DB2 Spatial Extender

Administration Guide and Reference

Version 2 Release 1 Modification 1

SC26-9316-00

DB2 Spatial Extender

Administration Guide and Reference

Version 2 Release 1 Modification 1

SC26-9316-00

Note

Before using this information and the product it supports, please read the general information under “Notices” on page 209.

First Edition (July 1998)

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, BWE/H3
P. O. Box 49023
San Jose, CA 95161-9023
U.S.A.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

About This Book

This book contains information about installing, configuring, administering, programming, and troubleshooting the DB2 Spatial Extender. Because the DB2 Spatial Extender is bundled with other applications, there are references to other documents. These documents are provided in HTML, PDF, and PostScript formats on the respective product CDs.

Who Should Use This Book

This book is for administrators setting up the spatial environment and for application programmers developing applications with spatial data.

Terms for Products

Some product names in the documentation refer to more than one product, some refer to specific product levels, and some are shortened versions of full names. These product names are:

DataJoiner
Refers to DB2 DataJoiner Version 2. References specific to or including DataJoiner Version 1 will include the version.

DB2
By itself, refers to any one or all of the DB2 for common server Version 2 database server products on all platforms, which includes DataJoiner.

If a DB2 reference is qualified with a specific operating system or version, the reference applies only to that particular version.

DB2 Family
Refers to all DataJoiner-supported versions of DATABASE 2 (DB2) database server products on all platforms (DB2 for OS/390, DB2 for VM, DB2 for common servers, DataJoiner, and so on). Supported versions are listed in the *DataJoiner Planning, Installation, and Configuration Guide* for your platform.

DB2 for CS
Refers to any DB2 for common servers Version 2 database server product. This term is often used when describing DataJoiner and DB2 for common servers functional differences.

RDB
Refers to Oracle RDB Version 6 or above.

SQL Anywhere
Refers to Sybase SQL Anywhere Version 5.

What's New in DataJoiner Version 2?

DataJoiner Version 2 offers new features and enhancements. They include:

DB2 Version 2 functionality

DataJoiner is built on the DB2 Version 2 code base, which means that DataJoiner provides all the major functional enhancements provided by DB2, including:

- Extended SQL capabilities
- An enhanced SQL optimizer
- Improved database performance
- Systems management support
- Robust integrity and data protection
- Object relational capabilities
- National language support (NLS)
- Support for the Java Development Kit (JDK) 1.1 for the Java Database Connectivity (JDBC) API

For detailed information about many of these features, see the *DB2 Administration Guide.*

DataJoiner for Windows NT

DataJoiner has extended its reach to provide industrial strength heterogeneous database management on Windows NT systems. DataJoiner for Windows NT supports the same SQL and features as DataJoiner for UNIX-based platforms.

Support for Oracle 8, RDB, and SQL Anywhere

With Version 2, DataJoiner continues to increase the number of natively-supported data sources. The most recent additions are:

- Oracle 8 (on any system that DataJoiner accesses from AIX or Windows NT)
- Oracle RDB Version 6 and above (on any system that DataJoiner accesses from Windows NT)
- Sybase SQL Anywhere Version 5.0 (on any system that DataJoiner accesses from Windows NT)

Spatial Extender

DataJoiner now supports geographic information system (GIS) data (also known as spatial or geographic data). New data types, spatially-enabled columns, and spatial join capability allow you to take advantage of geographic data in your applications. Included are powerful two-dimensional functions that allow you to create specific relationships among the geographic objects you define. Included with the spatial extender are the following components:

- A set of spatial data types
- A set of spatial operations and predicates
- A set of spatial index data types

- An administration tool suite for the spatial extender
- Sample programs

You can also take advantage of existing geographic data stores using the load and transform capability of the Spatial Extender.

Expanded DataJoiner SQL support
This version of DataJoiner contains many new and modified SQL statements. New DDL statements provide greater flexibility and safety in defining your DataJoiner environment—users can create, alter, and drop mappings for data sources, users, user-defined and built-in functions, and data types. Additionally, new SQL DML statements provide enhanced functions for local and distributed queries; an example is the CASE expression, which is useful for selecting an expression based on the evaluation of one or more conditions.

DataJoiner SQL for creating, altering, and deleting data source tables
Version 2 includes a new DataJoiner SQL statement for creating tables in different types of data sources. If the native SQL for creating tables in these data sources includes a unique option—for example, the option in DB2 for OS/390 for specifying what database you want a table to reside in—you can code this option in the new DataJoiner statement. If you create a data source table with this new statement, you can also alter and delete it with DataJoiner SQL.

Heterogeneous data replication
DataJoiner now provides replication administration as an integrated component. You can define, automate, and manage replication data from a single control point across your enterprise. The replication administration tool provides administrative support for the replication environment, with objects and actions that define and manage source and target table definitions. DataJoiner's Apply component performs the actual replication, tailoring and enhancing data as you specify, and serving as the interface point to and from your various data sources. DataJoiner also supplies an executable, IBM DB2 DataPropagator for Microsoft Jet, that allows you to replicate server data for browsing and updating in LAN, occasionally connected, and mobile environments.

Distributed heterogeneous update support
DataJoiner now allows you to update multiple heterogeneous data sources within a distributed unit of work while maintaining transaction atomicity. This task is accomplished through adherence to the two-phase commit model. Supported data sources include most versions of the DB2 Family and, with the appropriate XA libraries, various other data sources as well.

New graphical installation, configuration, and administration tools
A variety of new tools is available to help you accomplish administrative chores. Wizards walk you through data source configuration. And the Administrator's Toolkit provides a collection of tools designed to assist you with the day-to-day operation of DataJoiner. It includes the following components:

The Database Director
Allows you to perform configuration, backup and recovery, directory management, and media management tasks.

Visual Explain
A tool for graphically viewing and navigating complex SQL access plans.

The DB2 Performance Monitor
Monitors the performance of your DB2 system for tuning purposes.

Stored procedures
DataJoiner now supports stored procedures at remote data sources as well as the local DataJoiner database. Use stored procedures to speed application performance. For example, applications that process huge amounts of data at a server but return smaller result sets should run faster as stored procedures. Another benefit is that stored procedures usually reduce network traffic between clients and databases.

DataJoiner stored procedures can augment standard data security. For example, in a 3-tier environment, data can be retrieved from a remote server and then processed at the DataJoiner server; only a subset of data needs to be available to the client.

System catalog information available in views
DataJoiner provides views from which you can access system catalog information about each DataJoiner database. Some of these views contain data—for example, data about tables, indexes, and servers—that was accessible only from tables in previous versions of DataJoiner. Other views contain data—for example, data about stored procedures, server options, and server functions—that is now available in Version 2.

Performance enhancements
In addition to general engine performance improvements, this latest version offers new query rewrite capabilities, improved pushdown performance, and remote query caching.

Chapter 1. About the Spatial Extender

The Spatial Extender imbeds a *Geographic Information System* (GIS) into DataJoiner. The Spatial Extender implements the SQL3 specification of data types, which is a standard set by the Open GIS consortium (OGIS). These data types are capable of storing spatial data, such as the actual location of a landmark, a street, or a parcel of land.

GISs of the past were spatially centric. They focused on gathering spatial data and attaching a non-spatial *attribute* data to it. The Spatial Extender integrates spatial and non-spatial data, providing a single point of access through the SQL interface of DataJoiner.

In addition to new data types, the Spatial Extender also provides new capabilities, such as spatial joins. Application programmers typically join tables by comparing two or more columns to determine whether their values are equal, not equal, greater than, and so on. The Spatial Extender includes functions capable of comparing the values of spatial columns to determine if they intersect, if one is inside the other, if one overlaps the other, and so on. These two-dimensional functions are very powerful because they make it possible to join tables based on the spatial relationship of geographic elements.

These functions can be used to answer questions such as: ″Is this school within a five-mile radius of a hazardous waste site?″ To answer this question, the Spatial Extender’s *overlaps* function would be used: ″Does this polygon, (the building footprint of a school), *overlap* this circular area, (the 5-mile radius of a hazardous waste site)?″ An application programmer would join a table that stores schools, playgrounds, hospitals, and other sensitive sites with another table that contains the location of hazardous sites. The results of that join would return a list of sensitive areas at risk.

The Complete Spatial System

The DB2 Spatial Extender is implemented as a separately-installable component of DB2 DataJoiner Version 2.1.1. It is one of three products provided for you to create a spatial system. The other two products are:

- ESRI Spatial Database Engine™ Version 3.0.2 (Windows NT or AIX)
- ESRI ArcView™ GIS Version 3.0a (Windows NT only)

This book contains instructions for installing and configuring the Spatial Database Engine (SDE) and the Spatial Extender. Administration and Programming information for the Spatial Extender is also provided in this book. For information on the other two products, refer to the ESRI documentation provided in PDF format on each product’s CD-ROM.

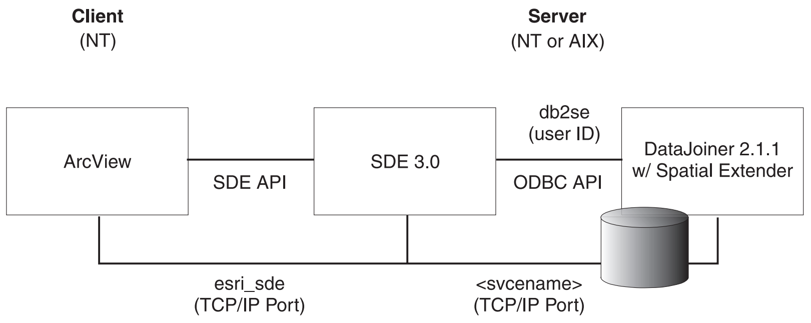


Figure 1. The components of a complete spatial system

The DB2 Spatial Extender Component

Once the Spatial Extender has been installed, configured, and spatially enabled, you can create tables that include spatial columns, or you can add spatial columns to existing tables. The geographic features can be inserted into the spatial columns. The Spatial Extender includes functions that will convert spatial data into its own storage format from one of three external formats including:

- OGIS well-known text representation
- OGIS well-known binary representation
- ESRI shapes

Using the Spatial Extender functions, an application can be written to populate columns in a spatially enabled database.

After integrating spatial data into the database, you can include Spatial Extender functions in your SQL statements that compare the values of spatial columns, transform the values into other spatial data, and describe the properties of the data.

The Spatial Database Engine Component

The ESRI Spatial Database Engine (SDE) is a gateway that interfaces between GIS clients and database engines. The SDE for DataJoiner is designed to work specifically with the Spatial Extender to interface between GIS clients (using SDE APIs) and DataJoiner (using ODBC).

In most scenarios, the SDE for DataJoiner will reside on the same server as DataJoiner and the Spatial Extender.

The SDE comes with many utilities to work with GIS data, including:

geocoder
Loads spatial data into a DataJoiner database

map loader and map exporter
Convert map data into spatial database data

import and export
Imports and exports various other types of data

For more information on using the SDE, refer to the *SDE Administrator's Guide* .

The ArcView Client Component

ArcView is a powerful client used to view and analyze geographic data.

To interface with relational databases, ArcView has a database extension, which it uses to create themes. These themes contain data that can be displayed both geographically and in tabular form.

ArcView also comes with a query composer, which allows you to interact with the database directly, as well as with the themes themselves.

For more information, refer to the ArcView documentation that comes with the product.

Current Limitations

Only DataJoiner databases can be spatially enabled. Nicknames, type mappings, and function mappings are not applicable to spatially-enabled databases. Spatial function is available from a remote DataJoiner server only by using pass-through. Furthermore, replication between tables with spatially-enabled columns is not supported.

The Spatial Extender implements the OpenGIS SQL specification through spatial data types, spatial functions, and spatial indexes. The currently supported data types are point, line, and polygon. These can be used as column data types in both **CREATE TABLE** and **ALTER TABLE** statements. However, the following limitations apply:

- The **with default** clause is not allowed. Use of the clause will generate an error.
- The **lob options** clause is not necessary, nor is it allowed. Use of the clause will produce a syntax error.

Some general limitations on spatially-enabled columns are:

- The functions **order by**, **group by**, **avg**, **sum**, **min**, **max**, **count**, **distinct**, and **length** are not allowed.
- The column cannot be a primary key or foreign key. If it is, a syntax error will result.
- The column cannot be included in the insert column list for **import** and **load** commands.
- The column cannot be included in the select column list for **export**.
- Capture and apply have the same restrictions for spatial data types as they do for LOBs.
- **runstats** will not gather statistics on spatial columns or spatial indexes.
- **db2 reorg table** cannot reorganize a table with a spatial index.

- **db2 reorg** will not reorganize any LOB data for a table that has a spatially-enabled column.

Chapter 2. Setting-up the Environment

This section details the step-by-step procedures for installing the three products provided in the entitled release of the DB2 Spatial Extender:

- DB2® Spatial Extender Version 2.1.1 (Windows NT® or AIX®)
- ESRI Spatial Database Engine™ Version 3.0.2 (Windows NT or AIX)
- ESRI ArcView™ GIS Version 3.0a (Windows NT only)

Types of Configuration

These products can be installed on a stand-alone system or in a client-server configuration.

Stand-Alone on Windows NT

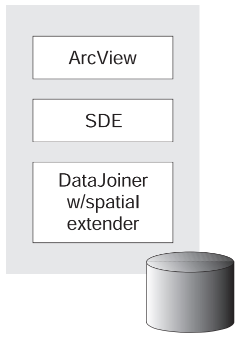


Figure 2. Stand-alone setup

	NT stand-alone
Disk space:	300M
Memory:	64M

A stand-alone system has all the components installed on the same Windows NT machine. This is a relatively simple configuration and is good for demonstration purposes.

If your organization has a GIS analyst who will be connecting with a shape database on a large remote system, and if you have a powerful NT workstation, you might choose this setup as a ″heavy″ client to that remote database.

Client-Server

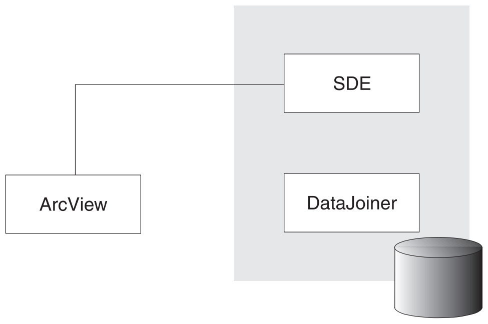


Figure 3. Client-server setup

	NT|95 client	NT|AIX server
Disk space:	100M	200M
Memory:	32M	64M

In this scenario, the Spatial Database Engine (SDE) and DataJoiner with the Spatial Extender can co-exist on an NT or AIX server to which one or more ArcView clients can connect. This is the recommended configuration.

Installing the Components

Before proceeding, ensure you have checked the README file on the CD for any changes to the installation process. The following sequence to install the components is presented from back-to-front:

1. Install and configure DataJoiner with the Spatial Extender.
2. Install and configure SDE.
3. Install and configure ArcView.

Configuring DataJoiner (AIX or NT)

If you have not already done so, install DataJoiner with the Spatial Extender. The instructions are in the *DataJoiner Planning, Installation, and Configuration Guide* for either AIX or NT. You do not yet need to configure DataJoiner to its data sources, unless you want to use DataJoiner in that context. You do, however, need to configure it to communicate with clients using TCP/IP. When you have completed these steps, proceed with the following configuration instructions that are relevant to your server platform:

For AIX

1. Log on with the user ID of a DataJoiner instance. This instance will manage the spatial database, and must be configured to communicate with ODBC clients using TCP/IP. You can ensure this has been done by starting DB2 and checking if the **db2tcpcm** process is running:

   ```
   $ db2start
   $ ps -ef | grep 'whoami'
   ```

   If the process is not running, refer to the *IBM DataJoiner for AIX Planning, Installation, and Configuration Guide* and configure TCP/IP client communications before proceeding.
2. Swich user to **root** and create a new user ID, named **db2se**, which the SDE will use to communicate to DataJoiner:

   ```
   $ su root
   $ mkuser pgrp='groupname' admgroups='groupname' db2se
   $ passwd password
   ```

   where *groupname* is the name of the primary group of the DataJoiner instance. (You can determine the name of the group with the command: **lsuser -a pgrp** *instance_ID*.)
3. Switch user to **db2se** and add the DataJoiner instance's profile information to the .profile file. One way to accomplish this is by appending the following entry to the file:

   ```
   . /djinst_path/sqllib/db2profile
   ```

   where *djinst_path* is the complete path to the home directory of the instance.
4. After switching back to the instance user (exit twice), increase the UDF memory size.

   ```
   $ db2 update dbm cfg using UDF_MEM_SZ 4096
   ```

   For more details on the memory requirements for the Spatial Extender, refer to the Software Requirements table in the *DataJoiner Planning, Installation, and Configuration Guide*.
5. Create a spatially-enabled database:

   ```
   $ create db database_name
   $ db2seadm enable_db database_name
   ```
6. Grant dbadm authority to the **db2se** user ID.

   ```
   $ db2 connect to database_name
   $ db2 grant dbadm on database to db2se
   ```

For NT

1. Log on to the Windows NT system with a user ID in the Administrators group that also has DataJoiner SYSADM authority for the instance (for example, the user ID that you installed DataJoiner under). This instance will manage the spatial database, and must be configured to communicate with ODBC clients using TCP/IP. You can

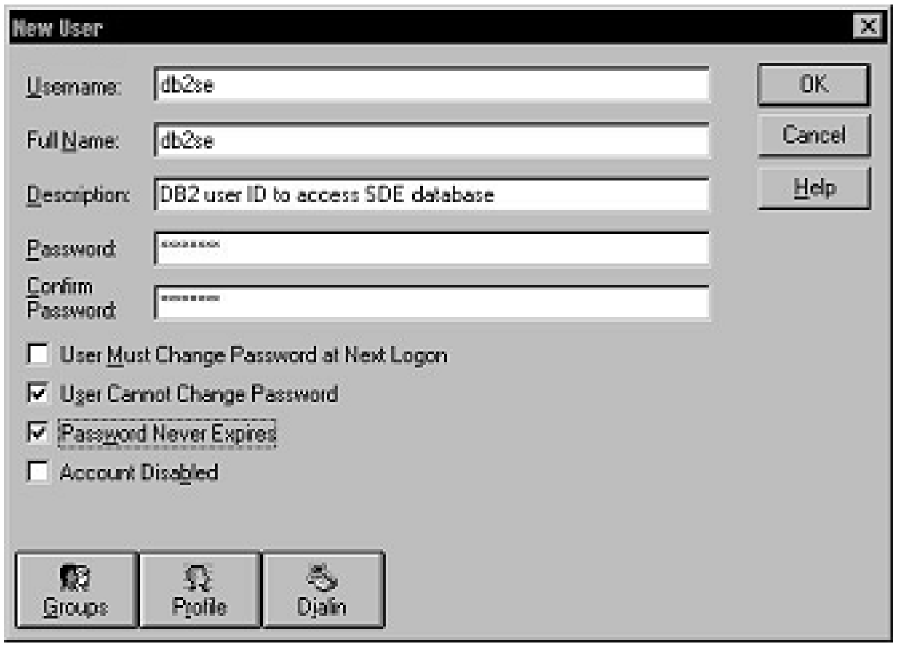
verify this by viewing the database manager configuration and checking that the title states that it is configured for local and remote access, and that there is an existing SVCENAME.

```
db2 get dbm cfg
```

If DataJoiner is only configured for local access, refer to the *IBM DataJoiner for Windows NT Planning, Installation, and Configuration Guide* and configure TCP/IP client communications before proceeding.

2. Create a new user ID to be used to access the spatial data. Name this user **db2se**.
   a. From the Start menu, select **Programs** ->**Administrator Tools** ->**User Manager**.
   b. From the **User** menu, select **New User**.
   c. In the **New User** dialog box:
      1) Enter db2se as the **New User** name.
      2) Enter a password in the **Password** and **Confirm Password** fields.

         Note: You will need to know this password when configuring SDE.
      3) Click the **Groups** button, add db2se to the Administrators group, and click **OK** to close the window.
      4) Click **OK** to close the **New User** window.


New User

Username: db2se

Full Name: db2se

Description: DB2 user ID to access SDE database

Password: *******

Confirm Password: *******

- [ ] User Must Change Password at Next Logon
- [x] User Cannot Change Password
- [x] Password Never Expires
- [ ] Account Disabled

Figure 4. Create the db2se user

3. Increase the UDF memory heap size:

   ```
   D:\sqllib> db2 update dbm cfg using UDF_MEM_SZ 4096
   ```

   For more details on the memory requirements for the Spatial Extender, refer to the Software Requirements table in the *DataJoiner Planning, Installation, and Configuration Guide*.
4. Create a database:

   ```
   D:\sqllib> db2 create db sde
   ```

   The name **sde** is the default database name for SDE. For simplicity, it is recommended you use this name.
5. Spatially enable the database:

   ```
   D:\sqllib> db2seadm enable_db sde
   ```
6. Grant the user ID, **db2se**, database administrator authority:

   ```
   D:\sqllib> db2 connect to database_name
   D:\sqllib> db2 grant dbadm on database to user db2se
   ```
7. Start the DB2 security service:
   a. Open the **Services** control panel, which can be found in **Start->Settings->Control Panel**.
   b. Find the ″DB2 Security Server″ and click **Start**.
8. Close all programs and log in as user **sde**.

9. Verify that the **db2se** user can log on to DB2 from this ID:

```
D:\sqllib> db2 connect reset
D:\sqllib> db2 connect to sde user db2se
```

Configuring the SDE (AIX or NT)

Using the values you assigned to DataJoiner and the SDE database in the previous section, follow the instructions relevant to your server platform:

For AIX

1. Log in as **root** and create a user ID to install and run the SDE server software (for these examples the user is named **sde**):

```
$ mkuser pgrp='staff' admgroups='staff' sde
$ passwd sde
```

2. Insert the CD and mount the CD-ROM drive:

```
 $ mount -v 'cdrfs' -p ' ' -r ' ' /device_name  /cdrom
```

3. Edit the /etc/services file, and add an entry for the SDE listener:

```
esri_sde  5150/tcp     # SDE
```

4. Log on with the **sde** user ID and install the SDE:

```
$ cd /usr/lpp  (or the directory where you install system-wide programs)
$ mkdir sde
$ cd sde
$ /cdrom/SDE/RS6000/install -load
```

5. Add the following lines to the .profile of the **sde** user:

```
export SDEHOME=/usr/lpp/sde/sdeexe302
export PATH=$PATH:$SDEHOME/bin
export LIBPATH=$LIBPATH:$SDEHOME/lib
export DB2COMM=TCPIP
```

6. Start the SDE server, and provide the password of the **db2se** user on the DataJoiner system:

```
$ sdemon -o start
```

7. Verify the SDE processes are running:

```
$ sdemon -o status
```

For NT

1. Log on with a user ID in the Administrators group.
2.

   Modify the system environment:

   a. Right-click the **My Computer** icon, and select **Properties**.
   b. Click the **Environment** tab, and add the following entries to the system variables:

Variable	Value
SDEHOME	*SDE_install_path*\db2exe\sdeexe302
PATH	%PATH%;%SDEHOME%\bin

3. Use the same procedure as in step 2 on page 8 to create a user ID to be used to run the SDE service (named **sde** in the figures). The user must be a member of the *Power Users* group. If the user is a Domain user, it must be able to start the Net Logon service. Workgroup users cannot start the service.

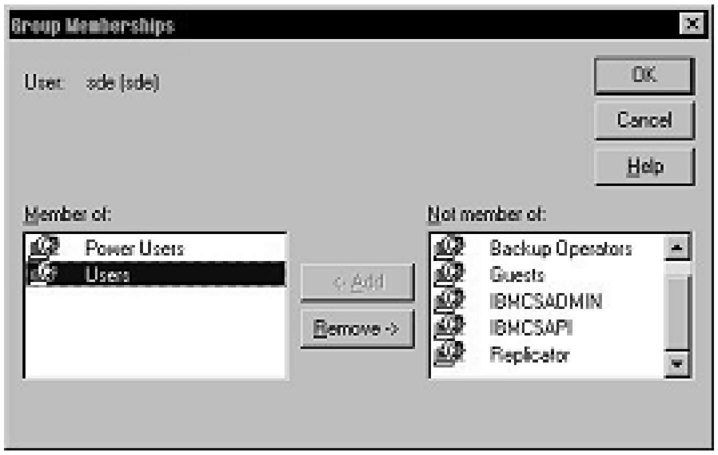
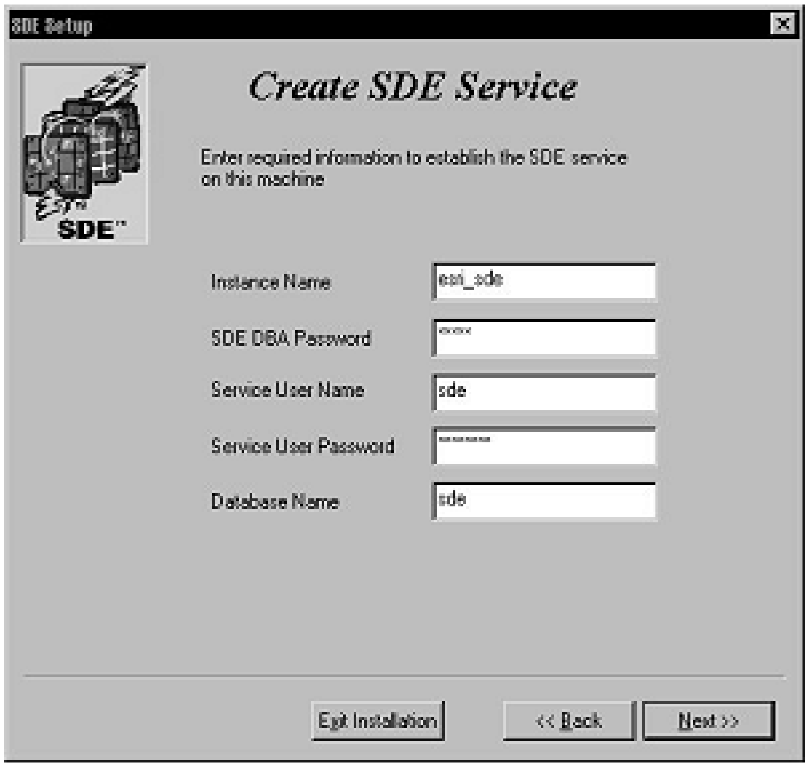


Figure 5. Adding the SDE Service ID to the Power User group

4. Edit the C:\winnt\system32\drivers\etc\services file and add an entry for the SDE listener:

```
esri_sde  5150/tcp     # SDE
```

5. Run the SDE setup program, which is located in the directory, SDE\intel_nt, on the CD.
6. Follow the prompts of the installation wizard, and ensure you enter the correct values on the **Create SDE Service** panel:


SDE Setup
Create SDE Service
Enter required information to establish the SDE service on this machine
SDE
Exit Installation
<< Back
Next >>

Instance Name	esri_sde
SDE DBA Password	*****
Service User Name	sde
Service User Password	*******
Database Name	sde

Figure 6. SDE Installation - Create SDE Service panel

Instance Name
The TCPIP service name. `esri_sde` is the default.

SDE DBA Password
The password of the **db2se** user ID.

Service User Name
The user ID that will run the SDE Service (**sde** in the examples).

Service User Password
The password of the SDE Service ID

Database Name
The name of the spatially enabled database. `sde` is the default, which is the name of the database created in the section, "Configuring DataJoiner (AIX or NT)" on page 6.

7. When the installation is complete, restart the system and verify the SDE service (**esri_sde**) is running.
   a. Open the **Services** control panel, found in **Start->Settings->Control Panel**.

b. Scroll to find the SDE Service (esri_sde), which should be set as ″Started″ and ″Automatic.″

Configuring ArcView (Windows 95 or NT only)

1. Install ArcView following the instructions in the *ArcView GIS Guide*.
2. Install the Database Access Extension, which is available from the ESRI web site at http://www.esri.com. Search for the ArcView Database Access Extension, which at the time of this publication is located at: http://www.esri.com/base/products/arcview/extensions/dbaccess/windows.html

Installation Verification

Once the system is up and running, you can verify the installation by loading a shape file and creating a database theme using the ArcView client:

- To learn more about loading a shape file, refer to the **shp2sde** command in the *SDE Administrator's Guide*. Sample shape files are located on the ArcView CD.
- To create a database theme, refer to the *Introduction to ArcView Database Access* book or the ″Extensions″ chapter of the ArcView online help.

Chapter 3. Working with Spatial Extender Data

A spatial database is a relational database like any other, administered using the same familiar tasks, such as authorizing users, cataloging, and indexing. The Spatial Extender provides metadata and an indexing scheme to help the DBA carry out effective administration of the system. Proper administration enables the programmer to create high performance spatial applications. This chapter describes these database extensions, and presents the tools necessary to ensure proper administration is carried out.

Administrative Privileges

Administrative processes for the Spatial Extender are performed by the **db2se** user ID, which is granted DBADM authority during setup. DBADM or SYSADM privileges are required to run **db2seadm**.

If a user other than the DBA of the DB2 database will own the Spatial Extender functions and data types, the DBA must grant that user CREATE_NOT_FENCED privileges. The Spatial Extender creates its functions in unfenced mode so that it can run them in the database address space for better performance.

Users who want to use the Spatial Extender must add db2se to their function path:

```
SET CURRENT FUNCTION PATH = SYSTEM PATH, db2se
```

Administering the Spatial Extender with db2seadm

The Spatial Extender comes with a program, **db2seadm**, that is used to enable or disable spatial capabilities for a database. This program has the following switches:

enable_db
: Enables a database for spatial data. It creates the meta tables and views, and the spatial data types, functions, predicates, and index types.

disable_db
: Disables a spatial database by dropping the data types, functions, index types, and the meta tables and views.

enable_col
: Performed on a column in a spatially-enabled database, this option registers the column in the DB2_GEO_COLUMNS table and creates a check constraint on the column.

disable_col
: Disables a spatial column by dropping the check constraint and removing the entry from the DB2_GEO_COLUMNS table.

enable_index
Creates a spatial index on the designated spatial column and updates the DB2_GEO_COLUMNS table to include the index information.
- The *update* option updates the index information in the table.
- The *recover* option creates an index and adds a check constraint.

disable_index
Drops the spatial index on a column.
- The *drop* option drops the index and updates the DB2_GEO_COLUMNS to not include the index information.
- The *invalidate* option drops the index, but only removes the check constraint and leaves the information in the DB2_GEO_COLUMNS table.

register
Registers the entitled product using a registration key.

Administering the Spatial Extender with SDE

After spatially enabling a database, you can use the SDE administration tools to perform many of the tasks described later in this chapter. Refer to the *SDE Administrator's Guide* for information on using the SDE tools.

With the SDE, you can:
- Create a table with a spatial column (use sdetable).
- Spatially enable a column, and generate the spatial index on it (use sdelayer).
- Accept a grid cell size of the spatial index (use sdelayer). "Selecting the grid cell size" on page 32 has more information on selecting the grid cell size for the spatial index.

The SDE automatically adds a record to the spatial reference table whenever it cannot find a compatible one.

SDE uses the term *layer* to refer to a spatial column of a DataJoiner table. SDE layers may be in either LOAD ONLY I/O mode or NORMAL I/O mode.

When a layer is in LOAD ONLY I/O mode, the spatial index is dropped and queries through the SDE server are disallowed. This state allows the tables to be efficiently loaded by a program created with the SDE C API or with the shp2sde, cov2sde, and sdeimport utilities. These utilities convert shape files, coverages, and SDE export files. When the layer is returned to NORMAL I/O mode, the spatial index is created and queries are once again allowed.

All SDE client applications work with the SDE included with DataJoiner, including ArcView, MapObjects, ArcInfo, ArcExplorer applications. MicroStation and AutoCad are also accessible through the ESRI SDE CAD Client product.

A Sample Program on AIX

A set of sample scripts is provided with the Spatial Extender AIX installation. These scripts perform the following tasks:

- Enable a database for spatial data
- Create a table that contains both spatial and traditional columns
- Create spatial indexes over spatial columns
- Load spatial and traditional data into a spatially enabled table
- Query tables and views using traditional predicates and spatial predicates, such as: within, contain, and intersect
- Disable the spatial database
- Disable a spatial column

The scripts are located in the directory, *$INSTHOME*/sqllib/samples/db2sampl, where *$INSTHOME* represents the home directory of a DataJoiner instance.

1. Log in as the user you want to own the sample tables and data. To demonstrate all the functions in the sample, this user must have DBADM or SYSADM authority to this instance.
2. Create a directory in the user’s home directory, and copy the geocoder and all the script files to this directory. This step is necessary to give the user a set of scripts with execute permission. For example:
   ```
   mkdir ˜/db2sampl
   cp $INSTHOME /sqllib/samples/db2sampl/* ˜/db2sampl
   ```
3. Print these scripts for reference.
4. Ensure DB2 is started:
   ```
   db2start
   ```
5. Run the top-level script, redirecting the output to a file:
   ```
   db2seDemo >demo.txt
   ```
6. Browse the output file to examine the results of the demonstration.

Tuning DataJoiner for Spatial Data

The following tuning tips will improve DataJoiner performance when handling spatial data:

- Using a DMS tablespace is generally faster than using an SMS tablespace, especially when populating an empty table:
  ```
  CREATE TABLESPACE dmsTs MANAGED BY DATABASE USING
          (FILE '/drive1/dms1' 100000, FILE '/drive2/dms1' 100000, ...)
           EXTENTSIZE 32
           PREFETCHSIZE 128
  ```
- Set the prefetch size to be a multiple of the extent size. For example, in the CREATE statement above, the prefetch size is set to 32 * 4, assuming that the extent size is 32 and the number of physical drives used by the table space is 4.

- Set the number of prefetcher processes to the number of physical drives plus 2. For example, for 4 drives, set the number of processes to 6 using this command:

```
UPDATE DB CFG FOR sampleDb UPDATE NUM_IOSERVERS 6
```

For more information on tuning DataJoiner, see the *DB2 Administration Guide* and *IBM DataJoiner Administration Supplement*.

Spatial Data Types

When you spatially enable a database using the **enable_db** command, seven spatial data types are added to your database: geometry, point, linestring, polygon, multipoint, multilinestring, and multipolygon.

A thorough discussion of these data types is provided in “About Geometry” on page 37.

Spatial Functions and Predicates

The Spatial Extender includes many functions and predicates that store, access, and model spatial data:

Client-server conversion functions
These convert spatial data between the client and the DB2 Spatial Extender database. This data can be in ESRI Binary, OGIS Binary, or OGIS Text formats. The specifications for these data formats are included in the Appendixes.

Property functions
These describe the properties of spatial data. These functions answer questions about a particular geometric shape (also called a geometry) such as: is this geometry empty? ...what is it’s type? ...how many rings does it include?

Relational functions
These determine the relationships between geometries, and answer questions such as: do these lines cross? ...what is the distance between two shapes? ...taking the union of two areas, what points are located within the boundaries, give or take a couple of miles?

Spatial functions and predicates are described in detail in “Chapter 4. Programming Concepts” on page 37 and “Chapter 5. SQL Reference” on page 67.

Spatial Index Extensions

The two-dimensional nature of spatial data requires an indexing structure beyond the capabilities of the DataJoiner B+ Tree index. Therefore, the Spatial Extender extends DataJoiner’s index to include a strategy that uses grids. For more information, see “Spatial Indexes” on page 25.

Meta Tables and Views

The complex nature of spatial data requires that relationships be maintained in various meta tables. These tables store particular sets of information about the spatial data, such as names, types, keys, schemas, and index information.

SPATIAL_REFERENCES and SPATIAL_REF_SYS

The SPATIAL_REFERENCES table stores all of the possible spatial reference systems of a database. The SPATIAL_REFERENCES table is created with the following create table statement:

```
create table db2se.SPATIAL_REFERENCES (
       srid         int           not null,
       auth_name    varchar(256),
       auth_srid    int,
       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       varchar(2048)  not null,
       constraint   sp_ref_pk     primary key (srid));
```

The SPATIAL_REF_SYS view is created with the following create view statement:

```
create view db2se.SPATIAL_REF_SYS as
select srid, auth_name, auth_srid, srtext
from db2se.SPATIAL_REFERENCES;
```

This view is required by the OGIS specification and used by the SDE.

See “Chapter 4. Programming Concepts” on page 37 for information on spatial reference systems.

DB2_GEO_COLUMNS and GEOMETRY_COLUMNS

The DB2_GEO_COLUMNS table is created with the following create table statement:

```
create table db2se.DB2_GEO_COLUMNS (
f_table_catalog       varchar(256),
f_table_schema        varchar(8)     not null,
f_table_name          varchar(18)    not null,
f_geometry_column     varchar(18)    not null,
geometry_type         int,
srid                  int,
storage_type          int,
coordinate_dimension  int,
b_table_schema        varchar(8),
b_table_name          varchar(18),
b_geometry_column      varchar(18),
constraint_name        varchar(18),
idx_schema             varchar(18),
idx_name               varchar(18),
constraint geocol_pk primary key
   (f_table_schema,f_table_name,f_geometry_column),
constraint geocol_fk foreign key (srid)
   references db2se.SPATIAL_REFERENCES (srid));
```

The GEOMETRY_COLUMNS view is created with the following create view statement:

```
create view db2se.GEOMETRY_COLUMNS as
  select f_table_catalog,  f_table_schema,
         f_table_name,     f_geometry_column,
         geometry_type,    srid,
         storage_type
  from db2se.DB2_GEO_COLUMNS;
```

When a database is spatially enabled, the DB2_GEO_COLUMNS table stores meta data about each spatial column in the database.

This view is required by the OGIS specification and used by the SDE.

Database Privileges

Any user of the database may query the DB2_GEO_COLUMNS table and GEOMETRY_COLUMNS view, but only the db2se user is granted permission to change their contents.

Any user of the database may query or change the contents of the SPATIAL_REFERENCES table and its view, SPATIAL_REF_SYS.

The following grant statements are executed to grant privileges on the db2se tables and views.

```
grant select on DB2_GEO_COLUMNS to public;
grant select on GEOMETRY_COLUMNS to public;
grant select on SPATIAL_REFERENCES to public;
grant select on SPATIAL_REF_SYS to public;
grant insert on DB2_GEO_COLUMNS to db2se;
grant insert on GEOMETRY_COLUMNS to db2se;
grant insert on SPATIAL_REFERENCES to public;
grant delete on DB2_GEO_COLUMNS to db2se;
grant delete on GEOMETRY_COLUMNS to db2se;
grant delete on SPATIAL_REFERENCES to public;
grant delete on SPATIAL_REF_SYS to public;
grant update on DB2_GEO_COLUMNS to db2se;
grant update on GEOMETRY_COLUMNS to db2se;
grant update on SPATIAL_REFERENCES to public;
grant update on SPATIAL_REF_SYS to public;
```

Adding records to the spatial reference table

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 created by the db2seadm enable_db option.

Internal functions use the parameters of a spatial reference system to translate and scale each floating point coordinate of the geometry into 32-bit positive integers prior to storage. Upon retrieval, the Spatial Extender restores the coordinates to their external floating point format.

The Spatial Extender converts the floating point coordinates to integers by first subtracting FALSEX and FALSEY, which translates to the false origin and then scales by multiplying by the XYUNITS, adding a half unit, and truncating the remainder.

The optional Z coordinate and measure are dealt with in the same fashion, except that they are translated with FALSEZ and FALSEM and scaled with ZUNITS and MUNITS, respectively.

The SPATIAL_REFERENCES primary key, spatial reference identifier (SRID), contains a unique number for each spatial reference system. The SRTEXT contains the Well-Known Text representation of the Spatial Reference System. (see "Appendix B. The OGIS Well-Known Text Representation" on page 187).

The AUTH_NAME contains the name of the standard or standards body cited for the reference system, and AUTH_ID is the identifier number of the Spatial Reference System as defined by the authority cited in AUTH_NAME.

The spatial reference system is assigned to a geometry during its construction. The spatial reference system must exist in the Spatial Reference Table. The Spatial Extender requires that all geometries in a column be of the same spatial reference system.

Below is an example of a spatial reference system inserted into the SPATIAL_REFERENCES table.

```
insert into db2se.SPATIAL_REFERENCES
values ( 1,'NULL',NULL,-1000.0,-1000.0,1000,-1000.0,1000,-1000.0,1000,'UNKNOWN' );
```

Table 1. Columns and the values stored in the SPATIAL_REFERENCES table.

Column	**Value**
srid	1
auth_name	NULL
auth_srid	NULL
falsex	-1000.0
falsey	-1000.0

Table 1. Columns and the values stored in the SPATIAL_REFERENCES table. (continued)

Column	Value
xyunits	1000
falsez	-1000.0
zunits	1000
falsem	-1000.0
munits	1000
srtext	UNKNOWN

Because the coordinates are stored as positive 32-bit integers, the maximum range of values is between 0 and 2,147,483,648, which, of course, is dependent on the false origin and system units of the spatial reference system.

A negative false origin will shift the range of values in the negative direction and positive direction otherwise. For example, a false origin of 1000.0 with a system unit of 0 will store a range of values between 1000.0 to 2,147,482,648.

The system units control scale. The larger the system unit the greater the scale that can be stored, but it also reduces the range of values. For example, given a system unit of 1000 and a false origin of 0, the Spatial Extender will store a scale that is three digits to the right of the decimal point; the range of values stored will be reduced to 0.001 and 2,147,483.648. In contrast, decreasing the system unit will increase the range of values, but will proportionately decrease the scale as truncation occurs. For example, a system unit of 0.001 and a false origin of 0 will store a range of values that is between 1000.0 and 2147,483,648,000.0.

The following formula converts the floating point ordinates into system units.

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

The Spatial Extender will return an error if the application attempts to store a value that is less than or greater than the range of acceptable values. Therefore, it is important that you select a false origin and system unit that will store all of your ordinate values at an acceptable scale. To do so, you must know the range of your data and the scale you wish to maintain. For example, if you wish to maintain a scale of three digits to the right of the decimal point, set your system units to 1000. Set the false origin less than the minimum ordinate value in your data set. Note that the false origin must be small enough to account for any buffering of the data. So if the minimum ordinate value is 10000, and your application includes functions which buffer the data by 5000, then the false origin must be set less than 15000. Finally make sure that the maximum ordinate value will not be greater than 2,147,483,648 after it has been converted to a positive 32-bit integer. To do this, apply the formula to the maximum value. For example, given a false origin of 15000 and a system unit of 1000, the maximum ordinate 9302912.021 would be converted to 9,317,912,021.

```
9,317,912,021 = truncate(((9302912.021  (-15000) * 1000) + 0.5)
```

This value is larger than 2,147,483,648; therefore, the Spatial Extender would reject errors for all ordinates larger than 2132483.648. Because it is not possible to shift the false origin to the positive direction, the only alternative in this case is to reduce the system units to 100. Now the maximum value is calculated as 931,791,202.

```
931,791,202 = truncate(((9302912.021  (-15000) * 100) + 0.5)
```

Because the value is less than the 2,147,483,648, all of the ordinates will now fit. Note that the need to lower the system unit rarely occurs. The range of values in the vast majority of data sets are not large enough to require this action.

Creating a Spatial Table

A spatial table is simply a table that includes one or more spatial columns. Spatial columns are columns created with one of the spatial data types. These columns will contain spatial data or geometry. (Geometry is the term adopted by the Open GIS Consortium to refer to two-dimensional spatial data). To create such a table, simply include a spatial column in the column clause of the CREATE TABLE statement. This column can only accept data of the type required by the spatial column. For example, a polygon column rejects integers, characters, and even other types of geometry.

Consider again the sensitive areas and hazardous waste sites example. Stored in the SENSITIVE_AREAS table are the threatened schools, hospitals and playgrounds, while the HAZARDOUS_SITES table maintains the hazardous waste sites. The polygon data type is used to store the sensitive areas, while the hazardous sites are stored as points.

```
create table SENSITIVE_AREAS (id        integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      polygon);

create table HAZARDOUS_SITES (site_id   integer,
                              name      varchar(128),
                              location  point);
```

Inserting Data into a Spatial Column

The geometry stored in a spatial column actually combines several other base data types and as such cannot be entered directly. Instead the standard geometry formats must be loaded with special conversion functions. The Spatial Extender supports three external geometry formats: text representation, well known binary representation, and ESRIs shape representation. For each of these formats a function exists to convert the data into each of the Spatial Extenders data types. (See “Geometry Data Exchange” on page 45 for a complete list of functions.)

In the code fragment below, a record is inserted into the SENSITIVE_AREAS and HAZARDOUS_SITES tables. The polyfromshape function converts an ESRI polygon shape into a Spatial Extender polygon before inserting it into the ZONE column of SENSITIVE_AREAS. The pointfromshape function converts an ESRI point shape into a Spatial Extender point before inserting it into the LOCATION column of HAZARDOUS_SITES. If the original form of the geometry were a text representation, you would use the functions polyfromtext and pointfromtext. Likewise, a geometry in well-known binary representation can be converted using polyfromwkb and pointfromwkb.

```
/* Create the SQL insert statements to populate the sensitive areas table.
   The question mark is a parameter marker that indicates the zone polygon
   that will be retrieved at runtime from the variable shape_poly. */
strcpy (shp_sql," insert into SENSITIVE_AREAS values('408',
                  'Summerhill Elementary School',
                   100493.94,
                  'SCHOOL',
                   polyfromshape(cast(? as blob(1m)),coordref()..srid(1))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the shape to the second parameter. */
pcbvalue1 = blob_len;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, shape_poly, blob_len, &pcbvalue1);

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

/* Create the SQL insert statements to populate the hazardous
   sites table. The question mark is a parameter marker that
   indicates the location point that will be retrieved at
   runtime from the variable shape_point. */
 strcpy (shp_sql,"insert into HAZARDOUS_SITES (102,
'W. H. Kleenare Chemical Repository',
pointfromshape(cast(? as blob(1m)),coordref()..srid(1))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the shape to the second parameter. */
```

```
 pcbvalue1 = blob_len;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, shape_point, blob_len, &pcbvalue1);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

In the example above, two records are simply inserted into the tables. However, the actual amount of data that must be loaded into a GIS system usually ranges between ten thousand records for smaller systems and one hundred million records for larger systems. The seamless design of the Spatial Extender is capable of handling the entire range of these systems. To load vast amounts of spatial data it is necessary for an application to convert the data into one of the three accepted formats before it can store the data.

Spatial Indexes

Because spatial columns contain two-dimensional geographic data, applications querying those columns require an index strategy that quickly identifies all geometries that lie within a given extent. For this reason, the Spatial Extender provides the three-tiered spatial index based on a grid.

Before reading more about spatial indexes, here is an example of how an index might be created and used in SQL. Notice that once the index is created, you can then perform standard DDL and DML statements on the database, using the spatial functions and predicates:

```
create table customers (cid int, addr varchar(40), ..., loc point)
create table stores (sid int, addr varchar(40), ..., loc point, zone polygon)

create index customersx1 on customers(loc) using spatial_index(1000e0, 100e0, 10e0)
create index storesx1 on customers(loc) using spatial_index(1000e0, 100e0, 10e0)
create index storesx2 on customers(loc) using spatial_index(10000e0, 1000e0, 100e0)

insert into customers (cid, addr, loc) values (:cid, :addr, sdeFromBinary(:loc))
insert into customers (cid, addr, loc) values (:cid, :addr, geocode(:addr))
insert into stores (sid, addr, loc) values (:sid, :addr, sdeFromBinary(:loc))

update stores set zone = buffer(loc, 2)

select cid, loc from customers
 where within(loc, :polygon) = 1

select cid, loc from customers
 where within(loc, :circle1) = 1 OR
       within(loc, :circle2) = 1

select c.cid, loc from customers c, stores s
 where contains(s.loc, c.loc) = 1
```

```
select avg(c.income) from customers c
 where not exist (select * from stores s
                  where distance(c.loc, s.loc) < 10)
```

The B+ tree indexes

The two-dimensional spatial index differs from the traditional hierarchical B+ tree index provided by DataJoiner. To better understand the difference, let us review how a B+ tree index is structured and used.

The top level of a B+ tree index, called the root node, contains one key for each node at the next level. The value of each of these keys is the largest existing key value for the corresponding node at the next level. Depending on the number of values in the base table, several intermediate nodes may be needed. These nodes form a bridge between the root node and the leaf nodes that hold the actual base table row IDs.

The DataJoiner database manager searches a B+ tree index starting at the root node and then continues through the intermediate nodes until it reaches the leaf node with the row ID of the base table.

The B+ tree index cannot be applied to a spatial column because the two-dimensional characteristic of the spatial column requires the structure of a spatial index. For the same reason, you cannot apply a spatial index to a non-spatial column. Further, a spatial index cannot be applied to a composite column of any kind.

To create a spatial index, use the enable_index parameter of the **db2seadm** command. You can also use the **db2 create index** command with the spatial_index function in the USING clause.

```
db2seadm enable_index create <database>, <schema>, <table>, <spatial column>,
                             <grid level 1>, <grid level 2>, <grid level 3>

create index storesx1 on customers(loc) using spatial_index(1000e0, 100e0, 10e0)
```

Because of the simple nature of the data, a B+ tree was used to design the index. The database designer directs DataJoiner to create the index on one or more table columns. The nature of spatial data requires the designer understand its relative size distribution. The designer must determine the optimum size and number of grid levels with which to create the spatial index.

The grid levels, <grid level 1>, <grid level 2>, and <grid level 3>, are entered by increasing the cell size. Thus, the second level must have a larger cell size than the first, and the third larger than the second. The first grid level is mandatory, but you can disable the second and third with a double precision zero value (0.0e0).

Generating the Spatial Index

A spatial index is generated using *envelopes*. The envelope, a geometry itself, represents the minimum and maximum X and Y extent of a geometry. For most geometries, the envelope is a box, but for horizontal and vertical linestrings the envelope is a two point linestring. For points, the envelope is the point itself. See "Chapter 4. Programming Concepts" on page 37 for more information about envelopes and other geometries.

The spatial index is constructed on a spatial column by making one or more entries for the intersections of each geometry's envelope with the grid. An intersection is recorded as the internal ID of the geometry and minimum X and Y coordinates of the grid cell intersected.

If multiple grid levels exist, the Spatial Extender attempts to use the lowest grid level possible. When a geometry has intersected four or more grid cells at a given level, it is promoted to the next higher level. Therefore, given a spatial index that has the three grid levels of 10.0e0, 100.0e0, and 1000.0e0, the Spatial Extender will first intersect each geometry with the level 10.0e0 grid. If a given geometry intersects with four or more 10.0e0 grid cells, it is promoted and intersected with the level 100.0e0 grid. If four or more intersections result at the 100.0e0 level, the geometry is promoted again to the 1000.0e0 level at which point the intersections must be entered into the spatial index since this is the highest possible level.

Figure 7 on page 28 illustrates how four different types of geometries intersect a 10.0e grid. All 23 intersections for the four geometries are recorded in the spatial index.

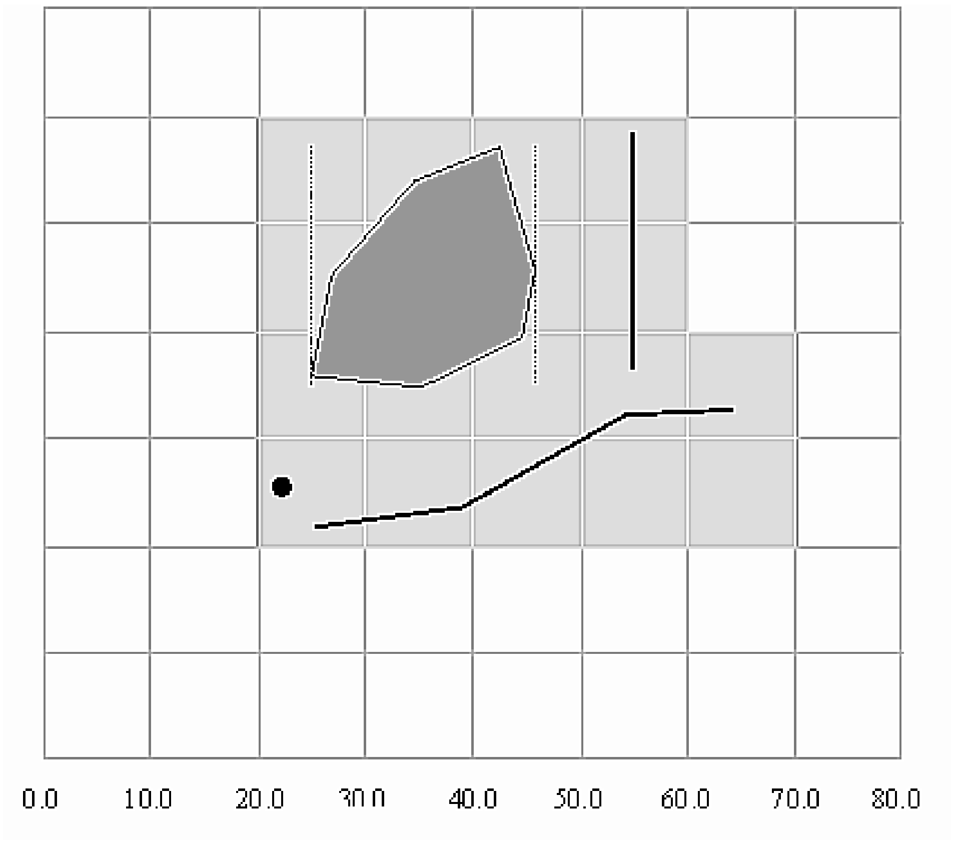


Figure 7. Application of a 10.0e0 grid level

Table 2 lists the geometries and their corresponding grid intersections. The envelopes of four different types of geometries intersect the 10.0e grid. The minimum X and Y coordinate of each grid cell that it intersects are entered into the spatial index.

Table 2. The 10.0e0 grid cell entries

Geometry	**Grid X**	**Grid Y**
Polygon	20.0	30.0
Polygon	30.0	30.0
Polygon	40.0	30.0
Polygon	20.0	40.0
Polygon	30.0	40/0
Polygon	40.0	40.0
Polygon	20.0	50.0
Polygon	30.0	50.0

Table 2. The 10.0e0 grid cell entries (continued)

Geometry	Grid X	Grid Y
Polygon	40.0	50.0
Vertical linestring	50.0	30.0
Vertical linestring	50.0	40.0
Vertical linestring	50.0	50.0
Point	20.0	20.0
Horizontal linestring	20.0	20.0
Horizontal linestring	30.0	20.0
Horizontal linestring	40.0	20.0
Horizontal linestring	50.0	20.0
Horizontal linestring	60.0	20.0
Horizontal linestring	20.0	30.0
Horizontal linestring	30.0	30.0
Horizontal linestring	40.0	30.0
Horizontal linestring	50.0	30.0
Horizontal linestring	60.0	30.0

Now let us examine a multilevel grid index of the geometries. Figure 8 on page 30 displays how the number of intersections is greatly reduced to eight by the addition of grid levels 30.0e0 and 60.0e0. In this case, the polygon identified as geometry 1 is promoted to grid level 30.0e0 and the linestring identified as geometry 4 is promoted to grid level 60.0e0. Instead of the nine and ten intersections that geometries had at the 10.0e0 level, they have only two after promotion.

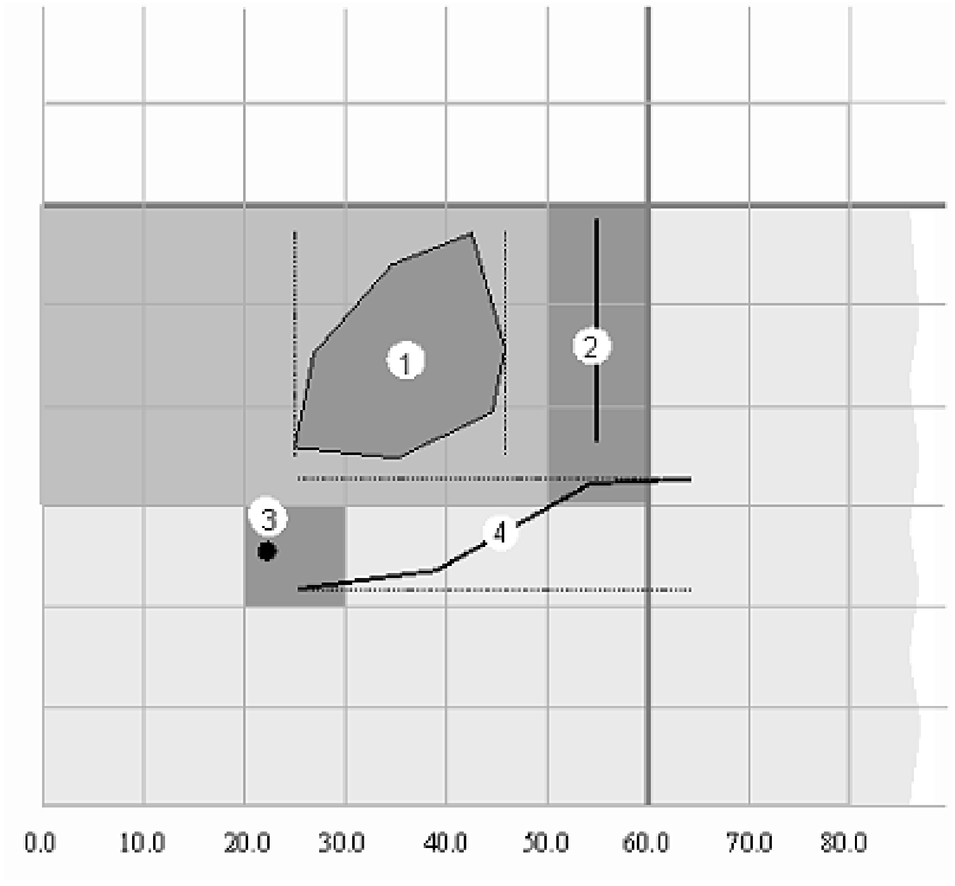


Figure 8. Effect of adding grid levels 30.0e0 and 60.0e0. The envelope of the polygon identified as geometry 1 intersects nine grid cells. The envelope of the vertical linestring identified as geometry 2 intersects three grid cells. The envelope of the point identified as geometry 3 intersects just one grid cell. The envelope of the linestring identified as geometry 4 intersects ten grid cells.

The grid levels, 10.0e0, 30.0e0, and 60.0e0, are displayed with ever-increasing line weights and different shades of gray. The vertical linestring and the point envelope cell intersections are entered into the index at the 10.0e0 grid level, because both generate less than four intersections. The polygon intersects nine 10.0e0 grid cells, and is therefore promoted to the 30.0e0 grid level. At this level, the polygon intersects two grid cells, which are entered into the index. The linestring identified as geometry 4 intersects ten 10.0e0 grid cells, and is therefore promoted to the 30.0e0 grid level. Yet at this level, it intersects six grid cells, so it is again promoted to the 60.0e0 grid level, where it generates two intersections. The linestring 60.0e0 grid intersections are then entered into the index. Had the linestring generated four or more intersections at this level, they still would have been entered into the index because this is the highest level at which a geometry can be promoted.

Table 3. The intersections of the geometries in the three-tiered index

Geometry	Grid X	Grid Y
The intersections between the vertical linestring and the point in the 10.0e0 grid level.		
2	50.0	30.0
2	50.0	40.0
2	50.0	50.0
3	20.0	20.0
The intersections of the polygon in the 30.0e0 grid level		
1	0.0	30.0
1	30.0	30.0
The intersections of the linestring in the 60.0e0 grid level		
4	0.0	0.0
4	60.0	0.0

The Spatial Extender does not actually create a polygon grid structure of any kind. The Spatial Extender manifests each grid level parametrically by defining the origin at the X,Y offset of the columns' spatial reference system. It then extends the grid into positive coordinate space. Using a parametric grid, the Spatial Extender generates the intersections mathematically.

Using the spatial index

The Spatial Extender works with a spatial index to improve the performance of a spatial query. Consider the most basic and probably most popular spatial query, the box query. This query asks the Spatial Extender to return all geometries that are either fully or partially within a user-defined box. If an index does not exist, the Spatial Extender must compare all of the geometries with the box. However, with an index, the Spatial Extender can locate all the index entries that have a lower left-hand coordinate greater than or equal the box's and an upper right-hand coordinate less than or equal to the box's. Because the index is ordered by this coordinate system, the Spatial Extender is able to quickly obtain a list of candidate geometries. The process just described is referred to as the *first pass*.

A *second pass* determines if each candidate's envelope intersects the box. A geometry that qualifies for first pass because its grid cells' envelope intersects the box may itself have an envelope that does not.

A *third pass* compares the actual coordinates of the candidate with the box to determine if any part of the geometry +is actually within the box. This last and rather complex process of comparison operates on a list of candidates composed of a sub-set of the total population, which is significantly reduced by the first two passes.

All spatial queries perform the three passes except for the *envelopesintersect* function. It performs only the first two passes and was designed for display operations because these types of operations often employ their own built-in clipping routines and don't require the granularity of the third pass.

Selecting the grid cell size

The irregular shape of the geometry envelopes complicates the selection of the grid cell size. Because of this irregularity, some geometry envelopes intersect several grids, while others fit inside a single grid cell. Conversely, depending on the spatial distribution of the data, some grid cells intersect many geometry envelopes.

For a spatial index to function well, it is essential that the correct number and size of grids are selected. To simplify this discussion, let's first consider a spatial column containing uniformly sized geometry. In this case, a single grid level will suffice. Start with a grid cell size that encompasses the average geometry envelope. While testing your application you may find that increasing the grid cell size improves the performance of your queries because each grid cell contains more geometries, and the first pass is able to discard non-qualifying geometries faster. However, you will find that as you continue to increase the cell size, performance will begin to deteriorate because eventually the second pass will have to contend with more candidates.

Selecting the number of levels

Not all columns will contain geometry of the same relative size. More often geometries of most spatial columns can be grouped into several size intervals. For example, consider a road network in which the geometries are divided into streets, major roads and highways. Or, consider a county parcel column that contains clusters of small urban parcels surrounded by larger rural parcels. These situations are very common and require the use of a multilevel grid.

To select the cell sizes of each level, first determine the intervals of geometry envelopes. Create a spatial index with grid level cell sizes that are slightly larger than each interval. Test the index by performing queries against the spatial column.

Each additional level requires an extra index scan, so try adjusting the grid sizes up or down slightly to determine if an appreciable improvement in performance can be obtained.

Querying a Spatial Table

Once the spatial indexes have been constructed, the spatial tables are ready for use. Querying the spatial columns requires that the data be converted to one of the three supported external formats. The function *astext* converts a spatial column value to the text representation, while the *asbinary* and *asbinaryshape* functions convert the values to WKB and ESRI shape representation. Once converted, the applications can display the data or manipulate it in whatever way they are required to do so.

The SDE server supplied with the Spatial Extender automatically converts the spatial column data into ESRI shape representation. The ESRI shape representation is available to all ESRI supported applications, as well as those applications of other vendors capable of reading this format.

In the example below, a small program segment illustrates how polygons are read from the SENSITIVE_AREAS table and passed to an applications display system. The program makes the standard ODBC function calls to establish an ODBC environment and then connects to the database. The current display window is fetched from the application's display system as polygon shape. The Spatial Extender function call *envelopesintersect* returns 1 (TRUE) only for those zone polygons that are visible within the current display window. Each polygon in the result set is fetched by the ODBC *SQLFetch* function into the fetched_binary variable and passed to the applications draw_polygon function for display. Once all the polygons have been displayed, a call to *SQLFreeStmt* closes the cursor and frees the resource associated with the SQL statement variable *hstmt*. The remainder of the program disconnects from the database, and frees the resources associated with the connection.

The functions *get_window* and *draw_polygon* are application-level function calls. In this case, the *get_window* function returns the coordinates of the display window as a polygon shape. The *draw_polygon* function would call the appropriate display drivers to draw each polygon.

```
/* Allocate memory for the ODBC environment handle henv and initialize application. */
rc = SQLAllocEnv(henv);

/* Allocate memory for a connection handle within the henv  environment. */
rc = SQLAllocConnect(*henv, handle);

/* Load the DB2 ODBC driver and connect to the data source identified by
   database, user and password.*/
SQLConnect(*handle, database, SQL_NTS, user, SQL_NTS, password, SQL_NTS,);

/* Allocate memory to the SQL statement handle hstmt. */
SQLAllocStmt(handle, &hstmt);

/* Get the display windows coordinates from the display system as a polygon
   shape. Blob_len contains the number of bytes in the shape.*/
get_window(&window, &blob_len);

/* Create the SQL expression. */
strcpy(sqlstmt, "select AsBinaryShape(zone) from SENSITIVE_AREAS where
envelopesintersect(zone,polyfromshape(cast (? as blob(1m)), coordref..srid(1)))
= 1");

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

/* Set the pcbvalue1 to the window shape */
```

```
pcbvalue1 = blob_len;

/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB, blob_len,
0, window, blob_len, &pcbvalue1);

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

/* Assign the results of the query, (the Zone polygons) to the
   fetched_binary variable. */
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);

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

/* All polygons have been displayed, so close the cursor, and free all
   memory and resources associated with the statement handle hstmt. */
SQLFreeStmt (hstmt, SQL_DROP);

/* Close the connection. */
SQLDisconnect(handle);

/* Release the connection handle and free the memory associated with it. */
SQLFreeConnect(handle);

/* Free the ODBC environment handle henv and free the memory associated
   with it. */
SQLFreeEnv(henv);
```

Not all queries include the spatial column in the result set. Sometimes the spatial column qualifies the result set. For example, the following SQL statement lists the sensitive areas with the hazardous sites where the sensitive areas are within five miles of a hazardous site. The Spatial Extender's *buffer* function generates a polygon, which is a circle representing the 5-mile radius around each hazardous location. The polygon returned by the *buffer* function is the argument of the *overlaps* function, which returns 1 (TRUE) if the zone polygon of the SENSITIVE_AREAS table overlaps the polygon generated by the *buffer* function. This is an example of a spatial join, in which the relationship of two spatial columns defines the result set of a query.

```
select sa.name "Sensitive Areas", hs.name "Hazardous Sites"
from SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
where overlaps(sa.zone, buffer(hs.location,26400)) = 1;
```

Changing the Values of a Spatial Column

The SQL update statement alters the values of a spatial column just as it does any other type of column. However, a conversion function must be called to convert the spatial columns data into a supported external format and then a reverse function must be used during the update of the data altered by the application. The following example program segment illustrates the ODBC function calls that use the Spatial Extenders functions to update a value of the spatial column. location of the HAZARDOUS_SITES table.

```
/* Allocate memory for the ODBC environment handle henv and initialize
   the application. */
rc = SQLAllocEnv(&henv);

/* Allocate memory for a connection handle within the henv environment. */
rc = SQLAllocConnect(henv, &handle);

/* Load the DB2 ODBC driver and connect to the data source identified by
   database, user and password. */
SQLConnect(handle, database, SQL_NTS, user, SQL_NTS, password, SQL_NTS,);

/* Allocate memory to the SQL statement handle hstmt. */
SQLAllocStmt(handle, &hstmt);

/* Get the display windows coordinates from the display system.
 * Create the SQL expression.
 * Set the site_id value to 102.
 *
 * (The where clause of the SQL select statement returns the record
 *  of the HAZARDOUS_SITES table whose site_id is equal to 102.)
site_id = 102;

/* Create the SQL SELECT statement that will return the location (point)
   of HAZARDOUS_SITES for site_id 102 */
sprintf(sqlstmt, "select AsBinaryShape(location) from HAZARDOUS_SITES where
site_id = %d",site_id);

/* Execute the SELECT statement directly */
rc = SQLExecDirect(hstmt, (UCHAR *)sqlstmt, SQL_NTS);

/* Assign the results of the query, (the location point) to the
   fetched_binary variable. */
rc = SQLBindCol (hstmt, 1, SQL_C_BINARY, fetched_binary, 100000, &ind_blob);

/* Fetch the point. */
rc = SQLFetch(hstmt);

/* If a single record is not returned generate an error and exit */
if(rc == SQL_NO_DATA) bailout(-1);

/* Free all memory and resources associated with the statement handle
   hstmt. */
```

```
rc = SQLFreeStmt(hstmt,SQL_DROP);

/* Call update_shape an application function presents the current
   site_id 102 location point to the user and accepts the new value. */
update_shape(&fetched_binary,&blob_len);

/* Allocate memory to the SQL statement handle hstmt. */
SQLAllocStmt(handle, &hstmt);

/* Create the SQL Update statement. */
sprintf(sqlstmt,
"update HAZARDOUS_SITES set location =
pointfromshape(cast(?as blob(1m)), coordref..srid(1)) where site_id = %d",
site_id);

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

/* Set the pcbvalue1 to the length of the binary returned by update_shape */
pcbvalue1 = blob_len;
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB,
0, 0, fetched_binary, &pcbvalue1);

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

/* Free all memory and resources associated with the statement handle
   hstmt. */
SQLFreeStmt (hstmt, SQL_DROP);

/* Close the connection. */
SQLDisconnect(handle);

/* Release the connection handle and free the memory associated with it.*/
SQLFreeConnect(handle);

/* Free the ODBC environment handle henv and free the memory associated with it. */
SQLFreeEnv(henv);
```

Chapter 4. Programming Concepts

To program applications for the Spatial Extender, it is important to understand some GIS concepts and the way in which they can be implemented in a relational database model. This chapter provides you with an understanding of the geometric data types available in the Spatial Extender and the possible relations you can make between them.

About Geometry

The Oxford American Dictionary defines the noun *geometry* as ″the branch of mathematics dealing with the properties of and relations of lines, angles, surfaces and solids.″ On August 11, 1997, the Open GIS Consortium Inc. (OGC) in its publication, *Open GIS Features for ODBC (SQL) Implementation Specification* , coined another definition for the noun. The word *geometry* was selected to represent the geometric features that, for the past millennium or more, cartographers have used to map the world.

Point geometries represent objects at distinct locations, *linestrings* represent linear characteristics, and *polygons* represent spatial extents. A very abstract definition of this new representation of the word geometry might be ″a point or aggregate of points symbolizing a feature on the ground″. This definition, however, fails to describe the rich set of properties and functionality associated with the OGIS geometry.

Geometries have been implemented within the Spatial Extender as a group of specialized data types with a unique set of properties and methods. These data types allow you to define columns that store spatial data, and then manipulate that data in much the same way you would any other data type.

Geometry Properties

Each subclass inherits the properties of the geometry superclass but they also have properties of their own. Functions that operate on the geometry data type will accept any of the subclass data types. However, some functions have been defined at the subclass level and will only accept data types of certain subclasses.

Interior, boundary, 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 it interior and exterior. The interior is the space occupied by the geometry. The subclass inherits the interior and exterior properties directly, yet the boundary property differs for each.

The **boundary** function takes a geometry and returns a geometry that represents the source geometry's boundary.

Simple or non-simple

Some subclasses of geometry (linestrings, multipoints, and multilinestrings) are either simple or non-simple. They are simple if they obey all the topological rules imposed on the subclass, and non-simple if they don't. A linestring is simple if it does not intersect its interior. A multipoint is simple if none of its elements occupy the same coordinate space. A multilinestring is simple if none of its element's interiors are intersected by its own interior.

The **issimple** predicate function takes a geometry and returns 1 (TRUE) if the geometry is simple and 0 (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 a 0 area.

The **isempty** predicate function takes a geometry and returns 1 (TRUE) if the geometry is empty and 0 (FALSE) otherwise.

Number of points

A geometry can have 0 or more points. A geometry is considered empty if it has 0 points. The point subclass is the only geometry that is restricted to 0 or 1 points, all other subclasses can have 0 or more.

Envelope

The envelope of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates. With the following exceptions, the envelopes of most geometries form a boundary rectangle:

- The envelope of a point is the point itself, because its minimum and maximum coordinates are the same.
- The envelope of a horizontal or vertical linestring is a linestring represented by the boundary (the endpoints) of the source linestring.

The **envelope** function takes a geometry and returns a bounding geometry, which represents its envelope.

Dimension

A geometry can have a dimension of 0, 1, or 2. The dimensions are listed as follows:

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

The point and multipoint subclasses have a dimension of 0. Points represent 0 dimensional features that can be modeled with a single coordinate while multipoint subclasses represent data the must be modeled with a cluster of disconnected coordinates.

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

Polygon and multipolygon subclasses have a dimension of two. 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 it 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 Spatial Extender examines the dimension of the features to determine how they should be compared.

The *dimension* function takes a 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 earths surface.

The **is3d** predicate function takes a geometry and returns 1 (TRUE) if the function has Z coordinates and 0 (FALSE) otherwise.

Measures

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

The **ismeasured** predicate takes a geometry and returns a 1 (TRUE) if it contains measures and 0 (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 a complete discussion on the creation and maintenance of the SPATIAL_REFERENCES table refer to “Meta Tables and Views” on page 19.

The **srid** function takes a geometry and returns its spatial reference identifier as an integer.

Instantiable Subclasses

The geometry data type is not instantiable but instead must store its instantiable subclasses. The subclasses are divided into two categories: the base geometry subclasses, and the homogeneous collection subclasses. The base geometries include point, linestrings and polygons while the homogeneous collections include multipoint, multilinestring, and multipolygon. As the names imply, the homogeneous collections are collections of base geometries. In addition to sharing base geometry properties, homogeneous collections have some of their own properties as well.

geometrytype
takes a geometry and returns the instantiable subclass in the form of a character string.

numgeometries
takes a homogeneous collection and returns the number of base geometry elements it contains.

geometryn
takes a homogeneous collection and an index and returns the nth base geometry.

Point

A point is a zero-dimensional geometry that occupies a single location in coordinate space. A point has a single Xi coordinate value. A point is always simple. Points have a NULL boundary. Points are often used to define features such as oil wells, landmarks, and elevations.

Functions that operate solely on the point data type:

x	returns a point data types X coordinate value as a double precision number.
y	returns a point data types Y coordinate values as a double precision number.
z	returns a point data types Z coordinate values as a double precision number.

m returns a point data types M coordinate values as a double precision number.

Linestring

A linestring is a one-dimensional object stored as a sequence of points defining a linear interpolated path. The linestring is simple if it does not intersect its interior. The endpoints (the boundary) of a closed linestring occupy the same point in space. A linestring is a ring if it is both closed and simple. As well as the other properties inherited from the superclass geometry, linestrings have length. Linestrings are often used to define linear features such as roads, rivers and power lines.

The endpoints normally form the boundary of a linestring unless the linestring is closed in which case the boundary is NULL. The interior of a linestring is the connected path that lies between the endpoints, unless it is closed in which case the interior is continuous.

Functions that operate on linestrings:

startpoint
Takes a linestring and returns its first point.

endpoint
Takes a linestring and returns its last point.

pointn Takes a linestring and an index to nth point and returns that point.

length Takes a linestring and returns its length as a double precision number.

numpoints
Takes a linestring and returns the number of points in its sequence as an integer.

isring Takes a linestring and returns 1 (TRUE) if the linestring is a ring and 0 (FALSE) otherwise.

isclosed
Takes a linestring and returns 1 (TRUE) if the linestring is closed and 0 (FALSE) otherwise.

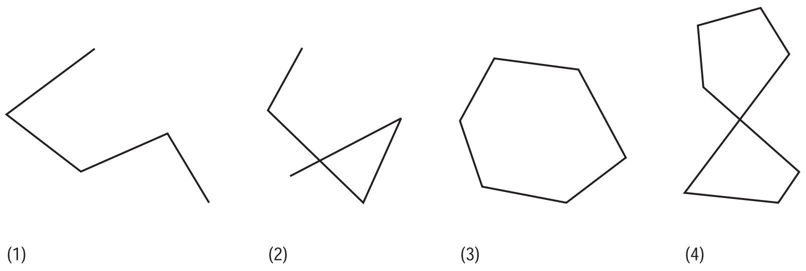


Figure 9. Linestring objects.
1. *A simple non-closed linestring.*
2. *A non-sismple non-closed linestring.*
3. *A closed simple linestring and therefore a ring.*
4. *A closed non-simple linestring. It is not a ring.*

Polygon

A polygon is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. Polygons by definition are always simple. Most often they define parcels of land, water bodies and other features that have a spatial extent.

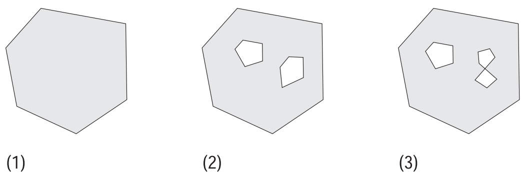


Figure 10. Polygons.
1. *A polygon whose boundary is defined by an exterior ring.*
2. *A polygon whose boundary is defined by an exterior ring and two interior rings. The area inside the interior rings is part of the polygons exterior.*
3. *A legal polygon because the rings intersect at a single tangent point.*

The exterior and any interior rings define the boundary of a polygon, and the space enclosed between the rings define the polygons interior. The rings of a polygon can intersect at a tangent point but never cross. In addition to the other properties inherited from the superclass geometry, polygons have area.

Functions that operate on polygons:

area Takes a polygon and returns its area as a double precision number.

exteriorring
Takes a polygon and returns its exterior ring as a linestring.

numinteriorrings
Takes a polygon and returns the number of interior rings that it contains.

interiorringn
Takes a polygon and an index and returns the nth interior ring as a linestring.

centroid
Takes a polygon and returns a point that is the center of the multipolygon's extent.

pointonsurface
Takes a polygon and returns a point that is guaranteed to be on the surface of the polygon.

Multipoint

A multipoint is a collection of points and just like its elements it has a dimension of 0. A multipoint is simple if none of its elements occupy the same coordinate space. The boundary of a multipoint is NULL. Multipoints define aerial broadcast patterns, and incidents of an epidemic outbreak.

Multilinestring

A multilinestring is a collection of linestrings. Multilinestrings are simple if they only intersect at the endpoints of the linestring elements. Multilinestrings are non-simple if the interiors of the linestring elements intersect.

The boundary of a multilinestring is the non-intersected endpoints of the linestring elements. The multilinestring is closed if all of its linestring elements are closed. The boundary of a multilinestring is NULL if all of the endpoints of all of the elements are intersected. In addition to the other properties inherited from the superclass geometry, multilinestrings have length as well. Multilinestrings are used to define streams or road networks.

Functions that operate on multilinestrings:

length Takes a multilinestring and returns the cumulative length of all its linestring elements as a double precision number.

isclosed
Takes a multilinestring and returns 1 (TRUE) if the multilinestring is closed and 0 (FALSE) otherwise.

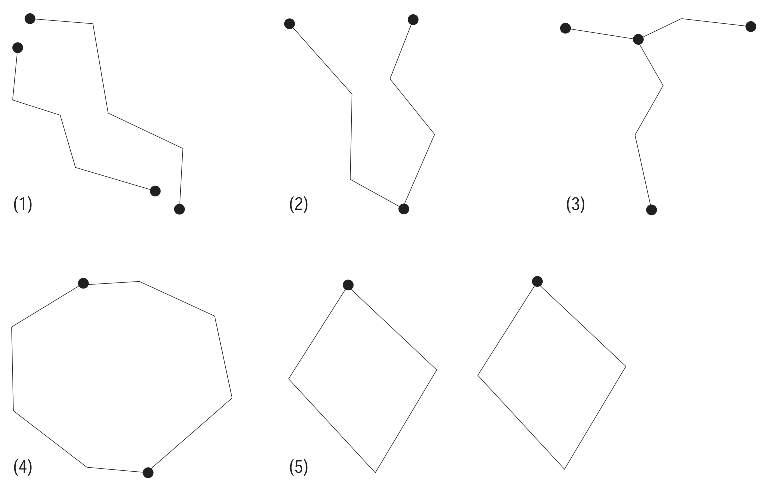


Figure 11. Multilinestrings.

1. *A simple multilinestring whose boundary is defined by the four endpoints of its two linestring elements.*
2. *A simple multilinestring because only the endpoints of the linestring elements intersect. The boundary is defined by the two non-intersecting endpoints.*
3. *A non-simple linestring because the interior of one of its linestring elements is intersected. The boundary of this multilinestring is defined by the four endpoints, including the intersecting point.*
4. *A simple non-closed multilinestring. It is not closed because its element linestrings are not closed. It is simple because none of the interiors of any of the element linestrings are intersected.*
5. *A simple closed multilinestring. It is closed because all of its elements are closed. It is simple because none of its elements are intersected at the interiors.*

Multipolygon

The boundary of a multipolygon is the cumulative length of its element's exterior and interior rings. The interior of a multipolygon is defined as the cumulative interiors of its element polygons. The boundary of a multipolygons elements can only intersect at a tangent point. In addition to the other properties inherited from the superclass geometry, multipolygons have area. Multipolygons define features such as a forest stratum or a non-contiguous parcel of land such as a pacific island chain.

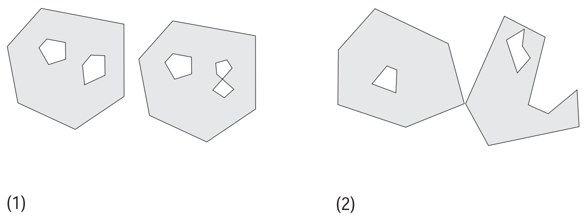


Figure 12. Multipolygons.

1. *A multipolygon with two polygon elements. The boundary is defined by the two exterior rings and the three interior rings.*
2. *A multipolygon with two polygon elements. The boundary is defined by the two exterior rings and the two interior rings. The two polygon elements intersect at a tangent point.*

Functions that operate on multipolygons:

area Takes a multipolygon and returns the cumulative area of its polygon elements as a double precision number.

centroid
Takes a multipolygon and returns a point that is its geometric-weighted center.

Geometry Data Exchange

The Spatial Extender supports three GIS data exchange formats:

- Well-known text representation
- Well-known binary representation
- ESRI binary shape representation

Well-Known Text Representation

The Spatial Extender has several functions that generate geometries from text descriptions.

geometryfromtextcreates
A geometry from a text representation of any geometry type.

pointfromtext
Creates a point from a point text representation.

linefromtext
Creates a linestring from a linestring text representation.

polyfromtext
Creates a polygon from a polygon text representation.

mpointfromtext
Creates a multipoint from a multipoint representation.

mlinefromtext
Creates a multilinestring from a multilinestring representation.

mpolyfromtext
Creates a multipolygon from a multipolygon representation.

The text representation is an ASCII string. It permits geometry to be exchanged in ASCII text form. These functions do not require the definition of any special program structures to map a binary representation. So, they can be used in either a 3GL or 4GL program.

The **astext** function converts an existing geometry value into text representation.

See "Appendix B. The OGIS Well-Known Text Representation" on page 187 for a detailed description of the text representation.

Well-Known Binary Representation

The Spatial Extender has several functions that generate geometries from well-known binary (WKB) representations.

geometryfromwkb
Creates a geometry from a WKB representation of any geometry type.

pointfromwkb
Creates a point from a point WKB representation.

linefromwkb
Creates a linestring from a linestring WKB representation.

polyfromwkb
Creates a polygon from a polygon WKB representation.

mpointfromwkb
Creates a multipoint from a multipoint WKB representation.

mlinefromwkb
Creates a multilinestring from a multilinestring WKB representation.

mpolyfromwkb
Creates a multipolygon from a multipolygon WKB representation.

The well-known binary representation is a contiguous stream of bytes. It permits geometry to be exchanged between an ODBC client and an SQL database in binary form. These geometry functions require the definition of a C structures to map the binary representation. So, they are intended for use within a 3GL program, and are not suited to a 4GL environment.

The **asbinary** function converts an existing geometry value into well-known binary representation.

See “Appendix C. The OGIS Well-Known Binary Representation” on page 193 for a detailed description of WKB.

ESRI Shape Representation

The Spatial Extender has several functions that generate geometries from an ESRI shape representation. Like the text and WKB representations, the ESRI shape representation supports two-dimensional representations. However, it also supports Z coordinates and measures.

geometryfromshape
Creates a geometry from a shape of any geometry type.

pointfromshape
Creates a point from a point shape.

linefromshape
Creates a linestring from a polyline shape.

polyfromshape
Creates a polygon from a polyline shape.

mpointfromshape
Creates a multipoint from a multipoint shape.

mlinefromshape
Creates a multilinestring from a multipart polyline shape.

mpolyfromshape
Creates a multipolygon from a multipart polygon shape.

The general syntax of these functions is the same. The first argument is the shape representation that is entered as a BLOB data type. The second argument is the spatial reference identifier that will be assigned to the geometry. For example, the geometryfromshape function has the following syntax:

```
geometryfromshape(shapegeometry, SRID)
```

To map the binary representation, these shape functions require the definition of a C structures. So, they are intended for use within a 3GL program, and are not suited to a 4GL environment.

The **asbinaryshape** function converts a geometry value into an ESRI shape representation. See “Appendix D. The ESRI Shape Representations” on page 197 for a detailed description.

Geometry Relations

As one of its primary functions, a geographic information system determines the spatial relationships that exist between features. The distance separating a hazardous waste disposal site and a hospital, school or housing development is an example of a spatial relationship.

The Spatial Extender employs predicates, which are boolean functions used to determine if a specific relationship exists between a pair of geometries.

Other functions return a value as a result of a spatial relationship. For example, the result returned by the *distance* function, the space separating two geometries, is a double precision number. And a function such as*intersection* returns a geometry, resulting from the intersection of two geometries.

Predicates

Predicates return 1 (TRUE) if a comparison meets the function's criteria, or 0 (FALSE) if the comparison fails. Predicates that test for a spatial relationship compare pairs of geometries that can be a different type or dimension.

Predicates compare the X and Y coordinates of the submitted geometries. The Z coordinates and the measure (if they exist) are ignored. This allows geometries that have Z coordinates and/or measure to be compared with those that do not.

The *Dimensionally Extended 9 Intersection Model (DE-9IM*)[1]is a mathematical approach that defines the pair-wise spatial relationship between geometries of different types and dimensions. This model expresses spatial relationships between all types of geometries 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) produces a 3 by 3 matrix. Each intersection can result in geometries of different dimensions. For example, the intersection of the boundaries of two polygons consists 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 or dim(null), which is returned when no intersection was found.

1. The DE-91M was developed by Clementini and Felice, who dimensionally extended the 9 Intersection Model of Egenhofer and Herring. DE-91M is collaboration of three authors, Clementini, Eliseo, Di Felice, P., van Osstrom, P.. They published the model in ″A Small Set of Formal Topological Relationships Suitable for End-User Interaction,″ D. Abel and B.C. Ooi (Ed.), *Advances in Spatial Database—Third International Symposium. SSD '93*. LNCS 692. Pp. 277-295. The 9 Intersection model by Springer-Verlag Singapore (1993) Egenhofer M.J. and Herring, J., was published in ″Categorizing binary topological relationships between regions, lines, and points in geographic databases,″ *Tech. Report., Department of Surveying Engineering*, University of Maine, Orono, ME 1991.

	Interior	Boundary	Exterior
Interior	dim(I(a)∩ I(b))	dim(I(a) ∩ B(b))	dim(I(a) ∩ E(b))
Boundary	dim(B(a) ∩ I(b))	dim(B(a) ∩ B(b))	dim(B(a) ∩ E(b))
Exterior	dim(E(a) ∩ I(b))	dim(E(a) ∩ B(b))	dim(E(a) ∩ E(b))

The results of the spatial relationship predicates can be understood or verified by comparing the results of the predicate 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, 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.

Each predicate has at least one pattern matrix, but some require more than one to describe the relationships of various geometry type combinations.

Table 4. Matrix for within. The pattern matrix of the within predicate for geometry combinations.

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

The within predicate returns true when the interiors of both geometries intersect and when the interior and boundary of *a* does not intersect the exterior of *b.* All other conditions do not matter.

Equals

Equals returns 1 (TRUE) if two geometries of the same type have identical X,Y coordinate values.

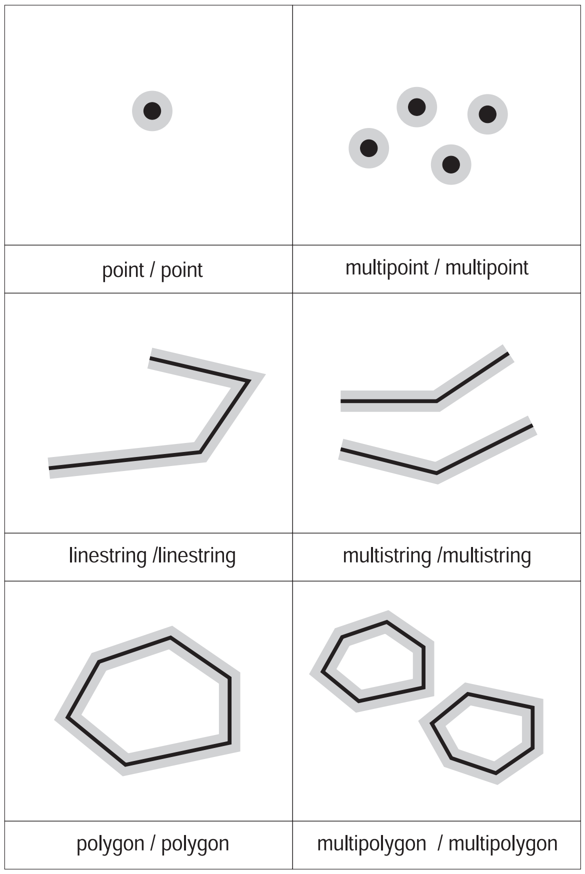


Figure 13. Equals. Geometries are equal if they have matching X,Y coordinates.

Table 5. Matrix for equality. The DE-9IM pattern matrix for equality ensures that the interiors intersect and that no part interior or boundary of either geometry intersects the exterior of the other.

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

Disjoint

Disjoint returns 1 (TRUE) if the intersection of the two geometries is an empty set.

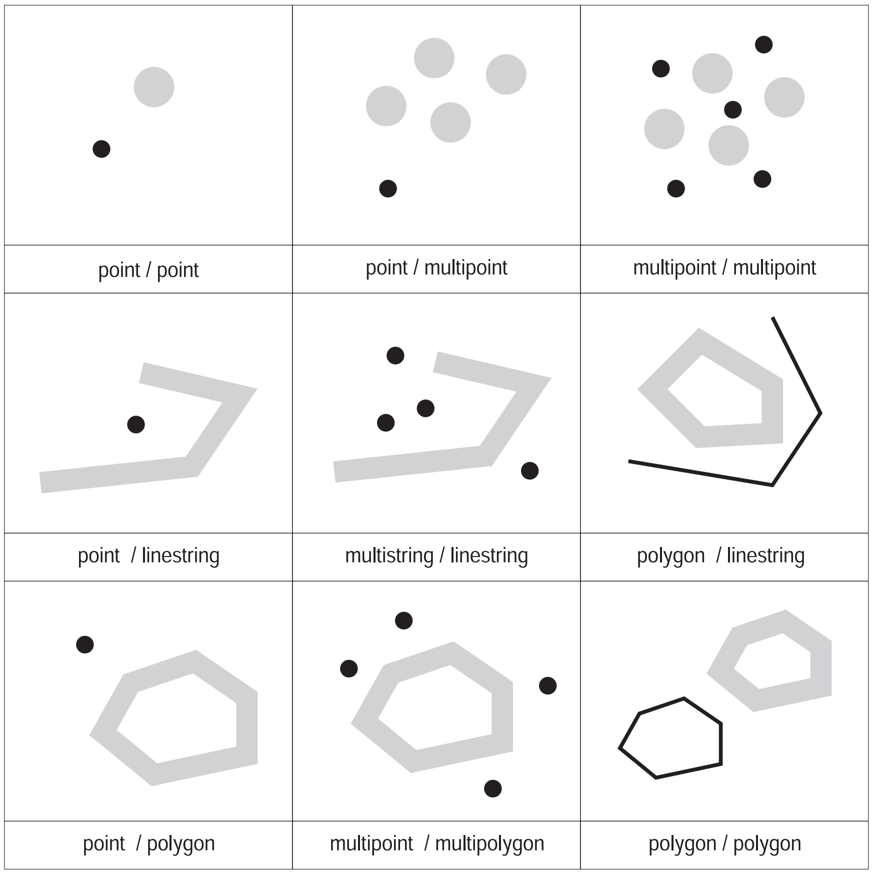


Figure 14. Disjoint. Geometries are disjoint if they do not intersect one another in any way.

Table 6. Matrix for disjoint. The disjoint predicate's pattern matrix simple states that neither the interiors nor the boundaries of either geometry intersect.

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

Intersects

Intersects returns 1 (TRUE) if the intersection does not result in an empty set. Intersects returns the exact opposite result of disjoint.

The intersect predicate will return TRUE if the conditions of any of the following pattern matrices returns TRUE.

Table 7. Matrix for intersects (1). The intersects predicate returns TRUE if the interiors of both geometries intersect.

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

Table 8. Matrix for intersects (2). The intersects predicate returns TRUE if the boundary of the first geometry intersects the boundary of the second geometry.

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

Table 9. Matrix for intersects (3). The intersects predicate returns TRUE if the boundary of the first geometry intersects the interior of the second.

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

Table 10. Matrix for intersects (4). The intersects predicate returns TRUE if the boundaries of either geometry intersect.

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

Touch

Touch returns 1 (TRUE) if none of the points common to both geometries intersect the interiors of both geometries. At least one geometry must be a linestring, polygon, multilinestring or multipolygon.

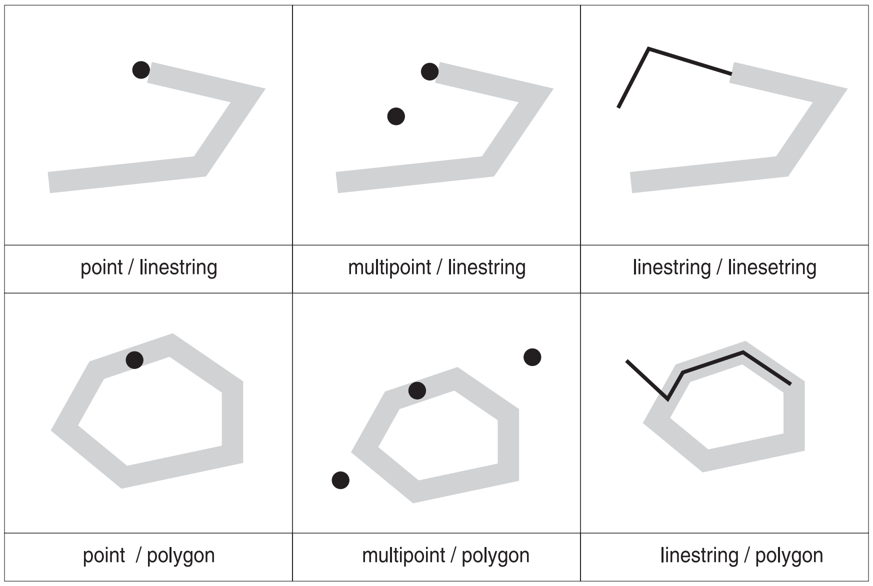


Figure 15. Touch

The pattern matrices show us that the touch predicate returns TRUE when the interiors of the geometry do not intersect, and the boundary of either geometry intersects the other's interior or its boundary.

Table 11. Matrix for touch (1)

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

Table 12. Matrix for touch (2)

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

Table 13. Matrix for touch (3)

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

Table 13. Matrix for touch (3) (continued)

Boundary	*	T	*
Exterior	*	*	*

Overlap

Overlap compares two geometries of the same dimension. It returns 1 (TRUE) if their intersection set results in a geometry different from both, but that has the same dimension.

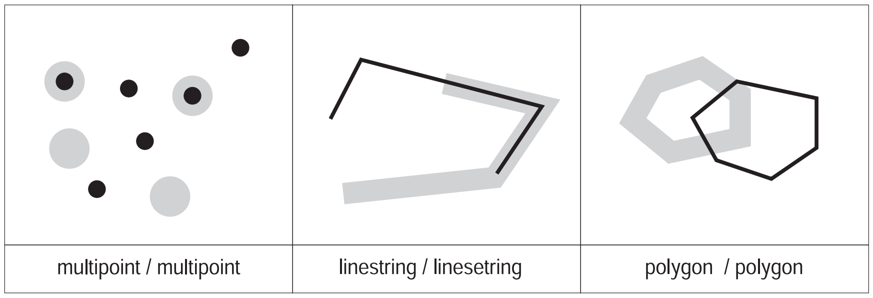


Figure 16. Overlap

This pattern matrix applies to polygon/polygon, multipoint/multipoint and multipolygon/multipolygon overlays. For these combinations the overlay predicate returns TRUE if the interior of both geometries intersect the others interior and exterior.

Table 14. Matrix for overlap (1)

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

This pattern matrix applies to linestring/linestring and multilinestring/multilinestring overlays. In this case the intersection of the geometries must result in a geometry that has a dimension of 1 (another linestring). If the dimension of the intersection of the interiors had been 1 the overlay predicate would return FALSE, however the cross predicate would have returned TRUE.

Table 15. Matrix for overlap (2)

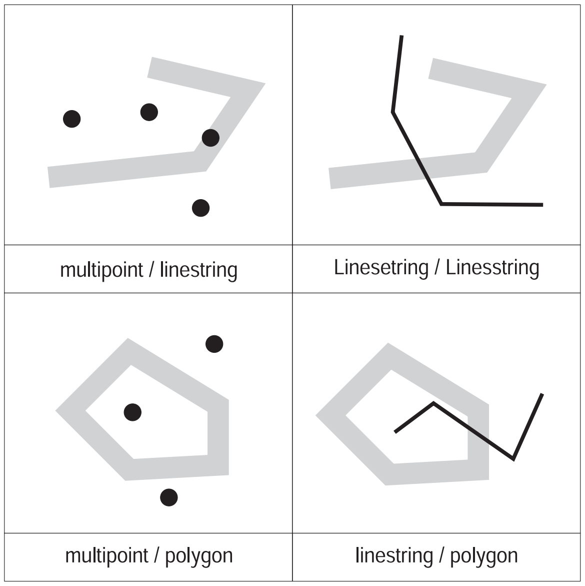
		b		
		Interior	**Boundary**	**Exterior**
a	**Interior**	1	*	T
	Boundary	*	*	*

Table 15. Matrix for overlap (2) (continued)

Exterior	T	*	*

Cross

Cross returns 1 (TRUE) if the intersection results in a geometry whose dimension is one less than the maximum dimension of the two source geometries and the intersection set is interior to both source geometries. Cross returns t (TRUE) for only a multipoint/polygon, multipoint/linestring, linestring/linestring, linestring/polygon, and linestring/multipolygon comparisons.



This pattern matrix applies to multipoint/linestring, multipoint/multilinestring, multipoint/polygon, multipoint/multipolygon, linestring/polygon, linestring/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*)

Table 16. Matrix for cross (1)

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

This pattern matrix applies to the linestring/linestring, linestring/multilinestring and multilinestring/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 cross predicate would return FALSE but the overlay predicate would return TRUE.

Table 17. Matrix for cross (2)

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

Within

Within returns 1 (TRUE) if the first geometry is completely within the second geometry. Within returns the exact opposite result of contains.

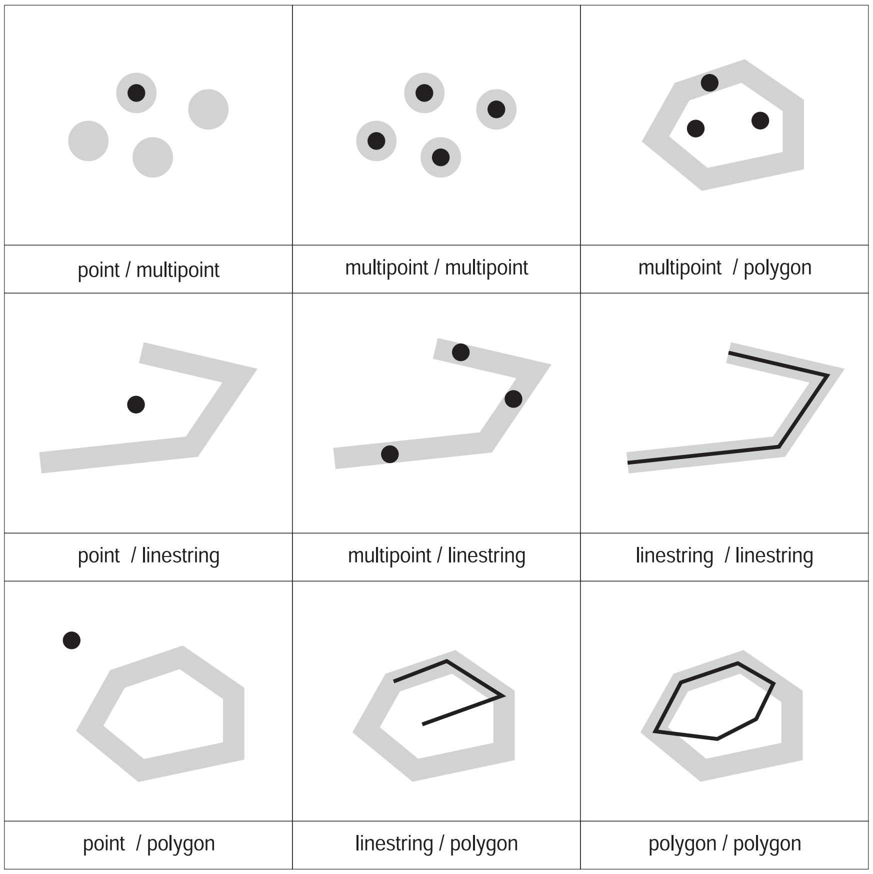


Figure 17. Within

The within predicate 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*).

Table 18. Matrix for within

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

Contains

Contains returns 1 (TRUE) if the second geometry is completely contained by the first geometry. The contains predicate returns the exact opposite result of the within predicate.

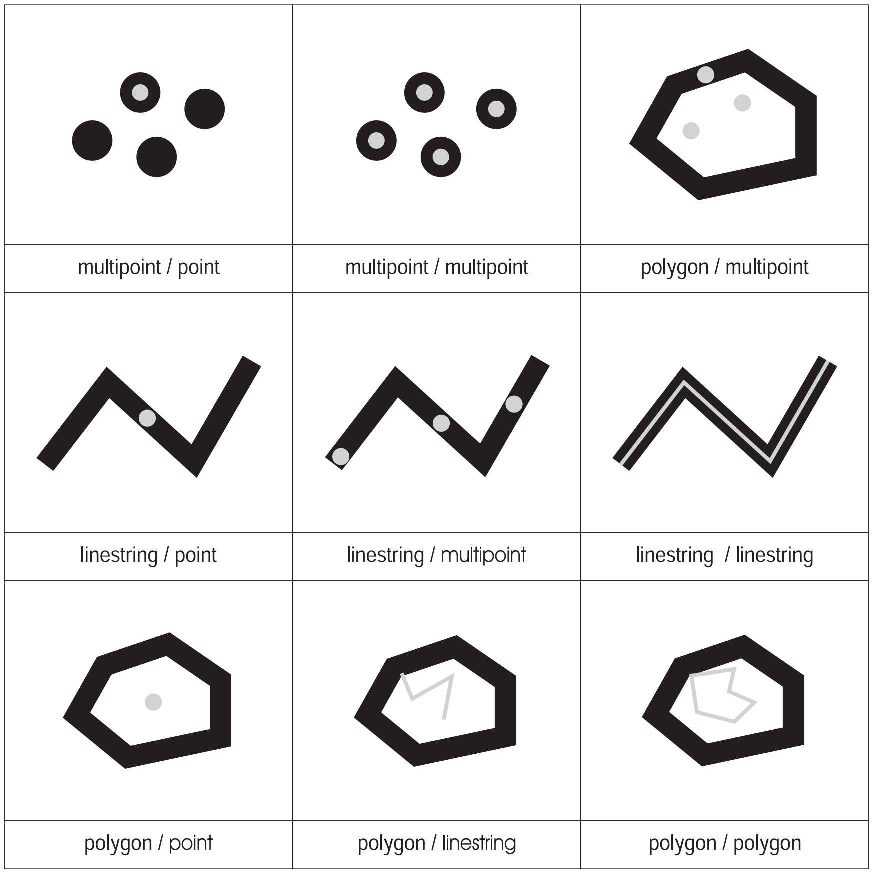


Figure 18. Contains

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

Table 19. Matrix for contains

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

Table 19. Matrix for contains *(continued)*

Boundary	*	*	*
Exterior	F	F	*

Minimum Distance

The distance function reports the minimum distance separating two disjoint features. If the features are not disjoint, the function will report a 0 minimum distance.

The minimum distance separating disjoint features could, for example, represent the shortest distance an aircraft must travel between two locations.

Intersection of Geometries

The intersection function returns the intersection set of two geometries. The intersection set is always returned as a collection that is the minimum dimension of the source geometries. For example, for a linestring that intersects a polygon, the intersection function returns a multilinestring comprised of that portion of the linestring common to the interior and boundary of the polygon. The multilinestring contains more than one linestring if the source linestring intersects the polygon with two or more discontinuous segments. If the geometries do not intersect or if the intersection results in a dimension less that both of the source geometries, an empty geometry is returned.

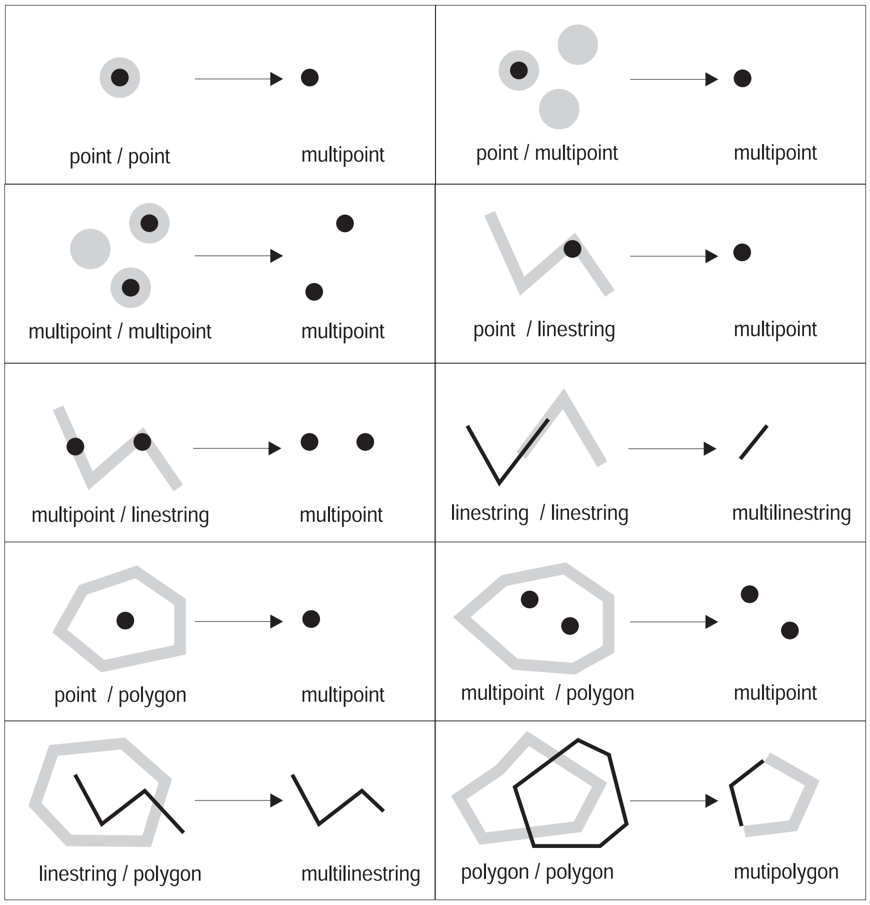


Figure 19. Intersection. Examples of the intersection function.

Difference of geometries

The difference function returns the portion of the primary geometry that is not intersected by the secondary geometry. This is the logical AND NOT of space. The difference function only operates on geometries of like dimension and returns a collection that has the same dimension as the source geometries. In the event that the source geometries are equal, an empty geometry is returned.

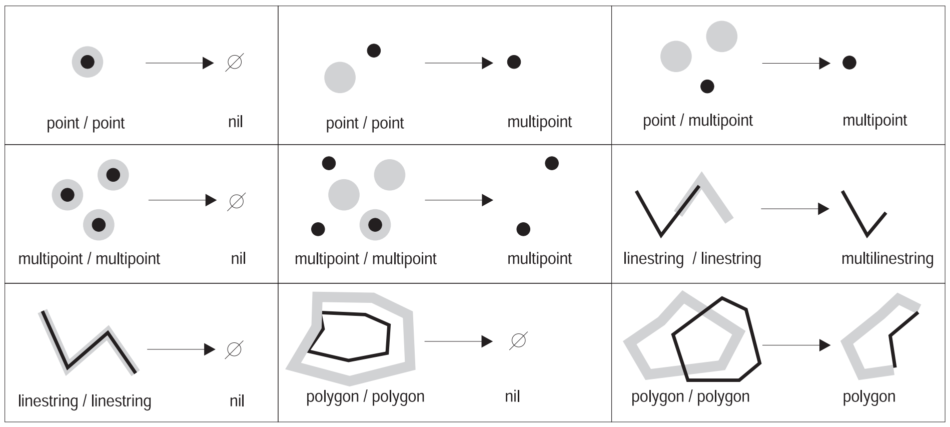


Figure 20. Difference

Union of geometries

The union function returns the union set of two geometries. This is the logical OR of space. The source geometries must be of like dimension. Union always returns the result as a collection.

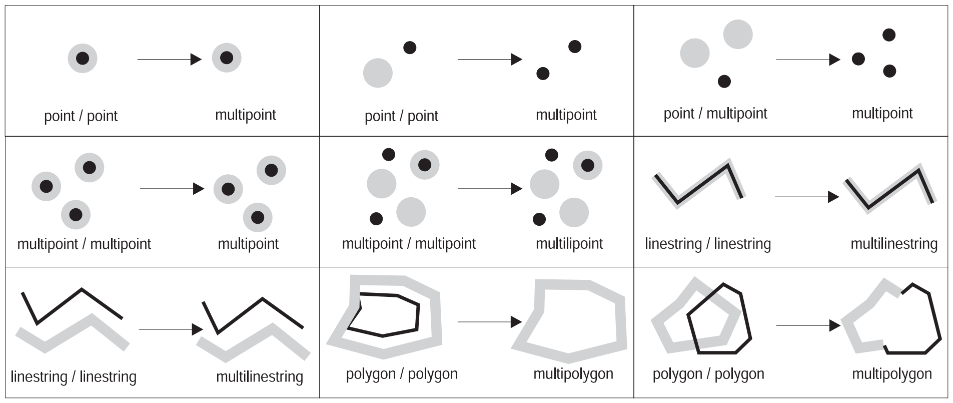


Figure 21. Union

Symmetric Difference of Geometries

The symmetricdiff function returns the symmetric difference of two geometries. This is the logical XOR of space. The source geometries must be of like dimension. If the geometries are equal, then the symmetric difference function returns an empty geometry; otherwise, the function returns the result as a collection.

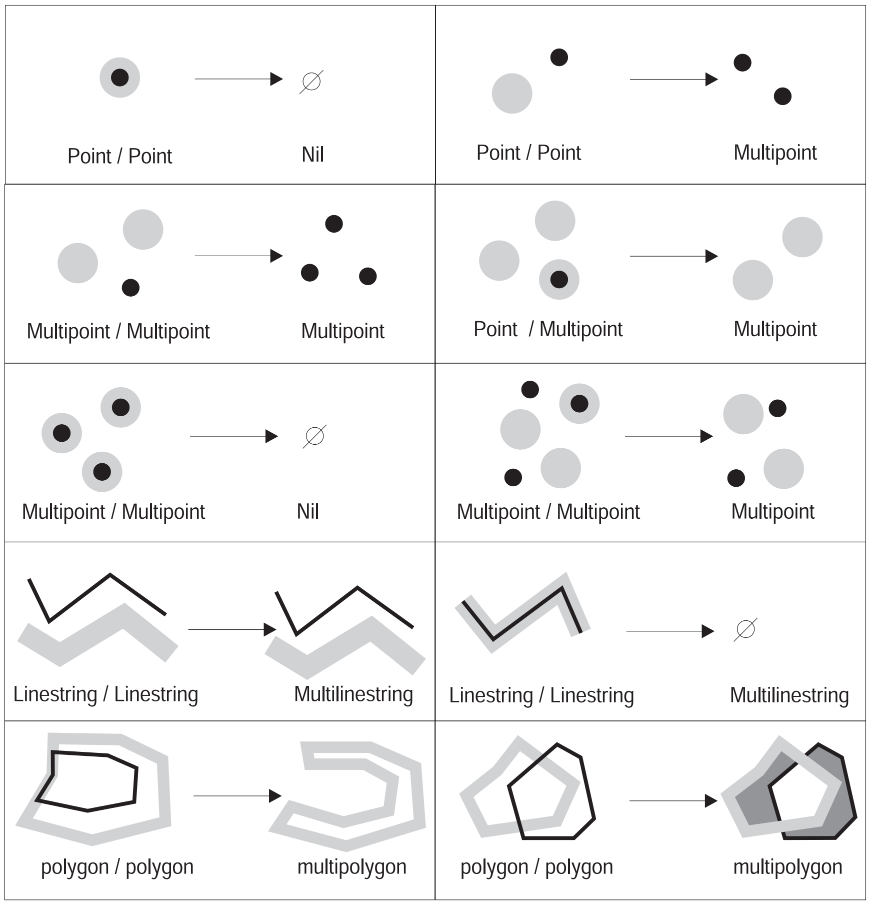


Figure 22. Symmetric difference

Geometry Transforms

The Spatial Extender provides four additional transformation functions that generate new geometry from existing geometry and a formula.

Buffering Geometries

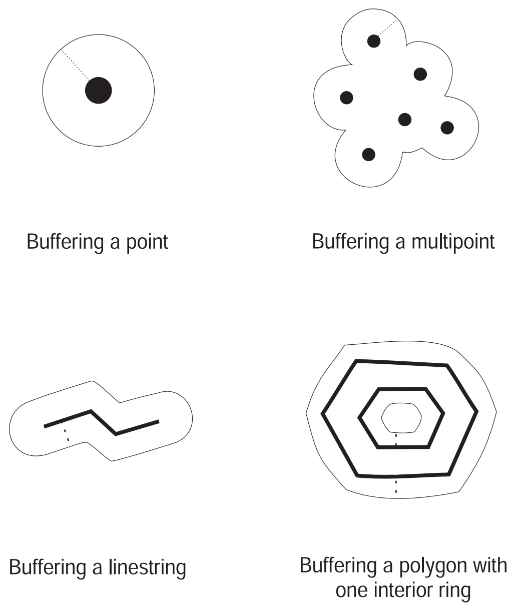


Figure 23. Buffer

The *buffer* function generates a geometry by encircling a geometry at a specified distance. A polygon results when a primary geometry is buffered or whenever the elements of a collection are close enough such that all of the buffer polygons all overlap. However, when there is enough separation between the elements of a buffered collection individual buffer polygons will result in which case the *buffer* function returns a multipolygon.

The buffer function accepts both positive and negative distance, however, only geometries with a dimension of two (polygons and multipolygons) apply a negative buffer. The absolute value of the buffer distance is used whenever the dimension of the source geometry is less than 2 (all geometries that are not polygon or 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 polygon or multipolygon toward the center when distance is negative. For interior rings of a polygon or 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 polygons that overlap. Negative distances greater than one half the maximum interior width of a polygon result in an empty geometry.

Locatealong

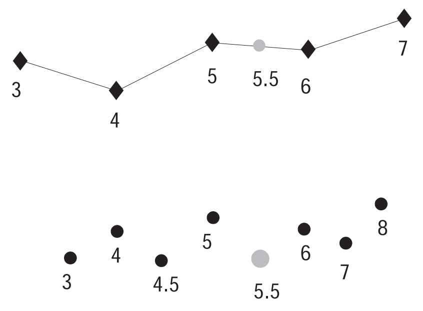


Figure 24. Locatealong

For geometries that have measures, the location of a particular measure can be found with the *locatealong* function. *Locatealong* returns the location as a multipoint. If the source geometrys dimension is 0 (i.e. point and multipoint), an exact match is required and those points having a matching measure value are returned as a multipoint. However, for source geometries whose dimension is greater than 0, the location is interpolated. For example, if the measure value entered is 5.5 and the measures on vertices of a linestring are a respective 3, 4, 5, 6, and 7, the interpolated point that falls exactly halfway between the vertices with measure values 5 and 6 is returned.

Locatebetween

The *locatebetween* function returns either the set of paths or locations that lie between two measure values from a source geometry that has measures. If the source geometrys dimension is 0, *locatebetween* returns a multipoint containing all points whose measures lie between the two source measures. For source geometries whose dimension is greater than 0, *locatebetween* returns a multilinestring if a path can be interpolated; otherwise *locatebetween* returns a multipoint containing the point locations. An empty point is returned whenever *locatebetween* cannot interpolate a path or find a location between the measures. *Locatebetween* performs an inclusive search of the geometries; therefore the geometries measures must be greater than or equal to the from measure and less than or equal to the to measure.

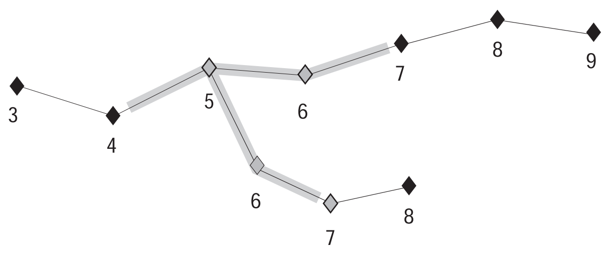


Figure 25. Locatebetween

Convexhull

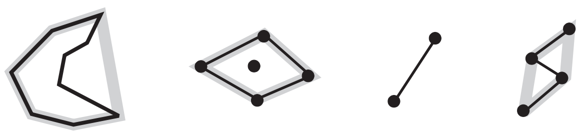


Figure 26. Convexhull

The *convexhull* function returns the convex hull polygon of any geometry that has at least three vertices forming a convex. If vertices of the geometry do not form a convex, convexhull returns a null. *Convexhull* is often the first step in tesselation used to create a TIN network from a set of points.

Chapter 5. SQL Reference

area

Area takes a polygon or multipolygon and returns its area.

Syntax

area(pl1 polygon)
area(mpl1 multipolygon)

Return type

Double precision

Examples

The City Engineer needs a list of building areas. To obtain the list a GIS technician selects the building id and area of each building's 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   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, area (footprint) "Area"
  from BUILDINGFOOTPRINTS;

building_id   Area
------------  ------------------------
         506    +1.40768000000000E+003
        1208    +2.55759000000000E+003
         543    +1.80786000000000E+003
         178    +2.08671000000000E+003
         .
         .
         .
```

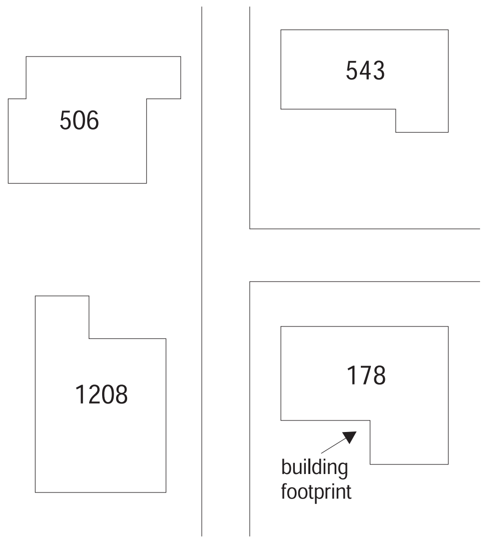


Figure 27. Using area to find a building footprint. Four of the building footprints labeled with their building id numbers are displayed along side their adjacent street.

asbinary

Asbinary takes a geometry object and returns its well-known binary representation.

Syntax

asbinary(g1 geometry)

Return type

blob(1m)

Examples

Below the code fragment illustrates how the asbinary function converts the footprint multipolygons of the BUILDINGFOOTPRINTS table into WKB multipolygons. These multipolygons are passed to the application's draw_polygon function for display.

```
/* Create the SQL expression. */
strcpy(sqlstmt, "select asbinary (footprint) from BUILDINGFOOTPRINTS
where envelopesintersect(footprint,polyfromwkb(cast(? as blob(1m)),
coordref()..srid(1)))");

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

/* Set the pcbvalue1 length of the shape. */
```

```
pcbvalue1 = blob_len;

/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB, blob_len,
0, shape, blob_len, &pcbvalue1);

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

/* Assign the results of the query, (the Zone polygons) to the
 * fetched_binary variable.
 */
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);

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

asbinaryshape

Asbinaryshape takes a geometry object and returns a blob.

Syntax

asbinaryshape(g1 geometry)

Return type

blob(1m)

Examples

Below the code fragment illustrates how the asbinaryshape function converts the zone polygons of the SENSITIVE_AREAS table into shape polygons. These shape polygons are passed to the applications draw_polygon function for display.

```
/* Create the SQL expression. */
strcpy(sqlstmt, "select asbinaryshape (zone) from SENSITIVE_AREAS
where envelopesintersect(zone,polyfromshape(cast(? as blob(1m)),
coordref..srid(1)))");

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

/* Set the pcbvalue1 length of the shape. */
pcbvalue1 = blob_len;

/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB, blob_len,
0, shape, blob_len, &pcbvalue1);

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

/* Assign the results of the query, (the Zone polygons) to the
   fetched_binary variable. */
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);

/* Fetch each polygon within the display window and display it. */

while(SQL_SUCCESS == (rc = SQLFetch(hstmt))
  draw_polygon(fetched_binary);
```

astext

Astext takes a geometry object and returns its well-known text representation.

Syntax

astext(g1 geometry)

Return type

varchar(4000)

Examples

The astext function converts the HAZARDOUS_SITES location point into its text description.

```
create table HAZARDOUS_SITES (site_id   integer,
                              name      varchar(40),
                              location  point);

insert into HAZARDOUS_SITES
values (102,
        W. H. Kleenare Chemical Repository,
        pointfromtext(point (1020.12 324.02),coordref()..srid(1)));

select site_id, name, cast(astext (location) as varchar(40)) "Location" from HAZARDOUS_SITES;

SITE_ID Name                                          Location
------- --------------------------------------------- ----------------------------------------
    102 W. H. Kleenare Chemical Repository            POINT ( 1020.12000000 32402000000)
```

boundary

Boundary takes a geometry object and returns its combined boundary as a geometry object

Syntax

boundary(g1 geometry)

Return type

Geometry

Examples

In this example the BOUNDARY_TEST table is created with two columns, geotype defined as a varchar and g1 defined as the superclass geometry. The insert statements that follow insert each one of the subclass geometries. The boundary function retrieves the boundary of each subclass that is 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 multilinestring return a multipoint boundary, dimension 0. A polygon or multipolygon always return a multilinestring boundary, dimension 1.

```
create table BOUNDARY_TEST (geotype varchar(20), g1 geometry)

insert into BOUNDARY_TEST
values('Point',
       pointfromtext('point (10.02 20.01)',
                     coordref()..srid(1)))

insert into BOUNDARY_TEST
values('Linestring',
       linefromtext('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',
                    coordref()..srid(1)))

insert into BOUNDARY_TEST
values('Polygon',
       polyfromtext('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                               19.15 33.94, 10.02 20.01))',
                    coordref()..srid(1)))

insert into BOUNDARY_TEST
values('Multipoint',
       mpointfromtext('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
                      coordref()..srid(1)))

insert into BOUNDARY_TEST
values('Multilinestring',
       mlinefromtext('multilinestring ((10.02 20.01,10.32 23.98,11.92 25.64),
                                       (9.55 23.75,15.36 30.11))',
                     coordref()..srid(1)))

insert into BOUNDARY_TEST
values('Multipolygon',
       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)))',
                     coordref()..srid(1)))

select geotype,
       cast(astext(boundary (g1)) as varchar(280)) "The boundary"
from BOUNDARY_TEST

GEOTYPE              The boundary
-------------------- ----------------------------------------------------------
Point                POINT EMPTY
Linestring           MULTIPOINT ( 10.02000000 20.01000000, 11.92000000
 25.64000000)
Polygon              MULTILINESTRING (( 10.02000000 20.01000000, 19.15000000
 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000
 20.01000000))
Multipoint           POINT EMPTY
Multilinestring      MULTIPOINT ( 9.55000000 23.75000000, 10.02000000
 20.01000000, 11.92000000 25.64000000, 15.36000000 30.11000000)
Multipolygon         MULTILINESTRING (( 51.71000000 21.73000000, 73.36000000
 27.04000000, 71.52000000 32.87000000, 52.43000000 31.90000000, 51.71000000
 21.73000000),( 10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000
 34.15000000, 11.92000000 35.64000000, 10.02000000 20.01000000))

  6 record(s) selected.
```

buffer

Buffer takes a geometry object and distance and returns the geometry object that surrounds the source object.

Syntax

buffer(g1 geometry , distance double_precision)

Return type

Geometry

Examples

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 that follows. The zone column is defined as a polygon, which is stored as the outline of each of the sensitive areas.

```
create table SENSITIVE_AREAS (id    integer,
                              name  varchar(128),
                              size  float,
                              type  varchar(10),
                              zone  polygon);
```

The hazardous sites are stored in the HAZARDOUS_SITES table that is 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(128),
                              location  point);
```

The SENSITIVE_AREAS and HAZARDOUS_SITES table are joined by the overlap function. The function returns 1 (TRUE) for all SENSITIVE_AREAS rows whose zone polygons overlap the buffered 5-mile radius of the HAZARDOUS_SITES location point.

```
select sa.name "Sensitive Areas", hs.name "Hazardous Sites"
from SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
where overlap(sa.zone, buffer (hs.location,(5 * 5280))) = 1;
```

In Figure 28 on page 76, some of the sensitive areas in this administrative district lie within the 5-mile buffer of the hazardous site locations. It is clear that both the 5-mile buffer intersects the hospital and one of them intersects the school. However the nursing home lies safely outside both radii.

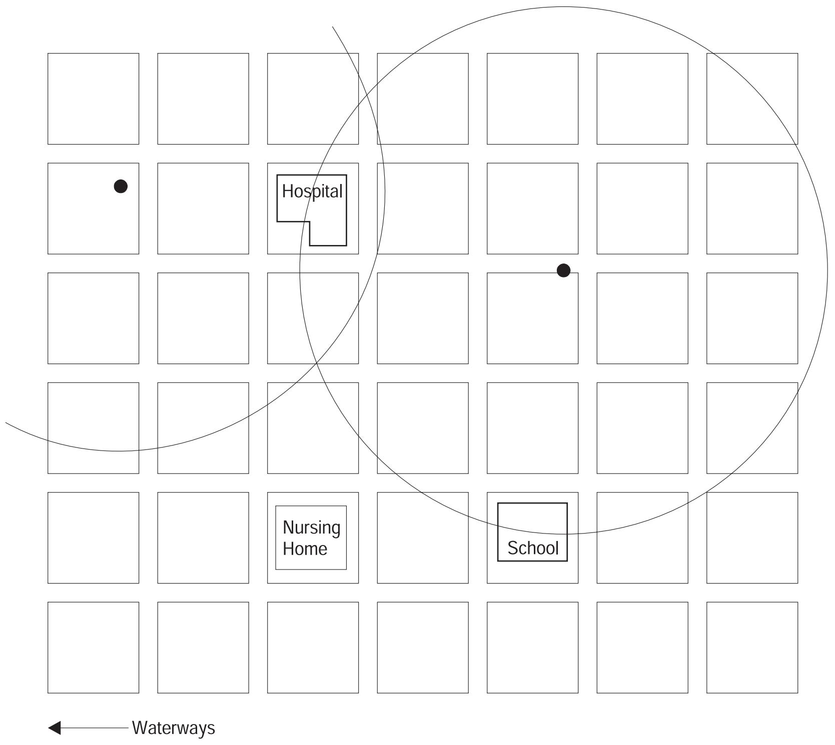


Figure 28. A buffer with a 5-mile radius is applied to a point

centroid

Centroid takes a polygon or multipolygon and returns its geometric center as a point.

Syntax

```
centroid( pl1 polygon )
centroid( mpl1 multipolygon )
```

Return type

Point

Examples

The City GIS technician wants to display the multipolygons of the building footprints 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   multipolygon);
```

The centroid function returns the centroid of each building footprint multipolygon. The asbinaryshape function converts centroid point into a shape, the external representation that is recognized by the application.

```
select building_id,
       cast(asbinaryshape(centroid (footprint)) as blob(1m)) "Centroid"
from BUILDINGFOOTPRINTS;
```

contains

Contains takes two geometry objects and returns 1 (TRUE) if the first object completely contains the second. Otherwise it returns 0 (FALSE).

Syntax

contains(g1 geometry, g2 geometry)

Return type

integer

Examples

In the example below two tables are created. One table contains a city's building footprints while the other table contains its lots. The city engineer wants to make sure that all the building footprints are completely inside their lots.

In both tables the multipolygon data type stores the geometry of the building footprints and the lots. The database designer selected multipolygons for both features. She realized that often the lots can be disjointed by natural features, such as a river, and that the building footprints can often be made of several buildings.

```
create table BUILDINGFOOTPRINTS (building_id integer,
                                 lot_id      integer,
                                 footprint   multipolygon);

create table LOTS (lot_id integer, lot multipolygon);
```

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

```
select building_id
from BUILDINGFOOTPRINTS, LOTS
where contains(lot,footprint) = 0;
```

The city engineer is smart. She realizes that the first query will provide her with a list of all building_id that have footprints outside of a lot polygon. But, she also knows that this information will not tell her if the other buildings 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 "building id", bf.lot_id "buildings lot_id",
       LOTS.lot_id "LOTS lot_id"
from BUILDINGFOOTPRINTS bf, LOTS
where contains(lot,footprint) = 1 and LOTS.lot_id <> bf.lot_id;
```

In Figure 29, the building footprints labeled with their building IDs lie inside their lots. The lot lines are illustrated with dotted lines. Although not shown, these lines extend to the street centerline and completely encompass the lots and the building footprints within them.

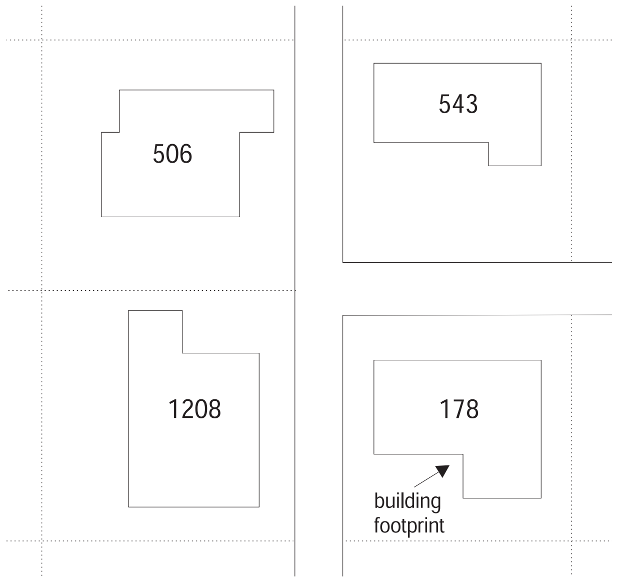


Figure 29. Using contains to ensure all buildings are contained within their lots

convexhull

Convexhull takes a geometry object and returns the convex hull.

Syntax

convexhull(g1 geometry)

Return type

Geometry

Examples

The example creates the CONVEXHULL_TEST table that has two columns: geotype and g1. Geotype, a varchar(20), will store the name of the subclass of geometry that is stored in g1, which is defined as a geometry.

```
create table CONVEXHULL_TEST (geotype varchar(20), g1 geometry)
```

Each insert statement inserts a geometry of each subclass type into the CONVEXHULL_TEST table.

```
insert into CONVEXHULL_TEST
values('Point',
       pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into CONVEXHULL_TEST
values('Linestring',
       linefromtext('linestring  (10.02 20.01,10.32 23.98,11.92 25.64)',
                    coordref()..srid(1)))

insert into CONVEXHULL_TEST
values('Polygon',
       polyfromtext('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                               19.15 33.94,10.02 20.01))',
                    coordref()..srid(1)))

insert into CONVEXHULL_TEST
values('Multipoint',
       mpointfromtext('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
                      coordref()..srid(1)))

insert into CONVEXHULL_TEST
values('Multilinestring',
       mlinefromtext('multilinestring ((10.02 20.01,10.32 23.98,11.92 25.64),
                                       (9.55 23.75,15.36 30.11))',
                     coordref()..srid(1)))

insert into CONVEXHULL_TEST
values('Multipolygon',
       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)))',
                     coordref()..srid(1)))
```

The select statement list the subclass name stored in the geotype column and the convex hull. The convexhull generated by the convexhull function is converted to text by the astext function. It is then cast to a varchar(256) because the default output of astext is varchar(4000).

```
select geotype, cast(astext(convexhull(g1))) as varchar(256) "The convexhull"
from CONVEXHULL_TEST
```

cross

Cross takes two geometry objects and returns 1 (TRUE) if their intersection results in a geometry object whose dimension is one less than the maximum dimension of the source objects. The intersection object contains points that are interior to both source geometries and is not equal to either of the source objects. Otherwise it returns 0 (FALSE).

Syntax

cross(g1 geometry, g2 geometry)

Return type

integer

Examples

The county government is considering a new regulation, which states that all hazardous waste storage facilities within the county cannot be within 5-miles of any waterway. The county GIS manager has an accurate representation of rivers and streams, which are stored as multilinestrings in the WATERWAYS table. But, he only has a single point location for each of the hazardous waste storage facilities.

```
create table WATERWAYS (id     integer,
                        name   varchar(128),
                        water  multilinestring);

create table HAZARDOUS_SITES ( site_id   integer,
                               name      varchar(128),
                               location  point);
```

To determine if he must alert the County Supervisor to any existing facilities that would violate the proposed regulation, the GIS manager will have to buffer the hazardous site locations and see if any rivers or streams cross the buffer polygon. The cross predicate compares the buffered HAZARDOUS_SITES with WATERWAYS. So, only those records in which the waterway crosses over the counties proposed regulated radius are returned.

```
select ww.name "River or stream", hs.name "Hazardous site"
from WATERWAYS ww, HAZARDOUS_SITES hs
where cross(buffer(hs.location,(5 * 5280)),ww.water) = 1;
```

In Figure 30 on page 83, the 5-mile buffer of the hazardous waste sites crosses the stream network that runs through the county’s administrative district. The stream network has been defined as a multilinestring. So, the result set includes all linestring segments that are part of those segments that cross the radius.

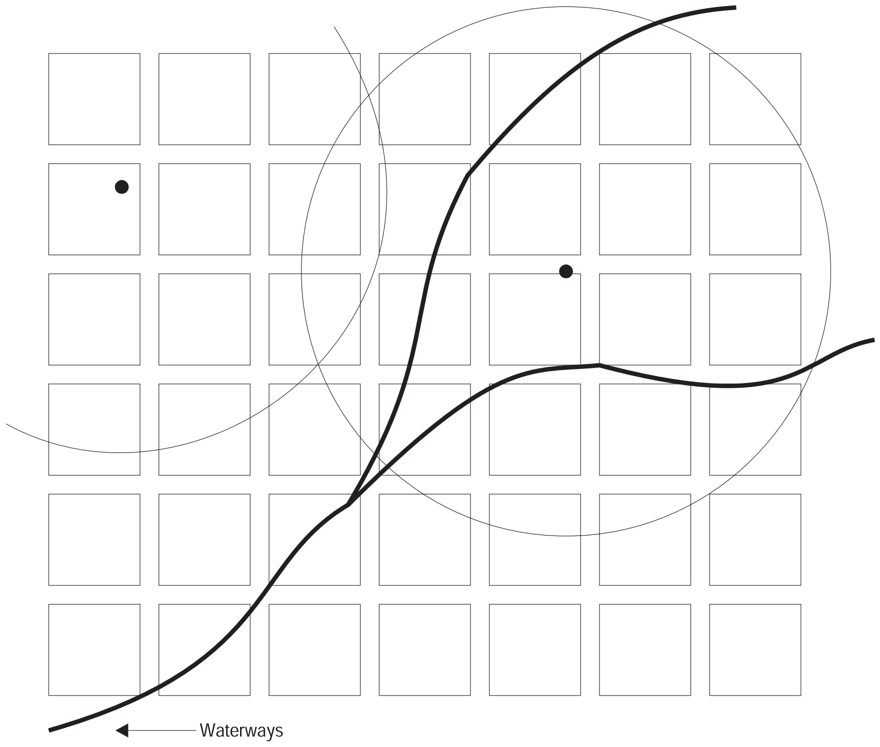


Figure 30. Using cross to find the waterways that pass through a hazardous waste area

difference

Difference takes two geometry objects and returns a geometry object that is the difference of the source objects

Syntax

difference(g1 geometry, g2 geometry)

Return type

Geometry

Examples

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

```
create table BUILDINGFOOTPRINTS (building_id integer,
                                 lot_id      integer,
                                 footprint   multipolygon);

create table LOTS (lot_id   integer,
                   lot   multipolygon);
```

The city engineer equijoins the BUILDINGFOOTPRINTS and LOTS table on the lot_id. She then takes the sum of the area of the difference of the lots, less the building footprints.

```
select sum(area(difference(lot,footprint)))
from BUILDINGFOOTPRINTS bf, LOTS
where bf.lot_id = LOTS.lot_id;
```

dimension

Dimension takes a geometry object and returns its dimension

Syntax

dimension(g1 geometry)

Return type

Integer

Examples

The DIMENSION_TEST table is created with the columns geotype and g1. The geotype column stores the name of the geometry subclass that is stored in the g1 geometry column.

```
create table DIMENSION_TEST (geotype varchar(20), g1 geometry)
```

The insert statements insert a sample subclass into the DIMENSION_TEST table.

```
insert into DIMENSION_TEST
values('Point',
       pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into DIMENSION_TEST
values('Linestring',
       linefromtext('linestring  (10.02 20.01,10.32 23.98,11.92 25.64)',
                    coordref()..srid(1)))

insert into DIMENSION_TEST
values('Polygon',
       polyfromtext('polygon  ((10.02 20.01,11.92 35.64,25.02 34.15,
                                19.15 33.94,10.02 20.01))',
                    coordref()..srid(1)))

insert into DIMENSION_TEST
values('Multipoint',
       mpointfromtext('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
                      coordref()..srid(1)))

insert into DIMENSION_TEST
values('Multilinestring',
       mlinefromtext('multilinestring ((10.02 20.01,10.32 23.98,11.92 25.64),
                                       (9.55 23.75,15.36 30.11))',
                     coordref()..srid(1)))

insert into DIMENSION_TEST
values('Multipolygon',
       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)))',
                     coordref()..srid(1)))
```

The select statement lists the subclass name stored in the geotype column with the dimension of that geotype.

```
select geotype, dimension(g1) "The dimension"
from DIMENSION_TEST

GEOTYPE              The dimension
-------------------- -------------
Point                            0
Linestring                       1
Polygon                          2
Multipoint                       0
Multilinestring                  1
Multipolygon                     2

  6 record(s) selected.
```

disjoint

Disjoint takes two geometries and returns 1 (TRUE) if the intersection of two geometries produces an empty set, otherwise it returns 0 (FALSE)

Syntax

disjoint(g1 geometry, g2 geometry)

Return type

Integer

Examples

An insurance company wants to assess the insurance coverage for the towns hospital, nursing homes, and schools. Part of this process includes determining the threat that the hazardous waste sites pose to each institution. 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 not within a 5-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 institution's polygon geometry.

```
create table SENSITIVE_AREAS (id        integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      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(128),
                              location  point);
```

The select statement lists the names of all sensitive areas that are not within the five mile radius of a hazardous waste site. The intersects function could replace the disjoint function in this query as long as the result of the function is set equal to 0 instead of 1, since intersects and disjoint return the exact opposite result.

```
select sa.name
from SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
where disjoint(buffer(hs.location,(5 * 5280)),sa.zone) = 1;
```

In Figure 31, sensitive are sites are compared to the 5-mile radius of the hazardous waste sites. The nursing home is the only sensitive area where the disjoint function will return 1 (TRUE). The disjoint function returns 1 whenever two geometries do not intersect in any way.

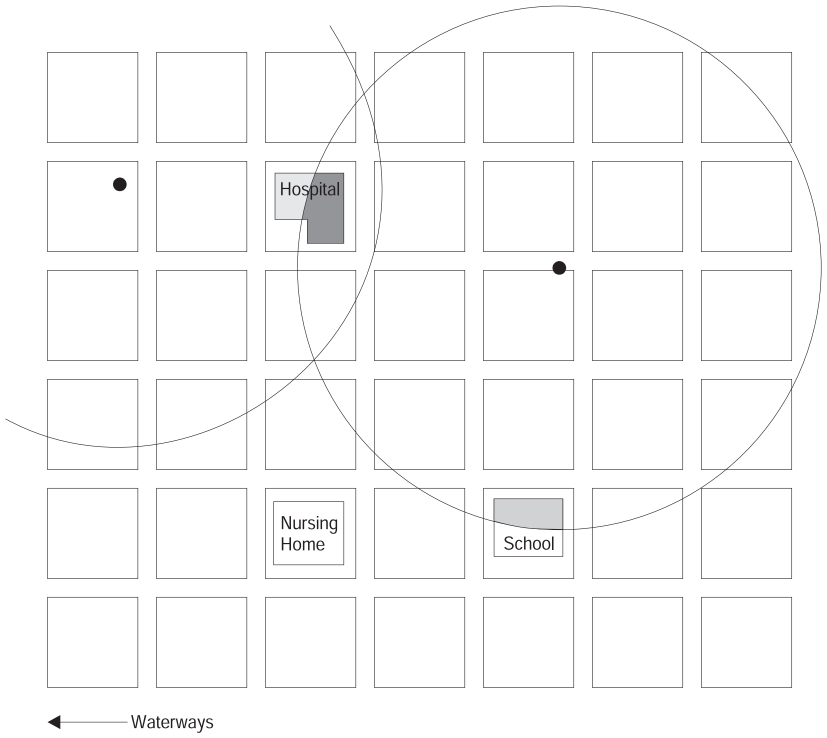


Figure 31. Using disjoint to find the buildings that do not lie within (intersect) any hazardous waste area

distance

Distance takes two geometries and returns the closest distance separating them.

Syntax

distance(g1 geometry, g2 geometry)

Return type

Double precision

Examples

The city engineer needs a list of all buildings that are within 1 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 footprint multipolygon stores the geometry of each building's footprint.

```
create table BUILDINGFOOTPRINTS (  building_id integer,
                                   lot_id      integer,
                                   footprint   multipolygon);
```

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   multipolygon);
```

The query returns a list of building IDs that are within one foot of their lot lines. The distance function performs a spatial join between the footprints and the boundary of the lot multipolygons. However, the equijoin between the bf.lot_id and LOTS.lot_id insures that only the multipolygons belonging to the same lot are compared by the **distance** function.

```
select bf.building_id
  from BUILDINGFOOTPRINTS bf, LOTS
 where bf.lot_id = LOTS.lot_id AND
       distance(bf.footprint,boundary(LOTS.lot)) <= 1.0;
```

endpoint

Endpoint takes a linestring and returns a point that is the linestring's last point.

Syntax

endpoint(ln1 linestring)

Return type

Point

Examples

The ENDPOINT_TEST table stores the gid integer column which uniquely identifies each row and the ln1 linestring column that stores linestrings.

```
create table ENDPOINT_TEST (gid integer, ln1 linestring)
```

The insert statements insert linestrings into the ENDPOINT_TEST table. The first one does not have Z coordinates or measures, while the second one does.

```
insert into ENDPOINT_TEST
values( 1,
        linefromtext('linestring (10.02 20.01,23.73 21.92,30.10 40.23)',
                     coordref()..srid(1)))

insert into ENDPOINT_TEST
values(2,
       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)',
                    coordref()..srid(1)))
```

The query lists the gid column with the output of the endpoint function. The endpoint function generates a point geometry which is converted to text by the astext function. The cast function is used to shorten the default varchar(4000) value of the astext function to a varchar(60).

```
select gid, cast(astext(endpoint(ln1)) as varchar(60)) "Endpoint"
from ENDPOINT_TEST

GID         Endpoint
----------- ------------------------------------------------------------
          1 POINT ( 30.10000000 40.23000000)
          2 POINT ZM ( 30.10000000 40.23000000 7.00000000 7.20000000)

  2 record(s) selected.
```

envelope

Envelope takes a geometry object and returns its bounding box as a geometry.

Syntax

envelope(g1 geometry)

Return type

Geometry

Examples

The geotype column in the ENVELOPE_TEST tables stores the name of the geometry subclass stored in the g1 geometry column.

create table ENVELOPE_TEST (geotype varchar(20), g1 geometry)

The insert statements insert each geometry subclass into the ENVELOPE_TEST table.

```
insert into ENVELOPE_TEST
values('Point',
       pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into ENVELOPE_TEST
values ('Linestring',
        linefromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)',
        coordref()..srid(1)))

insert into ENVELOPE_TEST
values('Linestring',
       linefromtext('linestring  (10.02 20.01,10.32 23.98,11.92 25.64)',
       coordref()..srid(1)))

insert into ENVELOPE_TEST
values('Polygon',
       polyfromtext('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                            19.15 33.94,10.02 20.01))',
                   coordref()..srid(1)))

insert into ENVELOPE_TEST
values('Multipoint',
       mpointfromtext('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
       coordref()..srid(1)))

insert into ENVELOPE_TEST
values('Multilinestring',
       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))',
                    coordref()..srid(1)))
```

```
insert into ENVELOPE_TEST
values('Multilinestring',
       mlinefromtext('multilinestring ((10.02 20.01,10.32 23.98,11.92 25.64),
                                       ( 9.55 23.75,15.36 30.11))',
                     coordref()..srid(1)))

insert into ENVELOPE_TEST
values('Multipolygon',
       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)))',
                     coordref()..srid(1)))
```

The query lists the subclass name along side its envelope. Since the envelope function returns either a point, linestring or polygon its output is converted to text with the astext function. The cast function converts the default varchar(4000) result of the astext function to a varchar(280).

```
select geotype, cast(astext(envelope(g1)) as varchar(280)) "The envelope"
from ENVELOPE_TEST

GEOTYPE              The envelope
-------------------- ----------------------------------------------------------
Point                POINT ( 10.02000000 20.01000000)
Linestring           LINESTRING ( 10.01000000 20.01000000, 10.01000000
 40.01000000)
Linestring           POLYGON (( 10.02000000 20.01000000, 11.92000000
 20.01000000, 11.92000000 25.64000000, 10.02000000 25.64000000, 10.02000000
 20.01000000))
Polygon              POLYGON (( 10.02000000 20.01000000, 25.02000000
 20.01000000, 25.02000000 35.64000000, 10.02000000 35.64000000, 10.02000000
 20.01000000))
Multipoint           POLYGON (( 10.02000000 20.01000000, 11.92000000
 20.01000000, 11.92000000 25.64000000, 10.02000000 25.64000000, 10.02000000
 20.01000000))
Multilinestring      LINESTRING ( 10.01000000 20.01000000, 50.01000000
 20.01000000)
Multilinestring      POLYGON (( 9.55000000 20.01000000, 15.36000000
 20.01000000, 15.36000000 30.11000000, 9.55000000 30.11000000, 9.55000000
 20.01000000))
Multipolygon         POLYGON (( 10.02000000 20.01000000, 73.36000000
 20.01000000, 73.36000000 35.64000000, 10.02000000 35.64000000, 10.02000000
 20.01000000))

  8 record(s) selected.
```

envelopesintersect

Envelopesintersect returns 1 (TRUE) if the envelopes of two geometries intersect. Otherwise it returns 0 (FALSE).

Syntax

envelopesintersect(g1 geometry, g2 geometry)

Return type

Integer

Examples

The get_window function retrieves the display windows coordinates from the application. The window parameter is actually a polygon shape structure containing a string of coordinates that represent the display polygon. The polygonfromshape function converts the display window shape into a Spatial Extender polygon which the envelopesintersect function uses as its intersection envelopes. 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(&window)

/* Create the SQL expression. The envelopesintersect function
  will be used to limit the result set to only those zone polygons
  that intersect the envelope of the display window. */
strcpy(sqlstmt, "select AsBinaryShape(zone) from SENSITIVE_AREAS where
envelopesintersect (zone,polyfromshape(cast(? as blob(1m)),
coordref..srid(1)))");

/* Set blob_len to the byte length of a 5 point shape polygon. */
blob_len = 128;

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

/* Set the pcbvalue1 to the window shape */
pcbvalue1 = blob_len;

/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB, blob_len,
0, window, blob_len, &pcbvalue1);

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

/* Assign the results of the query, (the Zone polygons) to the
   fetched_binary variable. */
```

```
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);

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

equals

Equals compares two geometries and returns 1 (TRUE) if the geometries are identical, otherwise it returns 0 (FALSE)

Syntax

equals(g1 geometry, g2 geometry)

Return type

Integer

Examples

The City GIS technician suspects that some of the data in the BUILDINGFOOTPRINTS table was somehow duplicated. To alleviate his concern he queries the table to determine if any of the footprint's 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   multipolygon);
```

The BUILDINGFOOTPRINTS table is spatially joined to itself by the equals predicate which returns 1 whenever it finds two of the multipolygons that are equal. The bf1.building_id <> bf2.building_id condition is required to eliminate the comparison of the same geometry.

```
select bf1.building_id, bf2.building_id
from BUILDINGFOOTPRINTS bf1, BUILDINGFOOTPRINTS bf2
where equals(bf1.footprint,bf2.footprint) = 1
      and bf1.building_id <> bf2.building_id;
```

exteriorring

Exteriorring takes a polygon and returns its exterior ring as a linestring

Syntax

exteriorring(pl1 polygon)

Return type

Linestring

Examples

An ornithologist, wishing to study the bird population on several south sea islands, knows that the feeding zone of the bird species she is interested in is restricted to the shoreline. As part of her calculation of the island's carrying capacity, the ornithologist requires the island's perimeter. Some of the islands are so large they have several ponds on them. However, the shoreline of the ponds are inhabited exclusively by another more aggressive bird species. Therefore, the ornithologist requires the perimeter of the exterior ring of the islands.

The ID and name columns of the ISLANDS table identifies each island, and the land polygon column stores the geometry of each.

```
create table ISLANDS (id   integer,
                         name   varchar(32),
                         land   polygon);
```

The exteriorring function extracts the exterior ring from each island polygon as a linestring. The length of the linestring is established by the length function. The linestring lengths are summarized by the sum function.

```
select sum(length(exteriorring (land))) from ISLANDS;
```

In Figure 32 on page 97, 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.

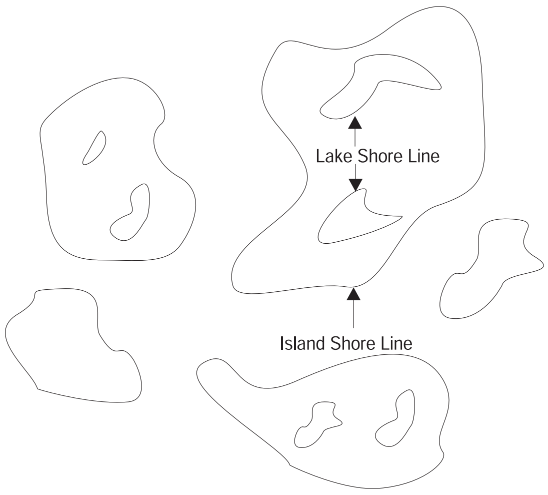


Figure 32. Using exteriorring to determine the length of an island shore line

geometryfromshape

Geometryfromshape takes a shape and a spatial reference system identity to return a geometry object.

Syntax

geometryfromshape(s1 blob(1m), srid coordref)

Return type

Geometry

Examples

The following C code fragment contains ODBC functions embedded with Spatial Extender 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 that contains the geometry of each lot.

```
create table LOTS ( lot_id  integer,
                    lot     multipolygon);
```

The geometryfromshape function converts shapes into Spatial Extender 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 lot id and the
  lot multipolygon. The question marks are parameter markers that
  indicate the lot_id and lot values that will be retrieved at
  runtime. */
strcpy (shp_sql,"insert into LOTS (lot_id, lot) values (?, geometryfromshape
(cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer key value 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 shape to the second parameter. */
pcbvalue2 = blob_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, shape_blob, blob_len, &pcbvalue2);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

geometryfromtext

Geometryfromtext takes a well-known text representation and a spatial reference system identity and returns a geometry object.

Syntax

geometryfromtext(wkt varchar(4000), srid coordref)

Return type

Geometry

Examples

The GEOMETRY_TEST table contains the integer gid column which uniquely identifies each row and the g1 column that stores the geometry.

```
create table GEOMETRY_TEST (gid smallint, g1 geometry)
```

The insert statements inserts the data into the gid and g1 columns of the GEOMETRY_TEST table. The geometryfromtext function converts the text representation of each geometry into its corresponding Spatial Extender instantiable subclass.

```
insert into GEOMETRY_TEST
values(1, geometryfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into GEOMETRY_TEST
values (2,
        geometryfromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)',
                         coordref()..srid(1)))

insert into GEOMETRY_TEST
values(3,
       geometryfromtext('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                                   19.15 33.94,10.02 20.01))',
                        coordref()..srid(1)))

insert into GEOMETRY_TEST
values(4,
       geometryfromtext('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
                        coordref()..srid(1)))

insert into GEOMETRY_TEST
values(5,
       geometryfromtext('multilinestring ((10.02 20.01,10.32 23.98,
                                           11.92 25.64),
                                          ( 9.55 23.75,15.36 30.11))',
                        coordref()..srid(1)))
```

```
insert into GEOMETRY_TEST
values(6,
       geometryfromtext('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)))',
                        coordref()..srid(1)))
```

geometryfromwkb

Geometryfromwkb takes a well-known binary representation and a spatial reference system identity to return a geometry object.

Syntax

geometryfromwkb(wkb blob(1m), srid coordref)

Return type

Geometry

Examples

The following C code fragment contains ODBC functions embedded with Spatial Extender 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 that contains the geometry of each lot.

```
create table LOTS ( lot_id   integer,
                         lot      multipolygon);
```

The geometryfromwkb function converts WKB representations into Spatial Extender geometry. The entire insert statement is copied into wkb_shp 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 lot id and the
   lot multipolygon. The question marks are parameter markers that
   indicate the lot_id and lot values that will be retrieved at
   runtime. */
strcpy (wkb_sql,"insert into LOTS (lot_id, lot) values (?, geometryfromwkb
(cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the integer key value 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 shape to the second parameter. */
pcbvalue2 = blob_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, shape_blob, blob_len, &pcbvalue2);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

geometryn

geometryn takes a collection and an integer index and returns the nth geometry object in the collection

Syntax

geometryn(mpt1 multipoint, index integer)
geometryn(mln1 multilinestring, index integer)
geometryn(mpl1 multipolygon, index integer)

Return type

Geometry

Examples

The city engineer wants to know if the building footprints are all inside the first polygon of the lot's multipolygon.

The building_id column uniquely identifies each row of the BUILDINGFOOTPRINTS table. The lot_id column identifies the building's lot. The footprints column stores the building's geometry.

```
create table BUILDINGFOOTPRINTS ( building_id integer,
                                  lot_id      integer,
                                  footprint   multipolygon);

create table LOTS ( lot_id   integer,
                    lot   multipolygon);
```

The query lists the BUILDINGFOOTPRINTS building_id and lot_id for all building footprints that are all within the first lot polygon. The geometryn function returns a first lot polygon in the multipolygon array.

```
select bf.building_id,bf.lot_id
from BUILDINGFOOTPRINTS bf,LOTS
where within(footprint,geometryn (lot,1)) = 1
      and bf.lot_id = LOTS.lot_id;
```

geometrytype

Geometrytype takes a geometry object and returns its geometry type as a string.

Syntax

geometrytype (g1 geometry)

Return type

Varchar(32) containing either: Point, LineString, Polygon, MultiPoint, MultiLineString, or MultiPolygon

Examples

The GEOMETRYTYPE_TEST table contains the g1 geometry column.

```
create table GEOMETRYTYPE_TEST(g1 geometry)
```

The insert statements insert each geometry subclass into the g1 column.

```
insert into GEOMETRYTYPE_TEST
values(geometryfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into GEOMETRYTYPE_TEST
values (geometryfromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)',
                         coordref()..srid(1)))

insert into GEOMETRYTYPE_TEST
values(geometryfromtext('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                                   19.15 33.94, 10.02 20.01))',
                        coordref()..srid(1)))

insert into GEOMETRYTYPE_TEST
values(geometryfromtext('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
                        coordref()..srid(1)))

insert into GEOMETRYTYPE_TEST
values(geometryfromtext('multilinestring ((10.02 20.01,10.32 23.98,
                                           11.92 25.64),
                                          (9.55 23.75,15.36 30.11))',
                        coordref()..srid(1)))

insert into GEOMETRYTYPE_TEST
values(geometryfromtext('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)))',
                        coordref()..srid(1)))
```

The query lists the geometry type of each subclass that is stored in the g1 geometry column.

```
select **geometrytype**(g1) "Geometry type" from GEOMETRYTYPE_TEST

Geometry type
--------------------------------
Point
LineString
Polygon
MultiPoint
MultiLineString
MultiPolygon

  6 record(s) selected.
```

interiorringn

Returns the nth interior ring of a polygon as a linestring. The rings are not organized by geometric orientation. They are organized according to the rules defined by the internal geometry verification routines. So, the order of the rings cannot be predefined.

Syntax

interiorringn(pl1 polygon, index integer)

Return type

Linestring

Examples

An ornithologist is studying the bird population on several south sea islands. He knows that the feeding zone of this passive species is restricted to the seashore. Some of the islands are so large that they have several lakes on them. The shorelines of the lakes are inhabited exclusively by another more aggressive species. The ornithologist knows that for each island, if the perimeter of the ponds 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.

In Figure 33 on page 107, 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.

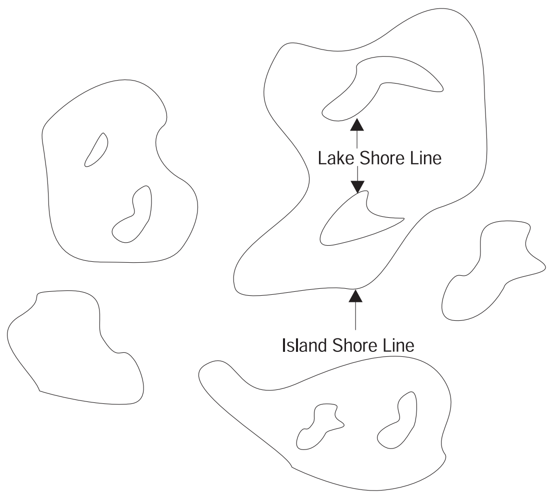


Figure 33. Using interiorringn to determine the length of the lakeshores within each island

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  polygon);
```

The following ODBC program uses the interiorringn function to extract the interior ring (lake) from each island polygon as a linestring. The perimeter of the linestring that is returned by the length function is totaled and displayed along with the island's ID.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "sg.h"
#include "sgerr.h"
#include "sqlcli1.h"

/***                        ***
 *** Change these constants ***
 ***                        ***/

#define USER_NAME   "sdetest"   /* your user name */
#define USER_PASS   "acid.rain" /* your user password */
#define DB_NAME     "mydb"      /* database to connect to */
```

```
static void check_sql_err (SQLHDBC  handle,
                           SQLHSTMT hstmt,
                           LONG     rc,
                           CHAR     *str);

void main( argc, argv )
int argc;
char *argv[];
{
  SQLHDBC       handle;
  SQLHENV       henv;
  CHAR          sql_stmt[256];
  LONG          rc,
                total_perimeter,
                num_lakes,
                lake_number,
                island_id,
                lake_perimeter;
  SQLHSTMT      island_cursor,
                lake_cursor;
  SDWORD        pcbvalue,
                id_ind,
                lake_ind,
                length_ind;

/* Allocate memory for the ODBC environment handle henv and initialize the application. */

  rc = SQLAllocEnv (&henv);
  if (rc != SQL_SUCCESS)
  {
    printf ("SQLAllocEnv failed with %d\n", rc);
    exit(0);
  }

/* Allocate memory for a connection handle within the henv environment. */

  rc = SQLAllocConnect (henv, &handle);
  if (rc != SQL_SUCCESS)
  {
    printf ("SQLAllocConnect failed with %d\n", rc);
    exit(0);
  }

/* Load the ODBC driver and connect to the data source identified by the database,
   user, and password.*/

  rc = SQLConnect (handle,
                   (UCHAR *)DB_NAME,
                   SQL_NTS,
                  (UCHAR *)USER_NAME,
                  SQL_NTS,
                  (UCHAR *)USER_PASS,
                  SQL_NTS);

  check_sql_err (handle, NULL, rc, "SQLConnect");

/* Allocate memory to the SQL statement handle island_cursor. */

  rc = SQLAllocStmt (handle, &island_cursor);
  check_sql_err (handle, NULL, rc, "SQLAllocStmt");

/* Prepare and execute the query to get the island IDs and number of
   lakes (interior rings) */

  strcpy (sql_stmt, "select id, numinteriorrings(land) from ISLANDS");

  rc = SQLExecDirect (island_cursor, (UCHAR *)sql_stmt, SQL_NTS);
  check_sql_err (NULL, island_cursor, rc, "SQLExecDirect");

/* Bind the island table's ID column to the variable island_id */

  rc = SQLBindCol (island_cursor, 1, SQL_C_SLONG, &island_id, 0, &id_ind);
  check_sql_err (NULL, island_cursor, rc, "SQLBindCol");

/* Bind the result of numinteriorrings(land) to the num_lakes variable. */

  rc = SQLBindCol (island_cursor, 2, SQL_C_SLONG, &num_lakes, 0, &lake_ind);
  check_sql_err (NULL, island_cursor, rc, "SQLBindCol");

/* Allocate memory to the SQL statement handle lake_cursor. */

rc = SQLAllocStmt (handle, &lake_cursor);
  check_sql_err (handle, NULL, rc, "SQLAllocStmt");

/* Prepare the query to get the length of an interior ring. */

  strcpy (sql_stmt,
          "select Length(interiorringn(land, cast (? as
           integer))) \ from ISLANDS where id = ?");

  rc = SQLPrepare (lake_cursor, (UCHAR *)sql_stmt, SQL_NTS);
  check_sql_err (NULL, lake_cursor, rc, "SQLPrepare");

/* Bind the lake_number variable as the first input parameter */

  pcbvalue = 0;
  rc = SQLBindParameter (lake_cursor, 1, SQL_PARAM_INPUT, SQL_C_LONG,
       SQL_INTEGER, 0, 0, &lake_number, 0, &pcbvalue);
  check_sql_err (NULL, lake_cursor, rc, "SQLBindParameter");

/* Bind the island_id as the second input parameter */

  pcbvalue = 0;
  rc = SQLBindParameter (lake_cursor, 2, SQL_PARAM_INPUT, SQL_C_LONG,
       SQL_INTEGER, 0, 0, &island_id, 0, &pcbvalue);
  check_sql_err (NULL, lake_cursor, rc, "SQLBindParameter");

/* Bind the result of the Length(interiorringn(land, cast (? as integer)))
   to the variable lake_perimeter */

  rc = SQLBindCol (lake_cursor, 1, SQL_C_SLONG, &lake_perimeter, 0,
                  &length_ind);
  check_sql_err (NULL, island_cursor, rc, "SQLBindCol");

  /* Outer loop, get the island ids and the number of lakes (interior rings) */

  while (SQL_SUCCESS == rc)
  {
    /* Fetch an island */

    rc = SQLFetch (island_cursor);

    if (rc != SQL_NO_DATA)
    {
      check_sql_err (NULL, island_cursor, rc, "SQLFetch");

      /* Inner loop, for this island, get the perimeter of all of
         its lakes (interior rings) */

      for (total_perimeter = 0,lake_number = 1;
           lake_number <= num_lakes;
           lake_number++)
      {
        rc = SQLExecute (lake_cursor);
        check_sql_err (NULL, lake_cursor, rc, "SQLExecute");

        rc = SQLFetch (lake_cursor);
        check_sql_err (NULL, lake_cursor, rc, "SQLFetch");

        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);
  SQLDisconnect (handle);
  SQLFreeConnect (handle);
  SQLFreeEnv (henv);

  printf( "\nTest Complete ...\n" );

}

static void check_sql_err (SQLHDBC handle, SQLHSTMT  hstmt, LONG rc,
                           CHAR *str)
{

    SDWORD dbms_err = 0;
    SWORD  length;
    UCHAR   err_msg[SQL_MAX_MESSAGE_LENGTH], state[6];

    if (rc != SQL_SUCCESS)
    {
      SQLError (SQL_NULL_HENV, handle, hstmt, state, &dbms_err,
                err_msg, SQL_MAX_MESSAGE_LENGTH - 1, &length);
      printf ("%s ERROR (%d): DBMS code:%d, SQL state: %s, message: \n %s\n",
              str, rc, dbms_err, state, err_msg);

      if (handle)
      {
        SQLDisconnect (handle);
        SQLFreeConnect (handle);
      }
      exit(1);
    }
}
```

intersection

Intersection takes two geometry objects and returns the intersection set as a geometry object.

Syntax

intersection(g1 geometry, g2 geometry)

Return type

Geometry

Examples

The Fire Marshall must obtain the areas of the hospitals, schools, and nursing homes that are intersected by the radius of a possible hazardous waste contamination.

The sensitive areas are stored in the table SENSITIVE_AREAS that is created with the CREATE TABLE statement that follows. The zone column is defined as a polygon that stores the outline of each of the sensitive areas.

```
create table SENSITIVE_AREAS (id    integer,
                              name  varchar(128),
                              size  float,
                              type  varchar(10),
                              zone  polygon);
```

The hazardous sites are stored in the HAZARDOUS_SITES table that is 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(128),
                              location  point);
```

The buffer function generates a 5-mile buffer that surrounds the hazardous waste site locations. The intersection function generates polygons from the intersection of the buffered hazardous waste site polygons and the sensitive areas. The area function returns the intersection polygon's area, which is summarized for each hazardous site by the sum function. The group by clause directs the query to aggregate the intersected areas by the hazardous waste site_ID.

```
select hs.name,sum(area(intersection (sa.zone,buffer hs.location,(5 * 5280)))))
from SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
group by hs.site_id;
```

In Figure 34, the circles represent the buffered polygons that surround the hazardous waste sites. The intersection of these buffer polygons with the sensitive area polygons produces three other 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.

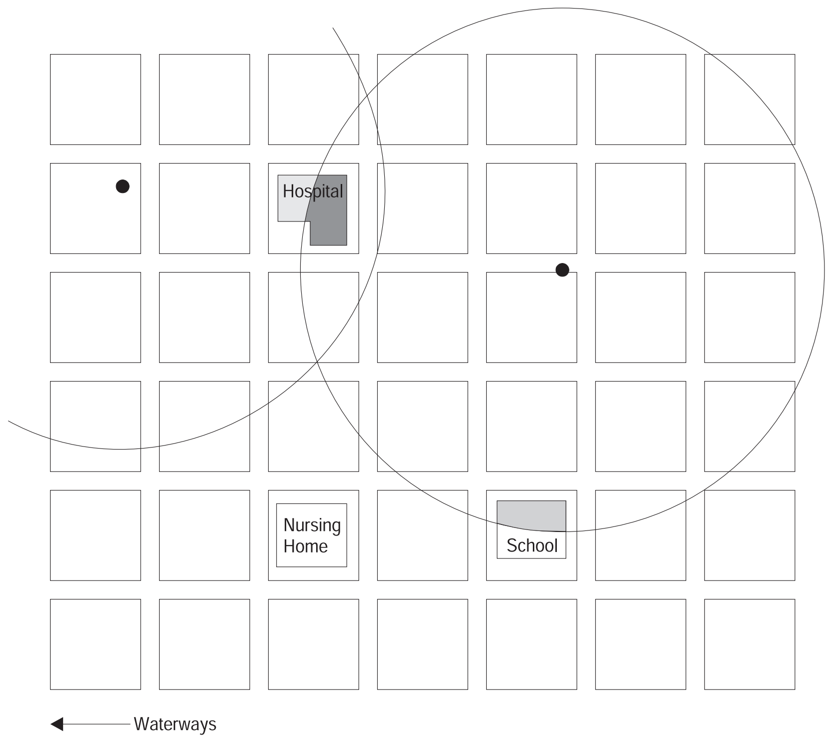


Figure 34. Using intersection to determine how large an area in each of the buildings might be affected by hazardous waste

intersects

Intersects takes two geometries and returns 1 (TRUE), if the intersection of two geometries does not result in an empty set. Otherwise, it returns 0 (FALSE).

Syntax

intersects (g1 geometry, g2 geometry)

Return type

Integer

Examples

The Fire Marshall wants a list of all sensitive areas within a five mile radius of a hazardous waste site.

The sensitive areas are stored in the table SENSITIVE_AREAS that is created with the CREATE TABLE statement that follows. The zone column is defined as a polygon that stores the outline of each of the sensitive areas.

```
create table SENSITIVE_AREAS (id       integer,
                              name     varchar(128),
                              size     float,
                              type     varchar(10),
                              zone     polygon);
```

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(128),
                              location  point);
```

The query returns a list of sensitive areas and hazardous site names for sensitive areas that intersect the 5-mile buffer of the hazardous sites.

```
select sa.name, hs.name
from SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
where intersects(buffer(hs.location,(5 * 5280)),sa.zone) = 1;
```

is3d

Is3D takes a geometry object and returns 1 (TRUE) if the object has 3D coordinates; otherwise, it returns 0 (FALSE)

Syntax

is3d(g1 geometry)

Return type

Integer

Examples

The THREED_TEST table is created with the integer gid column and the g1 geometry column.

```
create table THREED_TEST (gid smallint, g1 geometry)
```

The 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, pointfromtext(point (10 10),coordref()..srid(1)))

insert into THREED_TEST
values (2, pointfromtext(point z (10.92 10.12 5),coordref()..srid(1)))
```

The query lists the contents of the gid column with the results of the is3d function. The function returns a 0 for the first row which does not have Z coordinates, and a 1 for the second row which does have Z coordinates.

```
select gid,is3d (g1) "Is it 3d?" from THREED_TEST

gid      Is it 3d?
------   ----------
     1            0
     2            1
```

isclosed

Isclosed takes a linestring or multilinestring and returns 1 (TRUE) if it is closed; otherwise it returns 0 (FALSE)

Syntax

isclosed(ln1 linestring)
isclosed(mln1 multilinestring)

Return type

Integer

Examples

The CLOSED_LINESTRING table is created with a single linestring column.

```
create table CLOSED_LINESTRING (ln1 linestring)
```

The 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(linefromtext('linestring  (10.02 20.01,10.32 23.98,11.92 25.64)',
                    coordref()..srid(1)))

insert into CLOSED_LINESTRING
values(linefromtext('linestring  (10.02 20.01,11.92 35.64,25.02 34.15,
                                  19.15 33.94,10.02 20.01)',
                    coordref()..srid(1)))
```

The query returns the results of the 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 isclosed(ln1) "Is it closed" from CLOSED_LINESTRING

Is it closed
------------
           0
           1

  2 record(s) selected.
```

The CLOSED_MULTILINESTRING table is created with a single multilinestring column.

```
create table CLOSED_MLINESTRING (mln1 multilinestring)
```

The insert statements insert a multilinestring record that is not closed and another that is.

```
insert into CLOSED_MLINESTRING
values(mlinefromtext('multilinestring ((10.02 20.01,10.32 23.98,11.92 25.64),
                                       (9.55 23.75,15.36 30.11))',
                     coordref()..srid(1)))

insert into CLOSED_MLINESTRING
values(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))',
                     coordref()..srid(1)))
```

The query lists the results of the isclosed function. The 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 isclosed(mln1) "Is it closed" from CLOSED_MLINESTRING

Is it closed
------------
           0
           1

  2 record(s) selected.
```

isempty

Isempty takes a geometry object and returns 1 (TRUE) if it is empty; otherwise it returns 0 (FALSE)

Syntax

isempty(g1 geometry)

Return type

Integer

Examples

The create table statement below creates the EMPTY_TEST table with two columns. Geotype stores the data type of the subclasses that are stored in the g1 geometry column.

```
create table EMPTY_TEST (geotype varchar(20), g1 geometry)
```

The insert statements insert two records for the geometry subclasses point, linestring and polygon. One record is empty and one is not.

```
insert into EMPTY_TEST
values('Point',pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into EMPTY_TEST
values('Point', pointfromtext('point empty', coordref()..srid(1)))

insert into EMPTY_TEST
values('Linestring',linefromtext('linestring (10.02 20.01,10.32 23.98,
                                              11.92 25.64)',
                                 coordref()..srid(1)))

insert into EMPTY_TEST
values('Linestring',linefromtext('linestring  empty',coordref()..srid(1)))

insert into EMPTY_TEST
values('Polygon',polyfromtext('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                                         19.15 33.94,10.02 20.01))',
                              coordref()..srid(1)))

insert into EMPTY_TEST
values('Polygon', polyfromtext('polygon  empty',coordref()..srid(1)))
```

The query returns the geometry type from the geotype column and the results of the isempty function.

```
select geotype, isempty(g1) "The empty" from EMPTY_TEST

GEOTYPE              Is it empty
-------------------- -----------
Point                          0
Point                          1
Linestring                     0
Linestring                     1
Polygon                        0
Polygon                        1

  6 record(s) selected.
```

ismeasured

Ismeasured takes a geometry object and returns 1 (TRUE) if the object has measures; otherwise it returns 0 (FALSE)

Syntax

ismeasured(g1 geometry)

Return type

Integer

Examples

The MEASURE_TEST table is created with two columns. The Gid uniquely identifies the rows, and g1 stores the point geometries. One insert stores a point with measures and the other stores one without.

```
create table MEASURE_TEST (gid smallint, g1 geometry)
```

The 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 records point does have a measure.

```
insert into MEASURE_TEST
values(1,pointfromtext(point (10 10),coordref()..srid(1)))

insert into MEASURE_TEST
values (2,pointfromtext(point m (10.92 10.12 5),coordref()..srid(1)))
```

The query lists the gid column along with the results of the ismeasured function. The 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,ismeasured (g1) "Has measures?" from MEASURE_TEST

gid      Has measures?
------   -------------
0
1
```

isring

Isring takes a linestring and returns 1 (TRUE) if it is a ring (namely, the linestring is closed and simple); otherwise, it returns 0 (FALSE)

Syntax

isring(ln1 linestring)

Return type

Integer

Examples

The RING_LINESTRING table is created with a single linestring column that is called ln1.

```
create table RING_LINESTRING (ln1 linestring)
```

The insert statements insert three linestrings into the ln1 column. The first row contains a linestrings that is not closed and therefore is not a ring. The second row contains a linestring that is closed and is simple and therefore is a ring. The third row contains a linestring that is closed but is not simple because it intersects its own interior; therefore it is not a ring.

```
insert into RING_LINESTRING
values(linefromtext('linestring  (10.02 20.01,10.32 23.98,11.92 25.64)',
                    coordref()..srid(1)))

insert into RING_LINESTRING
values(linefromtext('linestring (10.02 20.01,11.92 35.64,25.02 34.15,
                                 19.15 33.94, 10.02 20.01)',
                    coordref()..srid(1)))

insert into RING_LINESTRING
values(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)',
                    coordref()..srid(1)))
```

The query returns the results of the isring function. The first and third rows return a 0 because the linestrings are not rings, while the second row returns a 1 because it is a ring.

```
select isring(ln1) "Is it ring" from RING_LINESTRING

Is it ring
-----------
          0
```

```
        1
        0

3 record(s) selected.
```

issimple

issimple takes a geometry object and returns 1 (TRUE) if the object is simple; otherwise, it returns 0 (FALSE).

Syntax

issimple (g1 geometry)

Return type

Integer

Examples

The table ISSIMPLE_TEST is created with two columns. The pid column, which is a smallint, contains the unique identifier for each row. The g1 geometry column stores the simple and non-simple geometry samples.

create table ISSIMPLE_TEST (pid smallint, g1 geometry)

á

The insert statements insert two records into the ISSIMPLE_TEST table. The first is simple because it is a linestring that does not intersect its interior. The second is non-simple because it does intersect its interior.

```
insert into ISSIMPLE_TEST
values (1,linefromtext(linestring (10 10, 20 20, 30 30),coordref()..srid(1)))

insert into ISSIMPLE_TEST
values (2,linefromtext(linestring (10 10,20 20,20 30,10 30,10 20,20 10), coordref()..srid(1)))
```

The query returns the results of the issimple function. The first record returns a 1 because the linestring is simple while the second record returns a 1 because the linestring is not simple.

```
select issimple(g1)
from ISSIMPLE_TEST

g1
--------------
1
0
```

length

Length takes a linestring or multilinestring and returns its length

Syntax

length(ln1 linestring)
length(mln1 multilinestring)

Return type

Double precision

Examples

A local ecologist is studying the migratory patterns of the salmon population in the county's waterways. He wants to obtain the length of all stream and river systems running through the county.

The WATERWAYS table is created with the id and name columns which identify each stream and river system that is stored in the table. The water column is a multilinestring since the river and stream systems are often an aggregate of several linestrings.

```
create table WATERWAYS (id integer, name varchar(128), water multilinestring);
```

The query returns the name of each system along with the length of the system that is generated by the length function.

The figure displays a the river and stream systems that lie within the county boundary.

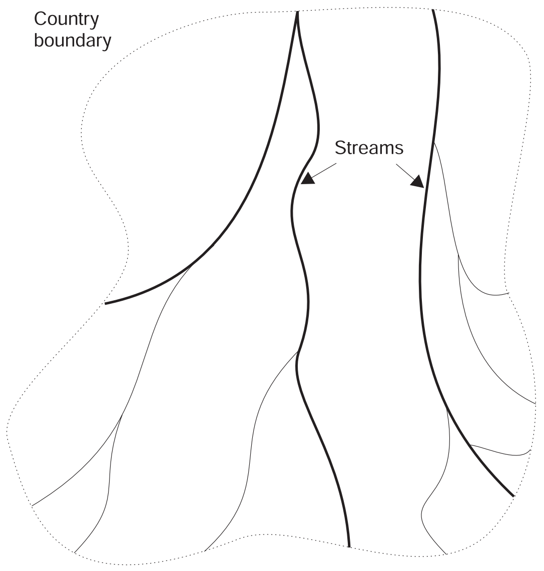


Figure 35. Using length to determine the total length of the many waterways in a county

```
select name, length(water) "Length"
from WATERWAYS;
```

linefromshape

Linefromshape takes a shape of type point and a spatial reference system identity to return a linestring.

Syntax

linefromshape(s1 blob(1m), srid coordref)

Return type

Linestring

Examples

This code fragment populates the sewerlines table with the unique id, size class and geometry of each 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 lines capacity. The sewer linestring column stores the sewer lines geometry.

```
create table sewerlines (sewer_id   integer, class  integer,
                         sewer linestring);

/* Create the SQL insert statement to populate the sewer_id, size class and
   the sewer linestring. The question marks are parameter markers that
   indicate the sewer_id, class and sewer geometry values that will be
   retrieved at runtime. */
strcpy (shp_sql,"insert into sewerlines (sewer_id,class,sewer)
values (?,?, linefromshape (cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer key value 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 integer class value 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 shape to the third parameter. */
```

```
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, sewer_shape, blob_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

linefromtext

Linefromtext takes a well-known text representation of type linestring and a spatial reference system identity and returns a linestring

Syntax

linefromtext(wkt varchar(4000), srid coordref)

Return type

Linestring

Examples

The LINESTRING_TEST table is created with a single ln1 linestring column.

```
create table LINESTRING_TEST (ln1 linestring)
```

The insert statement inserts a linestring into the ln1 column using the linefromtext function.

```
insert into LINESTRING_TEST
values (linefromtext(linestring(10.01 20.03,20.94 21.34,35.93 19.04),
coordref()..srid(1)))
```

linefromwkb

Linefromwkb takes a well-known binary representation of type linestring and a spatial reference system identity returning a linestring.

Syntax

linefromwkb(wkb blob(1m), srid coordref)

Return type

Linestring

Examples

The following code fragment populates the sewerlines table with the unique id, size class and geometry of each 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 lines capacity. The sewer linestring column store the sewer lines geometry.

```
create table sewerlines (sewer_id   integer,
                         class  integer,
                         sewer  linestring);

/* Create the SQL insert statement to populate the sewer_id, size class
   and the sewer linestring. The question marks are parameter markers that
   indicate the sewer_id, class and sewer geometry values that will be
   retrieved at runtime. */
strcpy (wkb_sql,"insert into sewerlines (sewer_id,class,sewer)
values (?,?, linefromwkb (cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the integer sewer_id value 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 integer class value 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 shape to the third parameter. */
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, sewer_wkb, blob_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

locatealong

Locatelong takes a geometry object and a measure to return as a multipoint the set of points found at the measure

Syntax

locatelong(g1 geometry, m1 double precision)

Return type

Geometry

Examples

The LOCATEALONG_TEST table is create with two columns. The gid column uniquely identifies each row and the g1 geometry column stores sample geometry.

```
create table LOCATEALONG_TEST (gid integer, g1 geometry)
```

The insert statements insert two rows. The first is a multilinestring, while the second is a multipoint.

```
insert into LOCATEALONG_TEST values(
1, 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)),
                                   coordref()..srid(1)))

insert into LOCATEALONG_TEST values(
2, 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),
                 coordref()..srid(1)))
```

In this query the locatealong function is directed to find points whose measure is 6.5. The first row returns a multipoint containing two points. However, the second row returned an empty point. For linear features, (geometry with a dimension greater than 0), locatealong can interpolate the point, however for multipoints the target measure must match exactly.

```
select gid, cast(astext(locatealong (g1,6.5)) as
varchar(96)) "Geometry"
from LOCATEALONG_TEST

GID         Geometry
----------- --------------------------------------------------------------

          1 MULTIPOINT M ( 27.01000000 19.38000000 6.50000000, 27.40000000
22.14000000 6.50000000)
          2 POINT EMPTY

  2 record(s) selected.
```

In this query the locatealong function returns multipoints for both rows. The target measure of 7 matches the measures in both the multilinestring and multipoint source data.

```
select gid,cast(astext(locatealong (g1,7)) as varchar(96)) "Geometry"
from LOCATEALONG_TEST

GID      Geometry

----------- ---------------------------------------------------------------
          1 MULTIPOINT M ( 30.19000000 18.47000000 7.00000000, 30.98000000
 23.98000000 7.00000000)
          2 MULTIPOINT M ( 30.19000000 18.47000000 7.00000000, 30.98000000
 23.98000000 7.00000000)

  2 record(s) selected.
```

locatebetween

Locatebetween takes a geometry object and two measure locations and returns a linestring that represents the set of disconnected paths between the two measure locations

Syntax

locatebetween(g1 geometry, fm double precision, tm double precision)

Return type

Geometry

Examples

The LOCATEBETWEEN_TEST table is created with two columns. The gid integer column uniquely identifies each row, while the g1 multilinestring stores the sample geometry.

create table LOCATEBETWEEN_TEST (gid integer, g1 geometry)

The insert statements insert two rows into the LOCATEBETWEEN_TEST table. The first row is a multilinestring and the second is a multipoint.

```
insert into LOCATEBETWEEN_TEST
values(1,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)),
                      coordref()..srid(1)))

insert into LOCATEBETWEEN_TEST
values(2,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),
                        coordref()..srid(1)))
```

The locatebetween function locates measures lying between measures 6.5 and 7.5 inclusive. The first row returns a multilinestring containing several linestrings. The second row returns a multipoint because the source data was multipoint. When the source data has a dimension of 0 (point or multipoint) an exact match is required.

```
select gid, cast(astext(locatebetween (g1,6.5,7.5))
       as varchar(96)) "Geometry"
from LOCATEBETWEEN_TEST

GID         Geometry
----------- ------------------------------------------------------------------
          1 MULTILINESTRING M ( 27.01000000 19.38000000 6.50000000, 31.19000000
18.47000000 7.00000000,38.09000000 19.61000000 7.50000000),(27.40000000 22.1400
0000 6.50000000, 30.98000000 23.98000000 7.00000000,36.95000000 24.98000000 7.5
0000000)
          2 MULTIPOINT M ( 30.19000000 18.47000000 7.00000000, 30.98000000 23.9
8000000 7.00000000)

  2 record(s) selected.
```

m

M takes a point and returns its measure

Syntax

m(p1 point)

Return type

Double precision

Examples

The M_TEST table is created with the gid integer column which uniquely identifies the row and the pt1 point column that stores the sample geometry.

```
create table M_TEST (gid integer, pt1 point)
```

The insert statements insert a row that contains a point with measures and row that contains a point without measures.

```
insert into M_TEST
values(1, pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into M_TEST
values(2, pointfromtext('point  zm(10.02 20.01 5.0 7.0)',coordref()..srid(1)))
```

In this query the m function list the measure values of the points. Since the first point does not have measures the m function returns a NULL.

```
select gid, m (pt1) "The measure" from M_TEST

GID         The measure
----------- ------------------------
          1                        -
          2   +7.00000000000000E+000

  2 record(s) selected.
```

mlinefromshape

Mlinefromshape takes a shape of type multilinestring and a spatial reference system identity to return a multilinestring.

Syntax

mlinefromshape(s1 blob(1m), srid coordref)

Return type

Multilinestring

Examples

This code fragment populates the WATERWAYS table with a unique id, a name and a water multilinestring.

The WATERWAYS table is created with the id and name columns which identify each stream and river system stored in the table. The water column is a multilinestring since the river and stream systems are often an aggregate of several linestrings.

```
create table WATERWAYS (id        integer,
                        name       varchar(128),
                        water      multilinestring);

/* Create the SQL insert statement to populate the id, name and
   multilinestring. The question marks are parameter markers that
   indicate the id, name and water values that will be retrieved at
   runtime. */
strcpy (shp_sql,"insert into WATERWAYS (id,name,water)
values (?,?, mlinefromshape (cast(? as blob(1m)),
coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer id value 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 varchar name value 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 shape to the third parameter. */
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, water_shape, blob_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

mlinefromtext

Mlinefromtext takes a well-known text representation of type multilinestring and a spatial reference system identity and returns a multilinestring

Syntax

mlinefromtext(wkt varchar(4000), srid coordref)

Return type

Multilinestring

Examples

The MLINESTRING_TEST is created with the gid smallint column which uniquely identifies the row and the ml1 multilinestring column.

```
create table MLINESTRING_TEST (gid smallint, ml1 multilinestring)
```

The insert statement inserts the multilinestring with the mlinefromtext function.

```
insert into MLINESTRING_TEST
values (1, 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)),
                         coordref()..srid(1)))
```

mlinefromwkb

Mlinefromwkb takes a well-known binary representation of type multilinestring and a spatial reference system identity to return a multilinestring.

Syntax

mlinefromwkb(wkb blob(1m), srid coordref)

Return type

Multilinestring

Examples

This code fragment populates the WATERWAYS table with a unique id, a name and a water multilinestring.

The WATERWAYS table is created with the id and name columns which identify each stream and river system stored in the table. The water column is a multilinestring since the river and stream systems are often an aggregate of several linestrings.

```
create table WATERWAYS (id        integer,
                             name       varchar(128),
                             water      multilinestring);

/* Create the SQL insert statement to populate the id, name and
   multilinestring. The question marks are parameter markers that
   indicate the id, name and water values that will be retrieved at
   runtime. */
strcpy (shp_sql,"insert into WATERWAYS (id,name,water)
values (?,?, mlinefromwkb (cast(? as blob(1m)),
coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer id value 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 varchar name value 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 shape to the third parameter. */
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, blob_len, 0, water_shape, blob_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

mpointfromshape

Mpointfromshape takes a shape of type multipoint and a spatial reference system identity to return a multipoint.

Syntax

mpointfromshape(s1 blob(1m), srid coordref)

Return type

Multipoint

Examples

This code fragment populates a biologist's SPECIES_SITINGS table

The SPECIES_SITINGS table is created with three columns. The species and genus columns uniquely identify each row while the sitings multipoint stores the locations of the species sitings.

```
create table SPECIES_SITINGS (species  varchar(32),
                              genus varchar(32),
                              sitings  multipoint);

/* Create the SQL insert statement to populate the species, genus and
   sitings. The question marks are parameter markers that indicate the
   name and water values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into SPECIES_SITINGS (species,genus,sitings)
values (?,?, mpointfromshape (cast(? as blob(1m)),
coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the varchar species value 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 varchar genus value to the second parameter. */
pcbvalue2 = genus_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
   SQL_CHAR, genus_len, 0, name, genus_len, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = blob_len;
```

```
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, sitings_len, 0, sitings_shape, sitings_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

mpointfromtext

Mpointfromtext takes a well-known text representation of type multipoint and a spatial reference system identity and returns a multipoint

Syntax

mpointfromtext(wkt blob(1m), srid coordref)

Return type

Multipoint

Examples

The MULTIPOINT_TEST table is created with the single multipoint mpt1 column.

```
create table MULTIPOINT_TEST (mpt1 multipoint)
```

The insert statement inserts a multipoint into the mpt1 column using the mpointfromtext column.

```
insert into MULTIPOINT_TEST
values (1,mpointfromtext(multipoint(10.01 20.03,10.52 40.11,
                                    30.29 41.56,31.78 10.74),
                         coordref()..srid(1)))
```

mpointfromwkb

Mpointfromwkb takes a well-known binary representation of type multipoint and a spatial reference system identity to return a multipoint.

Syntax

mpointfromwkb(wkb blob(1m), srid coordref)

Return type

Multipoint

Examples

This code fragment populates a biologist's SPECIES_SITINGS table.

The SPECIES_SITINGS table is created with three columns. The species and genus columns uniquely identify each row while the sitings multipoint stores the locations of the species sitings.

```
create table SPECIES_SITINGS (species  varchar(32),
                              genus varchar(32),
                              sitings  multipoint);

/* Create the SQL insert statement to populate the species, genus and
   sitings. The question marks are parameter markers that
   indicate the species, genus and sitings values that will be retrieved at
   runtime. */
strcpy (wkb_sql,"insert into SPECIES_SITINGS (species,genus,sitings)
values (?,?, mpointfromwkb (cast(? as blob(1m)),
coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the varchar species value 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 varchar genus value to the second parameter. */
pcbvalue2 = genus_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
   SQL_CHAR, genus_len, 0, &name, genus_len, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = sitings_len;
```

```
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, sitings_len, 0, sitings_wkb, sitings_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

mpolyfromshape

Mpolyfromshape takes a shape of type multipolygon and a spatial reference system identity to return a multipolygon.

Syntax

mpolyfromshape(s1 blob(1m), srid coordref)

Return type

Multipolygon

Examples

This code fragment populates the LOTS table.

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   multipolygon);

/* Create the SQL insert statement to populate the lot_id, and lot. The
   question marks are parameter markers that indicate the lot_id, and lot
   values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into LOTS (lot_id,lot)
values (?, mpolyfromshape (cast(? as blob(1m)),
coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the lot_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
   SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);

/* Bind the lot shape to the second parameter. */
pcbvalue2 = lot_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, lot_len, 0, lot_shape, lot_len, &pcbvalue2);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

mpolyfromtext

Mpolyfromtext takes a well-known text representation of type multipolygon and a spatial reference system identity and returns a multipolygon

Syntax

mpolyfromtext(wkt varchar(4000), srid coordref)

Return type

Multipolygon

Examples

The MULTIPOLYGON_TEST table is created with the single multipolygon mpl1 column.

```
create table MULTIPOLYGON_TEST (mpl1 multipolygon)
```

The insert statement inserts the a multipolygon into the mp11 column using the mpolyfromtext function.

```
insert into MULTIPOLYGON_TEST values (
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))),
coordref()..srid(1)))
```

mpolyfromwkb

Mpolyfromwkb takes a well-known binary representation of type multipolygon and a spatial reference system identity to return a multipolygon.

Syntax

mpolyfromwkb(wkb blob(1m), srid coordref)

Return type

Multipolygon

Examples

This code fragment populates the LOTS table.

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   multipolygon);

/* Create the SQL insert statement to populate the lot_id, and lot. The
   question marks are parameter markers that indicate the lot_id, and lot
   values that will be retrieved at runtime. */
strcpy (wkb_sql,"insert into LOTS (lot_id,lot)
values (?, mpolyfromwkb (cast(? as blob(1m)),
coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the lot_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
     SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);

/* Bind the lot shape to the second parameter. */
pcbvalue2 = lot_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, lot_len, 0, lot_wkb, lot_len, &pcbvalue2);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

numgeometries

Numgeometries takes a collection and returns the number of geometries in the collection.

Syntax

numgeometries(mpt1 multipoint)
numgeometries(mln1 multilinestring)
numgeometries(mpl1 multipolygon)

Return type

Integer

Examples

The city engineer needs to know the actual 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   multipolygon);
```

The query lists the building_id that uniquely identifies each building and the number of buildings in each footprint with the numgeometries function.

```
select building_id, numgeometries (footprint) "Number of buildings"
from BUILDINGFOOTPRINTS;
```

numinteriorrings

Numinteriorrings takes a polygon and returns the number of its interior rings

Syntax

numinteriorrings(pl1 polygon)

Return type

Integer

Examples

An ornithologist, wishing to study a bird population on several south sea islands, knows that the feeding zone of the bird species she is interested in is restricted to islands containing fresh water lakes. Therefore she wants to know which islands contain one or more lakes.

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 polygon);
```

Since interior rings represents the lakes the numinteriorrings function is used to list only those islands that have at list one interior ring.

```
select name from ISLANDS where numinteriorrings(land) > 0;
```

numpoints

Numpoints takes a linestring and returns its number of points

Syntax

numpoints(ln1 linestring)

Return type

Integer

Examples

The NUMPOINTS_TEST table is created with the geotype column which contains the geometry type stored in the g1 geometry column.

```
create table NUMPOINTS_TEST (geotype varchar(12), g1 geometry)
```

The insert statements insert point, linestring and polygon.

```
insert into NUMPOINTS_TEST values(point,
pointfromtext(point (10.02 20.01),coordref()..srid(1)))

insert into NUMPOINTS_TEST values( linestring,
linefromtext(linestring (10.02 20.01, 23.73 21.92),coordref()..srid(1)))

insert into NUMPOINTS_TEST values(
   polygon,
   polyfromtext(polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64
   13.42, 10.02 20.01)),coordref()..srid(1)))
```

The query lists the geometry type and the number of points contained within each.

```
select geotype, numpoints(g1)
from NUMPOINTS_TEST

GEOTYPE      Number of points
------------ ----------------
point                       1
linestring                  2
polygon                     5

  3 record(s) selected.
```

overlap

Overlap takes two geometry objects and returns 1 (TRUE) if the intersection of the objects results in a geometry object of the same dimension but not equal to either source objects, otherwise it returns 0 (FALSE)

Syntax

overlap(g1 geometry, g2 geometry)

Return type

Integer

Examples

The County Supervisor needs a list of hazardous wastes 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's polygon geometry.

```
create table SENSITIVE_AREAS (id        integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      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(128),
                              location  point);
```

The SENSITIVE_AREAS and HAZARDOUS_SITES table are joined by the **overlap** function. It returns 1 (TRUE) for all rows in the SENSITIVE_AREAS table, whose zone polygons overlap the buffered 5-mile radius of the HAZARDOUS_SITES location point.

```
select hs.name
from HAZARDOUS_SITES hs, SENSITIVE_AREAS sa
where overlap (buffer(hs.location,(5 * 5280)),sa.zone) = 1;
```

In Figure 36 on page 153, the hospital and the school overlap the 5-mile radius of the counties two hazardous waste sites, while the nursing home does not.

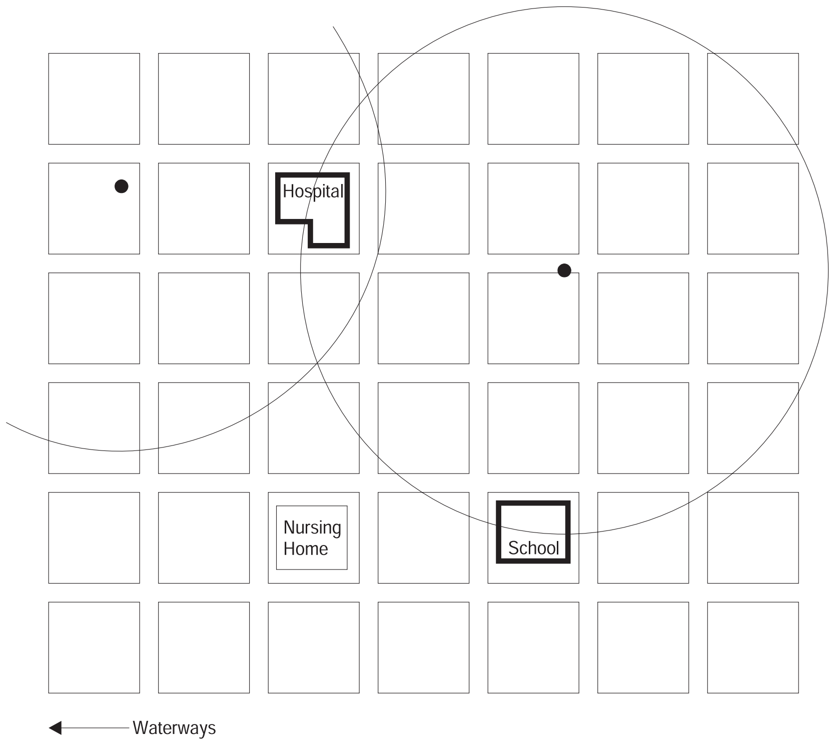


Figure 36. Using overlap to determine the buildings that are at least partially within of a hazardous waste area

pointfromshape

Pointfromshape takes a shape of type point and a spatial reference system identity to return a point.

Syntax

pointfromshape(s1 blob(1m),srid coordref)

Return type

Point

Examples

The program fragment populates the HAZARDOUS_SITES table.

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(128),
                              location  point);

/* Create the SQL insert statement to populate the site_id, name and
   location. The question marks are parameter markers that indicate the
   site_id, name and location values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into HAZARDOUS_SITES (site_id, name, location)
values (?,?, pointfromshape (cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the site_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
   SQL_INTEGER, 0, 0, &site_id, 0, &pcbvalue1);

/* Bind the name varchar value 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 shape to the third parameter. */
pcbvalue3 = location_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, location_len, 0, location_shape, location_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

pointfromtext

Pointfromtext takes a well-known text representation of type point and a spatial reference system identity and returns a point

Syntax

pointfromtext(wkt varchar(4000), srid coordref)

Return type

Point

Examples

The POINT_TEST table is created with the single point column pt1.

create table POINT_TEST (pt1 point)

The pointfromtext function converts the point text coordinates to the $se;'s point format before the insert statement inserts the point into the pt1 column.

```
insert into POINT_TEST values (
       pointfromtext (point(10.01 20.03),coordref()..srid(1)))
```

pointfromwkb

Pointfromwkb takes a well-known binary representation of type point and a spatial reference system identity to return a point.

Syntax

pointfromwkb(wkb blob(1m), srid coordref)

Return type

Point

Examples

The program fragment populates the HAZARDOUS_SITES table.

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(128),
                              location  point);

/* Create the SQL insert statement to populate the site_id, name and
   location. The question marks are parameter markers that indicate the
   site_id, name and location values that will be retrieved at runtime. */
strcpy (wkb_sql,"insert into HAZARDOUS_SITES (site_id, name, location)
values (?,?, pointfromwkb(cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the site_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
     SQL_INTEGER, 0, 0, &site_id, 0, &pcbvalue1);

/* Bind the name varchar value 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 shape to the third parameter. */
pcbvalue3 = location_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, location_len, 0, location_wkb, location_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

pointn

Pointn takes a linestring and an integer index and returns a point that is the nth vertice in the linestrings path

Syntax

pointn(ln1 linestring, index integer)

Return type

Point

Examples

The POINTN_TEST table is create with the gid column which uniquely identifies each row and the ln1 linestring column.

```
create table POINTN_TEST (gid integer, ln1 linestring)
```

The 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,
linefromtext('linestring (10.02 20.01,23.73 21.92,30.10 40.23)',coordref()..srid(1)))

insert into POINTN_TEST values(2,
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)',coordref()..srid(1)))
```

The query lists the gid column and the second vertice of each linestring. The first row results in a point without a Z coordinate or measure while the second row results in a point with a Z coordinate and a measure. The pointn function will return points with a Z coordinate or a measure if they exist in the source linestring.

```
select gid, cast(astext(pointn(ln1,2)) as varchar(60)) "The 2nd vertice"
from POINTN_TEST

GID         The 2nd vertice
----------- ------------------------------------------------------------
          1 POINT ( 23.73000000 21.92000000)
          2 POINT ZM ( 23.73000000 21.92000000 7.00000000 7.10000000)

  2 record(s) selected.
```

pointonsurface

pointonsurface takes a polygon or multipolygon and returns a point guaranteed to lie on its surface

Syntax

pointonsurface(pl1 polygon)
pointonsurface(mpl1 multipolygon)

Return type

Point

Examples

The city engineer wants to create a label point for each of the building footprints.

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   multipolygon);
```

The pointonsurface function generates a point that is guaranteed to be on the surface of the building footprints. The pointonsurface function returns a point that the asbinaryshape function converts to a shape casted to 1 megabyte character string for use by the application.

```
select cast(asbinaryshape(pointonsurface(footprint)) as blob(1m))
from BUILDINGFOOTPRINTS;
```

polyfromshape

Polyfromshape takes a shape of type polygon and a spatial reference system identity to return a polygon.

Syntax

polyfromshape(s1 blob(1m), srid coordref)

Return type

Polygon

Examples

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 runtime.

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the zone column which stores the institution's polygon geometry.

```
create table SENSITIVE_AREAS (id        integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      polygon);

/* Create the SQL insert statement to populate the id, name, size, type and
   zone. The question marks are parameter markers that indicate the
   id, name, size, type and zone values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into SENSITIVE_AREAS (id, name, size, type, zone)
values (?,?,?,?, polyfromshape (cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
   SQL_INTEGER, 0, 0, &site_id, 0, &pcbvalue1);
/* Bind the name varchar value 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 float 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 varchar 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 polygon to the fifth parameter. */
pcbvalue5 = zone_len;
rc = SQLBindParameter (hstmt, 5, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, zone_len, 0, zone_shp, zone_len, &pcbvalue5);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

polyfromtext

Polyfromtext takes a well-known text representation of type polygon and a spatial reference system identity and returns a polygon

Syntax

polyfromtext(wkt varchar(4000), srid coordref)

Return type

Polygon

Examples

The POLYGON_TEST table is created with the single polygon column.

```
create table POLYGON_TEST (pl1 polygon)
```

The insert statement inserts a polygon into the polygon column using the polygonfromtext function.

```
insert into POLYGON_TEST values (1,
polyfromtext(polygon((10.01 20.03,10.52 40.11,30.29 41.56,31.78 10.74,10.01
20.03)),coordref()..srid(1)))
```

polyfromwkb

Polyfromwkb takes a well-known binary representation of type polygon and a spatial reference system identity to return a polygon.

Syntax

polyfromwkb(wkb blob(1m), srid coordref)

Return type

Polygon

Examples

The program fragment populates the SENSITIVE_AREAS table.

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the zone column which stores the institution’s polygon geometry.

```
create table SENSITIVE_AREAS (id       integer,
                              name     varchar(128),
                              size     float,
                              type     varchar(10),
                              zone     polygon);


/* Create the SQL insert statement to populate the id, name, size, type and
   zone. The question marks are parameter markers that indicate the id,name,
   size, type and zone values that will be retrieved at runtime. */
strcpy (shp_wkb,"insert into SENSITIVE_AREAS (id, name, size, type, zone)
values (?,?,?,?, polyfromwkb (cast(? as blob(1m)),coordref()..srid(1)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
   SQL_INTEGER, 0, 0, &id, 0, &pcbvalue1);

/* Bind the name varchar value 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 float 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 varchar to the fourth parameter. */
pcbvalue4 = type_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR,
     SQL_VARCHAR, type_len, 0, type, type_len, &pcbvalue4);

/* Bind the zone polygon to the fifth parameter. */
pcbvalue5 = zone_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BLOB, zone_len, 0, zone_wkb, zone_len, &pcbvalue5);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

srid

Srid takes a geometry object and returns its spatial reference system identity

Syntax

srid (g1 geometry)

Return type

Integer

Examples

á

During the installation of the Spatial Extender the SPATIAL_REFERENCES table is created. When a geometry is created, the SRID of that geometry is entered into the SPATIAL_REFERENCES table. The **srid** function returns the value of that entry.

For example, a geometry type is used in a create table statement.

```
create table SRID_TEST(g1 geometry)
```

In the next statement, a point geometry located at coordinate 10.01,50.76 is inserted into the geometry column g1. When the point geometry was created by the pointfromtext function it was assigned the srid value of 1.

```
insert into SRID_TEST
values (pointfromtext(point(10.01 50.76),coordref()..srid(1)))
```

The srid function returns the spatial reference system identity of the geometry just entered.

```
select srid(g1) from SRID_TEST

g1
--------------
1
```

startpoint

Startpoint takes a linestring and returns a point that is the linestrings first point

Syntax

startpoint(ln1 linestring)

Return type

Point

Examples

The STARTPOINT_TEST table is created with the gid integer column which uniquely identifies the rows of the table and the ln1 linestring column.

```
create table STARTPOINT_TEST (gid integer, ln1 linestring)
```

The insert statements insert the linestrings into the ln1 column. The first linestring does not have Z coordinates or measures while the second linestring has both.

```
insert into STARTPOINT_TEST values(1,
linefromtext('linestring (10.02 20.01,23.73 21.92,30.10 40.23)',coordref()..srid(1)))

insert into STARTPOINT_TEST values(2,
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)',coordref()..srid(1)))
```

The startpoint function extracts the first point of each linestring. The astext function converts the point to its text format. 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 did.

```
select gid, cast(astext(startpoint (ln1)) as varchar(60)) "Startpoint"
from STARTPOINT_TEST

GID         Startpoint
----------- ------------------------------------------------------------
          1 POINT ( 10.02000000 20.01000000)
          2 POINT ZM ( 10.02000000 20.01000000 5.00000000 7.00000000)

  2 record(s) selected.
```

symmetricdiff

symmetricdiff takes two geometry objects and returns a geometry object that is the symmetrical difference of the source objects

Syntax

symmetricdiff(g1 geometry, g2 geometry)

Return type

Geometry

Examples

For a special report, the county supervisor must determine the area of sensitive areas and 5-mile hazardous site radius that is not intersected.

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the zone column which stores the institution's polygon geometry.

```
create table SENSITIVE_AREAS (id        integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      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(128),
                              location  point);
```

The buffer function generates a 5-mile buffer surrounding the hazardous waste site locations. The symmetricdiff function generates polygons from the intersection of the buffered hazardous waste site polygons and the sensitive areas. The area function returns the intersection polygon;s area for each hazardous site.

```
select sa.name, hs.name,
       area(symmetricdiff (buffer(hs.location,(5 * 5280)),sa.zone))
from HAZARDOUS_SITES hs, SENSITIVE_AREAS sa
```

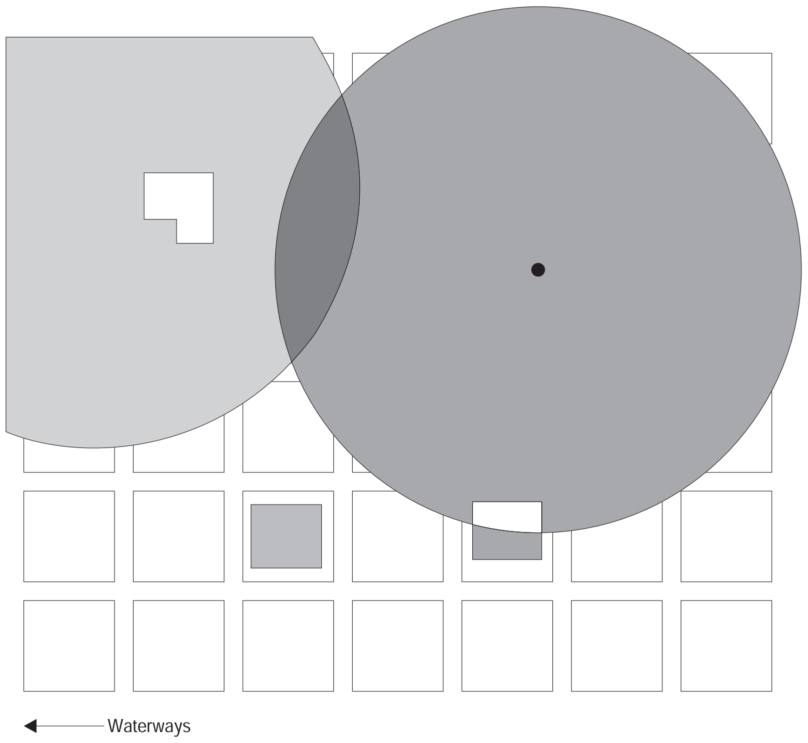


Figure 37. Using symmetricdiff to determined the hazardous waste areas that don't contain sensitive areas (inhabited buildings)

In Figure 37, the symmetric difference of the hazardous waste sites and the sensitive areas results in the subtraction of the intersected areas.

touch

Touch returns 1 (TRUE) if none of the points common to both geometries intersect the interiors of both geometries. Otherwise, it returns 0 (FALSE). At least one geometry must be a linestring, polygon, multilinestring or multipolygon.

Syntax

touch(g1 geometry, g2 geometry)

Return type

Integer

Examples

The GIS technician has been asked by his boss to provide a list of all sewer lines whose endpoints intersect another sewerline.

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 lines capacity. The sewer linestring column stores the sewer lines geometry.

```
create table sewerlines (sewer_id integer, class integer, sewer linestring);
```

The query returns an ordered list of SEWER_IDS that touch one another.

```
select s1.sewer_id, s2.sewer_id
from sewerlines s1, sewerlines s2
where touch (s1.sewer, s2.sewer) = 1,
order by 1,2;
```

union

Union takes two geometry objects and returns a geometry object that is the union of the source objects.

Syntax

union(g1 geometry, g2 geometry)

Return type

Geometry

Examples

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the zone column which stores the institution’s polygon geometry.

```
create table SENSITIVE_AREAS (id        integer,
                              name      varchar(128),
                              size      float,
                              type      varchar(10),
                              zone      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(128), location point);
```

The buffer function generates a 5-mile buffer surrounding the hazardous waste site locations. The union function generates polygons from the union of the buffered hazardous waste site polygons and the sensitive areas. The area function returns the unioned polygon;s area.

```
select sa.name, hs.name,
       area(union(buffer(hs.location,(5 * 5280)),sa.zone))
from HAZARDOUS_SITES hs, SENSITIVE_AREAS sa;
```

within

Within takes two geometry objects and returns 1 (TRUE) if the first object is completely within the second, otherwise it returns 0 (FALSE).

Syntax

within(g1 geometry, g2 geometry)

Return type

Integer

Examples

In the example below two tables are created. One, 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 multipolygon data type stores the geometry of the building footprints and the lots. The database designer selected multipolygons for both features because she realized that often lots can be disjointed by natural features such as a river and building footprints can often be made up of several buildings.

```
create table BUILDINGFOOTPRINTS (   building_id integer,
                                    lot_id      integer,
                                    footprint   multipolygon);


create table LOTS (  lot_id integer, lot multipolygon );
```

The city engineer first selects the buildings that are not completely within a lot.

```
select building_id
  from BUILDINGFOOTPRINTS, LOTS
 where within(footprint,lot) = 0;
```

The city engineer is smart. She realizes that although the first query will provide her with a list of all building_id that have footprints outside of a lot polygon, it will not tell her 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 "building id",
       bf.lot_id "buildings lot_id",
       LOTS.lot_id "LOTS lot_id"
  from BUILDINGFOOTPRINTS bf, LOTS
 where within(footprint,lot) = 1 AND
       LOTS.lot_id <> bf.lot_id;
```

x

X takes a point and returns its X coordinate

Syntax

x(pt1 point)

Return type

Double precision

Examples

The X_TEST table is created with two columns: the gid column uniquely identifies the row, and the pt1 point column.

```
create table X_TEST (gid integer, pt1 point)
```

The 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,
pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into X_TEST values(2,
pointfromtext('point  zm (10.02 20.01 5.0 7.0)',coordref()..srid(1)))
```

The query lists the gid column and the double precision X coordinate of the points.

```
select gid, x(pt1) "The X coordinate" from X_TEST

GID         The X coordinate
----------- ------------------------
          1   +1.00200000000000E+001
          2   +1.00200000000000E+001

  2 record(s) selected.
```

y

Y takes a point and returns its Y coordinate

Syntax

y(p1 point)

Return type

Double precision

Examples

The Y_TEST table is created with two columns: the gid column uniquely identifies the row, and the pt1 point column.

```
create table Y_TEST (gid integer, pt1 point)
```

The 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 Y_TEST values(1,
pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into Y_TEST values(2,
pointfromtext('point  zm (10.02 20.01 5.0 7.0)',coordref()..srid(1)))
```

The query lists the gid column and the double precision Y coordinate of the points.

```
select gid, y(pt1) "The Y coordinate" from Y_TEST

GID         The Y coordinate
----------- ------------------------
          1   +2.00100000000000E+001
          2   +2.00100000000000E+001

  2 record(s) selected.
```

z

Z takes a point and returns its Z coordinate

Syntax

z(p1 point)

Return type

Double precision

Examples

The Z_TEST table is created with two columns: the gid column uniquely identifies the row, and the pt1 point column.

```
create table Z_TEST (gid integer, pt1 point)
```

The 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 Z_TEST values(1,
pointfromtext('point (10.02 20.01)',coordref()..srid(1)))

insert into Z_TEST values(2,
pointfromtext('point  zm (10.02 20.01 5.0 7.0)',coordref()..srid(1)))
```

The query lists 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, z(pt1) "The Z coordinate" from Z_TEST

GID         The Z coordinate
----------- ------------------------
          1                        -
          2   +5.00000000000000E+000

  2 record(s) selected.
```

Chapter 6. Messages

38600 Invalid Well-Known Text representation string

Explanation: The text string entered in the Well-Known Text representation function is invalid.

User Response: Correct the string and execute the function again. Refer to “Appendix B. The OGIS Well-Known Text Representation” on page 187 for the valid format.

38601 Invalid Spatial Reference Identifier

Explanation: The Spatial Reference Identifier (SRID) supplied with the geometry does not exist in the SPATIAL_REFERENCES table.

User Response: Either use an SRID that is currently in the SPATIAL_REFERENCES table or add a spatial reference system to the table that corresponds to the rejected SRID.

38602 System has run out of memory

Explanation: Not enough memory was available. The Spatial Extender requires up to a maximum of 1 Megabyte of memory to create features.

User Response: Reallocate memory to make more available to the Spatial Extender, or if this is not possible upgrade the system by adding more physical memory.

38603 The spatial reference systems of two geometries are not same.

Explanation: Geometries passed to a Spatial Extender function must share the same spatial reference identifier.

User Response: Recreate one of the geometries so that its spatial reference system matches that of the other.

38604 The binary string is invalid

Explanation: The Well-Known Binary representation or ESRI Binary representation string was not constructed properly.

User Response: Reconstruct the string with the correct format. Refer to “Appendix C. The OGIS Well-Known Binary Representation” on page 193 or “Appendix D. The ESRI Shape Representations” on page 197 for the correct format.

38605 The geometrytype is invalid

Explanation: An invalid geometry type was passed to the function. Valid geometry types are geometry, point, linestring, polygon, multipoint, multilinestring, or multipolygon.

User Response: Resubmit the SQL statement with a valid geometry type.

38606 Parenthesis mismatch

Explanation: The parentheses of the Well-Known Text representation string do not match.

User Response: Balance the parentheses and reenter the text description.

38607 Too many parts specified

Explanation: The number of parts indicated in the binary or text string is greater than the actual number of parts supplied.

User Response: Reenter the string with the correct number of parts.

38608 Geometry type mismatch

Explanation: The wrong geometry type was passed to the function. For instance **polygonfromtext** expects polygon and was passed linestring.

User Response: Use a function that accepts the geometry type.

38609 Text string is too long

Explanation: The geometry text string exceeds the maximum length of 4000 characters.

User Response: The geometry contains too much detail to be converted to text. However, you can instead convert it to either the WKB or the ESRI shape binary formats.

38610 Invalid parameter value

Explanation: An invalid parameter was passed to the function.

User Response: Compare the syntax of the function with that listed in "Chapter 5. SQL Reference" on page 67. Correct the invalid parameter and resubmit the function.

38612 Invalid grid size

Explanation: Returned by CREATE INDEX if any grid size is less than 0, the first grid size is equal 0, the second grid size is less than the first or the third grid size is less than the second.

User Response: Make sure all of the grid sizes are correct and resubmit the CREATE INDEX statement.

38613 The grid size is too small

Explanation: Returned by CREATE INDEX if a grid size results in more than 1000 grid cells per geometry.

User Response: Increase the grid size or add another grid level. Then, resubmit the CREATE INDEX statement.

38800 Invalid geometry

Explanation: The parameters entered have produced an invalid geometry. For example, the parameters entered with **linefromshape** produce an invalid geometry. An invalid geometry is one that violates a geometry property.

User Response: Correct the parameter and resubmit the geometry.

38801 Incompatible geometry

Explanation: The function expected two geometries of a certain type and did not receive them. For example, the **union** function expects two geometries of the same dimension and received a point and a linestring, which are of different dimensions.

User Response: Refer to Chapter 6, SQL Reference, and resubmit the function with the correct geometry types.

38802 Geometry integrity error

Explanation: The function cannot construct a geometry because one or more of its properties have been violated.

User Response: Refer to "About Geometry" on page 37 or "Chapter 5. SQL Reference" on page 67 , and resubmit the geometry with the correct properties.

38803 Too many points

Explanation: The construction of a geometry has exceeded the 1 MB storage limit; the geometry has too many points.

User Response: Remove unnecessary points. For performance and storage considerations, only those points needed to render a geometry should be included. All non-essential points should be excluded.

38804 Result is too small

Explanation: The resulting polygons of the **intersections**, **union**, **difference** or **symmetricdiff** functions are too small to be represented with the current coordinate system.

User Response: If a result is absolutely required, increase the XYUNITS of the source geometrys coordinate reference system., and recreate the table. Changing the coordinate reference system requires that the table containing the source geometry be recreated.

38805 Buffer out of bounds

Explanation: The buffer function has created a buffer outside the coordinate system.

User Response: Either reduce the buffer distance or change the source geometry's coordinate system. In most cases changing the coordinate system requires the spatial column be reloaded.

38806 Invalid system units

Explanation: The system units (XYUNITS, ZUNITS or MUNITS of the SPATIAL_REFERENCES table) cannot be less than 1.

User Response: Correct any of the system unit values in the SPATIAL_REFERENCES table that are less than 1. (See "Meta Tables and Views" on page 19 .)

38807 Ordinate out of bounds

Explanation: An ordinate is either too large or too small to fit within the bounds of the coordinate system.

User Response: Ensure the ordinates value is correct and if it is, ensure the coordinate system can accept the ordinate value. (See "Meta Tables and Views" on page 19.)

38811 Polygon rings overlap

Explanation: The rings of a polygon cannot overlap, but they may intersect at a tangent.

User Response: Correct the coordinates of the polygon and resubmit it.

38812 Too few coordinates

Explanation: Linestring geometries require at least two coordinates and polygons require three.

User Response: Resubmit the geometry with the correct number of coordinates.

38813 Polygon is not closed

Explanation: The start and end point coordinates of the polygon are not the same.

User Response: Edit the coordinate list of the polygon, making sure the start and end points are the same, and resubmit it.

38814 Exterior ring is invalid

Explanation: The exterior ring does not enclose the interior ring. The interior ring is completely outside the exterior ring with no overlap.

User Response: Make sure the coordinates of the interior ring are completely inside the exterior ring. If the interior ring actually represents the exterior ring of another polygon, then enter the geometry as a multipolygon. Correct the geometry and resubmit it.

38815 The polygon has no area

Explanation: A geometry is a polygon only if its coordinates span two dimensions in space.

User Response: Edit the coordinates of the polygon so they enclose an area and resubmit the polygon. Or, submit a linestring if appropriate.

38816 The polygon contains a spike

Explanation: Only the end point and start point of a polygon must be same. All other coordinates of a polygon ring must be different and collectively enclose an area.

User Response: Look for coordinate pairs that have the same X and Y values. Edit these points so that the polygon encloses a single area and resubmit the polygon.

38817 Multipolygon exterior rings overlap

Explanation: The exterior rings of a multipolygon may intersect at a tangent, but they cannot overlap.

User Response: Edit the coordinates of the exterior rings so that they do not overlap, then resubmit the multipolygon.

38818 Polygon self intersects

Explanation: The ring of a polygon cannot intersect itself.

User Response: Edit the coordinates of the ring that intersects itself and resubmit the polygon.

38819 Invalid number of measures

Explanation: The *number of measures* parameter of the binary string contains a different number than was supplied.

User Response: Edit the *number of measures* parameter so that it corresponds to the number supplied in the binary string.

38820 Invalid number of parts

Explanation: The *number of parts* parameter of the binary string specified a number of parts different than what has been supplied with the string.

User Response: Edit the *number of parts* parameter so that it corresponds to the number supplied in the binary string.

38821 Invalid part offset

Explanation: The *part offset* parameter of the binary string specified a part offset different than what has been supplied within the string.

User Response: Edit the *part offset* parameter so that it corresponds to the part offsets supplied within the binary string.

38823 Binary too small

Explanation: The number of bytes in the blob specified is less than the number of bytes in the blob supplied.

User Response: Make the blob length equal to the number of bytes in the blob and resubmit the function.

38825 Invalid byte order

Explanation: The byte order must be 0 or 1.

User Response: Edit the byte order so that it is either 0 for little endian or 1 for big endian.

38826 Invalid part

Explanation: A function parameter indexed a part that does not exist. For example, this error would occur if the **geometryn** function was passed a 3 to return the third point in a multipoint, when the multipoint only contains two points.

User Response: Edit the parameter and resubmit the function.

38999 Unknown system failure

Explanation: An unexpected system error has occurred.

User Response: Correct the syntax and resubmit the function. If you still encounter the problem contact technical support.

Appendix A. Representing Spatial Reference Systems as Text

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

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

```
<coordinate system> = <projected cs> | <geographic cs> | <geocentric cs>
<projected cs> = PROJCS["<name>", <geographic cs>, <projection>, {<parameter>,}*
                <linear unit>]
<projection> = PROJECTION["<name>"]
<parameter> = PARAMETER["<name>", <value>]
<value> = <number>
```

A data set’s coordinate system is identified by the PROJCS keyword if the data are in projected coordinates, by GEOGCS if in geographic coordinates, or by GEOCCS if in geocentric coordinates. The PROJCS keyword is followed by all of the ″pieces″ which define the projected coordinate system. The first piece of any object is always the name. Several objects follow the projected coordinate system name: the geographic coordinate system, the map projection, 1 or more parameters, and the linear unit of measure. All projected coordinate systems are based upon a geographic coordinate system so we will describe the pieces specific to a projected coordinate system first. As an example, UTM zone 10N on the NAD83 datum is defined as:

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

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

```
<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>
    (semi-major axis is measured in meters and must be > 0.)
<inverse flattening> = <number>
<prime meridian> = PRIMEM["<name>", <longitude>]
<longitude> = <number>
```

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

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

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

```
<angular unit> = <unit>
<linear unit> = <unit>
<unit> = UNIT["<name>", <conversion factor>]
<conversion factor> = <number>
```

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

So the full string representation of UTM Zone 10N is as follows:

```
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:

```
<geocentric cs> = GEOCCS["<name>", <datum>, <prime meridian>, <linear unit>]
```

Supported Linear Units

Unit	Conversion Factor
Meter	1.0
Foot (International)	0.3048
U.S. Foot	12/39.37
Modified American Foot	12.0004584/39.37
Clarke's Foot	12/39.370432
Indian Foot	12/39.370141
Link	7.92/39.370432

Unit	Conversion Factor
Link (Benoit)	7.92/39.370113
Link (Sears)	7.92/39.370147
Chain (Benoit)	792/39.370113
Chain (Sears)	792/39.370147
Yard (Indian)	36/39.370141
Yard (Sears)	36/39.370147
Fathom	1.8288
Nautical Mile	1852.0

Supported Angular Units

Unit	Conversion Factor
Radian	1.0
Decimal Degree	p/180
Decimal Minute	(p/180)/60
Decimal Second	(p/180)/36000
Gon	p/200
Grad	p/200

Supported Spheroids

Name	Semi-major Axis	Inverse Flattening
Airy	6377563.396	299.3249646
Modified Airy	6377340.189	299.3249646
Australian	6378160	298.25
Bessel	6377397.155	299.1528128
Modified Bessel	6377492.018	299.1528128
Bessel (Namibia)	6377483.865	299.1528128
Clarke 1866	6378206.4	294.9786982
Clarke 1866 (Michigan)	6378693.704	294.978684677
Clarke 1880	6378249.145	293.465
Clarke 1880 (Arc)	6378249.145	293.466307656
Clarke 1880 (Benoit)	6378300.79	293.466234571
Clarke 1880 (IGN)	6378249.2	293.46602
Clarke 1880 (RGS)	6378249.145	293.465
Clarke 1880 (SGA)	6378249.2	293.46598
Everest 1830	6377276.345	300.8017

Name	Semi-major Axis	Inverse Flattening
Everest 1975	6377301.243	300.8017
Everest (Sarawak and Sabah)	6377298.556	300.8017
Modified Everest 1948	6377304.063	300.8017
Fischer 1960	6378166	298.3
Fischer 1968	6378150	298.3
Modified Fischer (1960)	6378155	298.3
GEM10C	6378137	298.257222101
GRS 1980	6378137	298.257222101
Hayford 1909	6378388	297.0
Helmert 1906	6378200	298.3
Hough	6378270	297.0
International 1909	6378388	297.0
International 1924	6378388	297.0
New International 1967	6378157.5	298.2496
Krasovsky	6378245	298.3
Mercury 1960	6378166	298.3
Modified Mercury 1968	6378150	298.3
NWL9D	6378145	298.25
OSU_86F	6378136.2	298.25722
OSU_91A	6378136.3	298.25722
Plessis 1817	6376523	308.64
South American 1969	6378160	298.25
Southeast Asia	6378155	298.3
Sphere (radius = 1.0)	1	0
Sphere (radius = 6371000 m)	6371000	0
Sphere (radius = 6370997 m)	6370997	0
Struve 1860	6378297	294.73
Walbeck	6376896	302.78
War Office	6378300.583	296
WGS 1960	6378165	298.3
WGS 1966	6378145	298.25
WGS 1972	6378135	298.26
WGS 1984	6378137	298.257223563

Supported Geodetic Datums

Adindan	Lisbon

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

Fahud	South American Datum 1969
Gandajika 1970	Sapper Hill 1943
Garoua	Schwarzeck
Geocentric Datum of Australia 1994	Segora
Guyane Francaise	Serindung
Herat North	Stockholm 1938
Hito XVIII 1963	Sudan
Hu Tzu	Shan Tananarive 1925
Hungarian Datum 1972	Timbalai 1948
Indian 1954	TM65
Indian 1975	TM75
Indonesian Datum 1974	Tokyo
Jamaica 1875	Trinidad 1903
Jamaica 1969	Trucial Coast 1948
Kalianpur	Voirol 1875
Kandawala	Voirol Unifie 1960
Kertau	WGS 1972
Kuwait Oil Company	WGS 1972 Transit Broadcast Ephemeris
La Canoa	WGS 1984
Lake	Yacare
Leigon	Yoff
Liberia 1964	Zanderij

Supported Prime Meridians

Greenwich	0° 0’ 0″
Bern	7° 26’ 22.5″ E
Bogota	74° 4’ 51.3″ W
Brussels	4° 22’ 4.71″ E
Ferro	17° 40’ 0″ W
Jakarta	106° 48’ 27.79″ E
Lisbon	9° 7’ 54.862″ W
Madrid	3° 41’ 16.58″ W
Paris	2° 20’ 14.025″E
Rome	12° 27’ 8.4″ E
Stockholm	18° 3’ 29″ E

Supported Map Projections

Cylindrical Projections	Pseudocylindrical Projections
Behrmann	Craster parabolic
Cassini	Eckert I
Cylindrical equal area	Eckert II
Equirectangular	Eckert III
Gall's stereographic	Eckert IV
Gauss-Kruger	Eckert V
Mercator	Eckert VI
Miller cylindrical	McBryde-Thomas flat polar quartic
Oblique	Mercator (Hotine) Mollweide
Plate-Carée	Robinson
Times	Sinusoidal (Sansom-Flamsteed)
Transverse Mercator	Winkel I

Conic Projections

Albers conic equal-area	Chamberlin trimetric
Bipolar oblique conformal conic	Two-point equidistant
Bonne	Hammer-Aitoff equal-area
Equidistant conic	Van der Grinten I
Lambert conformal conic	Miscellaneous
Polyconic	Alaska series E
Simple conic	Alaska Grid (Modified-Stereographic by Snyder)

Azimuthal or Planar Projections

- Azimuthal equidistant
- General vertical near-side perspective
- Gnomonic
- Lambert Azimuthal equal-area
- Orthographic
- Polar Stereographic
- Stereographic

Map Projection Parameters

central_meridian	the line of longitude chosen as the origin of x-coordinates.
scale_factor	used generally to reduce the amount of distortion in a map projection.
standard_parallel_1	a line of latitude that has no distortion generally. Also used for ″latitude of true scale.″
standard_parallel_2	a line of latitude that has no distortion generally.
longitude_of_center	the longitude which defines the center point of the map projection.
latitude_of_center	the latitude which defines the center point of the map projection.
latitude_of_origin	the latitude chosen as the origin of y-coordinates.
false_easting	added to x-coordinates. Used to give positive values.
false_northing	added to y-coordinates. Used to give positive values.
azimuth	the angle east of north which defines the center line of an oblique projection.
longitude_of_point_1	the longitude of the first point needed for a map projection.
latitude_of_point_1	the latitude of the first point needed for a map projection.
longitude_of_point_2	the longitude of the second point needed for a map projection.
latitude_of_point_2	the latitude of the second point needed for a map projection.
longitude_of_point_3	the longitude of the third point needed for a map projection.
latitude_of_point_3	the latitude of the third point needed for a map projection.
landsat_number	the number of a Landsat satellite.
path_number	the orbital path number for a particular satellite.
perspective_point_height	the height above the earth of the perspective point of the map projection.
fipszone	State Plane Coordinate System zone number.
zone	UTM zone number.

Appendix B. The OGIS Well-Known Text Representation

Each geometry type has a well-known text representation that can be used both to construct new instances of the type and to convert existing instances to textual form for alphanumeric display.

The well-known text representation of Geometry is defined below; the notation {}* denotes 0 or more repetitions of the tokens within the braces, the braces do not appear in the output token list.

```
<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 > }*  )
```

The basic function syntax is:

```
function (<text description>,<SRID>)
```

The *SRID*, the spatial reference identifier and primary key to the SPATIAL_REFERENCES table. identifies the geometrys spatial reference system that are stored in the SPATIAL_REFERENCES table. 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 four basic components enclosed in single quotes defined as follows:

<*geometry type*> [*coordinate type*] [*coordinate list*]

geometry type
: one of the following: point, linestring, polygon, multipoint, multilinestring, or multipolygon.

coordinate type
: specifies whether or not the geometry has Z coordinates and/or measures. Leave this argument blank if the geometry does has neither, otherwise set the coordinate type to Z for geometries containing a Z coordinates, M for a geometries with measures, and ZM for geometries that have both.

coordinate list
: defines the vertices of the geometry. Coordinate lists are comma delimited and enclosed by parenthesis. Geometries having multiple components require sets

of parenthesis to enclose each component part. If the geometry is empty, the EMPTY keyword replaces the coordinate.

The following examples provide complete list of all possible permutations of text representations:

Geometry type	Text Description	Comment
point	point empty	empty point
point	point z empty	empty point with z coordinate
point	point m empty	empty point with measure
point	point zm empty	empty point with z coordinate and measure
point	point (10.05 10.28)	point
point	point z (10.05 10.28 2.51)	point with z coordinate
point	point m (10.05 10.28 4.72)	point with measure
point	point zm (10.05 10.28 2.51 4.72)	point with z coordinate and measure
linestring	linestring empty	empty linestring
linestring	linestring z empty	empty linestring with z coordinates
linestring	linestring m empty	empty linestring with measures
linestring	linestring zm empty	empty linestring with z coordinates and measures
linestring	linestring (10.05 10.28 , 20.95 20.89)	linestring
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
linestring	linestring m (10.05 10.28 5.84, 20.95 31.98 9.01, 21.98 29.80 12.84)	linestring with measures
linestring	linestring zm ( )	linestring with z coordinates and measures
polygon	polygon empty	empty polygon
polygon	polygon z empty	empty polygon with z coordinates
polygon	polygon m empty	empty polygon with measures
polygon	polygon zm empty	empty polygon with z coordinates and measures
polygon	polygon ((10 10, 10 20, 20 20, 20 15, 10 10))	polygon
polygon	polygon z (( ))	polygon with z coordinates
polygon	polygon m (( ))	polygon with measures

Geometry type	Text Description	Comment
polygon	polygon zm (( ))	polygon with z coordinates and measures
multipoint	multipoint empty	empty multipoint
multipoint	multipoint z empty	empty multipoint with z coordinates
multipoint	multipoint m empty	empty multipoint with measures
multipoint	multipoint zm empty	empty multipoint with z coordinates with measures
multipoint	multipoint empty	empty multipoint
multipoint	multipoint (10 10, 20 20)	multipoint with two points
multipoint	multipoint z (10 10 2, 20 20 3)	multipoint with z coordinates
multipoint	multipoint m (10 10 4, 20 20 5)	multipoint with measures
multipoint	multipoint zm (10 10 2 4, 20 20 3 5)	multipoint with z coordinates and measures
multilinestring	multilinestring empty	empty multilinestring
multilinestring	multilinestring z empty	empty multilinestring with z coordinates
multilinestring	multilinestring m empty	empty multilinestring with measures
multilinestring	multilinestring zm empty	empty multilinestring with z coordinates and measures
multilinestring	multilinestring (( ))	multilinestring
multilinestring	multilinestring z (( ))	multilinestring with z coordinates
multilinestring	multilinestring m (( ))	multilinestring with measures
multilinestring	multilinestring zm (( ))	multilinestring with z coordinates and measures
multipolygon	multipolygon empty	empty multipolygon
multipolygon	multipolygon z empty	empty multipolygon with z coordinates
multipolygon	multipolygon m empty	empty multipolygon with measures
multipolygon	multipolygon z	empty multipolygon with z coordinates and measures
multipolygon	multipolygon ((( )))	multipolygon
multipolygon	multipolygon z ((( )))	multipolygon with z coordinates
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

Geometry type	Text Description	Comment
multipolygon	multipolygon zm ((( )))	multipolygon with z coordinates and measures

Appendix C. The OGIS Well-Known Binary Representation

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

The Well-Known Binary Representation for Geometry is obtained by serializing a geometry instance as a sequence of numeric types drawn from the set {Unsigned Integer, Double} and then serializing each numeric type as a sequence of bytes using one of two well defined, standard, binary representations for numeric types (NDR, XDR). The specific binary encoding (NDR or XDR) used for a geometry byte stream is described by a one byte tag that precedes the serialized bytes. The only difference between the 2 encodings of geometry is one of byte order, the XDR encoding is Big Endian, 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 NDR and XDR

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 which consists of two doubles. The byte streams for other geometries are built using the byte streams for geometries that have already been defined.

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

// Building Blocks : Point, LinearRing

Point {
  double x;
  double y;
};
LinearRing   {
  uint32  numPoints;
  Point   points[numPoints];
};
enum wkbGeometryType {
  wkbPoint = 1,
  wkbLineString = 2,
  wkbPolygon = 3,
  wkbMultiPoint = 4,
  wkbMultiLineString = 5,
  wkbMultiPolygon = 6,
};
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;
    WKBMultiPoint            mpoint;
    WKBMultiLineString       mlinestring;
    WKBMultiPolygon          mpolygon;
  }
};
```

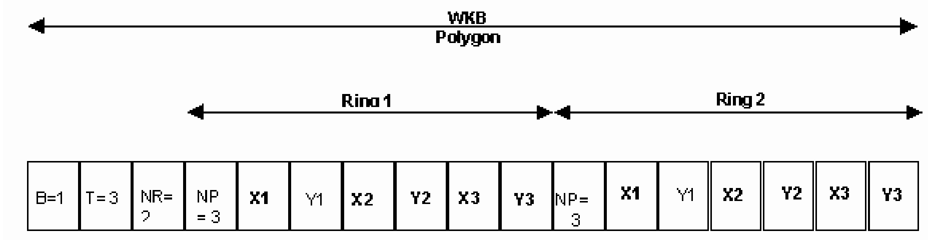


Figure 38. Representation in NDR format. (B=1) of type polygon (T=3) with 2 linears (NR=2), each ring having 3 points (NP=3).

Assertions for the WKB Representation

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 2 Polygons that are elements of a MultiPolygon may not intersect. The Boundaries of any 2 Polygons that are elements of a MultiPolygon may touch at only a *finite* number of points.

Appendix D. The ESRI Shape Representations

A shape type of 0 indicates a null shape, with no geometric data for the shape.

Value	Shape Type
0	Null Shape
1	Point
3*	PolyLine
5	Polygon
8	MultiPoint
11	PointZ
13	PolyLineZ
15	PolygonZ
18	MultiPointZ
21	PointM
23	PolyLineM
25	PolygonM
28	MultiPointM

Note: * Shape types not specified above (2, 4, 6, etc.) are reserved for future use.

Shape Types in XY Space

Point

A point consists of a pair of double precision coordinates in the order X, Y.

Table 20. Point Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	1	Integer	1	Little
Byte 4	X	X	Double	1	Little
Byte 12	Y	Y	Double	1	Little

MultiPoint

A MultiPoint consists of a collection of points. The bounding box is stored in the order Xmin, Ymin, Xmax, Ymax.

Table 21. MultiPoint Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	8	Integer	1	Little

Table 21. MultiPoint Byte Stream Contents (continued)

Position	Field	Value	Type	Number	Order
Byte 4	Box	Box	Double	4	Little
Byte 36	NumPoints	NumPoints	Integer	1	Little
Byte 40	Points	Points	Point	NumPoints	Little

PolyLine

A PolyLine is an ordered set of vertices which 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.

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:

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 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.

Table 22. PolyLine Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	3	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little

Note: 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 counter-clockwise 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:

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 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.

Important notes about Polygon shapes::

1. The rings are closed (the first and last vertex of a ring MUST be the same).
2. The order of rings in the points array is not significant.
3. Polygons stored in a shapefile must be clean. A clean polygon is one that:
   a. 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.
   b. 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.

An Example Polygon Instance

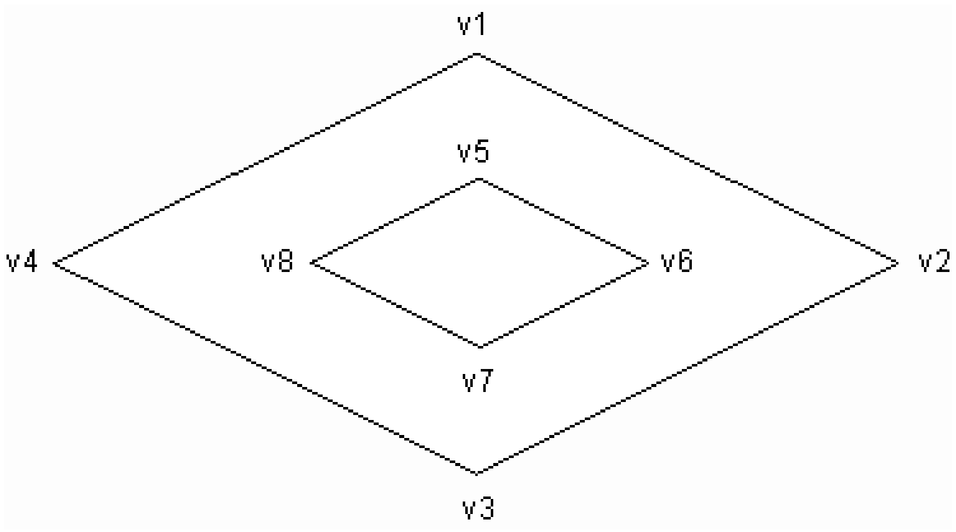


Figure 39. A polygon with a hole and eight vertices

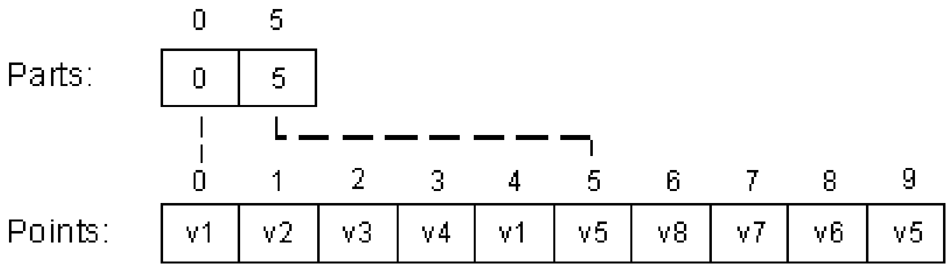


Figure 40. Contents of the polygon byte stream. NumParts equals 2 and NumPoints equals 10. Note that the order of the points for the donut (hole) polygon are reversed.

Table 23. Polygon Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	5	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little

Note: X = 44 + 4 * NumParts.

Measured Shape Types in XY Space

PointM

A PointM consists of a pair of double precision coordinates in the order X, Y, plus a measure M.

Table 24. PointM Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	21	Integer	1	Little
Byte 4	X	X	Double	1	Little
Byte 12	Y	Y	Double	1	Little
Byte 20	M	M	Double	1	Little

MultiPointM

The fields for a MultiPointM:

Box The bounding box for the MultiPointM stored in the order Xmin, Ymin, Xmax, Ymax.

NumPoints
The number of Points.

Points An array of Points of length NumPoints.

NumMs
The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

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.

Table 25. MultiPointM Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	28	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumPoints	NumPoints	Integer	1	Little
Byte 40	Points	Points	Point	NumPoints	Little
Byte X	NumMs	NumMs	Integer	1	Little
Byte X+4*	Mmin	Mmin	Double	1	Little

Table 25. MultiPointM Byte Stream Contents (continued)

Position	Field	Value	Type	Number	Order
Byte X+12*	Mmax	Mmax	Double	1	Little
Byte X+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. X = 40 + (16 * NumPoints)
2. * optional

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:

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 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.

NumMs
The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

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.

Table 26. PolyLineM Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	13	Integer	1	Little

Table 26. PolyLineM Byte Stream Contents (continued)

Position	Field	Value	Type	Number	Order
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	NumMs	NumMs	Integer	1	Little
Byte Y+4*	Mmin	Mmin	Double	1	Little
Byte Y+12*	Mmax	Mmax	Double	1	Little
Byte Y+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. X = 44 + (4 * NumParts), Y = X + (16 * NumPoints).
2. * optional

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:

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 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.

NumMs
The number of Measures that follow. NumMs can have only two zero values if no Measures follow this field, or equal to NumPoints if Measures are present.

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.

Important notes about PolygonM shapes::

1. The rings are closed (the first and last vertex of a ring must be the same).
2. The order of rings in the points array is not significant.

Table 27. PolygonM Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	15	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	NumMs	NumMs	Integer	1	Little
Byte Y+4*	Mmin	Mmin	Double	1	Little
Byte Y+12*	Mmax	Mmax	Double	1	Little
Byte Y+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. X = 44 + (4 * NumParts), Y = X + (16 * NumPoints).
2. * optional

Shape Types in XYZ Space

PointZ

A PointZ consists of a triplet of double precision coordinates in the order X, Y, Z plus a measure.

Table 28. PointZ Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	11	Integer	1	Little
Byte 4	X	X	Double	1	Little
Byte 12	Y	Y	Double	1	Little
Byte 20	Z	Z	Double	1	Little
Byte 28	Measure	M	Double	1	Little

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. Bounding M Range is stored in the order Mmin, Mmax.

Table 29. MultiPointZ Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	18	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumPoints	NumPoints	Integer	1	Little
Byte 40	Points	Points	Point	NumPoints	Little
Byte X	Zmin	Zmin	Double	1	Little
Byte X+8	Zmax	Zmax	Double	1	Little
Byte X+16	Zarray	Zarray	Double	NumPoints	Little
Byte Y	NumMs	NumMs	Integer	1	Little
Byte Y+4*	Mmin	Mmin	Double	1	Little
Byte Y+12*	Mmax	Mmax	Double	1	Little
Byte Y+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. X = 40 + (16 * NumPoints); Y = X + 16 + (8 * NumPoints)
2. * optional

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:

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 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.

NumMs
The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

M Range
The minimum and maximum measures for the PolyLineZ stored in the order Mmin, Mmax.

M Array
An array of length NumPoints. The measures for each part in the PolyLineZ 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 measure for each part. There is no delimiter in the measure array between parts.

Table 30. PolyLineZ Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	13	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	Zmin	Zmin	Double	1	Little
Byte Y+8	Zmax	Zmax	Double	1	Little
Byte Y+16	Zarray	Zarray	Double	NumPoints	Little
Byte Z	NumMs	NumMs	Integer	1	Little
Byte Z+4*	Mmin	Mmin	Double	1	Little
Byte Z+12*	Mmax	Mmax	Double	1	Little
Byte Z+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. X = 44 + (4 * NumParts), Y = X + (16 * NumPoints), Z = Y + 16 + (8 * NumPoints)
2. * optional

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:

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 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 arc 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.

NumMs
The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

M Range
The minimum and maximum measures for the PolygonZ stored in the order Mmin, Mmax.

M Array
An array of length NumPoints. The measures for each ring in the PolygonZ 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.

Important notes about PolygonZ shapes::

1. The rings are closed (the first and last vertex of a ring MUST be the same).
2. The order of rings in the points array is not significant.

Table 31. PolygonZ Byte Stream Contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	15	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	Zmin	Zmin	Double	1	Little
Byte Y+8	Zmax	Zmax	Double	1	Little
Byte Y+16	Zarray	Zarray	Double	NumPoints	Little
Byte Z	NumMs	NumMs	Integer	1	Little
Byte Z+4*	Mmin	Mmin	Double	1	Little
Byte Z+12*	Mmax	Mmax	Double	1	Little
Byte Z+20*	Marray	Marray	Double	NumPoints	Little

Notices

Trademarks

Index

Readers' Comments — We'd Like to Hear from You

DB2 Spatial Extender
Administration Guide and Reference
Version 2 Release 1 Modification 1

Publication No. SC26-9316-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC26-9316-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
Department BWE/H3
PO Box 49023
San Jose, CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

SC26-9316-00

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-9316-00

Spine information:

DB2 Spatial Extender

Administration Guide and Reference

Version 2
Release 1
Modification 1

SC26-
9316-00