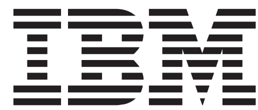
Informix Product Family
Informix
Version 12.10

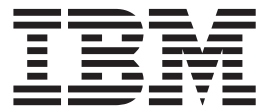
IBM Informix Spatiotemporal Search for Moving Objects User's Guide



SC27-8019-00

Informix Product Family
Informix
Version 12.10

IBM Informix Spatiotemporal Search for Moving Objects User's Guide



SC27-8019-00

Note

Before using this information and the product it supports, read the information in "Notices" on page B-1.

Edition

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

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.

© Copyright IBM Corporation 2015.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction . . . v
About this publication . . . v
Types of users . . . v
Example code conventions. . . . v
Additional documentation . . . vi
Compliance with industry standards . . . vi
How to read the syntax diagrams . . . vi
How to provide documentation feedback . . . viii

Chapter 1. Getting started with spatiotemporal search for moving objects . . . 1-1
Spatiotemporal search solution architecture . . . 1-2
Software requirements for the Spatiotemporal Search extension . . . 1-3
Preparing for spatiotemporal search . . . 1-4
Example for spatiotemporal data: Create, load, and search a time series . . . 1-5
Time series requirements for spatiotemporal search . . . 1-7
Spatiotemporal search indexing . . . 1-8
Spatiotemporal searches . . . 1-8
Spatial data types for spatiotemporal searches . . . 1-11
Stopping spatiotemporal search indexing . . . 1-12

Chapter 2. Spatiotemporal search routines . . . 2-1
STS_Cleanup function . . . 2-2
STS_GetCompactTrajectory function . . . 2-3
STS_GetFirstTimeByPoint function . . . 2-4
STS_GetIntersectSet function . . . 2-6
STS_GetLastPosition function . . . 2-7
STS_GetLocWithinSet function . . . 2-8
STS_GetNearestObject function . . . 2-9
STS_GetPosition function . . . 2-11
STS_GetTrajectory function . . . 2-12
STS_Init function . . . 2-14
STS_Release function . . . 2-15
STS_Set_Trace procedure . . . 2-16
STS_TrajectoryCross function . . . 2-16
STS_TrajectoryDistance function . . . 2-18
STS_TrajectoryIntersect function . . . 2-20
STS_TrajectoryWithin function . . . 2-22

Appendix. Accessibility . . . A-1
Accessibility features for IBM Informix products . . . A-1
Accessibility features . . . A-1
Keyboard navigation . . . A-1
Related accessibility information . . . A-1
IBM and accessibility. . . . A-1
Dotted decimal syntax diagrams . . . A-1

Notices . . . B-1
Privacy policy considerations . . . B-3
Trademarks . . . B-3

Index . . . X-1

© Copyright IBM Corp. 2015

Introduction

About this publication

This publication describes how to use the spatiotemporal search extension to write an application that tracks moving objects.

Types of users

This publication is for database application programmers.

To write an application for spatiotemporal search, you must have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Experience working with relational databases or exposure to database concepts
- Experience with computer programming in the C or Java™ programming language
- Experience with writing applications for spatial data
- An understanding of basic time series concepts

Example code conventions

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

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

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

© Copyright IBM Corp. 2015

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access Informix technical information such as information centers, technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

How to read the syntax diagrams

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

Read the syntax diagrams from left to right and top to bottom, following the path of the line.

The double right arrowhead and line symbol ►►── indicates the beginning of a syntax diagram.

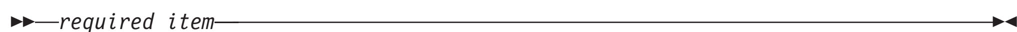
The line and single right arrowhead symbol ──► indicates that the syntax is continued on the next line.

The right arrowhead and line symbol ►── indicates that the syntax is continued from the previous line.

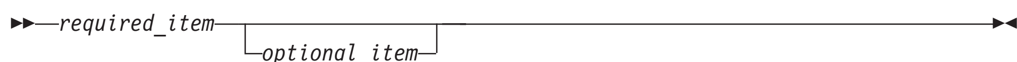
The line, right arrowhead, and left arrowhead symbol ──►◄ symbol indicates the end of a syntax diagram.

Syntax fragments start with the pipe and line symbol |── and end with the ──| line and pipe symbol.

Required items appear on the horizontal line (the main path).



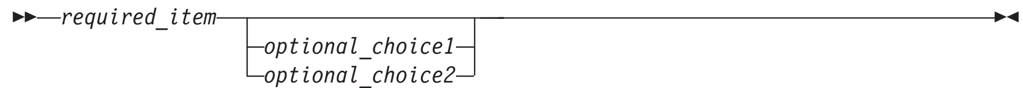
Optional items appear below the main path.



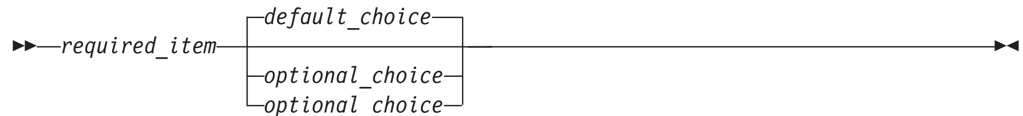
If you can choose from two or more items, they appear in a stack.

If you *must* choose one of the items, one item of the stack appears on the main path.

If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.



An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.

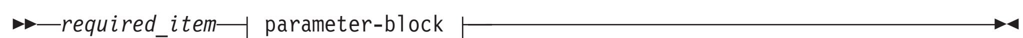
If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

SQL keywords appear in uppercase (for example, `FROM`). They must be spelled exactly as shown. Variables appear in lowercase (for example, `column-name`). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a syntax segment. For example, in the following diagram, the variable `parameter-block` represents the syntax segment that is labeled **parameter-block**:



parameter-block:

How to provide documentation feedback

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

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- Add comments to topics directly in IBM Knowledge Center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. Getting started with spatiotemporal search for moving objects

You use spatiotemporal searches to track moving objects. You create a spatiotemporal search index on the time-stamped GPS data that is stored in a historian database as time series.

You can query on either time or on location to determine the relationship of one to the other. You can query when an object was at a specified location, or where an object was at a specified time. You can also find the path, or trajectory, of a moving object over a range of time or the relationship between a region and the trajectories of moving objects.

You can use spatiotemporal searches to find the following types of information:
- Find the location of a moving object at a specific time. For example, find the location of bus number 3435 at 2014-03-01 15:30.
- Find the last known time and location of a specific moving object. For example, find the last known location of taxi number 324.
- Find when, in a time range, a moving object was in a region around a point of interest. For example, find when a deliver truck was within 100 meters of the Mom and Pop Diner between February 2-4, 2015.
- Find when a moving object was at a specific location. For example, find when tax number 324 was at the Four Seasons Hotel.
- Find the trajectories, of a specific moving object for a time range. For example, find the trajectories of bus number 1543 between 9:00-17:00 yesterday.
- Find the trajectories of moving objects near a point of interest during a time range. For example, find which taxi driver witnessed an accident by finding which taxi was nearest to the location of the accident at 9:00, as shown in the following illustration.

© Copyright IBM Corp. 2015

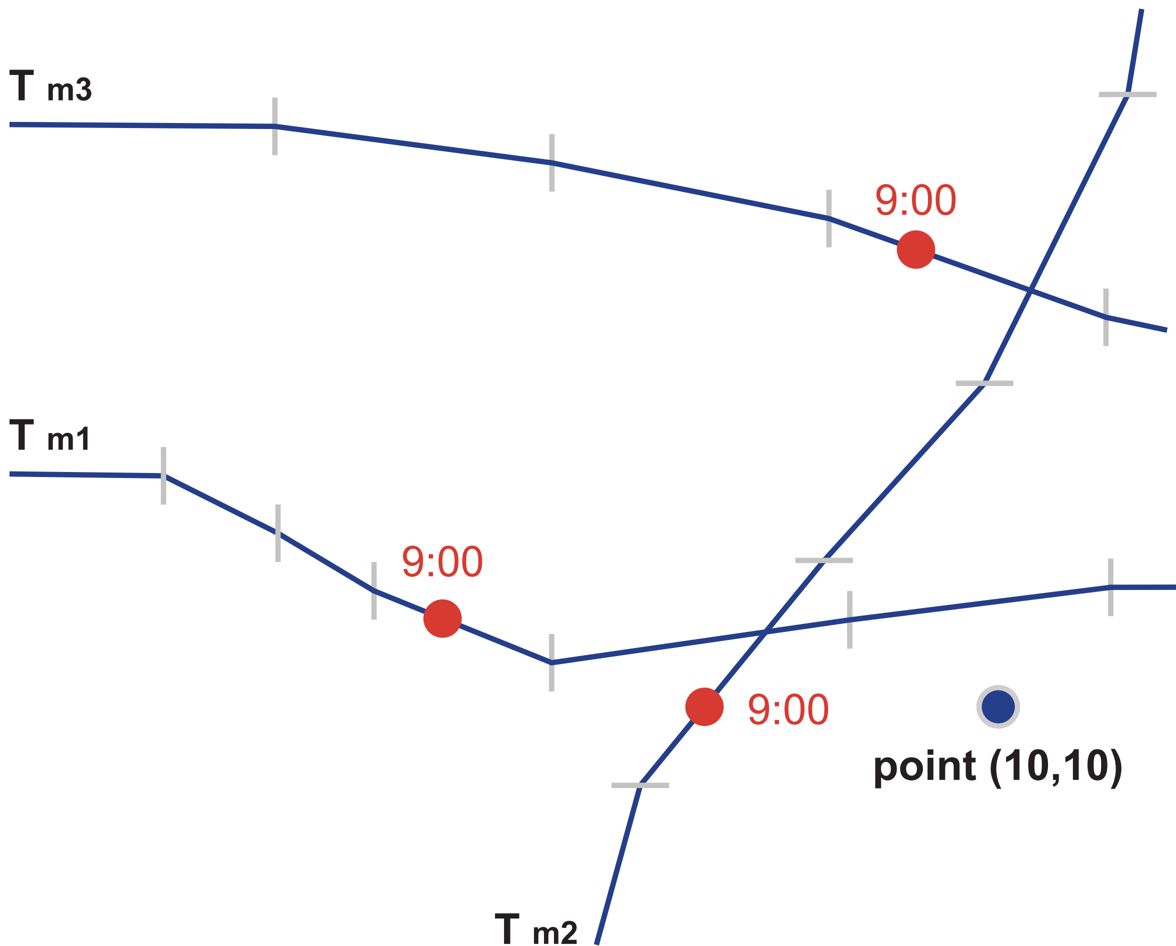


Figure 1-1. Trajectories near a point at a specific time

Spatiotemporal search solution architecture

The Informix spatiotemporal search solution consists of built-in functions.

The Informix database server includes the following functionality for searching spatiotemporal data:

- An SQL function to index spatiotemporal data.
- SQL functions to query spatiotemporal data.
- SQL functions to remove spatiotemporal indexes.

The spatiotemporal search solution builds upon the data types and routines from the TimeSeries solution and the data types, routines, and R-tree indexes from the spatial solution. The following illustration shows how the spatiotemporal search solution and related products interact.

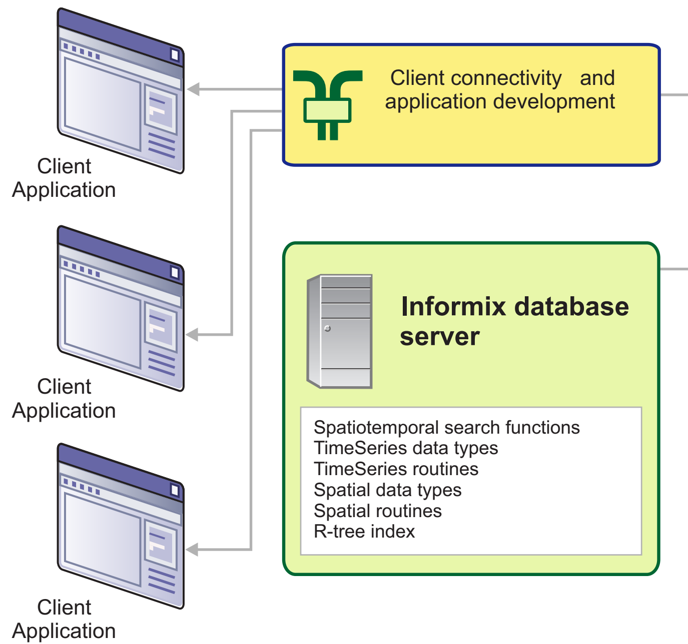


Figure 1-2. Spatiotemporal search architecture

Spatiotemporal searches return location data as spatial data types from the Informix spatial extension. You must provide your own visualization software, such as client programs from ESRI.

Related concepts:

Informix spatial solution architecture (Spatial Data Guide)

Informix TimeSeries solution architecture (TimeSeries Data Guide)

Software requirements for the Spatiotemporal Search extension

The spatiotemporal search extension requires the TimeSeries and Spatial extensions and that the Scheduler is running.

Requirements

The Spatiotemporal Search extension is is subject to the hardware and software requirements of the TimeSeries and the Spatial extensions. For hardware restrictions, see https://www.ibm.com/support/docview.wss?rs=630&uid=swg27020937.

Creating a spatiotemporal search index has the following prerequisites:

- The Scheduler must be running. The Scheduler automatically registers the Spatiotemporal Search and Spatial extensions and runs the task to index spatiotemporal data. The name of the Spatiotemporal Search extension is `sts.bld` and it is in the `$INFORMIXDIR/extend/sts.`*`version`*`/` directory, where version is the *version* number of the extension.
- A time series table must exist and the TimeSeries extension must be registered in the database.
- If you are using multiple CPU virtual processors, the PRELOAD_DLL_FILE configuration parameter must specify the path for the spatiotemporal shared library file in the `onconfig` file:

```
PRELOAD_DLL_FILE $INFORMIXDIR/extend/sts.version/sts.bld
```

version is the version number of the extension. You must restart the database server after setting the PRELOAD_DLL_FILE configuration parameter.

Limitations

The following limitations apply to spatiotemporal searching:

- The time series table that contains spatiotemporal data cannot have more than 10 000 rows, where each row contains data for a specific moving object.
- Spatiotemporal searches might be inaccurate if moving objects stay in place or fail to transmit location information.
- If you modify or delete data from the time series table, you must recreate the spatiotemporal index.
- The distance parameter of spatiotemporal functions currently defines the region of interest with a Euclidean calculation that is based on the Cartesian system instead of a spherical calculation that is based on longitude and latitude coordinates.
- Any data that you insert with timepoints that are earlier than the last time point that was indexed are not indexed.

Replication

You can replicate spatiotemporal indexes between a high-availability data replication primary server and a read-only secondary server.

Related concepts:

"Spatial data types for spatiotemporal searches" on page 1-11

The Scheduler (Administrator's Guide)

Related reference:

PRELOAD_DLL_FILE configuration parameter (Administrator's Reference)

"Time series requirements for spatiotemporal search" on page 1-7

Preparing for spatiotemporal search

Before you can run spatiotemporal searches, you must create a time series that includes spatial data and start indexing that data.

To prepare for spatiotemporal searching:

1. Create and load a time series that conforms to the requirements for spatiotemporal search.
2. Start the spatiotemporal search indexing process by running the **STS_Init** function on the time series table. The **STS_Init** function starts a Scheduler task that indexes the data. When the Scheduler task starts indexing, the following message appears in the database server message log:

   ```
   INFO (STSMessage) Building trajectories for table_name is started.
   ```

   When the Scheduler task finishes indexing, the following message appears in the database server message log:

   ```
   INFO (STSMessage) Building trajectories for table_name is stopped.
   ```

When the index is complete, you can run spatiotemporal searches.

Related reference:

Create and manage a time series through SQL (TimeSeries Data Guide)

Example for spatiotemporal data: Create, load, and search a time series

In this example, you create a time series that contains location points. Readings are allowed every second. The following table lists the time series properties that are used in this example.

Table 1-1. Time series properties used in this example

Time series property	Definition
Timepoint size	1 second
When timepoints are valid	Any second, with no invalid times
Data in the time series	• Timestamp • FLOAT column for longitude readings in the spatial reference system 4326 • FLOAT column for latitude readings in the spatial reference system 4326
Time series table	• An object ID column of type VARCHAR • A **TimeSeries** data type column
Origin	2014-01-01 00:00:00.00000
Regularity	Irregular
Metadata	No metadata
Where to store the data	In a container that you create
How to load the data	Through a loader program

To create a time series for moving objects and run spatiotemporal queries:

1. Create a calendar pattern for one second by running the following SQL statement:

```
INSERT INTO CalendarPatterns values ('calpat_1s', '{1 on} second');
```

2. Create a calendar that is named **cal_1s** by running the following SQL statement:

```
INSERT INTO Calendartable (c_name, c_calendar) values (
    'cal_1s',
    'startdate(2007-01-01 00:00:00.00000),
     pattstart(2007-01-01 00:00:00.00000), pattname(calpat_1s)'
);
```

3. Create a **TimeSeries** subtype that is named **rt_track** in a database by running the following SQL statement:

```
CREATE ROW TYPE rt_track(
      tstamp     DATETIME YEAR TO FRACTION(5),
      longitude  FLOAT,
      latitude   FLOAT
);
```

   The **longitude** and **latitude** fields, must have FLOAT data types and be the second and third fields, respectively.

4. Create a time series table that is named **T_Vehicle** by running the following SQL statement:

```
CREATE TABLE T_Vehicle(
 modid    VARCHAR(60),
 ts_track TimeSeries(rt_track),
 type     VARCHAR(60),
 color    VARCHAR(60),
 owner    VARCHAR(60),
 PRIMARY KEY (modid) CONSTRAINT pk_modid
);
```

5. Create a container that is named **c_track** in a dbspace by running the following SQL statement:

   ```
   EXECUTE PROCEDURE
      TSContainerCreate('c_track', 'tsdbs1', 'rt_track', 0, 0);
   ```

   Substitute the name of your dbspace for `tsdbs1`.

6. Create two time series instances by running the following SQL statements:

   ```
   INSERT INTO T_Vehicle VALUES('1','calendar(cal_1s),
                       origin(2014-01-01 00:00:00),threshold(0),
                       container(c_track),irregular', '', '', '');

   INSERT INTO T_Vehicle VALUES('2','calendar(cal_1s),
                       origin(2014-01-01 00:00:00),threshold(0),
                       container(c_track),irregular', '', '', '');
   ```

7. Create a pipe-delimited file in any directory with the name `sts.unl` that contains the following data to load:

   ```
   1|2014-02-02 13:34:06|116.40061|39.90605|
   1|2014-02-02 13:39:07|116.40121|39.9139|
   1|2014-02-02 13:40:55|116.40117|39.91159|
   1|2014-02-02 13:44:09|116.39245|39.90635|
   1|2014-02-02 13:49:10|116.36999|39.90594|
   1|2014-02-02 13:54:12|116.34526|39.90589|
   1|2014-02-02 15:10:00|116.34526|39.90589|
   1|2014-02-02 16:30:00|116.42000|40.10000|
   1|2014-02-02 17:10:10|116.40100|39.90700|
   1|2014-02-02 17:15:30|116.40100|39.90700|
   1|2014-02-02 17:20:00|116.40200|39.90800|
   1|2014-02-02 18:40:00|116.40201|39.90801|
   2|2014-02-02 13:34:06|116.40061|39.90605|
   2|2014-02-02 13:39:07|116.40121|39.9139|
   2|2014-02-02 13:40:55|116.40117|39.91159|
   2|2014-02-02 13:44:09|116.39245|39.90635|
   2|2014-02-02 13:49:10|116.36999|39.90594|
   2|2014-02-02 13:54:12|116.34526|39.90589|
   2|2014-02-02 14:10:12|116.34526|39.90589|
   2|2014-02-02 14:30:00|116.3452|39.9058|
   2|2014-02-02 16:00:00|116.42000|40.10000|
   2|2014-02-02 16:10:10|116.40100|39.90700|
   2|2014-02-02 16:20:00|116.40200|39.90800|
   ```

8. Initialize a global context and open a database session by running the **TSL_Init** and the **TSL_Attach** functions:

   ```
   EXECUTE FUNCTION TSL_Init('T_Vehicle','rt_track');
   EXECUTE FUNCTION TSL_Attach('T_Vehicle','rt_track');
   ```

9. Load the data by running the **TSL_Put** function with an SQL statement that selects the data from the file:

   ```
   EXECUTE FUNCTION TSL_Put('T_Vehicle|rt_track',
            "FILE:path/sts.unl");
   ```

   Substitute *path* with the directory for the `sts.unl` file.

10. Save the data to disk by running the **TSL_FlushAll** function:

```
BEGIN;
EXECUTE FUNCTION TSL_FlushAll('T_Vehicle|rt_track');
COMMIT WORK;
```

11. Close the session and remove the global context by running the **TSL_SessionClose** and **TSL_Shutdown** functions:

```
EXECUTE FUNCTION TSL_SessionClose('T_Vehicle|rt_track');
EXECUTE PROCEDURE TSL_Shutdown('T_Vehicle|rt_track');
```

12. Start spatiotemporal search indexing by running the **STS_Init** function:

```
EXECUTE FUNCTION STS_Init(T_Vehicle);
```

13. Run any of the queries in the examples of the spatiotemoral search functions. For example, you can run the following query:

```
SELECT STS_GetFirstTimeByPoint('T_Vehicle', modid, ts_track, null, null,
                               '4326 point(116.401 39.911)', 100)
       FROM T_Vehicle
       WHERE modid = '1';

(expression)

2014-02-02 13:37:15.00000

1 row(s) retrieved.
```

14. Optional: Stop spatiotemporal indexing and remove the internal tables by running the **STS_CleanUp** function:

```
EXECUTE FUNCTION STS_CleanUp('T_Vehicle');
```

Related reference:

Create and manage a time series through SQL (TimeSeries Data Guide)

TSL_Put function (TimeSeries Data Guide)

Time series requirements for spatiotemporal search

The time series that you create for spatiotemporal search must conform to certain requirements.

Database and table requirements

The database cannot be a tenant database.

The time series table must conform to the following requirements and restrictions:

- The first column must be a primary key column that represents an object ID and has a data type of INTEGER, CHAR, or VARCHAR. A composite primary key is not allowed.
- The second column must be a **TimeSeries** subtype column.
- The table can have more columns, however, any additional **TimeSeries** columns are not indexed.
- The table name must be unique. The table name is used to identify the spatiotemporal search. If the table name is longer than 100 bytes, the first 100 bytes of the name must be unique.
- The table must have fewer than 10 000 rows.

TimeSeries subtype requirements

The **TimeSeries** subtype must have the following structure:

1. The first field is the time stamp field. This requirement is true of all **TimeSeries** subtypes.

2. The second field has a FLOAT data type to hold longitude data that is in the spatial reference system 4326 (WGS 84).
3. The third field has a FLOAT data type to hold latitude data that is in the spatial reference system 4326 (WGS 84).
4. Optional additional fields can have any data type that is supported in a **TimeSeries** subtype.

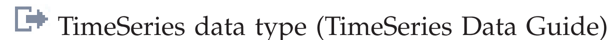
Time series definition restrictions

Although a regular time series is supported, an irregular time series is more appropriate for moving object data.

Hertz and compressed data are not supported. The time series definition cannot include the *hertz* or *compression* parameters.

Related concepts:

"Software requirements for the Spatiotemporal Search extension" on page 1-3


TimeSeries data type (TimeSeries Data Guide)

Related reference:

Create the database table (TimeSeries Data Guide)

Spatiotemporal search indexing

After you start spatiotemporal search indexing, the process of indexing time series data for spatiotemporal searches continues automatically.

When you run the **STS_Init()** function to start spatiotemporal search indexing for a time series table, the following tasks are performed:

1. An internal subtrack table is created.
2. The data from the time series table is converted into compact trajectories and stored in the subtrack table.
3. An R-tree index is created on the column in the subtrack table that contains spatial data.
4. An internal lasttime table is created. The lasttime table stores the last time that spatiotemporal data was processed for each object ID.
5. A Scheduler task is created with the name of the time series table. The task runs at a preset frequency. The task adds any new data from the time series table into the subtrack table and updates the corresponding R-tree index.

The trajectories that are added to the subtrack table are simplified versions of the actual trajectories. The data is compacted to make queries faster.

Related reference:

"STS_Init function" on page 2-14

Spatiotemporal searches

You run spatiotemporal search functions within SQL statements to find information about moving objects.

Find the relationship between an object, time, and a point

Run the following functions to find an object, a time, or a position in terms of the other two criteria:

- Find the position of an object at a specific time: **STS_GetPosition**
- Find the most recent position of any object in the time series: **STS_LastPosition**
- Find the first time within a time range when an object is near a position: **STS_GetFirstTimeByPoint**
- Find the nearest object to a point at a specific time: **STS_GetNearestObject**

Find a trajectory of an object

You can find the exact trajectory or the compressed trajectory of an object:

- Find the exact trajectory for a time range. The exact trajectory includes every location for every timepoint, from the data in the time series table: **STS_GetTrajectory**
- Find the compact trajectory for a time range: **STS_GetCompactTrajectory**. The compacted trajectory is an approximation of the trajectory from the internal subtrack table.

The following illustration shows the difference between the exact trajectory and the compressed trajectory.

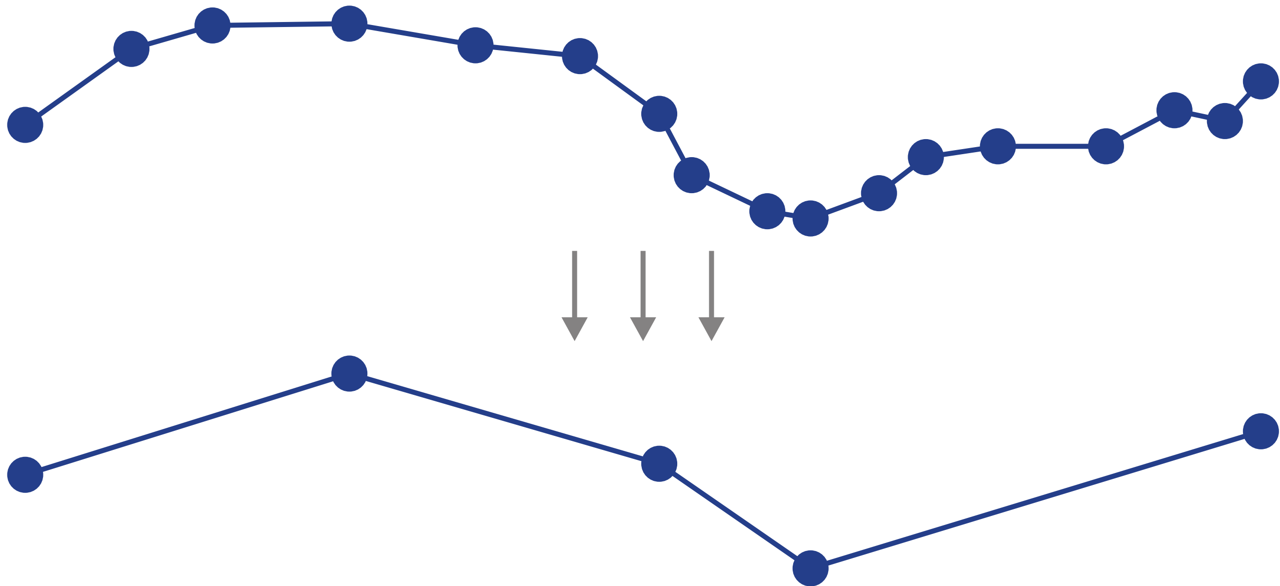


Figure 1-3. Exact and compressed trajectories

In this illustration, the **STS_GetTrajectory** function returns a trajectory with 16 points. The **STS_GetCompactTrajectory** function returns a trajectory with 5 points.

The **STS_GetTrajectory** function provides more accurate results, but the **STS_GetCompactTrajectory** function runs faster.

Find the objects in a region

You can find which objects were in a region a specific times:

- Find the set of objects whose trajectories intersected a region during the time range: **STS_GetIntersectSet**
- Find the set of objects that were within a region at a specific time: **STS_GetLocWithinSet**

Find the relationship between trajectories and a region

You can find the shortest distance between a point and the trajectory of an object during a time range by running the **STS_TrajectoryDistance** function.

You can find out whether an object was in a region during a range of time:

- Find whether the trajectory remained within the boundary of the region for the time range: **STS_TrajectoryWithin**
- Find whether the trajectory crossed the boundary of the region in the time range: **STS_TrajectoryCross**
- Find whether the trajectory either crossed the boundary of the region or remained within the boundary of the region for the time range: **STS_TrajectoryIntersect**

The following illustration shows a trajectory that goes through a region.

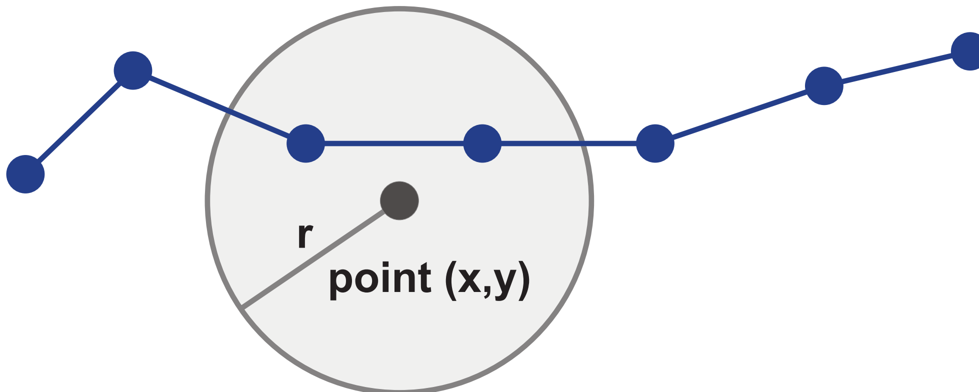


Figure 1-4. A trajectory that crosses the region boundary twice

The following functions return the following results for this trajectory and region:

- The **STS_TrajectoryWithin** function returns FALSE because the trajectory does not remain within the boundaries of the region.
- The **STS_TrajectoryCross** function returns TRUE because the trajectory crosses the boundary of the region.
- The **STS_TrajectoryIntersect** function returns TRUE because the trajectory intersects the region.

The following image shows a trajectory that is entirely within a region.

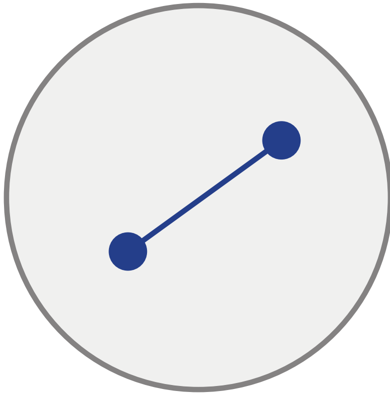


Figure 1-5. A trajectory within a region

The following functions return the following results for this trajectory and region:

- The **STS_TrajectoryWithin** function returns TRUE because the trajectory remains within the boundaries of the region.
- The **STS_TrajectoryCross** function returns FALSE because the trajectory does not cross the boundary of the region.
- The **STS_TrajectoryIntersect** function returns TRUE because the trajectory stays within the region.

Related reference:

Time series SQL routines (TimeSeries Data Guide)

Spatial functions (Spatial Data Guide)

Spatial data types for spatiotemporal searches

Spatiotemporal search functions either take a spatial data type as an argument or return a spatial data type.

Spatiotemporal seach functions use the following spatial data types:

- ST_Point: A location that is specified by longitude (X) and latitude (Y) coordinate values. For functions that take an ST_Point argument, supply an X,Y coordinate value. ST_Points are also returned by some functions.
- ST_MultiLineString: A set of one or more linestrings that represent a trajectory. ST_MultiLineStrings are returned by functions that find trajectories.
- ST_Geometry: An abstract noninstantiable superclass. For functions that take an ST_Geometry, supply an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon value.

Spatial data types require a spatial reference ID (SRID) that identifies the type of map projection system. For spatiotemporal search data, the SRID must be 4326, which the SRID that is commonly used by global positioning system (GPS) devices.

Spatiotemporal search functions that take a distance parameter to define a region of interest also take an optional unit of measure parameter. By default, the unit of measurement for distance is meters. You can specify a unit of measure that is listed in the **unit_name** column of the **st_units_of_measure** table.

Restriction: The distance parameter currently defines the region of interest with a Euclidean calculation that is based on the Cartesian system instead of a spherical calculation that is based on longitude and latitude coordinates.

Related concepts:

"Software requirements for the Spatiotemporal Search extension" on page 1-3

Related reference:

Spatial data types (Spatial Data Guide)

The st_units_of_measure table (Spatial Data Guide)

Chapter 2, "Spatiotemporal search routines," on page 2-1

Stopping spatiotemporal search indexing

When you stop spatiotemporal search indexing, you remove the spatiotemporal search internal tables, Scheduler tasks, and indexes.

To stop spatiotemporal search indexing for a specific time series, run the **STS_Cleanup** function and specify the time series table.

To stop spatiotemporal search indexing for a database, run the **STS_Cleanup** function without any parameters while connected to the database.

To remove all spatiotemporal search software in the database in one step, run the following statement:

```
EXECUTE FUNCTION SYSBldPrepare('sts*', 'drop');
```

Related reference:

"STS_Cleanup function" on page 2-2

Chapter 2. Spatiotemporal search routines

Spatiotemporal search routines index spatiotemporal data, query spatiotemporal data, and perform maintenance tasks.

The following table sorts spatiotemporal search routines by task.

Table 2-1. Spatiotemporal search functions sorted by task

Header	Header
Start or stop spatiotemporal search indexing	Start spatiotemporal indexing for a table: "STS_Init function" on page 2-14 Stop spatiotemporal search indexing and drop internal tables: "STS_Cleanup function" on page 2-2
Find the relationship between an object, time, and a point	Find the position of an object at a specific time: "STS_GetPosition function" on page 2-11 Find the most recent position of any object in the time series: "STS_GetLastPosition function" on page 2-7 Find the first time within a time range when an object is near a position: "STS_GetFirstTimeByPoint function" on page 2-4 Find the nearest object to a point at a specific time: "STS_GetNearestObject function" on page 2-9
Find a trajectory of an object	Find the exact trajectory for a time range: "STS_GetTrajectory function" on page 2-12 Find the compressed trajectory for a time range: "STS_GetCompactTrajectory function" on page 2-3
Find the objects in a region	Find the set of objects whose trajectories intersected a region during the time range: "STS_GetIntersectSet function" on page 2-6 Find the set of objects that were within a region at a specific time: "STS_GetLocWithinSet function" on page 2-8

© Copyright IBM Corp. 2015

Table 2-1. Spatiotemporal search functions sorted by task (continued)

Header	Header
Find the relationship between trajectories and a region	Find the shortest distance between a point and the trajectory of an object during a time range: "STS_TrajectoryDistance function" on page 2-18 Find whether the trajectory remained within the boundary of the region for the time range: "STS_TrajectoryWithin function" on page 2-22 Find whether the trajectory crossed the boundary of the region in the time range: "STS_TrajectoryCross function" on page 2-16 Find whether the trajectory either crossed the boundary of the region or remained within the boundary of the region for the time range: "STS_TrajectoryIntersect function" on page 2-20
Return release information	"STS_Release function" on page 2-15
Enable tracing	"STS_Set_Trace procedure" on page 2-16

Related concepts:

"Spatial data types for spatiotemporal searches" on page 1-11

STS_Cleanup function

The **STS_Cleanup** function stops the indexing of spatiotemporal search data and drops the internal tables that contain spatiotemporal search data.

Syntax

```
STS_Cleanup(ts_tabname   VARCHAR(128))
returns INTEGER

STS_Cleanup()
returns INTEGER
```

ts_tabname **(Optional)**
The name of the time series table.

Usage

Run the **STS_Cleanup** function with the *ts_tabname* parameter when you want to stop spatiotemporal indexing and drop the existing spatiotemporal search tables for the specified time series table. For example, when the spatiotemporal search tables become large, you can drop them and then restart spatiotemporal search indexing with a more recent start time.

Run the **STS_Cleanup** function without a parameter to stop spatiotemporal indexing and drop the existing spatiotemporal search tables for the current database.

Returns

An integer that indicates the status of the function:
- 0 = Spatiotemporal search indexing was removed.
- 1 = An error occurred.

Example: Stop indexing and drop index tables for a table

The following statement stops spatiotemporal search indexing and deletes the internal tables for the time series table that is named **T_Vehicle**:

```
EXECUTE FUNCTION STS_Cleanup('T_Vehicle');
```

Example: Stop indexing and drop index tables for a database

The following statement stops spatiotemporal search indexing and deletes the internal tables for the current database:

```
EXECUTE FUNCTION STS_Cleanup();
```

Related tasks:

"Stopping spatiotemporal search indexing" on page 1-12

STS_GetCompactTrajectory function

The **STS_GetCompactTrajectory** function returns the compressed trajectory of a specified object for the specified time range.

Syntax

```
STS_GetCompactTrajectory(ts_tabname    LVARCHAR,
                         obj_id        LVARCHAR,
                         ts            TimeSeries,
                         starttime     DATETIME YEAR TO FRACTION(5),
                         endtime       DATETIME YEAR TO FRACTION(5))
returns LVARCHAR
```

ts_tabname
The name of the time series table.

obj_id The ID of the object. Must be a value from the primary key column of the time series table. Can be the name of the column that stores the object IDs if the WHERE clause specifies a specific object ID.

ts The name of the **TimeSeries** column.

starttime
The start of the time range.

endtime
The end of the time range.

Usage

Run the **STS_GetCompactTrajectory** function to find where an object went during a time range, based on the compressed spatiotemporal search data. The trajectory information is retrieved from the **subtack** table and returned as one or more linestrings.

Returns

An LVARCHAR string that represents the trajectory of the object. The string includes the spatial reference ID and an ST_MultiLinestring.

NULL, if nothing found.

Example

The following query returns the trajectory of the vehicle 1 between 2014-02-02 13:00:00 and 2014-02-02 16:30:00:

```
SELECT STS_GetCompactTrajectory('T_Vehicle', modid, 'ts_track',
                 '2014-02-02 13:00:00', '2014-02-02 16:30:00')
      FROM T_Vehicle
      WHERE modid='1';


(expression)  4326 multilinestring((116.40061 39.90605, 116.40121 39.9139, 116.
              40117 39.91159, 116.39245 39.90635, 116.36999 39.90594, 116.34526
              1 39.905891, 116.345261 39.905891))

1 row(s) retrieved.
```

STS_GetFirstTimeByPoint function

The **STS_GetFirstTimeByPoint** function returns the first time within a time range when an object is near the specified position.

Syntax

```
STS_GetFirstTimeByPoint(ts_tabname   LVARCHAR,
                        obj_id       LVARCHAR,
                        ts           TimeSeries,
                        starttime    DATETIME YEAR TO FRACTION(5),
                        endtime      DATETIME YEAR TO FRACTION(5),
                        geometry     LVARCHAR,
                        max_distance REAL)
returns DATETIME

STS_GetFirstTimeByPoint(ts_tabname   LVARCHAR,
                        obj_id       LVARCHAR,
                        ts           TimeSeries,
                        starttime    DATETIME YEAR TO FRACTION(5),
                        endtime      DATETIME YEAR TO FRACTION(5),
                        geometry     LVARCHAR,
                        max_distance REAL,
                        uom          LVARCHAR)
returns DATETIME
```

ts_tabname
The name of the time series table.

obj_id The ID of the object. Must be a value from the primary key column of the time series table. Can be the name of the column that stores the object IDs if the WHERE clause specifies a specific object ID.

ts The name of the **TimeSeries** data type.

starttime
The start of the time range. Can be NULL.

endtime
The end of the time range. Can be NULL.

geometry
: The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
: The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
: The unit of measure for the *max_distance* parameter. Default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_GetFirstTimeByPoint** function to find when an object first passed within the specified distance of the specified position during the specified time range. If you do not specify a time range, the function returns the first time that an object passed close enough to the position. If the object was too far away from the position during the time range, the **STS_GetFirstTimeByPoint** function returns NULL.

The following illustration shows the trajectory of a vehicle and the point at which the trajectory is first within the specified distance to the point (x,y). The query returns the time stamp that is associated with the point on the trajectory.

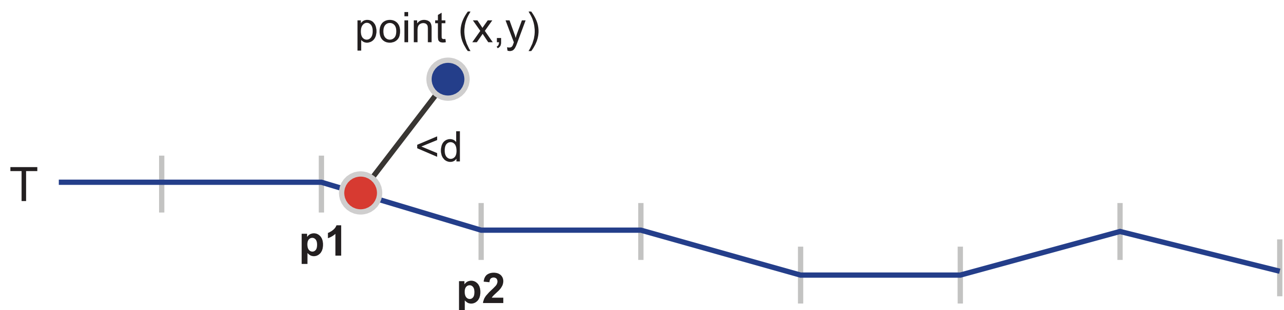


Figure 2-1. A trajectory near a point

Returns

A time stamp

NULL if the trajectory of the object during the time range was always farther than the maximum distance from the position.

Example: Find the first time that the vehicle ever passed the position

The following query returns the first time ever that vehicle 1 passed within 100 meters of the point (116.401 39.911):

```
SELECT STS_GetFirstTimeByPoint('T_Vehicle', modid, ts_track, null, null,
                                '4326 point(116.401 39.911)', 100)
       FROM T_Vehicle
       WHERE modid = '1';

(expression)
```

```
2014-02-02 13:37:15.00000

1 row(s) retrieved.
```

Example: Find the first time that the vehicle passed the position in a time range

The following query returns the first time that vehicle 1 passed within 100 meters of the point (116.401 39.911) between 2014-02-02 13:39:00 and 2014-02-02 16:30:00:

```
SELECT STS_GetFirstTimeByPoint('T_Vehicle', modid, ts_track, '2014-02-02 13:39:00',
                  '2014-02-02 16:30:00', '4326 point(116.40100 39.91100)', 100)
         FROM T_Vehicle
         WHERE modid='1';

(expression)

2014-02-02 13:40:55.00000

1 row(s) retrieved.
```

STS_GetIntersectSet function

The **STS_GetIntersectSet** function returns the set of objects whose trajectories intersect the specified region during the specified time range.

Syntax

```
STS_GetIntersectSet(ts_tabname   LVARCHAR,
                    ts_colname   LVARCHAR,
                    starttime    DATETIME YEAR TO FRACTION(5),
                    endtime      DATETIME YEAR TO FRACTION(5),
                    geometry     LVARCHAR,
                    max_distance REAL)
returns Set(LVARCHAR)

STS_GetIntersectSet(ts_tabname   LVARCHAR,
                    ts_colname   LVARCHAR,
                    starttime    DATETIME YEAR TO FRACTION(5),
                    endtime      DATETIME YEAR TO FRACTION(5),
                    geometry     LVARCHAR,
                    max_distance REAL,
                    uom          LVARCHAR)
returns Set(LVARCHAR)
```

ts_tabname
: The name of the time series table.

ts_column
: The name of the **TimeSeries** column.

starttime
: The start of the time range.

endtime
: The end of the time range.

geometry
: The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
: The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
: The unit of measure for the *max_distance* parameter. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_GetIntersectSet** function to find which objects intersected a region during a time range.

The following illustration shows a region that is intersected by three trajectories.

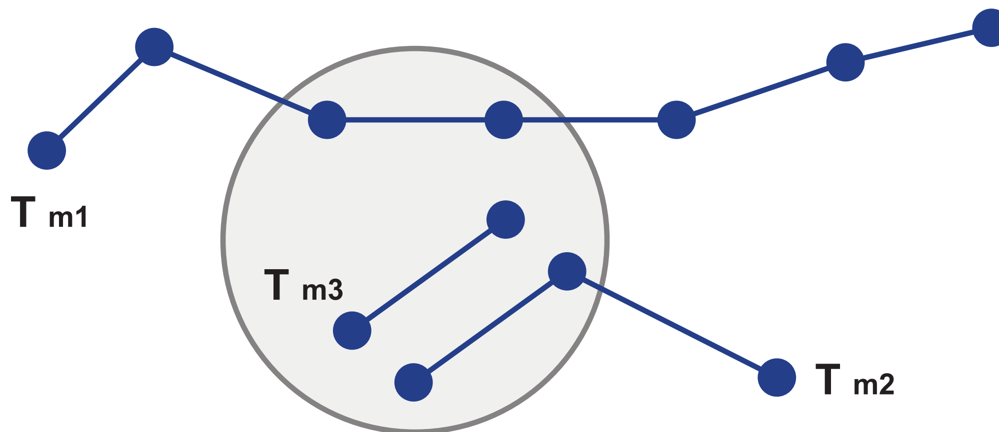


Figure 2-2. Trajectories that intersect a region

Returns

A set of object IDs.

NULL, if nothing found.

Example

The following statement returns the vehicles IDs that intersected the region within 1000 meters of the point (116.4, 39.91) during the time between 2014-02-02 13:36:00 and 2014-02-02 13:54:00:

```
EXECUTE FUNCTION STS_GetIntersectSet('T_Vehicle', 'ts_track', '2014-02-02 13:36:00',
                      '2014-02-02 13:54:00', '4326 point(116.4 39.91)', 1000);


(expression)  SET{'1','2'}

1 row(s) retrieved.
```

The query returns the IDs 1 and 2.

STS_GetLastPosition function

The **STS_GetLastPosition** function returns the location of the most recent time series element.

Syntax

```
STS_GetLastPosition(ts   TimeSeries )
returns LVARCHAR
```

ts The name of the **TimeSeries** data type.

Usage

Run the **STS_GetLastPosition** function if you want to know the location of the object with the latest time stamp.

Returns

An LVARCHAR string that represents the position of the object. The string includes the spatial reference ID and a point that consists of a longitude value and a latitude value.

NULL, if nothing found.

Example: Find the last position of a vehicle

The following statement returns the last position of the vehicle 1:

```
EXECUTE FUNCTION STS_GetLastPosition(ts_track)
     FROM T_Vehicle
     WHERE modid='1';

(expression)  4326 point(116.402010 39.908010)
```

Example: Find the last positions of all vehicles

The following statement returns the last position of all vehicles in the table:

```
SELECT modid, STS_GetLastPosition(ts_track)
    FROM T_Vehicle;


modid         1
(expression)  4326 point(116.402010 39.908010)

modid         2
(expression)  4326 point(116.402000 39.908000)

2 row(s) retrieved.
```

STS_GetLocWithinSet function

The **STS_GetLocWithinSet** function returns the set of objects whose position is within the specified region during the specified time range.

Syntax

```
STS_GetLocWithinSet(ts_tabname   LVARCHAR,
                    ts_colname   LVARCHAR,
                    timestamp    DATETIME YEAR TO FRACTION(5),
                    geometry     LVARCHAR,
                    max_distance REAL)
returns Set(LVARCHAR)

STS_GetLocWithinSet(ts_tabname   LVARCHAR,
                    ts_colname   LVARCHAR,
                    timestamp    DATETIME YEAR TO FRACTION(5),
```

```
                    geometry     LVARCHAR,
                    max_distance REAL,
                    uom           LVARCHAR)
returns Set(LVARCHAR)
```

ts_tabname
The name of the time series table.

ts_colname
The name of the **TimeSeries** column.

timestamp
The time point to query.

geometry
The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
The unit of measure for the *max_distance* parameter. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_GetLocWithinSet** function to find which objects were in a region at a specific time.

Returns

A set of object IDs.

NULL, if nothing found.

Example

The following statement returns the IDs of vehicles that were within 1000 meters of the point (116.4, 39.91) at 2014-02-02 13:36:00:

```
EXECUTE FUNCTION STS_GetLocWithinSet('T_Vehicle', 'ts_track',
                    '2014-02-02 13:36:00', '4326 point(116.4 39.91)', 1000);

(expression)  SET{'1','2'}
```

The query returns the IDs 1 and 2.

STS_GetNearestObject function

The **STS_GetNearestObject** function returns the nearest object to the specified point and within the specified distance at the specified time.

Syntax

```
STS_GetNearestObject(ts_tabname    LVARCHAR,
                     ts_colname    LVARCHAR,
                     timestamp     DATETIME YEAR TO FRACTION(5),
                     geometry      LVARCHAR,
```

```
                     max_distance  REAL)
returns LVARCHAR

STS_GetNearestObject(ts_tabname    LVARCHAR,
                     ts_colname    LVARCHAR,
                     timestamp     DATETIME YEAR TO FRACTION(5),
                     geometry      LVARCHAR,
                     max_distance  REAL,
                     uom           LVARCHAR)
returns LVARCHAR
```

ts_tabname
The name of the time series table.

ts_colname
The name of the **TimeSeries** column.

timestamp
The time point to query.

geometry
The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
The unit of measure for the *max_distance* parameter. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_GetNearestObject** function to find which object was closest to a location but within a specific distance at a specific time. If you specify that the distance from the point is 0 meters, the **STS_GetNearestObject** function returns the closest object regardless of the distance.

The following illustration shows that of the three trajectories near the point (10,10), the trajectory of object **m2** is the closest.

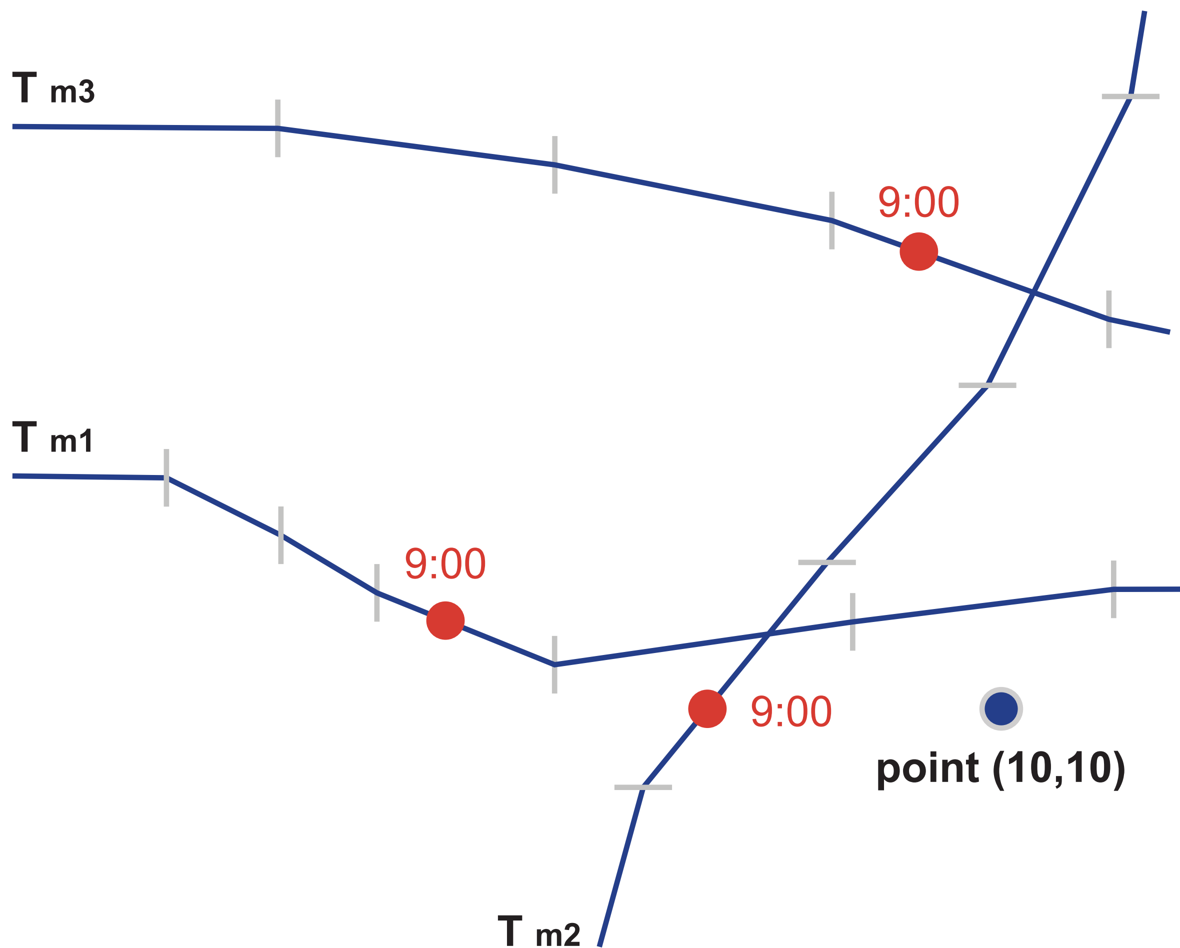


Figure 2-3. Trajectories near a point at a specific time

Returns

The object ID.

NULL, if nothing found.

Example

The following statement returns the vehicle ID whose location was the closest to the point (10,10), but within 1010 meters, at 2014-02-02 13:36:00:

```
EXECUTE FUNCTION STS_GetNearestObject('T_Vehicle', 'ts_track',
                     '2014-02-02 13:36:00', '4326 point(116.4 39.9)', 1010);


(expression)  1

1 row(s) retrieved.
```

STS_GetPosition function

The **STS_GetPosition** function returns the position of an object at a specified time.

Syntax

```
STS_GetPosition(ts          TimeSeries,
                tstamp      DATETIME YEAR TO FRACTION(5))
returns LVARCHAR
```

ts The time series.

tstamp The time stamp to query.

Usage

Run the **STS_GetPosition** function to find where a moving object was at a specific time.

Identify which object to track in the WHERE clause of the query.

The following illustration shows the trajectory of a vehicle and the position of (10, 10) at 2014-01-08 00:00:00.

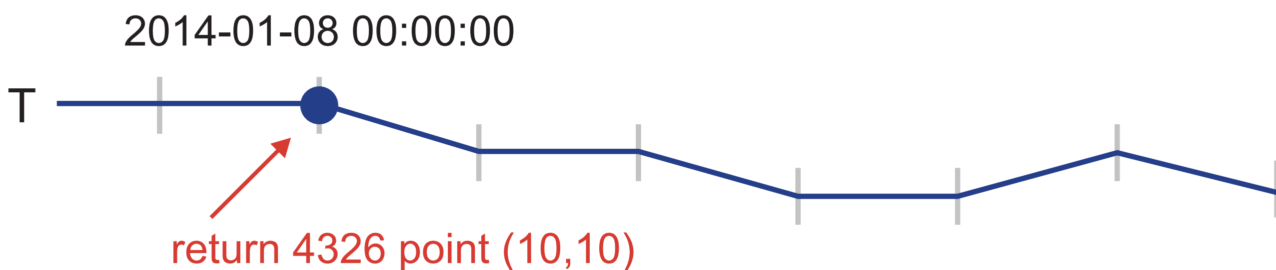


Figure 2-4. The path of a moving vehicle

Returns

An LVARCHAR string that represents the position of the object. The string includes the spatial reference ID and a point that consists of a longitude value and a latitude value.

NULL, if nothing found.

Example

The following query returns the position of the vehicle 1 at 2014-02-02 13:34:06:

```
SELECT STS_GetPosition(ts_track, '2014-02-02 13:34:06')
   FROM T_Vehicle
   WHERE modid='1';

(expression)  4326 point(116.400610 39.906050)
```

STS_GetTrajectory function

The **STS_GetTrajectory** function returns the exact trajectory of a specified object for the specified time range.

Syntax

```
STS_GetTrajectory(ts            TimeSeries,
                  starttime     DATETIME YEAR TO FRACTION(5),
                  endtime       DATETIME YEAR TO FRACTION(5))
returns LVARCHAR
```

ts The time series.

starttime
The start of the time range.

endtime
The end of the time range.

Usage

Run the **STS_GetTrajectory** function to find where an object went during a time range, which is based on the data in the time series table. The location for each time point in the range is extracted from the time series table and converted into one or more linestrings.

Identify which object to track in the WHERE clause of the query.

The following graphic illustrates a trajectory.

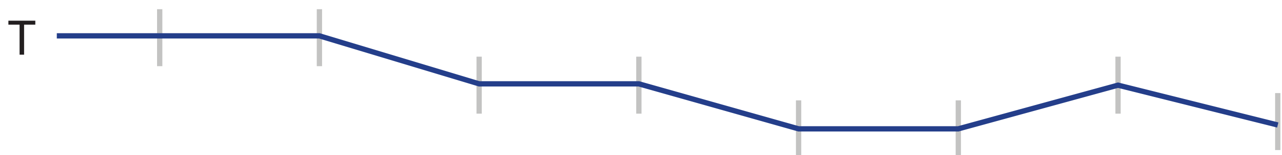


Figure 2-5. Trajectory of an object for a time range

Returns

An LVARCHAR string that represents the trajectory of the object. The string includes the spatial reference ID and a multilinestring that consists of multiple sets of longitude and latitude values.

NULL, if nothing found.

Example: Get the trajectory between specific times

The following query returns the trajectory of the vehicle 1 between 2014-02-02 13:00:00 and 2014-02-02 16:30:00:

```
SELECT STS_GetTrajectory(ts_track, '2014-02-02 13:00:00', '2014-02-02 16:30:00')
       FROM T_Vehicle
       WHERE modid='1';


(expression)  4326 multilinestring((116.400610 39.906050, 116.401210 39.913900,
              116.401170 39.911590, 116.392450 39.906350, 116.369990 39.905940,
              116.345260 39.905890))

1 row(s) retrieved.
```

Example: Get the trajectory from a specific time until the current time

The following query returns the trajectory of the vehicle 1 between 2014-02-02 13:00:00 and the current time:

```
SELECT STS_GetTrajectory(ts_track, '2014-02-02 13:00:00', current)
    FROM T_Vehicle
    WHERE modid='1';
```

```
(expression)  4326 multilinestring((116.400610 39.906050, 116.401210 39.913900,
               116.401170 39.911590, 116.392450 39.906350, 116.369990 39.905940
              , 116.345260 39.905890),(116.420000 40.100000, 116.401000 39.9070
              00, 116.402000 39.908000, 116.402010 39.908010))

1 row(s) retrieved.
```

STS_Init function

The **STS_Init** function creates internal tables and starts a Scheduler task that builds the initial spatiotemporal index, and then periodically indexes new spatiotemporal data.

Syntax

```
STS_Init(ts_tabname             VARCHAR(128)
)
returns INTEGER

STS_Init(ts_tabname             VARCHAR(128)
       task_frequency         INTERVAL DAY TO SECOND default "0 01:00:00",
       task_starttime         DATETIME HOUR TO SECOND default NULL,
       ts_default_starttime   DATETIME YEAR TO SECOND default "1970-01-01 00:00:00",
       ts_interval_to_process INTERVAL DAY TO SECOND default "0 01:00:00",
       ts_interval_to_avoid   INTERVAL DAY TO SECOND default "1 00:00:00"
)
returns INTEGER
```

ts_tabname
: The name of the time series table.

task_frequency **(Optional)**
: How frequently to index new data. Default is every hour.

task_starttime **(Optional)**
: The first time to start the task. Default is NULL, which means to start the task when the **STS_Init** function is run.

ts_default_starttime **(Optional)**
: The first time stamp in the time series from which to index. Default is 1970-01-01 00:00:00.

ts_interval_to_process **(Optional)**
: The time interval in the time series to process each time that the task is run. Default is one hour. Set to a value that takes less time to index than the value of the *task_frequency* parameter.

ts_interval_to_avoid **(Optional)**
: The indexing lag time. The time interval in the time series before the current time to avoid indexing. Default is one day.

Usage

Run the **STS_Init** function to start the indexing process by creating internal spatiotemporal search tables and starting a Scheduler task for the specified table. The Scheduler task, which has a prefix of **autosts**, starts at the specified time, indexes the initial set of data, and periodically indexes new data. The task prints messages in the database server message log when indexing starts and completes.

If spatiotemporal search indexing is already running for the specified table, you can run the **STS_Init** function to change the properties of the Scheduler task.

The Scheduler task is started at the time that is specified by the *task_starttime* parameter. The first run of the task processes the data in the time interval that is defined by the value of the *ts_default_starttime* parameter to the time calculated by subtracting the value of the *ts_interval_to_avoid* parameter from the current time. The end time of the processing interval is saved in the internal lasttime table. Subsequent runs of the task start based on the value of the *task_frequency* parameter and index the data between the last end time that is saved in the lasttime table and the earlier of the following times:

- The last end time plus the value of the *ts_interval_to_process* parameter
- The current time minus the value of the *ts_interval_to_avoid* parameter

Any data that you insert with timepoints that are earlier than the last end time that is saved in the lasttime table are not indexed.

Important: If you run the task the first time on an empty time series, the recorded last end time is the current time minus the value of the *ts_interval_to_avoid* parameter. Any data that you insert with earlier timepoints are not indexed.

Returns

An integer that indicates the status of the function:

- 0 = The Scheduler task for spatiotemporal search indexing is started.
- 1 = An error occurred.

Example

The following statement is run at 2015-02-01 08:00:00 to start spatiotemporal search indexing on the table that is named **T_Vehicle**:

```
EXECUTE FUNCTION STS_Init('T_Vehicle');
```

The Scheduler task is created with default values for the **T_Vehicle** table. The task runs for the first time at 08:00:00 and processes the time series data with timepoints between 1970-01-01 00:00:00 and 2015-01-31 08:00:00. The last end time of 2015-01-31 08:00:00 is recorded in the lasttime table. The task takes about 30 minutes to index the data. The task runs again at 09:00:00 and indexes data with timepoints between 2015-01-31 08:00:00 and 2015-01-31 09:00:00. The last end time of 2015-01-31 09:00:00 is recorded in the lasttime table. Any data with timepoints earlier than 2015-01-31 08:00:00 that was inserted after the first task was run is not indexed.

Related concepts:

"Spatiotemporal search indexing" on page 1-8

STS_Release function

The STS_Release function returns the internal version number and build date for the spatiotemporal search extension.

Syntax

```
STS_Release()
Returns LVARCHAR;
```

Returns

A string with the version number and build date.

Example

The following statement returns the version number and build date:

```
EXECUTE FUNCTION STS_Release;
```

STS_Set_Trace procedure

The **STS_Set_Trace** procedure enables tracing and sets the tracing file.

Syntax

```
STS_Set_Trace(trace_params   LVARCHAR,
              trace_file     LVARCHAR);
```

trace_params
: The tracing parameters in the following format: *tracing_type tracing_level*:

  tracing_type
  : **STSQuery**: Set tracing on spatiotemporal queries.

    STSBuild: Set tracing on spatiotemporal indexing.

tracing_level
: 0 = Turn off tracing.

  An integer greater than 1 = Turn on tracing.

trace_file
: The full path and name of the tracing file.

Usage

Run the **STS_Set_Trace** procedure with the **STSQuery** value to enable tracing if you want to view the entry points of spatiotemporal query functions. Run the **STS_Set_Trace** procedure with the **STSBuild** value to enable tracing if you want to view the entry points of spatiotemporal indexing functions. You must specify the full path and name of the tracing file.

Example: Set query tracing

The following statement starts tracing on spatiotemporal queries and sets the tracing file name and path:

```
EXECUTE PROCEDURE STS_Set_Trace('STSQuery 2', '/tms/sts_query.log');
```

Example: Stop query tracing

The following statement stops tracing on spatiotemporal queries:

```
EXECUTE PROCEDURE STS_Set_Trace('STSQuery 0', '/tms/sts_query.log');
```

STS_TrajectoryCross function

The **STS_TrajectoryCross** function indicates whether the trajectory of a specified object crosses the specified region during the specified time range.

Syntax

```
STS_TrajectoryCross(ts_tabname   LVARCHAR,
                    obj_id       LVARCHAR,
                    ts_colname   LVARCHAR,
                    starttime    DATETIME YEAR TO FRACTION(5),
                    endtime      DATETIME YEAR TO FRACTION(5),
```

```
                    geometry     LVARCHAR,
                    max_distance REAL)
returns Boolean

STS_TrajectoryCross(ts_tabname   LVARCHAR,
                    obj_id       LVARCHAR,
                    ts_colname   LVARCHAR,
                    starttime    DATETIME YEAR TO FRACTION(5),
                    endtime      DATETIME YEAR TO FRACTION(5),
                    geometry     LVARCHAR,
                    max_distance REAL,
                    uom              LVARCHAR)
returns Boolean
```

ts_tabname
: The name of the time series table.

obj_id
: The ID of the object. Must be a value from the primary key column of the time series table. Can be the name of the column that stores the object IDs if the WHERE clause specifies a specific object ID.

ts_colname
: The name of the **TimeSeries** column.

starttime
: The start of the time range.

endtime
: The end of the time range.

geometry
: The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
: The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
: The unit of measure for the *max_distance* parameter. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_TrajectoryCross** function to know whether an object crossed the boundary of a specific region during a time range. The **STS_TrajectoryCross** function returns t if the object crossed the boundary of the region one or more times during the time range. The **STS_TrajectoryCross** function returns f if the object remained either outside or inside of the region for the time range.

Returns

t if the object crossed the boundary of the region during the time range.

f if the object did not cross the boundary of the region during the time range.

Example

The following query returns whether vehicle 1 crossed the boundary of the region, which is specified by the point (116.4, 39.91) and the distance of 1000 meters, between 2014-02-02 13:34:00 and 2014-02-02 13:54:00:

```
SELECT STS_TrajectoryCross('T_Vehicle', modid, 'ts_track', '2014-02-02 13:34:00',
                           '2014-02-02 13:54:00', '4326 point(116.4 39.91)', 1000)
        FROM T_Vehicle
        WHERE modid = '1';

(expression)

           t

1 row(s) retrieved.
```

The following illustration shows a trajectory that crosses the boundary of the region.

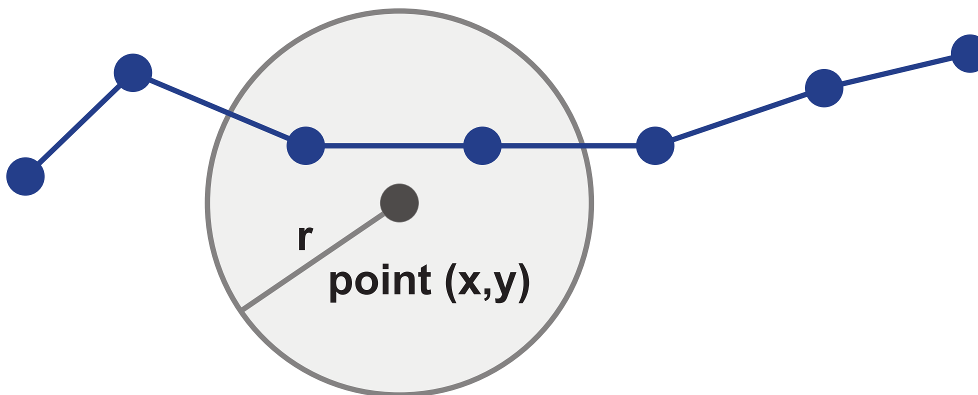


Figure 2-6. A trajectory that crosses the region boundary twice

STS_TrajectoryDistance function

The **STS_TrajectoryDistance** function returns the shortest distance between the specified point and the trajectory of the specified moving object in the specified time range.

Syntax

```
STS_TrajectoryDistance(ts_tabname      LVARCHAR,
                       obj_id          LVARCHAR,
                       ts_colname      LVARCHAR,
                       starttime       DATETIME YEAR TO FRACTION(5),
                       endtime         DATETIME YEAR TO FRACTION(5),
                       geometry        LVARCHAR)
returns FLOAT

STS_TrajectoryDistance(ts_tabname      LVARCHAR,
                       obj_id          LVARCHAR,
                       ts_colname      LVARCHAR,
                       starttime       DATETIME YEAR TO FRACTION(5),
                       endtime         DATETIME YEAR TO FRACTION(5),
                       geometry        LVARCHAR,
                       uom             LVARCHAR)
returns FLOAT
```

ts_tabname
The name of the time series table.

obj_id The ID of the object. Must be a value from the primary key column of the time series table. Can be the name of the column that stores the object IDs if the WHERE clause specifies a specific object ID.

ts_colname
The name of the **TimeSeries** column.

starttime
The start of the time range. Can be NULL.

endtime
The end of the time range. Can be NULL.

geometry
The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

uom **(Optional)**
The unit of measure for the return value. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_TrajectoryDistance** function to find how close and object came to a specific point during a time range.

The following illustration shows a trajectory near the point (20,20). A line connects the closest part of the trajectory to the point.

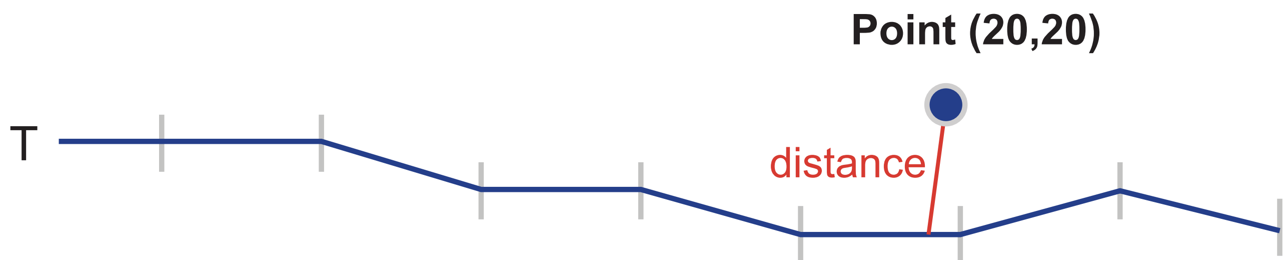


Figure 2-7. The shortest distance between a trajectory and a point

Returns

A FLOAT value that represents the distance in the specified unit of measure.

NULL, if nothing found.

Example

The following query returns the shortest distance in meters between the trajectory of vehicle 1 and the point (116.4. 39.9) between 2014-02-02 13:35:00 and 2014-02-02 13:54:00:

```
SELECT STS_TrajectoryDistance('T_Vehicle', modid, 'ts_track', '2014-02-02 13:35:00',
              '2014-02-02 13:54:00', '4326 point(116.4 39.9)')::decimal(10,2)
     FROM T_Vehicle
     WHERE modid='1';

(expression)
```

```
     830.36

1 row(s) retrieved.
```

STS_TrajectoryIntersect function

The **STS_TrajectoryIntersect** function indicates whether the trajectory of an object intersected the region during the time range.

Syntax

```
STS_TrajectoryIntersect(ts_tabname   LVARCHAR,
                        obj_id       LVARCHAR,
                        ts_colname   LVARCHAR,
                        starttime    DATETIME YEAR TO FRACTION(5),
                        endtime      DATETIME YEAR TO FRACTION(5),
                        geometry     LVARCHAR,
                        max_distance REAL)
returns Boolean

STS_TrajectoryIntersect(ts_tabname   LVARCHAR,
                        obj_id       LVARCHAR,
                        ts_colname   LVARCHAR,
                        starttime    DATETIME YEAR TO FRACTION(5),
                        endtime      DATETIME YEAR TO FRACTION(5),
                        geometry     LVARCHAR,
                        max_distance REAL,
                        uom          LVARCHAR)
returns Boolean
```

ts_tabname
: The name of the time series table.

obj_id
: The ID of the object. Must be a value from the primary key column of the time series table. Can be the name of the column that stores the object IDs if the WHERE clause specifies a specific object ID.

ts_colname
: The name of the **TimeSeries** column.

starttime
: The start of the time range.

endtime
: The end of the time range.

geometry
: The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
: The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
: The unit of measure for the *max_distance* parameter. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_TrajectoryIntersect** function to find out whether the specified object went through the specified region during the specified time range. Intersection means either crossed the boundary of the region or remained within the boundary of the region.

The following image shows a trajectory that crosses the boundary of a region.

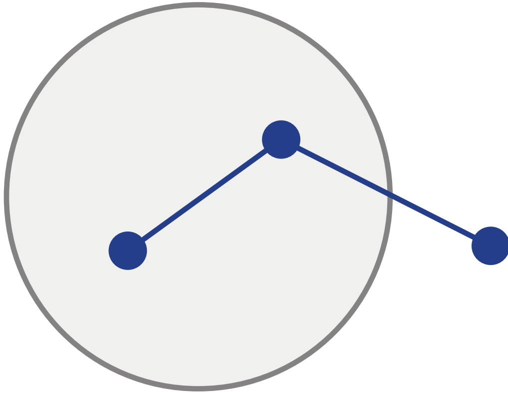


Figure 2-8. A trajectory that crosses a region

The following image shows a trajectory that is entirely within a region.

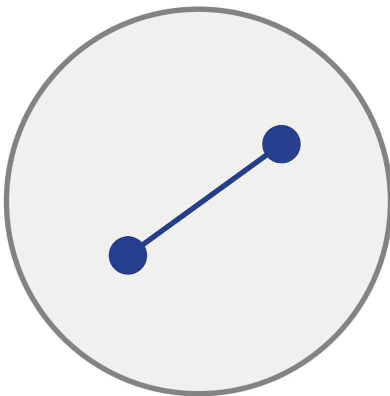


Figure 2-9. A trajectory within a region

Returns

`t` if the trajectory of the object crossed the boundary of the region during the time range.

`t` if the trajectory of the object remained within the region during the time range.

`f` if the trajectory of the object did not intersect the region during the time range.

Example

The following query returns whether vehicle 1 intersected the boundary of the region, which is described by the point (116.4, 39.91) and the distance of 1000 meters, between 2014-02-02 13:34:00 and 2014-02-02 13:54:00:

```
SELECT STS_TrajectoryIntersect('T_Vehicle', modid, 'ts_track',
                      '2014-02-02 13:34:00', '2014-02-02 13:54:00',
                      '4326 point(116.4 39.91)', 1000)
       FROM T_Vehicle
       WHERE modid = '1';

(expression)

          t

1 row(s) retrieved.
```

STS_TrajectoryWithin function

The **STS_TrajectoryWithin** function indicates whether the trajectory of a specified object stayed within the specified region during the specified time range.

Syntax

```
STS_TrajectoryWithin(ts_tabname   LVARCHAR,
                     obj_id       LVARCHAR,
                     ts_colname   LVARCHAR,
                     starttime    DATETIME YEAR TO FRACTION(5),
                     endtime      DATETIME YEAR TO FRACTION(5),
                     geometry     LVARCHAR,
                     max_distance REAL)
returns Boolean

STS_TrajectoryWithin(ts_tabname   LVARCHAR,
                     obj_id       LVARCHAR,
                     ts_colname   LVARCHAR,
                     starttime    DATETIME YEAR TO FRACTION(5),
                     endtime      DATETIME YEAR TO FRACTION(5),
                     geometry     LVARCHAR,
                     max_distance REAL,
                     uom          LVARCHAR)
returns Boolean
```

ts_tabname
The name of the time series table.

obj_id The ID of the object. Must be a value from the primary key column of the time series table. Can be the name of the column that stores the object IDs if the WHERE clause specifies a specific object ID.

ts_colname
The name of the **TimeSeries** column.

starttime
The start of the time range. Can be NULL.

endtime
The end of the time range. Can be NULL.

geometry
The geometry at the center of the region of interest. Can be an ST_Point, ST_MultiPoint, ST_LineString, ST_MultiLineString, ST_Polygon, or ST_MultiPolygon. Must use the SRID 4326.

max_distance
The distance from the *geometry* that defines the border of the region of interest. The unit of measure is specified by the *uom* parameter.

uom **(Optional)**
The unit of measure for the *max_distance* parameter. The default is meters. Must be the name of a linear unit of measure from the **unit_name** column of the **st_units_of_measure** table.

Usage

Run the **STS_TrajectoryWithin** function to find out whether an object stayed within a region during the entire time range. The **STS_TrajectoryWithin** function returns `f` if the object was in the region for only part of the time range or if the object was never in the region during the time range.

Returns

`t` if the trajectory of the object was within the region during the entire time range.

`f` if the trajectory of the object was within the region for only part of the time range.

`f` if the trajectory of the object was not within the region during the time range.

Example

The following query returns whether vehicle 1 stayed within 1000 meters of the point (116.4, 39.91) between 2014-02-02 13:34:00 and 2014-02-02 13:54:00:

```
SELECT STS_TrajectoryWithin('T_Vehicle', modid, 'ts_track', '2014-02-02 13:34:00',
                            '2014-02-02 13:54:00', '4326 point(116.4 39.91)', 1000)
      FROM T_Vehicle
      WHERE modid='1';

(expression)

           f

1 row(s) retrieved.
```

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

IBM and accessibility

For more information about the IBM commitment to accessibility, see the *IBM Accessibility Center* at http://www.ibm.com/able.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number `3.1`) are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number `3` is followed by a series of syntax elements with dotted decimal number `3.1`, all the syntax elements numbered `3.1` are subordinate to the syntax element numbered `3`.

© Copyright IBM Corp. 2015

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be

repeated. For example, if you hear the line 5.1* data-area, you know that you can include more than one data area or you can include none. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. This material may be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

© Copyright IBM Corp. 2015

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies", and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

Accessibility A-1
- dotted decimal format of syntax diagrams A-1
- keyboard A-1
- shortcut keys A-1
- syntax diagrams, reading in a screen reader A-1

C

compliance with standards vi

D

Disabilities, visual
- reading syntax diagrams A-1

Disability A-1
Dotted decimal format of syntax diagrams A-1

F

Functions
- STS_Cleanup() 2-2
- STS_GetCompactTrajectory() 2-3
- STS_GetFirstTimeByPoint() 2-4
- STS_GetIntersectSet() 2-6
- STS_GetLastPosition() 2-8
- STS_GetLocWithinSet() 2-8
- STS_GetNearestObject() 2-9
- STS_GetPosition() 2-12
- STS_GetTrajectory() 2-12
- STS_Init() 2-14
- STS_Release() 2-15
- STS_TrajectoryCross() 2-16
- STS_TrajectoryDistance() 2-18
- STS_TrajectoryIntersect() 2-20
- STS_TrajectoryWithin() 2-22

I

industry standards vi

P

Procedure
- STS_Set_Trace() 2-16

R

Requirements
- database 1-7
- table 1-7
- time series definition 1-7
- TimeSeries subtype 1-7

Routines for spatiotemporal search 2-1

S

Screen reader
- reading syntax diagrams A-1

Shortcut keys
- keyboard A-1

Spatial data types 1-11
Spatial extension 1-3
Spatial solution 1-2
Spatiotemporal search
- architecture 1-2
- disabling 1-12
- example 1-5
- indexing 1-8
- overview 1-1
- preparing 1-4
- queries 1-8
- registration 1-3
- requirements 1-3
- restrictions 1-3
- Schema requirements 1-7
- spatial data types 1-11

Spatiotemporal search routines 2-1
standards vi
STS_Cleanup() function 2-2
STS_GetCompactTrajectory() function 2-3
STS_GetFirstTimeByPoint() function 2-4
STS_GetIntersectSet() function 2-6
STS_GetLastPosition() function 2-8
STS_GetLocWithinSet() function 2-8
STS_GetNearestObject() function 2-9
STS_GetPosition() function 2-12
STS_GetTrajectory() function 2-12
STS_Init() function 2-14
STS_Release() function 2-15
STS_Set_Trace() procedure 2-16
STS_TrajectoryCross() function 2-16
STS_TrajectoryDistance() function 2-18
STS_TrajectoryIntersect() function 2-20
STS_TrajectoryWithin() function 2-22
Syntax diagrams
- reading vi
- reading in a screen reader A-1

T

TimeSeries extension 1-3
TimeSeries solution 1-2

V

Visual disabilities
- reading syntax diagrams A-1

© Copyright IBM Corp. 2015

Printed in USA

SC27-8019-00