimension="4"><gml:LineStringMember>
            <gml:LineString srsName="EPSG:4326" srsDimension="4"><gml:posList
            dimension="4">-110.449999933 45.2559999343 10 54 -109.47999994
            46.4600005499 10 58 -109.86000008 43.8400000201 20 64</gml:posList>
            </gml:LineString></gml:LineStringMember><gml:LineStringMember>
            <gml:LineString srsName="EPSG:4326" srsDimension="4"><gml:posList
            dimension="4">-99.45 33.256 10 54 -99.48 36.46 10 58 -99.86 33.84 20 64
            </gml:posList></gml:LineString></gml:LineStringMember>
            </gml:MultiLineString>'));
```

The first record specifies a spatial reference ID of 4 (WGS84) and a default dimension of 2. The second and third records contain Z and M measures and pass the spatial reference ID through the srsName attribute.

Output:

```
SELECT * FROM gml_linetest;

gid  1
ln1  4 MULTILINESTRING ((-110.449999933 45.2559999343, -109.47999994 46.4600000
     469, -109.86000008 43.8400000201),(-99.4499999329 33.2559999343, -99.47999
     99397 36.4600000469, -99.8600000805 33.8400000201))

gid  2
```

```
ln1  4 MULTILINESTRING Z ((-110.449999933 45.2559999343 10, -109.47999994 46.46
     00005499 10, -109.86000008 43.8400000201 20),(-99.4499999329 33.2559999343
      10, -99.4799999397 36.4600000469 10, -99.8600000805 33.8400000201 20))

gid  3
ln1  4 MULTILINESTRING ZM ((-110.449999933 45.2559999343 10 54, -109.47999994 4
     6.4600005499 10 58, -109.86000008 43.8400000201 20 64),(-99.4499999329 33.
     2559999343 10 54, -99.4799999397 36.4600000469 10 58, -99.8600000805 33.84
     00000201 20 64))

3 row(s) retrieved.
```

ST_MLineFromKML()

ST_MLineFromKML() takes a KML MultiLineString string and an optional spatial reference ID and returns a multipart polyline object. A KML MultiLineString string can contain the KML shape attributes <extrude>, <tessellate>, and <altitudeMode>, but they are ignored in the ST_MultiLineString representation.

Syntax

```
ST_MLineFromKML(kmlstring lvarchar)
ST_MLineFromKML(kmlstring lvarchar, SRID integer)
```

Return Type

ST_MultiLineString

Example

```
EXECUTE FUNCTION ST_MLineFromKML(<MultiGeometry>
    <LineString>
      <coordinates>
        -122.4425587930444,37.80666418607323,0
        -122.4428379594768,37.80663578323093,0
      </coordinates>
    </LineString>
    <LineString>
      <coordinates>
        -122.4425509770566,37.80662588061205,0
        -122.4428340530617,37.8065999493009,0
      </coordinates>
    </LineString>
  </MultiGeometry>,4);
```

SE_MLineFromShape()

SE_MLineFromShape() creates an ST_MultiLineString from a shape of type polyline and a spatial reference ID. A polyline with only one part is appropriate as an ST_LineString (see "SE_LineFromShape()" on page 8-82) and a polyline with multiple parts is appropriate as an ST_MultiLineString.

Syntax

```
SE_MLineFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_MultiLineString

Example

The **waterways** table is created with the **ID** and **name** columns that identify each stream and river system stored in the table. The **water** column is an ST_MultiLineString because the river and stream systems are often an aggregate of several linestrings:

```
CREATE TABLE waterways (id     integer,
                        name   varchar(128),
                        water  ST_MultiLineString);
```

This code fragment populates the **waterways** table with a unique ID, a name, and a **water** multilinestring:

```
/* Create the SQL insert statement to populate the waterways
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO waterways (id,name,water) "
            "VALUES(?, ?, ST_MlineFromShape(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &id, 0, &pcbvalue1);

    /* Bind the name to the second parameter. */
    pcbvalue2 = name_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, name_len, 0,
                           name, name_len, &pcbvalue2);

    /* Bind the water geometry to the third parameter. */
    pcbvalue3 = water_shape_len;
    rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, water_shape_len, 0,
                           water_shape_buf, water_shape_len, &pcbvalue3);
    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_MLineFromText()

ST_MLineFromText() takes a well-known text representation of type ST_MultiLineString and a spatial reference ID and returns an ST_MultiLineString.

Syntax

```
ST_MLineFromText(wkt lvarchar, SRID integer)
```

Return Type

ST_MultiLineString

Example

The **mlinestring_test** is created with the **gid** SMALLINT column that uniquely identifies the row and the **ml1** ST_MultiLineString column:

```
CREATE TABLE mlinestring_test (gid smallint,
                               ml1 ST_MultiLineString);
```

The following INSERT statement inserts the ST_MultiLineString with the **ST_MLineFromText()** function:

```
INSERT INTO mlinestring_test VALUES(
   1,
   ST_MLineFromText('multilinestring((10.01 20.03,10.52
40.11,30.29 41.56,31.78 10.74),(20.93 20.81, 21.52 40.10))', 1000)
)
```

ST_MLineFromWKB()

ST_MLineFromWKB() creates an ST_MultiLineString from a well-known binary representation of type ST_MultiLineString and a spatial reference ID.

Syntax

```
ST_MLineFromWKB(WKB lvarchar, SRID integer)
```

Return Type

ST_MultiLineString

Example

The **waterways** table is created with the **ID** and **name** columns that identify each stream and river system stored in the table. The **water** column is an ST_MultiLineString because the river and stream systems are often an aggregate of several linestrings:

```
CREATE TABLE waterways (id     integer,
                        name   varchar(128),
                        water  ST_MultiLineString);
```

This code fragment populates the **waterways** table with a unique ID, a name, and a **water** ST_MultiLineString:

```
/* Create the SQL insert statement to populate the waterways
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO waterways (id,name,water) "
            "VALUES(?, ?, ST_MlineFromWKB(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &id, 0, &pcbvalue1);

    /* Bind the name to the second parameter. */
    pcbvalue2 = name_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, name_len, 0,
                           name, name_len, &pcbvalue2);

    /* Bind the water geometry to the third parameter. */
    pcbvalue3 = water_wkb_len;
    rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, water_wkb_len, 0,
                           water_wkb_buf, water_wkb_len, &pcbvalue3);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

SE_Mmax() and SE_Mmin()

SE_Mmax() and **SE_Mmin()** return the maximum and minimum measure values of a geometry.

Syntax

```
SE_Mmax(ST_Geometry)

SE_Mmin(ST_Geometry)
```

Return Type

DOUBLE PRECISION

ST_MPointFromGML()

ST_MPointFromGML() takes a GML2 or GML3 string representation of an ST_MultiPoint and an optional spatial reference ID and returns a polygon object.

Syntax

```
ST_MPointFromGML(gmlstring lvarchar)
ST_MPointFromGML(gmlstring lvarchar, SRID integer)
```

Return Type

ST_MultiPoint

Example

The **gml_pointtest** table is created with the SMALLINT column **gid** and the ST_MultiPoint column **mpt1**:

```
CREATE TABLE gml_pointtest(gid smallint, mpt1 ST_MultiPoint);

INSERT INTO gml_pointtest VALUES (1, ST_MPointFromGML('<gml:MultiPoint>
            <gml:PointMember><gml:Point><gml:pos>-110.45 45.256</gml:pos>
            </gml:Point></gml:PointMember><gml:PointMember><gml:Point>
            <gml:pos>-99.45 33.256</gml:pos></gml:Point></gml:PointMember>
            </gml:MultiPoint>',4));

INSERT INTO gml_pointtest VALUES (2, ST_MPointFromGML('<gml:MultiPoint
           srsName="EPSG:4326" srsDimension="3"><gml:PointMember><gml:Point
           srsName="EPSG:4326" srsDimension="3"><gml:pos>-110.449999933
           45.2559999343 10</gml:pos></gml:Point></gml:PointMember>
           <gml:PointMember><gml:Point srsName="EPSG:4326" srsDimension="3">
           <gml:pos>-99.86 33.84 20</gml:pos></gml:Point></gml:PointMember>
           </gml:MultiPoint>'));

INSERT INTO gml_pointtest VALUES (3, ST_MPointFromGML('<gml:MultiPoint
          srsName="EPSG:4326" srsDimension="4"><gml:PointMember><gml:Point
          srsName="EPSG:4326" srsDimension="4"><gml:pos>-109.47999994
          46.4600005499 10 58</gml:pos></gml:Point></gml:PointMember>
          <gml:PointMember><gml:Point srsName="EPSG:4326" srsDimension="4">
          <gml:pos>-99.45 33.256 10 54</gml:pos></gml:Point></gml:PointMember>
          </gml:MultiPoint>'));
```

The first record specifies a spatial reference ID of 4 (WGS84) and a default dimension of 2. The second and third records contain Z and M measures and pass the spatial reference ID through the srsName attribute.

Output:

```
SELECT * FROM gml_pointtest;

gid   1
mpt1  4 MULTIPOINT (-110.449999933 45.2559999343, -99.4499999329 33.2559999343)


gid   2
mpt1  4 MULTIPOINT Z (-110.449999933 45.2559999343 10, -99.8600000805 33.840000
      0201 20)

gid   3
mpt1  4 MULTIPOINT ZM (-109.47999994 46.4600005499 10 58, -99.4499999329 33.255
      9999343 10 54)

3 row(s) retrieved.
```

ST_MPointFromKML()

ST_MPointFromKML() takes a KML MultiGeometry and Point combination and an optional spatial reference ID and returns a multipoint object. A Point string can contain the elements of <coordinates>, <extrude>, <tessellate>, and <altitudeMode>, but they are ignored.

Syntax

```
ST_MPointFromKML(kmlstring lvarchar)
ST_MPointFromKML(kmlstring lvarchar, SRID integer)
```

Return Type

ST_MultiPoint

Example

```
EXECUTE FUNCTION ST_MPointFromKML('<MultiGeometry><Point><coordinates>
             -122.365662,37.826988, 0</coordinates></Point><Point>
             <coordinates>-122.365038,37.82655,0</coordinates>
             </Point></MultiGeometry>',4);
```

Output:

```
4 MULTIPOINT Z (-122.365662056 37.8269879529 0, -122.36503794
                37.8265500822 0)
```

SE_MPointFromShape()

SE_MPointFromShape() takes a shape of type multipoint and a spatial reference ID to return an ST_MultiPoint.

Syntax

```
SE_MPointFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_MultiPoint

Example

The **species_sitings** table is created with three columns. The **species** and **genus** columns uniquely identify each row, while the **sitings** ST_MultiPoint stores the locations of the species sitings:

```
CREATE TABLE species_sitings (species  varchar(32),
                              genus    varchar(32),
                              sitings  ST_MultiPoint);
```

This code fragment populates the **species_sitings** table:

```
/* Create the SQL insert statement to populate the species_sitings
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO species_sitings (species,genus,sitings) "
            "VALUES(?, ?, SE_MpointFromShape(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the species to the first parameter. */
    pcbvalue1 = species_len;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, species_len, 0,
                           species, species_len, &pcbvalue1);

    /* Bind the genus to the second parameter. */
    pcbvalue2 = genus_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, genus_len, 0,
                           genus, genus_len, &pcbvalue2);

    /* Bind the sitings geometry to the third parameter. */
    pcbvalue3 = sitings_shape_len;
    rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, sitings_shape_len, 0,
                           sitings_shape_buf, sitings_shape_len, &pcbvalue3);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_MPointFromText()

ST_MPointFromText() creates an ST_MultiPoint from a well-known text representation of type ST_MultiPoint and a spatial reference ID.

Syntax

```
ST_MPointFromText(WKT lvarchar, SRID integer)
```

Return Type

ST_MultiPoint

Example

The **multipoint_test** table is created with the single ST_MultiPoint **mpt1** column:

```
CREATE TABLE multipoint_test (gid  smallint,
                              mpt1 ST_MultiPoint);
```

The following INSERT statement inserts a multipoint into the **mpt1** column using the **ST_MPointFromText()** function:

```
INSERT INTO multipoint_test VALUES(
    1,
    ST_MPointFromText('multipoint(10.01 20.03,10.52 40.11,30.29
41.56,31.78 10.74)',1000)
);
```

ST_MPointFromWKB()

ST_MPointFromWKB() creates an ST_MultiPoint from a well-known binary representation of type ST_MultiPoint and a spatial reference ID.

Syntax

```
ST_MPointFromWKB (WKB lvarchar, SRID integer)
```

Return Type

ST_MultiPoint

Example

The **species_sitings** table is created with three columns. The **species** and **genus** columns uniquely identify each row, while the **sitings** ST_MultiPoint stores the locations of the species sightings:

```
CREATE TABLE species_sitings (species  varchar(32),
                              genus    varchar(32),
                              sitings  ST_MultiPoint);
```

This code fragment populates the **species_sitings** table:

```
/* Create the SQL insert statement to populate the species_sitings
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO species_sitings (species,genus,sitings) "
            "VALUES(?, ?, ST_MpointFromWKB(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the species to the first parameter. */
    pcbvalue1 = species_len;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, species_len, 0,
                           species, species_len, &pcbvalue1);

    /* Bind the genus to the second parameter. */
    pcbvalue2 = genus_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, genus_len, 0,
                           genus, genus_len, &pcbvalue2);

    /* Bind the sitings geometry to the third parameter. */
    pcbvalue3 = sitings_wkb_len;
    rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, sitings_wkb_len, 0,
                           sitings_wkb_buf, sitings_wkb_len, &pcbvalue3);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_MPolyFromGML()

ST_MPolyFromGML() takes a GML2 or GML3 string representation of an ST_MultiPoly and an optional spatial reference ID and returns a multipart polygon object.

Syntax

```
ST_MPolyFromGML(gmlstring lvarchar)
ST_MPolyFromGML(gmlstring lvarchar, SRID integer)
```

Return Type

ST_MultiPolygon

Example

The **test_multipoly** table is created with the INTEGER column **id** and the ST_MultiPolygon column **geom**:

```
CREATE TABLE test_multipoly (id INTEGER, geom ST_MultiPolygon);

INSERT INTO test_multipoly VALUES(1,ST_MPolyFromGML
     ('<gml:MultiPolygon srsName="EPSG:4326" srsDimension="2">
     <gml:PolygonMember><gml:Polygon srsName="EPSG:4326" srsDimension="2">
     <gml:exterior><gml:LinearRing><gml:posList dimension="2">
     -94.36  32.49 -92.66 32.69 -92.15 32.46 -93.09 33.08 -93.37 33.19
     -94.36 32.49</gml:posList></gml:LinearRing>
     </gml:exterior></gml:Polygon></gml:PolygonMember>
     <gml:PolygonMember><gml:Polygon srsName="EPSG:4326" srsDimension="2">
     <gml:exterior><gml:LinearRing><gml:posList dimension="2">
     -84.36 32.49 -82.66 32.69 -82.15 32.46 -83.09 33.08 -83.37 33.19
     -84.36 32.49</gml:posList></gml:LinearRing></gml:exterior>
     < /gml:Polygon></gml:PolygonMember></gml:MultiPolygon>',4));
```

This record specifies a spatial reference ID of 4 (WGS84) and both member polygons have a dimension of 2 and six sides.

Output:

```
SELECT * FROM test_multipoly;

id    1
geom  4 MULTIPOLYGON (((-94.3600000805 32.4900000536, -92.6599999799
                      32.6899999866, -92.1500000335 32.4600000469,
                      -93.0900000201 33.0800000738, -93.3700000268
                      33.1899999866, -94.3600000805 32.4900000536)),
                      ((-84.3600000805 32.4900000536, -82.6599999799
                      32.6899999866, -82.1500000335 32.4600000469,
                      -83.0900000201 33.0800000738, -83.3700000268
                      33.1899999866, -84.3600000805 32.4900000536)))
```

ST_MPolyFromKML()

ST_MPolyFromKML() takes a MultiGeometry and Polygon combination and an optional spatial reference ID and returns a polygon object.

Syntax

```
ST_MPolyFromKML(kmlstring lvarchar)
ST_MPolyFromKML(kmlstring lvarchar, SRID integer)
```

Return Type

ST_MultiPolygon

Example

The **test_multipoly** table is created with the INTEGER column **id** and the ST_MultiPolygon column **geom**:

```
CREATE TABLE test_multipoly (id INTEGER, geom ST_MultiPolygon);

INSERT INTO test_multipoly VALUES(1,ST_MPolyFromKML('<MultiGeometry>
              <Polygon><outerBoundaryIs><LinearRing><coordinates>
              -94.36,32.49 -92.65,32.68 -92.15,32.46 -93.09,33.08
              -93.37,33.18 -94.36,32.49</coordinates>
             </LinearRing></outerBoundaryIs></Polygon><Polygon>
             <outerBoundaryIs><LinearRing><coordinates>-84.36,32.49
             -82.65,32.68 -82.15,32.46 -83.09,33.08 -83.37,33.18
             -84.3600000805,32.4900000536</coordinates>
             </LinearRing></outerBoundaryIs></Polygon><
             /MultiGeometry>',4));
```

This record specifies a spatial reference ID of 4 (WGS84) and both member polygons have two dimensions and six sides.

Output:

```
SELECT * FROM test_multipoly;

id    1
geom  4 MULTIPOLYGON (((-94.3600000805 32.4900000536, -92.6599999799
        32.6899999866, -92.1500000335 32.4600000469, -93.0900000201
        33.0800000738, -93.3700000268 33.1899999866, -94.3600000805
        32.4900000536)),((-84.3600000805 32.4900000536, -82.6599999799
        32.6899999866, -82.1500000335 32.4600000469, -83.0900000201
        33.0800000738, -83.3700000268 33.1899999866, -84.3600000805
        32.4900000536)))
```

SE_MPolyFromShape()

SE_MPolyFromShape() takes a shape of type polygon and a spatial reference ID to return an ST_MultiPolygon.

Syntax

```
SE_MPolyFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_MultiPolygon

Example

The **lots** table stores the **lot_id**, which uniquely identifies each lot, and the **lot** multipolygon that contains the lot line geometry:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

This code fragment populates the **lots** table:

```
/* Create the SQL insert statement to populate the lots table.
     * The question marks are parameter markers that indicate the
     * column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO lots (lot_id,lot)"
            "VALUES(?, SE_MpolyFromShape(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the lot_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &lot_id, 0, &pcbvalue1);

    /* Bind the lot geometry to the second parameter. */
    pcbvalue2 = lot_shape_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, lot_shape_len, 0,
                           lot_shape_buf, lot_shape_len, &pcbvalue2);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_MPolyFromText()

ST_MPolyFromText() takes a well-known text representation of type ST_MultiPolygon and a spatial reference ID and returns an ST_MultiPolygon.

Syntax

```
ST_MPolyFromText(WKT lvarchar, SRID integer)
```

Return Type

ST_MultiPolygon

Example

The **multipolygon_test** table is created with the single ST_MultiPolygon **mpl1** column:

```
CREATE TABLE multipolygon_test (mpl1 ST_MultiPolygon);
```

The following INSERT statement inserts an ST_MultiPolygon into the **mp11** column using the **ST_MPolyFromText()** function:

```
INSERT INTO multipolygon_test VALUES(
   ST_MPolyFromText('multipolygon(((10.01 20.03,10.52 40.11,30.29
41.56,31.78 10.74,10.01 20.03),(21.23 15.74,21.34 35.21,28.94
35.35,29.02 16.83,21.23 15.74)),((40.91 10.92,40.56 20.19,50.01
21.12,51.34 9.81,40.91 10.92)))',1000)
);
```

ST_MPolyFromWKB()

ST_MPolyFromWKB() takes a well-known binary representation of type ST_MultiPolygon and a spatial reference ID to return an ST_MultiPolygon.

Syntax

```
ST_MPolyFromWKB (WKB lvarchar, SRID integer)
```

Return Type

ST_MultiPolygon

Example

The **lots** table stores the **lot_id**, which uniquely identifies each lot, and the lot multipolygon that contains the lot line geometry:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

This code fragment populates the **lots** table:

```
/* Create the SQL insert statement to populate the lots table.
     * The question marks are parameter markers that indicate the
     * column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO lots (lot_id, lot) "
            "VALUES(?, ST_MpolyFromWKB(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the lot_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &lot_id, 0, &pcbvalue1);

    /* Bind the lot geometry to the second parameter. */
    pcbvalue2 = lot_wkb_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, lot_wkb_len, 0,
                           lot_wkb_buf, lot_wkb_len, &pcbvalue2);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

SE_Nearest() and SE_NearestBbox()

The **SE_Nearest()** function allows you to query for the nearest-neighbors to a specified geometry. The **SE_Nearest()** function uses an R-tree index, which you must create if it does not already exist. Query results are returned in order of increasing distance from the query object. The distance is measured using the same algorithm as used by the ST_Distance() function.

The **SE_NearestBBox()** function is similar to **SE_Nearest()**, but distances are measured between objects' bounding boxes (envelopes). Because this is a simpler calculation, **SE_NearestBBox()** executes more quickly, but may return objects in a different order, depending on the actual shape of the objects.

These functions can only be used in the WHERE clause of a query.

Important: To use the **SE_Nearest()** and **SE_NearestBBox()** functions, you must be using IBM Informix Dynamic Server, Version 9.3 or later.

Important: You must create an R-tree index on the geometry column on which you want to perform nearest-neighbor queries.

Syntax

```
SE_Nearest (g1 ST_Geometry, g2 ST_Geometry)

SE_NearestBbox (g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The **cities** table contains the names and locations of world cities:

```
CREATE TABLE cities (name varchar(255),
                     locn ST_Point);
```

Populate this table with data from a DB-Access load file, **cities.load**, (this data file is included with the IBM Informix Spatial DataBlade Module as part of this example; it contains the names and locations of approximately 300 world cities), as the example shows:

```
LOAD FROM cities.load INSERT INTO cities;
```

Create an R-tree index on the **locn** column (this is required by the **SE_Nearest()** function):

```
CREATE INDEX cities_idx ON cities (locn ST_Geometry_ops) USING RTREE;

UPDATE STATISTICS FOR TABLE cities (locn);
```

Now search for the five cities nearest London:

```
SELECT FIRST 5 name FROM cities
  WHERE SE_Nearest(locn, '0 point(0 51)');


name  London

name  Birmingham

name  Paris
```

```
name  Nantes

name  Amsterdam
```

Warning: Using a fragmented R-tree index for nearest-neighbor queries raises an error. Results will not be returned in nearest distance order because the query is executed on each separate index fragment, and results from each fragment are combined in an unspecified order.

ST_NumGeometries()

ST_NumGeometries() takes a ST_GeomCollection (ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon) and returns the number of geometries in the collection.

Syntax

```
ST_NumGeometries(mpt1 ST_MultiPoint)
ST_NumGeometries(mln1 ST_MultiLineString)
ST_NumGeometries(mpl1 ST_MultiPolygon)
```

Return Type

INTEGER

Example

The city engineer needs to know the number of distinct buildings associated with each building footprint.

The building footprints are stored in the **buildingfootprints** table that was created with the following CREATE TABLE statement:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

The query lists the **building_id** that uniquely identifies each building and the number of buildings in each footprint with the **ST_NumGeometries()** function:

```
SELECT building_id, ST_NumGeometries(footprint) no_of_buildings
   FROM buildingfootprints;

building_id no_of_buildings

        506               1
        543               1
       1208               1
        178               1
```

ST_NumInteriorRing()

ST_NumInteriorRing() takes an ST_Polygon and returns the number of its interior rings.

Syntax

```
ST_NumInteriorRing(pl1 ST_Polygon)
```

Return Type

INTEGER

Example

An ornithologist studying a bird population on several South Sea islands wants to identify which islands contain one or more lakes, because the bird species of interest feeds only in freshwater lakes.

The **ID** and **name** columns of the **islands** table identifies each island, while the **land** ST_Polygon column stores the island geometry:

```
CREATE TABLE islands (id    integer,
                      name  varchar(32),
                      land  ST_Polygon);
```

Because interior rings represent the lakes, the **ST_NumInteriorRing()** function lists only those islands that have at least one interior ring:

```
SELECT name
   FROM islands
   WHERE ST_NumInteriorRing(land) > 0;
```

ST_NumPoints()

ST_NumPoints() returns the number of points in an ST_Geometry.

Syntax

```
ST_NumPoints(g1 ST_Geometry)
```

Return Type

INTEGER

Example

The **numpoints_test** table has two columns: **geotype**, a VARCHAR column that contains a description of the type of geometry; and **gl**, an ST_Geometry type that contains the geometry itself:

```
CREATE TABLE numpoints_test (geotype varchar(12),
                             g1      ST_Geometry);
```

The following INSERT statements insert a point, a linestring, and a polygon:

```
INSERT INTO numpoints_test VALUES(
   'point',
   ST_PointFromText('point (10.02 20.01)',1000)
);

INSERT INTO numpoints_test VALUES(
   'linestring',
   ST_LineFromText('linestring (10.02 20.01, 23.73 21.92)',1000)
);

INSERT INTO numpoints_test VALUES(
   'polygon',
   ST_PolyFromText('polygon ((10.02 20.01, 23.73 21.92, 24.51
12.98,  11.64 13.42, 10.02 20.01))',1000)
);
```

The query lists the geometry type and the number of points in each:

```
SELECT geotype, ST_NumPoints(g1) Number_of_points
   FROM numpoints_test;

geotype      number_of_points

point                       1
linestring                  2
polygon                     5
```

SE_OutOfRowSize()

SE_OutOfRowSize() returns the size of the out-of-row portion of a geometry. Geometries which are larger than 930 bytes (for example, polygons with many vertices) have an in-row component and an out-of-row component; the out-of-row component is stored in an sbspace.

If a geometry has no out-of-row component, **SE_OutOfRowSize()** returns `0`.

You can use this function to obtain an estimate of the amount of diskspace consumed by one or more geometries. However, this function does not account for dbspace and sbspace overhead, so cannot be used to obtain an exact total.

Syntax

```
SE_OutOfRowSize(ST_Geometry)
```

Return Type

INTEGER

See Also

"SE_InRowSize()" on page 8-64

"SE_TotalSize()" on page 8-145

ST_Overlaps()

ST_Overlaps() takes two ST_Geometry objects and returns `t` (TRUE) if the intersection of the objects results in an ST_Geometry object of the same dimension but not equal to either source object; otherwise, it returns `f` (FALSE).

Syntax

```
ST_Overlaps(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The county supervisor needs a list of hazardous waste sites whose five-mile radius overlaps sensitive areas.

The **sensitive_areas** table contains several columns that describe the threatened institutions in addition to the **zone** column, which stores the institution ST_Polygon geometries:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The **hazardous_sites** table stores the identity of the sites in the **site_id** and **name** columns, while the actual geographic location of each site is stored in the **location** point column:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The **sensitive_areas** and **hazardous_sites** tables are joined by the **ST_Overlaps()** function, which returns `t` (TRUE) for all **sensitive_areas** rows whose **zone** polygons overlap the buffered five-mile radius of the **hazardous_sites** location points:

```
SELECT hs.name hazardous_site, sa.name sensitive_area
   FROM hazardous_sites hs, sensitive_areas sa
   WHERE ST_Overlaps(ST_Buffer(hs.location,(26400)),sa.zone);


hazardous_site  Landmark Industrial
sensitive_area  Johnson County Hospital

hazardous_site  Landmark Industrial
sensitive_area  Summerhill Elementary School
```

Figure 8-10 shows that the hospital and the school overlap the five-mile radius of the county's two hazardous waste sites, while the nursing home does not.

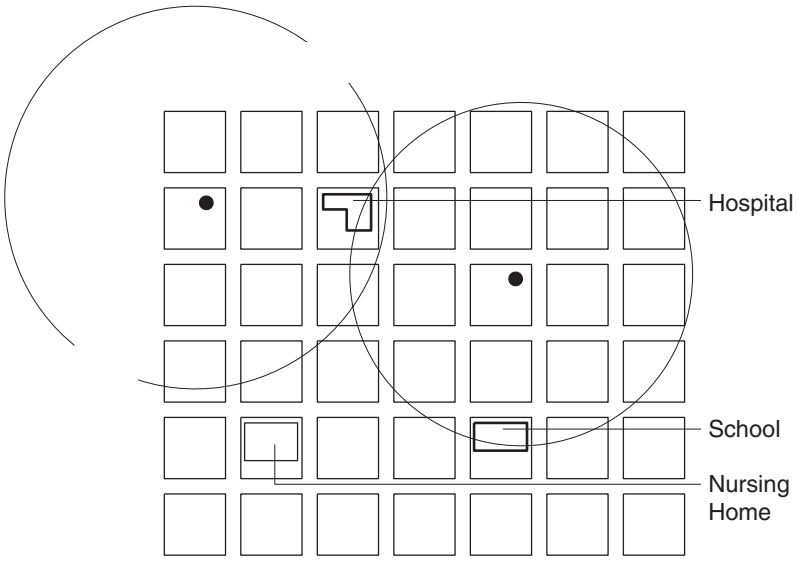


Figure 8-10. Using the ST_Overlaps() Function

SE_ParamGet() function

SE_ParamGet() with no arguments returns the values of all parameters. Calling SE_ParamGet() with a parameter name in quotes returns the current value of the named parameter.

Syntax

```
SE_ParamSet(name lvarchar) returns lvarchar
SE_ParamSet() returns lvarchar
```

Return Type

LVARCHAR

Example

```
execute function SE_ParamGet();
execute function SE_ParamGet('MemMode');
```

SE_ParamSet() function

SE_ParamSet() with no arguments returns usage information. Calling SE_ParamSet() with a parameter name in quotes and a value sets the parameter to the specified value. The new value is returned. Parameter names are not case-sensitive. Quotes are not required for numeric values.

Syntax

```
SE_ParamSet(name lvarchar, value lvarchar) returns lvarchar
SE_ParamSet() returns lvarchar
```

Return Type

LVARCHAR

Example

```
execute function SE_ParamSet();
execute function SE_ParamSet('memmode', '0');
execute function SE_ParamSet('MemMode', 1);
```

ST_Perimeter()

ST_Perimeter() returns the perimeter of a polygon or multipolygon.

Syntax

```
ST_Perimeter(s ST_Polygon)
ST_Perimeter(ms ST_MultiPolygon)
```

Return Type

DOUBLE PRECISION

Example

An ecologist studying shoreline birds needs to determine the shoreline for the lakes within a particular area. The lakes are stored as ST_MultiPolygon type in the **waterbodies** table, created with the following CREATE TABLE statement:

```
CREATE TABLE WATERBODIES (wbid integer, waterbody ST_MultiPolygon);
```

In the following SELECT statement, the **ST_Perimeter()** function returns the perimeter surrounding each body of water, while the SUM operator aggregates the perimeters to return their total:

```
SELECT SUM(ST_Perimeter(waterbody)) FROM waterbodies;
```

SE_PerpendicularPoint()

SE_PerpendicularPoint() finds the perpendicular projection of a point on to the nearest segment of a linestring or multilinestring. If two or more such perpendicular projected points are equidistant from the input point, they are all returned. If no perpendicular point can be constructed, an empty point is returned.

If the input linestring has Z values or measures, the Z value or measure of the perpendicular point are computed by linear interpolation between the adjacent vertices.

Syntax

```
SE_PerpendicularPoint(ST_LineString, ST_Point)

SE_PerpendicularPoint(ST_MultiLineString, ST_Point)
```

Return Type

ST_MultiPoint

Example

```
CREATE TABLE linestring_test (line ST_LineString);

-- Create a U-shaped linestring:
INSERT INTO linestring_test VALUES (
   ST_LineFromText('linestring z (0 10 1, 0 0 3, 10 0 5, 10 10 7)', 0)
);


-- Perpendicular point is coincident with the input point,
-- on the base of the U:
SELECT SE_PerpendicularPoint(line, ST_PointFromText('point(5 0)', 0))
  FROM linestring_test;

(expression)  0 MULTIPOINT Z (5 0 4)


-- Perpendicular points are located on all three segments of the U:
SELECT SE_PerpendicularPoint(line, ST_PointFromText('point(5 5)', 0))
  FROM linestring_test;

(expression)  0 MULTIPOINT Z (0 5 2, 5 0 4, 10 5 6)


-- Perpendicular points are located at the endpoints of the U:
SELECT SE_PerpendicularPoint(line, ST_PointFromText('point(5 10)', 0))
  FROM linestring_test;

(expression)  0 MULTIPOINT Z (0 10 1, 10 10 7)


-- Perpendicular point is on the base of the U:
SELECT SE_PerpendicularPoint(line, ST_PointFromText('point(5 15)', 0))
  FROM linestring_test;

(expression)  0 MULTIPOINT Z (5 0 4)


-- No perpendicular point can be constructed:
SELECT SE_PerpendicularPoint(line, ST_PointFromText('point(15 15)', 0))
  FROM linestring_test;

(expression)  0 POINT EMPTY
```

ST_Point()

ST_Point() returns an ST_Point, given an X-coordinate, Y-coordinate, and spatial reference ID.

Syntax

```
ST_Point(X double precision, Y double precision, SRID integer)
```

Return Type

ST_Point

Example

The following CREATE TABLE statement creates the **point_test** table, which has a single point column, **pt1**:

```
CREATE TABLE point_test (pt1 ST_Point);
```

The **ST_Point()** function converts the point coordinates into an ST_Point geometry before the INSERT statement inserts it into the **pt1** column:

```
INSERT INTO point_test VALUES(
   ST_Point(10.01,20.03,1000)
);
```

ST_PointFromGML()

ST_PointFromGML() takes a GML2 or GML3 string representation of an ST_Point and an optional spatial reference ID and returns a point object.

Syntax

```
ST_PointFromGML(gmlstring lvarchar)
ST_PointFromGML(gmlstring lvarchar, SRID integer)
```

Return Type

ST_Point

Example

The **point_t** table contains the gid INTEGER column, which uniquely identifies each row, the pdesc column which describes the point, and the p1 column which stores the point. In this example, GML3 is shown.

```
CREATE TABLE point_t (gid INTEGER, pdesc VARCHAR(30), p1 ST_Point);

INSERT INTO point_t VALUES(
1,
'This point is a simple XY point',
ST_PointFromGML('<gml:Point srsName="DEFAULT" srsDimension="2">
<gml:pos>10.02 20.01</gml:pos></gml:Point>',1000)) ;

INSERT INTO point_t VALUES(
2,
'This point is a XYZ point',
ST_PointFromGML('<gml:Point srsName="DEFAULT" srsDimension="3">
<gml:pos>10.02 20.01 5</gml:pos></gml:Point>',1000)) ;

INSERT INTO point_t VALUES(
3,
'This point is a XYM point',
ST_PointFromGML('<gml:Point srsName="DEFAULT" srsDimension="3">
<gml:pos>10.02 20.01 7</gml:pos></gml:Point>',1000));

INSERT INTO point_t VALUES(
4,
'This point is a XYZM point',
ST_PointFromGML('<gml:Point srsName="DEFAULT" srsDimension="4">
<gml:pos>10.02 20.01 5 7</gml:pos></gml:Point>',1000)) ;

INSERT INTO point_t VALUES(
5,
'This point is an empty point',
ST_PointFromGML('<gml:Point xsi:nil="true" srsName="UNKNOWN:0"
srsDimension="2"/>',1000));
```

ST_PointFromKML()

ST_PointFromKML() takes a KML Point string and an optional spatial reference ID and returns a point object. A Point string can contain the elements of <coordinates>, <extrude>, <tessellate>, and <altitudeMode>.

Syntax

```
ST_PointFromKML(kmlstring lvarchar)
ST_PointFromKML(kmlstring lvarchar, SRID integer)
```

Return Type

ST_Point

Example

```
EXECUTE FUNCTION ST_PointFromKML('<Point><coordinates>-122.44255879,37.80666418
                </coordinates></Point>',3);
```

Output:

```
   3 POINT (-122.44255879 37.80666418)
```

SE_PointFromShape()

SE_PointFromShape() creates an ST_Point from a shape of type point and a spatial reference ID.

Syntax

```
SE_PointFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_Point

Example

The hazardous sites are stored in the **hazardous_sites** table created with the CREATE TABLE statement that follows. The **location** column, defined as a point, stores a location that is the geographic center of each hazardous site:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The program fragment populates the **hazardous_sites** table:

```
/* Create the SQL insert statement to populate the hazardous_sites
     * table. The question marks are parameter markers that indicate
     * the column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO hazardous_sites (site_id, name, location) "
            "VALUES(?, ?, SE_PointFromShape(?, %d))", srid);

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the site_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &site_id, 0, &pcbvalue1);

    /* Bind the name to the second parameter. */
    pcbvalue2 = name_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, 0, 0,
                           name, 0, &pcbvalue2);

    /* Bind the location geometry to the third parameter. */
    pcbvalue3 = location_shape_len;
    rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, location_shape_len, 0,
                           location_shape_buf, location_shape_len, &pcbvalue3);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_PointFromText()

ST_PointFromText() takes a well-known text representation of type point and a spatial reference ID and returns a point.

Syntax

```
ST_PointFromText(WKT lvarchar, SRID integer)
```

Return Type

ST_Point

Example

The **point_test** table is created with the single ST_Point column **pt1**:

```
CREATE TABLE multipoint_test (gid  smallint,
                              mpt1 ST_MultiPoint);
```

The **ST_PointFromText()** function converts the point text coordinates to the ST_Point format before the INSERT statement inserts the point into the **pt1** column:

```
INSERT INTO multipoint_test VALUES(
    1,
    ST_MPointFromText('multipoint(10.01 20.03,10.52 40.11,30.29
41.56,31.78 10.74)',1000)
);
```

ST_PointFromWKB()

ST_PointFromWKB() takes a well-known binary representation of type ST_Point and a spatial reference ID to return an ST_Point.

Syntax

```
ST_PointFromWKB (WKB lvarchar, SRID integer)
```

Return Type

ST_Point

Example

The hazardous sites are stored in the **hazardous_sites** table created with the CREATE TABLE statement that follows. The **location** column, defined as an ST_Point, stores a location that is the geographic center of each hazardous site:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The program fragment populates the **hazardous_sites** table:

```
/* Create the SQL insert statement to populate the hazardous_sites
 * table. The question marks are parameter markers that indicate
 * the column values that will be inserted at run time. */
sprintf(sql_stmt,
        "INSERT INTO hazardous_sites (site_id, name, location) "
        "VALUES(?, ?, ST_PointFromWKB(?, %d))", srid);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

/* Bind the site_id to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                       SQL_INTEGER, 0, 0,
                       &site_id, 0, &pcbvalue1);

/* Bind the name to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                       SQL_CHAR, 0, 0,
                       name, 0, &pcbvalue2);

/* Bind the location geometry to the third parameter. */
pcbvalue3 = location_wkb_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
                       SQL_INFX_UDT_LVARCHAR, location_wkb_len, 0,
                       location_wkb_buf, location_wkb_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_PointN()

ST_PointN() takes an ST_LineString and an INTEGER index and returns a point that is the *n*th vertex in the ST_LineString's path. (The numbering of the vertices in the linestring starts with 1.)

Syntax

```
ST_PointN (ln1 ST_LineString, index integer)
```

Return Type

ST_Point

Example

The **pointn_test** table is created with the **gid** column, which uniquely identifies each row, and the **ln1** ST_LineString column:

```
CREATE TABLE pointn_test (gid integer,
                          ln1 ST_LineString);
```

The following INSERT statements insert two linestring values. The first linestring does not have Z coordinates or measures, while the second linestring has both:

```
INSERT INTO pointn_test VALUES(
   1,
   ST_LineFromText('linestring (10.02 20.01,23.73 21.92,30.10 40.23)',1000)
);

INSERT INTO pointn_test VALUES(
   2,
   ST_LineFromText('linestring zm (10.02 20.01 5.0 7.0,23.73 21.92
6.5 7.1,30.10 40.23 6.9 7.2)',1000)
);
```

The query lists the **gid** column and the second vertex of each linestring. The first row results in an ST_Point without a Z coordinate or measure, while the second row results in an ST_Point with a Z coordinate and a measure. The **ST_PointN()** function will also include a Z coordinate or measure value if they exist in the source linestring:

```
SELECT gid, ST_PointN(ln1,2) the_2nd_vertex
   FROM pointn_test;


gid             1
the_2nd_vertex  1000 POINT (23.73 21.92)

gid             2
the_2nd_vertex  1000 POINT ZM (23.73 21.92 6.5 7.1)
```

ST_PointOnSurface()

ST_PointOnSurface() takes an ST_Polygon or ST_MultiPolygon and returns an ST_Point guaranteed to lie on its surface.

Syntax

```
ST_PointOnSurface (pl1 ST_Polygon)
ST_PointOnSurface (mpl1 ST_MultiPolygon)
```

Return Type

ST_Point

Example

The city engineer wants to create a label point for each building footprint.

The **buildingfootprints** table that was created with the following CREATE TABLE statement stores the building footprints:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);
```

The **ST_PointOnSurface()** function generates a point that is guaranteed to be on the surface of the building footprints:

```
SELECT building_id, ST_PointOnSurface(footprint)
  FROM buildingfootprints;


building_id   506
(expression)  1000 POINT (12.5 49.5)

building_id   543
(expression)  1000 POINT (32 52.5)

building_id   1208
(expression)  1000 POINT (12.5 27.5)

building_id   178
(expression)  1000 POINT (32 30)
```

ST_PolyFromGML()

ST_PolyFromGML() takes a GML2 or GML3 string representation of an ST_Polygon and an optional spatial reference ID and returns a polygon object.

Syntax

```
ST_PolyFromGML(gmlstring lvarchar)
ST_PolyFromGML(gmlstring lvarchar, SRID integer)
```

Return Type

ST_Polygon

Example

The **test_poly** table is created with the INTEGER column **id** and the ST_Polygon column **geom**:

```
CREATE TABLE test_poly(id INTEGER, geom ST_Polygon);

INSERT INTO test_poly VALUES(1,ST_PolyFromGML('<gml:Polygon
             srsName="EPSG:4326" srsDimension="2">
             <gml:exterior><gml:LinearRing><gml:posList dimension="2">
             -84.36 32.49 -82.66 32.69 -82.15 32.46 -83.09 33.08 -83.37
             33.19 -84.36 32.49</gml:posList></gml:LinearRing>
             </gml:exterior>< /gml:Polygon>',4));
```

This record specifies a spatial reference id of 4 (WGS84) and represents a two-dimensional six-sided polygon.

Output:

```
SELECT * FROM test_poly;

id    1
geom  4 POLYGON ((-84.3600000805 32.4900000536, -82.6599999799
                  32.6899999866, -82.1500000335 32.4600000469,
                  -83.0900000201 33.0800000738, -83.3700000268
                  33.1899999866, -84.3600000805 32.4900000536))
```

ST_PolyFromKML()

ST_PolyFromKML() takes a KML Polygon string representation and an optional spatial reference ID and returns a polygon object.

Syntax

```
ST_PolyFromKML(kmlstring lvarchar)
ST_PolyFromKML(kmlstring lvarchar, SRID integer)
```

Return Type

ST_Polygon

Example

```
EXECUTE FUNCTION ST_PolyFromKML('<Polygon><outerBoundaryIs>
                <LinearRing>
                <coordinates>-122.365662,37.826988,0
                -122.365202,37.826302,0 -122.364581,37.82655,0
                -122.365038,37.827237,0 -122.365662,37.826988,0
                </coordinates></LinearRing></outerBoundaryIs>
                </Polygon>',4);
```

Output:

```
4 POLYGON Z ((-122.365662056 37.8269879529 0, -122.365202058
              37.8263019779 0, -122.364580958 37.8265500822 0,
              -122.36503794 37.827237063 0, -122.365662056
              37.8269879529 0))
```

SE_PolyFromShape()

SE_PolyFromShape() returns an ST_Polygon from a shape of type polygon and a spatial reference ID.

Syntax

```
SE_PolyFromShape(s1 lvarchar, SRID integer)
```

Return Type

ST_Polygon

Example

The **sensitive_areas** table contains several columns that describe the threatened institutions in addition to the **zone** column, which stores the institution polygon geometries:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The program fragment populates the **sensitive_areas** table. The question marks represent parameter markers for the **ID**, **name**, **size**, **type**, and **zone** values that will be retrieved at run time:

```
/* Create the SQL insert statement to populate the sensitive_areas
 * table. The question marks are parameter markers that indicate
 * the column values that will be inserted at run time. */
sprintf(sql_stmt,
        "INSERT INTO sensitive_areas (id, name, size, type, zone) "
        "VALUES(?, ?, ?, ?, SE_PolyFromShape(?, %d))", srid);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

/* Bind the id to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                       SQL_INTEGER, 0, 0,
                       &id, 0, &pcbvalue1);

/* Bind the name to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                       SQL_CHAR, 0, 0,
                       name, 0, &pcbvalue2);

/* Bind the size to the third parameter. */
pcbvalue3 = 0;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT,
                       SQL_REAL, 0, 0,
                       &size, 0, &pcbvalue3);

/* Bind the type to the fourth parameter. */
pcbvalue4 = type_len;
rc = SQLBindParameter (hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR,
                       SQL_VARCHAR, type_len, 0,
                       type, type_len, &pcbvalue4);

/* Bind the zone geometry to the fifth parameter. */
pcbvalue5 = zone_shape_len;
rc = SQLBindParameter (hstmt, 5, SQL_PARAM_INPUT, SQL_C_BINARY,
                       SQL_INFX_UDT_LVARCHAR, zone_shape_len, 0,
```

```
                        zone_shape_buf, zone_shape_len, &pcbvalue5);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_PolyFromText()

ST_PolyFromText() takes a well-known text representation of type ST_Polygon and a spatial reference ID and returns an ST_Polygon.

Syntax

```
ST_PolyFromText(wkt lvarchar, SRID integer)
```

Return Type

ST_Polygon

Example

The **polygon_test** table is created with the single polygon column:

```
CREATE TABLE polygon_test (pl1 ST_Polygon);
```

The following INSERT statement inserts a polygon into the **pl1** polygon column using the **ST_PolyFromText()** function:

```
INSERT INTO polygon_test VALUES(
    ST_PolyFromText('polygon((10.01 20.03,10.52 40.11,30.29
41.56,31.78 10.74,10.01 20.03))',1000)
);
```

ST_PolyFromWKB()

ST_PolyFromWKB() takes a well-known binary representation of type ST_Polygon and a spatial reference ID to return an ST_Polygon.

Syntax

```
ST_PolyFromWKB(wkb lvarchar, SRID integer)
```

Return Type

ST_Polygon

Example

The **sensitive_areas** table contains several columns that describe the threatened institutions in addition to the **zone** column, which stores the institution ST_Polygon geometries:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The program fragment populates the **sensitive_areas** table:

```
/* Create the SQL insert statement to populate the sensitive_areas
 * table. The question marks are parameter markers that indicate
 * the column values that will be inserted at run time. */
sprintf(sql_stmt,
        "INSERT INTO sensitive_areas (id, name, size, type, zone) "
        "VALUES(?, ?, ?, ?, ST_PolyFromWKB(?, %d))", srid);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

/* Bind the id to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                       SQL_INTEGER, 0, 0,
                       &id, 0, &pcbvalue1);

/* Bind the name to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                       SQL_CHAR, 0, 0,
                       name, 0, &pcbvalue2);

/* Bind the size to the third parameter. */
pcbvalue3 = 0;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT,
                       SQL_REAL, 0, 0,
                       &size, 0, &pcbvalue3);

/* Bind the type to the fourth parameter. */
pcbvalue4 = type_len;
rc = SQLBindParameter (hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR,
                       SQL_VARCHAR, type_len, 0,
                       type, type_len, &pcbvalue4);

/* Bind the zone geometry to the fifth parameter. */
pcbvalue5 = zone_wkb_len;
rc = SQLBindParameter (hstmt, 5, SQL_PARAM_INPUT, SQL_C_BINARY,
                       SQL_INFX_UDT_LVARCHAR, zone_wkb_len, 0,
                       zone_wkb_buf, zone_wkb_len, &pcbvalue5);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_Polygon()

ST_Polygon() generates an ST_Polygon from a ring (an ST_LineString that is both simple and closed).

Syntax

```
ST_Polygon (ln ST_LineString)
```

Return Type

ST_Polygon

Example

The following CREATE TABLE statement creates the **polygon_test** tables, which has a single column, **p1**:

```
CREATE TABLE polygon_test (p1 ST_polygon);
```

The INSERT statement converts a ring (an ST_LineString that is both closed and simple) into an ST_Polygon and inserts it into the **p1** column using the **ST_LineFromText()** function within the **ST_Polygon()** function:

```
INSERT INTO polygon_test VALUES(
   ST_Polygon(ST_LineFromText('linestring (10.01 20.03, 20.94
21.34, 35.93 10.04, 10.01 20.03)',1000))
);
```

ST_Relate()

ST_Relate() compares two geometries and returns 1 (TRUE) if the geometries meet the conditions specified by the DE-9IM pattern matrix string; otherwise, 0 (FALSE) is returned.

Syntax

```
ST_Relate(g1 ST_Geometry, g2 ST_Geometry, patternMatrix lvarchar)
```

Return Type

BOOLEAN

Example

A DE-9IM pattern matrix is a device for comparing geometries. There are several types of such matrices. For example, the *equals* pattern matrix will tell you if any two geometries are equal.

In this example, an equals pattern matrix, shown below, is read left to right, and top to bottom into the string (″T*F**FFF*″):

		b		
		Interior	Boundary	Exterior
	Interior	T	*	F
a	Boundary	*	*	F
	Exterior	F	F	*

Next, the table **relate_test** is created with the following CREATE TABLE statement:

```
CREATE TABLE relate_test (g1 ST_Geometry,
                          g2 ST_Geometry,
                          g3 ST_Geometry);
```

The following INSERT statements insert a sample subclass into the **relate_test** table:

```
INSERT INTO relate_test VALUES(
   ST_PointFromText('point (10.02 20.01)',1000),
   ST_PointFromText('point (10.02 20.01)',1000),
   ST_PointFromText('point (30.01 20.01)',1000)
);
```

The following SELECT statement and the corresponding result set lists the subclass name stored in the **geotype** column with the dimension of that geotype:

```
SELECT ST_Relate(g1,g2,"T*F**FFF*") equals,
       ST_Relate(g1,g3,"T*F**FFF*") not_equals
   FROM relate_test;

equals not_equals

     t          f
```

SE_Release()

The **SE_Release()** function returns a text string containing the installed version and release date of the IBM Informix Spatial DataBlade Module.

Syntax

```
SE_Release()
```

Return Type

A text string containing the installed version and release date

SE_ShapeToSQL()

SE_ShapeToSQL() constructs an ST_Geometry given its ESRI shape representation. The SRID of the St_Geometry is 0.

Syntax

```
SE_ShapeToSQL(ShapeGeometry lvarchar)
```

Return Type

ST_Geometry

Example

The following C code fragment contains ODBC functions that insert data into the **lots** table. The **lots** table was created with two columns: **lot_id**, which uniquely identifies each lot, and the **lot** polygon column, which contains the geometry of each lot:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The **SE_ShapeToSQL()** function converts shapes into ST_Geometry values. The INSERT statement contains parameter markers to accept the **lot_id** and the lot data, dynamically:

```
/* Create the SQL insert statement to populate the lots table.
     * The question marks are parameter markers that indicate the
     * column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO lots (lot_id, lot) "
            "VALUES(?, SE_ShapeToSQL(?))");

    /* Prepare the SQL statement for execution. */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the lot_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &lot_id, 0, &pcbvalue1);

    /* Bind the lot geometry the second parameter. */
    pcbvalue2 = lot_shape_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, lot_shape_len, 0,
                           lot_shape_buf, lot_shape_len, &pcbvalue2);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

SE_SpatialKey()

SE_SpatialKey() generates a *sort key* for an ST_Geometry. A sort key is a numeric value that can be used to sort spatial objects according to their proximity to one another.

The sort key is computed by applying the Hilbert space-filling curve algorithm to the center point of an object's bounding box.

Syntax

```
SE_SpatialKey(g1 ST_Geometry)
```

Return Type

A numeric sort key, as an INT8

Example

Create and populate a **cities** table containing the names and locations of world cities:

```
CREATE TABLE cities (name varchar(255),
                     locn ST_Point);

LOAD FROM cities.load INSERT INTO cities;
```

Create a clustered functional B-tree index. This rearranges the table data, placing it in spatial key sort order. Chapter 4, "Using R-Tree Indexes," on page 4-1, provides information about using indexes, in particular, R-tree indexes. For example:

```
CREATE CLUSTER INDEX cbt_idx ON cities (SE_SpatialKey(locn));
```

Create an R-tree index with the NO_SORT option:

```
CREATE INDEX locn_idx ON cities (locn ST_Geometry_ops)
  USING RTREE (BOTTOM_UP_BUILD='yes', NO_SORT='yes');
```

Drop the B-tree index; it is no longer needed:

```
DROP INDEX cbt_idx;
```

Important: The *BOTTOM_UP_BUILD* and *NO_SORT* options for building an R-tree index are only available with IBM Informix Dynamic Server, Version 9.21 and later.

ST_SRID()

ST_SRID() takes an ST_Geometry object and returns its spatial reference ID.

Syntax

```
ST_SRID(g1 ST_Geometry)
```

Return Type

INTEGER

Example

During the installation of the IBM Informix Spatial DataBlade Module, the **spatial_references** table is created by this CREATE TABLE statement:

```
CREATE TABLE sde.spatial_references
(
    srid               integer       NOT NULL,
    description        varchar(64),
    auth_name          varchar(255),
    auth_srid          integer,
    falsex             float         NOT NULL,
    falsey             float         NOT NULL,
    xyunits            float         NOT NULL,
    falsez             float         NOT NULL,
    zunits             float         NOT NULL,
    falsem             float         NOT NULL,
    munits             float         NOT NULL,
    srtext             char(2048)    NOT NULL,
  PRIMARY KEY (srid)
      CONSTRAINT sde.sp_ref_pk
);
```

Before you can create geometry and insert it into a table, you must enter the SRID of that geometry into the **spatial_references** table. This is a sample insert of a spatial reference system. The spatial reference system has an SRID value of 1, a false X, Y of (0,0), and its system units are 100. The Z coordinate and measure offsets are 0, while the Z coordinate and measure units are 1.

```
INSERT INTO spatial_references
     (srid, description, auth_name, auth_srid, falsex, falsey,
      xyunits, falsez, zunits, falsem, munits, srtext)
   VALUES (1, NULL, NULL, NULL, 0, 0, 100, 0, 1, 0, 1, 'UNKNOWN');
```

Important: Choose the parameters of a **spatial_references** table entry with care. See "The spatial_references Table" on page 1-4, for more information.

The following table is created for this example:

```
CREATE TABLE srid_test(g1 ST_Geometry);
```

In the next statement, an ST_Point geometry located at coordinate (10.01,50.76) is inserted into the geometry column **g1**. When the ST_Point geometry is created by the **ST_PointFromText()** function, it is assigned the SRID value of 1:

```
INSERT INTO srid_test VALUES(
   ST_PointFromText('point(10.01 50.76)',1000)
);
```

The **ST_SRID()** function returns the spatial reference ID of the geometry just entered:

```
SELECT ST_SRID(g1)
   FROM srid_test;

(expression)

        1000
```

SE_SRID_Authority()

SE_SRID_Authority() takes a spatial reference ID and returns the Authority Name and Authority SRID as a lvarchar string in the form AuthName:SRID. If the AuthName is null in the sde.spatial_references table for a given spatial reference ID, the srtext is returned.

Syntax

```
SE_SRID_Authority(SRID integer)
```

Return Type

lvarchar

Example

```
select SE_SRID_Authority(srid) from sde.spatial_references;

(expression)  UNKNOWN:0
(expression)  EPSG:4135
(expression)  EPSG:4267
(expression)  EPSG:4269
(expression)  EPSG:4326
(expression)  UNKNOWN:0
(expression)  UNKNOWN:0
(expression)  UNKNOWN:0
(expression)  AUTH_NAME:1234
(expression)  GEOGCS["GCS_Old_Hawaiian",
              DATUM["D_OldHawaiian",
              SPHEROID["Clarke_1866",6378206.4,294.9786982]],
              PRIMEM["Greenwich",0],UNIT["Degree",0.01745329 25199433]]

10 row(s) retrieved.
```

ST_StartPoint()

ST_StartPoint() returns the first point of a linestring.

Syntax

```
ST_StartPoint(ln1 ST_LineString)
```

Return Type

ST_Point

Example

The **startpoint_test** table is created with the **gid** INTEGER column, which uniquely identifies the rows of the table, and the **ln1** ST_LineString column:

```
CREATE TABLE startpoint_test (gid integer,
                              ln1 ST_LineString);
```

The following INSERT statements insert the ST_LineStrings into the **ln1** column. The first ST_LineString does not have Z coordinates or measures, while the second ST_LineString has both:

```
INSERT INTO startpoint_test VALUES(
   1,
   ST_LineFromText('linestring (10.02 20.01,23.73 21.92,30.10 40.23)', 1000)
);

INSERT INTO startpoint_test VALUES(
   2,
   ST_LineFromText('linestring zm (10.02 20.01 5.0 7.0,23.73 21.92
6.5 7.1,30.10 40.23 6.9 7.2)', 1000)
);
```

The **ST_StartPoint()** function extracts the first point of each ST_LineString. The first point in the list does not have a Z coordinate or a measure, while the second point has both because the source linestring does:

```
SELECT gid, ST_StartPoint(ln1) Startpoint
   FROM startpoint_test;


gid         1
startpoint  1000 POINT (10.02 20.01)

gid         2
startpoint  1000 POINT ZM (10.02 20.01 5 7)
```

See Also

"ST_EndPoint()" on page 8-43

ST_SymDifference()

ST_SymDifference() takes two ST_Geometry objects and returns a ST_Geometry object that is composed of the parts of the source objects that are not common to both.

Syntax

```
ST_SymDifference (g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

ST_Geometry

Example

For a special report, the county supervisor must determine the area of sensitive areas and five-mile hazardous site radii that do not intersect.

The **sensitive_areas** table contains several columns that describe the threatened institutions, in addition to the **zone** column, which stores the institutions' ST_Polygon geometries:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The **hazardous_sites** table stores the identity of the sites in the **site_id** and **name** columns, while the actual geographic location of each site is stored in the **location** point column:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The **ST_Buffer()** function generates a five-mile buffer surrounding the hazardous waste site locations. The **ST_SymDifference()** function returns the polygons of the buffered hazardous waste site polygons and the sensitive areas that do not intersect:

```
SELECT sa.name sensitive_area, hs.name hazardous_site,
       ST_Area(ST_SymDifference(ST_buffer(hs.location,(5 * 5280)),sa.zone)::
   ST_MultiPolygon) area
   FROM hazardous_sites hs, sensitive_areas sa;
```

Figure 8-11 shows that the symmetric difference of the hazardous waste sites and the sensitive areas results in the subtraction of the intersected areas.

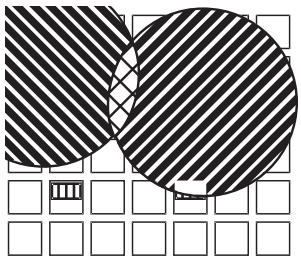


Figure 8-11. Using the ST_SymDifference Function

SE_TotalSize()

SE_TotalSize() returns the sum of the in-row and out-of-row components of a geometry.

You can use this function to obtain an estimate of the amount of diskspace consumed by one or more geometries. However, this function does not account for dbspace and sbspace overhead, so cannot be used to obtain an exact total.

Syntax

```
SE_TotalSize(ST_Geometry)
```

Return Type

INTEGER

See Also

"SE_InRowSize()" on page 8-64

"SE_OutOfRowSize()" on page 8-113

ST_Touches()

ST_Touches() returns `t` (TRUE) if none of the points common to both geometries intersect the interiors of both geometries; otherwise, it returns `f` (FALSE). At least one geometry must be an ST_LineString, ST_Polygon, ST_MultiLineString, or ST_MultiPolygon.

Syntax

```
ST_Touches(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

The GIS technician has been asked to provide a list of all sewer lines whose endpoints intersect another sewer line.

The **sewerlines** table is created with three columns. The first column, **sewer_id**, uniquely identifies each sewer line. The INTEGER **class** column identifies the type of sewer line, generally associated with the line's capacity. The **sewer** ST_LineString column stores the sewer line's geometry:

```
CREATE TABLE sewerlines (sewer_id  integer,
                         class     integer,
                         sewer     ST_LineString);
```

The query returns a list of **sewer_ids** that touch one another:

```
SELECT s1.sewer_id, s2.sewer_id
   FROM sewerlines s1, sewerlines s2
   WHERE ST_Touches(s1.sewer, s2.sewer);
```

SE_Trace()

The **SE_Trace()** function provides three levels of tracing for SQL functions:

- **Level 1.** SQL function entry and exit
- **Level 2.** Secondary function entry and exit
- **Level 3.** Miscellaneous additional tracing

Important: You should turn off tracing during normal use, because it can generate tremendous amounts of trace output data. It is intended for use by IBM Informix Technical Support in isolating problems.

Syntax

```
SE_Trace(pathname lvarchar, level integer)
```

Where *pathname* is a filename and full path on the server machine and *level* is 1, 2 or 3.

Return Type

The text string `OK`, if the function was successfully executed

Example

This example shows the file separator for UNIX®: a forward slash (/). If you are using the Windows version of the IBM Informix Spatial DataBlade Module, substitute the UNIX file separator with a backslash (\ ) in the pathname.

```
execute function SE_Trace ('/tmp/spatial.trc', 1);
```

ST_Transform()

ST_Transform() transforms an ST_Geometry into the specified spatial reference system.

The transformations allowed in the current version of the Spatial DataBlade are:

- Between two UNKNOWN coordinate systems (that is, the **srtext** column in the **spatial_references** table for both SRIDs contains UNKNOWN)
- Between a projected coordinate system and an unprojected coordinate system, in which the underlying geographic coordinate systems are the same
- Between two projected coordinate systems, in which the underlying geographic coordinate systems are the same
- Between two coordinate systems with the same geographic coordinate system (these are coordinate systems with a difference in false origin or system unit only)

Appendix B, "OGC Well-Known Text Representation of Spatial Reference Systems," on page B-1, contains information about supported coordinate systems and their constituent parts.

Syntax

```
ST_Transform(g ST_Geometry, SRID integer)
```

Return Type

ST_Geometry

Example

Example 1: Changing the false origin of a spatial reference system

Suppose you created a **spatial_references** table entry suitable for Australia. You can do this with the **SE_CreateSrid()** function as follows:

```
EXECUTE FUNCTION SE_CreateSrid (110, -45, 156, -10,
                                "Australia: lat/lon coords");

(expression)

        1002
```

Now load all of your data for Australia. In this example we just create a table with a few points.

```
CREATE TABLE aus_locns (name varchar(128), locn ST_Point);

INSERT INTO aus_locns VALUES ("Adelaide",    '1002 point(139.14 -34.87)');
INSERT INTO aus_locns VALUES ("Brisbane",    '1002 point(153.36 -27.86)');
INSERT INTO aus_locns VALUES ("Canberra",    '1002 point(148.84 -35.56)');
INSERT INTO aus_locns VALUES ("Melbourne",   '1002 point(145.01 -37.94)');
INSERT INTO aus_locns VALUES ("Perth",       '1002 point(116.04 -32.12)');
INSERT INTO aus_locns VALUES ("Sydney",      '1002 point(151.37 -33.77)');
```

After loading all of your data for the Australian mainland, you realize you need to include data for a few outlying islands, such as Norfolk Island and the Cocos Islands. However, the false origin and scale factor that you chose for SRID 1002 will not work for these islands as well:

```
INSERT INTO aus_locns VALUES ("Norfolk Is.", '1002 point(167.83 -29.24)');
(USE19) - Coordinates out of bounds in ST_PointIn.

INSERT INTO aus_locns VALUES ("Cocos Is.",   '1002 point( 96.52 -12.08)');
(USE19) - Coordinates out of bounds in ST_PointIn.
```

The solution is to create a new **spatial_references** table entry with a false origin and scale factor that accommodates both the old data and new data, and then update the old data:

```
EXECUTE FUNCTION SE_CreateSrid (95, -55, 170, -10,
                                "Australia + outer islands: lat/lon coords");

(expression)

        1003


INSERT INTO aus_locns VALUES ("Norfolk Is.", '1003 point(167.83 -29.24)');
INSERT INTO aus_locns VALUES ("Cocos Is.",   '1003 point( 96.52 -12.08)');

UPDATE aus_locns
   SET locn = ST_Transform(locn, 1003)::ST_Point
   WHERE ST_Srid(locn) = 1002;
```

Example 2: Projecting data dynamically

In a typical application, spatial data is stored in unprojected latitude and longitude format. Then, when you draw a map, you retrieve the data in a particular projection, letting the IBM Informix Spatial DataBlade Module do the necessary transformations as it retrieves data from a table.

First, create a **spatial_references** table entry that is suitable for your unprojected data. For this example we use the1983 North American datum. Because this is a well-known, standard datum we can use the **SE_CreateSrtext()** function to create the **srtext** field:

```
INSERT INTO spatial_references
  (srid, description, falsex, falsey, xyunits,
   falsez, zunits, falsem, munits, srtext)
VALUES (1004, "Unprojected lat/lon, NAD 83 datum",
        -180, -90, 5000000, 0, 1000, 0, 1000,
        SE_CreateSrtext(4269));
```

Now create a table and load your data:

```
CREATE TABLE airports (id    char(4),
                       name  varchar(128),
                       locn  ST_Point);

INSERT INTO airports VALUES(
   'BTM', 'Bert Mooney',          '1004 point(-112.4975 45.9548)');
INSERT INTO airports VALUES(
   'BZN', 'Gallatin Field',       '1004 point(-111.1530 45.7769)');
INSERT INTO airports VALUES(
   'COD', 'Yellowstone Regional', '1004 point(-109.0238 44.5202)');
INSERT INTO airports VALUES(
   'JAC', 'Jackson Hole',         '1004 point(-110.7377 43.6073)');
INSERT INTO airports VALUES(
   'IDA', 'Fanning Field',        '1004 point(-112.0702 43.5146)');
```

Create one or more **spatial_references** table entries for any projections that you will need. Be sure that the underlying geographic coordinate system (in this case, NAD 83) is the same; this version of the IBM Informix Spatial DataBlade Module does not perform datum conversions:

```
INSERT INTO spatial_references
  (srid, description, falsex, falsey, xyunits,
   falsez, zunits, falsem, munits, srtext)
VALUES (1005, "UTM zone 12N, NAD 83 datum",
        336000, 4760000, 1000, 0, 1000, 0, 1000,
        SE_CreateSrtext(26912));
```

Transform the data to a projected coordinate system on an as needed basis:

```
SELECT id, ST_Transform(locn, 1005) as utm FROM airports;


id   BTM
utm  1005 POINT (383951.152 5090115.666)

id   BZN
utm  1005 POINT (488105.331 5069271.419)

id   COD
utm  1005 POINT (657049.762 4931552.365)

id   JAC
utm  1005 POINT (521167.881 4828291.447)

id   IDA
utm  1005 POINT (413500.979 4818519.081)
```

ST_Union()

ST_Union() returns an ST_Geometry object that is the union of two source objects.

Syntax

```
ST_Union(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

ST_Geometry

Example

The **sensitive_areas** table contains several columns that describe the threatened institutions in addition to the **zone** column, which stores the institutions' ST_Polygon geometries:

```
CREATE TABLE sensitive_areas (id     integer,
                              name   varchar(128),
                              size   float,
                              type   varchar(10),
                              zone   ST_Polygon);
```

The **hazardous_sites** table stores the identity of the sites in the **site_id** and **name** columns, while the actual geographic location of each site is stored in the **location** point column:

```
CREATE TABLE hazardous_sites (site_id   integer,
                              name      varchar(40),
                              location  ST_Point);
```

The **ST_Buffer()** function generates a five-mile buffer surrounding the hazardous waste site locations. The **ST_Union()** function generates polygons from the union of the buffered hazardous waste site polygons and the sensitive areas. The **ST_Area()** function returns the unioned polygon area:

```
SELECT sa.name sensitive_area, hs.name hazardous_site,
       ST_Area(ST_Union(ST_Buffer(hs.location,(5 * 5280)),sa.zone)::
   ST_MultiPolygon) area
   FROM hazardous_sites hs, sensitive_areas sa;
```

See Also

"SE_Dissolve()" on page 8-41

SE_VertexAppend()

SE_VertexAppend() appends a vertex to the end of an ST_LineString. If the linestring has Z values or measures, the vertex to be appended must also have Z values or measures.

Syntax

```
SE_VertexAppend (ST_LineString, ST_Point)
```

Return Type

ST_LineString

SE_VertexDelete()

SE_VertexDelete() deletes a vertex from a geometry. You must supply the exact vertex to be deleted, including Z value and measure if applicable. All vertices in the geometry which match this value will be deleted.

Syntax

```
SE_VertexDelete (ST_Geometry, ST_Point)
```

Return Type

ST_Geometry

SE_VertexUpdate()

SE_VertexUpdate() changes the value of a vertex in a geometry. You must supply both the exact old value and the new value of the vertex to be altered. If the input geometry has Z values or measures, you must supply them as well. All vertices in the geometry which match the old value will be updated.

Syntax

```
SE_VertexUpdate (ST_Geometry, old ST_Point, new ST_Point)
```

Return Type

ST_Geometry

ST_Within()

ST_Within() returns `t` (TRUE) if the first object is completely within the second; otherwise, it returns `f` (FALSE).

Syntax

```
ST_Within(g1 ST_Geometry, g2 ST_Geometry)
```

Return Type

BOOLEAN

Example

In the example, two tables are created: **buildingfootprints** contains a city's building footprints, while the other, **lots**, contains its lots. The city engineer wants to make sure that all the building footprints are completely inside their lots.

In both tables the ST_MultiPolygon data type stores the ST_Geometry of the building footprints and the lots. The database designer selected ST_MultiPolygon for both features because a lot can be separated by a natural feature such as a river, and a building footprint can be made up of several buildings:

```
CREATE TABLE buildingfootprints (building_id  integer,
                                 lot_id       integer,
                                 footprint    ST_MultiPolygon);


CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The city engineer first retrieves the buildings that are not completely within a lot:

```
SELECT building_id
   FROM buildingfootprints, lots
   WHERE ST_Within(footprint,lot);

building_id

        506
        543
        178
```

The city engineer realizes that although the first query produces a list of all building IDs that have footprints outside a lot polygon, it does not ascertain whether the rest have the correct **lot_id** assigned to them. This second query performs a data integrity check on the **lot_id** column of the **buildingfootprints** table:

```
SELECT bf.building_id, bf.lot_id bldg_lot_id, lots.lot_id lots_lot_id
   FROM buildingfootprints bf, lots
   WHERE ST_Within(footprint,lot)
   AND lots.lot_id <> bf.lot_id;

building_id bldg_lot_id lots_lot_id

        178        5192         203
```

ST_WKBToSQL()

ST_WKBToSQL() constructs an ST_Geometry given its well-known binary representation. The SRID of the ST_Geometry is 0.

Syntax

```
ST_WKBToSQL(WKBGeometry lvarchar)
```

Return Type

ST_Geometry

Example

The following CREATE TABLE statement creates the **lots** table, which has two columns: **lot_id**, which uniquely identifies each lot, and the **lot** polygon column, which contains the geometry of each lot:

```
CREATE TABLE lots (lot_id  integer,
                   lot     ST_MultiPolygon);
```

The following C code fragment contains ODBC functions embedded with IBM Informix Spatial DataBlade Module SQL functions that insert data into the **lots** table.

The **ST_WKBToSQL()** function converts WKB representations into IBM Informix Spatial DataBlade Module geometry. The entire INSERT statement is copied into the *sql_stmt* string. The INSERT statement contains parameter markers to accept the **lot_id** data and the **lot** data, dynamically:

```
/* Create the SQL insert statement to populate the lots table.
     * The question marks are parameter markers that indicate the
     * column values that will be inserted at run time. */
    sprintf(sql_stmt,
            "INSERT INTO lots (lot_id, lot) "
            "VALUES (?, ST_WKBToSQL(?))");

    /* Prepare the SQL statement for execution */
    rc = SQLPrepare (hstmt, (unsigned char *)sql_stmt, SQL_NTS);

    /* Bind the lot_id to the first parameter. */
    pcbvalue1 = 0;
    rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                           SQL_INTEGER, 0, 0,
                           &lot_id, 0, &pcbvalue1);

    /* Bind the lot geometry to the second parameter. */
    pcbvalue2 = lot_wkb_len;
    rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                           SQL_INFX_UDT_LVARCHAR, lot_wkb_len, 0,
                           lot_wkb_buf, lot_wkb_len, &pcbvalue2);

    /* Execute the insert statement. */
    rc = SQLExecute (hstmt);
```

ST_WKTToSQL()

ST_WKTToSQL() constructs an ST_Geometry given its well-known text representation. The SRID of the ST_Geometry is 0.

Syntax

```
ST_WKTToSQL (WKT lvarchar)
```

Return Type

ST_Geometry

Example

The following CREATE TABLE statement creates the **geometry_test** table, which contains two columns: **gid**, of type INTEGER, which uniquely identifies each row, and the **g1** column, which stores the geometry:

```
CREATE TABLE geometry_test (gid integer,
                            g1  ST_Geometry);
```

The following INSERT statements insert the data into the **gid** and **g1** columns of the **geometry_test** table. The **ST_WKTToSQL()** function converts the text representation of each geometry into its corresponding IBM Informix Spatial DataBlade Module instantiable subclass:

```
INSERT INTO geometry_test VALUES(
   1,
   ST_WKTToSQL('point (10.02 20.01)')
);

INSERT INTO geometry_test VALUES(
   2,
   ST_WKTToSQL('linestring (10.02 20.01,10.01 30.01,10.01 40.01)')
);

INSERT INTO geometry_test VALUES(
   3,
   ST_WKTToSQL('polygon ((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94,10.02 20.01))')
);

INSERT INTO geometry_test VALUES(
   4,
   ST_WKTToSQL('multipoint (10.02 20.01,10.32 23.98,11.92 35.64)')
);

INSERT INTO geometry_test VALUES(
   5,
   ST_WKTToSQL('multilinestring ((10.02 20.01,10.32 23.98,11.92
25.64),(9.55 23.75,15.36 30.11))')
);

INSERT INTO geometry_test VALUES(
   6,
   ST_WKTToSQL('multipolygon (((10.02 20.01,11.92 35.64,25.02
34.15,19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52
32.87,52.43 31.90,51.71 21.73)))')
);
```

ST_X()

ST_X() returns the X coordinate of a point.

Syntax

```
ST_X(pt1 ST_Point)
```

Return Type

DOUBLE PRECISION

Example

The **x_test** table is created with two columns: **gid**, which uniquely identifies the row, and the **pt1** point column:

```
CREATE TABLE x_test (gid integer,
                     pt1 ST_Point);
```

The following INSERT statements insert two rows. One is a point without a Z coordinate or a measure. The other column has both a Z coordinate and a measure:

```
INSERT INTO x_test VALUES(
   1,
   ST_PointFromText('point (10.02 20.01)', 1000)
);

INSERT INTO x_test VALUES(
   2,
   ST_PointFromText('point zm (10.02 20.01 5.0 7.0)', 1000)
);
```

The query retrieves the values in the **gid** column and the DOUBLE PRECISION X coordinate of the points:

```
SELECT gid, ST_X(pt1) x_coord
   FROM x_test;

      gid        x_coord

        1 10.02000000000
        2 10.02000000000
```

SE_Xmax() and SE_Xmin()

The **SE_Xmax()** and **SE_Xmin()** functions return the maximum and minimum X coordinates of a geometry.

Syntax

```
ST_Xmax(ST_Geometry)

ST_Xmin(ST_Geometry)
```

Return Type

DOUBLE PRECISION

ST_Y()

ST_Y() returns the Y coordinate of a point.

Syntax

```
ST_Y(p1 ST_Point)
```

Return Type

DOUBLE PRECISION

Example

The **y_test** table is created with two columns: **gid**, which uniquely identifies the row, and the **pt1** point column:

```
CREATE TABLE y_test (gid integer,
                     pt1 ST_Point);
```

The following INSERT statements insert two rows. One is a point without a Z coordinate or a measure. The other has both a Z coordinate and a measure:

```
INSERT INTO y_test VALUES(
    1,
    ST_PointFromText('point (10.02 20.01)', 1000)
);

INSERT INTO y_test VALUES(
    2,
    ST_PointFromText('point zm (10.02 20.01 5.0 7.0)',1000)
);
```

The query retrieves the values in the **gid** column and the DOUBLE PRECISION Y coordinate of the points:

```
SELECT gid, ST_Y(pt1) y_coord
   FROM y_test;

        gid        y_coord

          1 20.01000000000
          2 20.01000000000
```

SE_Ymax() and SE_Ymin()

The **SE_Ymax()** and **SE_Ymin()** functions return the maximum and minimum Y coordinates of a geometry.

Syntax

```
ST_Ymax(ST_Geometry)

ST_Ymin(ST_Geometry)
```

Return Type

DOUBLE PRECISION

SE_Z()

SE_Z() returns the Z coordinate of a point.

Syntax

```
SE_Z(p1 ST_Point)
```

Return Type

DOUBLE PRECISION

Example

The **z_test** table is created with two columns: **gid**, which uniquely identifies the row, and the **pt1** point column:

```
CREATE TABLE z_test (gid integer,
                     pt1 ST_Point);
```

The following INSERT statements insert two rows. One is a point without a Z coordinate or a measure. The other has both a Z coordinate and a measure:

```
INSERT INTO z_test VALUES(
   1,
   ST_PointFromText('point (10.02 20.01)', 1000)
);

INSERT INTO z_test VALUES(
   2,
   ST_PointFromText('point zm (10.02 20.01 5.0 7.0)', 1000)
);
```

The query retrieves the values in the **gid** column and the DOUBLE PRECISION Z coordinate of the points. The first row is NULL because the point does not have a Z coordinate:

```
SELECT gid, SE_Z(pt1) z_coord
  FROM z_test;

       gid        z_coord

         1
         2 5.000000000000
```

ST_Zmax() and ST_Zmin()

The **ST_Zmax()** and **ST_Zmin()** functions return the maximum and minimum Z coordinates of a geometry.

Syntax

```
ST_Zmax(ST_Geometry)

ST_Zmin(ST_Geometry)
```

Return Type

DOUBLE PRECISION

Appendix A. Loading and Unloading Shapefile Data

The IBM Informix Spatial DataBlade Module provides three utilities for working with spatial data contained in ESRI shapefiles:

- The **infoshp** utility is useful for gathering information from ESRI shapefiles when you are preparing to load the shapefiles into a database.
- The **loadshp** utility loads spatial data from an ESRI shapefile into the database.
- The **unloadshp** utility copies data from the database to an ESRI shapefile.

The executable files for the utilities are located in:

$INFORMIXDIR/extend/spatial.version/bin

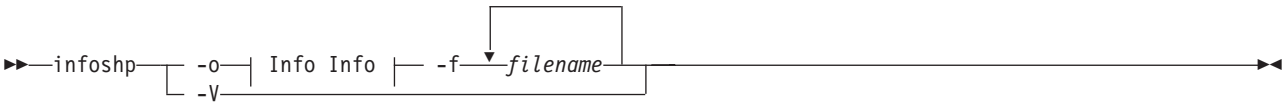
You can leave them in this directory, or you can copy them to **$INFORMIXDIR/bin**. If you leave them here, you may want to add **$INFORMIXDIR/extend/spatial.version/bin** to your $PATH environment variable. If you copy the files to **$INFORMIXDIR/bin**, you may not need to change your $PATH environment variable, if **$INFORMIXDIR/bin** is already included in the variable definition.

Attention: The **loadshp** utility does not add information to ESRI system tables, such as the **layers** table. Therefore, data loaded with the **loadshp** utility is not accessible to ArcSDE and other ESRI client tools. Use the ESRI **shp2sde** command to load data if you want to access it using ESRI client tools. Data loaded using the **loadshp** utility is accessible to client programs that do not depend on ESRI system tables other than the OGC-standard **geometry_columns** and **spatial_references** tables.

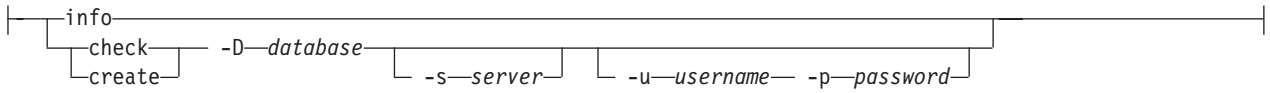
The infoshp Utility

The **infoshp** utility reports information extracted from headers of the **.shp**, **.shx**, and **.dbf** files that make up ESRI shapefiles. It also detects the presence of optional **.prj** files that may be associated with ESRI shapefiles and reports the coordinate system information. The **infoshp** utility can check for **spatial_references** table entries that are qualified to load one or more shapefiles. If no spatial reference exists that can be used to load the specified shapefiles, the **infoshp** utility can create a new entry in the **spatial_references** table for loading the shapefiles.

Syntax



Info Info:



Operation Modes

You use the **-o** flag to set the operation mode for the **infoshp** command. Set the **-o** flag to one of the following options:

- **check.** Checks for **spatial_references** table entries that are qualified to load the specified shapefiles.
- **create.** Creates a new entry in the **spatial_references** table for loading the specified shapefiles.
- **info.** Reports information extracted from the headers of the **.shp**, **.shx**, **.dbf**, and optional **.prj** files associated with the specified shapefiles.

Command-Line Switches

You can use the **infoshp** command flags to specify the following options.

-D	The database name
-f	The path and name of one or more ESRI shapefiles to process
-p	(optional) The Informix password If you specify the **-p** option, you must also specify the **-u** option.
-s	(optional) The IBM Informix server Defaults to the value of the INFORMIXSERVER environment variable
-u	(optional) The Informix user name If you specify the **-u** option, you must also specify the **-p** option.
-V	Prints version information for this utility

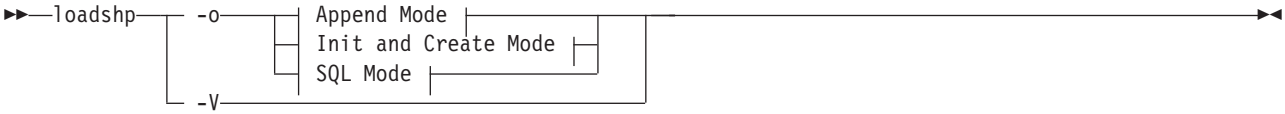
The loadshp Utility

The **loadshp** utility loads spatial features and associated attributes from an ESRI shapefile into a table in an IBM Informix Dynamic Server database. The IBM Informix Spatial DataBlade Module must be registered in the database.

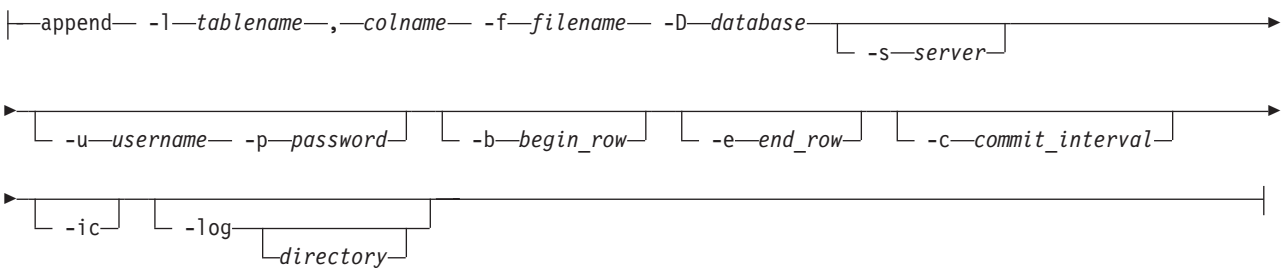
When **loadshp** creates a table, it inserts a row into the **geometry_columns** system table. When you want to drop a table created by **loadshp**, you should also delete the corresponding row from the **geometry_columns** table.

Tip: The **loadshp** utility creates a primary key constraint on the **se_row_id** column of the table that you are loading.

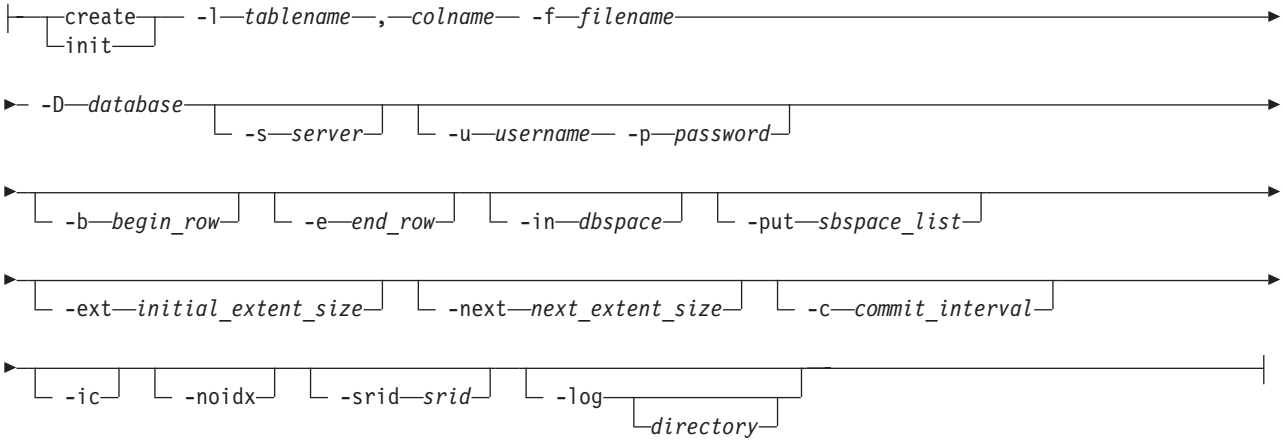
Syntax



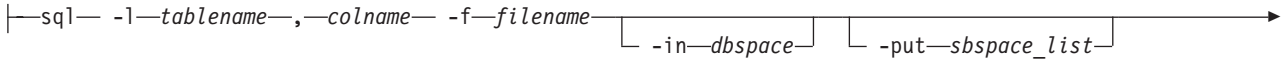
Append Mode:

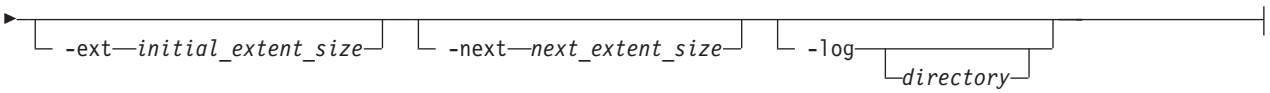


Init and Create Mode:



SQL Mode:





Operation Modes

You use the **-o** flag to set the operation mode for the **loadshp** command. Set the **-o** flag to one of the following options:

- **append.** Spatial features are added to the table specified by the **-l** flag. The structure of the existing database table must match the structure of a table derived from information specified by the **-l** command line option and from metadata stored in the shapefile's associated **.dbf** file.
- **create.** Spatial features are loaded into a newly created database table. An error is returned if a table with the name specified by the **-l** flag already exists. The structure of the new table is derived from the table name and column name specified by the **-l** command line option as well as from metadata stored in the shapefile's associated **.dbf** file.
- **init.** The table specified by the **-l** flag is first dropped and then spatial features are loaded into a newly created table of the same name.
- **sql.** The following SQL statements are displayed on the console (and can be logged to a file):
  - DROP TABLE ...
  - DELETE FROM **geometry_columns** ...
  - CREATE TABLE ...
  - INSERT INTO **geometry_columns** ...
  - CREATE UNIQUE INDEX ... USING btree;
  - ALTER TABLE ... ADD CONSTRAINT PRIMARY KEY (se_row_id) ...
  - CREATE INDEX ... USING rtree;
  - UPDATE STATISTICS ...

  The structure of the table to be loaded is derived from information specified by the **-l** command-line option and from metadata stored in the shapefile's associated **.dbf** file containing feature attributes.

Command-Line Switches

You can use the **loadshp** command-line switches to specify the following options.

-b (optional) The first row in the shapefile to load

-c (optional) The number of rows to load before committing work and beginning a new transaction

Defaults to 1000 rows

If you are loading data into a database that does not have transaction logging enabled, the commit interval determines how frequently information messages are displayed on the console.

-D The database name

-e (optional) The last row in the shapefile to load

-ext (optional) Specifies the initial extent size for the table to be loaded

This option is not valid in the **append** option of the **-o** flag.

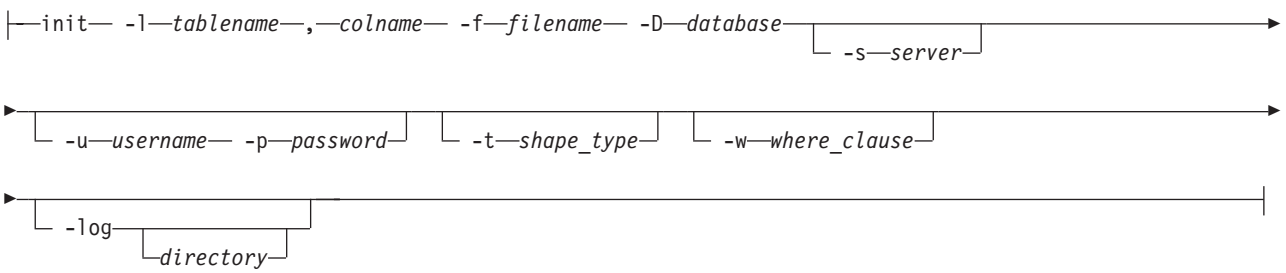
-f	The path and name of the ESRI shapefile to be loaded
-ic	(optional) Specifies use of an INSERT CURSOR Using an INSERT CURSOR to load data significantly reduces load time, but limits the client program's ability to handle errors. INSERT CURSORs buffer rows before writing them to the database to improve performance. If an error is encountered during the load, all buffered rows following the last successfully inserted row are discarded.
-in	(optional) Specifies the dbspace in which to create the table to be loaded This option is not valid in the **append** option of the **-o** flag.
-l	The table and geometry column to load data into
-log	(optional) Specifies whether to write information about the status of data loading to a log file The log file has the same name as the shapefile you are loading from with the extension **.log**. If you do not specify a directory, the log file is created in the same directory as the shapefile you are loading from.
-next	Specifies the next extent size for the table to be loaded This option is not valid in the **append** option of the **-o** flag.
-noidx	(optional) Specifies that indexes should not be built and statistics should not be updated after the shapefile data has been loaded Unless this option is specified when executing **loadshp -o create** or **loadshp -o init**, a unique B-tree index is built on the **se_row_id** column, an R-tree index is built on the **geometry** column, and statistics are updated for the table after the shapefile data has been loaded.
-p	(optional) The Informix password If you specify the **-p** option, you must also specify the **-u** option.
-put	(optional) Specifies the sbspaces in which large shapes inserted into the load table's geometry column will be stored Multiple sbspace names must be separated by commas and no white space must appear in the list of sbspace names. This option is not valid with the **append** option of the **-o** flag.
-s	(optional) The IBM Informix server Defaults to the value of the INFORMIXSERVER environment variable.
-srid	(optional) The spatial reference ID for the data you are loading The integer you specify must exist as a spatial reference ID in the **spatial_references** table. If you do not specify the **-srid** command line option, the spatial reference ID defaults to 0.
-u	(optional) The Informix user name If you specify the **-u** option, you must also specify the **-p** option.
-V	Prints version information for this utility

The unloadshp Utility

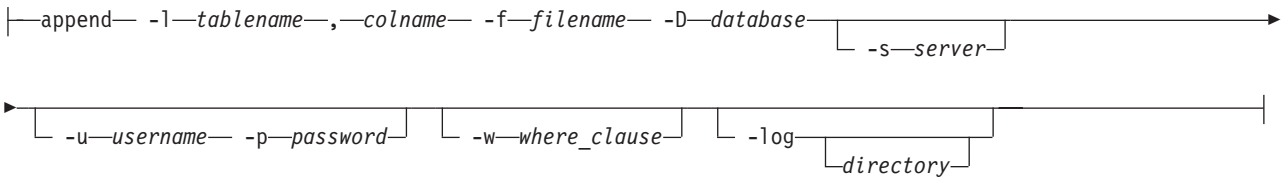
The **unloadshp** utility copies spatial features and associated attributes from a table in an IBM Informix Dynamic Server database into an ESRI shapefile. The IBM Informix Spatial DataBlade Module must be registered in the database.

Syntax

Init Mode:



Append Mode:



Operation Modes

You use the **-o** flag to set the operation mode for the **unloadshp** command. Set the **-o** flag to one of the following options:

- **append.** Spatial features are appended to the existing ESRI shapefile specified by the **-f** option.
- **init.** Spatial features are unloaded into a newly created ESRI shapefile.

Command-Line Switches

You can use the **unloadshp** command flags to specify the following options.

-D	The database name
-f	The path and name of the shapefile
-l	The table and geometry column from which to extract data The table and column must exist, and the executing user must either own the table or have access to it.
-log	(optional) Specifies whether to write information about the status of data loading to a log file

The log file has the same name as the shapefile you are loading into with the extension **.log**.

If you do not specify a directory, the log file is created in the same directory as the shapefile you are loading.

-p (optional) The Informix password

If you specify the **-p** option, you must also specify the **-u** option.

-s (optional) The IBM Informix server

Defaults to the value of the INFORMIXSERVER environment variable.

-t (optional) An integer indicating the type of shape to extract and write to the ESRI shapefile

The possible values are:

1 Point

3 PolyLine

5 Polygon

8 Multipoint

9 PointZ

10 PolyLineZ

11 PointZM

13 PolyLineZM

15 PolygonZM

18 MultiPointZM

19 PolygonZ

20 MultiPointZ

21 PointM

23 PolyLineM

25 PolygonM

28 MultiPointM

If you do not specify the **-t** option, the type defaults to the type of the first shape retrieved from the database table.

-u (optional) The Informix user name

If you specify the **-u** option, you must also specify the **-p** option.

-V Prints version information for this utility

-w (optional) The SQL WHERE clause to qualify the data extracted from the table

Enclose the WHERE clause in double quotes and enclose any string literals within the clause in single quotes. Omit the keyword WHERE.

Appendix B. OGC Well-Known Text Representation of Spatial Reference Systems

This appendix explains how to represent a spatial reference system using a text string and provides information about supported units of measure, spheroids, datums, prime meridians, and projections.

The information provided in this appendix is primarily intended for use with the **ST_Transform()** function.

The Text Representation of a Spatial System

The well-known text representation of spatial reference systems provides a standard textual representation for spatial reference system information.

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.

A coordinate system is composed of several objects. Each object is defined by an uppercase keyword (for example, DATUM or UNIT) followed by the defining, comma-delimited parameters of the object in brackets. Some objects are composed of other objects.

Implementations are free to substitute standard brackets ( ) for square brackets [ ] and should be prepared to read both forms of brackets.

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

```
<coordinate system> = <projected cs> | <geographic cs>
   | <geocentric cs>
<projected cs> = PROJCS["<name>", <geographic cs>,
   <projection>, {<parameter>,}* <linear unit>]
<projection> = PROJECTION["<name>"]
<parameter> = PARAMETER["<name>", <value>]
<value> = <number>
```

A dataset's coordinate system is identified by one of the following three keywords:

- PROJCS, if the data is in projected coordinates
- GEOGCS, if in geographic coordinates
- GEOCCS, if in geocentric coordinates

The PROJCS keyword is followed by all of the pieces that define the projected coordinate system. The first piece of any object is always the name. Several objects follow the name: the geographic coordinate system, the map projection, one or more parameters, and the linear unit of measure.

As an example, UTM zone 10N on the NAD83 datum is defined as:

```
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: the datum, the prime meridian, and the angular unit of measure:

```
<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 10N on NAD83 is:

```
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 units of measure:

```
<angular unit> = <unit>
<linear unit> = <unit>
<unit> = UNIT["<name>", <conversion factor>]
<conversion factor> = <number>
```

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

Therefore, the full string representation of UTM zone 10N is:

```
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:

```
<geocentric cs> = GEOCCS["<name>", <datum>, <prime meridian>, <linear unit>]
```

You can use the **SE_CreateSrtext()** function to assist you in constructing these spatial reference system text strings. Refer to "SE_CreateSrtext()" on page 8-33 for more information.

The remainder of this appendix shows the OGC well-known text "building blocks" of spatial reference systems that are supported by the IBM Informix Spatial DataBlade Module.

These text strings can be generated by the **SE_CreateSrtext()** function; you use the factory ID number in the first column of the table as the input argument to **SE_CreateSrtext()**.

Linear Units

The following table shows a representative sample of the linear units of measure supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported linear units of measure, see the file **INFORMIXDIR/extend/spatial.***version***/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
9001	International meter	UNIT["Meter",1]
9002	International foot	UNIT["Foot",0.3048]
9003	US survey foot	UNIT["Foot_US",0.3048006096012192]
9005	Clarke's foot	UNIT["Foot_Clarke",0.3047972650]
9014	Fathom	UNIT["Fathom",1.8288]
9030	International nautical mile	UNIT["Nautical_Mile",1852]
9031	German legal meter	UNIT["Meter_German",1.000001359650]
9033	US survey chain	UNIT["Chain_US",20.11684023368047]
9034	US survey link	UNIT["Link_US",0.2011684023368047]
9035	US survey mile	UNIT["Mile_US",1609.347218694438]
9036	Kilometer	UNIT["Kilometer",1000]
9037	Yard (Clarke)	UNIT["Yard_Clarke",0.914391795]
9038	Chain (Clarke)	UNIT["Chain_Clarke",20.11661949]
9039	Link (Clarke's ratio)	UNIT["Link_Clarke",0.2011661949]
9040	Yard (Sears)	UNIT["Yard_Sears",0.9143984146160287]
9041	Sear's foot	UNIT["Foot_Sears",0.3047994715386762]
9042	Chain (Sears)	UNIT["Chain_Sears",20.11676512155263]
9043	Link (Sears)	UNIT["Link_Sears",0.2011676512155263]
9050	Yard (Benoit 1895 A)	UNIT["Yard_Benoit_1895_A",0.9143992]
9051	Foot (Benoit 1895 A)	UNIT["Foot_Benoit_1895_A",0.3047997333333333]
9052	Chain (Benoit 1895 A)	UNIT["Chain_Benoit_1895_A",20.1167824]
9053	Link (Benoit 1895 A)	UNIT["Link_Benoit_1895_A",0.201167824]
9060	Yard (Benoit 1895 B)	UNIT["Yard_Benoit_1895_B",0.9143992042898124]
9061	Foot (Benoit 1895 B)	UNIT["Foot_Benoit_1895_B",0.3047997347632708]
9062	Chain (Benoit 1895 B)	UNIT["Chain_Benoit_1895_B",20.11678249437587]
9063	Link (Benoit 1895 B)	UNIT["Link_Benoit_1895_B",0.2011678249437587]
9070	Foot (1865)	UNIT["Foot_1865",0.3048008333333334]
9080	Indian geodetic foot	UNIT["Foot_Indian",0.3047995102481469]
9081	Indian foot (1937)	UNIT["Foot_Indian_1937",0.30479841]
9082	Indian foot (1962)	UNIT["Foot_Indian_1962",0.3047996]
9083	Indian foot (1975)	UNIT["Foot_Indian_1975",0.3047995]
9084	Indian yard	UNIT["Yard_Indian",0.9143985307444408]
9085	Indian yard (1937)	UNIT["Yard_Indian_1937",0.91439523]

Factory ID	Description	OGC Well-Known Text String
9086	Indian yard (1962)	UNIT["Yard_Indian_1962",0.9143988]
9087	Indian yard (1975)	UNIT["Yard_Indian_1975",0.9143985]

Angular Units

The following table shows a representative sample of the angular units of measure supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported angular units of measure, see the file **INFORMIXDIR/extend/spatial.***version***/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
9101	Radian	UNIT["Radian",1]
9102	Degree	UNIT["Degree",0.0174532925199433]
9103	Arc-minute	UNIT["Minute",0.0002908882086657216]
9104	Arc-second	UNIT["Second",4.84813681109536E-06]
9105	Grad (angle subtended by 1/400 circle)	UNIT["Grad",0.01570796326794897]
9106	Gon (angle subtended by 1/400 circle)	UNIT["Gon",0.01570796326794897]
9109	Microradian (1e-6 radian)	UNIT["Microradian",1E-06]
9112	Centesimal minute (1/100th Gon (Grad))	UNIT["Minute_Centesimal",0.0001570796326794897]
9113	Centesimal second(1/10000th Gon (Grad))	UNIT["Second_Centesimal",1.570796326794897E-06]
9114	Mil (angle subtended by 1/6400 circle)	UNIT["Mil_6400",0.0009817477042468104]

Geodetic Spheroids

The following table lists a representative sample of the geodetic spheroids supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported geodetic spheroids, see the file **INFORMIXDIR/extend/spatial.***version***/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
7001	Airy 1830	SPHEROID["Airy_1830",6377563.396,299.3249646]
7002	Airy modified	SPHEROID["Airy_Modified",6377340.189,299.3249646]
7041	Average Terrestrial System 1977	SPHEROID["ATS_1977",6378135,298.257]
7003	Australian National	SPHEROID["Australian",6378160,298.25]
7004	Bessel 1841	SPHEROID["Bessel_1841",6377397.155,299.1528128]
7005	Bessel modified	SPHEROID["Bessel_Modified",6377492.018,299.1528128]
7006	Bessel Namibia	SPHEROID["Bessel_Namibia",6377483.865,299.1528128]
7007	Clarke 1858	SPHEROID["Clarke_1858",6378293.639,294.260676369]
7008	Clarke 1866	SPHEROID["Clarke_1866",6378206.4,294.9786982]
7009	Clarke 1866 Michigan	SPHEROID["Clarke_1866_Michigan",6378450.047,294.978684677]

Factory ID	Description	OGC Well-Known Text String
7034	Clarke 1880	SPHEROID["Clarke_1880",6378249.138,293.466307656]
7013	Clarke 1880 (Arc)	SPHEROID["Clarke_1880_Arc",6378249.145,293.466307656]
7010	Clarke 1880 (Benoit)	SPHEROID["Clarke_1880_Benoit",6378300.79,293.466234571]
7011	Clarke 1880 (IGN)	SPHEROID["Clarke_1880_IGN",6378249.2,293.46602]
7012	Clarke 1880 (RGS)	SPHEROID["Clarke_1880_RGS",6378249.145,293.465]
7014	Clarke 1880 (SGA)	SPHEROID["Clarke_1880_SGA",6378249.2,293.46598]
7042	Everest 1830 (definition)	SPHEROID["Everest_1830",6377299.36,300.8017]
7018	Everest 1830 (modified)	SPHEROID["Everest_1830_Modified",6377304.063,300.8017]
7015	Everest (adjustment 1937)	SPHEROID["Everest_Adjustment_1937",6377276.345,300.8017]
7044	Everest (definition 1962)	SPHEROID["Everest_Definition_1962",6377301.243,300.8017255]
7016	Everest (definition 1967)	SPHEROID["Everest_Definition_1967",6377298.556,300.8017]
7045	Everest (definition 1975)	SPHEROID["Everest_Definition_1975",6377299.151,300.8017255]
7031	GEM gravity potential model	SPHEROID["GEM_10C",6378137,298.257222101]
7036	GRS 1967 = International 1967	SPHEROID["GRS_1967",6378160,298.247167427]
7019	GRS 1980	SPHEROID["GRS_1980",6378137,298.257222101]
7020	Helmert 1906	SPHEROID["Helmert_1906",6378200,298.3]
7021	Indonesian National	SPHEROID["Indonesian",6378160,298.247]
7022	International 1924	SPHEROID["International_1924",6378388,297]
7023	International 1967	SPHEROID["International_1967",6378160,298.25]
7024	Krasovsky 1940	SPHEROID["Krasovsky_1940",6378245,298.3]
7025	Transit precise ephemeris	SPHEROID["NWL_9D",6378145,298.25]
7032	OSU 1986 geoidal model	SPHEROID["OSU_86F",6378136.2,298.25722]
7033	OSU 1991 geoidal model	SPHEROID["OSU_91A",6378136.3,298.25722]
7027	Plessis 1817	SPHEROID["Plessis_1817",6376523,308.64]
7035	Authalic sphere	SPHEROID["Sphere",6371000,0]
7028	Struve 1860	SPHEROID["Struve_1860",6378298.3,294.73]
7029	War Office	SPHEROID["War_Office",6378300.583,296]
7026	NWL-10D == WGS 1972	SPHEROID["NWL_10D",6378135,298.26]
7043	WGS 1972	SPHEROID["WGS_1972",6378135,298.26]
7030	WGS 1984	SPHEROID["WGS_1984",6378137,298.257223563]
107001	WGS 1966	SPHEROID["WGS_1966",6378145,298.25]
107002	Fischer 1960	SPHEROID["Fischer_1960",6378166,298.3]
107003	Fischer 1968	SPHEROID["Fischer_1968",6378150,298.3]
107004	Fischer modified	SPHEROID["Fischer_Modified",6378155,298.3]
107005	Hough 1960	SPHEROID["Hough_1960",6378270,297]
107006	Everest modified 1969	SPHEROID["Everest_Modified_1969",6377295.664,300.8017]
107007	Walbeck	SPHEROID["Walbeck",6376896,302.78]
107008	Authalic sphere (ARC/INFO)	SPHEROID["Sphere_ARC_INFO",6370997,0]
107036	GRS 1967 Truncated	SPHEROID["GRS_1967_Truncated",6378160,298.25]

Horizontal Datums (Spheroid Only)

The following table lists a representative sample of the horizontal datums (spheroid only) supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported horizontal datums (spheriod only), see the file **INFORMIXDIR/extend/spatial.*version*/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
6001	Airy 1830	DATUM["D_Airy_1830",SPHEROID["Airy_1830",6377563.396,299.3249646]]
6002	Airy modified	DATUM["D_Airy_Modified",SPHEROID["Airy_Modified",6377340.189, 299.3249646]]
6003	Australian National	DATUM["D_Australian",SPHEROID["Australian",6378160,298.25]]
6004	Bessel 1841	DATUM["D_Bessel_1841",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6005	Bessel modified	DATUM["D_Bessel_Modified",SPHEROID["Bessel_Modified",6377492.018, 299.1528128]]
6006	Bessel Namibia	DATUM["D_Bessel_Namibia",SPHEROID["Bessel_Namibia",6377483.865, 299.1528128]]
6007	Clarke 1858	DATUM["D_Clarke_1858",SPHEROID["Clarke_1858",6378293.639, 294.260676369]]
6008	Clarke 1866	DATUM["D_Clarke_1866",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6009	Clarke 1866 Michigan	DATUM["D_Clarke_1866_Michigan",SPHEROID["Clarke_1866_Michigan", 6378450.047,294.978684677]]
6034	Clarke 1880	DATUM["D_Clarke_1880",SPHEROID["Clarke_1880",6378249.138, 293.466307656]]
6013	Clarke 1880 (Arc)	DATUM["D_Clarke_1880_Arc",SPHEROID["Clarke_1880_Arc",6378249.145, 293.466307656]]
6010	Clarke 1880 (Benoit)	DATUM["D_Clarke_1880_Benoit",SPHEROID["Clarke_1880_Benoit", 6378300.79,293.466234571]]
6011	Clarke 1880 (IGN)	DATUM["D_Clarke_1880_IGN",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6012	Clarke 1880 (RGS)	DATUM["D_Clarke_1880_RGS",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6014	Clarke 1880 (SGA)	DATUM["D_Clarke_1880_SGA",SPHEROID["Clarke_1880_SGA",6378249.2, 293.46598]]
6042	Everest 1830	DATUM["D_Everest_1830",SPHEROID["Everest_1830",6377299.36, 300.8017]]
6015	Everest (adjustment 1937)	DATUM["D_Everest_Adj_1937",SPHEROID["Everest_Adjustment_1937", 6377276.345,300.8017]]
6044	Everest (definition 1962)	DATUM["D_Everest_Def_1962",SPHEROID["Everest_Definition_1962", 6377301.243,300.8017255]]
6016	Everest (definition 1967)	DATUM["D_Everest_Def_1967",SPHEROID["Everest_Definition_1967", 6377298.556,300.8017]]
6045	Everest (definition 1975)	DATUM["D_Everest_Def_1975",SPHEROID["Everest_Definition_1975", 6377299.151,300.8017255]]

Factory ID	Description	OGC Well-Known Text String
6018	Everest modified	DATUM["D_Everest_Modified",SPHEROID["Everest_1830_Modified", 6377304.063,300.8017]]
6031	GEM gravity potential model	DATUM["D_GEM_10C",SPHEROID["GEM_10C",6378137,298.257222101]]
6036	GRS 1967	DATUM["D_GRS_1967",SPHEROID["GRS_1967",6378160,298.247167427]]
6019	GRS 1980	DATUM["D_GRS_1980",SPHEROID["GRS_1980",6378137,298.257222101]]
6020	Helmert 1906	DATUM["D_Helmert_1906",SPHEROID["Helmert_1906",6378200,298.3]]
6021	Indonesian National	DATUM["D_Indonesian",SPHEROID["Indonesian",6378160,298.247]]
6022	International 1927	DATUM["D_International_1924",SPHEROID["International_1924",6378388,297]]
6023	International 1967	DATUM["D_International_1967",SPHEROID["International_1967",6378160,298.25]]
6024	Krasovsky 1940	DATUM["D_Krasovsky_1940",SPHEROID["Krasovsky_1940",6378245, 298.3]]
6025	Transit precise ephemeris	DATUM["D_NWL_9D",SPHEROID["NWL_9D",6378145,298.25]]
6032	OSU 1986 geoidal model	DATUM["D_OSU_86F",SPHEROID["OSU_86F",6378136.2,298.25722]]
6033	OSU 1991 geoidal model	DATUM["D_OSU_91A",SPHEROID["OSU_91A",6378136.3,298.25722]]
6027	Plessis 1817	DATUM["D_Plessis_1817",SPHEROID["Plessis_1817",6376523,308.64]]
6035	Authalic sphere	DATUM["D_Sphere",SPHEROID["Sphere",6371000,0]]
6028	Struve 1860	DATUM["D_Struve_1860",SPHEROID["Struve_1860",6378298.3,294.73]]
6029	War Office	DATUM["D_War_Office",SPHEROID["War_Office",6378300.583,296]]
106001	WGS 1966	DATUM["D_WGS_1966",SPHEROID["WGS_1966",6378145,298.25]]
106002	Fischer 1960	DATUM["D_Fischer_1960",SPHEROID["Fischer_1960",6378166,298.3]]
106003	Fischer 1968	DATUM["D_Fischer_1968",SPHEROID["Fischer_1968",6378150,298.3]]
106004	Fischer modified	DATUM["D_Fischer_Modified",SPHEROID["Fischer_Modified",6378155, 298.3]]
106005	Hough 1960	DATUM["D_Hough_1960",SPHEROID["Hough_1960",6378270,297]]
106006	Everest modified 1969	DATUM["D_Everest_Modified_1969", SPHEROID["Everest_Modified_1969",6377295.664,300.8017]]
106007	Walbeck	DATUM["D_Walbeck",SPHEROID["Walbeck",6376896,302.78]]
106008	Authalic sphere (ARC/INFO)	DATUM["D_Sphere_ARC_INFO",SPHEROID["Sphere_ARC_INFO", 6370997,0]]

Horizontal Datums

The following table lists a representative sample of the horizontal datums supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported horizontal datums, see the file **INFORMIXDIR/extend/spatial.***version*/**include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
6201	Adindan	DATUM["D_Adindan",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6205	Afgooye	DATUM["D_Afgooye",SPHEROID["Krasovsky_1940",6378245,298.3]]

Factory ID	Description	OGC Well-Known Text String
6206	Agadez	DATUM["D_Agadez",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6202	Australian Geodetic Datum 1966	DATUM["D_Australian_1966",SPHEROID["Australian",6378160,298.25]]
6203	Australian Geodetic Datum 1984	DATUM["D_Australian_1984",SPHEROID["Australian",6378160,298.25]]
6204	Ain el Abd 1970	DATUM["D_Ain_el_Abd_1970",SPHEROID["International_1924", 6378388,297]]
6289	Amersfoort	DATUM["D_Amersfoort",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6208	Aratu	DATUM["D_Aratu",SPHEROID["International_1924",6378388,297]]
6209	Arc 1950	DATUM["D_Arc_1950",SPHEROID["Clarke_1880_Arc",6378249.145, 293.466307656]]
6210	Arc 1960	DATUM["D_Arc_1960",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6901	Ancienne Triangulation Francaise	DATUM["D_ATF",SPHEROID["Plessis_1817",6376523,308.64]]
6122	Average Terrestrial System 1977	DATUM["D_ATS_1977",SPHEROID["ATS_1977",6378135,298.257]]
6212	Barbados 1938	DATUM["D_Barbados_1938",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6211	Batavia	DATUM["D_Batavia",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6213	Beduaram	DATUM["D_Beduaram",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6214	Beijing 1954	DATUM["D_Beijing_1954",SPHEROID["Krasovsky_1940",6378245,298.3]]
6215	Reseau National Belge 1950	DATUM["D_Belge_1950",SPHEROID["International_1924",6378388,297]]
6313	Reseau National Belge 1972	DATUM["D_Belge_1972",SPHEROID["International_1924",6378388,297]]
6216	Bermuda 1957	DATUM["D_Bermuda_1957",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6217	Bern 1898	DATUM["D_Bern_1898",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6306	Bern 1938	DATUM["D_Bern_1938",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6218	Bogota	DATUM["D_Bogota",SPHEROID["International_1924",6378388,297]]
6219	Bukit Rimpah	DATUM["D_Bukit_Rimpah",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6220	Camacupa	DATUM["D_Camacupa",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6221	Campo Inchauspe	DATUM["D_Campo_Inchauspe",SPHEROID["International_1924", 6378388,297]]
6222	Cape	DATUM["D_Cape",SPHEROID["Clarke_1880_Arc",6378249.145, 293.466307656]]
6223	Carthage	DATUM["D_Carthage",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]

Factory ID	Description	OGC Well-Known Text String
6224	Chua	DATUM["D_Chua",SPHEROID["International_1924",6378388,297]]
6315	Conakry 1905	DATUM["D_Conakry_1905",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6225	Corrego Alegre	DATUM["D_Corrego_Alegre",SPHEROID["International_1924",6378388,297]]
6226	Cote d'Ivoire	DATUM["D_Cote_d_Ivoire",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6274	Datum 73	DATUM["D_Datum_73",SPHEROID["International_1924",6378388,297]]
6227	Deir ez Zor	DATUM["D_Deir_ez_Zor",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6316	Dealul Piscului 1933	DATUM["D_Dealul_Piscului_1933",SPHEROID["International_1924", 6378388,297]]
6317	Dealul Piscului 1970	DATUM["D_Dealul_Piscului_1970",SPHEROID["Krasovsky_1940", 6378245,298.3]]
6314	Deutsche Hauptdreiecksnetz	DATUM["D_Deutsche_Hauptdreiecksnetz",SPHEROID["Bessel_1841", 6377397.155,299.1528128]]
6228	Douala	DATUM["D_Douala",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6230	European Datum 1950	DATUM["D_European_1950",SPHEROID["International_1924",6378388, 297]]
6231	European Datum 1987	DATUM["D_European_1987",SPHEROID["International_1924",6378388, 297]]
6229	Egypt 1907	DATUM["D_Egypt_1907",SPHEROID["Helmert_1906",6378200,298.3]]
6258	European Terrestrial Reference Frame 1989	DATUM["D_ETRF_1989",SPHEROID["WGS_1984",6378137, 298.257223563]]
6232	Fahud	DATUM["D_Fahud",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6132	Final Datum 1958	DATUM["D_FD_1958",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6233	Gandajika 1970	DATUM["D_Gandajika_1970",SPHEROID["International_1924",6378388, 297]]
6234	Garoua	DATUM["D_Garoua",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6283	Geocentric Datum of Australia 1994	DATUM["D_GDA_1994",SPHEROID["GRS_1980",6378137, 298.257222101]]
6121	Greek Geodetic Reference System 1987	DATUM["D_GGRS_1987",SPHEROID["GRS_1980",6378137, 298.257222101]]
6120	Greek	DATUM["D_Greek",SPHEROID["Bessel_1841",6377397.155,299.1528128]]
6235	Guyane Francaise	DATUM["D_Guyane_Francaise",SPHEROID["International_1924", 6378388,297]]
6255	Herat North	DATUM["D_Herat_North",SPHEROID["International_1924",6378388, 297]]
6254	Hito XVIII 1963	DATUM["D_Hito_XVIII_1963",SPHEROID["International_1924",6378388,297]]
6236	Hu Tzu Shan	DATUM["D_Hu_Tzu_Shan",SPHEROID["International_1924",6378388, 297]]
6237	Hungarian Datum 1972	DATUM["D_Hungarian_1972",SPHEROID["GRS_1967",6378160, 298.247167427]]

Factory ID	Description	OGC Well-Known Text String
6239	Indian 1954	DATUM["D_Indian_1954",SPHEROID["Everest_Adjustment_1937", 6377276.345,300.8017]]
6240	Indian 1975	DATUM["D_Indian_1975",SPHEROID["Everest_Adjustment_1937", 6377276.345,300.8017]]
6238	Indonesian Datum 1974	DATUM["D_Indonesian_1974",SPHEROID["Indonesian",6378160, 298.247]]
6241	Jamaica 1875	DATUM["D_Jamaica_1875",SPHEROID["Clarke_1880",6378249.138, 293.466307656]]
6242	Jamaica 1969	DATUM["D_Jamaica_1969",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6243	Kalianpur 1880	DATUM["D_Kalianpur_1880",SPHEROID["Everest_1830",6377299.36, 300.8017]]
6244	Kandawala	DATUM["D_Kandawala",SPHEROID["Everest_Adjustment_1937", 6377276.345,300.8017]]
6245	Kertau	DATUM["D_Kertau",SPHEROID["Everest_1830_Modified",6377304.063, 300.8017]]
6123	Kartastokoordinaatti-jarjestelma	DATUM["D_KKJ",SPHEROID["International_1924",6378388,297]]
6246	Kuwait Oil Company	DATUM["D_Kuwait_Oil_Company",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6319	Kuwait Utility	DATUM["D_Kuwait_Utility",SPHEROID["GRS_1980",6378137, 298.257222101]]
6247	La Canoa	DATUM["D_La_Canoa",SPHEROID["International_1924",6378388,297]]
6249	Lake	DATUM["D_Lake",SPHEROID["International_1924",6378388,297]]
6250	Leigon	DATUM["D_Leigon",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6251	Liberia 1964	DATUM["D_Liberia_1964",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6207	Lisbon	DATUM["D_Lisbon",SPHEROID["International_1924",6378388,297]]
6126	Lithuania 1994	DATUM["D_Lithuania_1994",SPHEROID["GRS_1980",6378137, 298.257222101]]
6288	Loma Quintana	DATUM["D_Loma_Quintana",SPHEROID["International_1924",6378388,297]]
6252	Lome	DATUM["D_Lome",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6253	Luzon 1911	DATUM["D_Luzon_1911",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6903	Madrid 1870	DATUM["D_Madrid_1870",SPHEROID["Struve_1860",6378298.3,294.73]]
6128	Madzansua —superseded by Tete	DATUM["D_Madzansua",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6256	Mahe 1971	DATUM["D_Mahe_1971",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6257	Makassar	DATUM["D_Makassar",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6259	Malongo 1987	DATUM["D_Malongo_1987",SPHEROID["International_1924",6378388, 297]]
6260	Manoca	DATUM["D_Manoca",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]

Factory ID	Description	OGC Well-Known Text String
6262	Massawa	DATUM["D_Massawa",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6261	Merchich	DATUM["D_Merchich",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6312	Militar-Geographische Institut	DATUM["D_MGI",SPHEROID["Bessel_1841",6377397.155,299.1528128]]
6264	Mhast	DATUM["D_Mhast",SPHEROID["International_1924",6378388,297]]
6263	Minna	DATUM["D_Minna",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6265	Monte Mario	DATUM["D_Monte_Mario",SPHEROID["International_1924",6378388, 297]]
6130	Moznet	DATUM["D_Moznet",SPHEROID["WGS_1984",6378137,298.257223563]]
6266	M'poraloko	DATUM["D_Mporaloko",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6268	NAD Michigan	DATUM["D_North_American_Michigan", SPHEROID["Clarke_1866_Michigan",6378450.047,294.978684677]]
6267	North American Datum 1927	DATUM["D_North_American_1927",SPHEROID["Clarke_1866", 6378206.4,294.9786982]]
6269	North American Datum 1983	DATUM["D_North_American_1983",SPHEROID["GRS_1980",6378137, 298.257222101]]
6270	Nahrwan 1967	DATUM["D_Nahrwan_1967",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6271	Naparima 1972	DATUM["D_Naparima_1972",SPHEROID["International_1924", 6378388,297]]
6902	Nord de Guerre	DATUM["D_Nord_de_Guerre",SPHEROID["Plessis_1817",6376523, 308.64]]
6318	National Geodetic Network (Kuwait)	DATUM["D_NGN",SPHEROID["WGS_1984",6378137,298.257223563]]
6273	NGO 1948	DATUM["D_NGO_1948",SPHEROID["Bessel_Modified",6377492.018, 299.1528128]]
6307	Nord Sahara 1959	DATUM["D_Nord_Sahara_1959",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6276	NSWC 9Z-2	DATUM["D_NSWC_9Z_2",SPHEROID["NWL_9D",6378145,298.25]]
6275	Nouvelle Triangulation Francaise	DATUM["D_NTF",SPHEROID["Clarke_1880_IGN",6378249.2,293.46602]]
6272	New Zealand Geodetic Datum 1949	DATUM["D_New_Zealand_1949",SPHEROID["International_1924", 6378388,297]]
6129	Observatario —superseded by Tete	DATUM["D_Observatario",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6279	OS (SN) 1980	DATUM["D_OS_SN_1980",SPHEROID["Airy_1830",6377563.396, 299.3249646]]
6277	OSGB 1936	DATUM["D_OSGB_1936",SPHEROID["Airy_1830",6377563.396, 299.3249646]]
6278	OSGB 1970 (SN)	DATUM["D_OSGB_1970_SN",SPHEROID["Airy_1830",6377563.396, 299.3249646]]

Factory ID	Description	OGC Well-Known Text String
6280	Padang 1884	DATUM["D_Padang_1884",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6281	Palestine 1923	DATUM["D_Palestine_1923",SPHEROID["Clarke_1880_Benoit", 6378300.79,293.466234571]]
6282	Pointe Noire	DATUM["D_Pointe_Noire",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6248	Provisional South American Datum 1956	DATUM["D_Provisional_S_American_1956", SPHEROID["International_1924",6378388,297]]
6284	Pulkovo 1942	DATUM["D_Pulkovo_1942",SPHEROID["Krasovsky_1940",6378245, 298.3]]
6200	Pulkovo 1995	DATUM["D_Pulkovo_1995",SPHEROID["Krasovsky_1940",6378245, 298.3]]
6285	Qatar	DATUM["D_Qatar",SPHEROID["International_1924",6378388,297]]
6286	Qatar 1948	DATUM["D_Qatar_1948",SPHEROID["Helmert_1906",6378200,298.3]]
6287	Qornoq	DATUM["D_Qornoq",SPHEROID["International_1924",6378388,297]]
6124	RT 1990	DATUM["D_RT_1990",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6291	South American Datum 1969	DATUM["D_South_American_1969", SPHEROID["GRS_1967_Truncated",6378160,298.25]]
6125	Samboja	DATUM["D_Samboja",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6292	Sapper Hill 1943	DATUM["D_Sapper_Hill_1943",SPHEROID["International_1924", 6378388,297]]
6293	Schwarzeck	DATUM["D_Schwarzeck",SPHEROID["Bessel_Namibia",6377483.865, 299.1528128]]
6294	Segora	DATUM["D_Segora",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6295	Serindung	DATUM["D_Serindung",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6308	Stockholm 1938	DATUM["D_Stockholm_1938",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6138	Alaska, St. George Island	DATUM["D_St_George_Island",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6136	Alaska, St. Lawrence Island	DATUM["D_St_Lawrence_Island",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6137	Alaska, St. Paul Island	DATUM["D_St_Paul_Island",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6296	Sudan	DATUM["D_Sudan",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6297	Tananarive 1925	DATUM["D_Tananarive_1925",SPHEROID["International_1924",6378388,297]]
6127	Tete	DATUM["D_Tete",SPHEROID["Clarke_1866",6378206.4,294.9786982]]
6298	Timbalai 1948	DATUM["D_Timbalai_1948",SPHEROID["Everest_Definition_1967", 6377298.556,300.8017]]
6299	TM65	DATUM["D_TM65",SPHEROID["Airy_Modified",6377340.189, 299.3249646]]

Factory ID	Description	OGC Well-Known Text String
6300	TM75	DATUM["D_TM75",SPHEROID["Airy_Modified",6377340.189, 299.3249646]]
6301	Tokyo	DATUM["D_Tokyo",SPHEROID["Bessel_1841",6377397.155,299.1528128]]
6302	Trinidad 1903	DATUM["D_Trinidad_1903",SPHEROID["Clarke_1858",6378293.639, 294.260676369]]
6303	Trucial Coast 1948	DATUM["D_Trucial_Coast_1948",SPHEROID["Helmert_1906",6378200, 298.3]]
6304	Voirol 1875	DATUM["D_Voirol_1875",SPHEROID["Clarke_1880_IGN",6378249.2, 293.46602]]
6305	Voirol Unifie 1960	DATUM["D_Voirol_Unifie_1960",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6322	WGS 1972	DATUM["D_WGS_1972",SPHEROID["WGS_1972",6378135,298.26]]
6324	WGS 1972 Transit Broadcast Ephemeris	DATUM["D_WGS_1972_BE",SPHEROID["WGS_1972",6378135,298.26]]
6326	WGS 1984	DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137, 298.257223563]]
6309	Yacare	DATUM["D_Yacare",SPHEROID["International_1924",6378388,297]]
6310	Yoff	DATUM["D_Yoff",SPHEROID["Clarke_1880_IGN",6378249.2,293.46602]]
6311	Zanderij	DATUM["D_Zanderij",SPHEROID["International_1924",6378388,297]]
6600	Anguilla 1957	DATUM["D_Anguilla_1957",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6133	Estonia 1992	DATUM["D_Estonia_1992",SPHEROID["GRS_1980",6378137, 298.257222101]]
6602	Dominica 1945	DATUM["D_Dominica_1945",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6603	Grenada 1953	DATUM["D_Grenada_1953",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6609	NAD_1927 (CGQ77)	DATUM["D_NAD_1927_CGQ77",SPHEROID["Clarke_1866", 6378206.4,294.9786982]]
6608	NAD 1927 (1976)	DATUM["D_NAD_1927_Definition_1976",SPHEROID["Clarke_1866", 6378206.4,294.9786982]]
6134	PDO Survey Datum 1993	DATUM["D_PDO_1993",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6605	St. Kitts 1955	DATUM["D_St_Kitts_1955",SPHEROID["Clarke_1880_RGS",6378249.145,293.465]]
6606	St. Lucia 1955	DATUM["D_St_Lucia_1955",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6607	St. Vincent 1945	DATUM["D_St_Vincent_1945",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6140	NAD 1983 (Canadian Spatial Reference System)	DATUM["D_North_American_1983_CSRS98",SPHEROID["GRS_1980", 6378137,298.257222101]]
6141	Israel	DATUM["D_Israel",SPHEROID["GRS_1980",6378137,298.257222101]]
6142	Locodjo 1965	DATUM["D_Locodjo_1965",SPHEROID["Clarke_1880_RGS",6378249.145,293.465]]
6143	Abidjan 1987	DATUM["D_Abidjan_1987",SPHEROID["Clarke_1880_RGS",6378249.145,293.465]]
6144	Kalianpur 1937	DATUM["D_Kalianpur_1937",SPHEROID["Everest_Adjustment_1937", 6377276.345,300.8017]]

Factory ID	Description	OGC Well-Known Text String
6145	Kalianpur 1962	DATUM["D_Kalianpur_1962",SPHEROID["Everest_Definition_1962", 6377301.243,300.8017255]]
6146	Kalianpur 1975	DATUM["D_Kalianpur_1975",SPHEROID["Everest_Definition_1975", 6377299.151,300.8017255]]
6147	Hanoi 1972	DATUM["D_Hanoi_1972",SPHEROID["Krasovsky_1940",6378245,298.3]]
6148	Hartebeesthoek 1994	DATUM["D_Hartebeesthoek_1994",SPHEROID["WGS_1984",6378137, 298.257223563]]
6149	CH 1903	DATUM["D_CH1903",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6150	CH 1903+	DATUM["D_CH1903+",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6151	Swiss Terrestrial Reference Frame 1995	DATUM["D_Swiss_TRF_1995",SPHEROID["GRS_1980",6378137, 298.257222101]]
6152	NAD 1983 (HARN)	DATUM["D_North_American_1983_HARN",SPHEROID["GRS_1980", 6378137,298.257222101]]
6153	Rassadiran	DATUM["D_Rassadiran",SPHEROID["International_1924",6378388,297]]
6154	ED 1950 (ED77)	DATUM["D_European_1950_ED77",SPHEROID["International_1924", 6378388,297]]
6135	Old Hawaiian	DATUM["D_Old_Hawaiian",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6601	Antigua Astro 1943	DATUM["D_Antigua_1943",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6604	Montserrat Astro 1958	DATUM["D_Montserrat_1958",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
6139	Puerto Rico	DATUM["D_Puerto_Rico",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
6131	Indian 1960	DATUM["D_Indian_1960",SPHEROID["Everest_Adjustment_1937", 6377276.345,300.8017]]
6155	Dabola	DATUM["D_Dabola",SPHEROID["Clarke_1880_RGS",6378249.145, 293.465]]
6156	S-JTSK	DATUM["D_S_JTSK",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
6165	Bissau	DATUM["D_Bissau",SPHEROID["International_1924",6378388,297]]
106101	Estonia 1937	DATUM["D_Estonia_1937",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
106102	Hermannskogel	DATUM["D_Hermannskogel",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
106103	Sierra Leone 1960	DATUM["D_Sierra_Leone_1960",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
106201	European 1979	DATUM["D_European_1979",SPHEROID["International_1924",6378388, 297]]
106202	Everest, Bangladesh	DATUM["D_Everest_Bangladesh", SPHEROID["Everest_Adjustment_1937",6377276.345,300.8017]]
106203	Everest, India and Nepal	DATUM["D_Everest_India_Nepal", SPHEROID["Everest_Definition_1962",6377301.243,300.8017255]]
106204	Hjorsey 1955	DATUM["D_Hjorsey_1955",SPHEROID["International_1924",6378388, 297]]

Factory ID	Description	OGC Well-Known Text String
106205	Hong Kong 1963	DATUM[″D_Hong_Kong_1963″,SPHEROID[″International_1924″, 6378388,297]]
106206	Oman	DATUM[″D_Oman″,SPHEROID[″Clarke_1880_RGS″,6378249.145, 293.465]]
106207	South Asia Singapore	DATUM[″D_South_Asia_Singapore″,SPHEROID[″Fischer_Modified″, 6378155,298.3]]
106208	Ayabelle Lighthouse	DATUM[″D_Ayabelle″,SPHEROID[″Clarke_1880_RGS″,6378249.145, 293.465]]
106211	Point 58	DATUM[″D_Point_58″,SPHEROID[″Clarke_1880_RGS″,6378249.145, 293.465]]
106212	Astro Beacon E 1945	DATUM[″D_Beacon_E_1945″,SPHEROID[″International_1924″,6378388, 297]]
106213	Tern Island Astro 1961	DATUM[″D_Tern_Island_1961″,SPHEROID[″International_1924″, 6378388,297]]
106214	Astronomical Station 1952	DATUM[″D_Astro_1952″,SPHEROID[″International_1924″,6378388,297]]
106215	Bellevue IGN	DATUM[″D_Bellevue_IGN″,SPHEROID[″International_1924″,6378388, 297]]
106216	Canton Astro 1966	DATUM[″D_Canton_1966″,SPHEROID[″International_1924″,6378388, 297]]
106217	Chatham Island Astro 1971	DATUM[″D_Chatham_Island_1971″,SPHEROID[″International_1924″, 6378388,297]]
106218	DOS 1968	DATUM[″D_DOS_1968″,SPHEROID[″International_1924″,6378388,297]]
106219	Easter Island 1967	DATUM[″D_Easter_Island_1967″,SPHEROID[″International_1924″, 6378388,297]]
106220	Guam 1963	DATUM[″D_Guam_1963″,SPHEROID[″Clarke_1866″,6378206.4, 294.9786982]]
106221	GUX 1 Astro	DATUM[″D_GUX_1″,SPHEROID[″International_1924″,6378388,297]]
106222	Johnston Island 1961	DATUM[″D_Johnston_Island_1961″,SPHEROID[″International_1924″, 6378388,297]]
106259	Kusaie Astro 1951	DATUM[″D_Kusaie_1951″,SPHEROID[″International_1924″,6378388, 297]]
106224	Midway Astro 1961	DATUM[″D_Midway_1961″,SPHEROID[″International_1924″,6378388, 297]]
106226	Pitcairn Astro 1967	DATUM[″D_Pitcairn_1967″,SPHEROID[″International_1924″,6378388, 297]]
106227	Santo DOS 1965	DATUM[″D_Santo_DOS_1965″,SPHEROID[″International_1924″, 6378388,297]]
106228	Viti Levu 1916	DATUM[″D_Viti_Levu_1916″,SPHEROID[″Clarke_1880_RGS″, 6378249.145,293.465]]
106229	Wake-Eniwetok 1960	DATUM[″D_Wake_Eniwetok_1960″,SPHEROID[″Hough_1960″,6378270, 297]]
106230	Wake Island Astro 1952	DATUM[″D_Wake_Island_1952″,SPHEROID[″International_1924″, 6378388,297]]
106231	Anna 1 Astro 1965	DATUM[″D_Anna_1_1965″,SPHEROID[″Australian″,6378160,298.25]]
106232	Gan 1970	DATUM[″D_Gan_1970″,SPHEROID[″International_1924″,6378388,297]]

Factory ID	Description	OGC Well-Known Text String
106233	ISTS 073 Astro 1969	DATUM["D_ISTS_073_1969",SPHEROID["International_1924",6378388, 297]]
106234	Kerguelen Island 1949	DATUM["D_Kerguelen_Island_1949",SPHEROID["International_1924", 6378388,297]]
106235	Reunion	DATUM["D_Reunion",SPHEROID["International_1924",6378388,297]]
106237	Ascension Island 1958	DATUM["D_Ascension_Island_1958",SPHEROID["International_1924", 6378388,297]]
106238	Astro DOS 71/4	DATUM["D_DOS_71_4",SPHEROID["International_1924",6378388,297]]
106239	Cape Canaveral	DATUM["D_Cape_Canaveral",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
106240	Fort Thomas 1955	DATUM["D_Fort_Thomas_1955",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
106241	Graciosa Base SW 1948	DATUM["D_Graciosa_Base_SW_1948",SPHEROID["International_1924", 6378388,297]]
106242	ISTS 061 Astro 1968	DATUM["D_ISTS_061_1968",SPHEROID["International_1924",6378388, 297]]
106243	L.C. 5 Astro 1961	DATUM["D_LC5_1961",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
106245	Observ Meteorologico 1939	DATUM["D_Observ_Meteorologico_1939", SPHEROID["International_1924",6378388,297]]
106246	Pico de Las Nieves	DATUM["D_Pico_de_Las_Nieves",SPHEROID["International_1924", 6378388,297]]
106247	Porto Santo 1936	DATUM["D_Porto_Santo_1936",SPHEROID["International_1924", 6378388,297]]
106249	Sao Braz	DATUM["D_Sao_Braz",SPHEROID["International_1924",6378388,297]]
106250	Selvagem Grande 1938	DATUM["D_Selvagem_Grande_1938",SPHEROID["International_1924", 6378388,297]]
106251	Tristan Astro 1968	DATUM["D_Tristan_1968",SPHEROID["International_1924",6378388, 297]]
106252	American Samoa 1962	DATUM["D_Samoa_1962",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
106253	Camp Area Astro	DATUM["D_Camp_Area",SPHEROID["International_1924",6378388,297]]
106254	Deception Island	DATUM["D_Deception_Island",SPHEROID["Clarke_1880_RGS", 6378249.145,293.465]]
106255	Gunung Segara	DATUM["D_Gunung_Segara",SPHEROID["Bessel_1841",6377397.155, 299.1528128]]
106257	S-42 Hungary	DATUM["D_S42_Hungary",SPHEROID["Krasovsky_1940",6378245, 298.3]]
106260	Alaskan Islands	DATUM["D_Alaskan_Islands",SPHEROID["Clarke_1866",6378206.4, 294.9786982]]
106261	Hong Kong 1980	DATUM["D_Hong_Kong_1980", SPHEROID["International_1924", 6378388,297]
106262	Datum Lisboa Bessel	DATUM["D_Datum_Lisboa_Bessel", SPHEROID["Bessel_1841", 6377397.155,299.1528128]]
106263	Datum Lisboa Hayford	DATUM["D_Datum_Lisboa_Hayford", SPHEROID["International_1924", 6378388,297]]

Factory ID	Description	OGC Well-Known Text String
106264	Reseau Geodesique Francais 1993	DATUM[″D_RGF_1993″, SPHEROID[″GRS_1980″, 6378137,298.257222101]]
106265	New Zealand Geodetic Datum 2000	DATUM[″D_NZGD_2000″, SPHEROID[″GRS_1980″,6378137, 298.257222101]]

Prime Meridians

The following table lists a representative sample of the prime meridians supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported prime meridians, see the file **INFORMIXDIR/extend/spatial.***version***/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
8901	Greenwich 0°00′00″	PRIMEM[″Greenwich″,0]
8912	Athens 23°42′58″.815 E	PRIMEM[″Athens″,23.7163375]
8907	Bern 7°26′22″.5 E	PRIMEM[″Bern″,7.439583333333333]
8904	Bogota 74°04′51″.3 W	PRIMEM[″Bogota″,-74.08091666666667]
8910	Brussels 4°22′04″.71 E	PRIMEM[″Brussels″,4.367975]
8909	Ferro 17°40′00″ W	PRIMEM[″Ferro″,-17.66666666666667]
8908	Jakarta 106°48′27″.79 E	PRIMEM[″Jakarta″,106.8077194444444]
8902	Lisbon 9°07′54″.862 W	PRIMEM[″Lisbon″,-9.131906111111112]
8905	Madrid 3°41′16″.58 W	PRIMEM[″Madrid″,-3.687938888888889]
8913	Oslo 10°43′22″.5 E	PRIMEM[″Oslo″,10.72291666666667]
8903	Paris 2°20′14″.025 E	PRIMEM[″Paris″,2.337229166666667]
8906	Rome 12°27′08″.4 E	PRIMEM[″Rome″,12.45233333333333]
8911	Stockholm 18°03′29″.8 E	PRIMEM[″Stockholm″,18.05827777777778]

Projection Parameters

The following table lists a representative sample of the projection parameters supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported projection parameters, see the file **INFORMIXDIR/extend/spatial.***version***/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
100001	False Easting	PARAMETER[″False_Easting″,0]
100002	False Northing	PARAMETER[″False_Northing″,0]
100003	Scale Factor	PARAMETER[″Scale_Factor″,1]
100004	Azimuth	PARAMETER[″Azimuth″,45]
100010	Central Meridian	PARAMETER[″Central_Meridian″,0]
100011	Longitude Of Origin	PARAMETER[″Longitude_Of_Origin″,0]
100012	Longitude Of Center	PARAMETER[″Longitude_Of_Center″,-75]
100013	Longitude Of 1st Point	PARAMETER[″Longitude_Of_1st_Point″,0]
100014	Longitude Of 2nd Point	PARAMETER[″Longitude_Of_2nd_Point″,60]
100020	Central Parallel	PARAMETER[″Central_Parallel″,0]

Factory ID	Description	OGC Well-Known Text String
100021	Latitude Of Origin	PARAMETER[″Latitude_Of_Origin″,0]
100022	Latitude Of Center	PARAMETER[″Latitude_Of_Center″,40]
100023	Latitude Of 1st Point	PARAMETER[″Latitude_Of_1st_Point″,0]
100024	Latitude Of 2nd Point	PARAMETER[″Latitude_Of_2nd_Point″,60]
100025	Standard Parallel 1	PARAMETER[″Standard_Parallel_1″,60]
100026	Standard Parallel 2	PARAMETER[″Standard_Parallel_2″,60]
100027	Pseudo Standard Parallel 1	PARAMETER[″Pseudo_Standard_Parallel_1″,60]
100037	X Scale	PARAMETER[″X_Scale″,1]
100038	Y Scale	PARAMETER[″Y_Scale″,1]
100039	XY Plane Rotation	PARAMETER[″XY_Plane_Rotation″,0]
100040	X Axis Translation	PARAMETER[″X_Axis_Translation″,0]
100041	Y Axis Translation	PARAMETER[″Y_Axis_Translation″,0]
100042	Z Axis Translation	PARAMETER[″Z_Axis_Translation″,0]
100043	X Axis Rotation	PARAMETER[″X_Axis_Rotation″,0]
100044	Y Axis Rotation	PARAMETER[″Y_Axis_Rotation″,0]
100045	Z Axis Rotation	PARAMETER[″Z_Axis_Rotation″,0]
100046	Scale Difference	PARAMETER[″Scale_Difference″,0]
100047	Dataset Name	PARAMETER[″Dataset_″,0]

Map Projections

The following table lists a representative sample of the map projections supported by the IBM Informix Spatial DataBlade Module. For a complete list of supported map projections, see the file **INFORMIXDIR/extend/spatial.***version***/include/pedef.h**.

Factory ID	Description	OGC Well-Known Text String
43001	Plate Carree	PROJECTION[″Plate_Carree″]
43002	Equidistant Cylindrical	PROJECTION[″Equidistant_Cylindrical″]
43003	Miller Cylindrical	PROJECTION[″Miller_Cylindrical″]
43004	Mercator	PROJECTION[″Mercator″]
43005	Gauss-Kruger	PROJECTION[″Gauss_Kruger″]
43006	Transverse Mercator	PROJECTION[″Transverse_Mercator″]
43007	Albers	PROJECTION[″Albers″]
43008	Sinusoidal	PROJECTION[″Sinusoidal″]
43009	Mollweide	PROJECTION[″Mollweide″]
43010	Eckert VI	PROJECTION[″Eckert_VI″]
43011	Eckert V	PROJECTION[″Eckert_V″]
43012	Eckert IV	PROJECTION[″Eckert_IV″]
43013	Eckert III	PROJECTION[″Eckert_III″]
43014	Eckert II	PROJECTION[″Eckert_II″]
43015	Eckert I	PROJECTION[″Eckert_I″]

Factory ID	Description	OGC Well-Known Text String
43016	Gall Stereographic	PROJECTION[″Gall_Stereographic″]
43017	Behrmann	PROJECTION[″Behrmann″]
43018	Winkel I	PROJECTION[″Winkel_I″]
43019	Winkel II	PROJECTION[″Winkel_II″]
43020	Lambert Conformal Conic	PROJECTION[″Lambert_Conformal_Conic″]
43021	Polyconic	PROJECTION[″Polyconic″]
43022	Quartic Authalic	PROJECTION[″Quartic_Authalic″]
43023	Loximuthal	PROJECTION[″Loximuthal″]
43024	Bonne	PROJECTION[″Bonne″]
43025	Hotine 2 Pt Natural Origin	PROJECTION[″Hotine_Oblique_Mercator_Two_ Point_Natural_Origin″]
43026	Stereographic	PROJECTION[″Stereographic″]
43027	Equidistant Conic	PROJECTION[″Equidistant_Conic″]
43028	Cassini	PROJECTION[″Cassini″]
43029	Van der Grinten I	PROJECTION[″Van_der_Grinten_I″]
43030	Robinson	PROJECTION[″Robinson″]
43031	Two-Point Equidistant	PROJECTION[″Two_Point_Equidistant″]
43032	Azimuthal Equidistant	PROJECTION[″Azimuthal_Equidistant″]
43033	Lambert Azimuthal Equal Area	PROJECTION[″Lambert_Azimuthal_Equal_Area″]
43034	Cylindrical Equal Area	PROJECTION[″Cylindrical_Equal_Area″]
43035	Hotine 2 Point Center	PROJECTION[″Hotine_Oblique_Mercator_Two_ Point_Center″]
43036	Hotine Azimuth Natural Origin	PROJECTION[″Hotine_Oblique_Mercator_Azimuth _Natural_Origin″]
43037	Hotine Azimuth Center	PROJECTION[″Hotine_Oblique_Mercator_Azimuth _Center″]
43038	Double Stereographic	PROJECTION[″Double_Stereographic″]
43039	Krovak Oblique Lambert	PROJECTION[″Krovak″]
43040	New Zealand Map Grid	PROJECTION[″New_Zealand_Map_Grid″]
43041	Orthographic	PROJECTION[″Orthographic″]
43042	Winkel Tripel	PROJECTION[″Winkel_Tripel″]
43043	Aitoff	PROJECTION[″Aitoff″]
43044	Hammer Aitoff	PROJECTION[″Hammer_Aitoff″]
43045	Flat Polar Quartic	PROJECTION[″Flat_Polar_Quartic″]
43046	Craster Parabolic	PROJECTION[″Craster_Parabolic″]
43047	Gnomonic	PROJECTION[″Gnomonic″]
43048	Bartholomew Times	PROJECTION[″Times″]
43049	Vertical Near-Side Perspective	PROJECTION[″Vertical_Near_Side_Perspective″]

Appendix C. OGC Well-Known Text Representation of Geometry

Each geometry type has a well-known text representation from which new instances can be constructed or existing instances can be converted to textual form for alphanumeric display.

Using Well-Known Text Representation in a C Program

The well-known text representation of geometry can be incorporated into a C program. The structure for such an implementation is defined below. The notation {}* denotes zero or more repetitions of the tokens within the braces. The braces do not appear in the output token list.

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

<Point Text> := EMPTY
        |    <Point>
        | Z  <PointZ>
        | M  <PointM>
        | ZM <PointZM>


<Point> :=  <x>  <y>
       <x> := double precision literal
       <y> := double precision literal

<PointZ> :=  <x>  <y>  <z>
       <x> := double precision literal
       <y> := double precision literal
       <z> := double precision literal

<PointM> :=  <x>  <y>  <m>
        <x> := double precision literal
        <y> := double precision literal
        <m> := double precision literal


<PointZM> :=  <x>  <y>  <z>  <m>
        <x> := double precision literal
        <y> := double precision literal
        <z> := double precision literal
```

```
        <m> := double precision literal


<LineString Text> := EMPTY
          |    ( <Point Text >   {,  <Point Text> }*  )
          | Z  ( <PointZ Text >  {,  <PointZ Text> }*  )
          | M  ( <PointM Text >  {,  <PointM Text> }*  )
          | ZM ( <PointZM Text > {,  <PointZM Text> }* )


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

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

<MultiLineString Text> := EMPTY
       | ( <LineString Text > {,< LineString Text>}*  )

<MultiPolygon Text> := EMPTY
       | ( < Polygon Text > {,  < Polygon Text > }*  )
```

Using Well-Known Text Representation in an SQL Editor

Since the well-known text representation is *text*, it can be typed into an SQL script or directly into an SQL editor. The text is converted to and from a geometry by a function. Functions that convert text to geometry have the following syntax:

```
function ('text description',SRID)
```

Example:

```
ST_PointFromText('point zm (10.01 20.04 3.2 9.5)', 1)
```

The spatial reference identifier, SRID—the primary key to the **spatial_references** table—identifies the possible spatial reference systems within an Informix instance. An SRID is assigned to a spatial column when it is created. Before a geometry can be inserted into a spatial column, its SRID must match the SRID of the spatial column.

The *text description* is made up of three basic components enclosed in single quotes:

```
'geometry type [coordinate type] [coordinate list]'
```

The *geometry type* is defined as one of the following: point, linestring, polygon, multipoint, multilinestring, or multipolygon.

The *coordinate type* specifies whether or not the geometry has Z coordinates or measures. Leave this argument blank if the geometry has neither; otherwise, set the coordinate type to Z for geometries containing Z coordinates, M for geometries with measures, and ZM for geometries that have both.

The *coordinate list* defines the double-precision vertices of the geometry. Coordinate lists are comma-delimited and enclosed by parentheses. Geometries having multiple components require sets of parentheses to enclose each component part. If the geometry is empty, the EMPTY keyword replaces the coordinates.

The following examples provide a complete list of all possible permutations of the text description portion of the text representation.

Geometry type	Text Description	Comment
ST_Point	'point empty'	Empty point
ST_Point	'point z empty'	Empty point with Z coordinate
ST_Point	'point m empty'	Empty point with measure
ST_Point	'point zm empty'	Empty point with Z coordinate and measure
ST_Point	'point (10.05 10.28)'	Point
ST_Point	'point z (10.05 10.28 2.51)'	Point with Z coordinate
ST_Point	'point m (10.05 10.28 4.72)'	Point with measure
ST_Point	'point zm (10.05 10.28 2.51 4.72)'	Point with Z coordinate and measure
ST_LineString	'linestring empty'	Empty linestring
ST_LineString	'linestring z empty'	Empty linestring with Z coordinates
ST_LineString	'linestring m empty'	Empty linestring with measures
ST_LineString	'linestring zm empty'	Empty linestring with Z coordinates and measures
ST_LineString	'linestring (10.05 10.28 , 20.95 20.89)'	Linestring
ST_LineString	'linestring z (10.05 10.28 3.09, 20.95 31.98 4.72, 21.98 29.80 3.51)'	Linestring with Z coordinates
ST_LineString	'linestring m (10.05 10.28 5.84, 20.95 31.98 9.01, 21.98 29.80 12.84)'	Linestring with measures
ST_LineString	'linestring zm (10.05 10.28 3.09 5.84, 20.95 31.98 4.72 9.01, 21.98 29.80 3.51 12.84)'	Linestring with Z coordinates and measures
ST_Polygon	'polygon empty'	Empty polygon
ST_Polygon	'polygon z empty'	Empty polygon with Z coordinates
ST_Polygon	'polygon m empty	Empty polygon with measures
ST_Polygon	'polygon zm empty'	Empty polygon with Z coordinates and measures
ST_Polygon	'polygon ((10 10, 10 20, 20 20, 20 15, 10 10))'	Polygon
ST_Polygon	'polygon z ((10 10 3, 10 20 3, 20 20 3, 20 15 4, 10 10 3))'	Polygon with Z coordinates
ST_Polygon	'polygon m ((10 10 8, 10 20 9, 20 20 9, 20 15 9, 10 10 8))'	Polygon with measures
ST_Polygon	'polygon zm ((10 10 3 8, 10 20 3 9, 20 20 3 9, 20 15 4 9, 10 10 3 8))'	Polygon with Z coordinates and measures
ST_MultiPoint	'multipoint empty'	Empty multipoint
ST_MultiPoint	'multipoint z empty'	Empty multipoint with Z coordinates
ST_MultiPoint	'multipoint m empty'	Empty multipoint with measures
ST_MultiPoint	'multipoint zm empty'	Empty multipoint with Z coordinates and measures
ST_MultiPoint	'multipoint (10 10, 20 20)'	Multipoint with two points
ST_MultiPoint	'multipoint z (10 10 2, 20 20 3)'	Multipoint with Z coordinates
ST_MultiPoint	'multipoint m (10 10 4, 20 20 5)'	Multipoint with measures
ST_MultiPoint	'multipoint zm (10 10 2 4, 20 20 3 5)'	Multipoint with Z coordinates and measures
ST_MultiLineString	'multilinestring empty'	Empty multilinestring
ST_MultiLineString	'multilinestring z empty'	Empty multilinestring with Z coordinates
ST_MultiLineString	'multilinestring m empty'	Empty multilinestring with measures
ST_MultiLineString	'multilinestring zm empty'	Empty multilinestring with Z coordinates and measures
ST_MultiLineString	'multilinestring ((10.05 10.28 , 20.95 20.89),(20.95 20.89, 31.92 21.45))'	Multilinestring

Geometry type	Text Description	Comment
ST_MultiLineString	'multilinestring z ((10.05 10.28 3.4, 20.95 20.89 4.5),(20.95 20.89 4.5, 31.92 21.45 3.6))'	Multilinestring with Z coordinates
ST_MultiLineString	'multilinestring m ((10.05 10.28 8.4, 20.95 20.89 9.5),(20.95 20.89 9.5, 31.92 21.45 8.6))'	Multilinestring with measures
ST_MultiLineString	'multilinestring zm ((10.05 10.28 3.4 8.4, 20.95 20.89 4.5 9.5), (20.95 20.89 4.5 9.5, 31.92 21.45 3.6 8.6))'	Multilinestring with Z coordinates and measures
ST_MultiPolygon	'multipolygon empty'	Empty multipolygon
ST_MultiPolygon	'multipolygon z empty'	Empty multipolygon with Z coordinates
ST_MultiPolygon	'multipolygon m empty'	Empty multipolygon with measures
ST_MultiPolygon	'multipolygon zm empty'	Empty multipolygon with Z coordinates and measures
ST_MultiPolygon	'multipolygon (((10 10, 10 20, 20 20, 20 15, 10 10), (50 40, 50 50, 60 50, 60 40, 50 40)))'	Multipolygon
ST_MultiPolygon	'multipolygon z (((10 10 7, 10 20 8, 20 20 7, 20 15 5, 10 10 7), (50 40 6, 50 50 6, 60 50 5, 60 40 6, 50 40 7)))'	Multipolygon with Z coordinates
ST_MultiPolygon	'multipolygon m (((10 10 2, 10 20 3, 20 20 4, 20 15 5, 10 10 2), (50 40 7, 50 50 3, 60 50 4, 60 40 5, 50 40 7)))'	Multipolygon with measures
ST_MultiPolygon	'multipolygon zm (((10 10 7 2, 10 20 8 3, 20 20 7 4, 20 15 5 5, 10 10 7 2), (50 40 6 7, 50 50 6 3, 60 50 5 4, 60 40 6 5, 50 40 7 7)))'	Multipolygon with Z coordinates and measures

Modified Well-Known Text Representation

The IBM Informix software provides an additional modified well-known text representation for loading text strings directly into geometry columns without filtering the string through one of the Spatial DataBlade text functions. By placing the SRID in front of the text description, you can insert the resulting text string directly into a spatial column. The load statement in the DB-Access utility reads text files generated with this format and inserts the modified well-known text representation into the geometry columns.

To create a modified well-known text representation, remove the quotes and precede the text description with the SRID separated by a space.

For example, the well-known text representation of a point in the **ST_PointFromText()** function:

```
ST_PointFromText('Point zm (10.98  29.91 10.2 9.1)',1)
```

converts to:

```
1 ST_Point zm(10.98  29.91 10.2 9.1)
```

You can write the modified text string into a file and separate it from other column values by the standard delimiter.

Appendix D. OGC Well-Known Binary Representation of Geometry

The well-known binary representation for OGC geometry (WKBGeometry), provides a portable representation of a geometry value as a contiguous stream of bytes. It permits geometry values to be exchanged between a client application and an SQL database in binary form.

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 used for a geometry byte stream is described by a one-byte tag that precedes the serialized bytes. The only difference between the two encodings of geometry is byte order. The XDR encoding is big endian, while the NDR encoding is little endian.

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.

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).

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).

Conversion Between the NDR and XDR Representations of WKB Geometry

Conversion between the NDR and XDR data types for unsigned integers and doubles is a simple operation involving reversing the order of bytes within each unsigned integer or double in the byte stream.

Description of WKBGeometry Byte Streams

The well-known binary representation for geometry is described below. The basic building block is the byte stream for a point that consists of two doubles. The byte streams for other geometries are built using the byte streams for geometries that have already been defined.

Important: These structures are only intended to define the stream of bytes that is transmitted between the client and server. They should not be used as C-language data structures. See Figure D-1 on page D-4 for an example of such a byte stream.

```
// 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;
    double z;
    double m;
};

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

enum wkbGeometryType {
 wkbPoint                   = 1,
  wkbLineString             = 2,
  wkbPolygon                = 3,
  wkbMultiPoint             = 4,
  wkbMultiLineString        = 5,
  wkbMultiPolygon           = 6,
  wkbGeometryCollection     = 7,
  wkbPointZ                 = 1001,
  wkbLineStringZ            = 1002,
  wkbPolygonZ               = 1003,
  wkbMultiPointZ            = 1004,
  wkbMultiLineStringZ       = 1005,
  wkbMultiPolygonZ          = 1006,
  wkbGeometryCollectionZ    = 1007,
  wkbPointM                 = 2001,
  wkbLineStringM            = 2002,
  wkbPolygonM               = 2003,
  wkbMultiPointM            = 2004,
  wkbMultiLineStringM       = 2005,
  wkbMultiPolygonM          = 2006,
  wkbGeometryCollectionM    = 2007,
  wkbPointZM                = 3001,
  wkbLineStringZM           = 3002,
  wkbPolygonZM              = 3003,
  wkbMultiPointZM           = 3004,
  wkbMultiLineStringZM      = 3005,
  wkbMultiPolygonZM         = 3006,
  wkbGeometryCollectionZM   = 3007
};

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;
    WKBMultiPolygon       mpolygon;
}
};


WKBGeometryCollection {
byte        byte_order;
uint32      wkbType;                             // 7
uint32      num_wkbGeometries;
WKBGeometry              wkbGeometries[num_wkbGeometries]
}
```

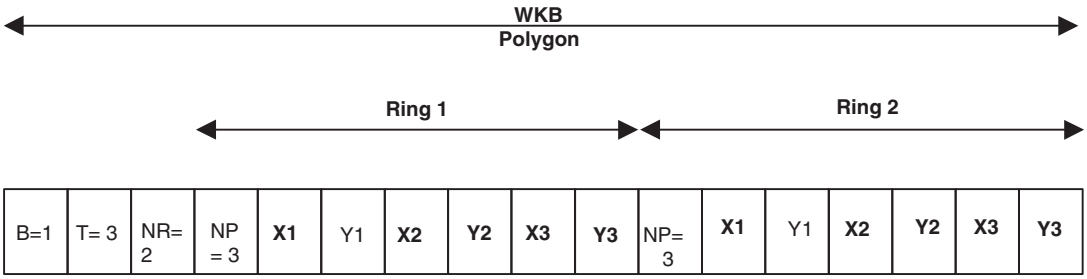


Figure D-1. Well-known binary representation for a geometry object in NDR format(B=1) of type polygon (T=3) with two rings (NR = 2) and each ring having three points (NP = 3)

The number of bytes in each box is shown below:

1	4	4	4	8	8	8	8	8	8	4	8	8	8	8	8	8

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.*

These assertions imply the following for rings, polygons, and multipolygons:

- **Linear rings.** Rings are simple and closed, which means that linear rings may not self intersect.
- **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.
- **Multipolygons.** The interiors of two polygons that are elements of a multipolygon may not intersect. The boundaries of any two polygons that are elements of a multipolygon may touch at only a finite number of points.

Appendix E. ESRI Shape Representation

The ESRI shape representation is an industry standard format that is used in ESRI shape files. You can use the **SE_AsShape()** function to retrieve geometries from a table in shape format. You can insert shape format data into database tables by using the **SE_GeomFromShape()**, **SE_PointFromShape()**, **SE_PolyFromShape()**, and other similar functions. All functions are described in detail in Chapter 8, "SQL Functions," on page 8-1.

This appendix provides background information about the ESRI shape representation.

Shape Type Values

A shape type of 0 indicates a null shape with no geometric data for the shape.

Value	Shape Type	Spatial Data Type
0	Null Shape	Empty ST_Geometry
1	Point	ST_Point
3	PolyLine	ST_MultiLineString
5	Polygon	ST_MultiPolygon
8	MultiPoint	ST_MultiPoint
9	PointZ	ST_Point with Z coordinates
10	PolyLineZ	ST_MultiLineString with Z coordinates
11	PointZM	ST_Point with Z coordinates and measures
13	PolyLineZM	ST_MultiLineString with Z coordinates and measures
15	PolygonZM	ST_MultiPolygon with Z coordinates and measures
18	MultiPointZM	ST_MultiPoint with Z coordinates and measures
19	PolygonZ	ST_MultiPolygon with Z coordinates
20	MultiPointZ	ST_MultiPoint with Z coordinates
21	PointM	ST_Point with measures
23	PolyLineM	ST_MultiLineString with measures
25	PolygonM	ST_MultiPolygon with measures
28	MultiPointM	ST_MultiPoint with measures

Shape types not specified above (2, 4, 6, and so on) are reserved for future use.

The ST_LineString and ST_Polygon Spatial data types do not have equivalent shape type values.

Shape Types in XY Space

This section describes shapes with X and Y coordinates.

Point

A Point consists of a pair of double-precision coordinates in the order X, Y. The following table shows Point byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	1	Integer	1	Little endian
Byte 4	X	X	Double	1	Little endian
Byte 12	Y	Y	Double	1	Little endian

MultiPoint

A MultiPoint consists of a collection of points. The bounding box is stored in the order Xmin, Ymin, Xmax, Ymax. The following table shows MultiPoint byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	8	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumPoints	NumPoints	Integer	1	Little endian
Byte 40	Points	Points	Point	NumPoints	Little endian

PolyLine

A PolyLine is an ordered set of vertices that consists of one or more parts. A part is a connected sequence of two or more points. Parts may be connected to and may intersect one another.

Because this specification does not forbid consecutive points with identical coordinates, shapefile readers must handle such cases. On the other hand, the degenerate zero length parts that might result are not allowed.

The fields for a PolyLine are described in detail below:

- **Box.** The bounding box for the PolyLine, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of parts in the PolyLine
- **NumPoints.** The total number of points for all parts
- **Parts.** An array of length NumParts. Stores, for each PolyLine, the index of its first point in the points array

  Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each part in the PolyLine are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.

The following table shows PolyLine byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	3	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts)

Polygon

A Polygon consists of one or more rings. A ring is a connected sequence of four or more points that form a closed, non-self-intersecting loop. A polygon may contain multiple outer rings. The order of vertices or orientation for a ring indicates which side of the ring is the interior of the polygon. The neighborhood to the right of an observer walking along the ring in vertex order is the neighborhood inside the polygon. Vertices of rings defining holes in polygons are in a counterclockwise direction. Vertices for a single, ringed polygon are, therefore, always in clockwise order. The rings of a polygon are referred to as its parts.

Because this specification does not forbid consecutive points with identical coordinates, shapefile readers must handle such cases. On the other hand, the degenerate zero length or zero area parts that might result are not allowed.

The fields for a polygon are described in detail below:

- **Box.** The bounding box for the polygon, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of rings in the polygon
- **NumPoints.** The total number of points for all rings
- **Parts.** An array of length NumParts

  Stores, for each ring, the index of its first point in the points array. Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each ring in the polygon are stored end to end. The points for ring 2 follow the points for ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.

The following are important notes about polygon shapes:

- The rings are closed (the first and last vertex of a ring must be the same).
- The order of rings in the points array is not significant.
- Polygons stored in a shapefile must be clean.
- A clean polygon is one that has no self-intersections. This means that a segment belonging to one ring may not intersect a segment belonging to another ring. The rings of a polygon can touch each other at vertices but not along segments. Colinear segments are considered intersecting.
- A clean polygon is one that has the inside of the polygon on the *correct* side of the line that defines it. The neighborhood to the right of an observer walking along the ring in vertex order is the inside of the polygon. Vertices for a single, ringed polygon are, therefore, always in clockwise order.

  Rings defining holes in these polygons have a counterclockwise orientation. *Dirty* polygons occur when the rings that define holes in the polygon also go clockwise, which causes overlapping interiors.

A Sample Polygon Instance

Figure E-1 shows a polygon with one hole and a total of eight vertices.

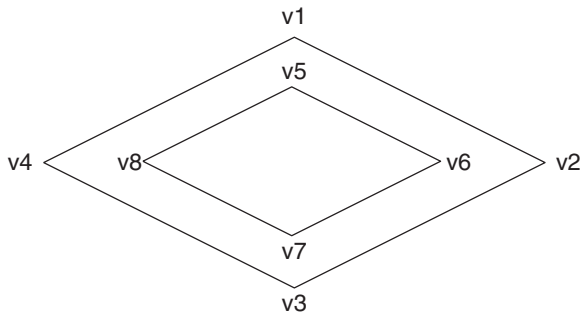
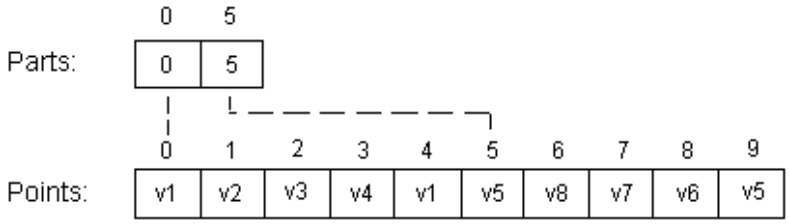


Figure E-1. A Sample Polygon

For this example, NumParts equals 2 and NumPoints equals 10. Note that the order of the points for the doughnut (hole) polygon is reversed below.



The following table shows Polygon byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	5	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts)

Measured Shape Types in XY Space

This section describes shapes with X and Y coordinates that also have measure values.

PointM

A PointM consists of a pair of double-precision coordinates in the order X, Y, plus a measure M. The following table shows PointM byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	21	Integer	1	Little endian
Byte 4	X	X	Double	1	Little endian
Byte 12	Y	Y	Double	1	Little endian
Byte 20	M	M	Double	1	Little endian

MultiPointM

The fields for a MultiPointM are:

- **Box.** The bounding box for the MultiPointM, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of Points
- **NumPoints.** An array of Points of length NumPoints
- **M Range.** The minimum and maximum measures for the MultiPointM stored in the order Mmin, Mmax
- **M Array.** An array of Measures of length NumPoints

The following table shows MultiPointM byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	28	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumPoints	NumPoints	Integer	1	Little endian
Byte 40	Points	Points	Point	NumPoints	Little endian
Byte X	Mmin	Mmin	Double	1	Little endian
Byte X+8	Mmax	Mmax	Double	1	Little endian
Byte X+16	M Array	M Array	Double	NumPoints	Little endian

Tip: X = 40 + (16 * NumPoints).

PolyLineM

A shapefile PolyLineM consists of one or more parts. A part is a connected sequence of two or more points. Parts may or may not be connected to one another. Parts may or may not intersect one another.

The fields for a PolyLineM are:

- **Box.** The bounding box for the PolyLineM stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of parts in the PolyLineM
- **NumPoints.** The total number of points for all parts
- **Parts.** An array of length NumParts

  Stores, for each part, the index of its first point in the points array. Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each part in the PolyLineM are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.
- **M Range.** The minimum and maximum measures for the PolyLineM stored in the order Mmin, Mmax
- **M Array.** An array of length NumPoints

  The measures for each part in the PolyLineM are stored end to end. The measures for part 2 follow the measures for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the measure array between parts.

The following table shows PolyLineM byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	23	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian
Byte Y	Mmin	Mmin	Double	1	Little endian
Byte Y+8	Mmax	Mmax	Double	1	Little endian
Byte Y+16	M Array	M Array	Double	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts), Y = X + (16 * NumPoints)

PolygonM

A PolygonM consists of a number of rings. A ring is a closed, non-self-intersecting loop. Note that intersections are calculated in XY space, *not* in XYM space. A PolygonM may contain multiple outer rings. The rings of a PolygonM are referred to as its parts.

The fields for a PolygonM are:

- **Box.** The bounding box for the PolygonM, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of rings in the PolygonM
- **NumPoints.** The total number of points for all rings
- **Parts.** An array of length NumParts

  Stores, for each ring, the index of its first point in the points array. Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each ring in the PolygonM are stored end to end. The points for Ring 2 follow the points for Ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.
- **M Range.** The minimum and maximum measures for the PolygonM stored in the order Mmin, Mmax
- **M Array.** An array of length NumPoints

  The measures for each ring in the PolygonM are stored end to end. The measures for Ring 2 follow the measures for Ring 1, and so on. The parts array holds the array index of the starting measure for each ring. There is no delimiter in the measure array between rings.

The following are important notes about PolygonM shapes:

- The rings are closed (the first and last vertex of a ring must be the same).
- The order of rings in the points array is not significant.

The following table shows PolygonM byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	15	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian
Byte Y	Mmin	Mmin	Double	1	Little endian
Byte Y+8	Mmax	Mmax	Double	1	Little endian
Byte Y+16	M Array	M Array	Double	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts), Y = X + (16 * NumPoints)

Shape Types in XYZ Space

This section describes shapes with X, Y, and Z coordinates.

PointZ

A PointZ consists of a triplet of double-precision coordinates in the order X, Y, Z.

The following table shows PointZ byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	9	Integer	1	Little endian
Byte 4	X	X	Double	1	Little endian
Byte 12	Y	Y	Double	1	Little endian
Byte 20	Z	Z	Double	1	Little endian

MultiPointZ

A MultiPointZ represents a set of PointZs, as follows:

- The bounding box is stored in the order Xmin, Ymin, Xmax, Ymax.
- The bounding Z range is stored in the order Zmin, Zmax.

The following table shows MultiPointZ byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	20	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumPoints	NumPoints	Integer	1	Little endian
Byte 40	Points	Points	Point	NumPoints	Little endian
Byte X	Zmin	Zmin	Double	1	Little endian
Byte X+8	Zmax	Zmax	Double	1	Little endian
Byte X+16	Z Array	Z Array	Double	NumPoints	Little endian

Tip: X = 40 + (16 * NumPoints)

PolyLineZ

A PolyLineZ consists of one or more parts. A part is a connected sequence of two or more points. Parts may or may not be connected to one another. Parts may or may not intersect one another.

The fields for a PolyLineZ are described in detail below:

- **Box.** The bounding box for the PolyLineZ, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of parts in the PolyLineZ
- **NumPoints.** The total number of points for all parts
- **Parts.** An array of length NumParts

  Stores, for each part, the index of its first point in the points array.

  Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each part in the PolyLineZ are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.
- **Z Range.** The minimum and maximum Z values for the PolyLineZ stored in the order Zmin, Zmax
- **Z Array.** An array of length NumPoints

  The Z values for each part in the PolyLineZ are stored end to end. The Z values for part 2 follow the Z values for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the Z Array between parts.

The following table shows PolyLineZ byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	10	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian
Byte Y	Zmin	Zmin	Double	1	Little endian
Byte Y+8	Zmax	Zmax	Double	1	Little endian
Byte Y+16	Z Array	Z Array	Double	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts), Y = X + (16 * NumPoints)

PolygonZ

A PolygonZ consists of a number of rings. A ring is a closed, non-self-intersecting loop. A PolygonZ may contain multiple outer rings. The rings of a PolygonZ are referred to as its parts.

The fields for a PolygonZ are:

- **Box.** The bounding box for the PolygonZ stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of rings in the PolygonZ

- **NumPoints.** The total number of points for all rings
- **Parts.** An array of length NumParts

  Stores, for each ring, the index of its first point in the points array. Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each ring in the PolygonZ are stored end to end. The points for ring 2 follow the points for ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.
- **Z Range.** The minimum and maximum Z values for the PolygonZ stored in the order Zmin, Zmax
- **Z Array.** An array of length NumPoints

  The Z values for each ring in the PolygonZ are stored end to end. The Z values for ring 2 follow the Z values for ring 1, and so on. The parts array holds the array index of the starting Z value for each ring. There is no delimiter in the Z value array between rings.

The following are important notes about PolygonZ shapes.

- The rings are closed (the first and last vertex of a ring must be the same).
- The order of rings in the points array is not significant.

The following table shows PolygonZ byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	19	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian
Byte Y	Zmin	Zmin	Double	1	Little endian
Byte Y+8	Zmax	Zmax	Double	1	Little endian
Byte Y+16	Z Array	Z Array	Double	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts), Y = X + (16 * NumPoints)

Measured Shape Types in XYZ Space

This section describes shapes with X, Y, and Z coordinates that also have measure values.

PointZM

A PointZM consists of a quadruplet of double-precision coordinates in the order X, Y, Z, M.

The following table shows PointZM byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	11	Integer	1	Little endian
Byte 4	X	X	Double	1	Little endian
Byte 12	Y	Y	Double	1	Little endian
Byte 20	Z	Z	Double	1	Little endian
Byte 28	M	M	Double	1	Little endian

MultiPointZM

A MultiPointZM represents a set of PointZMs, as follows:

- The bounding box is stored in the order Xmin, Ymin, Xmax, Ymax.
- The bounding Z range is stored in the order Zmin, Zmax.
- The bounding M range is stored in the order Mmin, Mmax.

The following table shows MultiPointZM byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	18	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumPoints	NumPoints	Integer	1	Little endian
Byte 40	Points	Points	Point	NumPoints	Little endian
Byte X	Zmin	Zmin	Double	1	Little endian
Byte X+8	Zmax	Zmax	Double	1	Little endian
Byte X+16	Z Array	Z Array	Double	NumPoints	Little endian
Byte Y	Mmin	Mmin	Double	1	Little endian
Byte Y+8	Mmax	Mmax	Double	1	Little endian
Byte Y+16	M Array	M Array	Double	NumPoints	Little endian

Tip: X = 40 + (16 * NumPoints), Y = X + 16 + (8 * NumPoints)

PolyLineZM

A PolyLineZM consists of one or more parts. A part is a connected sequence of two or more points. Parts may or may not be connected to one another. Parts may or may not intersect one another.

The fields for a PolyLineZM are described in detail below:

- **Box.** The bounding box for the PolyLineZM, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of parts in the PolyLineZM
- **NumPoints.** The total number of points for all parts
- **Parts.** An array of length NumParts

  Stores, for each part, the index of its first point in the points array.

  Array indexes are numbered with respect to 0
- **Points.** An array of length NumPoints

  The points for each part in the PolyLineZM are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.

- **Z Range.** The minimum and maximum Z values for the PolyLineZM stored in the order Zmin, Zmax
- **Z Array.** An array of length NumPoints

  The Z values for each part in the PolyLineZM are stored end to end. The Z values for part 2 follow the Z values for part 1, and so on. There is no delimiter in the Z Array between parts.
- **M Range.** The minimum and maximum measures for the PolyLineZM stored in the order Mmin, Mmax
- **M Array.** An array of length NumPoints

  The measures for each part in the PolyLineZM are stored end to end. The measures for part 2 follow the measures for part 1, and so on. There is no delimiter in the measure array between parts.

The following table shows PolyLineZM byte stream contents

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	13	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian
Byte Y	Zmin	Zmin	Double	1	Little endian
Byte Y+8	Zmax	Zmax	Double	1	Little endian
Byte Y+16	Z Array	Z Array	Double	NumPoints	Little endian
Byte Z	Mmin	Mmin	Double	1	Little endian
Byte Z+8	Mmax	Mmax	Double	1	Little endian
Byte Z+16	M Array	M Array	Double	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts), Y = X + (16 * NumPoints), Z=Y + 16 + (8 * NumPoints)

PolygonZM

A PolyLineZM consists of a number of rings. A ring is a closed, non-self-intersecting loop. A PolyLineZM may contain multiple outer rings. The rings of a PolyLineZM are referred to as its parts.

The fields for a PolyLineZM are:

- **Box.** The bounding box for the PolyLineZM, stored in the order Xmin, Ymin, Xmax, Ymax
- **NumParts.** The number of rings in the PolyLineZM
- **NumPoints.** The total number of points for all rings
- **Parts.** An array of length NumParts

  Stores, for each ring, the index of its first point in the points array.

  Array indexes are numbered with respect to 0.
- **Points.** An array of length NumPoints

  The points for each ring in the PolyLineZM are stored end to end. The points for ring 2 follow the points for ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.

- **Z Range.** The minimum and maximum Z values for the PolyLineZM are stored in the order Zmin, Zmax
- **Z Array.** An array of length NumPoints

  The Z values for each ring in the PolyLineZM are stored end to end. The Z values for ring 2 follow the Z values for ring 1, and so on. There is no delimiter in the Z value array between rings.
- **M Range.** The minimum and maximum measures for the PolyLineZM stored in the order Mmin, Mmax
- **M Array.** An array of length NumPoints

  The measures for each ring in the PolyLineZM are stored end to end. The measures for ring 2 follow the measures for ring 1, and so on. There is no delimiter in the measure array between rings.

The following are important notes about PolyLineZM shapes:

- The rings are closed (the first and last vertex of a ring must be the same).
- The order of rings in the points array is not significant.

The following table shows PolyLineZM byte stream contents.

Position	Field	Value	Type	Number	Byte Order
Byte 0	Shape Type	15	Integer	1	Little endian
Byte 4	Box	Box	Double	4	Little endian
Byte 36	NumParts	NumParts	Integer	1	Little endian
Byte 40	NumPoints	NumPoints	Integer	1	Little endian
Byte 44	Parts	Parts	Integer	NumParts	Little endian
Byte X	Points	Points	Point	NumPoints	Little endian
Byte Y	Zmin	Zmin	Double	1	Little endian
Byte Y+8	Zmax	Zmax	Double	1	Little endian
Byte Y+16	Z Array	Z Array	Double	NumPoints	Little endian
Byte Z	Mmin	Mmin	Double	1	Little endian
Byte Z+8	Mmax	Mmax	Double	1	Little endian
Byte Z+16	M Array	M Array	Double	NumPoints	Little endian

Tip: X = 44 + (4 * NumParts), Y = X + (16 * NumPoints), Z=Y + 16 + (8 * NumPoints)

Appendix F. Values for the geometry_type Column

The system table **geometry_columns** maps spatial data types to values in its **geometry_type** column. The following table shows valid entries for the **geometry_type** column and the shapes they represent.

geometry_type Column Value	Spatial Data Type
0	ST_Geometry
1	ST_Point
2	ST_Curve
3	ST_LineString
4	ST_Surface
5	ST_Polygon
6	ST_GeomCollection
7	ST_MultiPoint
8	ST_MultiCurve
9	ST_MultiLineString
10	ST_MultiSurface
11	ST_MultiPolygon

For information about spatial data types, see Chapter 2, "Using Spatial Data Types," on page 2-1.

Appendix G. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix Dynamic Server

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix Dynamic Server. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Dynamic Server Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft® Windows navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format. For more information about the dotted decimal format, go to "Dotted Decimal Syntax Diagrams."

You can view the publications for IBM Informix Dynamic Server in Adobe® Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Dotted Decimal Syntax Diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines

2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

* Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data-area, you know that you can include more than one data area or you can include none. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

\+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Error Messages

This appendix lists the IBM Informix Spatial DataBlade Module error messages.

Spatial DataBlade functions that generate any of these errors will abort. Likewise, if a Spatial DataBlade function is part of a transaction, the transaction also aborts. Error message parameters marked with percent signs—for example, %FUNCTION%—are replaced with appropriate values in the actual message text.

Error Messages and Their Explanations

USE01 **Unable to establish a connection in %FUNCTION%.**

Explanation: The IBM Informix server unexpectedly returned a null value instead of a connection handle when the function attempted to connect to the server. The function was unable to determine the exact cause of the error.

User response: If this error continues to occur, contact IBM Informix Technical Support for assistance.

7USE02 **Function %FUNCTION% is unable to allocate memory.**

Explanation: The function could not allocate the memory that it requires.

User response: Ensure that your hardware meets the minimum memory requirements specified in the IBM Informix installation guide. Also make sure that you have not overallocated memory to the IBM Informix server or other applications. Consider increasing the amount of your hardware physical memory.

USE03 **Invalid geometry in %FUNCTION%.**

Explanation: The parameters entered into the function have produced an invalid geometry.

User response: Check the parameters and review Chapter 2, "Using Spatial Data Types," on page 2-1, for a description of valid geometry.

USE04 **Function %FUNCTION% not applicable to type %TYPE%.**

Explanation: An invalid geometry type was passed to the function. Valid geometry types are geometry, point, linestring, polygon, multipoint, multilinestring, and multipolygon.

User response: Resubmit the function with one of the valid geometry types.

USE05 **This function is not yet implemented.**

Explanation: The function has not been implemented for the current release. However, it may be available in a future release.

USE06 **Unknown ESRI shape library error (%ERRCODE%) in %FUNCTION%.**

Explanation: Contact IBM Informix Technical Support for assistance. Please include this error number and any other related error messages and information from the **$INFORMIXDIR/online.log** file. If you are using the Spatial DataBlade module in conjunction with ArcSDE from ESRI, please also include any related information from the **$SDEHOME/etc/sde.errlog** file.

USE07 **Internal SAPI error. %SAPIFUNC% returned %RETVAL%. Failure in %FUNCNAME%.**

Explanation: An error has occurred in the SAPI subsystem of the IBM Informix server.

User response: Contact IBM Informix Technical Support.

USE08 **Nearest-neighbor queries require an index scan.**

Explanation: An attempt was made to use a nearest-neighbor function as a filter during a sequential scan of a table. This is not supported.

USE09 **Unknown or unsupported shape file type (%TYPE%) found in %FUNCTION%.**

Explanation: An unrecognized shapefile type was encountered.

User response: For more information on shapefile types, see "Shape Type Values" on page E-1.

USE10 **Unknown or unsupported OpenGIS WKB type (%TYPE%) found in %FUNCTION%.**

Explanation: An unrecognized OpenGIS well-known binary type was encountered. This version of the Spatial DataBlade only supports point, linestring, polygon, multipoint, multilinestring, and multipolygon.

USE11 Invalid SRID %SRID% or NULL in %FUNCTION%.

Explanation: A spatial reference identifier must be an integer value greater than 0. Negative values, real numbers, and characters are invalid.

USE12 Unknown or unsupported geometry type (%TYPE%) found in %FUNCTION%.

Explanation: An unrecognized geometry type was encountered. If you are inserting spatial data in a binary format it may be corrupt or malformed.

USE13 Spatial DataBlade not installed correctly: the spatial_references table does not exist.

Explanation: The Spatial DataBlade was not able to access the **spatial_references** table because it could not be found.

User response: Review the installation instructions for the Spatial DataBlade module and, if necessary, recreate the **spatial_references** table.

USE14 Unknown spatial reference identifier %SRID%.

Explanation: The **sde.spatial_references** table contains a list of all valid spatial reference identifiers. The function attempted to create a geometry with an SRID that does not exist in this table.

User response: Either use an SRID that is already in the **sde.spatial_references** table or add the unknown SRID to the table.

USE15 Invalid coordinate reference system object in function %FUNCTION%.

Explanation: A programmatic error has occurred in the Spatial DataBlade.

User response: Contact IBM Informix Technical Support.

USE16 Unable to get the geometry data pointer from the server in %FUNCTION%.

Explanation: A programmatic error has occurred in the Spatial DataBlade.

User response: Contact IBM Informix Technical Support.

USE17 Geometry verification failed.

Explanation: An error has occurred while the Spatial DataBlade was verifying the topological correctness of a geometry.

User response: Contact IBM Informix Technical Support.

USE18 Buffer operation failed.

Explanation: The source geometry and buffer distance submitted to the buffer function would result in a buffer with coordinates that fall outside the coordinate system specified in the source geometry's spatial reference system.

USE19 Coordinates out of bounds in %FUNCTION%.

Explanation: The function has created a geometry with coordinates that fall outside the coordinate system.

User response: Review "The spatial_references Table" on page 1-4 for information on selecting a coordinate system. Typically, this error occurs when the buffer function generates a geometry with coordinates that are beyond the source geometry's coordinate system, which the new geometry inherits.

You may need to adjust the false origin and system units for your data. Please refer to the description of the **ST_Transform()** function in Chapter 8, "SQL Functions," on page 8-1, for additional information.

USE20 Invalid parameter in function %FUNCTION%.

Explanation: One of the parameters passed to the function is invalid.

User response: Review the syntax of the function listed in Chapter 8, "SQL Functions," on page 8-1. Correct the invalid parameter and resubmit the function.

USE21 Geometry integrity error in function %FUNCTION%.

Explanation: An inconsistency has been detected in a geometry's internal data structure.

User response: If this error continues to occur, contact IBM Informix Technical Support for assistance.

USE22 Too many points in feature.

Explanation: Returned if the buffer function creates a feature that exceeds the maximum number of points specified as a parameter to the function.

User response: Increase the size of the maximum

number of points parameter and resubmit the function.

USE23 Spatial reference conflict, %SRID1% vs %SRID2%.

Explanation: The geometries passed to the function did not share the same spatial reference system.

User response: Convert one of the geometries to have the same spatial reference system as the other and resubmit the function.

USE24 Incompatible geometries in function %FUNCTION%.

Explanation: The function expected two geometries of a certain type and did not receive them.

User response: Review the syntax of the function described in Chapter 8, "SQL Functions," on page 8-1, correct the geometry, and resubmit the function.

USE25 Subscript %SUBSCRIPT% out of range in function %FUNCTION%.

Explanation: The function has detected that the subscript entered is outside the allowable range of values. For instance, the **ST_PointN()** function returns the *n*th point identified by the index parameter. If a negative value, 0, or a number greater than the number of points in the source linestring were entered, this error message would be returned.

User response: Correct the subscript value and resubmit the function.

USE26 Subtype mismatch: received subtype=%TYPE1%, expected subtype=%TYPE2%.

Explanation: This error can occur when you try to insert a geometry of one subtype into a column of a different subtype, for example an ST_Point into an ST_LineString column.

User response: To insert more than one subtype into a column, make that a column of type ST_Geometry.

USE27 Unknown or unsupported geometry data structure version (%VERSION%) found in %FUNCTION%.

Explanation: Future versions of the Spatial DataBlade may not be able to interpret geometries stored using this version of the DataBlade module. Similarly, this version of the Spatial DataBlade may not be able to interpret geometries stored using a future version.

If an upgrade mechanism is provided with a new version of this DataBlade module, use it on your data as described in the release notes. If an upgrade mechanism is not provided, you must unload your data with the old DataBlade module version and reload it with the new version.

USE28 Invalid text in %FUNCTION%.

Explanation: The text string entered with the well-known text representation function is invalid.

User response: Correct the string and resubmit the function. Refer to Appendix C, "OGC Well-Known Text Representation of Geometry," on page C-1, for a valid text string description.

USE29 Unexpected system error in %FUNCTION%.

Explanation: An internal error occurred while creating a geometry. The system was not able to determine why this error occurred.

User response: Contact IBM Informix Technical Support for assistance. Please include this error number and any other related error messages and information from the **$INFORMIXDIR/online.log** file. If you are using the Spatial DataBlade module in conjunction with ArcSDE from ESRI, please also include any related information from the **$SDEHOME/etc/sde.errlog** file.

USE30 Overlapping polygon rings in %FUNCTION%.

Explanation: The internal rings of a polygon may not overlap one another or the bounding external ring. Polygon rings may only intersect at a single point.

USE31 Too few points for geometry type in %FUNCTION%.

Explanation: The number of coordinates entered for the geometry was too few. Points and multipoints require a minimum of one point; linestrings and multilinestrings require a minimum of two points; and polygons and multipolygons require a minimum of four points.

USE32 Polygon does not close in %FUNCTION%.

Explanation: The first and last coordinates of a polygon ring must be the same. An exterior or interior ring did not close (did not have the same first and last coordinates).

USE33 Interior ring not enclosed by exterior ring in %FUNCTION%.

Explanation: The interior rings of a polygon must be inside the exterior rings. The interior ring was detected to be outside its exterior ring.

USE34 Polygon has no area in %FUNCTION%.

Explanation: The rings of a polygon must enclose an area. The first and last point of each polygon ring must be the same. A ring may not cross itself.

USE35 Polygon ring contains a spike in %FUNCTION%.

Explanation: Polygon rings contain spikes whenever coordinates other than the endpoints are the same. The boundary of a polygon must be a continuous ring or series of rings.

USE36 Multipolygon exterior rings overlap in %FUNCTION%.

Explanation: The exterior rings of a multipolygon must enclose independent areas. The exterior rings of each polygon of a multipolygon may not overlap. They may, however, intersect at a single point. Polygons whose intersection results in a linestring will automatically be merged after the intersecting linestring has been dissolved.

USE37 The geometry boundary is self-intersecting in %FUNCTION%.

USE38 The geometry has too many parts in %FUNCTION%.

Explanation: The string that defines the geometry has too many parts for its type. Points, linestrings, and polygons are single-part geometries. The string has defined more than one part for one of these geometries. If a multipart geometry is desired, use multipoint, multilinestring, or multipolygon.

USE39 Mismatched text string parentheses in %FUNCTION%.

Explanation: The parentheses of the text string defining the geometry do not match.

User response: For a description of the well-known text representation, review Appendix C, "OGC Well-Known Text Representation of Geometry," on page C-1.

USE40 Unknown or unsupported ESRI entity type (%TYPE%) found in %FUNCTION%.

Explanation: The internal type representation is invalid.

User response: Contact ESRI Technical Support.

USE41 The projection string for your SRID is invalid in %FUNCTION%.

Explanation: The projection string stored in the **spatial_references** table was determined to be invalid.

User response: Compare the projection string with the valid projection strings listed in Appendix B, "OGC Well-Known Text Representation of Spatial Reference Systems," on page B-1.

USE42 Nearest-neighbor queries are not supported by the current version of the server.

Explanation: Nearest-neighbor queries are supported by IBM Informix Dynamic Server Version 9.3 and later.

USE43 %PARAM1% value must be less than %PARAM2% value.

Explanation: When executing the **SE_CreateSrid()** function, xmin must be less than xmax and ymin must be less than ymax.

USE44 Unknown OGIS WKB byte-order byte encountered in %FUNCTION%.

Explanation: An unrecognized OpenGIS well-known binary byte-order byte was encountered. This is the first byte of the geometry data input byte stream. Valid values are 0x00 (big endian) and 0x01 (little endian).

USE45 OGIS WKB geometry collection type is not supported.

Explanation: The OpenGIS collection type (7) is not supported by this version of the Spatial DataBlade module.

USE46 Incompatible coordinate reference systems in function %FUNCTION%.

Explanation: This error can occur when you attempt to transform geometries using the **ST_Transform()** function. The only allowable transformations in this version of the Spatial DataBlade module are:

- Between two UNKNOWN coordinate systems (that is, the **srtext** column in the **spatial_references** table for both SRIDs is "UNKNOWN")
- Between a projected coordinate system and an unprojected coordinate system, in which the underlying geographic coordinate systems are the same
- Between two projected coordinate systems, in which the underlying geographic coordinate systems are the same
- Between two coordinate systems with the same geographic coordinate system (that is, a difference in false origin or system unit only)

USE47 **Cannot create SE_Metadata lohandle file %NAME%. Check directory permissions.**

Explanation: The metadata lohandle file is created at the time the DataBlade module is registered in your database. This file is located in the directory **$INFORMIXDIR/extend/spatial.***versno***/metadata**. The user who registers the DataBlade must have write permission on this directory.

If this error occurs after you have successfully registered the Spatial DataBlade module, you should correct the permissions on the **metadata** directory and then recreate the metadata lohandle file by running the following SQL statement:

```
execute function SE_MetadataInit();
```

USE48 **SE_Metadata lohandle file %FILE% not found, unreadable, or corrupt. Execute function SE_MetadataInit to reinitialize.**

Explanation: The purpose of the **SE_Metadata** lohandle file is to allow access to metadata by all parallelized functions of the Spatial DataBlade module. It can be restored by running the following SQL statement:

```
execute function SE_MetadataInit();
```

This will reread the **spatial_references** table, recreate a smart large object containing metadata, and re-create a file containing the large object handle for this smart large object.

USE49 **SE_MetadataTable is a read only table.**

Explanation: The **SE_MetadataTable** table is created and populated when the Spatial DataBlade module is registered.

User response: Do not attempt to modify this table in any way.

USE50 **Vertex not found in %FUNCTION%.**

Explanation: The specified vertex cannot be found in the original geometry.

User response: Verify that the X, Y, M, and Z values (if any) of the vertex to be updated or deleted exactly match.

USE51 **SE_Metadata smart blob is corrupt or unreadable.**

Explanation: Execute the **SE_MetadataInit()** function to repair the smart large object. To enable parallel data queries, a copy of the **spatial_references** table contents is stored in a smart large object. This smart large object is created when the Spatial DataBlade is registered and is synchronized with the **spatial_references** table by means of triggers. If the smart large object is corrupted, it can be re-created by running the following SQL statement:

```
EXECUTE FUNCTION SE_MetadataInit();
```

USE52 **SE_Metadata memory cache is locked.**

Explanation: Execute the **SE_MetadataInit()** function to reinitialize the memory cache. For computational efficiency, a copy of the **spatial_references** table contents is cached in memory. This cache is shared by all sessions and access to it is controlled by a spinlock. If a session failed to release this lock, another session may not be able to obtain access.

User response: To forcibly reset the lock, run the following SQL statement:

```
EXECUTE FUNCTION SE_MetadataInit();
```

USE53 **Spatial datablade assert failure. File = %FILE%, line = %LINE%.**

Explanation: A programmatic error has occurred in the Spatial DataBlade module. Contact IBM Informix Technical Support.

USE54 **You must create a default sbspace before you can register the Spatial DataBlade.**

Explanation: When the Spatial DataBlade is registered, a small (8 KB) smart large object is created and stored in the default sbspace. Registration fails if there is no default sbspace. This smart large object can be moved to a non-default sbspace after registration is complete; instructions for moving it are provided in the release notes.

-674 **Routine (%FUNCTION%) cannot be resolved.**

Explanation: This error message is generally returned by the Informix server whenever you try to apply the function to a nonsupported type.

User response: Check the geometry type entered and resubmit the function.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not

been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided ″AS IS″, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript® are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel®, Itanium®, and Pentium® are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT® are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

M

N

O

P

R

S

T

U

V

W

X

Z

Printed in USA

G229-6405-03

Spine information:

IBM Informix **Version 8.21**

IBM Informix Spatial DataBlade Module User's Guide