Informix Product Family
Informix
Version 11.70

IBM Informix TimeSeries Data User's Guide

IBM

SC27-3567-02

Informix Product Family
Informix
Version 11.70

IBM Informix TimeSeries Data User's Guide

SC27-3567-02

Note

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

This edition replaces SC27-3567-01.

Contents

Introduction

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

About this publication

This publication contains information to assist you in using the time series data types and supporting routines.

These topics discuss the organization of the publication, the intended audience, and the associated software products that you must have to develop and use time series.

Types of users

This publication is written for the following audience:
- Developers who write applications to access time series information stored in IBM® Informix® databases

Assumptions about your locale

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

The IBM Informix OLE DB Provider follows the ISO string formats for date, time, and money, as defined by the Microsoft OLE DB standards. You can override that default by setting an Informix environment variable or registry entry, such as **DBDATE**.

If you use Simple Network Management Protocol (SNMP) in your Informix environment, note that the protocols (SNMPv1 and SNMPv2) recognize only English code sets. For more information, see the topic about GLS and SNMP in the *IBM Informix SNMP Subagent Guide*.

The examples in this publication are written with the assumption that you are using one of these locales: en_us.8859-1 (ISO 8859-1) on UNIX platforms or en_us.1252 (Microsoft 1252) in Windows environments. These locales support U.S. English format conventions for displaying and entering date, time, number, and currency values. They also support the ISO 8859-1 code set (on UNIX and Linux) or the Microsoft 1252 code set (on Windows), which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

You can specify another locale if you plan to use characters from other locales in your data or your SQL identifiers, or if you want to conform to other collation rules for character data.

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

What's new in TimeSeries data for Informix, Version 11.70

This publication includes information about new features and changes in existing functions.

For a complete list of what's new in this release, see the release notes or the information center at http://publib.boulder.ibm.com/infocenter/idshelp/v117/topic/com.ibm.po.doc/new_features.htm.

Table 1. What's New in IBM Informix TimeSeries Data User's Guide for 11.70.xC4

Overview	Reference
IBM Informix TimeSeries Plug-in for Data Studio You can easily load data from an input file into an Informix table with a TimeSeries column by using IBM Informix TimeSeries Plug-in for Data Studio. You can also use the plug-in with IBM Optim™ Developer Studio.	"Installing the IBM Informix TimeSeries Plug-in for Data Studio" on page 1-13 "Loading data with the IBM Informix TimeSeries Plug-in for Data Studio" on page 3-16 "Example: Create and load a time series" on page 3-1
Aggregate time series data across multiple rows You can use a single TimeSeries function, **TSRollup**, to aggregate time series values by time for multiple rows in the table and return a time series that contains the results. Previously, you could aggregate time series values only for each row individually. For example, if you have a table that contains information about energy consumption for the meters attached to a specific energy concentrator, you can aggregate the values for all the meters and sum the values for specific time intervals to get a single total for each interval. The resulting time series represents the total energy consumption for each time interval for that energy concentrator.	"TSRollup function" on page 7-98
Delete a range of elements and free empty pages from a time series You can delete elements in a time series from a specified time range and free any resulting empty pages by using the **DelRange** function. The **DelRange** function is similar to the **DelTrim** function; however, unlike the **DelTrim** function, the **DelRange** function frees pages in any part of the range of deleted elements. You can free empty pages that have only null elements from a time series for a specified time range or throughout the time series by using the **NullCleanup** function.	"DelRange function" on page 7-34 "NullCleanup function" on page 7-63

Table 2. What's New in IBM Informix TimeSeries Data User's Guide for 11.70.xC3

Overview	Reference
Time series storage management Time series data that is too large to fit into a row in a table is stored in time series containers. You do not need to create a container before you insert data into a time series or specify a container name when you insert data into a time series. Containers for **TimeSeries** subtypes are created automatically in the same dbspaces in which the table that contains the subtype is stored. You can also create custom pools of containers specific to your needs.	"Time series data storage" on page 1-8 "Managing containers" on page 3-7
Monitor time series containers You can monitor the containers that store time series data to obtain the following information for a specific container or for all containers in the database: • The number of allocated pages • The number of pages containing time series data • The percentage of used space • The number of time series data elements	"Monitoring time series containers" on page 3-8
View the results of a time series expression in a virtual table You can create a virtual table based on the results of a time series expression, such as the **AggregateBy** function. Previously you needed to save the results of the expression in an intermediate table and create a virtual table based on the intermediate table.	"TSCreateExpressionVirtualTab procedure" on page 4-8
Output time series data in XML format You can produce an XML representation of a time series by using the new **TSToXML** function. You can use the XML data to send time series information to other applications.	"TSToXML function" on page 7-108
Time series data in the **stores_demo** database You can use the new time series tables in the **stores_demo** database to experiment with time series data by running SQL queries and time series routines. The **stores_demo** database has three new tables that contain smart meter time series data.	"Sample smart meter data" on page 1-15
Predefined calendars for time series You can use one of the seven predefined calendars when you create a time series instead of creating your own calendar. The calendars start at the beginning of 2011. The calendar patterns have interval durations of one minute, 15 minutes, 30 minutes, one hour, one day, one week, and one month.	"Predefined calendars" on page 3-5
Run the **Transpose** function in table expressions You can use the **Transpose** function in table expressions to return time series data in a tabular format that is easy to read.	"Transpose function" on page 7-77

Table 2. What's New in IBM Informix TimeSeries Data User's Guide for 11.70.xC3 (continued)

<table>
<tr><th>Overview</th><th>Reference</th></tr>
<tr><td>Easier time-series data updates from virtual tables

When you update time series data from a virtual table, by default only the primary key is used to find the row to update. You do not need to provide accurate values for columns that are not part of the primary key. Previously, all columns except the TimeSeries subtype column were used by the virtual table to identify the row to update.

When you create virtual tables, you can configure the update behavior:
• Update the values of columns in an existing row that are not part of the primary key. (To prevent updating a non-primary key column, set it to NULL in the INSERT statement.)
• Update the value of all the columns in an existing row that are not part of the primary key. Update columns that allow null values to NULL.</td><td>“The TSVTMode parameter” on page 4-11</td></tr>
<tr><td>Container name and TimeSeries subtype names extended to 128 bytes

The maximum length of container names and TimeSeries subtype names is 128 bytes. Previously, the maximum length of the names was 18 bytes.</td><td>“TSContainerCreate procedure” on page 7-83

“Creating a TimeSeries subtype” on page 3-6</td></tr>
<tr><td>Time series tables and containers can use non-default page sizes

You can now store time series tables and containers in dbspaces that use non-default page sizes. Previously, all time series tables and containers had to be stored in dbspaces with the default page size.</td><td>“TSContainerCreate procedure” on page 7-83

“Create the database table” on page 3-6</td></tr>
<tr><td>Faster deleting of time series data

When you delete time series data, the performance is faster than in previous releases. You can delete large amounts of time series data in less time.</td><td></td></tr>
</table>

Table 2. What's New in IBM Informix TimeSeries Data User's Guide for 11.70.xC3 (continued)

Overview	Reference
Informix TimeSeries Plug-in for OAT The IBM Informix TimeSeries Plug-in for OpenAdmin Tool (OAT) provides a graphical interface for reviewing and administering the **TimeSeries** data type provided by the Informix TimeSeries extension. A time series is a set of data recorded as it varies over time. With the TimeSeries plug-in, you can monitor the database objects related to your time series: • Review the **TimeSeries** subtypes, containers, and calendars that are used for the time series data in a database. • Review the tables and indexes that contain **TimeSeries** subtypes. • Review the columns and virtual tables for tables that contain **TimeSeries** subtypes. • Monitor the percentage of the space that is used in the containers and in the dbspaces for the containers. You can also create and drop containers.	

Table 3. What's New in IBM Informix TimeSeries Data User's Guide for 11.70.xC1

Overview	Reference
Time series data types and functions are built-in and automatically registered You can use the data types and functions of the TimeSeries extension (which was formerly known as the TimeSeries DataBlade® module) without performing some of the previously required prerequisites tasks, such as installing or registering the TimeSeries extension. If you are using a previous version of the IBM Informix TimeSeries DataBlade Module, when you install Informix 11.70, the TimeSeries extension is installed and registered automatically. You do not need to perform any actions to upgrade the DataBlade module, nor do you need to unload and load time series data during migration.	
New editions and product names IBM Informix Dynamic Server editions were withdrawn and new Informix editions are available. Some products were also renamed. The publications in the Informix library pertain to the following products: • IBM Informix database server, formerly known as IBM Informix Dynamic Server (IDS) • IBM OpenAdmin Tool (OAT) for Informix, formerly known as OpenAdmin Tool for Informix Dynamic Server (IDS) • IBM Informix SQL Warehousing Tool, formerly known as Informix Warehouse Feature	For more information about the Informix product family, go to http://www.ibm.com/software/data/informix/.

Example code conventions

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

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

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

Additional documentation

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

You can access or install the product documentation from the Quick Start CD that is shipped with Informix products. To get the most current information, see the Informix information centers at ibm.com®. You can access the information centers and other Informix technical information such as 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.

The IBM Informix Geodetic DataBlade Module supports a subset of the data types from the *Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173*, as referenced by the document *Content Standard for Geospatial Metadata*, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

Syntax diagrams

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

Table 4. Syntax Diagram Components

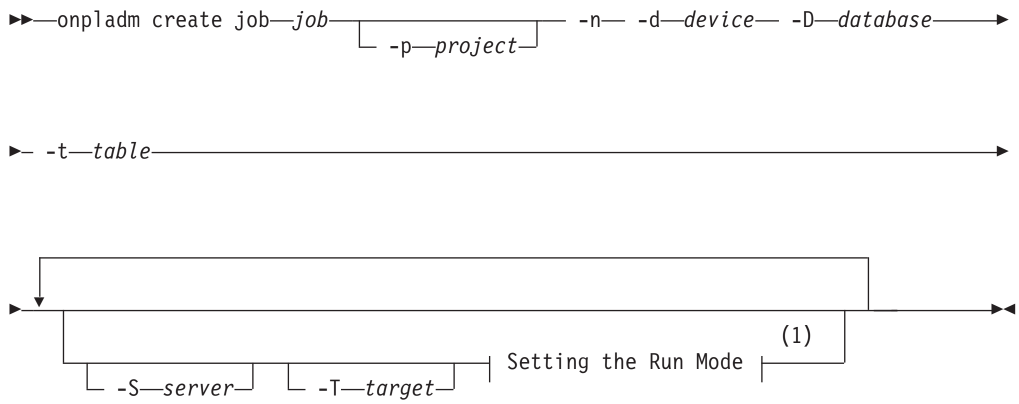
<table>
<tr><th>Component represented in PDF</th><th>Component represented in HTML</th><th>Meaning</th></tr>
<tr><td>▸▸────────────</td><td>>>----------------------</td><td>Statement begins.</td></tr>
<tr><td>──────────────▸</td><td>----------------------></td><td>Statement continues on next line.</td></tr>
<tr><td>▸────────────</td><td>>-----------------------</td><td>Statement continues from previous line.</td></tr>
<tr><td>────────────▸◂</td><td>-----------------------><</td><td>Statement ends.</td></tr>
<tr><td>SELECT</td><td>--------SELECT----------</td><td>Required item.</td></tr>
<tr><td>LOCAL</td><td>--+------------------+---
  '------LOCAL-------'</td><td>Optional item.</td></tr>
<tr><td>ALL
DISTINCT
UNIQUE</td><td>---+-----ALL-------+---
   +--DISTINCT-----+
   '---UNIQUE------'</td><td>Required item with choice. Only one item must be present.</td></tr>
<tr><td>FOR UPDATE
FOR READ ONLY</td><td>---+-------------------+---
    +--FOR UPDATE-----+
    '--FOR READ ONLY--'</td><td>Optional items with choice are shown below the main line, one of which you might specify.</td></tr>
<tr><td>NEXT
PRIOR
PREVIOUS</td><td>   .---NEXT---------.
----+----------------+---
    +---PRIOR--------+
    '---PREVIOUS-----'</td><td>The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line is used by default.</td></tr>
<tr><td>,
index_name
table_name</td><td> .-------,-----------.
 V |
---+----------------+---
    +---index_name---+
    '---table_name---'</td><td>Optional items. Several items are allowed; a comma must precede each repetition.</td></tr>
<tr><td>▸▸─┤Table Reference├─▸◂</td><td>>>-| Table Reference |-><</td><td>Reference to a syntax segment.</td></tr>
<tr><td>Table Reference
view
table
synonym</td><td>Table Reference
|--+-----view--------+--|
   +------table------+
   '----synonym------'</td><td>Syntax segment.</td></tr>
</table>

How to read a command-line syntax diagram

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

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

Creating a no-conversion job

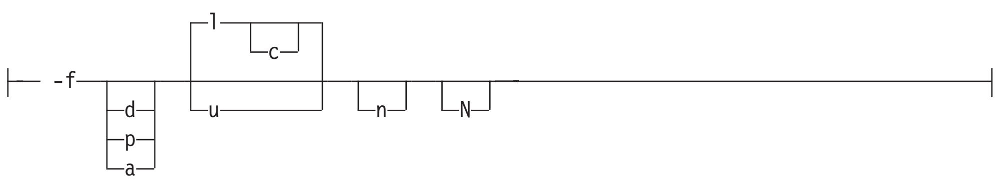


Notes:

1 See page Z-1

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

Setting the run mode:



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

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

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table

4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

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

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

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

Identifiers and names

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

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

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

```
►►──SELECT──column_name──FROM──table_name──────────────────────────────────►◄
```

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

How to provide documentation feedback

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

Use one of the following methods:
- Send email to docinf@us.ibm.com.
- In the Informix information center, which is available online at http://www.ibm.com/software/data/sw-library/, open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.

- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

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. Informix TimeSeries solution

Database administrators and applications developers use the Informix TimeSeries solution to store and analyze time series data.

A *time series* is a set of time-stamped data. Types of time series data vary enormously, for example, electricity usage that is collected from smart meters, stock price and trading volumes, ECG recordings, seismograms, and network performance records. The types of queries performed on time series data typically include a time criteria and often include aggregations of data over a longer period of time. For example, you might want to know which day of the week your customers use the most electricity.

The Informix TimeSeries solution provides the following capabilities to store and analyze time series data:

- Define the structure of the data
- Control when and how often data is accepted:
  - Set the frequency for regularly spaced records
  - Handle arbitrarily spaced records
- Control data storage:
  - Specify where to store data
  - Change where data is stored
  - Monitor storage usage
- Load data from a file or individually
- Query data:
  - Extract values for a time range
  - Find null data
  - Modify data
  - Display data in standard relational format
- Analyze data:
  - Perform statistical and arithmetic calculations
  - Aggregate data over time
  - Make data visible or invisible
  - Find the intersection or union of data

The Informix TimeSeries solution stores time series data in a special format within a relational database in a way that takes advantage of the benefits of both non-relational and standard relational implementations of time series data.

The Informix TimeSeries solution is more flexible than non-relational time series implementations because the Informix TimeSeries solution is not specific to any industry, is easily customizable, and can combine time series data with information in relational databases.

The Informix TimeSeries solution loads and queries time stamped data faster, requires less storage space, and provides more analytical capability than a standard relational table implementation. Although relational database management systems can store time series data for standard types by storing one row per time-stamped data entry, performance is poor and storage is inefficient. The Informix TimeSeries solution saves disk space by not storing duplicate information from the columns that do not contain the time-based data. The Informix TimeSeries solution loads and queries time series data quickly because the data is stored on disk in order by time stamp and by source.

For example, the following table shows a relational table that contains time-based information for two sources, or customers, whose identifiers are 1000111 and 1046021.

Table 1-1. Relational table with time-based data

Customer	Time	Value
1000111	2011-1-1 00:00:00.00000	0.092
1000111	2011-1-1 00:15:00.00000	0.082
1000111	2011-1-1 00:30:00.00000	0.090
1000111	2011-1-1 00:45:00.00000	0.085
1046021	2011-1-1 00:00:00.00000	0.041
1046021	2011-1-1 00:15:00.00000	0.041
1046021	2011-1-1 00:30:00.00000	0.040
1046021	2011-1-1 00:45:00.00000	0.041

The following table shows a representation of the same data stored in an Informix TimeSeries table. The information about the customer is stored once. All the time-based information for a source is customer together in a single row.

Table 1-2. Informix TimeSeries table with time-based data

Customer	Time	Value
1000111	2011-1-1 00:00:00.00000	0.092
	2011-1-1 00:15:00.00000	0.082
	2011-1-1 00:30:00.00000	0.090
	2011-1-1 00:45:00.00000	0.085
1046021	2011-1-1 00:00:00.00000	0.041
	2011-1-1 00:15:00.00000	0.041
	2011-1-1 00:30:00.00000	0.040
	2011-1-1 00:45:00.00000	0.041

The following table summarizes the advantages of using the Informix TimeSeries solution for time-based data over using a standard relational table.

Table 1-3. Comparison of time series data stored in a standard relational table and in an Informix TimeSeries table

	Standard relational table issue	**Informix TimeSeries table benefit**
Storage space	Stores one row for every record. Duplicates the information in non-time series columns. Stores timestamps. Null data takes as much space as actual data. The index typically includes the time stamp column and several other columns.	Significant reduction in disk space needed to store the same data. The index size on disk is also smaller. Stores all time series data for a single source in the same row. No duplicate information. Calculates instead of stores the time stamp. Null data does not require any space. The index does not include the time stamp column.
Query speed	Data for a single source can be intermixed on multiple data pages in no particular order.	Queries that use a time criteria require many fewer disk reads and significantly less I/O. Data is loaded very efficiently. Data for a single source is stored together in time stamp order.
Query complexity	Queries that aggregate data or apply an expression can be difficult or impossible to perform with SQL. Much of the query logic must be provided by the application.	Less application coding and faster queries. Allows complex SQL queries and analysis. Allows custom analytics written using the TimeSeries API.

Informix TimeSeries solution architecture

The Informix TimeSeries solution consists of built-in data types and routines. You can use other Informix tools to administer and load time series data.

The Informix database server includes the following functionality for managing time series data:

- The TimeSeries data type and other related data types to configure the data.
- TimeSeries SQL routines to run queries on time series data.
- TimeSeries API routines and Java classes to use in your applications to manipulate and analyze time series data.

You can use IBM Data Studio or IBM Optim Developer Studio along with the IBM Informix TimeSeries Plug-in for Data Studio to load data from a file into an Informix TimeSeries table.

You can use the IBM OpenAdmin Tool (OAT) for Informix along with the Informix TimeSeries Plug-in for OAT to administer database objects that are related to a time series.

The following illustration shows how the Informix TimeSeries solution and the related products interact.

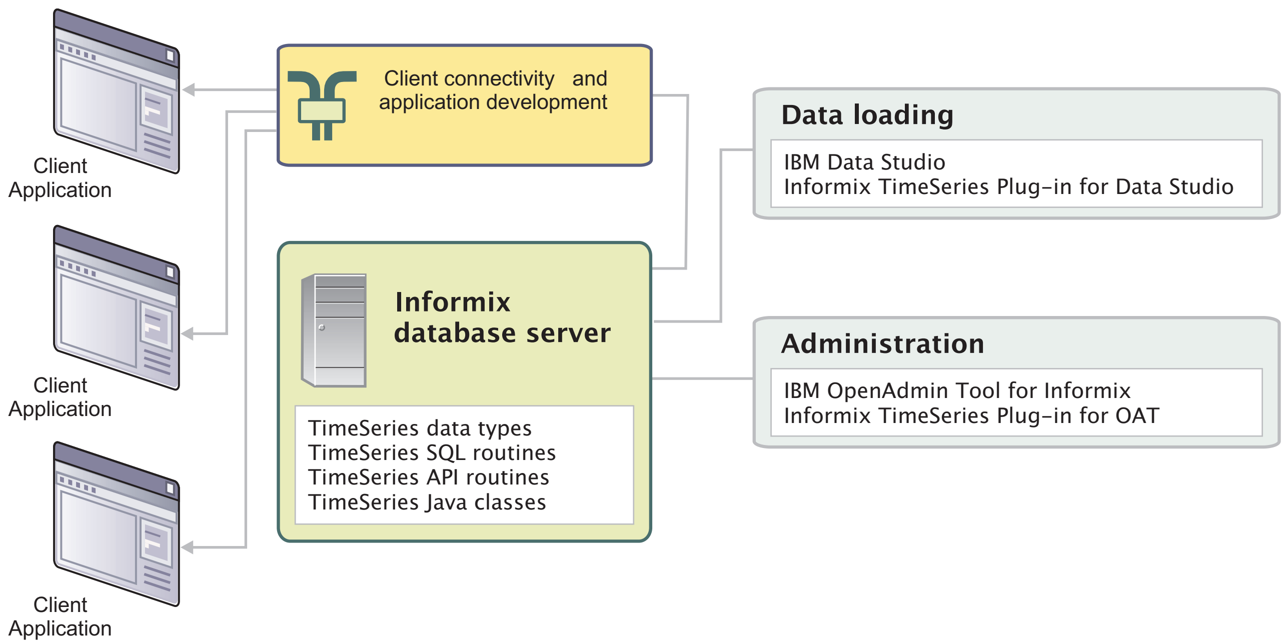


Figure 1-1. Informix TimeSeries architecture

Time series concepts

A time series as implemented by the Informix TimeSeries solution contains information about how the data is stored in the table column and additional information about valid data intervals and where the data is stored on disk.

You should understand the following concepts when you create a time series:

TimeSeries data type
The data type that defines the structure for the time series data.

Element
A set of time series data for one time stamp. For example, a value of 1.01 for the time stamp 2011-1-1 00:45:00.00000 is an element for customer 1001.

Timepoint
The time period for a single element: for example, 15 minutes. In some industries, a timepoint is referred to as an interval.

Origin
The element in the time series that has the earliest time stamp.

Calendar
A set of valid timepoints in a time series, as specified by the calendar pattern.

Calendar pattern
The length of the timepoint and which timepoints are valid. For example, if you collect electricity usage information every 15 minutes, the calendar pattern specifies that timepoints have a length of 15 minutes, and because you want to collect information continuously, all timepoints are valid.

Container
A named portion of a dbspace that contains the time series data for a

specific **TimeSeries** data type and regularity. The data is ordered by time stamp. You can control in which containers your time series data is stored.

Regularity
Whether a time series has regularly spaced timepoints or arbitrarily spaced timepoints.

Virtual table
Virtual tables display a view of the time series data in a relational format without duplicating the data. You can use standard SQL statements on virtual tables to select and insert data.

When you use a calendar and calendar pattern to specify when time series elements are valid, you prevent the storage of null elements for times when data cannot be valid. For example, if you want to track stock data, you would define your calendar to accept elements for timepoints only during trading hours. Also, you can easily find which elements are missing for valid timepoints by querying for null elements. If you do have missing elements, in many cases, the missing elements do not take up space on disk.

You can also aggregate information by selecting data and changing the calendar for the results of the query. For example, if you collect electricity usage information every 15 minutes, but you want to know the total usage per customer per day, you can use a daily calendar to aggregate the data.

TimeSeries data type technical overview

The **TimeSeries** data type defines the structure for the time series data within a single column in the database.

The **TimeSeries** data type is a constructor data type that groups together a collection of ROW data type in time stamp order. A ROW data type consists of a group of named columns. The rows in a **TimeSeries** data type, called elements, each represent one or more data values for a specific time stamp. The elements are ordered by time stamp. The time stamp column must be the first column in the **TimeSeries** ROW data type and must be of type DATETIME YEAR TO FRACTION(5). Time stamps must be unique; multiple entries in a single TimeSeries cannot have the same time stamp.

The following illustration shows the structure of a **TimeSeries** data type that is similar to the one used in the **stores_demo** database.

Database table ts_data

location_id	reads
1000111	TimeSeries(meter_data)
1046021	TimeSeries(meter_data)
1090954	TimeSeries(meter_data)

Time series data for ID 1000111

tstamp (DATETIME YEAR TO FRACTION(5)	value (DECIMAL)
2010-11-10 00:00:00.00000	0.092
2010-11-10 00:15:00.00000	0.082
2010-11-10 00:30:00.00000	0.090
...	...

Time series data for ID 1046021

tstamp (DATETIME YEAR TO FRACTION(5)	value (DECIMAL)
2010-11-10 00:00:00.00000	0.041
2010-11-10 00:15:00.00000	0.041
2010-11-10 00:30:00.00000	0.040
...	...

Time series data for ID 1090954

tstamp (DATETIME YEAR TO FRACTION(5)	value (DECIMAL)
2010-11-10 00:00:00.00000	0.026
2010-11-10 00:15:00.00000	0.035
2010-11-10 00:30:00.00000	0.062
...	...

Figure 1-2. TimeSeries data type architecture

The figure shows the **ts_data** table, which has two columns: the **location_id** column that identifies the source of the time series data, and the **reads** column that contains the time series data. The **reads** column has a data type of **TimeSeries(meter_data)**. The **TimeSeries(meter_data)** data type has two columns: **tstamp** and **value**. The **tstamp** column, as the first column in a **TimeSeries** data type, has a data type of DATETIME YEAR TO FRACTION(5). The **value** column has a data type of DECIMAL. For each source of data, the **reads** column contains multiple rows of time series data, which are ordered by time stamp. All time series data for a particular source is located in the same row of the table.

Related concepts:

"TimeSeries data type" on page 2-5

Related tasks:

"Creating a TimeSeries subtype" on page 3-6

Regular time series

A regular time series stores data for regularly spaced timepoints. A regular time series is appropriate for applications that record entries at predictable timepoints, such as electricity power usage data that is recorded by smart meters every 15 minutes.

Regular time series are stored very efficiently because, instead of storing the full time stamp of an element, regular time series store the *offset* of the element. The offset of an element is the relative position of the element to the origin of the time series. The time stamp for an element is computed from its offset. For example, suppose you have a calendar that has an interval duration of a day. The first element, or origin, is 2011-01-02. The offset for the origin is 0. The offset for the sixth element is 5. The time stamp for the sixth element is the origin plus 5 days: 2011-01-07. The following table shows the relationship between elements and offset.

Table 1-4. Offsets for a daily time series

Day of the month	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Offset		0	1	2	3	4	5

You can use TimeSeries SQL routines to convert between a time stamp and an offset. Some TimeSeries SQL routines require offset values as arguments. For example, you can return the 100th element in a time series with the **GetNthElem** function.

In a regular time series, each interval between elements is the same length. Regular elements persist only for the length of an interval as defined by the calendar associated with the time series. If a value for a timepoint is missing, that element is null. You can update null elements.

Related reference:

"Create a time series" on page 3-11

Irregular time series

An irregular time series stores data for a sequence of arbitrary timepoints. Irregular time series are appropriate when the data arrives unpredictably, such as when the application records every stock trade or when electricity meters record random events such as low battery warnings or low voltage indicators.

Irregular time series store the time stamps for each element instead of storing offsets because the interval between each element can be a different length. Irregular elements persist until the next element by default and cannot be null. For example, if you query for the value of a stock price at noon but the last recorded trade was at 11:59 AM, the query returns the value of the price at 11:59 AM, because that value is the nearest value equal to or earlier than noon. However, you can also create a query to return null if the specified time stamp does not exactly match the time stamp of an element. For example, if you query for the price that a stock traded for at noon, but the stock did not have a trade at noon, the query returns a null value.

Related reference:

"Create a time series" on page 3-11

Calendar

Every time series is associated with a calendar. A calendar defines a set of valid times for elements in a time series. A calendar determines when and how often entries are accepted.

Each calendar has a calendar pattern of timepoints that are either valid or invalid, with the beginning of the calendar pattern specified by the calendar pattern start date. Data is recorded during valid intervals but not during invalid intervals. The calendar pattern also indicates the time unit in which the interval is measured: for example, second, minute, hour, day, or month. The interval size specified in the calendar pattern is not necessarily the same as the size of the timepoint. For example, you can create a calendar pattern that specifies an interval length of minute and specifies that the pattern has one minute valid and 14 minutes invalid. The resulting timepoint is 15 minutes long.

Suppose you want to collect data once a day Monday through Friday. The following table illustrates when data collection is valid, or on, and invalid, or off. The calendar pattern has an interval of a day, has a calendar start date on a Sunday, and specifies one day off, five days on, and one day off.

Table 1-5. When data collection is on or off

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Off	On	On	On	On	On	Off

You can use a predefined calendar or define your own calendar. The seven predefined calendars each have a different interval duration that ranges from one minute to one month. All the predefined calendars start at the beginning of 2011, but you can alter the start date. You create a calendar by inserting a row into the **CalendarTable** table in the format of a **Calendar** data type. You can include the calendar pattern in the calendar definition, or create a separate calendar pattern by inserting a row into the **CalendarPatterns** table in the format of a **CalendarPattern** data type.

You can use calendar and calendar pattern routines to manipulate calendars and calendar patterns. For example, you can create the intersection of calendars or calendar patterns.

Related reference:

Chapter 2, “Data types and system tables,” on page 2-1

Chapter 5, “Calendar pattern routines,” on page 5-1

Chapter 6, “Calendar routines,” on page 6-1

Time series data storage

Time series data is stored in a *container* unless the data remains small enough to fit in a single row of a table. Time series containers are created automatically when they are needed.

A container exists in a dbspace, which is a logical grouping of physical storage (chunks). When a time series is stored in a container, the data is stored contiguously and is retrieved with a minimum number of disk reads.

The following illustration shows the architecture of containers in the database. A database usually contains multiple dbspaces. A dbspace can contain multiple containers along with tables and free space. A container can contain data for one or more sources, for example, customers. The time series data for a particular source is stored on pages in time stamp order.

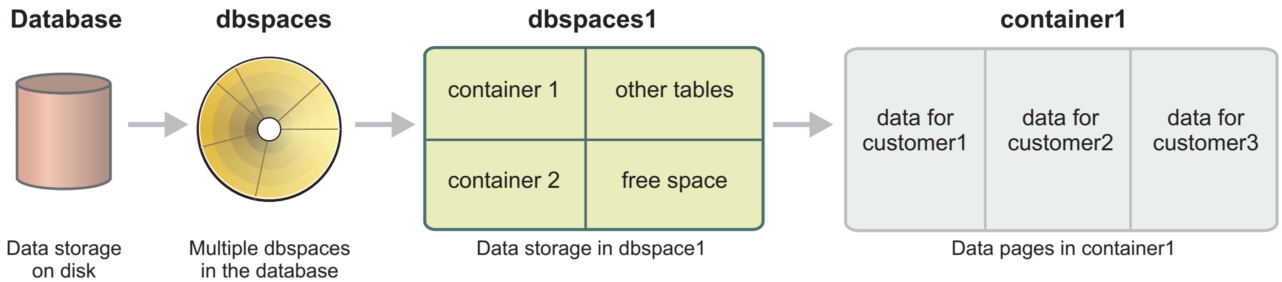


Figure 1-3. Architecture of a container in a database

When you insert data into a time series and you do not specify a container name, the database server checks for one or more containers that are appropriately configured for the time series. If any matching containers exist, the container with the most free space is assigned to the time series. If no matching containers exist, the database server creates a matching container in each of the dbspaces in which the table is stored. For example, if a table is not fragmented and is therefore stored in a single dbspace, one container is created. If a table is fragmented into three dbspaces, three containers are created. All containers that are created automatically by the database server belong to the default container pool, called **autopool**. A container pool is a group of containers.

You can store time series data in a different dbspace than the table is stored in by creating containers and then referencing them when you insert data into the time series. You can specify a container pool in which to store time series data by using the **TSContainerPoolRoundRobin** function or create your own container pool policy function.

Related concepts:

"Planning for data storage" on page 1-10

Related tasks:

"Managing containers" on page 3-7

"Monitoring time series containers" on page 3-8

Getting started with the Informix TimeSeries solution

Before you can create a time series, decide on the properties of the time series and where to store the time series data. After you create a time series, you load the data and query the data.

Planning for creating a time series

When you create a time series, you define a set of properties.

The following table lists the properties of a time series.

Table 1-6. Properties of a time series

Time series property	Description	How to define
Timepoint size	How long a timepoint lasts.	Define a calendar pattern.
When timepoints are valid	The times when elements can be accepted.	Define a calendar pattern.
Data in the time series	The time stamp and the other data that is collected for each time stamp.	Create a **TimeSeries** data type.

Table 1-6. Properties of a time series (continued)

Time series property	Description	How to define
Time series table	The table that contains the **TimeSeries** data type column.	Create a table with a **TimeSeries** column.
Origin	The earliest timestamp of any element	Create a time series.
Regularity	Whether the timepoints are evenly spaced or arbitrarily spaced.	Create a regular or an irregular time series.
Metadata	Optional information included with the time series that can be retrieved by routines.	Create a time series with metadata.

Related tasks:

"Defining a calendar" on page 3-5

Related reference:

"Create a time series column" on page 3-6

"Create a time series" on page 3-11

Planning for data storage

Time series data is stored in containers within dbspaces. You can use the default containers that are created in the same dbspace as the table into which you are loading data or you can create containers in separate dbspaces. You can estimate how much storage space you need.

If you are loading high volumes of data, you can improve the performance of loading the data if you use multiple dbspaces. Similarly, if you have multiple **TimeSeries** columns in the same table, consider creating additional containers that store data in different dbspaces.

Estimate the amount of storage space you need by using the following formula:

space = [*primary_key* + *index_entry* + (*time_series_columns* x *elements*)] x (*table_rows*) + *B-tree_size*

B-tree_size
: The size of the B-tree index, not including the index entries. Typically, the B-tree index is approximately 2% of the size of the data for a regular time series and is approximately 4% of the size of the data for an irregular time series.

elements
: The number of elements of time series data in each row. For example, the **ts_data** table in **stores_demo** database has 8640 elements for each of the 28 rows.

index_entry
: The size of an index entry, which is approximately the size of the primary key columns plus 4 bytes.

primary_key
: The size of the data types of the primary key columns and other non-time series columns in the time series table.

table_rows
: The number of rows in the time series table.

time_series_columns
: The size of the data types of the columns in the **TimeSeries** data type. For regular time series, do not include the size of the time stamp column. For irregular time series, include the size of the time stamp column. The CHAR data type requires an additional 4 bytes when it is included in a **TimeSeries** data type.

The equation is a guideline. The amount of required space can be affected by other factors, such as the small amount of overhead for the slot table and the null bitmap for each element.

The equation might underestimate the amount of required space if the row size of your time series data is very small. The maximum number of elements allowed on a data page is 254. If the row size of your time series data is very small, the page might contain the maximum number of elements but have additional space, especially if you are not using a 2 K page size.

Related concepts:

"Time series data storage" on page 1-8

Related tasks:

"Managing containers" on page 3-7

Planning for loading time series data

When you plan to load time series data, you must choose the loading method and where to store the data on disk.

The following table summarizes the methods of loading data that you can use, depending on how much data you need to load and the format of the data.

Table 1-7. Data loading methods

Data to load	Methods
Bulk data from a file that is created by your data collection application	Use IBM Data Studio and the IBM Informix TimeSeries Plug-in for Data Studio to create a load job for a delimited file. Create a virtual table and load data that is in standard relational format. Use the **BulkLoad** function. The file must be specifically formatted according to the **BulkLoad** function requirements.
Alter or add one or more elements to edit incorrect data or insert missing values	Use the **InsElem** function to insert an element or the **PutElem** function to update an element. Use the **InsSet** function to insert multiple elements or the **PutSet** function to update multiple elements. Create a virtual table and use a standard SQL INSERT statement. You can add or update elements.

Related concepts:

Chapter 4, “Virtual tables for time series data,” on page 4-1

Related tasks:

“Loading data with the IBM Informix TimeSeries Plug-in for Data Studio” on page 3-16

“Loading data from a file into a virtual table” on page 3-16

Related reference:

“Load data with BulkLoad” on page 3-17

“Load small amounts of data with functions” on page 3-18

Planning for accessing time series data

You use SQL functions, Java classes and methods, and C API routines to access and manipulate time series data.

Call routines from within SQL statements or from within Java or C applications on either the client or the server computer.

Use TimeSeries SQL routines to perform the following types of operations to access or manipulate time series data:

- Manipulate individual elements or sets of elements
- Perform statistical and arithmetic calculations
- Aggregate data
- Convert between time stamps and offsets
- Extract values for a time interval
- Find or delete null elements
- Remove older data by deleting a range of elements

The Java classes and methods and API routines perform many of the same tasks that the SQL routines do for time series data. You can use Java classes and methods in applications written in Java. You can use the API routines in applications written in C.

Create virtual tables to view and query time series data by using standard SQL statements. You can display the results of TimeSeries SQL functions on time series data in virtual tables.

You can output time series data in XML format to display in applications.

Related concepts:

Chapter 4, “Virtual tables for time series data,” on page 4-1

Related reference:

Chapter 7, “Time series SQL routines,” on page 7-1

Chapter 8, “Time series Java class library,” on page 8-1

Chapter 9, “Time series API routines,” on page 9-1

“TSToXML function” on page 7-108

Hardware and software requirements

Before you create a time series, ensure that you have the required hardware and software, a supported operating system, and that you understand the restrictions for SQL statements and data replication.

The Informix TimeSeries solution might not be supported on all platforms supported by Informix database servers. See the system requirements for the Informix TimeSeries solution at https://www.ibm.com/support/docview.wss?rs=630&uid=swg27020937.

Installing the IBM Informix TimeSeries Plug-in for Data Studio

The IBM Informix TimeSeries Plug-in for Data Studio is included with the database server installation. Install the TimeSeries plug-in by specifying its location from within IBM Data Studio.

IBM Data Studio version or IBM Optim Developer Studio, version 2.2.1 or later, must be installed and running.

To install the TimeSeries plug-in:

1. Move the plug-in file, `ts_datastudio.zip`, from the `$INFORMIXDIR/extend/TimeSeries.`*`version`*`/plugin` directory to the computer where you are running Data Studio.
2. From Data Studio, choose **Help** > **Software Updates**.
3. From the **Available Software** tab, click **Add Site** and then click **Archive** to select the plug-in file.
4. Select the plug-in directory from the **Available Software** list and click **Install**.
5. After the installation is complete, restart Data Studio.
6. To verify that the plug-in is installed, select **Help** > **About IBM Data Studio** and click **Plugin Details**. Look for Informix TimeSeries Loader in the Plug-in Name column.

Related tasks:

"Loading data with the IBM Informix TimeSeries Plug-in for Data Studio" on page 3-16

Database requirements for time series data

The database in which you implement the Informix TimeSeries solution must conform to requirements.

The database containing the time series data must meet the following requirements:

- The database must be logged.
- The database must not be defined as an ANSI database.

SQL restrictions for time series data

Some SQL statements cannot operate on time series data.

You cannot use the following SQL statements or keywords on **TimeSeries** columns:

- Boolean operators (<, <=, <>, >=, or >)
- SELECT UNIQUE statement
- GROUP BY or ORDER BY clauses
- FRAGMENT BY clause
- PRIMARY KEY clause

You cannot use the MERGE statement on a table with time series data.

You cannot use the ALTER TYPE statement on the **TimeSeries** data type.

Replication of time series data

You have limited options for replicating time series data.

You can replicate time series data only between a High-Availability Data Replication (HDR) primary server and a read-only secondary server. Because some time series calendar and container information is kept in memory, you must stop replication before you can drop and then recreate your calendar or container definitions with the same names but different definitions.

You cannot replicate time series data with the following types of servers and utilities:

- HDR secondary servers that allow updates
- Remote stand-alone secondary servers
- Shared disk secondary servers
- Enterprise Replication
- Change Data Capture API

Time series global language support

Time series data has limited support for non-default locales.

Datetime data

The DATETIME data type used in the **TimeSeries** subtype must be in the default U.S. format:

`"yyyy-mo-dd hh:mm:ss:fffff"`

yyyy Year, expressed in digits

mo Month of year, expressed in digits

dd Day of month, expressed in digits

hh Hour of day, expressed in digits

mm Minute of hour, expressed in digits

ss Seconds of minute, in digits

fffff Fraction of a second, in digits

Character data

Character I/O is not GLS-compliant. You can convert time series data only to character strings that are in the default U.S. locale. You can use the **BulkLoad** function only on character data that is in the default U.S. locale.

However, the following character strings can use any locale and can contain multibyte characters:

- Character fields in a **TimeSeries** data type
- Column names
- Table names
- Calendar names
- Calendar pattern names
- Container names

Numeric data

Floating point data must use the default U.S. format:
- The ASCII period (.) is the decimal separator.
- The ASCII plus (+) and minus (-) signs must be used.

Decimal and money data types are GLS-compliant except that the ASCII plus (+) and minus (-) signs must be used.

Sample smart meter data

If you want to practice querying time series data before you define and load your time series, you can use the sample data in the **stores_demo** database.

The following tables in the **stores_demo** database contain time series data based on electricity usage data collected by smart meters:

Customer_ts_data
: Contains customer numbers and location references.

ts_data_location
: Contains spatial location information.

ts_data
: Contains location references and smart meter time series data.

Related concepts:

Command-line syntax (DB-Access Guide)

Related reference:

The stores_demo Database Map (SQL Reference)

Setting up stock data examples

Set up the stock data examples. Use the sample queries and sample programs to practice handling time series data.

To install the sample database schema and to compile the sample C programs:

1. Set the following environment variables:
   - MACHINE=*machine*
   - PROD_VERSION=*version*
   - USERFUNCDIR=$INFORMIXDIR/extend/TimeSeries.*version*/examples

   The *version* is the internal TimeSeries version number, for example 5.00.UC1. Check the installation directory for the correct version number. The *machine* is the name of the operating system, as listed in the `$INFORMIXDIR/incl/dbdk/makeinc` file, for example, `linux`.
2. Run the **examples_setup.sql** command from the `$INFORMIXDIR/extend/TimeSeries.`*`version`*`/examples` directory: make -f Makefile MY_DATABASE=*dbname* The *dbname* is the name of a database.

Sample queries and programs are located in the same `examples` directory. Precede queries with the BEGIN WORK statement and follow them with the ROLLBACK WORK statement.

Chapter 2. Data types and system tables

Specialized data types and system tables handle time series data.

The data types for time series data are:
- **CalendarPattern**
- **Calendar**
- **TimeSeries**

The system tables for time series data are:
- **CalendarPatterns**
- **CalendarTable**
- **TSInstanceTable**
- **TSContainerTable**

These system tables are in the **sysmaster** database.

When a calendar is inserted into the **CalendarTable** table, it draws information from the **CalendarPatterns** table. The database server refers only to **CalendarTable** for calendar and calendar pattern information; changes to the **CalendarPatterns** table have no effect unless **CalendarTable** is updated or recreated.

TSInstanceTable contains information about all time series.

Related concepts:

"Calendar" on page 1-7

CalendarPattern data type

The **CalendarPattern** data type defines the interval duration and the pattern of valid and invalid intervals in a calendar pattern.

The **CalendarPattern** data type is an opaque data type that has the following format:

```
       ┌─,──────────────────────┐
       ▼                        │
►►─{───num_intervals──┬─on──┬───┴─}─,─interval──────────────────────►◄
                      └─off─┘
```

Table 2-1. CalendarPattern data type parameter values

Value	Description
interval	One of the following interval names: • Second • Minute • Hour • Day • Week • Month • Year
num_intervals	The number of interval units that are either valid intervals for time series data, if followed by `on`, or invalid intervals for time series data, if followed by `off`. The maximum number of interval units, either on or off, in a calendar pattern is 2035. Internal calculations take longer to perform if you use a long calendar pattern.

Usage

The information inside the curly brackets is the pattern specification. The pattern specification has one or more elements consisting of *n*, the number of interval units, and either `on` or `off`, to signify valid or invalid intervals. Elements are separated by commas.

The calendar pattern length is how many intervals before the calendar pattern starts over; after all timepoints in the pattern specification have been exhausted, the pattern is repeated. For this reason, a weekly calendar pattern with daily intervals must contain exactly seven intervals, a daily calendar pattern with hourly intervals must contain exactly 24 intervals, and so on. When the calendar pattern begins is specified by the calendar pattern start date.

For example, a calendar could be built around a normal five-day work week, with the time unit in days, and Saturday and Sunday as days off. Assuming that the calendar pattern start date is for a Sunday, the calendar pattern would be:

```
{ 1 off, 5 on, 1 off }, day
```

In the next example, the calendar is built around the same five-day work week, with the time unit in hours:

```
{ 32 off, 9 on, 15 off, 9 on, 15 off, 9 on, 15 off,
9 on, 15 off, 9 on, 31 off }, hour
```

Both examples have a calendar pattern length of seven days, or one week.

Note: Make sure that your calendar pattern length is correct or your time series data might not match your requirements. For example, the following pattern looks like it should repeat every week, but the pattern repeats every six days because the intervals add up to only six days:

```
{ 1 off, 4 on, 1 off }, day
```

You can manage exceptions to your calendar pattern by hiding elements for which there is no data by using the **HideElem** function.

The calendar pattern is stored in the **CalendarPatterns** table and can be used or reused in several calendars.

Calendar patterns can be combined using functions that form the Boolean AND, OR, and NOT of the calendar patterns. The resulting calendar patterns can be stored in a calendar pattern table or used as arguments to other functions.

You can use the calendar pattern interval with the **WithinR** and **WithinC** functions to search for data around a specified timepoint. The **WithinR** function performs a *relative* search. Relative searches search forward or backward from the starting timepoint, traveling the given number of intervals into the future or past. The **WithinC** function performs a *calibrated* search. A calibrated search proceeds both forward and backward to the interval boundaries surrounding the given starting timepoint.

Example

The following statement creates a pattern named **hour** that has a timepoint every hour:

```
INSERT INTO CalendarPatterns
   VALUES('hour', '{1 on} hour');
```

The following statement creates a pattern named **fifteen_min** that has a 15 minute timepoint:

```
INSERT INTO CalendarPatterns
   VALUES('fifteen_min', '{1 on, 14 off} minute');
```

Related concepts:

"Calendar data type"

"CalendarPatterns table" on page 2-7

Related tasks:

"Defining a calendar" on page 3-5

Related reference:

"GetInterval function" on page 7-43

"WithinC and WithinR functions" on page 7-116

"HideElem function" on page 7-55

Chapter 5, "Calendar pattern routines," on page 5-1

Calendar data type

The **Calendar** data type controls the times at which time series data can be stored.

The **Calendar** data type is an opaque data type that is composed of:

- A starting time stamp
- A calendar pattern
- A calendar pattern starting time stamp

For regular time series, calendars are also used to convert the time periods of interest to offsets of values in the vector, and vice versa.

The input format for the **Calendar** data type is a quoted text string.

```
>>—'——startdate—(—start_date—)——————————————————————————,————>
     |                        └pattstart—(—pattern_date—)┘ |
     └pattstart—(—pattern_date—)——————————————————————————┘

>——pattern—(—calendar_pattern—)——'——————————————————————————><
  └pattname—(—pattern_name—)————┘
```

Table 2-2. Calendar data type parameter values

Value	Data type	Description
start_date	DATETIME YEAR TO FRACTION(5)	Calendar start date. If you do not specify a start date, the calendar pattern start date is used. The calendar start date does not affect the origin of the time series. The origin of the time series specifies the earliest date for elements in the time series. The origin can be prior to the calendar start date.
pattern_date	DATETIME YEAR TO FRACTION(5)	Calendar pattern start date. If both the calendar start date and the pattern start date are included, the pattern start date must be the same as or later than the calendar start data by a number of intervals that is less than or equal to the number of interval lengths in the pattern length. If you do not specify a calendar pattern start date, the calendar start date is used.
calendar_pattern	**CalendarPattern**	Calendar pattern to use.
pattern_name	VARCHAR	Name of calendar pattern to use from **CalendarPatterns** table.

Usage

To create a calendar, insert the keywords and their values into the **CalendarTable** table.

Calendars can be combined using functions that form the Boolean AND, OR, and NOT of the calendars. The resulting calendars can be stored in the **CalendarTable** table or used as arguments to other functions.

You can define both a calendar pattern starting time and a calendar starting time if the calendar and calendar pattern starting times do not coincide. The calendar start date and the pattern start date can be one or more intervals apart, depending on the calendar pattern length. For example, if the calendar pattern is {1 one 14 off}, the pattern length is 15. The calendar start date and the pattern start date can be from 0 to 15 intervals apart.

Occasionally, if you have a regular time series, you will have elements for which there is no data. For example, if you have a daily calendar you might not obtain data on holidays. These exceptions to your calendar are marked as null elements. However, you can hide exceptions so that they are not included in calculations or analysis by using the **HideElem** function.

Examples

The following example inserts a calendar called **yearcal01** into the **CalendarTable** table:

```
insert into CalendarTable(c_name, c_calendar)
        values ('yearcal01 ',
        'pattstart(2001-01-07 00:00:00.00000),
         pattname(workweek_day)');
```

This calendar and its pattern starts on January 7, 2001 and it uses a pattern named **workweek_day**.

The following example creates an hourly calendar with the specified pattern:

```
insert into CalendarTable(c_name, c_calendar)
   values('my_cal',
      'startdate(2011-01-01 00:00:00.00000),
       pattstart(2011-01-02 00:00:00.00000),
       pattern({24 off, 120 on, 24 off}, hour)');
```

The calendar start date is 24 hours prior to the pattern start date. The pattern length is 168 hours, or one week.

Related concepts:

"CalendarTable table" on page 2-7

"CalendarPattern data type" on page 2-1

Related tasks:

"Defining a calendar" on page 3-5

Related reference:

Chapter 6, "Calendar routines," on page 6-1

"HideElem function" on page 7-55

TimeSeries data type

The **TimeSeries** data type is constructed from a row data type and is a collection of row subtypes.

To create a **TimeSeries** column, first you create the **TimeSeries** subtype, using the CREATE ROW TYPE statement.

```
>>--CREATE ROW TYPE--subtype_name--(---------------------------------->

>--timestamp_field--DATETIME YEAR TO FRACTION(5)--,------------------>

     .-,--------------------------------.
     V                                  |
>----field_name--data_type--+----------+-+--)--;----------------------><
                            '-NOT NULL-'
```

Table 2-3. TimeSeries data type parameter values

Value	Description
field_name	The name of the field in the row data type. Must be unique for the row data type. The number of fields in a subtype is not restricted. Must follow the Identifier syntax. For more information, see Identifier (SQL Syntax).

Table 2-3. TimeSeries data type parameter values (continued)

Value	Description
data_type	Can be any data type except the following data types: • SERIAL, SERIAL8, or BIGSERIAL data types • Types that have **Assign** or **Destroy** functions assigned to them, including large object types and some user-defined types
subtype_name	The name of the **TimeSeries** subtype. Can be a maximum of 128 bytes. Must follow the Identifier syntax. For more information, see Identifier (SQL Syntax).
timestamp_field	The name of the field that contains the time stamp. Must be unique for the row data type. Must follow the Identifier syntax. For more information, see Identifier (SQL Syntax).

After you create the **TimeSeries** subtype, you create the table containing the **TimeSeries** column using the CREATE TABLE statement. You can also use the CREATE DISTINCT TYPE statement to define a new data type of type **TimeSeries**.

A **TimeSeries** column can contain either regular or irregular time series; you specify regular or irregular when you create the time series.

The maximum allowable size for a single time series element is 32704 bytes.

You cannot put an index on a column of type **TimeSeries**.

After loading data into a **TimeSeries** column, run the following commands:

```
update statistics high for table tsinstancetable;

update statistics high for table tsinstancetable (id);
```

This improves performance for any subsequent **load**, **insert**, and **delete** operations.

Related concepts:

"TimeSeries data type technical overview" on page 1-5

Related tasks:

"Creating a TimeSeries subtype" on page 3-6

Related reference:

"Create the database table" on page 3-6

"Create a time series" on page 3-11

Time series return types

When a routine returns a time series, calendar information is preserved and, if possible, threshold and container information is preserved.

Some functions that return a **TimeSeries** subtype require that the return value be cast to a particular time series type. For functions like **Clip**, **WithinC**, and **WithinR**, the return type is always the same as the type of the argument time series, and no cast is required.

However, for other functions, such as **AggregateBy**, **Apply**, and **Union**, the type of the resulting time series is not necessarily the same as a time series argument. These functions require that their return types be cast to particular time series types.

If a time series returned by one of these functions cannot use the container of the original time series and a container name is not specified, the resulting time series is stored in a container associated with the matching **TimeSeries** subtype and regularity. If no matching container exists, a new container is created.

CalendarPatterns table

The **CalendarPatterns** table contains information about calendar patterns.

The **CalendarPatterns** table contains two columns: a VARCHAR(255) column (**cp_name**) and a **CalendarPattern** column (**cp_pattern**).

To insert a calendar pattern into the **CalendarPatterns** table, use the INSERT statement.

Related concepts:

"CalendarPattern data type" on page 2-1

Related tasks:

"Defining a calendar" on page 3-5

CalendarTable table

The **CalendarTable** table maintains information about the time series calendars used by the database.

When you create a calendar, you insert a row into the **CalendarTable** table.The **CalendarTable** table contains seven predefined calendars that you can use instead of creating calendars. You can change a calendar by running an UPDATE statement on a row in the **CalendarTable** table.

The following table contains the columns in the **CalendarTable** table.

Table 2-4. The CalendarTable table

Column name	Data type	Description
c_version	INTEGER	Internal. The version of the calendar. Currently, only version 0 is supported.
c_refcount	INTEGER	Internal. Counts the number of in-row time series that reference this calendar. The **c_refcount** column is maintained by the **Assign** and **Destroy** functions on **TimeSeries**. Rules attached to this table allow updates only if **c_refcount** is 0; this restriction ensures that referential integrity is not violated.
c_name	VARCHAR(255)	The name of the calendar.
c_calendar	**Calendar**	The **Calendar** type for the calendar.
c_id	SERIAL	Internal. The serial number of the calendar.

Related concepts:

"Calendar data type" on page 2-3

Related tasks:

"Defining a calendar" on page 3-5

Related reference:

"Predefined calendars" on page 3-5

TSInstanceTable table

The **TSInstanceTable** table contains one row for each large time series, no matter how many times it is referenced.

Time series smaller than the threshold you specify when you create them are stored directly in a column and do not appear in the **TSInstanceTable** table.

Table 2-5. The columns in the TSInstanceTable table

Column name	Data type	Description
id	SERIAL	The serial number of the time series. This is the primary key for the table. You can use the **InstanceId** function to return this number (see "InstanceId function" on page 7-59).
cal_id	INTEGER	The identification of the **CalendarTable** row for the time series.
flags	SMALLINT	Stores various flags for the time series, including one that indicates whether the time series is regular or irregular.
vers	SMALLINT	The version of the time series.
container_name	VARCHAR(128,1)	The name of the container of the time series. This is a reference to the primary key of the **TSContainerTable** table.
ref_count	INTEGER	The number of different references to the same time series instance.

The **TSInstanceTable** table is managed by the database server and users do not modify it directly, nor should they normally be required to view it. Rows in this table are automatically inserted or deleted when large time series are created or destroyed.

Related reference:

"TSContainerCreate procedure" on page 7-83
"TSContainerDestroy procedure" on page 7-85
"TSContainerSetPool procedure" on page 7-88

TSContainerTable table

The **TSContainerTable** table has one row for each container.

Table 2-6. The columns in the TSContainerTable table

Column name	Date type	Description
name	VARCHAR(128,1)	The name of the container of the time series. This is the primary key. Containers that are created automatically are named **autopool***nnnnnnnn*, where *n* is a positive integer eight digits long with leading zeros.
subtype	VARCHAR(128,1)	The name of the time series subtype.
partitionDesc	**tsPartitionDesec_t**	The description of the partition that is the container.
flags	INTEGER	Stores flags to indicate: • If the container is empty and always was empty. • If the time series is regular or irregular.
pool	VARCHAR(128,1) DEFAULT NULL	The name of the container pool to which the container belongs. NULL indicates that the container does not belong to a container pool. The default container pool is named **autopool**.

The **TSContainerTable** table is managed by the database server and users do not modify it directly, nor should they normally be required to view it. Rows in this table are automatically inserted or deleted when containers are created or destroyed.

You can create or destroy containers by using the **TSContainerCreate** and **TSContainerDestroy** procedures, which insert and delete rows in the **TSContainerTable** table. For more information, see "TSContainerCreate procedure" on page 7-83 and "TSContainerDestroy procedure" on page 7-85.

To get a list of containers in the database, run the following query:

```
SELECT NAME FROM TSContainerTable;
```

To get a list of the containers in the default container pool, run the following query:

```
SELECT NAME FROM TSContainerTable
WHERE pool = 'autopool';
```

Related reference:

"TSContainerCreate procedure" on page 7-83
"TSContainerDestroy procedure" on page 7-85

Chapter 3. Create and load a time series

Before you can load time series data into the database, you must configure database objects specific to your time series.

To create and load a time series:

1. Create a calendar or choose a predefined calendar.
2. Create a time series column.
3. Optional. Create additional containers and container pools.
4. Create the time series.
5. Load data into the time series.

Example: Create and load a time series

This example shows how to create a **TimeSeries** data type, create a time series table, create a time series by using the **TSCreate** procedure, and load data into the time series by using the IBM Informix TimeSeries Plug-in for Data Studio.

Prerequisites:

- IBM Data Studio or IBM Optim Developer Studio must be running and the Informix TimeSeries Plug-in for Data Studio must be installed. Data Studio can be installed on a different computer than the database server.
- The **stores_demo** database must exist. You create the **stores_demo** database by running the **dbaccessdemo** command.

In this example, you create a time series that contains electricity meter readings. Readings are taken every 15 minutes. The table and **TimeSeries** data type you create are similar to the examples in the **ts_data** table in the **stores_demo** database. The following table lists the time series properties used in this example.

Table 3-1. Time series properties used in this example

Time series property	Definition
Timepoint size	15 minutes
When timepoints are valid	Every 15 minutes with no invalid times
Data in the time series	The following data: • Timestamp • A decimal value representing electricity usage
Time series table	The following columns: • A meter ID column of type BIGINT • A **TimeSeries** data type column
Origin	All meter IDs have an origin of 2010-11-10 00:00:00.00000
Regularity	Regular
Metadata	No metadata
Amount of storage space	Approximately 1 MB (8640 timepoints for each of the 28 rows)

Table 3-1. Time series properties used in this example (continued)

Time series property	Definition
Where to store the data	In an automatically created container in the same dbspace as the **stores_demo** database, which is in the root dbspace by default
How to load the data	The TimeSeries plug-in
How to access the data	A virtual table

Creating a TimeSeries data type and table

You create a **TimeSeries** data type with columns for the timestamp and the electricity usage value. Then you create a table that has primary key column for the meter ID and a **TimeSeries** column.

To create the **TimeSeries** data type and table:

1. Create a **TimeSeries** subtype named **my_meter_data** in the **stores_demo** database by running the following SQL statement:

   ```
   CREATE ROW TYPE my_meter_data(
       timestamp    DATETIME YEAR TO FRACTION(5),
       data         DECIMAL(4,3)
   );
   ```

   The **timestamp** column contains the time of the meter reading and the **data** column contains the reading value.
2. Create a time series table named **my_ts_data** by running the following SQL statement:

   ```
   CREATE TABLE IF NOT EXISTS my_ts_data (
    meter_id BIGINT NOT NULL PRIMARY KEY,
    raw_reads  TIMESERIES(my_meter_data)
   ) LOCK MODE ROW;
   ```

Related tasks:

"Creating a TimeSeries subtype" on page 3-6

Related reference:

"Create the database table" on page 3-6

Creating regular, empty time series

You need to define the properties of the time series for each meter ID by loading the meter IDs into the time series table and creating a regular, empty time series for each meter ID. You use the meter IDs from the **ts_data** table in the **stores_demo** database to populate the **meter_id** column of your **my_ts_data** table.

To create regular, empty time series:

1. Create an unload file named `my_meter_id.unl` that contains the meter IDs from the **loc_esi_id** column of the **ts_data** table by running the following SQL statement:

   ```
   UNLOAD TO "my_meter_id.unl" SELECT loc_esi_id FROM ts_data;
   ```
2. Create a temporary table named **my_tmp** and load the meter IDs into it by running the following SQL statements:

```
CREATE TEMP TABLE my_tmp (
  id BIGINT NOT NULL PRIMARY KEY);

LOAD FROM "my_meter_id.unl" INSERT INTO my_tmp;
```

You use this table in the next step to create a time series for each meter ID with one SQL statement instead of running a separate SQL statement for each meter ID.

3. Create a regular, empty time series for each meter ID that uses the pre-defined calendar **ts_15min** by running the following SQL statement, which uses the time series input function:

```
INSERT INTO my_ts_data
  SELECT id,
         "origin(2010-11-10 00:00:00.00000),calendar(ts_15min),
          threshold(0),regular,[]"
  FROM my_tmp;
```

Because you did not specify a container name, the time series for each meter ID is stored in a container in the same dbspace in which the table resides. The container is created automatically and is a member of the default container pool.

Related reference:

"Create a time series with its input function" on page 3-13

Creating the data load file

You create a time series data load file by creating a virtual table based on the **ts_data** table and then unloading some of the columns.

To create the data load file:

1. Create a virtual table based on the **raw_reads** time series column of the **ts_data** table by running the following SQL statement:

```
EXECUTE PROCEDURE TSCreateVirtualTab("my_vt", "ts_data", 0, "raw_reads");
```

You use the virtual table to create a data load file.

2. Unload the data from the **tstamp** and **value** columns from the virtual table into a file named `my_meter_data.unl` by running the following SQL statement:

```
UNLOAD TO my_meter_data.unl
  SELECT loc_esi_id, tstamp, value
       FROM my_vt;
```

Related reference:

"TSCreateVirtualTab procedure" on page 4-4

Loading the time series data

You use the TimeSeries plug-in to load the data from the `my_meter_data.unl` file into the **my_ts_data** table. The TimeSeries plug-in has a cheat sheet that you use to guide you through the process of loading the data.

To load time series data:

1. If you are using Data Studio or Optim Developer Studio on a different computer, move the `$INFORMIXDIR\my_meter_data.unl` file to that computer and start Data Studio or Optim Developer Studio.

2. From the main menu, choose **Help** > **Cheat Sheets**, expand the **TimeSeries Data** category, choose **Load time-series data**, and click **OK**.
3. Open the TimeSeries perspective.
4. Create a project area named **my_test**.
5. Create a record format and define the format of the data file. Name the record format definition **my_format** and save it in the **my_test** project directory. Define the following record formats:
   - **meter_id**: choose the Big Integer type and specify the | (pipe) delimiter
   - **timestamp**: choose the Timestamp type and specify the | (pipe) delimiter
   - **data**: choose the Numeric type and specify the | (pipe) delimiter
6. Create the Informix table definition and define the columns of the table. Name the table definition **my_table** and save the definition in the **my_test** project directory. Define the following table columns:
   - **meter_id**: choose the Big Integer type and specify that it is the primary key
   - **raw_reads**: choose the TimeSeries type
7. Define the following subcolumns for the **raw_reads** column and then save the project:
   - **timestamp**: choose the Timestamp type
   - **data**: choose the Numeric type
8. Create a table map named **my_map** and map the data formats of the data file to the columns of the Informix table and then save it in the **my_test** project directory.
9. Create a connection profile to the Informix database server named **my_ifx**.
10. Define and start a load job. Specify the following values:
    - File format file: `my_format.udrf`
    - Table definition file: `my_table.tbl`
    - Mapping file: `my_map.tblmap`
    - Data file: `my_meter_data.unl`
    - Connection profile: **my_ifx**

    When you click **OK**, the load job starts and you see the status.

Related tasks:

"Loading data with the IBM Informix TimeSeries Plug-in for Data Studio" on page 3-16

Accessing time series data through a virtual table

You create a virtual table to view the time series data in relational data format.

To create a virtual table based on the time series table:

Use the **TSCreateVirtualTab** procedure to create a virtual table named **my_vt2** that is based on the **my_ts_data** table by running the following SQL statement:

```
EXECUTE PROCEDURE TSCreateVirtualTab("my_vt2", "my_ts_data",
   "calendar(ts_15min), origin(2010-11-10 00:00:00.00000)");
```

You can query the virtual table using standard SQL statements. For example, the following query returns the first value for each of the 28 meter IDs:

```
SELECT * FROM my_vt2 WHERE timestamp = "2010-11-10 00:00:00.00000";
```

Related reference:

"TSCreateVirtualTab procedure" on page 4-4

Defining a calendar

A time series definition must include a calendar. A calendar includes a calendar pattern, which can be defined separately or within the calendar definition.You can create a calendar or choose a predefined calendar.

To create a calendar:

1. Optional: Create a named calendar pattern by inserting a row into the **CalendarPatterns** table by using the format of the **CalendarPattern** data type. A named calendar pattern is useful if you plan to use the same calendar pattern in multiple calendars.
2. Create a calendar by inserting a row into the **CalendarTable** table by using the format of the **Calendar** data type. Include either the name of an existing calendar pattern or a calendar pattern definition.

To use a predefined calendar, specify one when you create a time series with the **TSCreate** or **TSCreateIrr** function. You can change a predefined calendar to meet your needs by updating the row for the calendar in the **CalendarTable** table.

Related concepts:

"CalendarPattern data type" on page 2-1

"Calendar data type" on page 2-3

"CalendarPatterns table" on page 2-7

"CalendarTable table" on page 2-7

"Planning for creating a time series" on page 1-9

Predefined calendars

You can use predefined calendars when you create a time series.

Predefined calendars are stored in rows in the **CalendarTable** table. You can change a predefined calendar by updating the row for the calendar in the **CalendarTable** table.

If you upgrade from a previous release of the Informix TimeSeries solution or the IBM Informix TimeSeries DataBlade Module and an existing calendar is defined with the same name as one of the predefined calendars, the existing calendar will not be replaced by the predefined calendar.

The following table contains the properties of predefined calendars.

Table 3-2. Predefined calendars

Calendar name	Interval duration	Start date and time
ts_1min	Once a minute	2011-01-01 00:00:00.00000
ts_15min	Once every 15 minutes	2011-01-01 00:00:00.00000
ts_30min	Once every 30 minutes	2011-01-01 00:00:00.00000
ts_1hour	Once an hour	2011-01-01 00:00:00.00000
ts_1day	Once a day	2011-01-01 00:00:00.00000
ts_1week	Once a week	2011-01-02 00:00:00.00000

Table 3-2. Predefined calendars (continued)

Calendar name	Interval duration	Start date and time
ts_1month	Once a month	2011-01-01 00:00:00.00000

Related concepts:

"CalendarTable table" on page 2-7

Create a time series column

To create a time series column:

Related concepts:

"Planning for creating a time series" on page 1-9

Creating a TimeSeries subtype

To create a column of type **TimeSeries**, you must first create a row subtype to represent the data held in each element of the time series.

Subtypes for both regular and irregular time series are created in the same way.

To create the row subtype, use the SQL CREATE ROW TYPE statement and specify that the first field has a DATETIME YEAR TO FRACTION(5) data type. The row type must conform to the syntax of the **TimeSeries** data type.

Examples

The following example creates a **TimeSeries** subtype, called **stock_bar**:

```
create row type stock_bar(
   timestamp      datetime year to fraction(5),
   high           real,
   low            real,
   final          real,
   vol            real
);
```

The following example creates a **TimeSeries** subtype, called **stock_trade**:

```
create row type stock_trade(
   timestamp       datetime year to fraction(5),
   price           double precision,
   vol             double precision,
   trade           int,
   broker          int,
   buyer           int,
   seller          int
);
```

Related concepts:

"TimeSeries data type" on page 2-5

"TimeSeries data type technical overview" on page 1-5

Create the database table

After you have created the subtype, use the CREATE TABLE statement to create a table with a column of that subtype.

You can create the table in a dbspace that uses non-default page size.

The syntax for creating a table with a **TimeSeries** subtype column is:

```
create table table_name(
   col1      any_data_type,
   col2      any_data_type,
   ...
   coln      TimeSeries(subtype_name)
);
```

Examples

The following example creates a table called **daily_stocks** that contains a time series column of type **TimeSeries(stock_bar)**:

```
create table daily_stocks (
   stock_id    int,
   stock_name  lvarchar,
   stock_data  TimeSeries(stock_bar)
);
```

Each row in the **daily_stocks** table can hold a **stock_bar** time series for a particular stock.

The following example creates a table called **activity_stocks** that contains a time series column of type **TimeSeries(stock_trade)**:

```
create table activity_stocks(
   stock_id      int,
   activity_data TimeSeries(stock_trade)
);
```

Each row in the **activity_stocks** table can hold a stock trade time series for a particular stock.

Related concepts:

"TimeSeries data type" on page 2-5

Managing containers

Containers are created automatically when they are needed, in the same dbspaces in which the table is stored. If you want to store your time series data in other dbspaces, you can create additional containers and move them between container pools.

To create a container, run the **TSContainerCreate** procedure.

To delete a container, run the **TSContainerDestroy** procedure.

To add a container into a container pool or move a container from one container pool to another, run the **TSContainerSetPool** procedure and specify the new container pool name. If the container pool does not exist, it is created.

To remove a container from a container pool, run the **TSContainerSetPool** procedure without specifying a container pool name.

To delete a container pool, remove all the containers from it.

To view container information, query the **TSContainerTable** table or view the container in the IBM OpenAdmin Tool (OAT) for Informix.

Examples

Example 1: Creating a new container and adding it to the default container pool

Suppose that you have a **TimeSeries** subtype named **smartmeter_row**, you want to store the time series data in a different dbspace than the table is in, and you do not want to specify the container name when you insert data. The following statements create a container called **ctn_sm1** for the **smartmeter_row** time series and add the container to the default container pool:

```
EXECUTE PROCEDURE TSContainerCreate
                    ('ctn_sm1','tsspace1','smartmeter_row',0,0);
EXECUTE PROCEDURE TSContainerSetPool('ctn_sm1','autopool');
```

When you insert data for the **smartmeter_row** time series without specifying a container name, the database server stores the data in the container named **cnt_sm1** in the dbspace named **tsspace1** instead of creating a new container in the same dbspace as the table.

Example 2: Removing a container from the default container pool

Suppose that a container was automatically created for your time series, but you want to stop automatically inserting data into that container. After you create the new container for the time series using the process in the first example, you can remove the original container from the default container pool. The following statement removes a container named **ctn_sm4** from the default container pool:

```
EXECUTE PROCEDURE TSContainerSetPool('ctn_sm4');
```

The container **ctn_sm4** still exists, but data is inserting into it only if the INSERT statement explicitly names **ctn_sm4** with the **container** argument.

Related concepts:

"Time series data storage" on page 1-8

"Planning for data storage" on page 1-10

Related reference:

"TSContainerCreate procedure" on page 7-83

"TSContainerDestroy procedure" on page 7-85

"TSContainerSetPool procedure" on page 7-88

Monitoring time series containers

You can view information about the size and capacity of time series containers.

If you monitor the containers over a period of time, you can predict how quickly containers fill and how much data fits into each container.

To view specific information about a container, run one of the following functions, specifying the container name:

- The **TSContainerTotalPages** function returns the number of pages allocated to the container.
- The **TSContainerTotalUsed** function returns the number of pages containing time series data.
- The **TSContainerPctUsed** function returns what percent of the container is full.
- The **TSContainerNElems** function returns the number of time series data elements stored in the container.

If you specify NULL instead of a container name, the functions return information about all containers in the database.

To view the number of elements, the number of pages used, and the number of pages allocated, run the **TSContainerUsage** function.

Example

The following statement returns the number of pages containing time series data in the pages column, the number of elements in the slots column, and the number of pages allocated in the total column for the container named **raw_container**:

```
EXECUTE FUNCTION TSContainerUsage("raw_container");

      pages                slots       total

       1999               241881        2119

1 row(s) retrieved.
```

Because 1999 of the total 2119 pages are used, the container is close to being full.

Related concepts:

"Time series data storage" on page 1-8

Related reference:

"TSContainerUsage function" on page 7-91

"TSContainerTotalPages function" on page 7-89

"TSContainerTotalUsed function" on page 7-90

"TSContainerPctUsed function" on page 7-86

"TSContainerNElems function" on page 7-85

Configuring additional container pools

You can create a container pool to manage how time series data is inserted into multiple containers. You can insert data into containers in round-robin order or by using a user-defined method.

If you want to use a container pool policy other than round-robin order, you must write the user-defined container pool policy function before you insert data into the container pool. For more information, see "User-defined container pool policy" on page 3-10.

To create a container pool and store data into containers by using a container pool policy:

1. Create containers by running the **TSContainerCreate** procedure.
2. Add each container to the container pool by using the **TSContainerSetPool** procedure.
3. Insert data into the time series by including the **TSContainerPoolRoundRobin** function with the container pool name or by including your user-defined container pool policy function in the **container** argument.

Example

This example uses a **TimeSeries** subtype named **smartmeter_row** that is in a column named **rawreadings**, which is in a table named **smartmeters**. Suppose you want to store the data for the time series in three containers, in a container pool you created.

The following statements create three containers for the **TimeSeries** subtype **smartmeter_row**:

```
EXECUTE PROCEDURE TSContainerCreate
                   ('ctn_sm0','tsspace0','smartmeter_row',0,0);
EXECUTE PROCEDURE TSContainerCreate
                   ('ctn_sm1','tsspace1','smartmeter_row',0,0);
EXECUTE PROCEDURE TSContainerCreate
                   ('ctn_sm2','tsspace2','smartmeter_row',0,0);
```

The following statements add the containers to a container pool named **readings**:

```
EXECUTE PROCEDURE TSContainerSetPool('ctn_sm0','readings');
EXECUTE PROCEDURE TSContainerSetPool('ctn_sm1','readings');
EXECUTE PROCEDURE TSContainerSetPool('ctn_sm2','readings');
```

The following statement inserts time series data into the column **rawreadings**. The **TSContainerPoolRoundRobin** function that specifies the container pool named **readings** is used instead of a container name in the **container** argument.

```
INSERT INTO smartmeters(meter_id,rawreadings)
     VALUES('met00001','origin(2006-01-01 00:00:00.00000),
            calendar(smartmeter),regular,threshold(0),
            container(TSContainerPoolRoundRobin(readings)),
                [(33070,-13.00,100.00,9.98e+34),
                 (19347,-4.00,100.00,1.007e+35),
                 (17782,-18.00,100.00,9.83e+34)]');
```

During the running of the INSERT statement, the **TSContainerPoolRoundRobin** function runs with the following values:

```
TSContainerPoolRoundRobin('smartmeters','rawreadings',
                          'smartmeter_row',0,'readings')
```

The **TSContainerPoolRoundRobin** function sorts the container names alphabetically, returns the container name **ctn_sm0** to the INSERT statement, and the data is stored in the **ctn_sm0** container. The **TSContainerPoolRoundRobin** function specifies to store the data from the next INSERT statement in the container named **ctn_sm1** and the data from the third INSERT statement in the container named **ctn_sm2**. For the fourth INSERT statement, the **TSContainerPoolRoundRobin** function returns to the beginning of the container list and specifies to store the data in the container named **ctn_sm0**, and so on.

Related reference:

"TSContainerCreate procedure" on page 7-83

"TSContainerPoolRoundRobin function" on page 7-87

"TSContainerSetPool procedure" on page 7-88

"User-defined container pool policy"

User-defined container pool policy

You can create a policy for inserting data into containers within a container pool.

The user-defined container policy you create must have one of the following function signatures.

Syntax

```
PolicyName(
     table_name lvarchar,
     column_name lvarchar,
     subtype lvarchar,
     irregular integer,
     user_data lvarchar
```

```
returns lvarchar;

PolicyName(
      table_name lvarchar,
      column_name lvarchar,
      subtype lvarchar,
      irregular integer,
returns lvarchar;
```

PolicyName
: The name of the user-defined function.

table_name
: The table into which the time series data is being inserted.

column_name
: The name of the time series column into which data is being inserted.

subtype
: The name of the **TimeSeries** subtype.

irregular
: Whether the time series is regular (0) or irregular (1).

user_data
: Optional argument for the name of the container pool.

Description

Write a container pool policy function to select containers in which to insert time series data. For example, the **TSContainerPoolRoundRobin** function inserts data into containers in a round-robin order. You can write a policy function to insert data into the container with the most free space or by using other criteria. You can either specify the name of the container pool with the **user_data** argument or include code for choosing the appropriate container pool in the policy function. The container pool must exist before you can insert data into it, and at least one container within the container pool must be configured for the same **TimeSeries** subtype as used by the data being inserted. Include the policy function in the **container** argument of an INSERT statement. The policy function returns container names to the INSERT statement in the order specified by the function.

Returns

The container name in which to store the time series value.

Related tasks:

"Configuring additional container pools" on page 3-9

Related reference:

"TSContainerPoolRoundRobin function" on page 7-87

Create a time series

There are several ways to create an instance of a time series, depending on whether there is existing data to load and, if so, the format of that data.

There are several ways to create an instance of a time series, depending on whether there is existing data to load and, if so, the format of that data. The following table lists the options for creating and populating a time series.

Task	Function
Create an empty time series	• **TSCreate** (regular time series) • **TSCreateIrr** (irregular time series)
Create an empty time series with metadata	• **TSCreate** with the *metadata* argument (regular time series) • **TSCreateIrr** with the *metadata* argument (irregular time series)
Create and populate a time series	• **TSCreate** with the *set_ts* argument (regular time series) • **TSCreateIrr** with the *set_ts* argument (irregular time series) • The implicit input function • The output of a function
Create and populate a time series with metadata	• **TSCreate** with the *set_ts* and *metadata* arguments (regular time series) • **TSCreateIrr** with the *set_ts* and *metadata* arguments (irregular time series)
Populate an existing time series	• **BulkLoad** • Other functions, such as **PutElem**

Related concepts:

"TimeSeries data type" on page 2-5

"Regular time series" on page 1-6

"Irregular time series" on page 1-7

"Planning for creating a time series" on page 1-9

Creating a time series with the TSCreate or TSCreateIrr function

You can create an empty time series or insert data simultaneously.

The **TSCreate** and **TSCreateIrr** functions create a time series based on the calendar name, the origin time stamp, the threshold, the flags, the number of elements, and the container name.

To create a time series:

Run the **TSCreate** function for regular time series or **TSCreateIrr** function for irregular time series. If you want to insert data into your time series when you create it, include the data in the *set_data* argument.

Examples

Example 1: Create an empty time series

The following example uses the **TSCreate** function to create an empty time series:

```
insert into daily_stocks values(
   901,'IBM', TSCreate('daycal',
      '2011-01-03 00:00:00.00000',20,0,0, NULL));
```

Example 2: Create a time series with data

For example, suppose a table called **activity_load_tab** has a column called **set_data** of type **SET(stock_trade)**. The following statement creates a time series and inserts it into the **activity_stocks** table:

```
insert into activity_stocks
    select 1234,
      TSCreateIrr('daycal',
         '2011-01-03 00:00:00.00000'::datetime year to fraction(5),
         20, 0, NULL,
         set_data)::timeseries(stock_trade)
    from activity_load_tab;
```

Related reference:

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

Creating a time series with metadata

You can create an empty or populated time series that also contains user-defined metadata. A time series column includes a header that holds information about the time series and can also contain user-defined metadata.

User-defined metadata allows the time series to be self-describing. The metadata can be information usually contained in additional columns in the table, such as the name of a stock, or the type of the time series. The advantage of keeping this type of information in the time series is that, when using an API routine, it is easier to retrieve the metadata than to pass additional columns to the routine. Metadata is stored in a distinct type based on the **TimeSeriesMeta** data type. The **TimeSeriesMeta** data type is an opaque data type of variable length, up to a maximum length of 512 bytes. The routines that accept the **TimeSeriesMeta** data type also accept its distinct type. The distinct type requires support functions, such as input, output, send, receive, and so on.

To create a time series with metadata:

1. Create a distinct data type based on the **TimeSeriesMeta** data type with the following SQL statement. Substitute *MyMetaData* with a name you choose.
   create distinct type *MyMetaData* as TimeSeriesMeta
2. Create support functions for your metadata data type. For information on creating support functions, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.
3. Run the **TSCreate** or **TSCreateIrr** function with the *metadata* argument.

After you have created a time series with metadata, you can add, change, remove, and retrieve the metadata. You can also retrieve the name of your metadata type.

Related reference:

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

"UpdMetaData function" on page 7-114

"GetMetaData function" on page 7-47

Create a time series with its input function

You can use the time series input function to create a time series with the INSERT statement.

The syntax for using INSERT to create a time series and insert data is:

```
insert into table_name values(
   'col1_value',
   'col2_value',
   ...,
   'parameter_input_string'
);
```

The *parameter_input_string* value contains the time series information. All data types have an associated input function that is automatically invoked when ASCII data is inserted into the column. In the case of the **TimeSeries** data type, the input has several pieces of data embedded in the text. This information is used to convey the name of the calendar, the time stamp of the origin, the threshold, the container, and the initial time series data. The format for the parameter input string is:

```
paramname(value), paramname(value), ..., [data_element, ...]
```

The values are specific to the parameters, and each has a different format. The following table indicates the value associated with each parameter.

Table 3-3. Parameters for inserting data into a time series

Parameter name	Required	Value
calendar	Yes	Name of the calendar to use. There is no default name.
container	No	Name of the container to use. The default is no container; the time series must fit in the database row or never be assigned to a table. If the time series exceeds the threshold size, you must set a container.
datafile	No	Name of the input file to use. The format is the same as for the **BulkLoad** function. If the data file is present, no "bracketed" data is permitted. Default is `NULL`.
irregular	Yes (for irregular)	No value, just the string `irregular`. This parameter must be included for an irregular time series but cannot be included for a regular time series.
metadata	No	The metadata to be added to the time series. Can be `NULL`. If metadata is supplied, then the metadata type must also be supplied.
metatype	No	The data type of the metadata.
origin	No	Time stamp of the origin of the time series. The default origin is the calendar start date.
regular	No	No value, just the string `regular`. This parameter is optional for a regular time series but cannot be included for an irregular time series.
threshold	No	Number of elements above which data is placed in a container rather than in the row. Default is 20. An in-row time series should not be larger than 1500 bytes.

If a parameter is not present in the input string, its default value is used.

If you did not specify a data file, then you can supply the data to be placed in the time series (the data element), surrounded by square brackets, after the parameters:

```
[(value, value, value, ...)@timestamp, (...), ...]
```

Elements consist of data values, each separated by a comma. The data values in each element correspond to the columns in the **TimeSeries** subtype, not including the initial time stamp column. Each element is surrounded by parentheses and followed by an @ symbol and a time stamp. The time stamp is optional for regular time series but mandatory for irregular time series. Null data values or elements are indicated with the word `NULL`. If no data elements are present, the function creates an empty time series.

Example 1: Create a regular time series

Following example shows an INSERT statement for a regular time series created in the table **daily_stocks**:

```
insert into daily_stocks values (1234, 'informix',
        'regular, calendar(daycal),
[(350, 310, 340, 1999), (362, 320, 350, 2500)]');
```

This INSERT statement creates a regular time series that starts at the date and time of day specified by the calendar called **daycal**. The first two elements in the time series are populated with the bracketed data. Since the threshold parameter is not specified, its default value is used. Therefore, if more than 20 elements are placed in the time series, the database server attempts to move the data to a container, but because there is no container specified, an error is raised.

Example 2: Create an irregular time series

The following example shows an INSERT statement for an irregular time series created in the table **activity_stocks**:

```
insert into activity_stocks values (
   600, 'irregular, container(ctnr_stock), origin(2005-10-06 00:00:00.00000),
calendar(daycal), [(6.25,1000,1,7,2,1)@2005-10-06 12:58:09.12345, (6.50, 2000,
1,8,3,1)@2005-10-06 12:58:09.23456]');
```

The INSERT statement creates an irregular time series that starts on 06 October 2005, at the time of day specified by the calendar called **daycal**. Two rows of data are inserted with the specified time stamps.

Create a time series with the output of a function

Many functions return a time series.

The container for a time series that is created by the output of a function is often implicitly determined. For example, if part of a time series is extracted using the **Clip** function and the result is stored in the database, the container for the original time series is used for the new time series.

If a time series returned by one of these functions cannot use the container of the original time series and a container name is not specified, the resulting time series is stored in a container associated with the matching **TimeSeries** subtype and regularity. If no matching container exists, a new container is created.

Load data into an existing time series

After you create a time series, you can use one of several methods to load data into the time series.

Choose the data loading method according to the amount of data and the format of the data.

Loading data with the IBM Informix TimeSeries Plug-in for Data Studio

Use the IBM Informix TimeSeries Plug-in for Data Studio to load data from a file into a time series.

You must have the following prerequisites before you load data:

- IBM Data Studio or IBM Optim Developer Studio with the Informix TimeSeries Plug-in for Data Studio installed.
- An existing table with a **TimeSeries** column.
- Primary key values in your table and a time series defined for each row. If your primary key has a data type of CHAR(*n*), and each value is not *n* bytes long, you must pad the values to be *n* bytes long or change the data type to VARCHAR(20).
- A file of time-based data that you want to load into the database.
- Connectivity information for the Informix database server that contains the time series table.

The TimeSeries plug-in includes a cheat sheet that provides detailed instructions for loading data.

To create load job for time series data in Data Studio:

Choose **Help** > **Cheat Sheets**, expand the **TimeSeries Data** category, choose **Load time-series data**, and click **OK**. Follow the instructions in the cheat sheet to perform the remaining steps.

You can reuse the file format definition, the table definition, the mapping definition, and the connection profile that you created in subsequent load jobs. If you change your table definition, you must update the corresponding mapping as well.

Related concepts:

"Planning for loading time series data" on page 1-11

Related tasks:

"Installing the IBM Informix TimeSeries Plug-in for Data Studio" on page 1-13

Loading data from a file into a virtual table

Data that you insert into a virtual table is written to the underlying base table. Therefore, you can use the virtual table to load your data that is in a relational format in a file into a **TimeSeries** column. Often it is easier to format your raw data to load a virtual table than to load a **TimeSeries** column directly, especially if you must perform incremental loading.

You can load data from a virtual table that was created by the **TSCreateVirtualTab** procedure. You cannot load data from a virtual table was created by the **TSCreateExpressionVirtualTab** procedure.

To load relational data through a virtual table:

1. Create a virtual table that is based on a time series table.
2. Put your input data in a single file.
3. Format the data according to the standard IBM Informix load file format.

4. Use any of the Informix load utilities: **pload**, **onpload**, **dbload**, or the **load** command in DB-Access, to load the file into the virtual table.

See the *IBM Informix Administrator's Guide* for information about Informix load file formats and load utilities.

Related concepts:

"Planning for loading time series data" on page 1-11

Chapter 4, "Virtual tables for time series data," on page 4-1

Related reference:

"TSCreateVirtualTab procedure" on page 4-4

Load data with BulkLoad

You can load data into an existing time series with the **BulkLoad** function. This function takes an existing time series and a file name as arguments. The file name is for a file on the client that contains row type data to be loaded into the time series.

The syntax for using **BulkLoad** with the UPDATE statement and the SET clause is:

```
update table_name
   set TimeSeries_col=BulkLoad(TimeSeries_col, 'filename')
   where col1='value';
```

The *TimeSeries_col* parameter is the name of the column containing the row type. The *filename* parameter is the name of the data file. The WHERE clause specifies which row in the table to update.

Related concepts:

"Planning for loading time series data" on page 1-11

Data file formats for BulkLoad

Two data formats are supported for the file loaded by **BulkLoad**:

- Using type constructors
- Using tabs

Each line of the client file must have all the data for one element.

The type constructor format follows the row type convention: comma-separated columns surrounded by parentheses and preceded by the ROW type constructor. The first two lines of a typical file look like this:

```
row(2011-01-03 00:00:00.00000, 1.1, 2.2)
row(2011-01-04 00:00:00.00000, 10.1, 20.2)
```

If you include collections in a column within the row data type, use a type constructor (SET, MULTISET, or LIST) and curly braces surrounding the collection values. A row including a set of rows has this format:

```
row(timestamp, set{row(value, value), row(value, value)}, value)
```

The tab format separates the values by tabs. It is only recommended for single-level rows that do not contain collections or row data types. The first two lines of a typical file in this format look like this:

```
2011-01-03 00:00:00.00000   1.1     2.2
2011-01-04 00:00:00.00000   10.1    20.2
```

The spaces between entries represent a tab.

In both formats, NULL indicates a null entry.

The first file format is also produced when you use the **onload** utility. This utility copies the contents of a table into a client file or a client file into a table. When copying a file into a table, the time series is created and then the data is written into the new time series. See the *IBM Informix Performance Guide* for more information about **onload**.

Example: Load data with BulkLoad

The following example uses **BulkLoad** in the SET clause of an UPDATE statement to populate the existing time series in the **daily_stocks** table:

```
insert into daily_stocks values
   (999, 'IBM', TSCreate ('daycal',
      '2011-01-03 00:00:00.00000',20,0,0, NULL));

update daily_stocks
   set stock_data=BulkLoad(stock_data,'sam.dat')
   where stock_name='IBM';
```

Load small amounts of data with functions

You can load individual elements or sets of elements by using TimeSeries functions.

Use any of the following functions to load data into a time series:

PutElem
Updates a time series with a single element.

PutSet Updates a time series with a set of elements.

InsElem
Inserts an element into a time series.

InsSet Inserts every element of a given set into a time series.

These functions add or update an element or set of elements to the time series. They must be used in an SQL UPDATE statement with the SET clause:

```
update table_name
   set TimeSeries_col=FunctionName(TimeSeries_type, data)
   where col1='value';
```

The *TimeSeries_col* argument is the name of the column in which the time series is located. The *FunctionName* argument is the name of the function. The *data* argument is in the row type data element format (see "Create a time series with its input function" on page 3-13). The WHERE clause specifies which row in the table to update.

The following example appends an element to a time series using **PutElem**:

```
update daily_stocks
set stock_data = PutElem(stock_data,
   row(NULL::datetime year to fraction(5),
   2.3, 3.4, 5.6, 67)::stock_bar)
   where stock_name = 'IBM';
```

You can also use more complicated expressions to load a time series. For examples, see "Binary arithmetic functions" on page 7-21.

Related concepts:

"Planning for loading time series data" on page 1-11

Chapter 4. Virtual tables for time series data

A virtual table provides a relational view of your time series data.

Virtual tables are useful for viewing time series data in a simple format. An SQL SELECT statement against a virtual table returns data in ordinary data type format, rather than in the **TimeSeries** data type format. Many of the operations that TimeSeries SQL functions and API routines perform can be done using SQL statements against a virtual table. Some SQL queries are easier to write for the virtual table than for an underlying time series table, especially SQL queries with qualifications on a **TimeSeries** column.

The virtual table is not a real table stored in the database. The data is not duplicated. At any given moment, data visible in the virtual table is the same as the content in the base table. You cannot create an index on a time series virtual table.

The performance of queries on virtual tables versus using TimeSeries functions is similar in most cases. For example, the **Clip** function is faster applied through a virtual table than directly on a time series. However, it is faster to run the **Apply** or the **Transpose** routines on a time series than to run them through a virtual table by using the **TSCreateExpressionVirtualTab** procedure.

Some operations are difficult or impossible in one interface but are easily accomplished in the other. For example, finding the average value of one of the fields in a time series over a period of time is easier with a query against a virtual table than by using TimeSeries functions. The following query against a virtual table finds the average stock price over a year:

```
select avg(vol) from daily_stocks_no_ts
where stock_name = 'IBM'
and timestamp between datetime(2010-1-1) year to day
and datetime(2010-12-31) year to day;
```

However, aggregating from one calendar to another is easier using the **AggregateBy** routine.

Selecting the *n*th element in a regular time series is easy using the **GetNthElem** routine but quite difficult using a virtual table.

You can insert data into a virtual table that is based on a time series table, which automatically updates the underlying base table. You can use SELECT and INSERT statements with time series virtual tables. You cannot use UPDATE or DELETE statements, but you can update a time series element in the base table by inserting a new element for the same time point into the virtual table.

You can create a virtual table based on an expression that is performed on a time series table.

You can create a virtual table based on only one **TimeSeries** column at a time. If the base table has multiple **TimeSeries** columns, you can create a virtual table for each of them.

Related concepts:

"Planning for accessing time series data" on page 1-12

"Planning for loading time series data" on page 1-11

Related tasks:

"Loading data from a file into a virtual table" on page 3-16

The structure of virtual tables

A virtual table that is based on a time series has the same schema as the base table, except for the **TimeSeries** column. The **TimeSeries** column is replaced with the columns of the **TimeSeries** subtype. A virtual table based on an expression on a time series displays the **TimeSeries** subtype that is the result of the expression, instead of the subtype from the base table.

For example, the table **ts_data** contains a **TimeSeries** column called **raw_reads** that contains a row type with **tstamp** and **value** columns. The following table displays part of the **ts_data** table. The actual time stamp values are shown for clarity, although the time stamp values are calculated instead of stored in regular time series.

Table 4-1. Data in a table with a TimeSeries column

loc_esi_id	measure_unit	direction	raw_reads
4727354321000111	KWH	P	(2010-11-10 00:00:00.00000, 0.092), (2010-11-10 00:15:00.00000, 0.084), ...
4727354321046021	KWH	P	(2010-11-10 00:00:00.00000, 0.041), (2010-11-10 00:15:00.00000, 0.041), ...
4727354321090954	KWH	P	(2010-11-10 00:00:00.00000, 0.026), (2010-11-10 00:15:00.00000, 0.035), ...

The virtual table that is based on the **ts_data** table converts the **raw_reads** column elements into individual columns. The following table displays part of the virtual table that is based on the **ts_data** table.

Table 4-2. Data in a virtual table based on a time series

loc_esi_id	measure_unit	direction	tstamp	value
4727354321000111	KWH	P	2010-11-10 00:00:00.00000	0.092
4727354321000111	KWH	P	2010-11-10 00:15:00.00000	0.084
. . .				
4727354321046021	KWH	P	2010-11-10 00:00:00.00000	0.041
4727354321046021	KWH	P	2010-11-10 00:15:00.00000	0.041
. . .				
4727354321090954	KWH	P	2010-11-10 00:00:00.00000	0.026
4727354321090954	KWH	P	2010-11-10 00:15:00.00000	0.035

When you create a virtual table that is based on the results of an expression that is performed on a time series, you specify the **TimeSeries** subtype appropriate for containing the results of the expression. The virtual table is based on the specified **TimeSeries** data type and the other columns from the base table.

The display of data in virtual tables

When you create virtual tables based on time series, you can customize how time series data is shown in the virtual tables and in the results of queries on the virtual tables.

Null elements in a time series are not included in the virtual table. If a base table has a null element at a specific timepoint, the virtual table has no entry for that timepoint. You can specify that null elements appear in the virtual table.

Hidden elements are not included in the virtual table. A hidden element is marked as invisible in the base table. You can specify if hidden elements appear as null values in the virtual table, or if their values are visible in the virtual table.

When you select data from a virtual table by timestamps, the rows whose timestamps are closest to being equal to or earlier than the timestamps specified in the query are returned. If the time series is irregular, the returned rows show the same timestamps as specified in the query, regardless if the actual timestamps are the same. You can specify that when you select data from a virtual table by timestamps, only rows whose timestamps are exactly equal to the timestamps specified in the query are returned.

You control the display of data by setting the *TSVTMode* parameter in the **TSCreateVirtualTab** procedure or the **TSCreateExpressionVirtualTab** procedure.

Related concepts:

"The TSVTMode parameter" on page 4-11

Related reference:

"TSCreateVirtualTab procedure" on page 4-4

"TSCreateExpressionVirtualTab procedure" on page 4-8

The insertion of data through virtual tables

You can insert data into a virtual table that is based on a time series table. You can control whether to allow a new time series, duplicate elements for the same timepoints, which columns in the base table can be updated, and how flexible the INSERT statement can be.

You can add an additional time series element to an existing time series through a virtual table. You can specify to be able to add a time series element into an existing row that does not have any time series data, or to add a new row to the base table.

When you insert an element that has the same timepoint as an existing element, the original element is replaced. You can specify to allow multiple elements with the same timepoint.

If the base table has a primary key, the primary key is used to find the row to update and updates to the base table do not require accurate values for columns that are not part of the primary key. If the base table does not have a primary key, all columns in the table except the **TimeSeries** column are used to identify the row to be updated and updates to the base table require accurate values for every column in the base table other than the **TimeSeries** column. You can only update the values in the **TimeSeries** column. You can specify the rules for the INSERT statement and which columns can be updated:

- You can update only the **TimeSeries** column, but you can specify NULL as the values for non-primary key columns
- You can update the **TimeSeries** column and all other non-primary key columns that do not have null values in the INSERT statement.
- You can update the **TimeSeries** column and all other non-primary key columns. You can set columns that do not have NOT NULL constraints to null values.
- You can update the **TimeSeries** column and all other non-primary key columns that have NOT NULL constraints. You can specify null values for columns that have NOT NULL constraints.

You can control data insertion by setting the *NewTimeSeries* parameter and the *TSVTMode* parameter in the **TSCreateVirtualTab** procedure.

Related concepts:

"The TSVTMode parameter" on page 4-11

Related reference:

"TSCreateVirtualTab procedure"

Creating a time series virtual table

You can create a virtual table based on a time series or based on the results of an expression on a time series.

You can update or insert data through a virtual table that is based on a time series. You cannot update or insert data through a virtual table that is based on an expression on a time series.

To create a virtual table based on a table that contains a **TimeSeries** column, run the **TSCreateVirtualTab** procedure.

To create a virtual table based on the results of an expression that is performed on a time series, run the **TSCreateExpressionVirtualTab** procedure.

Related reference:

"TSCreateVirtualTab procedure"

"TSCreateExpressionVirtualTab procedure" on page 4-8

TSCreateVirtualTab procedure

The **TSCreateVirtualTab** procedure creates a virtual table based on a table containing a **TimeSeries** column.

Syntax

```
TSCreateVirtualTab(VirtualTableName  lvarchar,
                   BaseTableName     lvarchar,
                   NewTimeSeries     lvarchar,
                   TSVTMode          integer default 0,
                   TSColName         lvarchar default NULL);
```

VirtualTableName
: The name of the new virtual table.

BaseTableName
: The name of the base table.

NewTimeSeries **(optional)**
: The definition of the new time series to create.

TSVTMode **(optional)**
Sets the virtual table mode, as described in "The TSVTMode parameter" on page 4-11.

TSColName **(optional)**
For base tables that have more than one **TimeSeries** column, specifies the name of the **TimeSeries** column to be used to create the virtual table. The default value for the *TSColName* parameter is `NULL`, in which case the base table must have only one **TimeSeries** column.

Usage

Use the **TSCreateVirtualTab** procedure to create a virtual table based on a table that contains a time series. Because the column names in the **TimeSeries** row type are used as the column names in the resulting virtual table, you must ensure that these column names do not conflict with the names of other columns in the base table. The total length of a row in the virtual table (non-time-series and **TimeSeries** columns combined) must not exceed 32 KB.

You can configure the time series virtual table to allow updating data in the base table through the virtual table. If you specify any of the optional parameters, you must include them in the order shown in the syntax, but you can use any one of them without using the others. For example, you can specify the *TSColName* parameter without including the *NewTimeSeries* and the *TSVTMode* parameters.

The NewTimeSeries parameter

The *NewTimeSeries* parameter specifies whether the virtual table allows elements to be inserted into a time series that does not yet exist in the base table either because the row does not exist or because the row does not yet have a time series element. To allow inserts if a time series does not yet exist, use the *NewTimeSeries* parameter to specify the time series input string. To prohibit inserts if a time series does not yet exist, omit the *NewTimeSeries* parameter when you create the virtual table.

The following table describes the results of attempting to update the base table for different goals.

Table 4-3. Behavior of updates to the base table

Goal	Result	Need to use the NewTimeSeries parameter?
Add a time series element into an existing row that does not have any time series data. For example, add the first meter reading for a specific meter.	A new time series is inserted in the existing row.	Yes

Table 4-3. Behavior of updates to the base table (continued)

Goal	Result	Need to use the NewTimeSeries parameter?
Add an additional time series element to an existing time series. For example, add a new meter reading for a meter that has previous readings.	If the timepoint is not the same as an existing element, the new element is inserted to the time series. If the timepoint is the same as an existing element, the existing element is updated with the new value. If the *TSVTMode* parameter includes the value 1, multiple elements for the same timepoint can coexist, therefore the new element is inserted and the existing element is also retained.	No
Add a new row. For example, add a row for a new meter ID.	A new row is inserted into the base table.	Yes

If you do not include the *NewTimeSeries* parameter and attempt to insert a time series element into an existing row that does not have any time series elements or into a new row, you receive an error.

Example

The following example creates a virtual table called **daily_stocks_virt** based on the table **daily_stocks**. Because this example specifies a value for the *NewTimeSeries* parameter, the virtual table **daily_stocks_virt** allows inserts if a time series does not exist for an element in the underlying base table. If you perform such an insert, the database server creates a new empty time series that uses the calendar **daycal** and has an origin of January 3, 2011.

```
execute procedure TSCreateVirtualTab('daily_stocks_virt',
        'daily_stocks', 'calendar(daycal),
        origin(2011-01-03 00:00:00.00000)' );
```

Related concepts:

"The display of data in virtual tables" on page 4-3

"The insertion of data through virtual tables" on page 4-3

Related tasks:

"Creating a time series virtual table" on page 4-4

"Loading data from a file into a virtual table" on page 3-16

Example of creating a virtual table

This example shows how to create a virtual table on a table that contains time series data and the difference between querying the base table and the virtual table.

To improve clarity, these examples use values *t1* through *t6* to indicate DATETIME values, rather than showing complete DATETIME strings.

Query the base table

The base table, **daily_stocks**, was created with the following statements:

```
create row type stock_bar(
   timestamp      datetime year to fraction(5),
   high      real,
   low      real,
   final      real,
   vol      real
);

create table daily_stocks (
   stock_id        int,
   stock_name        lvarchar,
   stock_data        TimeSeries(stock_bar)
);
```

The **daily_stocks** base table contains the following data.

stock_id	stock_name	stock_data
900	AA01	(*t1*, 7.25, 6.75, 7, 1000000), (*t2*, 7.5, 6.875, 7.125, 1500000), ...
901	IBM	(*t1*, 97, 94.25, 95, 2000000), (*t2*, 97, 95.5, 96, 3000000), ...
905	FNM	(*t1*, 49.25, 47.75, 48, 2500000), (*t2*, 48.75, 48, 48.25, 3000000), ...

To query on the **stock_data** column, you must use time series functions. For example, the following query uses the **Apply** function to obtain the closing price:

```
select stock_id,
Apply('$final', stock_data)::TimeSeries(one_real)
from daily_stocks;
```

In this query, *one_real* is a row type created to hold the results of the query and is created with this statement:

```
create row type one_real(
   timestamp datetime year to fraction(5),
   result real);
```

To obtain price and volume information within a specific time range, you use a query like this:

```
select stock_id, Clip(stock_data, t1, t2) from daily_stocks;
```

Create the virtual table

The following statement uses the **TSCreateVirtualTab** function to create a virtual table, called **daily_stocks_no_ts**, based on **daily_stocks**:

```
execute procedure
TSCreateVirtualTab('daily_stocks_no_ts', 'daily_stocks');
```

Because the statement does not specify the *NewTimeSeries* parameter, **daily_stocks_no_ts** does not allow inserts of elements that do not have a corresponding time series in **daily_stocks**.

Also, the statement omits the *TSVTMode* parameter, so *TSVTMode* assumes its default value of `0`. Therefore, if you insert data into **daily_stocks_no_ts**, the database server uses **PutElemNoDups** to add an element to the underlying time series in **daily_stocks**.

The virtual table, **daily_stocks_no_ts** looks like this.

Table 4-4. The daily_stocks_no_ts virtual table

stock_id	stock_name	timestamp*	high	low	final	vol
900	AA01	*t1*	7.25	6.75	7	1000000
900	AA01	*t2*	7.5	6.875	7.125	1500000
...	...	...			...	...
901	IBM	*t1*	97	94.25	95	2000000
901	IBM	*t2*	97	95.5	96	3000000
...	...	...			...	...
905	FNM	*t1*	49.25	47.75	48	2500000
905	FNM	*t2*	48.75	48	48.25	3000000
...	...	...			...	

* In this column, *t1* and *t2* are DATETIME values.

Query the virtual table

Certain SQL queries are much easier to write for a virtual table than for a base table. For example, the query to obtain the closing price now looks like this:

```
select stock_id, final from daily_stocks_no_ts;
```

And the query to obtain price and volume within a specific time range looks like this:

```
select * from daily_stocks_no_ts
where timestamp between t1 and t5;
```

Some tasks that are complex for time series functions to accomplish, such as use of the ORDER BY clause, are now simple:

```
select * from daily_stocks_no_ts
where timestamp between t1 and t5
order by volume;
```

Inserting data into the virtual table is also simple. To add a new element to the IBM stock, use the following query:

```
insert into daily_stock_no_ts
values('IBM', t6, 55, 53, 54, 2000000);
```

The element (*t6*, 55, 53, 54, 2000000) is added to **daily_stocks**.

TSCreateExpressionVirtualTab procedure

The **TSCreateExpressionVirtualTab** procedure creates a virtual table based on the results of an expression that was performed on a table containing a **TimeSeries** column. The resulting virtual table is read-only.

Syntax

```
TSCreateExpressionVirtualTab
                  (VirtualTableName  lvarchar,
                   BaseTableName     lvarchar,
                   expression        lvarchar,
                   subtype           lvarchar,
                   TSVTMode          integer default 0,
                   TSColName         lvarchar default NULL);
```

VirtualTableName
The name of the new virtual table.

BaseTableName
The name of the base table.

expression
The expression to be evaluated on time series data.

subtype
The name of the **TimeSeries** subtype for the values that are the results of the expression.

TSVTMode **(optional)**
Sets the virtual table mode, as described in "The TSVTMode parameter" on page 4-11.

TSColName **(optional)**
For base tables that have more than one **TimeSeries** column, specifies the name of the **TimeSeries** column to be used to create the virtual table. The default value for the *TSColName* parameter is `NULL`, in which case the base table must have only one **TimeSeries** column.

Usage

Use the **TSCreateExpressionVirtualTab** procedure to create a virtual table based on a time series that results from an expression that is performed on time series data each time a query, such as a SELECT statement, is performed. You specify the name of the **TimeSeries** subtype in the virtual table with the *subtype* parameter.

The total length of a row in the virtual table (non-time-series and **TimeSeries** columns combined) must not exceed 32 KB.

If you specify either of the optional parameters, you must include them in the order shown in the syntax, but you can use either one without the other. For example, you can specify the *TSColName* parameter without including the *TSVTMode* parameter.

The virtual table is read-only. You cannot run INSERT, UPDATE, or DELETE statements on a virtual table that is based on an expression. When you query the virtual table, the WHERE clause in the SELECT statement cannot have any predicates based on the columns in the virtual table that are derived from the resulting **TimeSeries** subtype.

In the expression, you can use time series SQL routines and other SQL statements to manipulate the data, for example, the **AggregateBy** function and the **Apply** function.

You can use the following variables in the expression:
- `$ts_column_name`: If the base table has multiple **TimeSeries** columns, instead of specifying the name of the **TimeSeries** column in the expression, you can use the `$ts_column_name` variable to substitute the value of the *TScolName* parameter in the **TSCreateExpressionVirtualTab** procedure. Because the column name is a variable, you can use the same expression for each of the **TimeSeries** columns in the table.
- `$ts_begin_time`: Instead of specifying a DATETIME value, you can use this variable and specify the beginning time point of the time series in the WHERE

clause of the SELECT statement when you query the virtual table. If the WHERE clause does not contain the beginning timepoint, the first timepoint in the time series is used.

- `$ts_end_time`: Instead of specifying a DATETIME value, you can use this variable and specify the ending time point of the time series in the WHERE clause of the SELECT statement when you query the virtual table. If the WHERE clause does not contain the ending timepoint, the last timepoint in the time series is used.

Examples

The following examples use a table named **smartmeters** that contains a column named **meter_id** and a **TimeSeries** column named **readings**. The **TimeSeries** subtype has the columns **t** and **energy**.

Example 1: Find the daily maximum and minimum values

The following statement creates a virtual table named **smartmeters_vti_agg_max_min** based on a time series that contains the maximum and minimum energy readings per day:

```
EXECUTE PROCEDURE TSCreateExpressionVirtualTab(
       'smartmeters_vti_agg_max_min', 'smartmeters',
       'AggregateBy("max($energy),min($energy)",
                    "smartmeter_daily", readings, 0)',
       'tworeal_row');
```

The following query shows the daily maximum and minimum of the energy reading between 2011-0-01 and 2011-01-02:

```
SELECT * FROM smartmeters_vti_agg_max_min
 WHERE t >= '2011-01-01 00:00:00.00000'::datetime year to fraction(5)
   AND t <= '2011-01-02 23:59:59.99999'::datetime year to fraction(5);

meter_id   t                                 value1         value2

met00000   2011-01-01 00:00:00.00000 37.00000000000 9.000000000000
met00000   2011-01-02 00:00:00.00000 34.00000000000 8.000000000000
met00001   2011-01-01 00:00:00.00000 36.00000000000 9.000000000000
met00001   2011-01-02 00:00:00.00000 36.00000000000 10.00000000000
met00002   2011-01-01 00:00:00.00000 34.00000000000 9.000000000000
met00002   2011-01-02 00:00:00.00000 36.00000000000 10.00000000000

6 row(s) retrieved.
```

Example 2: Find the daily maximum of a running average

The following statement creates a virtual table named **smartmeters_vti_daily_max** that contains the daily maximum of the running average of the energy readings:

```
EXECUTE PROCEDURE TSCreateExpressionVirtualTab(
      'smartmeters_vti_daily_max', 'smartmeters',
      'AggregateBy("max($value)","smartmeter_daily",
                   Apply("TSRunningAvg($energy, 4)",
                         $ts_begin_time, $ts_end_time,
                         $ts_col_name)
                   ::TimeSeries(onereal_row), 0)',
      'onereal_row', 0, 'readings');
```

The `$ts_col_name` parameter is replaced by the column name specified by the **TSCreateExpressionVirtualTab** procedure, in this case, **readings**. The `$ts_begin_time` and `$ts_end_time` parameters are replaced when the virtual table is queried.

The following query shows the maximum daily average energy readings for two days:

```
SELECT * FROM smartmeters_vti_daily_max
 WHERE t >= '2011-01-01 00:00:00.00000'::datetime year to fraction(5)
   AND t <= '2011-01-02 23:59:59.99999'::datetime year to fraction(5);

meter_id   t                                  value

met00000   2011-01-01 00:00:00.00000 30.25000000000
met00000   2011-01-02 00:00:00.00000 29.50000000000
met00001   2011-01-01 00:00:00.00000 29.75000000000
met00001   2011-01-02 00:00:00.00000 31.00000000000
met00002   2011-01-01 00:00:00.00000 31.25000000000
met00002   2011-01-02 00:00:00.00000 28.75000000000

6 row(s) retrieved.
```

Related concepts:

"The display of data in virtual tables" on page 4-3

Related tasks:

"Creating a time series virtual table" on page 4-4

The TSVTMode parameter

The *TSVTMode* parameter configures the behavior and display of the virtual table.

You use the *TSVTMode* parameter with the **TSCreateVirtualTab** procedure to control:

- How data is updated in the base table when you perform an insert in the virtual table
- Whether NULL time series elements appear in a virtual table
- Whether updates to existing rows in the base table require accurate values for columns that are not part of the primary key
- Whether existing values in columns other than the **TimeSeries** column or the primary key columns can be updated.
- Whether NULL values can be used in the INSERT statement for columns other than the primary key columns.
- Whether hidden time series elements appear in a virtual table
- Whether data selected by time stamp exactly matches the specified timestamps or includes the last rows that are equal to or earlier than the specified timestamps.

You use the *TSVTMode* parameter with the **TSCreateExpressionVirtualTab** procedure to control:

- Whether NULL time series elements appear in a virtual table
- Whether hidden time series elements appear in a virtual table
- Whether data selected by time stamp exactly matches the specified timestamps or includes the last rows that are equal to or earlier than the specified timestamps.

The default value of the *TSVTMode* parameter sets the behavior of the virtual table. Each of the other values of the *TSVTMode* parameter reverses one aspect of the default behavior. You can set the *TSVTMode* parameter to a combination of the values. For example, if you set the *TSVTMode* parameter to 514 (512 + 2), both null and hidden elements appear in the virtual table. You can specify values for the *TSVTMode* parameter as either decimal numbers, as shown in the table, or as hexadecimal numbers.

Table 4-5. Settings for the TSVTMode parameter

Flag	Value	Description
TS_VTI_PUT_ELEM_NO_DUPS	0	Default. The virtual table has the following behavior: • Multiple elements for the same timepoint are not allowed. Updates to the underlying time series update existing elements for the same timepoint. Uses the **PutElemNoDups** function. • Null elements are not included in the virtual table. • If the base table has a primary key, the primary key is used to find the row to update and updates to the base table do not require accurate values for columns that are not part of the primary key. If the base table does not have a primary key, all columns in the table except the **TimeSeries** column are used to identify the row to be updated and updates to the base table require accurate values for every column in the base table other than the **TimeSeries** column. NOT NULL constraints are included in the virtual table for the primary key columns and other columns that have NOT NULL constraints in the base table. • For updates to existing rows, only the **TimeSeries** column can be updated. • Hidden elements are not included in the virtual table. • When selecting data from a virtual table by timestamps, the rows whose timestamps are closest to being equal to or earlier than the timestamps specified in the query are returned. If the time series is irregular, the returned rows show the same timestamps as specified in the query, regardless if the actual timestamps are the same.
TS_VTI_PUT_ELEM	1	Multiple elements for the same timepoint are allowed. Updates to the underlying time series insert elements even if elements already exist for the timepoints. Uses the **PutElem** function.
TS_VTI_SHOW_NULLS	2	Null elements appear in the virtual table. Hidden elements appear as null elements, unless the value 512 is also set.
TS_VTI_DISABLE_NOT_NULL_CONSTRAINT	16	For existing rows, you can specify NULL values for columns that are not part of the primary key, regardless if those columns have NOT NULL constraints in the base table. NOT NULL constraints are not included in the virtual table, but are enforced in the base table. For new rows, you can specify null values for columns that are not part of the primary key and do not have NOT NULL constraints.

Table 4-5. Settings for the TSVTMode parameter (continued)

Flag	Value	Description
TS_VTI_UPDATE_NONKEY_NOT_NULLS	32	This setting is valid only if the base table has a primary key. You can update the value of columns in an existing row that are not part of the primary key. You can specify NULL for non-primary key columns that you do not want to update. All columns that have non-NULL values in the INSERT statement are updated in the base table, except the primary key columns.
TS_VTI_UPDATE_NONKEY_INCLUDE_NULLS	64	This setting is valid only if the base table has a primary key. You can update the value of all the columns in an existing row that are not part of the primary key, including using null values for columns that allow null values. Columns that are not part of the primary key are updated to the value included in the INSERT statement. Columns that allow null values can be set to NULL.
TS_VTI_SCAN_HIDDEN	512	Hidden elements appear in the virtual table.
TS_VTI_SCAN_DISCREET	1024	When selecting data from a virtual table by timestamps, only rows whose timestamps are exactly equal to the timestamps specified in the query are returned.

Update columns in the base table

When you create a virtual table with the **TSCreateVirtualTab** procedure, you can update the data in the base table from the virtual table.

The following table describes how to control updating columns in the base table, assuming that the base table has a primary key. Whether the *NewTimeSeries* parameter is specified also affects the behavior of inserting data into the base table. For information on the effect of the *NewTimeSeries* parameter, see “TSCreateVirtualTab procedure” on page 4-4.

Table 4-6. TSVTMode parameter settings that affect which columns are updated in the base table

Columns to update	TSVTMode parameter setting
Update only the **TimeSeries** column. You must specify valid, but not necessarily accurate, values for non-primary key columns.	0
Update only the **TimeSeries** column. You can specify NULL as the values for non-primary key columns	16
Update the **TimeSeries** column and all other non-primary key columns that do not have null values in the INSERT statement.	32
Update the **TimeSeries** column and all other non-primary key columns. You can set columns that do not have NOT NULL constraints to null values.	64, 64 + 16

Table 4-6. TSVTMode parameter settings that affect which columns are updated in the base table (continued)

Columns to update	TSVTMode parameter setting
Update the **TimeSeries** column and all other non-primary key columns that have NOT NULL constraints. You can specify null values for columns that have NOT NULL constraints.	32 + 16

The following examples illustrate some of the settings for the *TSVTMode* parameter. The examples use a base table with columns for the account number, the meter identifier, the time series data, the meter owner, and the meter address. The account number and meter identifier columns are the primary key. The **TimeSeries** column contains columns for the time stamp, energy, and temperature. The owner column has a NOT NULL constraint. Each of the virtual tables created in the examples have the following initial one row that represents one times series element:

```
acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000
energy       33070
temperature  -13.0000000000
owner        John
address      5 Nowhere Place

1 row(s) retrieved.
```

Example 1: Setting the TSVTMode parameter to 0

The following statement creates a virtual table named **smartmeters_vti_nn** with the *TSVTMode* parameter set to 0:

```
EXECUTE PROCEDURE TSCreateVirtualTab('smartmeters_vti_nn',
        'smartmeters', 'origin(2011-01-01 00:00:00.00000),
        calendar(ts_15min), regular,threshold(20), container()', 0);
```

The following statement inserts a new row into the virtual table and a new element in the time series in the base table:

```
INSERT INTO smartmeters_vti_nn(acct_no,meter_id,t,energy,temperature,owner,address)
 VALUES(6546, 234,
        '2011-01-01 00:45:00.00000'::datetime year to fraction(5),
        3234, -12.00,
        'Ignored_value', 'Ignored_value');
1 row(s) inserted.
```

The values of the primary key columns match the original row. The values of the **owner** and **address** columns are ignored; they are not used to identify the row that must be updated and those values are not updated in the base table. After the INSERT statement, the virtual table contains two rows, and each contains the original values of the **owner** and **address** columns:

```
SELECT * FROM smartmeters_vti_nn;


acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000
energy       33070
temperature  -13.0000000000
owner        John
```

```
address      5 Nowhere Place

acct_no      6546
meter_id     234
t            2011-01-01 00:45:00.00000
energy       3234
temperature  -12.0000000000
owner        John
address      5 Nowhere Place

2 row(s) retrieved.
```

Example 2: Setting the TSVTMode parameter to 32

The following statement creates a virtual table named **smartmeters_vti_nn_nk_nn** with the *TSVTMode* parameter set to 32:

```
EXECUTE PROCEDURE TSCreateVirtualTab('smartmeters_vti_nn_nk_nn',
           'smartmeters', 'origin(2011-01-01 00:00:00.00000),
           calendar(ts_15min), regular,threshold(20), container()', 32);
```

The following statement inserts a new row into the virtual table and a new element in the time series in the base table:

```
INSERT INTO smartmeters_vti_nn_nk_nn(acct_no,meter_id,t,energy,
                                     temperature,owner,address)
 VALUES(6546, 234,
        '2011-01-01 00:45:00.00000'::datetime year to fraction(5),
        3234, -12.00,
        'Jim', NULL);
1 row(s) inserted.
```

The value of the **owner** column is updated to `Jim`. The value of the **address** column is not changed, because null values are ignored. The virtual table now contains two rows, each of which have the new value for the **owner** column and the existing value for the **address** column:

```
SELECT * FROM smartmeters_vti_nn_nk_nn;


acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000
energy       33070
temperature  -13.0000000000
owner        Jim
address      5 Nowhere Place

acct_no      6546
meter_id     234
t            2011-01-01 00:45:00.00000
energy       3234
temperature  -12.0000000000
owner        Jim
address      5 Nowhere Place

2 row(s) retrieved.
```

Example 3: Setting the TSVTMode parameter to 64

The following statement creates a virtual table named **smartmeters_vti_nn_nk_in** with the *TSVTMode* parameter set to 64:

```
EXECUTE PROCEDURE TSCreateVirtualTab('smartmeters_vti_nn_nk_in',
         'smartmeters', 'origin(2011-01-01 00:00:00.00000),
         calendar(ts_15min), regular,threshold(20), container()', 64);
```

The following statement inserts a new row into the virtual table and a new element in the time series in the base table:

```
INSERT INTO smartmeters_vti_nn_nk_in(acct_no,meter_id,t,energy,
                                     temperature,owner,address)
 VALUES(6546, 234,
        '2011-01-01 00:45:00.00000'::datetime year to fraction(5),
        3234, -12.00,
        'Jim', NULL);
1 row(s) inserted.
```

The value of the **owner** column is updated to `Jim`. The value of the **address** column is updated to a null value. The virtual table now contains two rows, each of which have the new value for the **owner** column and a null value for the **address** column:

```
SELECT * FROM smartmeters_vti_nn_nk_in;


acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000
energy       33070
temperature  -13.0000000000
owner        Jim
address

acct_no      6546
meter_id     234
t            2011-01-01 00:45:00.00000
energy       3234
temperature  -12.0000000000
owner        Jim
address

2 row(s) retrieved.
```

Duplicate timepoints

By default, the database server uses the **PutElemNoDups** function to add an element to the underlying time series. If an element already exists at the same timepoint, the existing element is updated. You can perform bulk updates of the underlying time series without producing duplicate elements for the same timepoints.

When the *TSVTMode* parameter includes the value `1`, the database server uses the **PutElem** function to add an element to the underlying time series. The **PutElem** function handles updates to existing data in an underlying irregular time series differently than does the **PutElemNoDups** function.

Null and hidden elements

The *TSVTMode* parameter includes options to display null or hidden time series elements in the virtual table. By default, if a base table has a null element at a specific timepoint, the virtual table has no entries for that timepoint. You can use the *TSVTMode* parameter to display null elements as a row of null values, plus the **timestamp** column and any non-time-series columns from the base table.

If the *TSVTMode* parameter includes the value `2`, null time series elements appear as null values in the virtual table. Hidden elements also appear as null values. If the *TSVTMode* parameter does not include the value `2`, null time series elements do not appear in the virtual table.

If the *TSVTMode* parameter includes the value 512, hidden time series elements appear in the virtual table; otherwise, they do not.

The following statements create four virtual tables that are all based on the same base table, named **inst**, which contains the **TimeSeries** column named **bars**. Each of the tables uses a different value for the *TSVTMode* parameter. The **inst_vt0** table does not show null or hidden elements. The **inst_vt2** table shows null elements. The **inst_vt512** table shows hidden elements. The **inst_vt514** table shows null and hidden elements.

```
execute procedure TSCreateVirtualTab( 'inst_vt0', 'inst', 0);
execute procedure TSCreateVirtualTab( 'inst_vt2', 'inst', 2);
execute procedure TSCreateVirtualTab( 'inst_vt512', 'inst', 512);
execute procedure TSCreateVirtualTab( 'inst_vt514', 'inst', 514);
```

The following statement hides one element by using the **HideElem** function:

```
update inst set bars = HideElem( bars,
    datetime(2011-01-18) year to day) where code = 'AA';
1 row(s) updated.
```

The following query shows that the **inst_vt0** table does not contain the hidden element for 2011-01-18:

```
select * from inst_vt0
where code = 'AA'
and t between datetime(2011-01-14) year to day
and datetime(2011-01-19) year to day
order by t;

code AA
t     2011-01-14 00:00:00.00000
high  69.25000000000
low   68.37500000000
final 68.62500000000
vol   462.0000000000

code AA
t     2011-01-19 00:00:00.00000
high  69.62500000000
low   69.12500000000
final 69.62500000000
vol   96.69999700000
2 row(s) retrieved.
```

The following query shows that the **inst_vt2** table contains null elements:

```
select * from inst_vt2
where code = 'AA'
and t between datetime(2011-01-14) year to day
and datetime(2011-01-19) year to day
order by t;

code AA
t     2011-01-14 00:00:00.00000
high  69.25000000000
low   68.37500000000
final 68.62500000000
vol   462.0000000000

code AA
t     2011-01-17 00:00:00.00000
high
low
final
vol
```

```
code AA
t     2011-01-18 00:00:00.00000
high
low
final
vol

code AA
t     2011-01-19 00:00:00.00000
high  69.62500000000
low   69.12500000000
final 69.62500000000
vol   96.69999700000
4 row(s) retrieved.
```

The following query shows that the **inst_vt512** table does contain the hidden element:

```
select * from inst_vt512
where code = 'AA'
and t between datetime(2011-01-14) year to day
and datetime(2011-01-19) year to day
order by t;

code AA
t     2011-01-14 00:00:00.00000
high  69.25000000000
low   68.37500000000
final 68.62500000000
vol   462.0000000000

code AA
t     2011-01-18 00:00:00.00000
high  69.75000000000
low   68.75000000000
final 69.62500000000
vol   281.2000100000

code AA
t     2011-01-19 00:00:00.00000
high  69.62500000000
low   69.12500000000
final 69.62500000000
vol   96.69999700000
3 row(s) retrieved.
```

The following query shows that the **inst_vt514** table does contain the hidden element and the null element:

```
select * from inst_vt514
where code = 'AA'
and t between datetime(2011-01-14) year to day
and datetime(2011-01-19) year to day
order by t;

code AA
t     2011-01-14 00:00:00.00000
high  69.25000000000
low   68.37500000000
final 68.62500000000
vol   462.0000000000

code AA
t     2011-01-17 00:00:00.00000
high
low
```

```
final
vol

code AA
t     2011-01-18 00:00:00.00000
high  69.75000000000
low   68.75000000000
final 69.62500000000
vol   281.2000100000

code AA
t     2011-01-19 00:00:00.00000
high  69.62500000000
low   69.12500000000
final 69.62500000000
vol   96.6999970000
4 row(s) retrieved.
```

Related concepts:

"The display of data in virtual tables" on page 4-3

"The insertion of data through virtual tables" on page 4-3

Related reference:

"PutElemNoDups function" on page 7-67

"PutElem function" on page 7-65

Drop a virtual table

You use the DROP statement to destroy a virtual table in the same way as you destroy any other database table. When you drop a virtual table, the underlying base table is unaffected.

Manage performance

You can enhance the performance of your virtual tables by performing the following tasks:

- Create an index on the key column of the base table. If the table has more than one column in the key, create a composite index consisting of all key columns.
- Run UPDATE STATISTICS on the base table and on its key columns:

  ```
  update statistics high for table daily_stocks;

  update statistics high for table daily_stocks (stock_id);
  ```

  You should run UPDATE STATISTICS after any load or delete operation; you might want to make these commands part of your routine database maintenance.

Trace functions

Trace functions are available to help you debug your work with virtual tables.

Restriction: You should not use these trace functions unless you are working with an IBM Informix Technical Support or Engineering professional.

The functions are:

TSSetTraceFile
: Allows you to specify a file to which the trace information is appended.

TSSetTraceLevel
: Sets the level of tracing to perform: in effect, turns tracing either on or off.

The TSSetTraceFile function

The **TSSetTraceFile** function specifies a file to which trace information is appended.

Syntax

```
TSSetTraceFile(traceFileName  lvarchar)
returns integer;
```

traceFileName
: The full path and name of the file to which trace information is appended.

Description

The file you specify using **TSSetTraceFile** overrides any current trace file. The file is located on the server computer. The default trace file is `/tmp/session_number.trc`.

TSSetTraceFile calls the **mi_set_trace_file()** DataBlade API function. For more information about **mi_set_trace_file()**, see the *IBM Informix DataBlade API Programmer's Guide*.

Returns

Returns `0` on success, `-1` on failure.

Example

The following example sets the file `/tmp/test1.trc` to receive trace information:

```
execute function TSSetTraceFile('/tmp/test1.trc');
```

TSSetTraceLevel function

The **TSSetTraceLevel** function sets the trace level of a trace class.

Syntax

```
TSSetTraceLevel(traceLevelSpec  lvarchar)
returns integer;
```

traceLevelSpec
: A character string specifying the trace level for a specific trace class. The format is TS_VTI_DEBUG *traceLevel*.

Description

TSSetTraceLevel sets the trace level of a trace class. The trace level determines what information is recorded for a given trace class. The trace class for virtual table information is TS_VTI_DEBUG. The level to enable tracing for the TS_VTI_DEBUG trace class is 1001. You must set the tracing level to 1001 or greater to enable tracing. By default, the trace level is below 1001.

TSSetTraceLevel calls the **mi_set_trace_level()** DataBlade API function. For more information about **mi_set_trace_level()**, see the *IBM Informix DataBlade API Programmer's Guide*.

Returns

Returns `0` on success, `-1` on failure.

Example

The following example turns tracing on:

```
execute function TSSetTraceLevel('TS_VTI_DEBUG 1001');
```

Chapter 5. Calendar pattern routines

You can use calendar pattern routines to manipulate calendar patterns.

Calendar pattern routines can perform the following types of operations:
- Create the intersection of calendar patterns
- Create the union of calendar patterns
- Alter a calendar pattern

Calendar and calendar pattern routines can be useful when comparing time series that are based on different calendars. For example, to compare peak time business usage of a computer network across multiple countries requires accounting for different sets of public holidays in each country. An efficient way to handle this is to define a calendar for each country and then create the calendar intersections to perform business-day comparisons.

Calendar pattern routines can be run in SQL statements or sent from an application using the DataBlade API function **mi_exec**.

Related concepts:

"Calendar" on page 1-7
"CalendarPattern data type" on page 2-1

The AndOp function

The **AndOp** function returns the intersection of two calendar patterns.

Syntax

```
AndOp ( cal_patt1 CalendarPattern,
       cal_patt2 CalendarPattern)
returns CalendarPattern;
```

cal_patt1
: The first calendar pattern.

cal_patt2
: The second calendar pattern.

Description

This function returns a calendar pattern that has every interval on that was on in both calendar patterns; the rest of the intervals are off. If the given patterns do not have the same interval unit, the pattern with the larger interval unit is expanded to match the other.

Returns

A calendar pattern that is the result of two others combined by the AND operator.

Example

The first **AndOp** statement returns the intersection of two daily calendar patterns, and the second **AndOp** statement returns the intersection of one hourly and one daily calendar pattern:

```
select * from CalendarPatterns
                where cp_name = 'workweek_day';

cp_name         workweek_day
cp_pattern         {1 off,5 on,1 off},day

select * from CalendarPatterns
                where cp_name = 'fourday_day';

cp_name         fourday_day
cp_pattern         {1 off,4 on,2 off},day

select * from CalendarPatterns
                where cp_name = 'workweek_hour';

cp_name         workweek_hour
cp_pattern         {32 off,9 on,15 off,9 on,15 off,9 on,15 off, 9
         on,15 off,9 on,31 off},hour

select AndOp(p1.cp_pattern, p2.cp_pattern)
                from CalendarPatterns p1, CalendarPatterns p2
              where p1.cp_name = 'workweek_day'
              and p2.cp_name = 'fourday_day';

(expression) {1 off,4 on,2 off},day

select AndOp(p1.cp_pattern, p2.cp_pattern)
                from CalendarPatterns p1, CalendarPatterns p2
              where p1.cp_name = 'workweek_hour'
              and p2.cp_name = 'fourday_day';

(expression) {32 off,9 on,15 off,9 on,15 off,9 on,15 off,9
          on,55 off},hour
```

Related reference:

"The AndOp function" on page 6-1

The CalPattStartDate function

The **CalPattStartDate** function takes a calendar name and returns a DATETIME containing the start date of the pattern for that calendar.

Syntax

```
CalPattStartDate(calname lvarchar)
returns datetime year to fraction(5);
```

calname
: The name of the source calendar.

Description

The equivalent API function is **ts_cal_pattstartdate()**.

Returns

The start date of the pattern for the given calendar.

Example

The following example returns the start dates of the calendar patterns for each calendar in the **CalendarTable** table:

```
select c_name, CalPattStartDate(c_name) from CalendarTable;
```

Related reference:

"The CalStartDate function" on page 6-5

"The ts_cal_pattstartdate() function" on page 9-8

The Collapse function

The **Collapse** function collapses the given calendar pattern into destination units, which must have a larger interval unit than that of the given calendar pattern.

Syntax

```
Collapse (cal_patt  CalendarPattern,
          interval   lvarchar)
returns CalendarPattern;
```

cal_patt
: The calendar pattern to be collapsed.

interval
: The destination time interval: minute, hour, day, week, month, or year.

Description

If any part of a destination unit is on, the whole unit is considered on.

Returns

The collapsed calendar pattern.

Example

The following statements convert an hourly calendar pattern into a daily calendar pattern:

```
select * from CalendarPatterns
                where cp_name = 'workweek_hour';


cp_name         workweek_hour
cp_pattern         {32 off,9 on,15 off,9 on,15 off,9 on,15 off,9
         on,15 off,9 on,31 off},hour

select Collapse(cp_pattern, 'day')
from CalendarPatterns
            where cp_name = 'workweek_hour';

(expression) {1 off,5 on,1 off},day
```

Related reference:

"The Expand function"

The Expand function

The **Expand** function expands the given calendar pattern into the destination units, which must have a smaller interval unit than that of the given calendar pattern.

Syntax

```
Expand (cal_patt  CalendarPattern,
        interval  lvarchar)
returns CalendarPattern;
```

cal_patt
: The calendar pattern to expand.

interval
: The destination time interval: second, minute, hour, day, week, or month.

Description

When a month is expanded, it is assumed to have 30 days.

Returns

The expanded calendar pattern.

Example

The following statements convert a daily calendar pattern into an hourly calendar pattern:

```
select * from CalendarPatterns
                where cp_name = 'workweek_day';

cp_name         workweek_day
cp_pattern         {1 off,5 on,1 off},day

select Expand(cp_pattern, 'hour')
              from CalendarPatterns
            where cp_name = 'workweek_day';

(expression) {24 off,120 on,24 off},hour
```

Related reference:

"The Collapse function" on page 5-3

The NotOp function

The **NotOp** function turns all on intervals off and all off intervals on in the given calendar pattern.

Syntax

```
NotOp (cal_patt CalendarPattern)
returns CalendarPattern;
```

cal_patt
: The calendar pattern to convert.

Returns

The inverted calendar pattern.

Example

The following statement converts the **workweek_day** calendar:

```
select * from CalendarPatterns
                where cp_name = 'workweek_day';

cp_name               workweek_day
cp_pattern         {1 off,5 on,1 off},day

select NotOp(cp_pattern)
from CalendarPatterns
         where cp_name = 'workweek_day';

(expression) {1 on,5 off,1 on}, day
```

The OrOp function

The **OrOp** function returns the union of the two calendar patterns.

Syntax

```
OrOp (cal_patt1  CalendarPattern,
      cal_patt2  CalendarPattern)
returns CalendarPattern;
```

cal_patt1
The first calendar pattern.

cal_patt2
The second calendar pattern.

Description

This function returns a calendar pattern that has every interval on that was on in either of the calendar patterns; the rest of the intervals are off. If the two patterns have different sizes of interval units, the resultant pattern has the smaller of the two intervals.

Returns

A calendar pattern that is the result of two others combined with the OR operator.

Example

The first **OrOp** statement in the example returns the union of two daily calendar patterns. The second **OrOp** statement returns the union of one hourly and one daily calendar pattern:

```
select * from CalendarPatterns
                where cp_name = 'workweek_day';

cp_name         workweek_day
cp_pattern         {1 off,5 on,1 off},day

select * from CalendarPatterns
                where cp_name = 'fourday_day';
```

```
cp_name         fourday_day
cp_pattern         {1 off,4 on,2 off},day

select * from CalendarPatterns
                where cp_name = 'workweek_hour';

cp_name         workweek_hour
cp_pattern         {32 off,9 on,15 off,9 on,15 off,9 on,15 off, 9
         on,15 off,9 on,31 off},hour

select OrOp(p1.cp_pattern, p2.cp_pattern)
                from CalendarPatterns p1, CalendarPatterns p2
              where p1.cp_name = 'workweek_day'
              and p2.cp_name = 'fourday_day';

(expression) {1 off,5 on,1 off},day

(expression) {24 off,96 on,8 off,9 on,31 off},hour
```

Related reference:

"The OrOp function" on page 6-5

Chapter 6. Calendar routines

You can use calendar routines to manipulate calendars.

Calendar routines can perform the following types of operations:
- Create the intersection of calendars
- Create the union of calendars
- Return information about the calendar

Calendar routines can be useful when comparing time series that are based on different calendars. For example, to compare peak time business usage of a computer network across multiple countries requires accounting for different sets of public holidays in each country. An efficient way to handle this is to define a calendar for each country and then create the calendar intersections to perform business-day comparisons.

Calendar routines can be run in SQL statements or sent from an application using the DataBlade API function **mi_exec**.

Related concepts:

"Calendar" on page 1-7
"Calendar data type" on page 2-3

The AndOp function

The **AndOp** function returns the intersection of the two calendars.

Syntax

```
AndOp (cal1  Calendar,
       cal2  Calendar)
returns Calendar;
```

cal1 The first calendar.

cal2 The second calendar.

Description

This function returns a calendar that has every interval on that was on in both calendars; the rest of the intervals are off. The resultant calendar takes the later of the two start dates and the later of the two pattern start dates.

If the two calendars have different size interval units, the resultant calendar has the smaller of the two intervals.

Returns

A calendar that is the result of two other calendars combined with the AND operator.

Example

The following **AndOp** statement returns the intersection of an hourly calendar with a daily calendar having a different start date:

```
select c_calendar from CalendarTable
            where c_name = 'hourcal';

c_calendar   startdate(2011-01-01 00:00:00), pattstart(2011-
          01-02 00:00:00), pattern({32 off,9 on,15 off,9
          on,15 off,9 on,15 off,9 on,15 off, 9 on,31
          off},hour)

select c_calendar from CalendarTable
            where c_name = 'daycal';

c_calendar   startdate(2011-04-01 00:00:00), pattstart(2011-
          04-03 00:00:00), pattern({1 off,5 on,1 off},day)

select AndOp(c1.c_calendar, c2.c_calendar)
            from CalendarTable c1, CalendarTable c2
            where c1.c_name = 'daycal' and c2.c_name = 'hourcal';
```

(expression)

```
        startdate(2011-04-01 00:00:00), pattstart(2011-04-03
        00:00:00), pattern({32 off,9 on,15 off,9 on,15 off,9 on,15
        off,9 on,15 off, 9 on ,31 off},hour)
```

Related reference:

"The AndOp function" on page 5-1

"The OrOp function" on page 6-5

The CalIndex function

The **CalIndex** function returns the number of valid intervals in a calendar between two given time stamps.

Syntax

```
CalIndex(cal_name     lvarchar,
         begin_stamp   datetime year to fraction(5),
         end_stamp              datetime year to fraction(5))
returns integer;
```

cal_name
: The name of the calendar.

begin_stamp
: The begin point of the range. Must not be earlier than the calendar start date.

end_stamp
: The end point of the range.

Description

The equivalent API function is **ts_cal_index()**.

Returns

The number of valid intervals in the given calendar between the two time stamps.

Example

The following query returns the number of intervals in the calendar **daycal** between 2011-01-03 and 2011-01-05:

```
select CalIndex('daycal',
      '2011-01-03 00:00:00.00000',
      '2011-01-05 00:00:00.00000')
    from systables
    where tabid = 1;
```

Related reference:

"The ts_cal_range() function" on page 9-9

"The ts_cal_range_index() function" on page 9-10

"The ts_cal_stamp() function" on page 9-11

"GetIndex function" on page 7-43

"GetStamp function" on page 7-54

The CalRange function

The **CalRange** function returns a set of valid time stamps within a range.

Syntax

```
CalRange(cal_name                lvarchar,
        begin_stamp    datetime year to fraction(5),
        end_stamp          datetime year to fraction(5))
returns list(datetime year to fraction(5));

CalRange(cal_name                lvarchar,
        begin_stamp    datetime year to fraction(5),
        num_stamps          integer)
returns list(datetime year to fraction(5));
```

cal_name
    The name of the calendar.

begin_stamp
    The begin point of the range. Must be no earlier than the first time stamp in the calendar.

end_stamp
    The end point of the range.

num_stamps
    The number of time stamps to return.

Description

The first syntax specifies the range as between two given time stamps. The second syntax specifies the number of valid time stamps to return after a given time stamp.

The equivalent API function is **ts_cal_range()**.

Returns

A list of time stamps.

Example

The following query returns a list of all the time stamps between 2011-01-03 and 2011-01-05 in the calendar **daycal**:

```
execute function CalRange('daycal',
      '2011-01-03 00:00:00.00000',
      '2011-01-05 00:00:00.00000'::datetime year
          to fraction(5));
```

The following query returns a list of the two time stamps following 2011-01-03 in the calendar **daycal**:

```
execute function CalRange('daycal',
   '2011-01-03 00:00:00.00000', 2);
```

Related reference:

"The ts_cal_range() function" on page 9-9

"The ts_cal_range_index() function" on page 9-10

"The ts_cal_stamp() function" on page 9-11

"GetIndex function" on page 7-43

"GetStamp function" on page 7-54

The CalStamp function

The **CalStamp** function returns the time stamp at a given number of calendar intervals after a given time stamp.

Syntax

```
CalStamp(cal_name    lvarchar,
         tstamp      datetime year to fraction(5),
         num_stamps  integer)
returns datetime year to fraction(5);
```

cal_name
The name of the calendar.

tstamp The input time stamp.

num_stamps
The number of calendar intervals after the input time stamp. Cannot be negative.

Description

The equivalent API function is **ts_cal_stamp()**.

Returns

The time stamp representing the given offset.

Example

The following example returns the time stamp that is two intervals after 2011-01-03:

```
execute function CalStamp('daycal',
   '2011-01-03 00:00:00.00000', 2);
```

Related reference:

"The ts_cal_range() function" on page 9-9

"The ts_cal_range_index() function" on page 9-10

"The ts_cal_stamp() function" on page 9-11

The CalStartDate function

The **CalStartDate** function takes a calendar name and returns a DATETIME value containing the start date of that calendar.

Syntax

```
CalStartDate(cal_name    lvarchar)
returns datetime year to fraction(5);
```

cal_name
: The name of the calendar.

Description

The equivalent API function is **ts_cal_startdate()**.

Returns

The start date of the given calendar.

Example

The following example returns the start dates of all the calendars in the **CalendarTable** table:

```
select c_name, CalStartDate(c_name) from CalendarTable;
```

Related reference:

"The CalPattStartDate function" on page 5-2

"The ts_cal_startdate() function" on page 9-11

The OrOp function

The **OrOp** function returns the union of the two calendars.

Syntax

```
OrOp (cal1 Calendar,
      cal2 Calendar)
returns Calendar;
```

cal1 The first calendar to be combined.

cal2 The second calendar to be combined.

Description

This function returns a calendar that has every interval on that was on in either calendar; the rest of the intervals are off. The resultant calendar takes the earlier of the two start dates and the two pattern start dates.

If the two calendars have different sizes of interval units, the resultant calendar has the smaller of the two intervals.

Returns

A calendar that is the result of two others combined with the OR operator.

Example

The following **OrOp** function returns the union of an hourly calendar with a daily calendar having a different start date:

```
select c_calendar from CalendarTable
            where c_name = 'hourcal';

c_calendar      startdate(2011-01-01 00:00:00), pattstart(2011-
          01-02 00:00:00), pattern({32 off,9 on,15 off,9
          on,15 off,9 on,15 off,9 on,15 off, 9 on,31
          off},hour)

select c_calendar from CalendarTable
            where c_name = 'daycal';

c_calendar   startdate(2011-04-01 00:00:00), pattstart(2011-
          04-03 00:00:00), pattern({1 off,5 on,1 off},day)

select OrOp(c1.c_calendar, c2.c_calendar)
            from CalendarTable c1, CalendarTable c2
            where c1.c_name = 'daycal' and c2.c_name = 'hourcal';
```

(expression)

```
        startdate(2011-01-01 00:00:00), pattstart(2011-01-02
        00:00:00), pattern({24 off,120 on,24 off},hour)
```

Related reference:

"The OrOp function" on page 5-5
"The AndOp function" on page 6-1

Chapter 7. Time series SQL routines

Time series SQL routines create instances of a particular time series type, and then add data to or change data in the time series type. SQL routines are also provided to examine, analyze, manipulate, and aggregate the data within a time series.

The several data types and tables used throughout the examples in this chapter are listed in the following table.

Type/Table	Description
stock_bar	Type containing **timestamp**(DATETIME), **high**, **low**, **final**, and **vol** columns
daily_stocks	Table containing **stock_id**, **stock_name**, and **stock_data** columns
stock_trade	Type containing **timestamp**(DATETIME), **price**, **vol**, **trade**, **broker**, **buyer**, and **seller** columns
activity_stocks	Table containing **stock_id** and **activity_data** columns

For more information about these data types and tables, see "Creating a TimeSeries subtype" on page 3-6 and "Create the database table" on page 3-6.

The schema for these examples is in the `$INFORMIXDIR/TimeSeries.`*`version`*`/examples` directory.

Related concepts:

"Planning for accessing time series data" on page 1-12

Time series SQL routines sorted by task

Time series SQL routines are sorted into logical areas based on the type of task.

Table 7-1. Time series SQL routines by task type

Task type	Description	Routine name
Get information from a time series	Get the origin	**GetOrigin**
	Get the interval	**GetInterval**
	Get the calendar	**GetCalendar**
	Get the calendar name	**GetCalendarName**
	Get the container name	**GetContainerName**
	Get user-defined metadata	**GetMetaData**
	Get the metadata type	**GetMetaTypeName**
	Determine whether a time series is regular	**IsRegular**
	Get the instance ID if the time series is stored in a container	**InstanceId**
Convert between a time stamp and an offset	Return the offset, given the time stamp	**GetIndex**
	Return the time stamp, given the offset	**GetStamp**
Count the number of elements	Return the number of elements	**GetNelems**
	Get the number of elements between two time stamps	**ClipGetCount**

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description	Routine name
Select individual elements	Get the element associated with a given time stamp	**GetElem**
	Get the element at or before a time stamp	**GetLastValid**
	Get the element after a time stamp	**GetNextValid**
	Get the element before a time stamp	**GetPreviousValid**
	Get the element at a specified position	**GetNthElem**
	Get the first element	**GetFirstElem**
	Get the last element	**GetLastElem**
	Get the last non-null element	**GetLastNonNull**
	Get the next non-null element	**GetNextNonNull**
Modify elements or a set of elements	Add or update a single element	**PutElem**
	Add or update a single element	**PutElemNoDups**
	Add or update a single element at a given offset	**PutNthElem** (regular only)
	Add or update an entire set	**PutSet**
	Delete an element at a given timepoint	**DelElem**
	Delete all elements in a specified time range	**DelClip**
	Delete all elements in a specified time range and free space in any part of a time series	**DelRange**
	Delete all elements in a specified time range and free space at the end of a time series	**DelTrim**
	Insert an element	**InsElem**
	Insert a set	**InsSet**
	Free empty pages in a specified time range or throughout the time series	**NullCleanup**
	Update an element	**UpdElem**
	Update a set	**UpdSet**
	Put every element of one time series into another time series	**PutTimeSeries**
Modify metadata	Update user-defined metadata	**UpdMetaData**
Make elements visible or invisible to a scan	Make an element invisible	**HideElem**
	Make a range of elements invisible	**HideRange**
	Make an element visible	**RevealElem**
	Make a range of elements visible	**RevealRange**
Check for null or hidden elements	Determine if an element is hidden.	**ElemIsHidden**
	Determine if an element is null.	**ElemIsNull**
Extract and use part of a time series	Extract a period between two time stamps or corresponding to a set of values and run an expression or function on every entry	**Apply**
	Extract data between two timepoints	**Clip**
	Clip a certain number of elements	**ClipCount**
	Output values in XML format	**TStoXML**
	Extract a period that includes a given time	**WithinC**
	Extract a period starting or ending at a given time	**WithinR**

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description	Routine name
Apply a new calendar to a time series	Apply a calendar	**ApplyCalendar**
Create and load a time series	Load data from a client file	**BulkLoad**
	Create a regular empty time series, a regular populated time series, or a regular time series with metadata	**TSCreate**
	Create an irregular empty time series, an irregular populated time series, or an irregular time series with metadata	**TSCreateIrr**
Find the intersection or union of time series	Build the intersection of multiple time series and optionally clip the result	**Intersect**
	Build the union of multiple time series and optionally clip the result	**Union**
Iterator functions	Convert time series data to tabular form	**Transpose**
Aggregate functions	Return a list (collection of rows) containing all elements in a time series	**TSSetToList**
	Return a list of one column in the time series	**TSColNameToList**
	Return a list of one column in the time series	**TSColNumToList**
	Return a list containing the columns of the time series plus non-time-series columns	**TSRowToList**
	Return a list containing one column of the time series plus non-time-series columns	**TSRowNameToList**
	Return a list containing one column of the time series plus non-time-series columns	**TSRowNumToList**
Used within the **Apply** function to perform statistical calculations on a time series	Perform a sum over a time series type	**Sum**
	Sum SMALLFLOAT or DOUBLE PRECISION values	**TSAddPrevious**
	Compute the decay function	**TSDecay**
	Compute a running average over a specified number of values	**TSRunningAvg**
	Compute a running correlation between two time series over a specified number of values	**TsRunningCor**
	Compute a running median over a specified number of values	**TsRunningMed**
	Compute a running sum over a specified number of values	**TSRunningSum**
	Compute a running variance over a specified number of values	**TSRunningVar**
	Compare SMALLFLOAT or DOUBLE PRECISION values	**TSCmp**
	Return a previously saved value	**TSPrevious**

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description	Routine name
Perform an arithmetic operation on one or two time series	Add two time series together	**Plus**
	Subtract one time series from another	**Minus**
	Multiply one time series by another	**Times**
	Divide one time series by another	**Divide**
	Raise the first argument to the power of the second	**Pow**
	Get the absolute value	**Abs**
	Exponentiate the time series	**Exp**
	Get the natural logarithm of a time series	**Logn**
	Get the modulus or remainder of a division of one time series by another	**Mod**
	Negate a time series	**Negate**
	Return the argument and the argument is bound to the unary + operator	**Positive**
	Round the time series to the nearest whole number	**Round**
	Get the square root of the time series	**Sqrt**
	Get the cosine of the time series	**Cos**
	Get the sine of the time series	**Sin**
	Get the tangent of the time series	**Tan**
	Get the arc cosine of the time series	**Acos**
	Get the arc sine of the time series	**Asin**
	Get the arc tangent of the time series	**Atan**
	Get the arc tangent for two time series	**Atan2**
Perform an arithmetic operation on one or two time series (continued)	Apply a binary function to a pair of time series, or to a time series and a compatible row type or number	**ApplyBinaryTsOp**
	Apply a unary function to a time series	**ApplyUnaryTsOp**
	Apply another function to a set of time series	**ApplyOpToTsSet**
Aggregate time series values	Aggregate values in a time series from a single row	**AggregateBy**
	Aggregate values in a time series from a single row over a specified time range	**AggregateRange**
	Aggregate time series values across multiple rows	**TSRollup**
Create a time series that lags	Create a time series that lags the source time series by a given offset	**Lag** (regular only)
Reset the origin	Reset the origin	**SetOrigin**
Manage containers	Create a container	**TSContainerCreate**
	Destroy a container	**TSContainerDestroy**
	Set the container name	**SetContainerName**
	Specify the container pool for inserting data into a time series	**TSContainerPoolRoundRobin**
	Add a container into a container pool or remove a container from a container pool	**TSContainerSetPool**

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description	Routine name
Monitor containers	Return the number of elements in one or all containers	**TSContainerNElems**
	Return the percentage of space used in one or all containers	**TSContainerPctUsed**
	Return the total number of pages allocated to one or all containers	**TSContainerTotalPages**
	Return the number of pages used by one or all containers	**TSContainerTotalUsed**
	Return the number of elements, the number of pages used, and the total number of pages allocated for one or all containers	**TSContainerUsage**

The following routines are only used with regular time series:
- **Lag**
- **PutNthElem**
- **TSCreate**

The **TSCreateIrr** function is used only with irregular time series.

The *flags* argument values

Many of the time series SQL functions provide a *flags* argument to determine how the function handles null values and hidden elements. These values are described here.

Some functions have specific settings of the *flags* argument that are relevant only to that function. In these cases, the *flags* argument values are documented with the function.

The value of the *flags* argument is the sum of the desired flag values from the following table.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Indicates that the time series can be read and written to.
TSOPEN_READ_HIDDEN	1	Indicates that hidden elements should be treated as if they are not hidden.
TSOPEN_WRITE_HIDDEN	2	Allows hidden elements to be written to without first revealing them. The element remains hidden afterward.
TSOPEN_WRITE_AND_HIDE	4	Causes any elements written to a time series also to be marked as hidden.
TSWRITE_AND_REVEAL	8	Reveals any hidden element written to.
TSOPEN_NO_NULLS	32	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as `NULL`. If TSOPEN_NO_NULLS is set, an element that has each column set to `NULL` is returned instead.

These flags can be used in any combination except the following four:

- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_REVEAL
- TSOPEN_WRITE_AND_REVEAL and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN, and TSOPEN_WRITE_AND_HIDE, and TSOPEN_WRITE_AND_REVEAL

The TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_AND_HIDE flags cannot be used with TSOPEN_READ_HIDDEN.

Abs function

The **Abs** function returns the absolute value of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt** and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Acos function

The **Acos** function returns the arc cosine of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

AggregateBy function

The **AggregateBy** function aggregates the values in a time series using a new time interval that you specify by providing a calendar.

This function can be used to convert a time series with a small interval to a time series with a larger interval: for instance, to produce a weekly time series from a daily time series.

If you supply the optional *start* and *end* DATETIME parameters, just that part of the time series is aggregated to the new time interval.

Syntax

```
AggregateBy(agg_express   lvarchar,
           cal_name       lvarchar,
           ts            TimeSeries
           flags         integer default 0
           start datetime year to fraction(5) default NULL,
           end datetime year to fraction(5) default NULL
)
returns TimeSeries;
```

agg_express
: A comma-separated list of these SQL aggregate operators: MIN, MAX, MEDIAN, SUM, AVG, FIRST, LAST, or Nth.

cal_name
The name of a calendar that defines the aggregation period.

ts The time series to be aggregated.

flags **(optional)**
Determines how data points in off periods of calendars are handled during aggregation. See "The flags argument values" on page 7-8.

start **(optional)**
The date and time at which to start aggregation.

end **(optional)**
The date and time at which to end aggregation.

Description

The **AggregateBy** function converts the input time series to a regular time series with a calendar given by the *cal_name* argument. The *agg_express* expressions operate on a column of the input time series, specified as **$*colname*** or **$*colnumber***: for example, **$high** or **$1**. The resulting time series has a time stamp column plus one column for each expression in the list.

An error is raised if the MIN, MAX, MEDIAN, SUM, or AVG expression is used on a non-numeric column.

The Nth expression returns the value of a column for the specified aggregation period, using the following syntax:

```
Nth($col, n)
```

$col The name or number of a column within a **TimeSeries** row.

n A positive or negative number indicating the position of the **TimeSeries** row within the aggregation period. Positive values of *n* begin at the first row in the aggregation period; therefore, Nth($col, 1) is equivalent to FIRST($col). Negative values of *n* begin with the last row in the aggregation period; therefore, Nth($col, -1) is equivalent to LAST($col).

If an aggregation period does not have a value for the *n*th row, then the Nth function returns a null value for that period. The Nth function is more efficient for positive values of the *n* argument than for negative values.

An aggregation time period is denoted by the start date and time of the period.

The origin of the aggregated output time series is the first period on or before the origin of the input time series. Each output period is the aggregation of all input periods from the start of the output period up to, but not including, the start of the next output period.

For instance, suppose you want to aggregate a daily time series that starts on Tuesday, Jan. 4, 2011, to a weekly time series. The input calendar, named "days," starts at 12:00 a.m., and the output calendar, named "weeks," starts at 12:00 a.m., on Monday.

The first output time is 00:00 Jan. 3, 2011; it is the aggregation of all input values from the input origin, Jan. 4, 2011, to 23:59:59.99999 Jan. 9, 2011. The second output time is 00:00 Jan. 10, 2011; it is the aggregation of all input values from 00:00 Jan 10, 2011 to 23:59:59.99999 Jan. 16, 2011.

Normally, **AggregateBy** is used to aggregate from a fine-grained regular time series to a coarser-grained one. However, the following scenarios are also supported:

- Converting from a regular time series to a time series with a calendar of the same granularity. In this case, **AggregateBy** shifts the times back to accommodate differences in the calendar start times: for example, 00:00 from 8:00. Elements can be removed or null elements added to accommodate differences in the on/off pattern.
- Converting from a regular time series to one with a calendar of finer granularity. In this case, **AggregateBy** replicates values.
- The input time series is irregular. Because the granularity of an irregular time series does not depend on the granularity of the calendar, this case is treated like aggregation from a fine-grained time series to a coarser-grained one. This type of aggregation always produces a regular time series.

The flags argument values

The *flags* argument determines how data points in the off periods of calendars are handled during aggregation and how hidden elements are managed. It can have the following values.

0 (Default) Data in off periods is aggregated with the next output period.

1 Data in off periods is aggregated with the previous output period.

2 Indicates that the scan should run with the TS_SCAN_HIDDEN flag set (hidden elements are returned).

4 Indicates that the scan should run with the TS_SCAN_SKIP_HIDDEN flag set (hidden elements are not returned).

For example, consider an input time series that has a daily calendar with no off days: it has data from weekdays and weekends. If you aggregate this data by a business-day calendar (5 days on, 2 days off, starting on a Monday), a *flags* argument of 0 causes weekend data to be aggregated with the next Monday's data, and a *flags* argument of 1 causes weekend data to be aggregated with the previous Friday's data.

For another example, consider a quarterly calendar defined as:

```
'startdate(2010-1-1 00:00:00.00000), pattstart(2010-1-1 00:00:00.00000),
pattern({1 on, 2 off}, month'
```

If you aggregate this calendar with either a *flags* argument of 0 or no *flags* argument, all input points up to, but not including, 2010-2-1 00:00:00.00000 are aggregated into the first output element. All points from 2010-2-1 00:00:00.00000 up to, but not including, 2010-5-1 00:00:00.00000 are aggregated into the second output element, and so on.

If the *flags* argument is 1, all input points up to but not including 2010- 4-1 00:00:00.00000 are aggregated into the first output element. All points from 2010-4-1 00:00:00.00000 up to, but not including, 2010-7-1 00:00:00.00000 are aggregated into the second output element, and so on. The **AggregateBy** clause might look like this:

```
AggregateBy('max($high)', 'quarterlycal', ts, 1);
```

Returns

The aggregated time series, which is always regular, if you are aggregating to a new time interval.

Example

The following query aggregates the **daily_stocks** time series to a weekly time series:

```
insert into daily_stocks( stock_id, stock_name, stock_data)
   select stock_id, stock_name,
   AggregateBy( 'max($high), min($low),last($final),sum($vol)',
   'weekcal', stock_data)::TimeSeries(stock_bar)
   from daily_stocks;
```

The following query clause selects the second price from each week:

```
AggregateBy( 'Nth($price, 2)', 'weekly', ts)
```

This query clause selects the second to the last price from each week:

```
AggregateBy( 'Nth($price, -2)', 'weekly', ts)
```

Related reference:

"AggregateRange function"

"Apply function" on page 7-11

"PutTimeSeries function" on page 7-70

"TSRollup function" on page 7-98

AggregateRange function

The **AggregateRange** function produces an aggregate over each element for a time range specified by *start* and *end* DATETIME parameters.

Syntax

```
AggregateRange(agg_express lvarchar,
           ts             TimeSeries
           flags          integer default 0
           start    datetime year to fraction(5) default NULL,
           end     datetime year to fraction(5) default NULL
)
returns row;
```

agg_express
: A comma-separated list of these SQL aggregate operators: MIN, MAX, MEDIAN, SUM, AVG, FIRST, LAST, or Nth.

ts
: The time series to be aggregated.

flags **(optional)**
: See "The flags argument values" on page 7-10.

  You cannot use a *flags* argument value of 1 with this function.

start **(optional)**
: The date and time at which to start aggregation.

end **(optional)**
: The date and time at which to end aggregation.

Description

The **AggegateRange** function converts the input section of a time series to a row of aggregate values. The *agg_express* expressions operate on a column of the input time series, specified as **$colname** or **$colnumber**: for example, **$high**, or **$1**.

An error is raised if the MIN, MAX, MEDIAN, SUM, or AVG expression is used on a non-numeric column.

The Nth expression returns the value of a column for the specified aggregation period, using the following syntax:

Nth($*col*, *n*)

$col
: The name or number of a column within a **TimeSeries** row.

n
: A positive or negative number indicating the position of the TimeSeries row within the aggregation period. Positive values of *n* begin at the first row in the aggregation period; therefore, Nth($col, 1) is equivalent to FIRST($col). Negative values of *n* begin with the last row in the aggregation period; therefore, Nth($col, -1) is equivalent to LAST($col).

  If an aggregation period does not have a value for the *n*th row, then the Nth function returns a null value for that period. The Nth function is more efficient for positive values of the n argument than for negative values.

An aggregation time period is denoted by the start date and time of the period.

The flags argument values

The *flags* argument determines how data points in the off periods of calendars are handled during aggregation and how hidden elements are managed. It can have the following values.

0 (default)
: Data in off periods is aggregated with the next output period.

2
: Indicates that the scan should run with the TS_SCAN_HIDDEN flag set (hidden elements are returned).

4
: Indicates that the scan should run with the TS_SCAN_SKIP_HIDDEN flag set (hidden elements are not returned).

Returns

A single element (row).

Example

The following example produces an average of the values in the column **high** of the time series called **stock_data**. First, the example creates the row type, *elemval*, as a cast for the result.

```
create row type elemval (tstamp datetime year to fraction(5),
                        high double precision);


select
  AggregateRange('avg($high)', stock_data)::elemval
from daily_stocks;
```

Related reference:

"AggregateBy function" on page 7-6
"Apply function"

Apply function

The **Apply** function queries one or more time series and applies a user-specified SQL expression or function to the selected time series elements.

Syntax

```
Apply(sql_express  lvarchar,
     ts           TimeSeries, ...)
returns TimeSeries;

Apply(sql_express  lvarchar,
     multiset_ts  multiset(TimeSeries))
returns TimeSeries;

Apply(sql_express  lvarchar,
     filter       lvarchar,
     ts           TimeSeries, ...)
returns TimeSeries;

Apply(sql_express lvarchar,
     filter      lvarchar,
     multiset_ts multiset(TimeSeries))
returns TimeSeries;

Apply(sql_express lvarchar,
     begin_stamp datetime year to fraction(5),
     end_stamp   datetime year to fraction(5),
     ts          TimeSeries, ...)
returns TimeSeries with (handlesnulls);

Apply(sql_express lvarchar,
     begin_stamp datetime year to fraction(5),
     end_stamp    datetime year to fraction(5),
     multiset_ts multiset(TimeSeries))
returns TimeSeries with (handlesnulls);

Apply(sql_express lvarchar,
     filter      lvarchar,
     begin_stamp datetime year to fraction(5),
     end_stamp   datetime year to fraction(5),
     ts          TimeSeries, ...)
returns TimeSeries with (handlesnulls);

Apply(sql_express lvarchar,
     filter      lvarchar,
     begin_stamp datetime year to fraction(5),
     end_stamp   datetime year to fraction(5),
     multiset_ts multiset(TimeSeries))
returns TimeSeries with (handlesnulls);
```

sql_express
: The SQL expression or function to evaluate.

filter
: The filter expression used to select time series elements.

begin_stamp
: The begin point of the range. See "Clip function" on page 7-26 for more detail about range specifications.

end_stamp
: The end point of the range. See "Clip function" on page 7-26 for more detail about range specifications.

ts
: The first *ts* argument is the first series, the second *ts* argument is the second series, and so on. This function can take up to eight *ts* arguments. The order of the arguments must correspond to the desired order in the SQL expression or function. There is no limit to the number of $ parameters in the expression.

multiset_ts
: A multiset of time series.

Description

This function runs a user-specified SQL expression on the given time series and produces a new time series containing the result of the expression at each qualifying element of the input time series.

You can qualify the elements from the input time series by specifying a time period to clip and by using a filter expression.

The *sql_express* argument is a comma-separated list of expressions to run for each selected element. There is no limit to the number of expressions you can run. The results of the expressions must match the corresponding columns of the result time series minus the first time stamp column. Do not specify the first time stamp as the first expression; the first time stamp is generated for each expression result.

The parameters to the expression can be an input element or any column of an input time series. You should use $, followed by the position of a given time series on the input time series list to represent its data element, plus a dot, then the number of the column. Both the position number and column number are zero-based.

For example, **$0** means the element of the first input time series, **$0.0** represents its time stamp column, and **$0.1** is the column following the time stamp column. Another way to refer to a column is to use the column name directly, instead of the column number. Suppose the second time series has a column called **high** then you can use **$1.high** to refer to it. If the **high** column is the second column in the element, **$1.high** is equivalent to **$1.1**.

If **Apply** has only one time series argument, you can refer to the column name without the time series position part; hence, **$0.high** is the same as **$high**. Notice that **$0** always means the whole element of the first time series. It does *not* mean the first column of the time series, even if there is only one time series argument.

If you use a function as your expression, then it must take the subtype of each input time series in that order as its arguments and return a row type that corresponds to the subtype of the result time series of **Apply**. In most cases, it is faster to evaluate a function than to evaluate a generic expression. If performance is critical, you should implement the calculation to be performed in a function and use the function syntax. See "Example" on page 7-14 for how to achieve this.

The following examples show valid expressions for **Apply** to apply. Assume two argument time series with the same subtype **daybar(t** DATETIME YEAR TO FRACTION(5)**, high** REAL**, low** REAL**, close** REAL**, vol** REAL**)**. The expression could be any of:

- `"$0.high + $1.high)/2, ($0.low + $1.low)/2"`
- `"($0.1 + $1.1)/2, ($0.2 + $1.2)/2"`
- `"$0.high, $1.high"`
- `"avghigh"`

The signature of **avghigh** is:

```
"avghigh(arg1 daybar, arg2 daybar) returns (one_real)"
```

The syntax for the *filter* argument is similar to the previous expression, except that it must evaluate to a single-column Boolean result. Only those elements that evaluate to `TRUE` are selected.

```
"$0.vol > $1.vol and $0.close > ($0.high - $0.low)/2"
```

Apply with the *multiset_ts* argument assigns parameter numbers by fetching **TimeSeries** values from the set and processing them in the order in which they are returned by the set management code. Since sets are unordered, parameters might not be assigned numbers predictably. **Apply** with the *multiset_ts* argument is useful only if you can guarantee that the **TimeSeries** values are returned in a fixed order. There are two ways to guarantee this:

- Write a C function that creates the set and use the function as the *multiset_ts* argument to **Apply**. The C function can return the **TimeSeries** values in any order you want.
- Use ORDER BY in the *multiset_ts* expression

Apply with the *multiset_ts* argument evaluates the expression once for every timepoint in the resulting union of time series values. When all the data in the clipped period has been exhausted, **Apply** returns the resulting series.

Apply uses the optional clip time range to restrict the data to a particular time period. If the beginning timepoint is `NULL`, then **Apply** uses the earliest valid timepoint of all the input time series. If the ending timepoint is `NULL`, then **Apply** uses the latest valid timepoint of all the input time series. When the optional clip time range is not used, it is equivalent to both the beginning and ending timepoints being `NULL`: **Apply** considers all elements.

If both the clip time range and filter expression are given, then clipping is done before filtering.

If you use a string literal or `NULL` for the clip time range, you should cast to DATETIME YEAR TO FRACTION(5) on at least the beginning timepoint to avoid ambiguity in function resolution.

When more than one input time series is specified, a union of all input time series is performed to produce the source of data to be filtered and evaluated by **Apply**. Hence, **Apply** acts as a union function, with extra filtering and manipulation of union results. For details on how the **Union** function works, see "Union function" on page 7-111.

Returns

A new time series with the results of evaluating the expression on every selected element from the source time series.

Example

The following example uses **Apply** without a filter argument and without a clipped range:

```
select Apply('$high-$low',
      datetime(2011-01-01) year to day,
      datetime(2011-01-06) year to day,
      stock_data)::TimeSeries(one_real)
   from daily_stocks
   where stock_name = 'IBM';
```

The following example shows **Apply** without a filter and with a clipped range:

```
select Apply(
   '($0.high+$1.high)/2, ($0.low+$1.low)/2, ($0.final+$1.final)/2,
($0.vol+$1.vol)/2',
   datetime(2011-01-04) year to day,
   datetime(2011-01-05) year to day,
   t1.stock_data, t2.stock_data)
   ::TimeSeries(stock_bar)
from daily_stocks t1, daily_stocks t2
where t1.stock_name = 'IBM' and t2.stock_name = 'HWP';
```

The following example shows **Apply** with a filter and without a clip range. The resulting time series contains the closing price of the days that the trading range is more than 10% of the low:

```
create function ts_sum(a stock_bar)
    returns one_real;
    return row(null::datetime year to fraction(5),
   (a.high + a.low + a.final + a.vol))::one_real;
end function;

select Apply('ts_sum',
   '2011-01-03 00:00:00.00000'::datetime year
      to fraction(5),
   '2011-01-03 00:00:00.00000'::datetime year
      to fraction(5),
   stock_data)::TimeSeries(one_real)
   from daily_stocks
      where stock_id = 901;
```

The following example uses a function as the expression to evaluate to boost performance. The first step is to compile the following C function into **applyfunc.so**:

```
/* begin applyfunc.c */
#include "mi.h"
MI_ROW *
high_low_diff(MI_ROW *row, MI_FPARAM *fp)
{
    MI_ROW_DESC            *rowdesc;
    MI_ROW            *result;
    void            *values[2];
    mi_boolean            nulls[2];
    mi_real            *high, *low;
    mi_real            r;
    mi_integer            len;
    MI_CONNECTION            *conn;
    mi_integer            rc;

    nulls[0] = MI_TRUE;
    nulls[1] = MI_FALSE;
    conn = mi_open(NULL,NULL,NULL);
    if ((rc = mi_value(row, 1, (MI_DATUM *) &high,
      &len)) == MI_ERROR)
```

```
   mi_db_error_raise(conn, MI_EXCEPTION,
      "ts_test_float_sql: corrupted argument row");
    if (rc == MI_NULL_VALUE)
   goto retisnull;

    if ((rc = mi_value(row, 2, (MI_DATUM *) &low,
      &len)) == MI_ERROR)
   mi_db_error_raise(conn, MI_EXCEPTION,
      "ts_test_float_sql: corrupted argument row");
    if (rc == MI_NULL_VALUE)
   goto retisnull;

    r = *high - *low;
    values[1] = (void *) &r;
    rowdesc = mi_row_desc_create(mi_typestring_to_id(conn,
      "one_real"));
    result = mi_row_create(conn, rowdesc, (MI_DATUM *)
      values, nulls);
    mi_close(conn);
    return (result);
 retisnull:
    mi_fp_setreturnisnull(fp, 0, MI_TRUE);
    return (MI_ROW *) NULL;
}
/* end of applyfunc.c */
```

Then create the following SQL function:

```
create function HighLowDiff(arg stock_bar) returns one_real
external name '/tmp/applyfunc.bld(high_low_diff)'
language C;


select stock_name, Apply('HighLowDiff',
        stock_data)::TimeSeries(one_real)
from daily_stocks;
```

The following query is equivalent to the previous query, but it does not have the performance advantages of using a function as the expression to evaluate:

```
select stock_name, Apply('$high - $low',
   stock_data)::TimeSeries(one_real)
from daily_stocks;
```

Related reference:
"AggregateBy function" on page 7-6
"AggregateRange function" on page 7-9
"Clip function" on page 7-26
"ClipCount function" on page 7-29
"ClipGetCount function" on page 7-30
"Intersect function" on page 7-59
"TSAddPrevious function" on page 7-80
"TSCmp function" on page 7-81
"TSDecay function" on page 7-96
"TSPrevious function" on page 7-97
"TSRunningAvg function" on page 7-102
"TSRunningSum function" on page 7-105
"Union function" on page 7-111
"Binary arithmetic functions" on page 7-21
"SetOrigin function" on page 7-74
"TSRunningCor function" on page 7-103
"TSRunningMed function" on page 7-104
"TSRunningVar function" on page 7-106
"Unary arithmetic functions" on page 7-110

ApplyBinaryTsOp function

The **ApplyBinaryTsOp** function applies a binary arithmetic function to a pair of time series or to a time series and a compatible row type or number.

Syntax

```
ApplyBinaryTsOp(func_name  lvarchar,
              ts         TimeSeries,
              ts         TimeSeries)
returns TimeSeries;

ApplyBinaryTsOp(func_name     lvarchar,
               number_or_row scalar|row,
               ts            TimeSeries)
returns TimeSeries;

ApplyBinaryTsOp(func_name     lvarchar,
               ts            TimeSeries,
               number_or_row scalar|row)
returns TimeSeries;
```

func_name
The name of a binary arithmetic function.

ts
The time series to use in the operation. The second and third arguments can be a time series, a row type, or a number. At least one of the two must be a time series.

number_or_row
A number or a row type to use in the operation. The second and third arguments can be a time series, a row type, or a number. The second two arguments must be compatible under the function. See "Binary arithmetic functions" on page 7-21 for a description of the compatibility requirements.

Description

These functions operate in an analogous fashion to the arithmetic functions that have been overloaded to operate on time series. See the description of these functions in "Binary arithmetic functions" on page 7-21 for more information. For example, **Plus(ts1, ts2)** is equivalent to **ApplyBinaryTsOp('Plus', ts1, ts2)**.

Returns

A time series of the same type as the first time series argument, which can result in a loss of precision. The return type can be explicitly cast to a compatible time series type with more precision to avoid this problem. See "Binary arithmetic functions" on page 7-21 for more information.

Example

The following example uses **ApplyBinaryTSOp** to implement the **Plus** function:

```
create row type simple_series( stock_id int, data TimeSeries(one_real));
create table daily_high of type simple_series;
insert into daily_high
   select stock_id,
      Apply( '$0.high',
         NULL::datetime year to fraction(5),
         NULL::datetime year to fraction(5),
         stock_data)::TimeSeries(one_real)
      from daily_stocks;
create table daily_low of type simple_series;
insert into daily_low
   select stock_id,
      Apply( '$0.low',
         NULL::datetime year to fraction(5),
         NULL::datetime year to fraction(5),
         stock_data)::TimeSeries(one_real)
      from daily_stocks;
create table daily_avg of type simple_series;
insert into daily_avg
   select l.stock_id, ApplyBinaryTSOp("plus", l.data, h.data)/2
      from daily_low l, daily_high h
      where l.stock_id = h.stock_id;
```

You can receive the same results by substituting **(l.data + h.data)** for **ApplyBinaryTSOp('plus', 1.data, h.data)**.

Related reference:

"ApplyOpToTsSet function" on page 7-19

"Binary arithmetic functions" on page 7-21

ApplyCalendar function

The **ApplyCalendar** function applies a new calendar to a time series.

Syntax

```
ApplyCalendar (ts       TimeSeries,
             cal_name lvarchar,
             flags    integer default 0)
returns TimeSeries;
```

ts The given time series from which specific timepoints will be projected.

cal_name
The name of the calendar to apply.

flags Valid values for the *flags* argument are described later in this topic.

Description

If the calendar specified by the argument has an interval smaller than the calendar attached to the original time series, and the original time series is regular, then the resulting time series has a higher frequency and can therefore have more elements than the original time series. For example, applying an hourly calendar with eight valid timepoints per day to a daily time series converts each daily entry in the new time series into eight hourly entries.

The flags argument values

When opening the source time series, your setting of the *flags* argument is combined (using the AND operator) with the TSOPEN_READ_HIDDEN value. The returned time series is opened with your setting of the *flags* argument combined (using the AND operator) with TSOPEN_WRITE_AND_HIDE, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_HIDDEN.

The value of *flags* is the sum of the desired flag values from the following table.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Indicates that the time series can be read and written to.
TSOPEN_READ_HIDDEN	1	Indicates that hidden elements should be treated as if they are not hidden.
TSOPEN_WRITE_HIDDEN	2	Allows hidden elements to be written to without first revealing them. The element remains hidden afterward.
TSOPEN_WRITE_AND_HIDE	4	Causes any elements written to a time series also to be marked as hidden.
TSWRITE_AND_REVEAL	8	Reveals any hidden element that is written to.
TSOPEN_NO_NULLS	32	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as NULL. If TSOPEN_NO_NULLS is set, an element that has each column set to NULL is returned.

These flags can be used in any combination except the following four:
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_REVEAL
- TSOPEN_WRITE_AND_REVEAL and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_HIDE, and TSOPEN_WRITE_AND_REVEAL

The TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_AND_HIDE flags cannot be used with TSOPEN_READ_HIDDEN.

Returns

A new time series that uses the named calendar and includes entries from the original time series on active timepoints in the new calendar.

Example

Assuming **fourdaycal** is a calendar that contains four-day workweeks, the following query returns a time series of a given stock's data for each of the four working days:

```
select ApplyCalendar(stock_data,'fourdaycal')
   from daily_stocks
   where stock_name = 'IBM';
```

ApplyOpToTsSet function

The **ApplyOpToTsSet** function applies a binary arithmetic function to a set of time series.

Syntax

```
ApplyOpToTsSet(func_name   lvarchar,
             multiset_ts multiset(TimeSeries))
returns TimeSeries;
```

func_name
: The name of a binary function. See "Binary arithmetic functions" on page 7-21 for more information.

multiset_ts
: A multiset of time series that are compatible with the function. All the time series in the multiset must have the same type.

Description

All the time series must have the same type. If the multiset is empty, then **ApplyOpToTsSet** returns NULL. If the multiset contains only one time series, then **ApplyOpToTsSet** returns a copy of that time series. If the multiset contains exactly two time series, **ts1** and **ts2**, then **ApplyOpToTsSet** returns **ApplyBinaryTsOp(func_name, ts1, ts2)**. If the multiset contains three time series, **ts1**, **ts2**, and **ts3**, then **ApplyOpToTsSet** returns **ApplyBinaryTsOp(func_name, ApplyBinaryTsOp(func_name, ts1, ts2), ts3)**, and so on.

Returns

A time series of the same type as the time series in the multiset. The calendar of the resulting time series is the union of the calendars of the input time series. The resulting time series is regular if all the input times series are regular and irregular if any of the inputs are irregular.

Related reference:

"ApplyBinaryTsOp function" on page 7-16

"Binary arithmetic functions" on page 7-21

ApplyUnaryTsOp function

The **ApplyUnaryTsOp** function applies a unary arithmetic function to a time series.

Syntax

```
ApplyUnaryTsOp(func_name lvarchar,
               ts        TimeSeries)
returns TimeSeries;
```

func_name
The name of the unary arithmetic function.

ts The time series to act on.

Description

This function operates in an analogous fashion to the unary arithmetic functions that have been overloaded to operate on time series. See the description of these functions in the section "Unary arithmetic functions" on page 7-110 for more information. For example, **Logn(ts1)** is equivalent to **ApplyUnaryTsOp('Logn', ts1)**.

Returns

A time series of the same type as the supplied time series.

Example

The following example uses **ApplyUnaryTSOp** with the **Logn** function:

```
create row type simple_series( stock_id int, data TimeSeries(one_real));
create table daily_high of type simple_series;
insert into daily_high
   select stock_id,
      Apply( '$0.high',
         NULL::datetime year to fraction(5),
         NULL::datetime year to fraction(5),
         stock_data)::TimeSeries(one_real)
      from daily_stocks;
create table daily_low of type simple_series;
insert into daily_low
   select stock_id,
      Apply( '$0.low',
         NULL::datetime year to fraction(5),
         NULL::datetime year to fraction(5),
         stock_data)::TimeSeries(one_real)
      from daily_stocks;
create table daily_avg of type simple_series;
insert into daily_avg
   select l.stock_id, ApplyBinaryTSOp("plus", l.data, h.data)/2
      from daily_low l, daily_high h
      where l.stock_id = h.stock_id;
create table log_high of type simple_series;
insert into log_high
   select stock_id, ApplyUnaryTsOp( "logn",
   data) from daily_avg;
```

Related reference:

"Unary arithmetic functions" on page 7-110

Asin function

The **Asin** function returns the arc sine of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Atan function

The **Atan** function returns the arc tangent of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Atan2 function

The **Atan2** function returns the arc tangent of corresponding elements from two time series.

It is one of the binary arithmetic functions that work on time series. The others are **Divide**, **Minus**, **Mod**, **Plus**, **Pow**, and **Times**.

Related reference:

"Binary arithmetic functions"

Binary arithmetic functions

The standard binary arithmetic functions **Atan2**, **Plus**, **Minus**, **Times**, **Divide**, **Mod**, and **Pow** are extended to *operate* on time series. The normal operator aliasing applies; the **Plus**, **Minus**, **Times**, and **Divide** functions can also be denoted by the infix operators "+", "-", "*", and "/", respectively.

Syntax

```
Function(ts TimeSeries,
        ts TimeSeries)
returns TimeSeries;

Function(number_or_row  scalar|row,
        ts            TimeSeries)
returns TimeSeries;

Function(ts            TimeSeries,
        number_or_row  scalar|row)
returns TimeSeries;
```

ts
: The source time series. One of the two arguments must be a time series for this variant of the functions. The two inputs must be compatible under the function.

number_or_row
: A scalar number or a row type. The two inputs must be compatible under the function.

Description

In the first format, both arguments are time series. The result is a time series that starts at the later of the starting times of the inputs. The end point of the result is the later of the two input end points if both inputs are irregular. The result end point is the earlier of the input regular time series end points if one or more of the inputs is a regular time series. The result time series has one time point for each input time point in the interval.

The element at time *t* in the resulting time series is formed from the last elements at or before time *t* in the two input time series. Normally the function is applied column by column to the input columns, except for the time stamp, to produce the output element. In this case, the two input row types must have the same number of columns, and the corresponding columns must be compatible under the function.

However, if there is a variant of the function that operates directly on the row types of the two input time series, then that variant is used. Then the input row types can have different numbers of columns and the columns might be incompatible. The time stamp of the resulting element is ignored; the element placed in the resulting time series has the later of the time stamps of the input elements.

The resulting calendar is the union of the calendars of the input time series. If the input calendars are the same, then the resulting calendar is the same as the input calendar. Otherwise, a new calendar is made. The name of the resulting calendar is a string containing the names of the calendars of the input time series, separated by a vertical line (|). For example, if two time series are joined, and **mycal** and **yourcal** are the names of their corresponding calendars, the resulting calendar is named **mycal|yourcal**.

The resulting time series is regular if both the input time series are regular and irregular if either of the inputs is irregular.

One of the inputs can be a scalar number or a row type. In this case, the resulting time series has the same calendar, sequence of time stamps, and regularity as the input time series. If one of the inputs is a scalar number, then the function is applied to the scalar number and to each non-time stamp column of each element of the input time series.

If an input is a row type, then that row type must be compatible with the time series row type. The function is applied to the input row type and each element of the input time series. It is applied column by column or directly to the two row types, depending on whether there is a variant of the function that handles the row types directly.

Returns

The same type of time series as the first time series input, unless they are cast. If a function is cast, then it returns the type of time series to which it is cast.

For example, suppose that time series **tsi** has type **TimeSeries(ci)**, and that time series **tsr** has type **TimeSeries(cr)**, where **ci** is a row type with INTEGER columns and **cr** is a row type with SMALLFLOAT columns. Then **Plus(tsi, tsr)** has type **TimeSeries(ci)**; the fractional parts of the resulting numbers are discarded. This is generally not the desired effect. **Plus(tsi, tsr)::TimeSeries(cr)** has type **TimeSeries(cr)** and does not discard the fractional parts of the resulting numbers.

Example

The following query produces time series of daily average stock prices (actually, the average of the daily high and low). For convenience, the example starts by projecting the highs and lows into separate time series:

```
create row type price( timestamp datetime year to fraction(5),
   val real);
create row type simple_series( stock_id int, data
   TimeSeries(price));
create table daily_high of type simple_series;

insert into daily_high
    select stock_id,
      Apply('$high',
       '2011-01-03 00:00:00.00000'
       ::datetime year to fraction(5),
       '2011-01-10 00:00:00.00000'
       ::datetime year to fraction(5),
       stock_data)::TimeSeries(price)
   from daily_stocks;

create table daily_low of type simple_series;

insert into daily_low
    select stock_id,
      Apply('$low',
       '2011-01-03 00:00:00.00000'
       ::datetime year to fraction(5),
       '2011-01-10 00:00:00.00000'
       ::datetime year to fraction(5),
       stock_data)::TimeSeries(price)
   from daily_stocks;

create table daily_avg of type simple_series;

insert into daily_avg
    select l.stock_id, (l.data + h.data)/2
        from daily_low l, daily_high h
        where l.stock_id = h.stock_id;
```

Related reference:

"ApplyBinaryTsOp function" on page 7-16
"ApplyOpToTsSet function" on page 7-19
"Atan2 function" on page 7-21
"Apply function" on page 7-11
"Unary arithmetic functions" on page 7-110
"Divide function" on page 7-36
"Minus function" on page 7-63
"Mod function" on page 7-63
"Plus function" on page 7-65
"Pow function" on page 7-65
"Times function" on page 7-76

BulkLoad function

The **BulkLoad** function loads data from a client file into an existing time series.

Syntax

```
BulkLoad (ts       TimeSeries,
          filename lvarchar,
          flags    integer default 0)
returns TimeSeries;
```

ts The time series in which to load data.

filename
The path and file name of the file to load.

flags Valid values for the *flags* parameter are described later in this topic.

Description

The file is located on the client and can be an absolute or relative path name.

Two data formats are supported for the file loaded by **BulkLoad**:
- Using type constructors
- Using tabs

Each line of the client file must have all the data for one element.

The type constructor format follows the row type convention: comma-separated columns surrounded by parentheses and preceded by the ROW type constructor. The first two lines of a typical file look like this:

```
row(2011-01-03 00:00:00.00000, 1.1, 2.2)
row(2011-01-04 00:00:00.00000, 10.1, 20.2)
```

If you include collections in a column within the row data type, use a type constructor (SET, MULTISET, or LIST) and curly braces surrounding the collection values. A row including a set of rows has this format:

```
row(timestamp, set{row(value, value), row(value, value)}, value)
```

The tab format is to separate the values by tabs. It is only recommended for single-level rows that do not contain collections or row data types. The first two lines of a typical file in this format look like this:

```
2011-01-03 00:00:00.00000   1.1     2.2
2011-01-04 00:00:00.00000   10.1    20.2
```

The spaces between entries represent a tab.

In both formats, the word `NULL` indicates a null entry.

When **BulkLoad** encounters data with duplicate time stamps in a regular time series, the old values are replaced by the new values. In an irregular time series, when **BulkLoad** encounters data with duplicate time stamps, the following algorithm is used to determine where to place the data belonging to the duplicate time stamp:
1. Round the time stamp up to the next second.
2. Search backwards for the first element less than the new time stamp.
3. Insert the new data at this time stamp plus 10 microseconds.

This is the same algorithm as used by the **PutElem** function, described in "PutElem function" on page 7-65.

The flags argument values

The value of the *flags* argument is the sum of the desired flag values from the following table.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Indicates that the time series can be read and written to.
TSOPEN_READ_HIDDEN	1	Indicates that hidden elements should be treated as if they are not hidden.
TSOPEN_WRITE_HIDDEN	2	Allows hidden elements to be written to without first revealing them. The element remains hidden afterward.
TSOPEN_WRITE_AND_HIDE	4	Causes any elements written to a time series also to be marked as hidden.
TSWRITE_AND_REVEAL	8	Reveals any hidden element written to.
TSOPEN_NO_NULLS	32	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as `NULL`. If TSOPEN_NO_NULLS is set, instead an element is returned that has each column set to `NULL`.
TS_PUTELEM_NO_DUPS	64	Determines whether the **BulkLoad** function adds elements using the **PutElem** function (default) or the **PutElemNodups** function (see "PutElem function" on page 7-65 and "PutElemNoDups function" on page 7-67). If this flag is set, the **BulkLoad** function uses **PutElemNoDups**.

These flags can be used in any combination except the following four:

- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_REVEAL
- TSOPEN_WRITE_AND_REVEAL and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN, and TSOPEN_WRITE_AND_HIDE, and TSOPEN_WRITE_AND_REVEAL

The TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_AND_HIDE flags cannot be used with TSOPEN_READ_HIDDEN.

Returns

A time series containing the new data.

Example

The following example adds data from the `sam.dat` file to the **stock_data** time series:

```
update daily_stocks
set stock_data = BulkLoad(stock_data, 'sam.dat')
   where stock_name = 'IBM';
```

Clip function

The **Clip** function extracts data between two timepoints in a time series and returns a new time series containing that data. This allows you to extract periods of interest from a large time series and to store or operate on them separately from the large series.

Syntax

```
Clip(ts          TimeSeries,
    begin_stamp  datetime year to fraction(5),
    end_stamp    datetime year to fraction(5),
     flags       integer default 0)
returns TimeSeries;

Clip(ts          TimeSeries,
    begin_stamp  datetime year to fraction(5),
    end_offset   integer,
     flags       integer default 0)
returns TimeSeries;

Clip(ts          TimeSeries,
    begin_offset integer,
    end_stamp    datetime year to fraction(5),
     flags       integer default 0)
returns TimeSeries;

Clip(ts          TimeSeries,
    begin_offset integer,
    end_offset   integer,
     flags       integer default 0)
returns TimeSeries;
```

ts The time series to clip.

begin_stamp
: The begin point of the range. Can be NULL.

end_stamp
: The end point of the range. Can be NULL.

begin_offset
: The begin offset of the range (regular time series only).

end_offset
: The end offset of the range (regular time series only).

flags **(optional)**
: A flag specifying how to determine the resulting time series origin and whether to copy hidden elements to the resulting time series. See "The flags argument values" on page 7-27.

Description

The **Clip** functions all take a time series, a begin point, and an end point for the range.

For regular time series, the begin and end points can be either integers or time stamps. If the begin point is an integer, it is the absolute offset of an entry in the time series. If it is a time stamp, the **Clip** function uses the time series' calendar to find the offset that corresponds to the time stamp. If there is no entry in the time series exactly at the requested time stamp, **Clip** uses the calendar's time stamp that immediately follows the given time stamp as the begin point of the range.

The end point is used in the same way as the begin point, except that it specifies the end of the range, rather than its beginning. The begin and end points can be `NULL`, in which case the beginning or end of the time series is used.

For irregular time series, only time stamps are permitted for the begin and end points.

Data at the beginning and ending offsets is included in the resulting time series.

The flags argument values

The *flags* argument is an optional parameter that determines:
- The resulting time series origin
- Whether to copy hidden elements to the resulting time series and if the elements should be hidden or revealed in the new time series

If the *flags* argument is not set (has the default value of 0), then the origin of the resulting time series is the later of the *begin_offset* argument and the origin of the input time series. If the flag is set to 1, then the origin of the resulting time series is set to the earlier of the *begin_offset* argument and the origin of the input time series. In this case, timepoints before the origin of the time series are set to `NULL`.

When you clip a range that contains hidden elements, by default, hidden elements are not copied to the resulting time series. The *flags* argument allows you to include hidden elements in the result. You can also mark the elements as hidden or revealed in the resulting time series.

To extract data into a new time series, include hidden elements in that time range, and keep the elements hidden in the resulting time series, set the *flags* argument to `2`:

```
Clip(ts, begin, end, 2)
```

In this example, *ts* is the time series you are clipping, and *begin* and *end* are the timepoints marking the range to extract.

To extract data into a new time series, include hidden elements in that time range, and reveal the hidden elements in the resulting time series, set the *flags* argument to `6`:

```
Clip(ts, begin, end, 6)
```

In this example, *ts* is the time series you are clipping, and *begin* and *end* are the timepoints marking the range to extract.

You can use the *flags* argument for handling hidden elements in conjunction with the option for determining the origin of the resulting time series. For example, if you set the *flags* argument to `3`, the **Clip** function includes hidden elements in the resulting time series, the elements are marked as hidden, and the origin of the resulting time series is the earlier of the *begin_offset* you specify for the **Clip** function and the input time series origin:

```
Clip(ts, begin, end, 3)
```

Returns

A new time series containing only data from the requested range. The new series has the same calendar as the original, but it can have a different origin and number of entries.

Example

The results of the **Clip** function are slightly different for regular and irregular time series.

Example 1: Regular time series

The following query extracts data from a time series and creates a table containing a given stock's data for a single week:

```
create table week_1_analysis (stock_id int, stock_data
   TimeSeries(stock_bar));
insert into week_1_analysis
    select stock_id,
   Clip(stock_data,
      '2011-01-03 00:00:00.00000'
       ::datetime year to fraction(5),
      '2011-01-07 00:00:00.00000'
       ::datetime year to fraction(5))
from daily_stocks
where stock_name = 'IBM';
```

The following query displays the first six entries for a given stock in a time series:

```
select Clip(stock_data, 0, 5)
from daily_stocks
where stock_name = 'IBM';
```

Example 2: Irregular time series

An irregular time series has the following values:

```
2005-12-17 10:23:00.00000  26.46
2006-01-03 13:19:00.00000  27.30
2006-01-04 13:19:00.00000  28.67
2006-01-09 13:19:00.00000  30.56
```

The following statement extracts data from a time series over a five day period:

```
EXECUTE FUNCTION Transpose ((
  select Clip(
      tsdata,
      "2006-01-01 00:00:00.00000"::datetime year to fraction (5),
      "2006-01-05 00:00:00.00000"::datetime year to fraction (5),
      0)
  from ts_tab
  where station_id = 228820)) ;
```

The resulting irregular time series is this:

```
2006-01-01 00:00:00.00000  26.46
2006-01-03 13:19:00.00000  27.30
2006-01-04 13:19:00.00000  28.67
```

The first value has the beginning date of the clip and the value of the first original value. Because the time series is irregular, a value persists until the next element. Therefore, the value of `26.46` is still valid on 2006-01-01.

Related reference:
“Apply function” on page 7-11
“ClipCount function”
“ClipGetCount function” on page 7-30
“GetElem function” on page 7-40
“GetLastValid function” on page 7-46
“GetNthElem function” on page 7-50
“WithinC and WithinR functions” on page 7-116
“DelClip function” on page 7-32
“DelTrim function” on page 7-35
“SetOrigin function” on page 7-74

ClipCount function

The **ClipCount** function is a variation of **Clip** in which the first integer argument is interpreted as a count of entries to clip. If the count is positive, **ClipCount** begins with the first element at or after the time stamp and clips the next count entries. If the count is negative, **ClipCount** begins with the first element at or before the time stamp and clips the previous count entries.

Syntax

```
ClipCount(ts          TimeSeries,
         begin_stamp datetime year to fraction(5),
         num_stamps  integer,
         flags       integer default 0)
returns TimeSeries;
```

ts The time series to clip.

begin_stamp
The begin point of the range. Can be NULL.

num_stamps
The number of elements to be included in the resultant time series.

flags **(optional)**
A flag specifying how to determine the resulting time series origin and whether to copy hidden elements to the resulting time series. See “The flags argument values.”

Description

Begin points before the time series origin are permitted. Negative counts with such time stamps result in time series with no elements. Begin points before the calendar origin are not permitted.

If there is no entry in the calendar exactly at the requested time stamp, **ClipCount** uses the calendar's first valid time stamp that immediately follows the given time stamp as the begin point of the range. If the begin point is NULL, the origin of the time series is used.

The flags argument values

The *flags* argument is an optional parameter that works in the same way as the *flags* argument of the **Clip** function. See “The flags argument values” on page 7-27 for a description.

Returns

A new time series containing only data from the requested range. The new series has the same calendar as the original, but it can have a different origin and number of entries.

Example

The following example clips the first 30 elements at or after March 14, 2011, at 9:30 a.m. for the stock with ID 600, and it returns the entire resulting time series:

```
select ClipCount(activity_data,
     '2011-01-01 09:30:00.00000', 30)
    from activity_stocks
    where stock_id = 600;
```

The following example clips the previous 60 elements at or before August 22, 2011, at 12:00 midnight for the stock with ID 600:

```
select ClipCount(activity_data,
       '2011-08-22 00:00:00.00000', -60)
   from activity_stocks
   where stock_id = 600;
```

Related reference:

"Apply function" on page 7-11

"Clip function" on page 7-26

"ClipCount function" on page 7-29

"ClipGetCount function"

"GetElem function" on page 7-40

"GetLastValid function" on page 7-46

"GetNthElem function" on page 7-50

ClipGetCount function

The **ClipGetCount** function returns the number of elements in the current time series that occur in the period delimited by the time stamps.

Syntax

```
ClipGetCount(ts TimeSeries,
       begin_stamp datetime year to fraction(5) default NULL,
       end_stamp   datetime year to fraction(5) default NULL,
        flags       integer default 0)
returns integer;
```

ts The source time series.

begin_stamp
: The begin point of the range. Can be NULL.

end_stamp
: The end point of the range. Can be NULL.

flags Valid values for the *flags* argument are described later in this topic.

Description

For an irregular time series, deleted elements are not counted. For a regular time series, only entries that are non-null are counted, so **ClipGetCount** might return a different value than **GetNelems**.

If the begin point is NULL, the time series origin is used. If the end point is NULL, the end of the time series is used.

See "Clip function" on page 7-26 for more information about the begin and end points of the range.

The flags argument values

The *flags* argument determines how a scan should work on the returned set. If you set the *flags* argument to 0 (the default), null and hidden elements are not part of the count. If the *flags* argument has a value of 512 (0x200) (the TS_SCAN_HIDDEN bit is set), all non-null elements are counted whether they are hidden or not.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Hidden elements are not included in the count.
TS_SCAN_HIDDEN	512	Hidden elements marked by **HideElem** are included in the count (see "HideElem function" on page 7-55).

Returns

The number of elements in the given time series that occur in the period delimited by the time stamps.

Example

The following statement returns the number of elements between 10:30 a.m. on March 14, 2011, and midnight on March 19, 2011, inclusive:

```
select ClipGetCount(activity_data,
   '2011-03-14 10:30:00.00000','2011-03-19 00:00:00.00000')
    from activity_stocks
    where stock_id = 600;
```

Related reference:

"Apply function" on page 7-11

"Clip function" on page 7-26

"ClipCount function" on page 7-29

"GetIndex function" on page 7-43

"GetNelems function" on page 7-48

"GetNthElem function" on page 7-50

"GetStamp function" on page 7-54

"The ts_nelems() function" on page 9-38

Cos function

The **Cos** function returns the cosine of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

DelClip function

The **DelClip** function deletes all elements in the specified time range, including the delimiting timepoints. The **DelClip** function differs from the **DelTrim** function in its handling of deletions from the end of a regular time series. **DelTrim** shortens the time series and reclaims space, whereas **DelClip** replaces elements with null values.

Syntax

```
DelClip(ts          TimeSeries,
       begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5)
       flags       integer default 0
)
returns TimeSeries;
```

ts The time series to act on.

begin_stamp
The begin point of the range.

end_stamp
The end point of the range.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default value is 0.

Description

You can use **DelClip** to delete hidden elements.

If the begin or end point of the range falls before the origin of the time series or after the last element in the time series, an error is raised.

When **DelClip** operates on a regular time series, it replaces elements with null elements; it never changes the number of elements in a regular time series.

Returns

A time series with all elements in the range between the specified timepoints deleted.

Example

The following example removes all elements on a given day:

```
update activity_stocks
set activity_data = DelClip(activity_data,
        '2011-01-05 00:00:00.00000'
     ::datetime year to fraction(5),
        '2011-01-06 00:00:00.00000'
     ::datetime year to fraction(5))
where stock_id = 600;
```

Related reference:
“Clip function” on page 7-26
“DelElem function”
“DelTrim function” on page 7-35
“HideElem function” on page 7-55
“InsSet function” on page 7-58
“PutSet function” on page 7-69
“UpdSet function” on page 7-115

DelElem function

The **DelElem** function deletes the element at a given timepoint.

Syntax

```
DelElem(ts     TimeSeries,
       tstamp datetime year to fraction(5),
       flags  integer default 0)
returns TimeSeries;
```

ts The time series to act on.

tstamp The time stamp of the element to be deleted.

flags Valid values for the *flags* parameter are described in “The *flags* argument values” on page 7-5. The default is 0.

Description

If there is no element at the specified timepoint, no elements are deleted and no error is raised.

The API equivalent of **DelElem** is **ts_del_elem()**.

Hidden time stamps cannot be deleted.

Returns

A time series with one element deleted.

Example

The following example deletes an element from a time series:

```
update activity_stocks
set activity_data = DelElem(activity_data,
   '2011-01-05 12:58:09.23456'
   ::datetime year to fraction(5))
where stock_id = 600;
```

Related reference:
"DelClip function" on page 7-32
"DelTrim function" on page 7-35
"GetElem function" on page 7-40
"HideElem function" on page 7-55
"InsElem function" on page 7-57
"PutElem function" on page 7-65
"The ts_del_elem() function" on page 9-20
"UpdElem function" on page 7-113
"The ts_elem() function" on page 9-20

DelRange function

The **DelRange** function deletes all elements in the specified time range, including the delimiting timepoints. The **DelRange** function is similar to the **DelTrim** function except that the **DelRange** function deletes elements and reclaims space from any part of a regular time series.

Syntax

```
DelRange(ts          TimeSeries,
       begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5),
       flags       integer default 0)
returns TimeSeries;
```

ts The time series to act on.

begin_stamp
The begin point of the range.

end_stamp
The end point of the range.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

Use the **DelRange** function to delete elements in a time series from a specified time range and free any resulting empty pages. For example, you can remove data from the beginning of a time series to archive the data.

If you use the **DelRange** function to delete hidden elements, or if the begin point of the range falls before the origin of the time series, an error is raised.

Returns

A time series with all elements in the range between the specified timepoints deleted.

Example

The following example removes all elements in a one-day range on a given day:

```
UPDATE ts_data
SET meter_data = DelRange(meter_data,
        '2010-11-11 00:00:00.00000'
```

```
     ::datetime year to fraction(5),
        '2010-11-11 00:00:00.00000'
     ::datetime year to fraction(5))
WHERE loc_esi_id = 4727354321000111;
```

DelTrim function

The **DelTrim** function deletes all elements in the specified time range, including the delimiting timepoints. The **DelTrim** function is similar to the **DelClip** function except that the **DelTrim** function deletes elements and reclaims space from the end of a regular time series, whereas the **DelClip** function replaces elements with null values. The **DelTrim** function is also similar to the **DelRange** function except that the **DelRange** function deletes elements and reclaims space from any part of a regular time series.

Syntax

```
DelTrim(ts          TimeSeries,
       begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5),
       flags       integer default 0)
returns TimeSeries;
```

ts The time series to act on.

begin_stamp
The begin point of the range.

end_stamp
The end point of the range.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

If you use the **DelTrim** function to delete elements from the end of a time series, **DelTrim** trims off all null elements from the end of the time series and thus reduces the number of elements in the time series.

If you use the **DelTrim** function to delete hidden elements, or if the begin point of the range falls before the origin of the time series, an error is raised.

Returns

A time series with all elements in the range between the specified timepoints deleted.

Example

The following example removes all elements in a one-day range on a given day:

```
update activity_stocks
set activity_data = DelTrim(activity_data,
        '2011-01-05 00:00:00.00000'
     ::datetime year to fraction(5),
        '2011-01-06 00:00:00.00000'
     ::datetime year to fraction(5))
where stock_id = 600;
```

Related reference:
"DelClip function" on page 7-32
"DelElem function" on page 7-33
"Clip function" on page 7-26
"HideElem function" on page 7-55
"InsSet function" on page 7-58
"PutSet function" on page 7-69
"UpdSet function" on page 7-115

Divide function

The **Divide** function divides one time series by another.

It is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Minus**, **Mod**, **Plus**, **Pow**, and **Times**.

Related reference:
"Binary arithmetic functions" on page 7-21

ElemIsHidden function

The **ElemIsHidden** function determines if an element is hidden.

Syntax

```
ElemIsHidden(ts     TimeSeries,
             offset integer)
returns Boolean;

ElemIsHidden(ts     TimeSeries,
             tstamp datetime year to fraction(5))
returns Boolean;
```

ts The time series to act on.

offset The offset of the element to examine.

tstamp The time stamp of the element to examine.

Description

Use either offset or time stamp to locate the element you want to examine.

Returns

Returns TRUE if the element is hidden and FALSE if it is not.

Related reference:
"ElemIsNull function"
"FindHidden function" on page 7-37

ElemIsNull function

The **ElemIsNull** function determines if an element contains no data.

Syntax

```
ElemIsNull(ts     TimeSeries,
           offset integer)
returns Boolean;

ElemIsNull(ts     TimeSeries,
           tstamp datetime year to fraction(5))
returns Boolean;
```

ts The time series to act on.

offset The offset of the element to examine.

tstamp The time stamp of the element to examine.

Description

Use either offset or time stamp to locate the element you want to examine.

Returns

Returns TRUE if the element has never been written to or was written to and the data has since been deleted; returns FALSE if the element contains data or is hidden.

Related reference:

"ElemIsHidden function" on page 7-36

"FindHidden function"

Exp function

The **Exp** function exponentiates the time series.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

FindHidden function

The **FindHidden** function scans a time series and returns all elements that are hidden.

Syntax

```
FindHidden(ts TimeSeries,
      start datetime year to fraction(5) default NULL,
      end   datetime year to fraction(5) default NULL)
returns multiset;
```

ts The time series to act on.

start **(optional)**
The date from which to start the scan.

end **(optional)**
The date at which to end the scan.

Description

You can scan the whole time series or specify a start date and an end date for the scan.

Returns

A multiset containing all the hidden elements in the date range you specify.

Related reference:

"ElemIsHidden function" on page 7-36

"ElemIsNull function" on page 7-36

GetCalendar function

The **GetCalendar** function returns the calendar associated with the given time series.

Syntax

```
GetCalendar(ts TimeSeries)
returns Calendar;
```

ts The time series from which to obtain a calendar.

Returns

The calendar used by the time series.

Example

The following example returns the calendar used by the time series for IBM:

```
select GetCalendar(stock_data)
from daily_stocks
where stock_name = 'IBM';

(expression) startdate(2011-01-01 00:00:00),pattstart(2011-
         01-02 00:00:00),pattern({1 off,5 on,1 off},day)
```

Related reference:

"GetClosestElem function" on page 7-39

"GetInterval function" on page 7-43

"GetOrigin function" on page 7-52

"TSCreate function" on page 7-92

"GetCalendarName function"

"TSCreateIrr function" on page 7-94

GetCalendarName function

The **GetCalendarName** function returns the name of the calendar used by the given time series.

Syntax

```
GetCalendarName(ts TimeSeries)
returns lvarchar;
```

ts The time series from which to obtain a calendar name.

Returns

The name of the calendar used by the time series.

Example

The following example returns the name of the calendar used by the time series for IBM:

```
select GetCalendarName(stock_data)
from daily_stocks
where stock_name = 'IBM';
```

```
(expression) daycal
```

Related reference:

"GetCalendar function" on page 7-38

GetClosestElem function

The **GetClosestElem** function returns the first element that is non-null and closest to the given time stamp. Optionally, you can specify which column within the time series element must be non-null to satisfy the search.

Syntax

```
GetClosestElem(ts         TimeSeries,
              tstamp       datetime year to fraction(5),
              cmp          lvarchar,
              column_list  lvarchar default NULL,
              flags        integer default 0)
returns ROW
```

ts The time series to act on.

tstamp The time stamp to start searching from.

cmp A comparison operator used with *tstamp* to determine where to start the search. Valid values for *cmp* are <, <=, =, ==, >=, and >.

column_list
To search for an element with one or more columns non-null, specify a list of column names separated by a vertical bar (|). An error is raised if any of the column names does not exist in the time series type

To search for a null element, set *column_list* to `NULL`.

flags Determines whether hidden elements should be returned. Valid for the *flags* parameter values are defined in `tseries.h`. They are:
- TS_CLOSEST_NO_FLAGS (no special flags)
- TS_CLOSEST_RETNULLS_FLAGS (return hidden elements)

Description

The search algorithm **ts_closest_elem** is as follows:
- If *cmp* is any of : <=, =, ==, or >=, the search starts at *tstamp*.
- If *cmp* is <, the search starts at the first element before *tstamp*.
- If *cmp* is >, the search starts at the first element after *tstamp*.

The *tstamp* and *cmp* parameters are used to determine where to start the search. The search continues in the direction indicated by *cmp* until an element is found that qualifies. If no element qualifies, the return value is `NULL`.

Important: For irregular time series, values in an irregular element persist until the next element. This means that any of the "equals" operations on an irregular time series look for <= first. If *cmp* is >= and the <= operation fails, the operation then looks forward for the next element; otherwise, `NULL` is returned.

Returns

An element meeting the described criteria that is non-null and closest to the given time stamp.

Related reference:

"GetCalendar function" on page 7-38

"GetInterval function" on page 7-43

"GetOrigin function" on page 7-52

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

GetContainerName function

The **GetContainerName** function returns the name of the container for the given time series.

Syntax

```
GetContainerName(ts TimeSeries)
returns lvarchar;
```

ts The time series from which to obtain the container name.

Description

The API equivalent of this function is **ts_get_containername()**.

Returns

The name of the container for the given time series.

An empty string is returned if the time series is not located in a container.

Example

The following example gets the name of the container holding the stock with ID 600:

```
select GetContainerName(activity_data)
   from activity_stocks
   where stock_id = 600;
```

Related reference:

"The ts_get_containername() function" on page 9-27

GetElem function

The **GetElem** function extracts the element for the given time stamp.

Syntax

```
GetElem(ts     TimeSeries,
       tstamp datetime year to fraction(5),
       flags  integer default 0)
returns row;
```

ts The source time series.

tstamp The time stamp of the entry.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

If the time stamp is for a time that is not part of the calendar, or if it falls before the origin of the given time series, NULL is returned. In some cases, **GetLastValid**, **GetNextValid**, or **GetPreviousValid** might be more appropriate.

For a regular time series, the data extracted is associated with the time period containing the time stamp. For example, if the time series is set to hourly, 8:00 a.m. to 5:00 p.m., the time stamp 3:15 p.m. would return 3:00 p.m. and the data associated with that time.

The API equivalent of this function is **ts_elem()**.

Returns

A row type containing the time stamp and the data from the time series at that time stamp. The type of the row is the same as the time series subtype.

Example

The following query retrieves the stock data of two stocks for a particular day:

```
select GetElem(stock_data,'2011-01-04 00:00:00.00000')
   from daily_stocks
   where stock_name = 'IBM' or stock_name = 'HWP';
```

Related reference:
"Clip function" on page 7-26
"ClipCount function" on page 7-29
"DelElem function" on page 7-33
"GetLastElem function" on page 7-44
"GetLastValid function" on page 7-46
"GetNextValid function" on page 7-49
"GetNthElem function" on page 7-50
"GetPreviousValid function" on page 7-53
"InsElem function" on page 7-57
"PutElem function" on page 7-65
"Transpose function" on page 7-77
"The ts_elem() function" on page 9-20
"GetIndex function" on page 7-43
"GetStamp function" on page 7-54
"UpdElem function" on page 7-113
"The ts_first_elem() function" on page 9-23

GetFirstElem function

The **GetFirstElem** function returns the first element in a time series.

Syntax

```
GetFirstElem(ts    TimeSeries,
             flags integer default 0)
returns row;
```

ts The source time series.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

The API equivalent of this function is **ts_first_elem()**.

Returns

A row type containing the first element of the time series, or `NULL` if there are no elements. The type of the row is the same as the time series subtype.

Example

The following example gets the first element in the time series for the stock with ID 600:

```
select GetFirstElem(activity_data)
   from activity_stocks
   where stock_id = 600;
```

Related reference:

"GetLastElem function" on page 7-44
"The ts_first_elem() function" on page 9-23

GetIndex function

The **GetIndex** function returns the index (offset) of the time series entry associated with the supplied time stamp.

Syntax

```
GetIndex(ts     TimeSeries,
        tstamp datetime year to fraction(5))
returns integer;
```

ts The source time series.

tstamp The time stamp of the entry.

Description

The data extracted is associated with the time period that the time stamp is in. For example, if you have a time series set to hourly, 8:00 a.m. to 5:00 p.m., the time stamp 3:15 p.m. would return the index associated with 3:00 p.m.

The API equivalent of this function is **ts_index()**.

Returns

The integer offset of the entry for the given time stamp in the time series.

NULL is returned if the time stamp is not a valid day in the calendar, or if it falls before the origin of the time series.

Example

The following example returns the offset for the supplied time stamp:

```
select stock_name, GetIndex(stock_data,
      '2011-01-05 00:00:00.00000')
 from daily_stocks;
```

Related reference:

"ClipGetCount function" on page 7-30
"The CalIndex function" on page 6-2
"The CalRange function" on page 6-3
"GetElem function" on page 7-40
"GetNelems function" on page 7-48
"GetNthElem function" on page 7-50
"GetStamp function" on page 7-54
"The ts_index() function" on page 9-32

GetInterval function

The **GetInterval** function returns the interval used by a time series.

Syntax

```
GetInterval(ts TimeSeries)
returns lvarchar;
```

ts The source time series.

Description

The calendars used by time series values can record intervals of one second, minute, hour, day, week, month, or year. The underlying interval of the calendar describes how often a time series records data.

Returns

An LVARCHAR string that describes the time series interval.

Example

The following query finds all stocks that are not traded on a daily basis:

```
select stock_name
from daily_stocks
where GetInterval(stock_data) <> 'day';
```

Related concepts:

"CalendarPattern data type" on page 2-1

Related reference:

"GetCalendar function" on page 7-38

"GetClosestElem function" on page 7-39

"GetOrigin function" on page 7-52

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

GetLastElem function

The **GetLastElem** function returns the final entry stored in a time series.

Syntax

```
GetLastElem(ts   TimeSeries,
            flags integer default 0)
returns row;
```

ts The source time series.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

The API equivalent of this function is **ts_last_elem()**.

Returns

A row-type value containing the time series data and time stamp of the last entry in the time series. If the time series is empty, `NULL` is returned. The type of the row is the same as the time series subtype.

Example

The following query returns the final entry in a time series:

```
select GetLastElem(stock_data)
   from daily_stocks
   where stock_name = 'IBM';
```

The following query retrieves the final entries on a daily stocks table:

```
select GetLastElem(stock_data) from daily_stocks;
```

Related reference:

"GetElem function" on page 7-40

"GetFirstElem function" on page 7-42

"GetLastValid function" on page 7-46

"GetNthElem function" on page 7-50

"PutElem function" on page 7-65

"The ts_last_elem() function" on page 9-34

"GetPreviousValid function" on page 7-53

GetLastNonNull function

The **GetLastNonNull** function returns the last non-null element on or before the date you specify.

Syntax

```
GetLastNonNull(ts          TimeSeries,
              tstamp      datetime year to fraction(5),
              column_name lvarchar default null,
              flags       integer default 0
)
returns row;
```

ts The source time series.

tstamp The time stamp for the element you specify.

column_name **(optional)**
: If you specify a column using the *column_name* argument, the **GetLastNonNull** function returns the last non-null element on or before the specified date that has a non-null value in the specified column.

  If you do not specify the *column_name* argument, the **GetLastNonNull** function returns the last non-null element on or before the date. It is possible that all the columns except the time stamp could be NULL.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

There are no null elements in an irregular time series. Therefore, when you use the **GetLastNonNull** function on an irregular time series, always specify a column name. If you use the **GetLastNonNull** function on an irregular time series without specifying a column name, its effect is equivalent to that of the **GetLastValid** function.

Returns

A non-null element of the time series.

GetLastValid function

The **GetLastValid** function extracts the element for the given time stamp in a time series.

Syntax

```
GetLastValid(ts     TimeSeries,
            tstamp datetime year to fraction(5),
            flags integer default 0)
returns row;
```

ts The source time series.

tstamp The time stamp for the element.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

For regular time series, this function returns the element at the calendar's latest valid timepoint at or before the given time stamp. For irregular time series, it returns the latest element at or preceding the given time stamp.

The equivalent API function is **ts_last_valid()**.

Returns

A row type containing the nearest element at or before the given time stamp. The type of the row is the same as the time series subtype.

If the time stamp is earlier than the origin of the time series, NULL is returned.

Example

The following query returns the last valid entry in a time series at or before a given time stamp:

```
select GetLastValid(stock_data, '2011-01-08 00:00:00.00000')
from daily_stocks
where stock_name = 'IBM';
```

Related reference:

"Clip function" on page 7-26

"ClipCount function" on page 7-29

"GetElem function" on page 7-40

"GetLastElem function" on page 7-44

"GetNextValid function" on page 7-49

"GetNthElem function" on page 7-50

"GetPreviousValid function" on page 7-53

"PutElem function" on page 7-65

"The ts_last_valid() function" on page 9-35

"The ts_next_valid() function" on page 9-40

GetMetaData function

The **GetMetaData** function returns the user-defined metadata from the given time series.

Syntax

```
create function GetMetaData(ts TimeSeries)
returns TimeSeriesMeta;
```

ts The time series to retrieve metadata from.

Returns

This function returns the user-defined metadata contained in the given time series. If the time series does not contain user-defined metadata, then `NULL` is returned. This return value must be cast to the source data type to be useful.

Related tasks:

"Creating a time series with metadata" on page 3-13

Related reference:

"GetMetaTypeName function"

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

"UpdMetaData function" on page 7-114

"The ts_create_with_metadata() function" on page 9-17

"The ts_get_metadata() function" on page 9-28

"The ts_update_metadata() function" on page 9-52

GetMetaTypeName function

The **GetMetaTypeName** function returns the type name of the user-defined metadata type stored in the given time series.

Syntax

```
create function GetMetaTypeName(ts TimeSeries)
returns lvarchar;
```

ts The time series to retrieve the metadata from.

Returns

The type name of the user-defined metadata type stored in the given time series. Returns NULL if the given time series does not have user-defined metadata.

Related reference:

"GetMetaData function" on page 7-47

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

"UpdMetaData function" on page 7-114

"The ts_create_with_metadata() function" on page 9-17

"The ts_get_metadata() function" on page 9-28

"The ts_update_metadata() function" on page 9-52

GetNelems function

The **GetNelems** function returns the number of elements stored in a time series.

Syntax

```
GetNelems(ts TimeSeries)
returns integer;
```

ts The source time series.

Description

For regular time series, **GetNelems** also counts null elements before the last non-null element, so **GetNelems** might not return the same results as **ClipGetCount**, which does not count null elements.

Returns

The number of elements in the time series.

Example

The following query returns all stocks containing fewer than 355 elements:

```
select stock_name from daily_stocks
where GetNelems(stock_data) < 355;
```

The following query returns the last five elements of each time series:

```
select Clip(stock_data, GetNelems(stock_data) - 4,
   GetNelems(stock_data))
from daily_stocks where stock_name = 'IBM';
```

This example only works if the time series has more than four elements.

Related reference:
"ClipGetCount function" on page 7-30
"GetIndex function" on page 7-43
"GetNthElem function" on page 7-50
"GetStamp function" on page 7-54
"The ts_nelems() function" on page 9-38

GetNextNonNull function

The **GetNextNonNull** function returns the next non-null element on or after the date you specify.

Syntax

```
GetNextNonNull(ts        TimeSeries,
           tstamp      datetime year to fraction(5),
           column_name lvarchar default null
           flags       integer default 0
)
returns row;
```

ts The source time series.

tstamp The time stamp for the element.

column_name **(optional)**
: If you specify a column using the *column_name* argument, the **GetNextNonNull** function returns the next non-null element on or after the specified date that has a non-null value in the specified column.

  If you do not specify the *column_name* argument, the **GetLastNonNull** function returns the next non-null element on or after the date specified by *tstamp*. It is possible that all the columns except the time stamp could be `NULL`.

flags Valid values for the *flags* parameter are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

There are no null elements in an irregular time series. Therefore, when you use the **GetNextNonNull** function on an irregular time series, always specify a column name. If you use the **GetNextNonNull** function on an irregular time series without specifying a column name, the function's effect is equivalent to that of the **GetNextValid** function.

Returns

A non-null element of the time series.

GetNextValid function

The **GetNextValid** function returns the nearest entry after a given time stamp.

Syntax

```
GetNextValid(ts     TimeSeries,
            tstamp datetime year to fraction(5),
            flags  integer default 0)
returns row;
```

ts The source time series.

tstamp The time stamp of the entry.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

For regular time series, **GetNextValid** returns the element at the calendar's earliest valid timepoint following the given time stamp. For irregular time series, it returns the earliest element following the given time stamp.

The equivalent API function is **ts_next_valid()**.

Returns

A row type containing the nearest element after the given time stamp. The type of the row is the same as the time series subtype.

NULL is returned if the time stamp is later than that of the last time stamp in the time series.

Example

The following example gets the first element that follows time stamp 2011-01-03 in a regular time series:

```
select GetNextValid(stock_data,'2011-01-03 00:00:00.00000')
   from daily_stocks
   where stock_name = 'IBM';
```

The following example gets the first element that follows time stamp 2011-01-03 in an irregular time series:

```
select GetNextValid(activity_data,
   '2011-01-03 00:00:00.00000')
   from activity_stocks
   where stock_id = 600;
```

Related reference:

"GetElem function" on page 7-40

"GetLastValid function" on page 7-46

"GetNthElem function"

"GetPreviousValid function" on page 7-53

"The ts_next_valid() function" on page 9-40

GetNthElem function

The **GetNthElem** function extracts the entry at a particular offset or position in a time series.

Syntax

```
GetNthElem(ts     TimeSeries,
           N       integer,
           flags   integer default 0)
returns row;
```

ts The source time series.

N The offset or position of an entry in the time series. This value cannot be less than 0.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

For irregular time series, the **GetNthElem** function returns the *N*th element that is found. For regular time series, the *N*th element is also the *N*th interval from the beginning of the time series.

The API equivalent of this function is **ts_nth_elem()**.

Returns

A row value for the requested offset, including all the time series data at that timepoint and the time stamp of the entry in the time series' calendar. The type of the row is the same as the time series subtype.

If the offset is greater than the offset of the last element in the time series, NULL is returned.

Example

The following query returns the last element in a time series:

```
select GetNthElem(stock_data,GetNelems(stock_data)-1)
   from daily_stocks
   where stock_name = 'IBM';
```

The following query returns the element in a time series at a certain time stamp (this could also be done with **GetElem**):

```
select GetNthElem(stock_data,GetIndex(stock_data,
            '2011-01-04 00:00:00.00000'))
   from daily_stocks
   where stock_name = 'IBM';
```

Related reference:
"Clip function" on page 7-26
"ClipCount function" on page 7-29
"ClipGetCount function" on page 7-30
"GetElem function" on page 7-40
"GetIndex function" on page 7-43
"GetLastElem function" on page 7-44
"GetLastValid function" on page 7-46
"GetNelems function" on page 7-48
"GetNextValid function" on page 7-49
"GetPreviousValid function" on page 7-53
"PutElem function" on page 7-65
"Transpose function" on page 7-77
"The ts_nth_elem() function" on page 9-41
"GetStamp function" on page 7-54

GetOrigin function

The **GetOrigin** function returns the origin of the time series.

Syntax

```
GetOrigin(ts TimeSeries)
returns datetime year to fraction(5);
```

ts The source time series.

Description

Every time series value has a corresponding calendar and an origin within the calendar. The calendar describes how often data values appear in the time series. The origin of the time series is the first timepoint within the calendar for which the time series can contain data; however, the time series does not necessarily have data for that timepoint. The origin is set when the time series is created, and it can be changed with **SetOrigin**.

Returns

The time series origin.

Example

The following example returns the time stamp of the origin of the time series for a given stock:

```
select GetOrigin(stock_data)
from daily_stocks
where stock_name = 'IBM';
```

Related reference:
"GetCalendar function" on page 7-38
"GetInterval function" on page 7-43
"GetClosestElem function" on page 7-39
"TSCreate function" on page 7-92
"SetOrigin function" on page 7-74
"TSCreateIrr function" on page 7-94
"The ts_get_origin() function" on page 9-28

GetPreviousValid function

The **GetPreviousValid** function returns the last element before the given time stamp.

Syntax

```
GetPreviousValid(ts     TimeSeries,
                tstamp datetime year to fraction(5),
                flags  integer default 0)
returns row;
```

ts The source time series.

tstamp The time stamp of interest.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

The equivalent API function is **ts_previous_valid()**.

Returns

A row containing the last element before the given time stamp. The type of the row is the same as the time series subtype.

If the time stamp is less than or equal to the time series origin, `NULL` is returned.

Example

The following query gets the first element that precedes time stamp 2011-01-05 in a regular time series:

```
select GetPreviousValid(stock_data,
   '2011-01-05 00:00:00.00000')
   from daily_stocks
   where stock_name = 'IBM';
```

The following query gets the first element that precedes time stamp 2011-01-05 in an irregular time series:

```
select GetPreviousValid(activity_data,
   '2011-01-05 00:00:00.00000')
   from activity_stocks
   where stock_id = 600;
```

Related reference:

"GetElem function" on page 7-40
"GetLastElem function" on page 7-44
"GetLastValid function" on page 7-46
"GetNextValid function" on page 7-49
"GetNthElem function" on page 7-50
"The ts_previous_valid() function" on page 9-43

GetStamp function

The **GetStamp** function returns the time stamp associated with the supplied offset in a time series. Offsets can be positive or negative integers.

Syntax

```
GetStamp(ts     TimeSeries,
        offset integer)
returns datetime year to fraction(5);
```

ts The source time series.

offset The offset.

Description

The equivalent API function is **ts_time()**.

Returns

The time stamp that begins the interval at the specified offset.

Example

The following query returns the time stamp of the beginning of a time series:

```
select GetStamp(stock_data,0)
   from daily_stocks
   where stock_name = 'IBM';
```

Related reference:

"ClipGetCount function" on page 7-30
"GetIndex function" on page 7-43
"GetNelems function" on page 7-48
"The CalIndex function" on page 6-2
"The CalRange function" on page 6-3
"GetElem function" on page 7-40
"GetNthElem function" on page 7-50
"The ts_time() function" on page 9-49

GetThreshold function

The **GetThreshold** function returns the threshold associated with the specified time series.

Syntax

```
GetThreshold(ts     TimeSeries)
returns integer;
```

ts The source time series.

Description

The equivalent API function is **ts_get_threshold()**.

Returns

The threshold of the supplied time series.

Example

The following query returns the threshold of the specified time series:

```
select GetThreshold(stock_data) from daily_stocks;
```

Related reference:

"The ts_get_threshold() function" on page 9-30

HideElem function

The **HideElem** function marks an element, or a set of elements, at a given time stamp as invisible.

Syntax

```
HideElem(ts     TimeSeries,
       tstamp datetime year to fraction(5),
       flags  integer default 0)
returns TimeSeries;

HideElem(ts         TimeSeries,
      multiset_tstamps multiset(datetime year to fraction(5) not null),
       flags        integer default 0)
returns TimeSeries;
```

ts The source time series.

tstamp The time stamp to be made invisible.

multiset_tstamps
The multiset of time stamps to be made invisible.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

After an element is hidden, reading that element returns NULL and writing it results in an error message. It is, however, possible to use **ts_begin_scan()** to read hidden elements.

The API equivalent to this function is **ts_hide_elem()**.

If the time stamp is not a valid timepoint in the time series, an error is raised.

Returns

The modified time series.

Example

The following example hides the element at 2011-01-03 in the time series for IBM:

```
select HideElem(stock_data, '2011-01-03 00:00:00.00000')
   from daily_stocks
   where stock_name = 'IBM';
```

Related concepts:

"Calendar data type" on page 2-3

"CalendarPattern data type" on page 2-1

Related reference:

"DelClip function" on page 7-32

"DelElem function" on page 7-33

"DelTrim function" on page 7-35

"RevealElem function" on page 7-72

"The ts_begin_scan() function" on page 9-6

"The ts_hide_elem() function" on page 9-31

"The ts_reveal_elem() function" on page 9-48

"HideRange function"

HideRange function

The **HideRange** function marks as invisible a range of elements between a starting time stamp and an ending time stamp.

Syntax

```
HideRange(ts     TimeSeries,
        start   datetime year to fraction(5),
        end     datetime year to fraction(5),
        flags   integer default 0
)
returns TimeSeries;
```

ts The time series to act on.

start The starting time stamp.

end The ending time stamp.

flags Valid values for the flags parameter are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

After an element is hidden, reading that element returns `NULL` and writing it results in an error message. It is, however, possible to use **ts_begin_scan()** to read hidden elements, as described in "The ts_begin_scan() function" on page 9-6.

If the time stamp is not a valid timepoint in the time series, an error is raised.

Returns

The modified time series.

Related reference:

"HideElem function" on page 7-55
"RevealRange function" on page 7-73

InsElem function

The **InsElem** function inserts an element into a time series.

Syntax

```
InsElem(ts         TimeSeries,
       row_value   row,
       flags      integer default 0)
returns TimeSeries;
```

ts The time series to act on.

row_value
The row type value to be added to the time series.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

The element must be a row type of the correct type for the time series, beginning with a valid time stamp. If there is already an element with that time stamp in the time series, the insertion is void, and an error is raised. After the insertion is done, the time series must be assigned to a row in a table, or the insertion is lost.

InsElem should be used only within UPDATE and INSERT statements. If it is used within a SELECT statement or a qualification, unpredictable results can occur.

You cannot insert an element at a time stamp that is hidden.

The API equivalent of **InsElem** is **ts_ins_elem()**.

Returns

The new time series with the element inserted.

Example

The following example inserts an element into a time series:

```
update activity_stocks
set activity_data =
   InsElem(activity_data,
      row('2011-10-06 08:06:56.00000', 6.50, 2000,
         1, 007, 3, 1)::stock_trade)
where stock_id = 600;
```

Related reference:

"DelElem function" on page 7-33
"GetElem function" on page 7-40
"InsSet function"
"PutElem function" on page 7-65
"The ts_ins_elem() function" on page 9-33
"UpdElem function" on page 7-113

InsSet function

The **InsSet** function inserts every element of a given set into a time series.

Syntax

```
InsSet(ts           TimeSeries,
      multiset_rows multiset,
      flags        integer default 0)
returns TimeSeries;
```

ts The time series to act on.

multiset_rows
The multiset of new row type values to store in the time series.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

The supplied row type values must have a time stamp as their first attribute. This time stamp is used to determine where in the time series the insertions are to be performed. For example, to insert into a time series that stores a single double-precision value, the row type values passed to **InsSet** would have to contain a time stamp and a double-precision value.

If there is already an element at the given timepoint, the entire insertion is void, and an error is raised.

You cannot insert an element at a time stamp that has been hidden.

Returns

The time series with the multiset inserted.

Example

The following example inserts a set of **stock_trade** items into a time series:

```
update activity_stocks
set activity_data = (select InsSet(activity_data, set_data)
            from activity_load_tab where stock_id = 600)
where stock_id = 600;
```

Related reference:
"DelClip function" on page 7-32
"DelTrim function" on page 7-35
"InsElem function" on page 7-57
"PutSet function" on page 7-69
"UpdSet function" on page 7-115

InstanceId function

The **InstanceId** function determines if the time series is stored in a container and, if it is, returns the instance ID of that time series.

Syntax

```
InstanceId(ts TimeSeries)
returns integer;
```

ts The source time series.

Description

The instance ID is used as an index in the container. It can also be used to perform a lookup in the **TSInstanceTable** table.

Returns

The instance ID associated with the specified time series, unless the time series is stored in a row rather than in a container, in which case the return value is -1.

Example

The following example gets the instance IDs for each stock in the **activity_stocks** table:

```
select stock_id, InstanceId(activity_data) from activity_stocks;
```

Intersect function

The **Intersect** function performs an intersection of the specified time series over the entire length of each time series or over a clipped portion of each time series.

Syntax

```
Intersect(ts TimeSeries,
         ts TimeSeries,...)
returns TimeSeries;

Intersect(set_ts set(TimeSeries))
returns TimeSeries;

Intersect(begin_stamp  datetime year to fraction(5),
         end_stamp    datetime year to fraction(5),
         ts           TimeSeries,
         ts           TimeSeries,...)
returns TimeSeries;
```

```
Intersect(begin_stamp  datetime year to fraction(5),
         end_stamp   datetime year to fraction(5),
         set_ts      set(TimeSeries))
returns TimeSeries;
```

ts The time series that form the intersection. **Intersect** can take from two to eight time series arguments.

set_ts Indicates the intersection of a set of time series.

begin_stamp
The begin point of the clip.

end_stamp
The end point of the clip.

Description

The second and fourth forms of the function intersect a set of time series. The resulting time series has one DATETIME YEAR TO FRACTION(5) column followed by each column in each time series in order, not including the other time stamps. When using the second or fourth form, it is important to ensure that the order of the time series in the set is deterministic so that elements remain in the correct order.

Since the resulting time series is a different type from the input time series, the result of the intersection must be cast.

Intersect can be thought of as a join on the time stamp columns.

If any of the input time series is irregular, the resulting time series is irregular.

For the purposes of **Intersect**, the value at a given timepoint is that of the most recent valid element. For regular time series, this is the value corresponding to the current interval, which can be NULL; it is not necessarily the most recent non-null value. For irregular time series, this condition never occurs, because irregular time series do not have null intervals.

For example, consider the intersection of two irregular time series, one containing bid prices for a certain stock, and one containing asking prices. The intersection of the two time series contains bid and ask values for each timepoint at which a price was either bid or asked. Now consider a timepoint at which a bid was made but no price was asked. The intersection at that timepoint contains the bid price offered at that timepoint, along with the most recent asking price.

If an intersection involves one or more regular time series, the resulting time series starts at the latest of the start points of the input time series and ends at the earliest of the end points of the regular input time series. If all the input time series are irregular, the resulting irregular time series starts at the latest of the start points of the input time series and ends at the latest of the end points. If a union involves one or more time series, the resulting time series starts at the first of the start points of the input time series and ends at the latest of the end points of the input time series. Other than this difference in start and end points, and of the resulting calendar, there is no difference between union and intersection involving time series.

In an intersection, the resulting time series has a calendar that is the combination of the calendars of the input time series with the AND operator. The resulting calendar is stored in the **CalendarTable** table. The name of the resulting calendar is a string containing the names of the calendars of the input time series joined by an ampersand (&). For example, if two time series are intersected, and **mycal** and **yourcal** are the names of their corresponding calendars, the resulting calendar is named **mycal&yourcal**.

To be certain of the order of the columns in the resultant time series when using **Intersect** with the *set_ts* argument, use the ORDER BY clause.

Apply also combines multiple time series into a single time series. Therefore, using **Intersect** within **Apply** is often unnecessary.

Returns

The time series that results from the intersection.

Example

The following example returns the intersection of two time series:

```
select Intersect(d1.stock_data,
      d2.stock_data)::TimeSeries(stock_bar_union)
    from daily_stocks d1, daily_stocks d2
    where d1.stock_name='IBM' and d2.stock_name='HWP';
```

The following query intersects two time series and returns data only for time stamps between 2011-01-03 and 2011-01-05:

```
select Intersect('2011-01-03 00:00:00.00000'
            ::datetime year to fraction(5),
            '2011-01-05 00:00:00.00000'
            ::datetime year to fraction(5),
            d1.stock_data,
            d2.stock_data
          )::TimeSeries(stock_bar_union)
   from daily_stocks d1, daily_stocks d2
   where d1.stock_name = 'IBM' and d2.stock_name = 'HWP';
```

Related reference:

"Apply function" on page 7-11
"Union function" on page 7-111

IsRegular function

The **IsRegular** function tells whether a given time series is regular.

Syntax

```
IsRegular(ts TimeSeries)
returns boolean;
```

ts The source time series.

Returns

`TRUE` if the time series is regular; otherwise `FALSE`.

Example

The following query gets stock IDs for all stocks in irregular time series:

```
select stock_id
   from activity_stocks
   where not IsRegular(activity_data);
```

Related reference:

"The ts_get_flags() function" on page 9-27

Lag function

The **Lag** function creates a new regular time series in which the data values lag the source time series by a fixed offset.

Syntax

```
Lag(ts     TimeSeries,
   nelems integer)
returns TimeSeries;
```

ts The source time series.

nelems The number of elements to lag the series by. Positive values lag the result behind the argument, and negative values lead the result ahead.

Description

Lag shifts only offsets, not the source time series. Therefore, a lag of -2 eliminates the first two elements. For example, if there is a daily time series, Monday to Friday, and a one-day lag (an argument of -1) is imposed, then there is no first Monday, the first Tuesday is Monday, and the next Monday is Friday. It would be more typical of a daily time series to lag a full week.

For example, this function allows the user to create a hypothetical time series, with closing stock prices for each day moved two days ahead on the calendar.

Lag is valid only for regular time series.

Returns

A new time series with the same calendar and origin as the source time series but that has its elements assigned to different offsets.

Example

The following query creates a new time series that lags the original time series by three days:

```
select Lag(stock_data,3)
from daily_stocks
where stock_name = 'IBM';
```

Logn function

The **Logn** function returns the natural logarithm of a time series. It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Minus function

The **Minus** function subtracts one time series from another. It is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Mod**, **Plus**, **Pow**, and **Times**.

Related reference:

"Binary arithmetic functions" on page 7-21

Mod function

The **Mod** function computes the modulus or remainder of a division of one time series by another. It is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Minus**, **Plus**, **Pow**, and **Times**.

Related reference:

"Binary arithmetic functions" on page 7-21

Negate function

The **Negate** function negates a time series. It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

NullCleanup function

The **NullCleanup** function frees any pages that contain only null elements in a range or for the whole time series.

Syntax

```
NullCleanup(ts          TimeSeries,
       begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5),
       flags       integer default 0)
returns TimeSeries;

NullCleanup(ts          TimeSeries,
       flags       integer default 0)
returns TimeSeries;

NullCleanup(ts          TimeSeries,
       begin_stamp datetime year to fraction(5)
       flags       integer default 0)
returns TimeSeries;

NullCleanup(ts          TimeSeries,
       NULL,
       end_stamp   datetime year to fraction(5),
       flags       integer default 0)
returns TimeSeries;
```

ts The time series to act on.

begin_stamp
The begin point of the range.

end_stamp
The end point of the range.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

Use the **NullCleanup** function to free empty pages in one of the following time ranges:

- A specified begin point and a specified end point
- The whole time series
- A specified begin point and the end of the time series
- The beginning of the time series and a specified end point

If the begin point of the range falls before the origin of the time series, an error is raised.

Returns

A time series with all the empty pages in the range freed.

Examples

Example 1: Free empty pages between specified begin and end points

The following example frees the empty pages in a one-day range on a given day for a time series named **meter_data** for the location ID of 4727354321000111:

```
UPDATE ts_data
SET meter_data = NullCleanup(meter_data,
        '2010-11-11 00:00:00.00000'
     ::datetime year to fraction(5),
        '2010-11-11 00:00:00.00000'
     ::datetime year to fraction(5))
WHERE loc_esi_id = 4727354321000111;
```

Example 2: Free all empty pages in the time series

The following example frees all empty pages in the time series named **meter_data** for the location ID of 4727354321000111:

```
UPDATE ts_data
SET meter_data = NullCleanup(meter_data)
WHERE loc_esi_id = 4727354321000111;
```

Example 3: Free empty pages from the beginning of the time series to a specified date

The following example frees empty pages from the beginning of the time series to the specified end point for a time series named **meter_data** for the location ID of 4727354321000111:

```
UPDATE ts_data
SET meter_data = NullCleanup(meter_data, NULL,
        '2010-11-11 00:00:00.00000'
     ::datetime year to fraction(5))
WHERE loc_esi_id = 4727354321000111;
```

Plus function

The **Plus** function adds two time series together. It is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Minus**, **Mod**, **Pow**, and **Times**.

Related reference:

"Binary arithmetic functions" on page 7-21

Positive function

The **Positive** function returns the argument. It is bound to the unary "+" operator. It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Pow function

The **Pow** function raises the first argument to the power of the second. It is one of the binary arithmetic functions that work on time series. The others are **Atan**, **Divide**, **Minus**, **Mod**, **Plus**, and **Times**.

Related reference:

"Binary arithmetic functions" on page 7-21

PutElem function

The **PutElem** function adds an element to a time series at the timepoint indicated in the supplied row type.

Syntax

```
PutElem(ts         TimeSeries,
       row_value  row,
       flags      integer default 0)
returns TimeSeries;
```

ts The time series to act on.

row_value
The new row type value to store in the time series.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

For regular time series, if there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

For irregular time series, if there is no data at the given timepoint, the new data is inserted. If there is data at the given timepoint, the following algorithm is used to determine where to place the data:

1. Round the time stamp up to the next second.
2. Search backwards for the first element less than the new time stamp.
3. Insert the new data at this time stamp plus 10 microseconds.

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The API equivalent of **PutElem** is **ts_put_elem()**.

Returns

A modified time series that includes the new values.

Example

The following example appends an element to a time series:

```
update daily_stocks
set stock_data = PutElem(stock_data,
   row(NULL::datetime year to fraction(5),
   2.3, 3.4, 5.6, 67)::stock_bar)
   where stock_name = 'IBM';
```

The following example updates a time series:

```
update activity_stocks
set activity_data = PutElem(activity_data,
   row('2011-08-25 09:06:00.00000',
      6.25, 1000, 1, 007, 2, 1)::stock_trade)
   where stock_id = 600;
```

Related concepts:

"The TSVTMode parameter" on page 4-11

Related reference:

"DelElem function" on page 7-33
"GetElem function" on page 7-40
"GetLastElem function" on page 7-44
"GetLastValid function" on page 7-46
"GetNthElem function" on page 7-50
"InsElem function" on page 7-57
"PutElemNoDups function"
"PutSet function" on page 7-69
"TSCreate function" on page 7-92
"The ts_put_elem() function" on page 9-44
"PutNthElem function" on page 7-68
"UpdElem function" on page 7-113

PutElemNoDups function

The **PutElemNoDups** function inserts a single element into a time series. If there is already an element at the specified timepoint, it is replaced by the new element.

Syntax

```
PutElemNoDups(ts         TimeSeries,
             row_value  row,
             flags      integer default 0)
returns TimeSeries;
```

ts The time series to act on.

row_value
: The new row type value to store in the time series.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is `0`.

Description

If the time stamp is `NULL`, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

If there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The API equivalent of **PutElemNoDups** is **ts_put_elem_no_dups()**.

Returns

A modified time series that includes the new values.

Example

The following example updates a time series:

```
update activity_stocks
set activity_data = PutElemNoDups(activity_data,
   row('2011-08-25 09:06:00.00000', 6.25,
      1000, 1, 007, 2, 1)::stock_trade)
   where stock_id = 600;
```

Related concepts:

"The TSVTMode parameter" on page 4-11

Related reference:

"PutElem function" on page 7-65

"The ts_put_elem_no_dups() function" on page 9-45

PutNthElem function

The **PutNthElem** function puts the supplied row at the supplied offset in a regular time series.

Syntax

```
PutNthElem(ts         TimeSeries,
          row_value   row,
          N          integer,
          flags      integer default 0)
returns TimeSeries;
```

ts The time series to act on.

row_value
: The new row type value to store in the time series.

N The offset. Must be greater than or equal to 0.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

This function is similar to **PutElem**, except **PutNthElem** takes an offset instead of a time stamp.

If there is data at the given offset, it is updated with the new data; otherwise, the new data is inserted.

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

Returns

A modified time series that includes the new values.

Example

The following example puts data in the first element of the IBM time series:

```
update daily_stocks
set stock_data =
    PutNthElem(stock_data,
   row(NULL::datetime year to fraction(5), 355, 309,
   341, 999)::stock_bar, 0)
where stock_name = 'IBM';
```

Related reference:

"PutElem function" on page 7-65

PutSet function

The **PutSet** function updates a time series with the supplied multiset of row type values.

Syntax

```
PutSet(ts          TimeSeries,
      multiset_ts set,
      flags       integer default 0)
returns TimeSeries;
```

ts The time series to act on.

multiset_ts
The multiset of new row type values to store in the time series.

flags Valid values for the *flags* argument are described later in this section. The default is 0.

Description

For each element in the multiset of rows, if the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

For regular time series, if there is data at a given timepoint, it is updated with the new data; otherwise, the new data is inserted.

For irregular time series, if there is no data at a given timepoint, the new data is inserted. If there is data at the given timepoint, the following algorithm is used to determine where to place the data:
1. Round the time stamp up to the next second.
2. Search backward for the first element less than the new time stamp.
3. Insert the new data at this time stamp plus 10 microseconds.

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The flags argument values

The value of the *flags* argument is the sum of the desired flag values from the following table.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Indicates that the time series can be read and written to.

Flag	Value	Meaning
TSOPEN_READ_HIDDEN	1	Indicates that hidden elements should be treated as if they are not hidden.
TSOPEN_WRITE_HIDDEN	2	Allows hidden elements to be written to without first revealing them. The element remains hidden afterward.
TSOPEN_WRITE_AND_HIDE	4	Causes any elements written to a time series also to be marked as hidden.
TSWRITE_AND_REVEAL	8	Reveals any hidden element written to.
TSOPEN_NO_NULLS	32	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as `NULL`. If TSOPEN_NO_NULLS is set, an element that has each column set to `NULL` is returned instead.
TS_PUTELEM_NO_DUPS	64	Determines whether the **PutSet** function adds elements using the **PutElem** function (default) or the **PutElemNodups** function (see "PutElem function" on page 7-65 and "PutElemNoDups function" on page 7-67). If this flag is set, the **PutSet** function uses **PutElemNoDups**.

Returns

A modified time series that includes the new values.

Example

The following example updates a time series with a multiset:

```
update activity_stocks
set activity_data = (select PutSet(activity_data, set_data)
            from activity_load_tab where stock_id = 600)
where stock_id = 600;
```

Related reference:

"DelClip function" on page 7-32

"DelTrim function" on page 7-35

"InsSet function" on page 7-58

"PutElem function" on page 7-65

"TSCreate function" on page 7-92

"UpdSet function" on page 7-115

"PutTimeSeries function"

PutTimeSeries function

The **PutTimeSeries** function puts every element of the first time series into the second time series.

Syntax

```
PutTimeSeries(ts1   TimeSeries,
             ts2   TimeSeries,
             flags integer default 0)
returns TimeSeries;
```

ts1 The time series to be inserted.

ts2 The time series into which the first time series is to be inserted.

flags Valid values for the *flags* argument are described later in this topic.

Description

If both time series contain data at the same timepoint, the rule of **PutElem** is followed (see "PutElem function" on page 7-65), unless the TS_PUTELEM_NO_DUPS value of the *flags* parameter is set.

Both time series must have the same calendar. Also, the origin of the time series specified by the first argument must be later than or equal to the origin of the time series specified by the second argument.

This function can be used to convert a regular time series to an irregular one.

Important: Converting an irregular time series to regular often requires aggregation information, which can be provided using the **AggregateBy** function.

Elements are added to the second time series by calling **ts_put_elem()** (if the TS_PUTELEM_NO_DUPS value of the *flags* parameter is not set).

The API equivalent of this function is **ts_put_ts()**.

The flags argument values

When the source time series opens, your setting of the *flags* argument is combined (using the AND operator) with the TSOPEN_READ_HIDDEN value. The returned time series is opened with your setting of the *flags* argument combined (using the AND operator) with TSOPEN_WRITE_AND_HIDE, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_HIDDEN.

The value of *flags* is the sum of the desired flag values from the following table.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Hidden elements are read from the source time series as NULL.
TSOPEN_READ_HIDDEN	1	Indicates that the elements in the source time series that are hidden are read as if they are not hidden.
TSOPEN_WRITE_HIDDEN	2	Elements that were hidden in the source time series are hidden in the resulting time series.
TSOPEN_WRITE_AND_HIDE	4	Causes all elements written to a resulting time series also to be marked as hidden.
TSWRITE_AND_REVEAL	8	Reveals all elements written to the resulting time series.

Flag	Value	Meaning
TSOPEN_NO_NULLS	32	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as `NULL`. If TSOPEN_NO_NULLS is set, an element that has each column set to `NULL` is returned instead.
TS_PUTELEM_NO_DUPS	64	Determines whether the **PutTimeSeries** function adds elements using the **PutElem** function or the **PutElemNoDups** function (see "PutElem function" on page 7-65 and "PutElemNoDups function" on page 7-67). If this flag is set, **PutTimeSeries** uses **PutElemNoDups**.

These flags can be used in any combination except the following four combinations:

- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_REVEAL
- TSOPEN_WRITE_AND_REVEAL and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_HIDE, and TSOPEN_WRITE_AND_REVEAL

The TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_AND_HIDE flags cannot be used with TSOPEN_READ_HIDDEN.

Returns

A version of the second time series into which the first time series has been inserted.

Example

The following example converts a regular time series to an irregular one. The **daily_stocks** table holds regular time series data, and the **activity_stocks** table holds irregular time series data. Additionally, the elements in the **daily_stocks** time series are converted from **stock_bar** to **stock_trade**:

```
update activity_stocks
    set activity_data = PutTimeSeries(activity_data, 'calendar(daycal),
irregular'::TimeSeries(stock_trade))
    where stock_id = 600;
```

Related reference:

"AggregateBy function" on page 7-6

"PutSet function" on page 7-69

"The ts_put_ts() function" on page 9-47

"SetOrigin function" on page 7-74

RevealElem function

The **RevealElem** function makes an element at a given time stamp available for a scan. It reverses the effect of **HideElem**.

Syntax

```
RevealElem(ts     TimeSeries,
          tstamp datetime year to fraction(5))
returns TimeSeries;

RevealElem(ts        TimeSeries,
          set_stamps multiset(datetime year to fraction(5)))
returns TimeSeries;
```

ts The time series to act on.

tstamp The time stamp to be made visible to a scan.

set_stamps
The multiset of time stamps to be made visible to a scan.

Returns

The modified time series.

Example

The following example hides the element at 2011-01-03 in the IBM time series and then reveals it:

```
select HideElem(stock_data, '2011-01-03 00:00:00.00000')
   from daily_stocks
   where stock_name = 'IBM';

select RevealElem(stock_data, '2011-01-03 00:00:00.00000')
   from daily_stocks
   where stock_name = 'IBM';
```

Related reference:

"HideElem function" on page 7-55
"The ts_reveal_elem() function" on page 9-48

RevealRange function

The **RevealRange** function makes hidden elements in a specified date range visible. It reverses the effect of **HideRange**.

Syntax

```
RevealRange(ts     TimeSeries,
            start   datetime year to fraction(5),
            end     datetime year to fraction(5),
)
returns TimeSeries;
```

ts The time series to act on.

start The time stamp at the start of the range.

end The time stamp at the end of the range.

Returns

The modified time series.

Related reference:

"HideRange function" on page 7-56

Round function

The **Round** function rounds a time series to the nearest whole number. It is one of the unary arithmetic functions that work on time series.The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

SetContainerName function

The **SetContainerName** function sets the container name for a time series, even if the time series already has a container name.

Syntax

```
SetContainerName(ts             TimeSeries,
                 container_name  varchar(128,1))
returns TimeSeries;
```

ts The time series to act on.

container_name
The name of the container.

Description

If a time series is stored in a container, you can use the **SetContainerName** function to copy the time series from one container to another. The time series is copied to the container that you specify with the *container_name* parameter. The original time series is unaffected.

Returns

A time series with a new container set.

Example

The following example creates the container **tsirr** and sets a time series to it:

```
execute procedure TSContainerCreate('tsirr', 'rootdbs',
   'stock_bar_union', 0, 0);

select SetContainerName(Union(s1.stock_data,
      s2.stock_data)::TimeSeries(stock_bar_union),
      'tsirr')
from daily_stocks s1, daily_stocks s2
where s1.stock_name = 'IBM' and s2.stock_name = 'AA02';
```

Related reference:

"TSContainerCreate procedure" on page 7-83

SetOrigin function

The **SetOrigin** function moves the origin of a time series back in time.

Syntax

```
SetOrigin(ts     TimeSeries,
          origin datetime year to fraction(5))
returns TimeSeries;
```

ts The time series to act on.

origin The new origin of the time series.

Description

If the supplied origin is not a valid timepoint in the given time series calendar, the first valid timepoint following the supplied origin becomes the new origin. The new origin must be earlier than the current origin. To move the origin forward, use the **Clip** function.

Returns

The time series with the new origin.

Example

The following example sets the origin of the **stock_data** time series:

```
update daily_stocks
   set stock_data = SetOrigin(stock_data,
      '2011-01-02 00:00:00.00000');
```

Related reference:

"Apply function" on page 7-11

"Clip function" on page 7-26

"GetOrigin function" on page 7-52

"PutTimeSeries function" on page 7-70

Sin function

The **Sin** function returns the sine of its argument. It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Sqrt function

The **Sqrt** function returns the square root of its argument. It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, and **Tan**.

Related reference:

"Unary arithmetic functions" on page 7-110

Sum function

The **Sum** function is the sum aggregate function for the **TimeSeries** data type.

Syntax

```
Sum(ts TimeSeries)
returns TimeSeries;
```

ts The time series to act on.

Description

All time series summed must have the same calendar. They are not required to have the same start and end points. The columns of each element of the time series are summed with the corresponding columns of the elements in the other time series at the same timepoint. An error is raised if one or more columns cannot be summed. Null values are ignored.

Returns

A time series containing the sum of the columns of each element at the same timepoint in the source time series.

Example

The following query retrieves the sum of the volumes for a particular day for all stocks in the table **daily_stocks**:

```
select GetElem(stock_data,'2011-01-04 00:00:00.00000')
        from daily_stocks;

select Apply('$final * $vol',
          '2011-01-04 00:00:00.00000'
         ::datetime year to fraction(5),
          '2011-12-30 00:00:00.00000'
         ::datetime year to fraction(5),
          stock_data)::TimeSeries(one_real)
from daily_stocks;
```

Tan function

The **Tan** function returns the tangent of its argument. It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, and **Sin**.

Related reference:

"Unary arithmetic functions" on page 7-110

Times function

The **Times** function multiplies one time series by another. It is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Minus**, **Mod**, **Plus**, and **Pow**.

Related reference:

"Binary arithmetic functions" on page 7-21

TimeSeriesRelease function

The **TimeSeriesRelease** function returns an LVARCHAR string containing theTimeSeries extension version number and build date.

Syntax

```
TimeSeriesRelease()
returns lvarchar;
```

Returns

The version number and build date.

Example

The following example shows how to get the version number using DB-Access:

```
execute function TimeSeriesRelease();
```

Transpose function

The **Transpose** function converts time series data for processing in a tabular format.

Syntax

```
Transpose (ts       TimeSeries,
       begin_stamp datetime year to fraction(5) default NULL,
       end_stamp   datetime year to fraction(5) default NULL,
       flags    integer default 0)
returns row;

Transpose (query    lvarchar,
        dummy        row,
        begin_stamp  datetime year to fraction(5) default NULL,
        end_stamp    datetime year to fraction(5) default NULL,
       col_name   lvarchar default NULL,
       flags      integer default 0)
returns row with (iterator);
```

ts The time series to transpose.

begin_stamp
: The begin point of the range. Can be NULL.

end_stamp
: The end point of the range. Can be NULL.

flags Determines how a scan should work on the returned set.

query A string containing a SELECT statement that can return multiple columns but only one time series column. The non-time-series columns are concatenated with each time series element in the returned rows.

dummy
: A row type that must be passed in as NULL and cast to the expected return type of each row returned by the query string version of the **Transpose** function.

col_name
: If *col_name* is not NULL, only the column specified with this parameter will be used from the time series element, plus the non-time-series columns.

Description

The **Transpose** function is an iterator function. You can run the **Transpose** function with the EXECUTE FUNCTION statement or in a table expression.

Normally the transpose function skips NULL elements when returning the rows found in a time series. If the TS_SCAN_NULLS_OK (0x40) bit of the *flags* parameter is set, the **Transpose** function returns NULL elements.

If the beginning point is NULL, the scan starts at the first element of the time series, unless the TS_SCAN_EXACT_START value of the *flags* parameter is set.

If the end point is NULL, the scan ends at the last element of the time series, unless the TS_SCAN_EXACT_END value of the *flags* parameter is set.

The flags argument values

The *flags* argument determines how a scan should work on the returned set. The value of *flags* is the sum of the desired flag values from the following table.

Flag	Value	Meaning
TS_SCAN_HIDDEN	512	Return hidden elements marked by **HideElem** (see "HideElem function" on page 7-55).
TS_SCAN_EXACT_START	256	Return the element at the beginning timepoint, adding null elements if necessary.
TS_SCAN_EXACT_END	128	Return elements up to the end point (return NULL if necessary).
TS_SCAN_NULLS_OK	64	Return null time series elements (by default, time series elements that are NULL are not returned).
TS_SCAN_NO_NULLS	32	Instead of returning a null row, return a row with the time stamp set and the other columns set to NULL.
TS_SCAN_SKIP_END	16	Skip the element at the end timepoint of the scan range.
TS_SCAN_SKIP_BEGIN	8	Skip the element at the beginning timepoint of the scan range.
TS_SCAN_SKIP_HIDDEN	4	Used by **ts_begin_scan()** to tell **ts_next()** not to return hidden elements.

Returns

Multiple rows containing a time stamp and the other columns of the time series elements.

Examples

Example 1: Convert time series data to a table

The following statement converts the data from **stock_data** for IBM to tabular form:

```
execute function Transpose((select stock_data
   from daily_stocks where stock_name = 'IBM'));
```

Example 2: Transpose clipped data

The following statement converts data for a clipped range into tabular form:

```
execute function Transpose((select stock_data from daily_stocks
             where stock_name = 'IBM'),
             datetime(2011-01-05) year to day,
             NULL::datetime year to fraction(5));
```

The statement returns the following data in the form of a row data type:

```
ROW('2011-01-06 00:00:00.00000',99.00000
000000,54.00000000000,66.00000000000,888.0000000000)
```

Example 3: Convert time series and other data into tabular format

The following example returns the time series columns together with the non-time-series columns in tabular form:

```
execute function Transpose ('select * from daily_stocks',  NULL::row(stock_id
int, stock_name lvarchar,
   t datetime year to fraction(5), high real, low real, final real, volume real));
```

Example 4: Display specific data as multiple fields within a single column

The following statement selects the time and energy readings from a time series:

```
SELECT mr.t,mr.energy
  FROM TABLE(transpose
            ((SELECT readings FROM smartmeters
              WHERE meter_id = 13243))::smartmeter_row)
          AS tab(mr);
```

The statements returns a table named **tab** that contains one column, named **mr**. The **mr** column is an unnamed row type that has the same fields as the **TimeSeries** subtype named **smartmeter_row**. The output has a field for time and a field for energy:

```
t                                energy

2011-01-01 00:00:00.00000           29
2011-01-01 00:15:00.00000           18
2011-01-01 00:30:00.00000           13
2011-01-01 00:45:00.00000           26
2011-01-01 01:00:00.00000           21
2011-01-01 01:15:00.00000           15
2011-01-01 01:30:00.00000           20
2011-01-01 01:45:00.00000           24
2011-01-01 02:00:00.00000           30
2011-01-01 02:15:00.00000           30
2011-01-01 02:30:00.00000           29
2011-01-01 02:45:00.00000           32
2011-01-01 03:00:00.00000           29
```

Example 5: Display specific data in a table with multiple columns

The following statement uses the statement from the previous example inside a table expression in the FROM clause:

```
SELECT * FROM (
         SELECT mr.t,mr.energy,mr.temperature
             FROM TABLE(transpose
                 ((SELECT readings FROM smartmeters
                   WHERE meter_id = 13243))::smartmeter_row)
                     AS tab(mr)
        ) AS sm(t,energy,temp)
  WHERE temp < -10;
```

The statement returns the following data in the form of a table named **sm** that contains three columns:

```
t                             energy         temp

2011-01-01 00:00:00.00000         29  -13.0000000000
2011-01-01 00:30:00.00000         13  -18.0000000000
2011-01-01 01:00:00.00000         21  -13.0000000000
2011-01-01 01:15:00.00000         15  -11.0000000000
2011-01-01 03:15:00.00000         22  -19.0000000000
2011-01-01 03:45:00.00000         28  -14.0000000000
2011-01-01 04:00:00.00000         19  -14.0000000000
2011-01-01 04:30:00.00000         27  -14.0000000000
2011-01-01 04:45:00.00000         27  -15.0000000000
2011-01-01 05:00:00.00000         28  -11.0000000000
```

Related reference:

“GetElem function” on page 7-40

“GetNthElem function” on page 7-50

“TSColNameToList function” on page 7-82

“TSColNumToList function” on page 7-83

“TSRowNameToList function” on page 7-99

“TSRowNumToList function” on page 7-100

“TSRowToList function” on page 7-101

“TSSetToList function” on page 7-107

TSAddPrevious function

The **TSAddPrevious** function sums all the values it is called with and returns the current sum every time it is called. The current argument is not included in the sum.

Syntax

```
TSAddPrevious(current_value smallfloat)
returns smallfloat;

TSAddPrevious(current_value double precision)
returns double precision;
```

current_value
The current value.

Description

This function is useful only when used within an **AggregateBy** or **Apply** function.

Returns

The sum of all previous values returned by this function.

Example

The following example uses **TSAddPrevious** to calculate the summation of the average dollars into or out of a market or equity:

```
select Apply('TSAddPrevious($vol * (($final - $low) - ($high - $final) / (.0001
+ $high - $low)) * (($high + $low + $final) / 3))',
      '2011-01-03 00:00:00.00000'::datetime year to fraction(5),
```

```
      '2011-01-08 00:00:00.00000'::datetime year to fraction(5),
      stock_data)::TimeSeries(one_real)
from daily_stocks
where stock_name = 'IBM';
```

Related reference:

"Apply function" on page 7-11

"TSCmp function"

"TSDecay function" on page 7-96

"TSPrevious function" on page 7-97

"TSRunningAvg function" on page 7-102

"TSRunningSum function" on page 7-105

TSCmp function

The **TSCmp** function compares two values.

Syntax

```
TSCmp(value1 smallfloat,
       value2 smallfloat)
returns int;

TSCmp(value1 double precision,
       value2 double precision)
returns int;
```

value1 The first value to be compared.

value2 The second value to be compared.

Description

The **TSCmp** function returns -1, 0, and 1 if its first argument is, respectively, less than, equal to, or greater than its second.

This function is useful only when used within the **Apply** function.

TSCmp takes either two SMALLFLOAT values or two DOUBLE PRECISION values; both values must be the same type.

Returns

-1 If the first argument is less than the second.

0 If the first argument is equal to the second.

1 If the first argument is greater than the second.

Example

The following example uses **TSCmp** to calculate the on-balance volume, a continuous summation that adds the daily volume to the running total if the stock or index advances and subtracts the volume if it declines:

```
select Apply
   ('TSAddPrevious(TSCmp($final, TSPrevious($final)) * $vol)',
   '2011-01-03 00:00:00.00000'::datetime year to fraction(5),
```

```
   '2011-01-08 00:00:00.00000'::datetime year to fraction(5),
   stock_data)::TimeSeries(one_real)
from daily_stocks
where stock_name = 'IBM';
```

Related reference:

"Apply function" on page 7-11

"TSAddPrevious function" on page 7-80

"TSDecay function" on page 7-96

"TSPrevious function" on page 7-97

"TSRunningAvg function" on page 7-102

"TSRunningSum function" on page 7-105

TSColNameToList function

The **TSColNameToList** function takes a TimeSeries column and returns a list (collection of rows) containing the values of one of the columns in the elements of the time series. Null elements are not added to the list.

Syntax

```
TSColNameToList(ts     TimeSeries,
                colname lvarchar)
returns list
```

ts The time series to act on.

colname
The column to return.

Description

Because this aggregate function can return rows of any type, the return value must be explicitly cast at runtime.

Returns

A list (collection of rows).

Example

This query returns a list of all values in the column **high**:

```
select * from table((select
   TSColNameToList(stock_data, 'high')::list(real
   not null) from daily_stocks));
```

Related reference:

"Transpose function" on page 7-77

"TSColNumToList function"

"TSRowNameToList function" on page 7-99

"TSRowNumToList function" on page 7-100

"TSSetToList function" on page 7-107

"TSRowToList function" on page 7-101

TSColNumToList function

The **TSColNumToList** function takes a **TimeSeries** column and returns a list (collection of rows) containing the values of one of the columns in the elements of the time series. Null elements are not added to the list.

Syntax

```
TSColNumToList(ts     TimeSeries,
               colnum integer)
returns list
```

ts The time series to act on.

colnum The column to return.

Description

The column is specified by its number; column numbering starts at 1, with the first column following the time stamp column.

Because this aggregate function can return rows of any type, the return value must be explicitly cast at runtime.

Returns

A list (collection of rows).

Example

This query returns a list of all values in the column **high**:

```
select * from table((select
   TSColNumToList(stock_data, 1)::list(real
   not null) from daily_stocks));
```

Related reference:

"TSColNameToList function" on page 7-82

"Transpose function" on page 7-77

"TSRowNameToList function" on page 7-99

"TSRowNumToList function" on page 7-100

"TSSetToList function" on page 7-107

"TSRowToList function" on page 7-101

TSContainerCreate procedure

The **TSContainerCreate** procedure creates a new container with the specified name for the specified **TimeSeries** subtype.

Only users with update privileges on the **TSContainerTable** table can run this procedure.

Syntax

```
TSContainerCreate(container_name  varchar(128,1),
                  dbspace_name   varchar(128,1),
                  ts_type        varchar(128,1),
                  container_size integer,
                  container_grow integer);
```

container_name
: The name of the new container. The container name must be unique.

dbspace_name
: The name of the dbspace that will hold the container.

ts_type
: The name of the **TimeSeries** subtype that will be placed in the container. This argument must be the name of an existing row type that begins with a time stamp.

container_size
: The initial size of the container, in KB. If this argument is 0 or less, a default size of 16 KB is used. If this parameter is positive, it must be at least four pages. The maximum size of a container depends on the page size:

- For 2 KB pages, the maximum size is 32 GB.
- For 4 KB pages, the maximum size is 64 GB.
- For 8 KB pages, the maximum size is 128 GB.
- For 16 KB pages, the maximum size is 256 GB.

container_grow
: The increments by which the container grows, in KB. If this argument is 0 or less, a default size of 16 KB is used. If this parameter is positive, it must be at least four pages.

Description

By default, containers are created automatically as needed when you insert data into a time series. However, you can create additional containers by using the **TSContainerCreate** procedure.

As a result of the **TSContainerCreate** procedure, the database server creates the container when the first time series is inserted into that container. Both regular and irregular time series are stored in containers when they exceed a specified size, which is specified when the time series is created. You can create multiple containers in the same dbspace.

A row is inserted in the **TSContainerTable** table.

Example

The following example creates a new container called **new_cont** in the space **rootdbs** for the time series type **stock_bar**:

```
execute procedure TSContainerCreate('new_cont', 'rootdbs','stock_bar', 0, 0);
```

Related concepts:

"TSInstanceTable table" on page 2-8

"TSContainerTable table" on page 2-9

Related tasks:

"Managing containers" on page 3-7

"Configuring additional container pools" on page 3-9

Related reference:

"SetContainerName function" on page 7-74

"TSContainerDestroy procedure"

TSContainerDestroy procedure

The **TSContainerDestroy** procedure deletes the container row from the **TSContainerTable** table and removes the container and its corresponding system catalog rows.

Syntax

```
TSContainerDestroy(container_name varchar(128,1));
```

container_name
: The name of the container to destroy.

Description

Destroying a container is permitted only when no time series exist in that container; even an empty time series prevents a container from being destroyed.

Only users with update privileges on the **TSContainerTable** table can execute this procedure.

Example

The following example destroys the container **ctnr_stock**:

```
execute procedure TSContainerDestroy('ctnr_stock');
```

Related concepts:

"TSInstanceTable table" on page 2-8

"TSContainerTable table" on page 2-9

Related tasks:

"Managing containers" on page 3-7

Related reference:

"TSContainerCreate procedure" on page 7-83

TSContainerNElems function

The **TSContainerNElems** function returns the number of time series data elements stored in the specified container or in all containers.

Syntax

```
TSContainerNElems(container_name  varchar(128,1))
returns bigint;
```

container_name
The name of an existing container. Can be NULL.

Description

Use the **TSContainerNElems** function to view the number of elements stored in a container. If you specify NULL as the container name, information about all containers for the database is returned.

Returns

The number of elements stored in the specified container or in all containers.

Example

The following statement returns the number of elements stored in the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerNElems("mult_container");

           elements

                 26

1 row(s) retrieved.
```

The following statement returns the number of elements stored in all containers:

```
EXECUTE FUNCTION TSContainerNElems(NULL);

           elements

             241907

1 row(s) retrieved.
```

Related tasks:

"Monitoring time series containers" on page 3-8

TSContainerPctUsed function

The **TSContainerPctUsed** function returns the percentage of space used in the specified container or in all containers.

Syntax

```
TSContainerPctUsed(container_name  varchar(128,1))
returns decimal;
```

container_name
The name of an existing container. Can be NULL.

Description

Use the **TSContainerPctUsed** function to view the percentage of used space in a container. If you specify NULL as the container name, information about all containers for the database is returned.

Returns

The percentage of used space in the specified container or in all containers.

Example

The following statement returns the percentage of used space in the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerPctUsed("mult_container");

 percent

  60.000

1 row(s) retrieved.
```

The following statement returns the percentage of used space in all containers:

```
EXECUTE FUNCTION TSContainerPctUsed(NULL);

 percent

  93.545

1 row(s) retrieved.
```

Related tasks:

"Monitoring time series containers" on page 3-8

TSContainerPoolRoundRobin function

The **TSContainerPoolRoundRobin** function provides a round-robin policy for inserting time series data into containers in the specified container pool.

Syntax

```
TSContainerPoolRoundRobin(
      table_name lvarchar,
      column_name lvarchar,
      subtype lvarchar,
      irregular integer,
      pool_name lvarchar)
returns lvarchar;
```

table_name
The table into which the time series data is being inserted.

column_name
The name of the time series column into which data is being inserted.

subtype
The name of the **TimeSeries** subtype.

irregular
Whether the time series is regular (0) or irregular (1).

pool_name
The name of the container pool.

Description

Use the **TSContainerPoolRoundRobin** function to select containers in which to insert time series data from the specified container pool. The container pool must exist before you can insert data into it, and at least one container within the container pool must be configured for the same **TimeSeries** subtype as used by the data being inserted. Set the **TSContainerPoolRoundRobin** function to a container pool name and use it as the value for the **container** argument in the VALUES clause of an INSERT statement. The **TSContainerPoolRoundRobin** function returns container names to the INSERT statements in round-robin order.

Returns

The container name in which to store the time series value.

Example

The following statement inserts data into a time series. The **TSContainerPoolRoundRobin** function specifies that the container pool named **readings** is used in the **container** argument.

```
INSERT INTO smartmeters(meter_id,rawreadings)
     VALUES('met00001','origin(2006-01-01 00:00:00.00000),
            calendar(smartmeter),regular,threshold(0),
            container(TSContainerPoolRoundRobin(readings)),
                [(33070,-13.00,100.00,9.98e+34),
                 (19347,-4.00,100.00,1.007e+35),
                 (17782,-18.00,100.00,9.83e+34)]');
```

When the INSERT statement runs, the **TSContainerPoolRoundRobin** function runs with the following values:

```
TSContainerPoolRoundRobin('smartmeters','rawreadings',
                          'smartmeter_row',0,'readings')
```

The **TSContainerPoolRoundRobin** function sorts the container names alphabetically and returns the first container name to the INSERT statement. The next time an INSERT statement is run, the **TSContainerPoolRoundRobin** function returns the second container name, and so on.

Related tasks:

"Configuring additional container pools" on page 3-9

Related reference:

"User-defined container pool policy" on page 3-10

TSContainerSetPool procedure

The **TSContainerSetPool** procedure moves the specified container into the specified container pool.

Syntax

```
TSContainerSetPool(
      container_name varchar(128,1),
      pool_name varchar(128,1) default null);

TSContainerSetPool(
      container_name varchar(128,1));
```

container_name
The name of the container to move.

pool_name
The name of the container pool in which to move the container.

Description

You can use the **TSContainerSetPool** procedure to move a container into a container pool, move a container from one container pool to another, or remove a container from a container pool. Containers created automatically are in the container pool named **autopool** by default. If you create a container with the **TSContainerCreate** procedure, the container does not belong to a container pool until you run the **TSContainerSetPool** procedure to move it into a container pool.

If the container pool specified in the **TSContainerSetPool** procedure does not exist, the procedure creates it.

To move a container from one container pool to another, run the **TSContainerSetPool** procedure and specify the destination container pool name.

To move a container out of a container pool, run the **TSContainerSetPool** procedure without a container pool name.

The **TSContainerTable** table contains a row for each container and the container pool to which the container belongs.

Examples

Example 1: Move a container into a container pool

The following statement moves a container named **ctn_1** into a container pool named **smartmeter_pool**:

```
EXECUTE PROCEDURE TSContainerSetPool
   ('ctn_1', 'smartmeter_pool');
```

Example 2: Remove a container from a container pool

The following statement removes a container named **ctn_1** from its container pool:

```
EXECUTE PROCEDURE TSContainerSetPool
   ('ctn_1');
```

Related concepts:

"TSInstanceTable table" on page 2-8

Related tasks:

"Managing containers" on page 3-7

"Configuring additional container pools" on page 3-9

TSContainerTotalPages function

The **TSContainerTotalPages** function returns the total number of pages allocated to the specified container or in all containers.

Syntax

```
TSContainerTotalPages(container_name  varchar(128,1))
returns integer;
```

container_name
: The name of an existing container. Can be NULL.

Description

Use the **TSContainerTotalPages** function to view the size of a container. If you specify NULL as the container name, information about all containers for the database is returned.

Returns

The number of pages that are allocated to the specified container or that are allocated to all containers.

Example

The following statement returns the number of pages allocated to the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerTotalPages("mult_container");

      total

         50

1 row(s) retrieved.
```

The following statement returns the number of pages allocated to all the containers:

```
EXECUTE FUNCTION TSContainerTotalPages(NULL);

      total

       2169

1 row(s) retrieved.
```

Related tasks:

"Monitoring time series containers" on page 3-8

TSContainerTotalUsed function

The **TSContainerTotalUsed** function returns the total number of pages containing time series data in the specified container or in all containers.

Syntax

```
TSContainerTotalUsed(container_name  varchar(128,1))
returns integer;
```

container_name
The name of an existing container. Can be NULL.

Description

Use the **TSContainerTotalUsed** function to view the amount of data in a container. If you specify NULL as the container name, information about all containers for the database is returned.

Returns

The number of pages containing time series data in the specified container or in all containers.

Example

The following statement returns the number of pages used by time series data in the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerTotalUsed("mult_container");

      pages

         30

1 row(s) retrieved.
```

The following statement returns the number of pages used by time series data in all containers:

```
EXECUTE FUNCTION TSContainerTotalUsed(NULL);

      pages

       2029

1 row(s) retrieved.
```

Related tasks:

"Monitoring time series containers" on page 3-8

TSContainerUsage function

The **TSContainerUsage** function returns information about the size and capacity of the specified container or of all containers.

Syntax

```
TSContainerUsage(container_name  varchar(128,1))
returns integer, bigint, integer;
```

container_name
The name of an existing container. Can be NULL.

Description

Use the **TSContainerUsage** function to monitor how full the specified container is. If you specify NULL as the container name, information about all containers for the database is returned. You can use the information from this function to determine how quickly your containers are filling and whether you need to allocate additional storage space.

Returns

The number of pages containing time series data in the `pages` column, the number of elements in the `slots` column, and the number of pages allocated to the container in the `total` column, for the specified container or for all containers.

Example

The following statement returns the information for the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerUsage("mult_container");

      pages                slots       total

         30                   26          50

1 row(s) retrieved.
```

This container has 26 time series data elements using 30 pages out of the total 50 pages of space. Although the container is almost half empty, the container can probably accommodate fewer than 20 additional time series elements.

The following statement returns the information for all containers:

```
EXECUTE FUNCTION TSContainerUsage(NULL);

      pages              slots       total

       2029             241907        2169

1 row(s) retrieved.
```

The containers have only 140 pages of available space.

Related tasks:

"Monitoring time series containers" on page 3-8

TSCreate function

The **TSCreate** function creates an empty regular time series or a regular time series populated with the given set of data. The new time series can also have user-defined metadata attached to it.

Syntax

```
TSCreate(cal_name       lvarchar,
         origin         datetime year to fraction(5),
         threshold      integer,
         zero           integer,
         nelems         integer,
         container_name  lvarchar)
returns TimeSeries with (handlesnulls);

TSCreate(cal_name       lvarchar,
         origin         datetime year to fraction(5),
         threshold      integer,
         zero           integer,
         nelems         integer,
         container_name lvarchar,
         set_rows       set)
returns TimeSeries with (handlesnulls);

TSCreate(cal_name       lvarchar,
         origin         datetime year to fraction(5),
         threshold      integer,
         zero           integer,
         nelems         integer,
         container_name lvarchar,
         metadata       TimeSeriesMeta)
returns TimeSeries with (handlesnulls);

TSCreate(cal_name       lvarchar,
         origin         datetime year to fraction(5),
         threshold      integer,
         zero           integer,
         nelems         integer,
         container_name lvarchar,
         metadata       TimeSeriesMeta,
         set_rows       set)
returns TimeSeries with (handlesnulls);
```

cal_name

The name of the calendar for the time series.

origin The origin of the time series. This is the first valid date from the calendar for which data can be stored in the series.

threshold
: The threshold for the time series. If the time series stores more than this number of elements, it is converted to a container. Otherwise, it is stored directly in the row that contains it, not in a container. The default is `20`. The size of a row containing an in-row time series should not exceed 1500 bytes.

  If a time series has too many bytes to fit in a row before this threshold is reached, the time series is put into a container at that point.

zero Must be `0`.

nelems The number of elements allocated for the resultant time series. If the number of elements exceeds this value, the time series is expanded through reallocation.

container_name
: The name of the container used to store the time series. Can be `NULL`.

metadata
: The user-defined metadata to be put into the time series. See "Creating a time series with metadata" on page 3-13 for more information about metadata.

set_rows
: A set of row type values used to populate the time series. The type of these rows must be the same as the subtype of the time series.

Description

If **TSCreate** is called with a *metadata* argument, then the metadata is saved in the time series.

See "Creating a time series with the TSCreate or TSCreateIrr function" on page 3-12 for a description of how to use this function.

Returns

A regular time series that is empty or populated with the given set and optionally contains user-defined metadata.

Example

The following example creates an empty time series using **TSCreate**:

```
insert into daily_stocks values(
   901,'IBM', TSCreate('daycal',
      '2011-01-03 00:00:00.00000',20,0,0, NULL));
```

The following example creates a populated regular time series using **TSCreate**:

```
select TSCreate('daycal',
      '2011-01-05 00:00:00.00000',
      20,
      0,
      NULL,
      set_data)::TimeSeries(stock_trade)
    from activity_load_tab
    where stock_id = 600;
```

Related tasks:

"Creating a time series with the TSCreate or TSCreateIrr function" on page 3-12
"Creating a time series with metadata" on page 3-13

Related reference:

"GetCalendar function" on page 7-38
"GetInterval function" on page 7-43
"GetMetaData function" on page 7-47
"GetMetaTypeName function" on page 7-47
"GetOrigin function" on page 7-52
"PutElem function" on page 7-65
"PutSet function" on page 7-69
"GetClosestElem function" on page 7-39
"TSCreateIrr function"
"UpdMetaData function" on page 7-114
"The ts_create() function" on page 9-16
"The ts_create_with_metadata() function" on page 9-17
"The ts_get_metadata() function" on page 9-28
"The ts_update_metadata() function" on page 9-52

TSCreateIrr function

The **TSCreateIrr** function creates an empty irregular time series or an irregular time series populated with the given multiset of data. The new time series can also have user-defined metadata attached to it.

Syntax

```
TSCreateIrr(cal_name    lvarchar,
        origin         datetime year to fraction(5),
        threshold      integer,
        zero           integer,
        nelems         integer,
        container_name lvarchar)
returns TimeSeries with (handlesnulls);

TSCreateIrr(cal_name    lvarchar,
        origin         datetime year to fraction(5),
        threshold      integer,
        zero           integer,
        nelems         integer,
        container_name lvarchar,
        multiset_rows   multiset)
returns TimeSeries with (handlesnulls);

TSCreateIrr(cal_name    lvarchar,
        origin         datetime year to fraction(5),
        threshold      integer,
        zero           integer,
        nelems         integer,
        container_name lvarchar,
        metadata       TimeSeriesMeta)
returns TimeSeries with (handlesnulls);

TSCreateIrr(cal_name    lvarchar,
        origin         datetime year to fraction(5),
        threshold      integer,
        zero           integer,
```

```
        nelems         integer,
        container_name lvarchar,
        metadata       TimeSeriesMeta,
        multiset_rows   multiset)
returns TimeSeries with (handlesnulls);
```

cal_name
The name of the calendar for the time series.

origin The origin of the time series. This is the first valid date from the calendar for which data can be stored in the series.

threshold
The threshold for the time series. If the time series stores more than this number of elements, it is converted to a container. Otherwise, it is stored directly in the row that contains it. The default is `20`. The size of a row containing an in-row time series should not exceed 1500 bytes.

If a time series has too many bytes to fit in a row before this threshold is reached, the time series is put into a container.

zero Must be `0`.

nelems The number of elements allocated for the resultant time series. If the number of elements exceeds this value, the time series is expanded through reallocation.

container_name
The name of the container used to store the time series. Can be `NULL`.

metadata
The user-defined metadata to be put into the time series. See "Creating a time series with metadata" on page 3-13 for more information about metadata.

multiset_rows
A multiset of rows used to populate the time series. The type of these rows must be the same as the subtype of the time series.

Description

If **TSCreateIrr** is called with the *metadata* argument, then metadata is saved in the time series.

See "Creating a time series with the TSCreate or TSCreateIrr function" on page 3-12 for a description of how to use this function.

Returns

An irregular time series that is empty or populated with the given multiset and optionally contains user-defined metadata.

Example

The following example creates an empty irregular time series using **TSCreateIrr**:

```
select TSCreateIrr('daycal',
      '2011-01-05 00:00:00.00000',
      20,
      0,
```

```
      NULL,
      set_data)::TimeSeries(stock_trade)
    from activity_load_tab
    where stock_id = 600;
```

The following example creates a populated irregular time series using **TSCreateIrr**:

```
insert into activity_stocks
    select 1234,
      TSCreateIrr('daycal',
         '2011-01-03 00:00:00.00000'::datetime year to fraction(5),
         20, 0, NULL,
         set_data)::timeseries(stock_trade)
    from activity_load_tab;
```

Related tasks:

"Creating a time series with the TSCreate or TSCreateIrr function" on page 3-12

"Creating a time series with metadata" on page 3-13

Related reference:

"GetMetaData function" on page 7-47

"GetMetaTypeName function" on page 7-47

"TSCreate function" on page 7-92

"GetCalendar function" on page 7-38

"GetClosestElem function" on page 7-39

"GetInterval function" on page 7-43

"GetOrigin function" on page 7-52

"UpdMetaData function" on page 7-114

"The ts_get_metadata() function" on page 9-28

"The ts_update_metadata() function" on page 9-52

TSDecay function

The **TSDecay** function computes a decay function over its arguments.

Syntax

```
TSDecay(current_value smallfloat,
        initial_value  smallfloat,
        decay_factor   smallfloat)
returns smallfloat;

TSDecay(current_value double precision,
        initial_value  double precision,
        decay_factor   double precision)
returns double precision;
```

current_value
: The current datum (v_j in the sum shown next).

initial_value
: The initial value (*initial* in the sum shown next).

decay_factor
: The decay factor (*decay* in the sum shown next).

Description

All three arguments must be of the same type.

The function maintains a sum of all the arguments it has been called with so far. Every time it is called, the sum is multiplied by the supplied decay factor. Given a decay factor between 0 and 1, this causes the importance of older arguments to fall off over time. The first time that **TSDecay** is called, it includes the supplied initial value in the running sum. The actual function that **TSDecay** computes is:

$$((decay^{i})initial)+\sum_{j=1}^{i}((v_j)decay^{i-j})$$

In this computation, `i` is the number of times the function has been called so far, and v_j is the value it was called with in its `j`th invocation.

This function is useful only when used within the **Apply** function.

Returns

The result of the decay function.

Example

The following example computes the decay:

```
create function ESA18(a smallfloat) returns smallfloat;
return (.18 * a) + TSDecay(.18 * a, a, .82);
end function;
```

Related reference:

"Apply function" on page 7-11
"TSAddPrevious function" on page 7-80
"TSCmp function" on page 7-81
"TSPrevious function"
"TSRunningAvg function" on page 7-102
"TSRunningSum function" on page 7-105

TSPrevious function

The **TSPrevious** function records the supplied argument and returns the last argument it was passed.

Syntax

```
TSPrevious(value int)
returns int;

TSPrevious(value smallfloat)
returns smallfloat;

TSPrevious(value double precision)
returns double precision;
```

value The value to save.

Description

This function is useful in comparing a value in a time series with the value immediately preceding it.

This function is useful only when used within the **Apply** function.

Returns

The value previously saved. The first time **TSPrevious** is called, it returns NULL.

Example

See the example for "TSCmp function" on page 7-81.

Related reference:

"Apply function" on page 7-11

"TSAddPrevious function" on page 7-80

"TSCmp function" on page 7-81

"TSDecay function" on page 7-96

"TSRunningAvg function" on page 7-102

"TSRunningSum function" on page 7-105

TSRollup function

The **TSRollup** function aggregates time series values by time for multiple rows in the table.

Syntax

```
TSRollup(ts TimeSeries, 'agg_express' lvarchar )
RETURNS TimeSeries;
```

agg_express
: A comma-separated list of these SQL aggregate operators: AVG, COUNT, MIN, MAX, SUM. Each operator requires an argument that is the name of the column in a **TimeSeries** data type, prefixed by a $.

ts
: The name of the **TimeSeries** data type or a function that returns a **TimeSeries** data type, such as **AggregateBy**.

Description

Use the **TSRollup** function to run one or more aggregate operators on multiple rows of time series data in a table.

Returns

A **TimeSeries** data type that is the result of the expression or expressions.

Examples

The following examples show how to use the **TSRollup** function.

Example 1: Sum of all electricity usage in a zipcode

The following statement adds all the electricity usage values for each time stamp in the **ts_data** table in the **stores_demo** database for the customers that have a zipcode of 94063:

```
SELECT TSRollup(raw_reads, "sum($value)")
    FROM ts_data, customer, customer_ts_data
    WHERE customer.zipcode = "94063"
       AND customer_ts_data.customer_num = customer.customer_num
       AND customer_ts_data.loc_esi_id = ts_data.loc_esi_id;
```

Example 2: Sum of daily electricity usage by zipcode

Suppose you have a table named **ts_table** that contains a user ID, the zipcode of the user, and the electricity usage data for each customer, which is collected every 15 minutes and stored in a column named **value** in a time series named **ts_col**. The following query returns the total amounts of electricity used daily for each zipcode:

```
SELECT zipcode,
  TSRollup(
 AggregateBy('SUM($value)', 'cal1day', ts_col, 0,
     '2011-01-01 00:00:00.00000', '2011-01-31 23:45:00:00.00000'),
      'SUM($value)'
  )
FROM ts_table
GROUP BY zipcode;
```

The first argument to the **TSRollup** function is an **AggregateBy** function, which sums the electricity usage for each customer for each day of January 2011. The second argument is a SUM operator that sums the daily electricity usage by zipcode.

The resulting table contains a row for each zipcode. Each row has a time series that contains the sum of the electricity used by customers living in that zipcode for each day in January 2011.

Related reference:

"AggregateBy function" on page 7-6

TSRowNameToList function

The **TSRowNameToList** function returns a list (collection of rows) containing one individual column from a time series column plus the non-time-series columns of a table. Null elements are not added to the list.

Syntax

```
TSRowNameToList(ts_row     row,
                colname     lvarchar)
returns list (row not null)
```

ts_row The time series to act on.

colname
The time series column to return.

Description

The **TSRowNameToList** function can only be used on rows with one **TimeSeries** column.

You must cast the return variable to match the names and types of the columns being returned exactly.

Returns

A list (collection of rows).

Example

The query returns a list of rows, each containing the **ID** and **high** columns.

```
select
   TSRowNameToList(d, 'high')::list(
               row(id integer, name lvarchar, high real) not null)
   from daily_stocks d;
```

Related reference:

"TSColNameToList function" on page 7-82

"TSColNumToList function" on page 7-83

"Transpose function" on page 7-77

"TSRowNumToList function"

"TSRowToList function" on page 7-101

"TSSetToList function" on page 7-107

TSRowNumToList function

The **TSRowNumToList** function returns a list (collection of rows) containing one individual column from a time series column plus the non-time-series columns of a table. Null elements are not added to the list.

Syntax

```
TSRowNumToList(ts_row     row,
               colnum     integer)
returns list (row not null)
```

ts_row The time series to act on.

colnum The number of the time series column to return.

Description

The **TSRowNumToList** function can only be used on rows with one **TimeSeries** column.

The column is specified by its number; column numbering starts at 1, with the first column following the time stamp column.

You must cast the return variable to match the names and types of the columns being returned exactly.

Returns

A list (collection of rows).

Example

The query returns a list of rows, each containing the **ID** , **name**, and **high** columns.

```
select
   TSRowNumToList(d, 1)::list(row
   (id integer, name lvarchar, high real) not null)
   from daily_stocks d;
```

Related reference:
"TSColNameToList function" on page 7-82
"TSColNumToList function" on page 7-83
"TSRowNameToList function" on page 7-99
"Transpose function" on page 7-77
"TSRowToList function"
"TSSetToList function" on page 7-107

TSRowToList function

The **TSRowToList** function returns a list (collection of rows) containing the individual columns from a time series column plus the non-time-series columns of a table. Null elements are not added to the list.

Syntax

```
TSRowToList(ts_row     row)
returns list (row not null)
```

ts_row A row value that contains a time series as one of its columns.

Description

The **TSRowToList** function can only be used on rows with one **TimeSeries** column.

You must cast the return variable to match the names and types of the columns being returned exactly.

Returns

A list (collection of rows).

Example

The query returns a list of rows, each containing the following columns: **stock_id**, **stock_name**, **t**, **high**, **low**, **final**, **vol**.

```
select TSRowToList(d)::list(row(stock_id integer,
                                 stock_name lvarchar,
                                 t datetime year to fraction(5),
                                 high real,
                                 low real,
                                 final real,
                                 vol real) not null)
   from daily_stocks d;
```

Related reference:
"TSRowNameToList function" on page 7-99
"TSRowNumToList function" on page 7-100
"Transpose function" on page 7-77
"TSColNameToList function" on page 7-82
"TSColNumToList function" on page 7-83
"TSSetToList function" on page 7-107

TSRunningAvg function

The **TSRunningAvg** function computes a running average over SMALLFLOAT or DOUBLE PRECISION values.

Syntax

```
TSRunningAvg(value      double precision,
             num_values integer)
returns double precision;

TSRunningAvg(value      real,
             num_values integer)
returns double precision;
```

value The value to include in the running average.

num_values
The number of values to include in the running average, *k*.

Description

A running average is the average of the last *k* values, where *k* is supplied by the user. If a value is NULL, the previous value is used. The running average for the first *k*-1 values is NULL.

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

This function is useful only when used within the **Apply** function.

Returns

A SMALLFLOAT or DOUBLE PRECISION running average of the last *k* values.

Example

The SELECT query in the following example gets the closing price and the 30-day moving average from the stocks in the time series:

```
select stock_name, Apply('TSRunningAvg($final,30)',
   '2010-01-01 00:00:00.00000'::datetime year to fraction(5),
   '2011-01-01 00:00:00.00000'::datetime year to fraction(5),
   stock_data::TimeSeries(stock_bar))::
      TimeSeries(one_real)
from daily_stocks;
```

Related reference:

"Apply function" on page 7-11
"TSAddPrevious function" on page 7-80
"TSCmp function" on page 7-81
"TSDecay function" on page 7-96
"TSPrevious function" on page 7-97
"TSRunningSum function" on page 7-105
"TSRunningCor function"
"TSRunningMed function" on page 7-104
"TSRunningVar function" on page 7-106

TSRunningCor function

The **TSRunningCor** function computes the running correlation of two time series over a running window. The **TSRunningCor** function returns NULL if the variance of either input is zero or NULL over the window.

Syntax

```
TSRunningCor(value1     double precision,
             value2     double precision,
              num_values integer)
returns double precision;

TSRunningCor(value1     real,
             value2     real,
              num_values integer)
returns double precision;
```

value1 The column of the first time series to use to calculate the running correlation.

value2 The column of the second time series to use to calculate the running correlation.

num_values
The number of values to include in the running correlation, *k*.

Description

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

The first (num_values - 1) outputs result from shorter windows (the first output is derived from the first input time, the second output is derived from the first two input times, and so on). Null elements in the input also result in shortened windows.

This function is useful only when used within the **Apply** function.

Returns

A DOUBLE PRECISION running correlation of the last *k* values.

Example

This statement finds the running correlation between stock data for IBM and AT&T over a 20 element window. Again, the first 19 output elements are exceptions because they result from windows of fewer than 20 elements. The first is NULL because correlation is undefined for just one element.

```
select Apply('TSRunningCor($0.high, $1.high, 20)',
                ds1.stock_data::TimeSeries(stock_bar),
                ds1.stock_data::TimeSeries(stock_bar))::TimeSeries(one_real)
from daily_stocks ds1, daily_stocks ds2
where ds1.stock_name = 'IBM'
and ds2.stock_name = 'HWP';
```

Tip: When a start date is supplied to the **Apply** function, the first (num_values - 1) output elements are still formed from incomplete windows. The **Apply** function never looks at data before the specified start date.

Related reference:

"Apply function" on page 7-11

"TSRunningAvg function" on page 7-102

"TSRunningMed function"

"TSRunningSum function" on page 7-105

"TSRunningVar function" on page 7-106

TSRunningMed function

The **TSRunningMed** function computes the median of a time series over a running window. This function is useful only when used within the **Apply** function.

Syntax

```
TSRunningMed(value      double precision,
             num_values integer)
returns double precision;

TSRunningMed(value      real,
             num_values integer)
returns double precision;
```

value The first input value to use to calculate the running median. Typically, the name of a DOUBLE, FLOAT, or REAL column in your time series.

num_values
The number of values to include in the running median, *k*.

Description

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

The first (num_values - 1) outputs result from shorter windows (the first output is derived from the first input time, the second output is derived from the first two input times, and so on). Null elements in the input also result in shortened windows.

Returns

A DOUBLE PRECISION running median of the last *k* values.

Example

This statement produces a time series from the running median over a 10-element window of the column **high** of *stock_data*. You can refer to the columns of a time series as **$colname** or **$colnumber**: for example, **$high**, or **$1**.

```
select stock_name, Apply('TSRunningMed($high, 10)',
                stock_data::TimeSeries(stock_bar))::
      TimeSeries(one_real)
from daily_stocks;
```

Related reference:

"TSRunningCor function" on page 7-103

"Apply function" on page 7-11

"TSRunningAvg function" on page 7-102

"TSRunningSum function"

"TSRunningVar function" on page 7-106

TSRunningSum function

The **TSRunningSum** function computes a running sum over SMALLFLOAT or DOUBLE PRECISION values.

Syntax

```
TSRunningSum(value      smallfloat,
             num_values integer)
returns smallfloat;

TSRunningSum(value      double precision,
             num_values integer)
returns double precision;
```

value The input value to include in the running sum.

num_values
The number of values to include in the running sum, *k*.

Description

A running sum is the sum of the last *k* values, where *k* is supplied by the user. If a value is `NULL`, the previous value is used.

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

This function is useful only when used within the **Apply** function.

Returns

A SMALLFLOAT or DOUBLE PRECISION running sum of the last *k* values.

Example

The following function calculates the *volume accumulation percentage*. The columns represented by **a** through **e** are: **high**, **low**, **close**, **volume**, and **number_of_days**, respectively:

```
create function VAP(a float, b float,c float,d float, e int) returns int;
return cast(100 * TSRunningSum(d * ((c - b) - (a - c))/
(.0001 + a - b), e)/(.0001 + TSRunningSum(d,e)) as int);
end function;
```

Related reference:

"Apply function" on page 7-11

"TSAddPrevious function" on page 7-80

"TSCmp function" on page 7-81

"TSDecay function" on page 7-96

"TSPrevious function" on page 7-97

"TSRunningAvg function" on page 7-102

"TSRunningCor function" on page 7-103

"TSRunningMed function" on page 7-104

"TSRunningVar function"

TSRunningVar function

The **TSRunningVar** function computes the variance of a time series over a running window.

Syntax

```
TSRunningVar(value      double precision,
             num_values integer)
returns double precision;

TSRunningVar(value      real,
             num_values integer)
returns double precision;
```

value The first input value to use to calculate the running correlation.

num_values
The number of values to include in the running variance, *k*.

Description

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

The first (`num_values - 1`) outputs are exceptions because they result from shorter windows (the first output is derived from the first input time, the second output is derived from the first two input times, and so on). Null elements in the input also result in shortened windows.

This function is useful only when used within the **Apply** function.

Returns

A DOUBLE PRECISION running variance of the last *k* values.

Example

This statement produces a time series with the same length and calendar as *stock_data* but with one data column other than the time stamp. Element `n` of the output is the variance of column 1 of `stock_bar` elements `n-19`, `n-18`, `...` `n`. The first 19 elements of the output are a bit different: the first element is NULL, because variance is undefined for a series of 1. The second output element is the variance of the first two input elements, and so on.

If element i of *stock_data* is NULL, or if column 1 of element i of *stock_data* is NULL, output elements i, i + 1, ... i + 19, are variances of just 19 numbers (assuming that there are no other null values in the input window).

```
select stock_name, Apply('TSRunningVar($0.high, 20)',
                stock_data::TimeSeries(stock_bar))::
                TimeSeries(one_real)
from daily_stocks;
```

Related reference:

"Apply function" on page 7-11

"TSRunningAvg function" on page 7-102

"TSRunningCor function" on page 7-103

"TSRunningMed function" on page 7-104

"TSRunningSum function" on page 7-105

TSSetToList function

The **TSSetToList** function takes a **TimeSeries** column and returns a list (collection of rows) containing all the elements in the time series. Null elements are not added to the list.

Syntax

```
TSSetToList(ts     TimeSeries)
returns list (row not null)
```

ts The time series to act on.

Description

Because this aggregate function can return rows of any type, the return value must be explicitly cast at runtime.

Returns

A list (collection of rows).

Example

The following query collects all the elements in all the time series in the **stock_data** column into a list and then selects out the **high** column from each element.

```
select high from table((select
   TsSetToList(stock_data)::list(stock_bar
   not null) from daily_stocks));
```

Related reference:
"TSColNameToList function" on page 7-82
"TSColNumToList function" on page 7-83
"TSRowNameToList function" on page 7-99
"TSRowNumToList function" on page 7-100
"Transpose function" on page 7-77
"TSRowToList function" on page 7-101

TSToXML function

The **TSToXML** function returns an XML representation of a time series.

Syntax

```
TSToXML(doctype    lvarchar,
        id         lvarchar,
        ts         timeseries,
        output_max integer default 0)
returns lvarchar;

TSToXML(doctype    lvarchar,
        id         lvarchar,
        ts         timeseries)
returns lvarchar;
```

doctype
The name of the topmost XML element.

id The primary key value in the time series table that uniquely identifies the time series.

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

output_max
The maximum size, in bytes, of the XML output. If the parameter is absent, the default value is 32 768. The following table describes the results for each possible value of the *output_max* parameter.

Value	Result
no value	32 768 bytes
negative integer	2^{32}-1 bytes
1 through 4096	4096 bytes
4096 through 2^{32}-1	the specified number of bytes

Description

Use the **TSToXML** function to provide a standard representation for information exchange in XML format for small amounts of data.

The top-level tag in the XML output is the first argument to the **TSToXML** function.

The `id` tag must uniquely identify the time series and refer the XML output to the row on which it is based.

The `AllData` tag indicates whether all the data was returned or the data was truncated because it exceeded the size set by the *output_max* parameter.

The remaining XML tags represent the TimeSeries subtype and its columns, including the time stamp.

The special characters <, >, &, ', and " are replaced by their XML predefined entities.

Returns

The specified time series in XML format, up to the size set by the *output_max* parameter. The `AllData` tag indicates whether all the data was returned (1) or whether the data was truncated (0).

Example

The following query selects the time series data for one hour by using the **Clip** function from the **TimeSeries** subtype named **actual** to return in XML format:

```
SELECT TSToXML('meterdata', esi_id,
       Clip(actual, '2010-09-08 12:00:00'::datetime year to second,
                    '2010-09-08 13:00:00'::datetime year to second ) )
FROM ts_data
WHERE esi_id = '2250561334';
```

The following XML data is returned:

```
<meterdata>
  <id>2250561334</id>
  <AllData>1</AllData>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>0.9170000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>0.4610000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>4.1570000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>6.3280000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>2.6690000000</value>
  </meter_data>
</meterdata>
```

The name of the TimeSeries subtype is `meter_data` and its columns are `tstamp` and `value`.

The value of 1 in the `AllData` tag indicates that for this example, all data was returned.

Related concepts:

"Planning for accessing time series data" on page 1-12

Unary arithmetic functions

The standard unary functions **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan** are extended to operate on time series.

Syntax

```
Function(ts TimeSeries)
returns TimeSeries;
```

ts The time series to act on.

Description

The resulting time series has the same regularity, calendar, and sequence of time stamps as the input time series. It is derived by applying the function to each element of the input time series.

If there is a variant of the function that operates directly on the input element type, then that variant is applied to each element. Otherwise, the function is applied to each non-time stamp column of the input time series.

Returns

The same type of time series as the input; unless it is cast, then it returns the type of time series to which it is cast.

Example

The following query converts the daily stock price and volume data into log space:

```
create table log_stock (stock_id int, data TimeSeries(stock_bar));
insert into log_stock
   select stock_id, Logn(stock_data)
      from daily_stocks;
```

Related reference:
"Abs function" on page 7-6
"Acos function" on page 7-6
"ApplyUnaryTsOp function" on page 7-19
"Asin function" on page 7-20
"Atan function" on page 7-21
"Binary arithmetic functions" on page 7-21
"Cos function" on page 7-31
"Exp function" on page 7-37
"Logn function" on page 7-62
"Negate function" on page 7-63
"Positive function" on page 7-65
"Round function" on page 7-74
"Sin function" on page 7-75
"Sqrt function" on page 7-75
"Tan function" on page 7-76
"Apply function" on page 7-11

Union function

The **Union** function performs a union of multiple time series, either over the entire length of each time series or over a clipped portion of each time series.

Syntax

```
Union(ts TimeSeries,...)
returns TimeSeries;

Union(set_ts set(TimeSeries))
returns TimeSeries;

Union(begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5),
       ts          TimeSeries,...)
returns TimeSeries;

Union(begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5),
       set_ts      set(TimeSeries))
returns TimeSeries;
```

ts The time series that form the union. **Union** can take from two to eight time series arguments.

set_ts A set of time series.

begin_stamp
The begin point of the clip.

end_stamp
The end point of the clip.

Description

The second and fourth forms of the function perform a union of a set of time series. The resulting time series has one DATETIME YEAR TO FRACTION(5) column, followed by each column in each time series, in order. When using the second or fourth form, it is important to ensure that the order of the time series in the set is deterministic so that the elements remain in the correct order.

Since the type of the resulting time series is different from that of the input time series, the result of the union must be cast.

Union can be thought of as an outer join on the time stamp.

In a union, the resulting time series has a calendar that is the combination of the calendars of the input time series with the OR operator. The resulting calendar is stored in the **CalendarTable** table. The name of the resulting calendar is a string containing the names of the calendars of the input time series, separated by a vertical bar (|). For example, if two time series are combined, and **mycal** and **yourcal** are the names of their corresponding calendars, the resulting calendar is named **mycal|yourcal**. If all the time series have the same calendar, then **Union** does not create a new calendar.

For a regular time series, if a time series does not have a valid element at a timepoint of the resulting calendar, the value for that time series element is NULL.

To be certain of the order of the columns in the resultant time series when using **Union** over a set, use the ORDER BY clause.

For the purposes of **Union**, the value at a given timepoint is that of the most recent valid element. For regular time series, this is the value corresponding to the current interval, which can be NULL; it is not necessarily the most recent non-null value. For irregular time series, this condition never occurs since irregular time series do not have null intervals.

For example, consider the union of two irregular time series, one containing bid prices for a certain stock, and one containing asking prices. The union of the two time series contains bid and ask values for each timepoint at which a price was either bid or asked. Now consider a timepoint at which a bid was made but no price was asked. The union at that timepoint contains the bid price offered at that timepoint, along with the most recent asking price.

If an intersection involves one or more regular time series, the resulting time series starts at the latest of the start points of the input time series and ends at the earliest of the end points of the regular input time series. If all the input time series are irregular, the resulting irregular time series starts at the latest of the start points of the input time series and ends at the latest of the end points. If a union involves one or more time series, the resulting time series starts at the first of the start points of the input time series and ends at the latest of the end points of the input time series. Other than this difference in start and end points, and of the resulting calendar, there is no difference between union and intersection involving time series.

Apply also combines multiple time series into a single time series. Therefore, using **Union** within **Apply** is often unnecessary.

Returns

The time series that results from the union.

Example

The following query constructs the union of time series for two different stocks:

```
select Union(s1.stock_data,
   s2.stock_data)::TimeSeries(stock_bar_union)
from daily_stocks s1, daily_stocks s2
where s1.stock_name = 'IBM' and s2.stock_name = 'HWP';
```

The following example finds the union of two time series and returns data only for time stamps between 2011-01-03 and 2011-01-05:

```
select Union('2011-01-03 00:00:00.00000'
         ::datetime year to fraction(5),
         '2011-01-05 00:00:00.00000'
         ::datetime year to fraction(5),
          s1.stock_data,
          s2.stock_data)::TimeSeries(stock_bar_union)
    from daily_stocks s1, daily_stocks s2
    where s1.stock_name = 'IBM' and s2.stock_name = 'HWP';
```

Related reference:

"Apply function" on page 7-11
"Intersect function" on page 7-59

UpdElem function

The **UpdElem** function updates an existing element in a time series.

Syntax

```
UpdElem(ts         TimeSeries,
         row_value  row,
         flags      integer default 0)
returns TimeSeries;
```

ts The time series to update.

row_value
The new row data.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

The element must be a row type of the correct type for the time series, beginning with a time stamp. If there is no element in the time series with the given time stamp, an error is raised.

Hidden elements cannot be updated.

The API equivalent of **UpdElem** is **ts_upd_elem()**.

Returns

A new time series containing the updated element.

Example

The following example updates a single element in an irregular time series:

```
update activity_stocks
set activity_data = UpdElem(activity_data,
   row('2011-01-04 12:58:09.12345', 6.75, 2000,
   2, 007, 3, 1)::stock_trade)
where stock_id = 600;
```

Related reference:

"DelElem function" on page 7-33

"GetElem function" on page 7-40

"InsElem function" on page 7-57

"PutElem function" on page 7-65

"UpdSet function" on page 7-115

"Create a custom type map" on page 8-5

"The ts_upd_elem() function" on page 9-52

UpdMetaData function

The **UpdMetaData** function updates the user-defined metadata in the specified time series.

Syntax

```
create function UpdMetaData(ts       TimeSeries,
                            metadata TimeSeriesMeta)
returns TimeSeries;
```

ts The time series for which to update metadata.

metadata
The metadata to be added to the time series. Can be NULL.

Description

This function adds the supplied user-defined metadata to the specified time series. If the *metadata* argument is NULL, then the time series is updated to contain no metadata. If it is not NULL, then the user-defined metadata is stored in the time series.

Returns

The time series updated to contain the supplied metadata, or the time series with metadata removed, if the *metadata* argument is NULL.

Related tasks:

"Creating a time series with metadata" on page 3-13

Related reference:

"GetMetaData function" on page 7-47
"GetMetaTypeName function" on page 7-47
"TSCreate function" on page 7-92
"TSCreateIrr function" on page 7-94
"The ts_create_with_metadata() function" on page 9-17
"The ts_get_metadata() function" on page 9-28
"The ts_update_metadata() function" on page 9-52

UpdSet function

The **UpdSet** function updates a set of existing elements in a time series.

Syntax

```
UpdSet(ts     TimeSeries,
       set_ts multiset,
       flags  integer default 0)
returns TimeSeries;
```

ts The time series to update.

set_ts A set of rows that replace existing elements in the given time series, *ts*.

flags Valid values for the *flags* argument are described in "The *flags* argument values" on page 7-5. The default is 0.

Description

The rows in *set_ts* must be of the correct type for the time series, beginning with a time stamp; otherwise, an error is raised. If the time stamp of any element does not correspond to an element already in the time series, an error is raised, and the entire update is void.

Hidden elements cannot be updated.

Returns

The updated time series.

Example

The following example updates elements in a time series:

```
update activity_stocks
set activity_data = (select UpdSet(activity_data, set_data)
            from activity_load_tab where stock_id = 600)
where stock_id = 600;
```

Related reference:
"DelClip function" on page 7-32
"DelTrim function" on page 7-35
"InsSet function" on page 7-58
"PutSet function" on page 7-69
"UpdElem function" on page 7-113

WithinC and WithinR functions

The **WithinC** and **WithinR** functions perform calendar-based queries, converting among time units and doing the calendar math to extract periods of interest from a time series value.

Syntax

```
WithinC(ts           TimeSeries,
         tstamp       datetime year to fraction(5),
         interval     lvarchar,
         num_intervals integer,
         direction    lvarchar)
returns TimeSeries;

WithinR(ts           TimeSeries,
         tstamp       datetime year to fraction(5),
         interval     lvarchar,
         num_intervals integer,
         direction    lvarchar)
returns TimeSeries;
```

ts The source time series.

tstamp The timepoint of interest.

interval
The name of an interval: second, minute, hour, day, week, month, or year.

num_intervals
The number of intervals to include in the output.

direction
The direction in time to include intervals. Possible values are:
- FUTURE, or F, or f
- PAST, or P, or p

Description

Every time series has a calendar that describes the active and inactive periods for the time series and how often they occur. A regular time series records one value for every active period of the calendar. Calendars can have periods of a second, a minute, an hour, a day, a week, a month, or a year. Given a time series, you might want to pose calendar-based queries on it, such as, "Show me all the values in this daily series for six years beginning on May 31, 2004," or "Show me the values in this hourly series for the week including December 27, 2010."

The **Within** functions are the primary mechanism for queries of this form. They convert among time units and do the calendar math to extract periods of interest from a time series value. There are two fundamental varieties of **Within** queries: calibrated (**WithinC**) and relative (**WithinR**).

WithinC, or within calibrated, takes a time stamp and finds the period that includes that time. Weeks have natural boundaries (Sunday through Saturday), as do years (January 1 through December 31), months (first day of the month through the last), 24-hour days, 60-minute hours, and 60-second minutes. **WithinC** allows you to specify a time stamp and find the corresponding period (or periods) that include it.

For example, July 2, 2010, fell on a Friday. Given an hourly time series, **WithinC** allows you to ask for all the hourly values in the series beginning on Sunday morning at midnight of that week and ending on Saturday night at 11:59:59. Of course, the calendar might not mark all of those hours as active; only data from active periods is returned by the **Within** functions.

WithinR, or within relative, takes a time stamp from the user and finds the period beginning or ending at that time. For example, given a weekly time series, **WithinR** can extract all the weekly values for two years beginning on June 3, 2008. **WithinR** is able to convert weeks to years and count forward or backward from the supplied date for the number of intervals requested. Relative means that you supply the exact time stamp of interest as the begin point or end point of the range.

WithinR behaves slightly differently for irregular than for regular time series. With regular time series, the time stamp argument is always mapped to a timepoint in accordance with the argument time series calendar interval. Relative offsetting is then performed starting with that point.

In irregular time series, the corresponding calendar interval does not indicate where time series elements are, and therefore offsetting begins at exactly the time stamp specified. Also, since irregular elements can appear at any point within the calendar time interval, **WithinR** returns elements with time stamps up to the last instant of the argument interval.

For example, assume an irregular time series with a daily calendar turning on all weekdays. The following function returns elements in the following interval (excluding the endpoint):

```
WithinR(stock_data, '2010-07-11 07:37:18', 'day', 3, 'future')
[2010-07-11 07:37:18, 2010-07-14 07:37:18]
```

In a regular time series, the interval is as follows, since each timepoint corresponds to the period containing the entire following day:

```
[2010-07-11 00:00:00, 2010-07-13 00:00:00]
```

Both functions take a time series, a time stamp, an interval name, a number of intervals, and a direction.

The supplied interval name is not required to be the same as the interval stored by the time series calendar, but it cannot be smaller than that interval. For example, given an hourly time series, the **Within** functions can count forward or backward for hours, days, weeks, months, or years, but not for minutes or seconds.

The direction argument indicates which periods other than the period containing the time stamp should be included; if there is only one period, the direction argument is moot.

For both **WithinC** and **WithinR**, the requested timepoint is included in the output.

Returns

A new time series with the same calendar as the original, but containing only the requested values.

Example

The following query retrieves data from the calendar week that includes Friday, January 4, 2011:

```
select WithinC(stock_data, '2011-01-04 00:00:00.00000',
      'week', 1, 'PAST')
   from daily_stocks
   where stock_name = 'IBM';
```

(expression)

```
origin(2011-01-03 00:00:00.00000),calend ar(daycal),
container(),threshold(20),re
gular,[(356.0000000000,310.0000000000,340.000000000,
999.0000000000),(156.000000
0000,110.0000000000,140.0000000000,111.0000000000), NULL,
(99.00000000000,54.000 00000000,66.00000000000,
888.0000000000)]
```

The following query returns two weeks' worth of stock trades starting on January 4, 2011, at 9:30 a.m.:

```
select WithinR(activity_data, '2011-01-04 09:30:00.00000', 'week', 2, 'future')
    from activity_stocks
    where stock_id = 600;
```

The following query returns the preceding three months' worth of stock trades:

```
select WithinR(activity_data, '2011-02-01 00:00:00.00000',
      'month', 3, 'past')
    from activity_stocks
    where stock_id = 600;
```

Related concepts:

"CalendarPattern data type" on page 2-1

Related reference:

"Clip function" on page 7-26

Chapter 8. Time series Java class library

The time series Java class library enables you to access and manipulate **TimeSeries** type data from within Java applications or applets.

The time series Java class library uses the JDBC 2.0 specification for supporting User Defined Data Types (UDTs) in Java.

When you execute a Java application that uses **TimeSeries** data, it uses IBM Informix JDBC Driver to connect to an IBM Informix database, as shown in the following figure. See your *IBM Informix JDBC Driver Programmer's Guide* for information about how to set up your Java programs to connect to Informix databases.

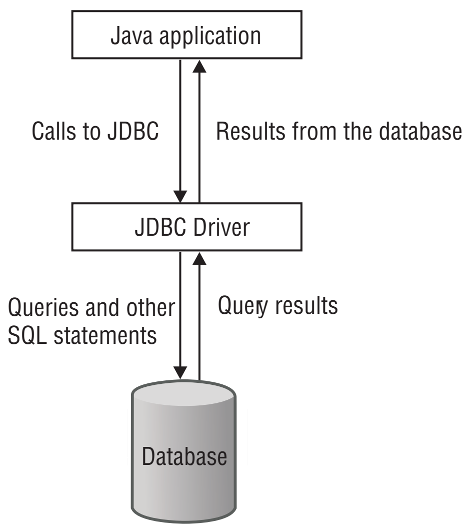


Figure 8-1. Runtime architecture for Java programs connecting to a database

You can also use the TimeSeries Java classes in Java applets and servlets, as shown in the following figures.

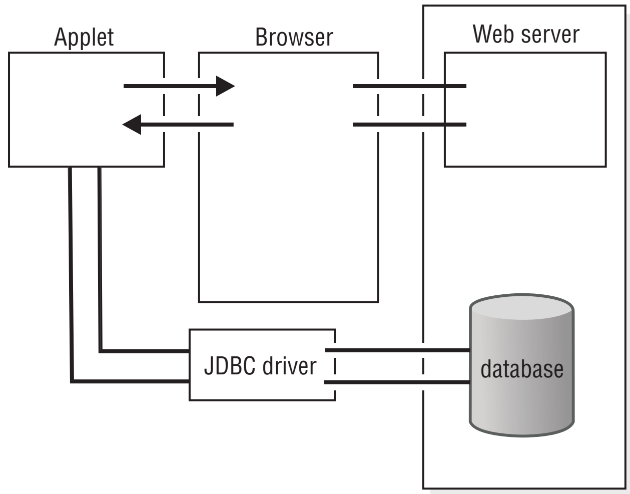


Figure 8-2. Runtime architecture for a Java applet

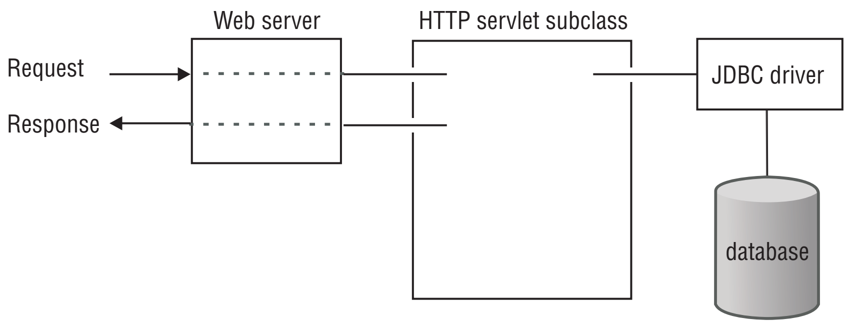


Figure 8-3. Runtime architecture for a Java servlet

Related concepts:
"Planning for accessing time series data" on page 1-12

System requirements for Java programs

Java program must use:
- Java Developers' Kit (JDK), Version 1.2.2 (also known as *Java 2*) or later
- IBM Informix JDBC Driver, Version 2.20 or later

See your *IBM Informix JDBC Driver Programmer's Guide* for information about using the Informix JDBC Driver.

Install the time series Java files

You can move the time series Java class and documentation files to a different location. You must modify your CLASSPATH variable to include the location of these files.

The Java class and documentation files are in the following directories:
- `$INFORMIXDIR/extend/TimeSeries.`*`version`*`/java/lib` contains the `.jar` files

There are two `.jar` files: `IfmxTimeSeries.jar` and `IfmxTimeSeries-g.jar`. The `-g` file is a debug version that supports tracing (only use this version if you are troubleshooting a problem); it was compiled using the **-g** option of the **javac** command. See "Problem solving" on page 8-17 for more information about tracing.

- `$INFORMIXDIR/extend/TimeSeries.`*`version`*`/java/doc` contains the JavaDoc files

You must do one of the following:

- Modify your CLASSPATH variable to point to these files.

```
CLASSPATH=$INFORMIXDIR/extend/TimeSeries.version
/java/lib/IfmxTimeSeries.jar:$CLASSPATH;export CLASSPATH
```

- Copy these files to an appropriate location and include this in your CLASSPATH variable.

```
CLASSPATH=new_location/IfmxTimeSeries.jar:$CLASSPATH;
     export CLASSPATH
```

- Copy the files to a new location where you undo the `.jar` file and modify your CLASSPATH variable to point to the new location.

```
CLASSPATH=new_location:$CLASSPATH;export CLASSPATH
```

  In this case, because the `.jar` file has been undone, the CLASSPATH variable is only required to point to where the `.jar` file has been extracted to.

Sample programs

Sample programs are included with the database server. To access the sample programs, undo the file `IfmxTimeSeries.jar` using the command **jar -xvf IfmxTimeSeries.jar**. This expands the `.jar` file into a directory structure; the examples are located in `com/informix/docExamples`. The examples include the SQL scripts **setup.sql** and **clean.sql** to set up data for the examples and clean it up afterward.

Time series Java classes

The three time series Java classes represent each of the time series data types: **CalendarPattern**, **Calendar**, and **TimeSeries.**

The IfmxCalendarPattern class

The **IfmxCalendarPattern** class represents a calendar pattern in Java. It has methods that allow you to read and write calendar patterns to and from a database.

The class diagram shows the time series and JDBC 2.0 interfaces that the class implements; SQLData is a JDBC interface and IfmxCalendarPatternUDT is a time series interface (described in "The IfmxCalendarPattern class" on page 8-8).

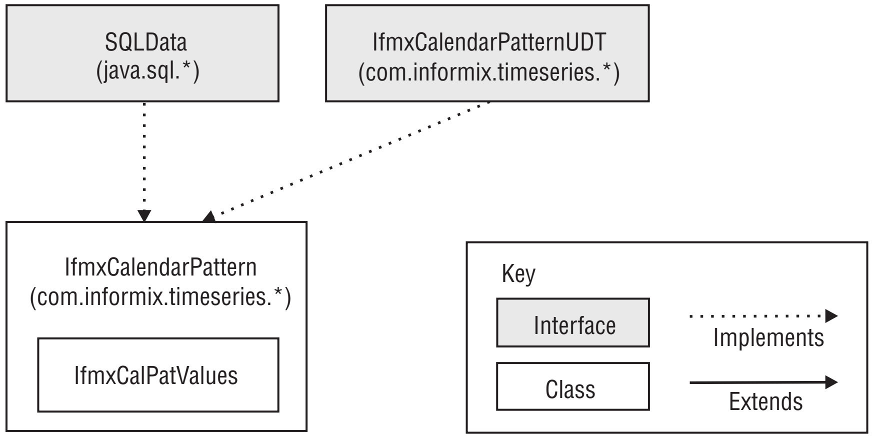


Figure 8-4. CalendarPattern class diagram

The IfmxCalendar class

The **IfmxCalendar** class represents **Calendar** types in Java. It contains methods related to calendar functions and to reading and writing calendars to and from a database.

The class diagram shows the time series and JDBC 2.0 interfaces that the class implements; SQLData is a JDBC interface and IfmxCalendarUDT is a time series interface (described in "The IfmxCalendar class" on page 8-9).

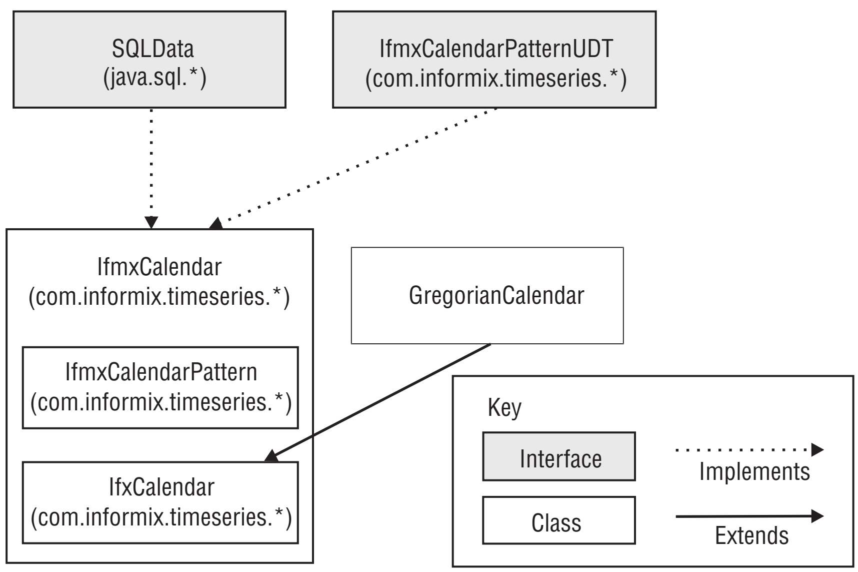


Figure 8-5. Calendar class diagram

The IfmxTimeSeries class

The **IfmxTimeSeries** class represents the **TimeSeries** SQL type in Java. The **IfmxTimeSeries** class is used to read and write **TimeSeries** types to and from a database. This class is based on the JDBC ResultSet interface and can be thought of as a set of rows. Each individual row has metadata associated with it that provides information such as the column name, type, and size.

The **IfmxTimeSeries** class contains most of the time series methods. The class also implements the methods in the ResultSet interface.

The class diagram shows the time series and JDBC 2.0 interfaces that the class implements; SQLData and ResultSet are JDBC interfaces and IfmxTimeSeriesUDT is a time series interface (described in "The IfmxTimeSeries class" on page 8-11).

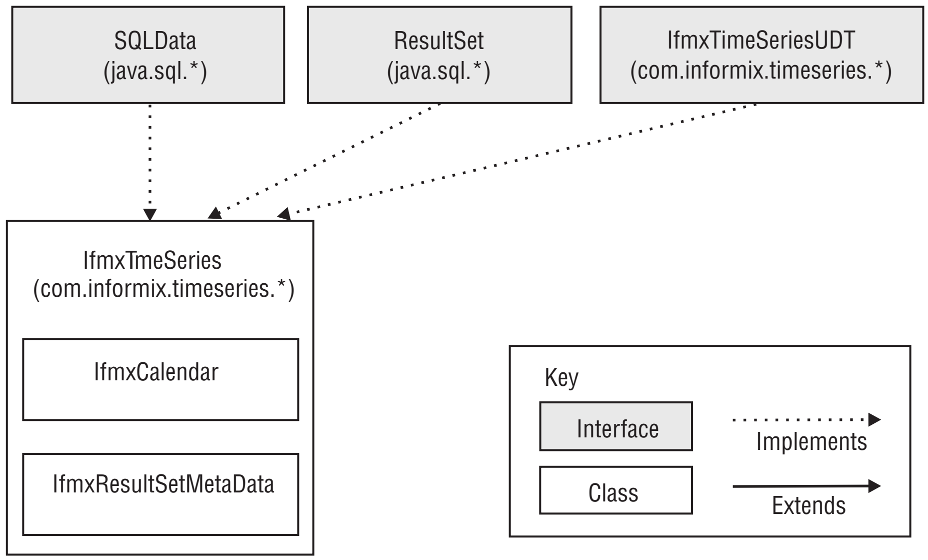


Figure 8-6. TimeSeries class diagram

Get data from the database

The JDBC 2.0 specification describes the procedure for retrieving and sending user-defined types to and from a database. This is achieved by using a type map system. To enable your Java program to retrieve or send **Calendar**, **CalendarPattern**, and **TimeSeries** data to or from an IBM Informix database, you must create a custom type map as described in "Create a custom type map."

After you have created the type map, you can select time series data and manipulate the data as shown in "The IfmxTimeSeries object" on page 8-7.

Note that the time series must be stored in containers. Also, if you clip a time series, the resultant time series is read-only.

Create a custom type map

To define mappings between Java classes and user-defined SQL data types, use a type map. For the Informix JDBC Driver to determine which class to call to handle extended data types, such as **TimeSeries**, your application must define an entry in the type map. You can do this in two ways:

- Create an entry in the type map of the current database connection (the entry is valid only for the current connection).
- Create a map independently and use the map in the getObject method call to extract the **TimeSeries** type from the result set in your application. This can be a better method if you have many time series types.

Related reference:

"UpdElem function" on page 7-113

Create an entry in the database connection

The following example makes an entry in the type map of a connection for handling **TimeSeries** data:

```
java.util.Map customTypeMap;
customTypeMap = conn.getTypeMap();

customTypeMap.put("timeseries(stock_bar)",
    Class.forName("com.informix.timeseries.IfmxTimeSeries"));
```

In this example, *conn* is a valid database connection and *timeseries(stock_bar)* is the **TimeSeries** type. When a **TimeSeries** type is extracted from the database, the IBM Informix JDBC Driver searches the type map for an entry for this data type: in this case, *timeseries(stock_bar)*. If a type map entry exists, an object of the appropriate class is instantiated (**IfmxTimeSeries**, in this example) and the readSQL method of that object is executed.

The readSQL method extracts the time series data from the database result set. There must be an entry in the type map for every **TimeSeries** type that your program uses. For example, you would also require an entry for the *timeseries(stock_trade)* **TimeSeries** type if your Java application accessed that type as well:

```
java.util.Map customTypeMap;
customTypeMap = conn.getTypeMap();

customTypeMap.put("timeseries(stock_trade)",
    Class.forName("com.informix.timeseries.IfmxTimeSeries"));
```

You must also add entries for the **Calendar** and **CalendarPattern** data types if your program selects those types from the database. For these types, you must have one entry for the **CalendarPattern** type and one entry for the **Calendar** type, as shown next.

For **CalendarPattern** data:

```
java.util.Map customTypeMap;
customTypeMap = conn.getTypeMap();

customTypeMap.put("calendarpattern",
    Class.forName
      ("com.informix.timeseries.IfmxCalendarPattern"));
```

For **Calendar** data:

```
java.util.Map customTypeMap;
customTypeMap = conn.getTypeMap();

customTypeMap.put("calendar",
    Class.forName("com.informix.timeseries.IfmxCalendar"));
```

Create a map independently

The following example shows how to create a type map independently of your database connection. For example, you could use just one type map that is global to the application:

```
Map typemap = new Map();
typeMap.add("timeseries(test)",
   Class.forName("com.informix.timeseries.IfmxTimeSeries"));
```

You can use the type map in your getObject method call:

```
ts = (IfmxTimeSeries)rSet.getObject(1, typeMap);
```

Instead of using the type map associated with the connection, the IBM Informix JDBC Driver uses the given type map. This example assumes that the time series column is the first column in the result set.

The IfmxTimeSeries object

Your Java program can use a SELECT statement to retrieve **TimeSeries** data, as in:

```
String sqlCmd = "SELECT ts FROM test WHERE id = 1";
PreparedStatement pStmt = conn.prepareStatement(sqlCmd);
ResultSet rSet = pStmt.executeQuery();

com.informix.timeseries.IfmxTimeSeries ts;

rSet.next()
ts = (IfmxTimeSeries)rSet.getObject(1);
```

In this example, *rSet* is a valid *java.sql.ResultSet* object. After executing the SELECT statement, the getObject method is used to put the time series data into the variable *ts* (an **IfmxTimeSeries** object). The **TimeSeries** type is at column 1 in the result set. The example assumes that an entry has been made into the *conn* object's type map for the **TimeSeries** column, *ts*.

Because the **IfmxTimeSeries** class implements the JDBC ResultSet interface, you can treat an **IfmxTimeSeries** object as if it is an ordinary result set. For example, you can use the **next** method to iterate through the elements of the time series, as in:

```
ts.beforeFirst();
while (ts.next())
{
    java.sql.Timestamp tStamp = ts.getTimestamp(1);
    int col1 = ts.getInt(2);
    int col2 = ts.getInt(3);
}
```

The example shows that you use the beforeFirst method to position the time series cursor before the beginning of the time series and then the next method to iterate through the elements. While looping through the elements, the program uses the getTimestamp method to extract the time stamp into the variable *tStamp* and the getInt method to extract the first column of data into *col1* and the second column into *col2*. The columns of a time series element are numbered, starting with the time stamp column as column 1.

All the available methods of the **IfmxTimeSeries**, **IfmxCalendar**, and **IfmxCalendarPattern** classes are described in the final topics of this section.

"Sample programs" on page 8-3 points to complete sample programs that demonstrate how to retrieve and update time series data.

Write TimeSeries data back to the database

You can write a time series back to the database by using the PreparedStatement.setObject() method:

```
pStmt.setObject(1, ts);
```

In this example, *pStmt* is a valid **PreparedStatement** object, *ts* is a valid **IfmxTimeSeries** object, and the **TimeSeries** type is the first argument in the prepared statement.

Similarly, the following example writes a **Calendar** object to the database:

```
pStmt.setObject(1, c);
```

In this example, *pStmt* is a valid **PreparedStatement** object, *c* is a valid **IfmxCalendar** object, and the **Calendar** type is the first argument in the prepared statement.

And, the following example writes a **CalendarPattern** object to the database:

```
pStmt.setObject(1, cp);
```

Where *pStmt* is a valid **PreparedStatement** object, *cp* is a valid **IfmxCalendarPattern** object, and the **CalendarPattern** type is the first argument in the prepared statement.

Writing data back to the database requires the IBM Informix JDBC Driver to use a type map, similarly to the way described for retrieving data from the database in "Create a custom type map" on page 8-5.

Obtain the time series Java class version

There are two ways to retrieve the version stamp for the time series Java classes:

- From the command line

  Execute the following to return the version stamp:

  ```
  java com.informix.timeseries.Version
  ```
- From within an application

  Use any of the following three methods to return the version stamp:

  ```
  String version;

  version = IfmxTimeSeries.getVersion();
  version = IfmxCalendar.getVersion();
  version = IfmxCalendarPattern.getVersion();
  ```

The getVersion method is a static method; therefore, you are not required to create a time series object to retrieve the version stamp. The version returned from the three classes is always the same.

The IfmxCalendarPattern class

The **IfmxCalendarPattern** class implements the IfmxCalendarPatternUDT TimeSeries interface. An interface provides an abstract description of the methods and any constants belonging to a class. The IfmxCalendarPatternUDT interface specifies the standard constants that any calendar pattern class might have to use. The interface is intended for use only by programmers who want to develop calendar pattern classes.

You can create an **IfmxCalendarPattern** object by:

- Selecting a calendar pattern from the database
- Selecting a calendar from the database
- Selecting a time series from the database

- Instantiating a new object to be inserted into the database

The **IfmxCalendarPattern** class defines the following constructors for these situations:

```
IfmxCalendarPattern()
IfmxCalendarPattern(String pat) throws SQLException
```

When you select a calendar pattern from the database, the getObject method is used to extract the calendar pattern from the result set. The IBM Informix JDBC Driver instantiates a new **IfmxCalendarPattern** object using the first constructor, **IfmxCalendarPattern()**. It creates an empty object with no variables initialized.

When you create a calendar pattern on the client to insert into the database, use the second constructor to instantiate the object. This constructor initializes the new object by parsing the input string. The format of the string is: *{pattern specification}, interval*: for example, `{5 on, 2 off}, day`.

The **IfmxCalendarPattern** class provides the following methods.

Method	Signature	Description
getInterval	public byte getInterval() throws SQLException	Returns the calendar pattern interval.
getIntervalStr	public String getIntervalStr() throws SQLException	Returns the string representation of the calendar pattern interval. Valid values are: • Second • Minute • Hour • Day • Week • Month • Year
getSQLTypeName	public String getSQLTypeName()	Returns the SQL type name for the object: in this case, calendar pattern.
readSQL	public void readSQL(SQLInput stream String type)	Called automatically by the IBM Informix JDBC Driver to initialize an IfmxCalendarPattern object from the binary result set stream.
toString	public String toString()	Returns the String representation of this calendar pattern object.
writeSQL	public void writeSQL(SQLOutput stream)	Called automatically by the Informix JDBC Driver when a setObject method is called to insert an **IfmxCalendarPattern** object into a prepared statement to be sent to the database server.

The IfmxCalendar class

The **IfmxCalendar** class implements the IfmxCalendarUDT TimeSeries interface. An interface provides an abstract description of the methods and any constants belonging to a class. The IfmxCalendarUDT interface specifies the standard constants that any calendar class might be required to use. The interface is intended for use only by programmers who want to develop calendar classes.

You create an **IfmxCalendar** object by:
- Selecting a calendar from the database
- Selecting a time series from the database
- Instantiating a new object to be inserted into the database

The **IfmxCalendar** class defines the following constructors for these situations:

```
IfmxCalendar()

IfmxCalendar(String calName, TimeStamp calStart,
   TimeStamp patStart, String pattern)
   throws SQLexception

IfmxCalendar(String calName, TimeStamp calStart,
    TimeStamp patStart, IfmxCalendarPattern cPat)
     throws SQLexception
```

When you select a calendar from the database, the getObject method is used to extract the calendar from the result set. The IBM Informix JDBC Driver instantiates a new **IfmxCalendar** object using the first constructor, **IfmxCalendar()**. It creates an empty object with no variables initialized.

When you create a calendar on the client to insert into the database, use the second or third constructor to instantiate the object. These constructors initialize the new object by parsing the input string and using the given arguments. A calendar contains an embedded calendar pattern. The first constructor instantiates an **IfmxCalendarPattern** object using the specified calendar pattern string. The second constructor copies the calendar pattern from the **IfmxCalendarPattern** object that you specify.

The **IfmxCalendar** class provides the following methods.

Method	**Signature**	**Description**
getName	public String getName() throws SQLException	Returns the calendar name.
getOffset	public int getOffset() throws SQLException	Returns the calendar offset.
getStartDate	public Timestamp getStartDate() throws SQLException	Returns the calendar start date.
getPatStartDate	public Timestamp getPatStartDate() throws SQLException	Returns the start date of the calendar pattern associated with the calendar.
getPattern	public IfmxCalendarPattern getPattern() throws SQLException	Returns the calendar pattern associated with the calendar.
getSQLTypeName	public String getSQLTypeName() throws SQLException	Returns the SQL type name for the object: in this case, calendar.
getStartDate	public Timestamp getStartDate() throws SQLException	Returns the calendar start date.
getTimestamp FromOffset	public Timestamp getTimestampFromOffset() throws SQLException	Returns the time stamp for a given offset.
getOffsetFrom Timestamp	public int getOffsetFromTimestamp() throws SQLException	Returns the offset for a given time stamp.
readSQL	public void readSQL() throws SQLException	Called automatically by the IBM Informix JDBC Driver to initialize an **IfmxCalendar** object from a binary result stream.
toString	public String toString()	Returns the string representation of an **IfmxCalendar** object.

Method	Signature	Description
writeSQL	public void writeSQL() throws SQLException	Called automatically by the Informix JDBC Driver when a setObject method is called to insert an **IfmxCalendar** object into a prepared statement to be sent to the database server.

The IfmxTimeSeries class

The **IfmxTimeSeries** class implements the IfmxTimeSeriesUDT TimeSeries interface. An interface provides an abstract description of the methods and any constants belonging to a class. The IfmxTimeSeriesUDT interface specifies the standard constants that any calendar class might require. The interface is intended for use only by programmers who want to develop calendar classes.

You create an **IfmxTimeSeries** object by:
- Selecting a time series from the database
- Instantiating a new object to be inserted into the database

The **IfmxTimeSeries** class defines the following constructors for these situations:

```
IfmxTimeSeries()

IfmxTimeSeries(java.sql.Timestamp startdate,
    IfmxCalendar cal,
    String containerName,
    int threshold,
    String sqlTypeName,
    Connection conn)
throws SQLexception
```

When you select a time series from the database, the **getObject** method is used to extract the time series from the result set. The IBM Informix JDBC Driver instantiates a new **IfmxTimeSeries** object using the first constructor, **IfmxTimeSeries()**. It creates an empty object with no variables initialized.

When you create a time series on the client to be inserted into the database, use the second constructor to instantiate the object. This constructor initializes the new object by parsing the input string. The following information is required:
- A start date
- A calendar (the calendar must exist in the database)
- A container name (the container must exist in the database).
- A threshold
- A row type name (the row type must exist in the database)
- A valid connection. The connection must include an entry for the time series type in its type map (see “Create a custom type map” on page 8-5).
- Whether the time series is regular or irregular

The **IfmxTimeSeries** class provides methods for manipulating time series. The **IfmxTimeSeries** class also provides methods that it inherits from the JDBC ResultSet interface. These methods are listed in “Public methods inherited from the JDBC ResultSet interface” on page 8-12; see your JDBC documentation for more information about these methods.

The IfmxTimeSeries class methods

The **IfmxTimeSeries** class provides methods for manipulating time series. The **IfmxTimeSeries** class also provides methods that it inherits from the JDBC ResultSet interface. These methods are listed in "Public methods inherited from the JDBC ResultSet interface"; see your JDBC documentation for more information about these methods.

Time series methods

The time series methods in the **IfmxTimeSeries** class are for manipulating time series.

Table 8-1. Time series methods

Method	Signature	Description
clip	public IfmxTimeSeries clip(java.sql.Timestamp start, java.sql.Timestamp end) throws SQLException	Returns a new **IfmxTimeSeries** object containing the elements between the start and end points.
getCalendar	public IfmxCalendar getCalendar() throws SQLException	Returns the **IfmxCalendar** object associated with the **IfmxTimeSeries** object.
getContainerName	public String getContainerName() throws SQLException	Returns the container name associated with the **IfmxTimeSeries** object. Returns NULL if no container name is set.
getNumberOf Elements	int getNumberOfElements()	Gets the number of elements in this **IfmxTimeSeries** object.
getOffset	int getOffset()	Returns the offset for the **IfmxTimeSeries** object.
getOrigin	public int getOrigin() throws SQLException	Returns the origin for the **IfmxTimeSeries** object.
inContainer	public boolean inContainer() throws SQLException	Returns TRUE if the **IfmxTimeSeries** object is in a container; FALSE otherwise.
isHidden	boolean isHidden()	Returns TRUE if the current element is hidden.
isNull	public boolean isNull(columnIndex int) throws SQLException public boolean isNull(columnName String) throws SQLException	The isNull method tests if the value on the specified column of the selected TimeSeries row is NULL and returns TRUE if it is; otherwise, FALSE. You can specify the column either by number (columnIndex) or by name (columnName).
isRegular	public boolean isRegular() throws SQLException	Returns TRUE if the **IfmxTimeSeries** object represents a regular time series; FALSE otherwise.
setConnection	public void setConnection(connection Conn) throws SQLException	Sets the connection to be used to update time series elements. Must be a valid connection with a correct type map.

The following methods are not currently supported:
- getNelems
- hideElem

Public methods inherited from the JDBC ResultSet interface

The **IfmxTimeSeries** class contains public methods inherited from the JDBC **ResultSet** interface.

The following table shows the supported signatures for the public methods. Other signatures for these methods are not supported.

Table 8-2. Public methods

Method	Signature	Description
absolute	boolean absolute(int row)	Moves the time series cursor to the row indicated by *row*.
afterLast	void afterLast()	Moves the time series cursor to the end of the **IfmxTimeSeries** object, just after the last element.
beforeFirst	void beforeFirst()	Moves the time series cursor to the beginning of the **IfmxTimeSeries** object, just before the first element.
deleteRow	void deleteRow(Connection conn) throws SQLException	Deletes the current row from the time series and uses the specified connection to delete the time series in the underlying database.
findColumn	int findColumn (java.lang.String columnName)	Retrieves the column index for *columnName*.
first	boolean first()	Moves the time series cursor to the first element in the **IfmxTimeSeries** object.
getBigDecimal	java.math.BigDecimal getBigDecimal (int columnIndex)	Gets the value in the current time series element as a *bigDecimal* value.
getBigDecimal	java.math.BigDecimal getBigDecimal (java.lang.String columnName)	Gets the value in the current time series element as a *bigDecimal* value.
getBoolean	boolean getBoolean(int columnIndex)	Gets the value in the current time series element as a *boolean* value.
getBoolean	boolean getBoolean (java.lang.String columnName)	Gets the value in the current time series element as a *boolean* value.
getByte	byte getByte(int columnIndex)	Gets the value in the current time series element as a *byte* value.
getByte	byte getByte (java.lang.String columnName)	Gets the value in the current time series element as a *byte* value.
getConcurrency	int getConcurrency()	Gets the concurrency type for this **IfmxTimeSeries** object.
getDate	java.sql.Date getDate(int columnIndex)	Gets the value in the current time series element as a *java.sql.date* value.
getDate	java.sql.Date getDate (java.lang.String columnName)	Gets the value in the current time series element as a *java.sql.date* value.
getDouble	double getDouble(int columnIndex)	Gets the value in the current time series element as a *double* value.
getDouble	double getDouble (java.lang.String columnName)	Gets the value in the current time series element as a *double* value.
getFloat	float getFloat(int columnIndex)	Gets the value in the current time series element as a *float* value.
getFloat	float getFloat (java.lang.String columnName)	Gets the value in the current time series element as a *float* value.
getInt	int getInt(int columnIndex)	Gets the value in the current time series element as an *int* value.
getInt	int getInt(java.lang.String columnName)	Gets the value in the current time series element as an *int* value.

Table 8-2. Public methods (continued)

Method	Signature	Description
getLong	int getLong(int columnIndex)	Gets the value in the current time series element as a *long* value.
getLong	int getLong (java.lang.String columnName)	Gets the value in the current time series element as a *long* value.
getMetaData	java.sql.ResultSetMetaData getMetaData()	Retrieves a **ResultSetMetaData** object that contains the number, types, and properties of the **IfmxTimeSeries** object elements.
getObject	java.lang.Object getObject (int columnIndex)	Gets the value in the current time series element as an *Object*.
getRow	int getRow()	Retrieves the number of the current time series element.
getShort	short getShort(int columnIndex)	Gets the value in the current time series element as a *short* value.
getShort	short getShort (java.lang.String columnName)	Gets the value in the current time series element as a *short* value.
getSQLTypeName	java.lang.String getSQLTypeName()	Returns the SQL type name used by the database for the data type.
getString	java.lang.String getString (int columnIndex)	Gets the value in the current time series element as a *String* value.
getString	java.lang.String getString (java.lang.String columnName)	Gets the value in the current time series element as a *String* value.
getTime	java.sql.Time getTime(int columnIndex)	Gets the value in the current time series element as a *java.sql.Time* value.
getTime	java.sql.Time getTime (java.lang.String columnName)	Gets the value in the current time series element as a *java.sql.Time* value.
getTimestamp	java.sql.Timestamp getTimestamp (int columnIndex)	Gets the value in the current time series element as a *Timestamp* object.
getTimestamp	java.sql.Timestamp getTimestamp (java.lang.String columnName)	Gets the value in the current time series element as a *Timestamp* object.
getTSMetaData	java.sql.ResultSetMetaData getTSMetaData()	Retrieves an *IfmxResultSetMetaData* object that contains the number, types, and properties of the time series elements in this **IfmxTimeSeries** object.
getType	int getType()	Retrieves the type of this **IfmxTimeSeries** object.
insertRow	void insertRow(Connection conn)	Inserts the current row using the connection set by the setConnection method to update the underlying time series stored in the database. The setConnection method is described in "Time series methods" on page 8-12.
isAfterLast	boolean isAfterLast()	Determines whether the time series cursor is after the last element.
isBeforeFirst	boolean isBeforeFirst()	Determines whether the time series cursor is before the first element.
isFirst	boolean isFirst()	Determines whether the time series cursor is on the first element in this time series.

Table 8-2. Public methods (continued)

Method	Signature	Description
isLast	boolean isLast()	Determines whether the current element is the last element in this time series.
last	boolean last()	Moves the time series cursor to the last element in the time series.
moveToCurrentRow	void moveToCurrentRow()	Moves the time series cursor to the remembered position in this **IfmxTimeSeries** object, usually the current element.
moveToInsertRow	void moveToInsertRow()	Moves the time series cursor to the insert row.
next	boolean next()	Initially moves the time series cursor to the first element in the time series. Subsequent calls move the cursor to the second element, then the third, and so on.
previous	boolean previous()	Moves the time series cursor to the previous row.
readSQL	void readSQL (java.sql.InputStream stream, java.lang.String type)	Populates the **IfmxTimeSeries** object with data read from the given binary input stream.
relative	boolean relative(int row)	Moves the time series cursor the number of rows specified by *row*.
updateBoolean	void updateBoolean (int columnIndex, boolean x)	Updates the current time series element with the given *boolean* object.
updateBoolean	void updateBoolean (java.lang.String columnName, boolean x)	Updates the current time series element with the given *boolean* object.
updateByte	void updateByte(int columnIndex, byte x)	Updates the current time series element with the given *byte* object.
updateByte	void updateByte (java.lang.String columnName, byte x)	Updates the current time series element with the given *byte* object.
updateDate	void updateDate (int columnIndex, java.sql.Date x)	Updates the current time series element with the given *date* object.
updateDate	void updateDate (java.lang.String columnName, java.sql.Date x)	Updates the current time series element with the given *date* object.
updateDouble	void updateDouble (int columnIndex, double x)	Updates the current time series element with the given *double* object.
updateDouble	void updateDouble (java.lang.String columnName, double x)	Updates the current time series element with the given *double* object.
updateFloat	void updateFloat (int columnIndex, float x)	Updates the current time series element with the given *float* object.
updateFloat	void updateFloat (java.lang.String columnName, float x)	Updates the current time series element with the given *float* object.
updateInt	void updateInt(int columnIndex, int x)	Updates the current time series element with the given *int* object.
updateInt	void updateInt (java.lang.String columnIndex, int x)	Updates the current time series element with the given *int* object.
updateLong	void updateLong (int columnIndex, long x)	Updates the current time series element with the given *long* object.

Table 8-2. Public methods (continued)

Method	Signature	Description
updateLong	void updateLong (java.lang.String columnName, long x)	Updates the current time series element with the given *long* object.
updateNull	void updateNull(int columnIndex)	Updates the current time series element with a null value.
updateNull	void updateNull (java.lang.String columnName)	Updates the current time series element with a null value.
updateRow	void updateRow(Connection conn)	Updates the underlying time series with the contents of the current row using the connection set by the setConnection method. The setConnection method is described in "Time series methods" on page 8-12.
updateShort	void updateShort (int columnIndex, short x)	Updates the current time series element with the given *short* object.
updateShort	void updateShort (java.lang.String columnName, short x)	Updates the current time series element with the given *short* object.
updateString	void updateString (int columnIndex, java.lang.String x)	Updates the current time series element with the given *String* object.
updateString	void updateString (java.lang.String columnName, java.lang.String x)	Updates the current time series element with the given *String* object.
updateTime	void updateTime (int columnIndex, java.sql.Time x)	Updates the current time series element with the given *Time* object.
updateTime	void updateTime (java.lang.String columnName, java.sql.Time x)	Updates the current time series element with the given *Time* object.
updateTimestamp	void updateTimestamp (int columnIndex, java.sql.Timestamp x)	Updates the current time series element with the given *Timestamp* object.
updateTimestamp	void updateTimestamp (java.lang.String columnName, java.sql.Timestamp x)	Updates the current time series element with the given *Timestamp* object.
wasNull	boolean wasNull()	Checks whether the current element is `NULL`.
writeSQL	void writeSQL(java.sql.Output stream)	Called automatically by the JDBC driver when a setObject method is called to insert an **IfmxTimeSeries** object into a prepared statement to be sent to the database server.

The following methods are not currently supported:
- cancelRowUpdates
- clearWarnings
- close
- getArray
- getAsciiStream
- getBytes
- getFetchSize
- getRef
- getUnicodeStream
- getWarnings
- updateBigDecimal

- updateBytes
- updateObject
- refreshRow
- rowDeleted
- rowInserted
- rowUpdated
- setFetchDirection
- setFetchSize
- updateAsciiStream
- updateBinaryStream
- updateCharacterStream

Problem solving

These topics contain suggestions to help solve problems if they should occur while you are using the time series Java class library.

If a user-defined error is returned, turn on JDBC tracing (PROTOCOLTRACE). The output file should contain the exact error message. See your JDBC documentation for information about how to turn on JDBC tracing.

Tracing with the Java class library

To turn on tracing with the time series Java class library, change your CLASSPATH variable to point to the debug version of the library, `AA01TimeSeries-g.jar`, and add the information described in the following to the database URL that you use at connection time.

To turn on tracing, specify the environment variables TRACE, TRACEFILE, PROTOCOLTRACE, and PROTOCOLTRACEFILE in the connection property list when you establish a connection to an IBM Informix database or database server. The following table describes the tracing environment variables.

Environment variable	Description
TRACE	Traces general information from IBM Informix JDBC Driver. Can be set to one of the following levels: **0** Tracing not enabled. This is the default value. **1** Traces the entry and exit points of methods. **2** Same as Level 1, except generic error messages are also traced. **3** Same as Level 2, except data variables are also traced.
TRACEFILE	Specifies the full path name of the operating system file on the client computer to which the TRACE messages are written.

Environment variable	Description
`PROTOCOLTRACE`	Traces the SQLI protocol messages sent between your Java program and the IBM Informix database server. Can be set to the following levels: **0** Protocol tracing not enabled. This is the default value. **1** Traces message IDs. **2** Same as Level 1, except the data in the message packets is also traced.
`PROTOCOLTRACEFILE`	Specifies the full path name of the operating system file on the client computer to which the PROTOCOLTRACE messages are written.

The following example of a database URL specifies the highest level of protocol tracing and sets tracing output to the operating system file `/tmp/trace.out`:

```
String url = "jdbc:informix-
sqli://123.45.67.89:1533:informixserver=myserver;user=rdtest;password=test;
PROTOCOLTRACE=2;PROTOCOLTRACEFILE=/tmp/trace.out";
```

For more information about establishing a connection to an IBM Informix database or database server using a property list, see the *IBM Informix JDBC Driver Programmer's Guide*.

Chapter 9. Time series API routines

The time series application programming interface routines allow application programmers to directly access a time series datum.

You can scan and update a set of time series elements, or a single element referenced by either a time stamp or a time series index. These routines can be used in client programs that fetch time series data in binary mode or in registered server or client routines that have an argument or return value of a time series type.

If there is a failure, these routines raise an error condition and do not return a value.

On UNIX, these routines exist in two archives: `tsfeapi.a` and `tsbeapi.a`. To use any of these routines, include the `tsbeapi.a` file when producing a shared library for the server, or use `tsfeapi.a` when compiling a client application.

The `tseries.h` header file must be included when there are calls to any of the time series interface routines.

On UNIX, `tsfeapi.a`, `tsbeapi.a`, and `tseries.h` are all in the `lib` directory in the database server installation.

On Windows, these routines exist in two archives: `tsfeapi.lib` and `tsbeapi.lib`. To use any of these routines, include the `tsbeapi.lib` file when producing a shared library for the server, or use `tsfeapi.lib` when compiling a client application.

The `tseries.h` header file must be included when there are calls to any of the time series interface routines.

On Windows, `tsfeapi.lib`, `tsbeapi.lib`, and `tseries.h` are all in the `lib` directory in the database server installation.

Important: Since values returned by **mi_value** are valid only until the next **mi_next_row** or **mi_query_finish** call, it might be necessary to put time series in save sets or to use **ts_copy** to access time series outside an **mi_get_results** loop.

Related concepts:

"Planning for accessing time series data" on page 1-12

Differences in using functions on the server and on the client

There are significant differences between using the client version of the time series API (tsfeapi) and the server version of the time series API (tsbeapi).

The client and server interfaces do not behave in exactly the same way when updating a time series. This is because tsbeapi operates directly on a time series, whereas tsfeapi operates on a private copy of a time series. This means that updates through tsbeapi are always reflected in the database, while updates through tsfeapi are not. For changes made by tsfeapi to become permanent, the client must write the updated time series back into the database.

Another difference between the two interfaces is in how time series are passed as arguments to the **mi_exec_prepare_statement()** function. On the server, no special steps are required: a time series can be passed as is to this function. However, on the client you must make a copy of the time series with **ts_copy** and pass the copy as an argument to the **mi_exec_prepare_statement()** function.

There can be a difference in efficiency between the client and the server APIs. Functions built to run on the server take advantage of the underlying paging mechanism. For instance, if a function must scan across 20 years worth of data, the tsbeapi interface keeps only a few pages in memory at any one time. For a client program to do this, the entire time series must be brought over to the client and kept in memory. Depending on the size of the time series and the memory available, this might cause swapping problems on the client. However, performance depends on many factors, including the pattern of usage and distribution of your hardware. If hundreds of users are performing complex analysis in the server, it can overwhelm the server, whereas if each client does their portion of the work, the load can be better balanced.

API data structures

These topics describe the data structures used by time series API routines.

The ts_timeseries structure

A **ts_timeseries** structure is the header for a time series. It can be stored in and retrieved from a time series column of a table.

The **ts_timeseries** structure contains pointers, so it cannot be copied directly. Use the **ts_copy()** function to copy a time series.

When you pass a binary time series value, *ts*, of type **ts_timeseries**, to **mi_exec_prepared_statement()**, you must pass *ts* in the values array and `0` in the lengths array.

The ts_tscan structure

A **ts_tscan** structure allows you to look at no more than two time series elements at a time. It maintains a current scan position in the time series and has two element buffers for creating elements. An element fetched from a scan is overwritten after two **ts_next()** calls.

A **ts_tscan** structure is created with the **ts_begin_scan()** function and destroyed with the **ts_end_scan()** procedure.

The ts_tsdesc structure

A **ts_tsdesc** structure contains a time series (**ts_timeseries**) and data structures for working with it. Among other things, **ts_tsdesc** tracks the current element and holds two element buffers for creating two elements.

Important: The two element buffers are shared by the element-fetching functions. An element that is fetched is overwritten two fetch calls later. Elements fetched by functions like **ts_elem()** should not be explicitly freed. They are freed when the **ts_tsdesc** is closed.

If you must look at more than two elements at a time, open a scan or use the **ts_make_elem()** or **ts_make_elem_with_buf()** routines to make a copy of one of your elements.

A **ts_tsdesc** structure is created by the **ts_open()** function and destroyed by the **ts_close()** procedure. It is used by most of the time series API routines.

The ts_tselem structure

A **ts_tselem** structure is a pointer to one element (row) of a time series.

When you use **ts_tselem** with a regular time series, the time stamp column in the element is left as `NULL`, allowing you to avoid the expense of computing the time stamp if it is not required. The time stamp is computed on demand in the **ts_get_col_by_name()**, **ts_get_col_by_number()**, and **ts_get_all_cols()** routines. For irregular time series, the time stamp column is never `NULL`.

You can convert a **ts_tselem** structure to and from an MI_ROW structure with the **ts_row_to_elem()** and **ts_elem_to_row()** routines.

If the element was created by the **ts_make_elem()** or **ts_make_elem_with_buf()** procedure, you must use the **ts_free_elem()** procedure to free the memory allocated for a **ts_tselem** structure.

API routines

This section contains:
- Time series API routines by task type
- The correspondence between API and SQL routines
- Individual routine reference pages

The following table shows the time series interface routines listed by task type. An uppercase routine name, such as TS_ELEM_NULL, denotes a macro.

Task type	Description	Routine name
Open and close a time series	Open a time series	**ts_open()**
	Close a time series	**ts_close()**
	Return a pointer to the time series associated with the given time series descriptor	**ts_get_ts()**
Create and copy a time series	Create a time series	**ts_create()**
	Create a time series with metadata	**ts_create_with_metadata()**
	Copy a time series	**ts_copy()**
	Free all memory associated with a time series created with **ts_copy()** or **ts_create()**	**ts_free()**
	Copy all elements of one time series into another	**ts_put_ts()**

Task type	Description	Routine name
Scan a time series	Start a scan	**ts_begin_scan()**
	Retrieve the next element from a scan	**ts_net()**
	End a scan	**ts_end_scan()**
	Find the time stamp of the last element retrieved from a scan	**ts_current_timestamp()**
	Return the offset for the last element returned by **ts_next()**	**ts_current_offset()** (regular only)
Make elements visible or invisible to a scan	Make an element invisible	**ts_hide_elem()**
	Make an element visible	**ts_reveal_elem()**
Select individual elements from a time series	Get the element closest to a given time stamp	**ts_closest_elem()**
	Get the element associated with a given time stamp	**ts_elem()**
	Get the element at a specified position	**ts_nth_elem())**
	Get the first element	**ts_first_elem()**
	Get the last element	**ts_last_elem()**
	Find the next element after a given time stamp	**ts_next_valid()**
	Find the last element before a given time stamp	**ts_previous_valid()**
	Find the last element at or before a given time stamp	**ts_last_valid()**
Update a time series	Insert an element	**ts_ins_elem()**
	Update an element	**ts_upd_elem()**
	Delete an element	**ts_del_elem()**
	Put an element in a place specified by a time stamp	**ts_put_elem()** **ts_put_elem_no_dups()**
	Append an element	**ts_put_last_elem()** (regular only)
	Put an element in a place specified by an offset	**ts_put_nth_elem()** (regular only)
Modify metadata	Update metadata	**ts_update_metadata()**
Convert between an index and a time stamp	Convert time stamp to index	**ts_index()** (regular only)
	Convert index to time stamp	**ts_time()** (regular only)
Transform an element	Create an element from an array of values and nulls	**ts_make_elem()** **ts_make_elem_rowdesc()** **ts_make_elem_with_buf()**
	Convert an MI_ROW value to an element	**ts_row_to_elem()**
	Convert an element to an MI_ROW value	**ts_elem_to_row()**
	Free memory from a time series element created by **ts_make_elem()** or **ts_row_to_elem()**	**ts_free_elem()**
Extract column data from an element	Get a column from an element by name	**ts_get_col_by_name()**
	Get a column from an element by number	**ts_get_col_by_number()**
	Pull columns from an element into *values* and *nulls* arrays	**ts_get_all_cols()**

Task type	Description	Routine name
Create and perform calculations with time stamps	Compare two time stamps	**ts_datetime_cmp()**
	Get fields from a time stamp	**ts_get_stamp_fields()**
	Create a time stamp	**ts_make_stamp()**
	Calculate the number of intervals between two time stamps	**ts_tstamp_difference()**
	Subtract *N* intervals from a time stamp	**ts_tstamp_minus()**
	Add *N* intervals to a time stamp	**ts_tstamp_plus()**
Get information about element data	Find the number of a column	**ts_col_id()**
	Return the number of columns contained in each element	**ts_col_cnt()**
	Get type information for a column specified by number	**ts_colinfo_number()**
	Get type information for a column specified by name	**ts_colinfo_name()**
	Determine if an element is hidden	TS_ELEM_HIDDEN
	Determine if an element is NULL	TS_ELEM_NULL
Get information about a time series	Get the name of a calendar associated with a time series	**ts_get_calname()**
	Return the number of elements in a time series	**ts_nelems()**
	Return the flags associated with the time series	**ts_get_flags()**
	Get the name of the container	**ts_get_containername()**
	Determine if the time series is in a container	TS_IS_INCONTAINER
	Get the origin of the time series	**ts_get_origin()**
	Get the metadata associated with the time series	**ts_get_metadata()**
	Determine if the time series is irregular	TS_IS_IRREGULAR
Get information about a calendar	Return the number of valid intervals between two time stamps	**ts_cal_index()**
	Return all valid timepoints between two time stamps	**ts_cal_range()**
	Return a specified number of time stamps starting at a given time stamp	**ts_cal_range_index()**
	Return the time stamp at a given number of intervals after a given time stamp	**ts_cal_stamp()**

The following functions are used only with regular time series:
- **ts_current_offset()**
- **ts_index()**
- **ts_nth_elem()**
- **ts_put_last_elem()**
- **ts_put_nth_elem()**
- **ts_time()**

Some of the API routines are much the same as SQL routines. The mapping is shown in the following table.

API routine	SQL routine
ts_cal_index()	**CalIndex**
ts_cal_range()	**CalRange**
ts_cal_stamp()	**CalStamp**
ts_create()	**TSCreate, TSCreateIrr**
ts_create_with_metadata()	**TSCreate, TSCreateIrr**
ts_del_elem()	**DelElem**
ts_elem()	**GetElem**
ts_first_elem()	**GetFirstElem**
ts_get_calname()	**GetCalendarName**
ts_get_containername()	**GetContainerName**
ts_get_metadata()	**GetMetaData**
ts_get_origin()	**GetOrigin**
ts_hide_elem()	**HideElem**
ts_index()	**GetIndex**
ts_ins_elem()	**InsElem**
ts_last_elem()	**GetLastElem**
ts_nelems()	**GetNelems**
ts_next_valid()	**GetNextValid**
ts_nth_elem()	**GetNthElem**
ts_previous_valid()	**GetPreviousValid**
ts_put_elem()	**PutElem**
ts_put_elem_no_dups()	**PutElemNoDups**
ts_put_ts()	**PutTimeSeries**
ts_reveal_elem()	**RevealElem**
ts_time()	**GetStamp**
ts_update_metadata()	**UpdMetaData**
ts_upd_elem()	**UpdElem**

The ts_begin_scan() function

The **ts_begin_scan()** function begins a scan of elements in a time series.

Syntax

```
ts_tscan *
ts_begin_scan(ts_tsdesc   *tsdesc,
              mi_integer  flags,
              mi_datetime *begin_stamp,
              mi_datetime *end_stamp)
```

tsdesc Returned by **ts_open()**.

flags Determines how a scan should work on the returned set.

begin_stamp
Pointer to **mi_datetime**, to specify where the scan should start. If *begin_stamp* is NULL, the scan starts at the beginning of the time series. The *begin_stamp* argument acts much like the *begin_stamp* argument to the **Clip** function (“Clip function” on page 7-26) unless TS_SCAN_EXACT_START is set.

end_stamp
Pointer to **mi_datetime**, to specify where the scan should stop. If *end_stamp* is NULL, the scan stops at the end of the time series. When *end_stamp* is set, the scan stops after the data at *end_stamp* is returned.

Description

This function starts a scan of a time series between two time stamps.

The scan descriptor is closed by calling **ts_end_scan()**.

The *flags* argument values

The *flags* argument determines how a scan should work on the returned set. Valid values for the *flags* argument are defined in `tseries.h`. The integer value is the sum of the desired values from the following table.

Flag	Value	Meaning
TS_SCAN_HIDDEN	512 (0x200)	Return hidden elements marked by **ts_hide_elem()**
TS_SCAN_EXACT_START	256 (0x100)	Return `NULL` if the begin point is earlier than the time series origin. (Normally a scan does not start before the time series origin.)
TS_SCAN_EXACT_END	128 (0x80)	Return `NULL` until the end timepoint of the scan is reached, even if the end timepoint is beyond the end of the time series.
TS_SCAN_NO_NULLS	32 (0x20)	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as `NULL`. If TS_SCAN_NO_NULLS is set, an element is returned that has each column set to `NULL` instead.
TS_SCAN_SKIP_END	16 (0x10)	Skip the element at the end timepoint of the scan range.
TS_SCAN_SKIP_BEGIN	8 (0x08)	Skip the element at the beginning timepoint of the scan range.
TS_SCAN_SKIP_HIDDEN	4 (0x04)	Skip hidden elements.

Returns

An open scan descriptor, or `NULL` if the scan times are both before the origin of the time series or if the end time is before the start time.

Example

See the **ts_interp()** function, in Appendix A, “The Interp function example,” on page A-1, for an example of the **ts_begin_scan()** function.

Related reference:
"HideElem function" on page 7-55
"The ts_current_offset() function" on page 9-18
"The ts_current_timestamp() function" on page 9-19
"The ts_end_scan() procedure" on page 9-23
"The ts_next() function" on page 9-39
"The ts_open() function" on page 9-41
"The ts_first_elem() function" on page 9-23

The ts_cal_index() function

The **ts_cal_index()** function returns the number of valid intervals in a calendar between two given time stamps.

Syntax

```
mi_integer *
ts_cal_index (MI_CONNECTION *conn,
              mi_string     *cal_name,
              mi_datetime   *begin_stamp,
              mi_datetime   *end_stamp)
```

conn A valid DataBlade API connection.

cal_name
The name of the calendar.

begin_stamp
The beginning time stamp. *begin_stamp* must not be earlier than the calendar origin.

end_stamp
The time stamp whose offset from *begin_stamp* is to be determined. This time stamp can be earlier than *begin_stamp*.

Description

The equivalent SQL function is **CalIndex**.

Returns

The number of valid intervals in the given calendar between the two time stamps. If *end_stamp* is earlier than *begin_stamp*, then the result is a negative number.

Related reference:
"The ts_cal_range() function" on page 9-9
"The ts_cal_range_index() function" on page 9-10
"The ts_cal_stamp() function" on page 9-11
"The ts_index() function" on page 9-32

The ts_cal_pattstartdate() function

The **ts_cal_pattstartdate()** function takes a calendar name and returns the start date of the pattern for that calendar.

Syntax

```
mi_datetime *
ts_cal_pattstartdate (MI_CONNECTION *conn,
                      mi_string     *cal_name)
```

conn A pointer to a valid DataBlade API connection structure.

cal_name
The name of the calendar.

Description

The equivalent SQL function is **CalPattStartDate**.

Returns

An mi_datetime pointer that points to the start date of a calendar pattern. You must free this value after use.

Related reference:

"The CalPattStartDate function" on page 5-2

"The ts_cal_startdate() function" on page 9-11

The ts_cal_range() function

The **ts_cal_range()** function returns a list of time stamps containing all valid timepoints in a calendar between two time stamps (inclusive of the specified time stamps).

Syntax

```
MI_COLLECTION *
ts_cal_range (MI_CONNECTION *conn,
              mi_string     *cal_name,
              mi_datetime   *begin_stamp,
              mi_datetime   *end_stamp)
```

conn A valid DataBlade API connection.

cal_name
The name of the calendar.

begin_stamp
The begin point of the range. It must not be earlier than the calendar origin.

end_stamp
The end point of the range.

Description

This function is useful if you must print out the time stamps of a series of regular time series elements. If the range is known, getting an array of all of the time stamps is more efficient than using **ts_time()** on each element.

The caller is responsible for freeing the result of this function.

The equivalent SQL function is **CalRange**.

Returns

A list of time stamps.

Related reference:

"The CalIndex function" on page 6-2
"The CalRange function" on page 6-3
"The CalStamp function" on page 6-4
"The ts_cal_index() function" on page 9-8
"The ts_cal_range_index() function"
"The ts_time() function" on page 9-49
"The ts_cal_stamp() function" on page 9-11

The ts_cal_range_index() function

The **ts_cal_range_index()** function returns a list containing a specified number of time stamps starting at a given time stamp.

Syntax

```
MI_COLLECTION *
ts_cal_range_index (MI_CONNECTION, *conn,
                    mi_string      *cal_name,
                    mi_datetime    *begin_stamp,
                    mi_integer     num_stamps)
```

conn A valid DataBlade API connection.

cal_name
The name of the calendar.

begin_stamp
The beginning of the range. It must be greater than or equal to the calendar origin.

num_stamps
The number of time stamps to return.

Description

This function is useful if you must print out the time stamps of a series of regular time series elements. If the range is known, getting an array of all of the time stamps is more efficient than using **ts_time()** on each element.

The caller is responsible for freeing the result of this function.

Returns

A list of time stamps.

Related reference:

"The CalIndex function" on page 6-2
"The CalRange function" on page 6-3
"The CalStamp function" on page 6-4
"The ts_cal_index() function" on page 9-8
"The ts_cal_range() function" on page 9-9
"The ts_cal_stamp() function"
"The ts_time() function" on page 9-49

The ts_cal_stamp() function

The **ts_cal_stamp()** function returns the time stamp at a given number of calendar intervals before or after a given time stamp. The returned time stamp is located in allocated memory, so the caller should free it using **mi_free()**.

Syntax

```
mi_datetime *
ts_cal_stamp (MI_CONNECTION *conn,
              mi_string     *cal_name,
              mi_datetime   *tstamp,
              mi_integer    offset)
```

conn A valid DataBlade API connection.

cal_name
The name of the calendar.

tstamp The input time stamp.

offset The number of calendar intervals before or after the input time stamp. Use a negative number to indicate an offset before the specified time stamp and a positive number to indicate an offset after the specified time stamp.

Description

The equivalent SQL function is **CalStamp**.

Returns

The time stamp representing the given offset, which must be freed by the caller.

Related reference:

"The CalIndex function" on page 6-2
"The CalRange function" on page 6-3
"The CalStamp function" on page 6-4
"The ts_cal_index() function" on page 9-8
"The ts_cal_range_index() function" on page 9-10
"The ts_cal_range() function" on page 9-9

The ts_cal_startdate() function

The **ts_cal_startdate()** function returns the start date of a calendar.

Syntax

```
mi_datetime *
ts_cal_startdate (MI_CONNECTION *conn,
                  mi_string     *cal_name)
```

conn A pointer to a valid DataBlade API connection structure.

cal_name
The name of the calendar.

Description

The equivalent SQL function is **CalStartDate**.

Returns

An mi_datetime pointer that points to the start date of a calendar. You must free this value after use.

Related reference:

"The CalStartDate function" on page 6-5

"The ts_cal_pattstartdate() function" on page 9-8

The ts_close() function

The **ts_close()** procedure closes the associated time series.

Syntax

```
void
ts_close(ts_tsdesc *tsdesc)
```

tsdesc A time series descriptor returned by **ts_open**.

Description

After a call to this procedure, *tsdesc* is no longer valid and so should not be passed to any routine requiring the *tsdesc* argument.

Example

See the **ts_interp()** function, Appendix A, "The Interp function example," on page A-1, for an example of **ts_close()**.

Related reference:

"The ts_open() function" on page 9-41

The ts_closest_elem() function

The **ts_closest_elem()** function returns the first element, or column(s) of an element, that is non-null and closest to the given time stamp.

Syntax

```
ts_tselem
ts_closest_elem(ts_tsdesc *tdesc,
       mi_datetime      *tstamp,
       mi_string        *cmp,
       mi_string        *col_list,
       mi_integer        flags, mi_integer *isNull,
       mi_integer       *off)
```

tdesc A time series descriptor returned by **ts_open**.

tstamp The time stamp to start searching from.

cmp A comparison operator. Valid values for *cmp* are <, <=, =, ==, '>=, and >.

col_list To search for an element with a particular set of columns non-null, specify a list of column names separated by a vertical bar (|). An error is raised if any of the column names do not exist in the time series sub-rowtype.

To search for a non-null element, set *col_list* to `NULL`.

flags Determines whether hidden elements should be returned. Valid values for the *flags* parameter are defined in `tseries.h`. They are:

- TS_CLOSEST_NO_FLAGS (no special flags)
- TS_CLOSEST_RETNULLS_FLAGS (return hidden elements)

isNull The *isNull* parameter must not be `NULL`. On return, it is set with the null indicator bits found in `tseries.h`. These are:

- 0 (element is not hidden and is allocated)
- TS_NULL_NOTALLOCED (element has not been written to)
- TS_NULL_HIDDEN (element is hidden)

off If the time series is regular, the offset of the returned element will be returned in the *off* parameter, if *off* is not `NULL`.

Description

The search algorithm that **ts_closest_elem** uses is as follows:

- If *cmp* is any of <=, =, ==, or >=, the search starts at *tstamp*.
- If *cmp* is <, the search starts at the first element before *tstamp*.
- If *cmp* is >, the search starts at the first element after *tstamp*.

The *tstamp* and *cmp* parameters are used to determine where to start the search. The search continues in the direction indicated by *cmp* until an element is found that qualifies. If no element qualifies, then the return value is `NULL`.

Important: For irregular time series, values in an irregular element persist until the next element. This means that any of the previous “equals” operations on an irregular time series will look for <= first. If *cmp* is >= and the <= operations fails, the operation then looks forward for the next element; otherwise, `NULL` is returned.

Returns

An element that meets the criteria described.

The ts_col_cnt() function

The **ts_col_cnt()** function returns the number of columns contained in each element of a time series.

Syntax

```
mi_integer
ts_col_cnt (ts_tsdesc *tsdesc)
```

tsdesc A time series descriptor returned by **ts_open**.

Returns

The number of columns.

Related reference:

"The ts_get_all_cols() procedure" on page 9-25

The ts_col_id() function

The **ts_col_id()** function takes a column name and returns the associated column number.

Syntax

```
mi_integer
ts_col_id(ts_tsdesc *tsdesc,
         mi_string *colname)
```

tsdesc A time series descriptor returned by **ts_open()**.

colname
The name of the column.

Description

Column numbers start at 0; therefore, the first time stamp column is always column 0.

Returns

The number of the column associated with *colname*.

Related reference:

"The ts_colinfo_name() function"
"The ts_colinfo_number() function" on page 9-15

The ts_colinfo_name() function

The **ts_colinfo_name()** function gets type information for a column in a time series.

Syntax

```
ts_typeinfo *
ts_colinfo_name (ts_tsdesc *tsdesc,
                 mi_string *colname)
```

tsdesc A time series descriptor returned by **ts_open()**.

colname
The name of the column to return information for.

Description

The resulting **typeinfo** structure and its **ti_typename** field must be freed by the caller.

Returns

A pointer to a **ts_typeinfo** structure. This structure is defined as follows:

```
typedef struct _ts_typeinfo
{
   MI_TYPEID        *ti_typeid; /* type id */
   mi_integer        ti_typelen; /* internal length */
   mi_smallint        ti_typealign; /* internal alignment */
   mi_smallint        ti_typebyvalue; /* internal byvalue flag */
   mi_integer        ti_typebound; /* internal bound */
   mi_integer        ti_typeparameter; /* internal parameter */
   mi_string      *ti_typename; /* name of the column */
} ts_typeinfo;
```

Related reference:

"The ts_col_id() function" on page 9-14

"The ts_colinfo_number() function"

The ts_colinfo_number() function

The **ts_colinfo_number()** function gets type information for a column in a time series.

Syntax

```
ts_typeinfo *
ts_colinfo_number (ts_tsdesc  *tsdesc,
                   mi_integer id)
```

tsdesc A time series descriptor returned by **ts_open()**.

id The column number to return information for. The *id* argument must be greater than or equal to 0 and less than the number of columns in a time series element. An *id* of 0 corresponds to the time stamp column.

Description

The resulting **typeinfo** structure and its **ti_typename** field must be freed by the caller.

Returns

A pointer to a **ts_typeinfo** structure. This structure is defined as follows:

```
typedef struct _ts_typeinfo
{
   MI_TYPEID        *ti_typeid; /* type id */
   mi_integer        ti_typelen; /* internal length */
   mi_smallint        ti_typealign; /* internal alignment */
   mi_smallint        ti_typebyvalue; /* internal byvalue flag */
   mi_integer        ti_typebound; /* internal bound */
   mi_integer        ti_typeparameter; /* internal parameter */
   mi_string         *ti_typename; /* name of the column */
} ts_typeinfo;
```

Example

See the **ts_interp()** function, Appendix A, "The Interp function example," on page A-1, for an example of **ts_colinfo_number()**.

Related reference:

"The ts_col_id() function" on page 9-14

"The ts_colinfo_name() function" on page 9-14

The ts_copy() function

The **ts_copy()** function makes and returns a copy of the given time series of the type in the *type_id* argument.

Syntax

```
ts_timeseries *
ts_copy(MI_CONNECTION *conn,
       ts_timeseries *ts,
       MI_TYPEID     *typeid)
```

conn A valid DataBlade API connection.

ts The time series to be copied.

typeid The ID of the row type of the time series to be copied.

Description

Since values returned by **mi_value()** are valid only until the next **mi_next_row()** or **mi_query_finish()** call, it is sometimes necessary to use **ts_copy()** to access a time series outside an **mi_get_result()** loop.

On the client, you must use the **ts_copy()** function to make a copy of a time series before you pass the time series as an argument to the **mi_exec_prepare()** statement.

Returns

A copy of the given time series. This value must be freed by the user by calling **ts_free()**.

Related reference:

"The ts_free() procedure" on page 9-24

"The ts_get_typeid() function" on page 9-31

The ts_create() function

The **ts_create()** function creates a time series.

Syntax

```
ts_timeseries *
ts_create(MI_CONNECTION *conn,
          mi_string     *calname,
          mi_datetime   *origin,
          mi_integer    threshold,
          mi_integer    flags,
          MI_TYPEID     *typeid,
          mi_integer    nelem,
          mi_string      *container)
```

conn A valid DataBlade API connection.

calname
The name of the calendar.

origin The time series origin.

threshold
: The time series threshold. If the time series stores this number or more elements, it is stored in a container. If the time series holds fewer than this number, it is stored directly in the row that contains it. *threshold* must be greater than or equal to 0 and less than 256.

flags Must be 0 for regular time series and TS_CREATE_IRR for irregular time series.

typeid The ID of the new type for the time series to be created.

nelems The initial number of elements to create space for in the time series. This space is reclaimed if not used, after the time series is written into the database.

container
: The container for holding the time series. Can be `NULL` if the time series can fit in a row or is not going to be assigned to a table.

Description

The equivalent SQL function is **TSCreate** or **TSCreateIrr**.

Returns

A pointer to a new time series. The user can free this value by calling **ts_free()**.

Related reference:

"TSCreate function" on page 7-92
"The ts_free() procedure" on page 9-24
"The ts_open() function" on page 9-41
"The ts_get_threshold() function" on page 9-30
"The ts_get_typeid() function" on page 9-31

The ts_create_with_metadata() function

The **ts_create_with_metadata()** function creates a time series with user-defined metadata attached.

Syntax

```
ts_timeseries *
ts_create_with_metadata(MI_CONNECTION  *conn,
                        mi_string      *calname,
                        mi_datetime    *origin,
                        mi_integer     threshold,
                        mi_integer     flags,
                        MI_TYPEID      *typeid,
                        mi_integer     nelem,
                        mi_string      *container,
                        mi_lvarchar    *metadata,
                        MI_TYPEID       *metadata_typeid)
```

conn A valid DataBlade API connection.

calname
: The name of the calendar.

origin The time series origin.

threshold
: The time series threshold. If the time series stores this number or more

elements, it is stored in a container. If the time series holds fewer than this number, it is stored directly in the row that contains it. *threshold* must be greater than or equal to 0 and less than 256.

flags Must be 0 for regular time series and TS_CREATE_IRR for irregular time series.

typeid The ID of the new type for the time series to be created.

nelems The initial number of elements to create space for in the time series. This space is reclaimed if not used, after the time series is written into the database.

container
The container for holding the time series. This parameter can be `NULL` if the time series can fit in a row or is not going to be assigned to a table.

metadata
The metadata to be put into the time series. See “Creating a time series with metadata” on page 3-13 for more information about metadata. Can be `NULL`.

metadata_typeid
The type ID of the metadata. Can be `NULL` if the metadata argument is `NULL`.

Description

This function behaves the same as **ts_create()**, plus it saves the supplied metadata in the time series. The metadata can be `NULL` or a zero-length LVARCHAR; if either, **ts_create_with_metadata()** acts exactly like **ts_create()**. If the *metadata* pointer points to valid data, the *metadata_typeid* parameter must be a valid pointer to a valid type ID for a user-defined type.

The equivalent SQL function is **TSCreate** or **TSCreateIrr**.

Returns

A pointer to a new time series. The user can free this value by calling **ts_free()**.

Related reference:

“GetMetaData function” on page 7-47

“GetMetaTypeName function” on page 7-47

“UpdMetaData function” on page 7-114

“TSCreate function” on page 7-92

“The ts_free() procedure” on page 9-24

“The ts_open() function” on page 9-41

“The ts_get_metadata() function” on page 9-28

“The ts_get_typeid() function” on page 9-31

“The ts_update_metadata() function” on page 9-52

The ts_current_offset() function

The **ts_current_offset()** function returns the offset for the last element returned by **ts_next()**.

Syntax

```
mi_integer
ts_current_offset(ts_tscan *tscan)
```

tscan The scan descriptor returned by **ts_begin_scan()**.

Returns

The offset of the last element returned. If no element has been returned yet, the offset of the first element is returned. For irregular time series, **ts_current_offset()** always returns -1.

Related reference:

"The ts_begin_scan() function" on page 9-6

The ts_current_timestamp() function

The **ts_current_timestamp()** function finds the time stamp that corresponds to the current element retrieved from the scan.

Syntax

```
mi_datetime *
ts_current_timestamp(ts_tscan *scan)
```

scan The scan descriptor returned by **ts_begin_scan()**.

Returns

If no elements have been retrieved, the value returned is the time stamp of the first element. This value cannot be freed by the user with **mi_free()**.

Related reference:

"The ts_begin_scan() function" on page 9-6

The ts_datetime_cmp() function

The **ts_datetime_cmp()** function compares two time stamps and returns a value that indicates whether *tstamp1* is before, equal to, or after *tstamp2*.

Syntax

```
mi_integer
ts_datetime_cmp(mi_datetime *tstamp1,
                mi_datetime *tstamp2)
```

tstamp1
The first time stamp to compare.

tstamp2
The second time stamp to compare.

Returns

< 0 If *tstamp1* comes before *tstamp2*.

0 If *tstamp1* equals *tstamp2*.

> 0 If *tstamp1* comes after *tstamp2*.

Related reference:

"The ts_get_all_cols() procedure" on page 9-25

"The ts_get_col_by_name() function" on page 9-26

"The ts_get_col_by_number() function" on page 9-26

The ts_del_elem() function

The **ts_del_elem()** function deletes an element from a time series at a given timepoint.

Syntax

```
ts_timeseries *
ts_del_elem(ts_tsdesc    *tsdesc,
            mi_datetime  *tstamp)
```

tsdesc The time series descriptor returned by **ts_open()**.

tstamp The timepoint from which to delete the element.

Description

If there is no element at the timepoint, no error is raised, and no change is made to the time series. It is an error to delete a hidden element.

The equivalent SQL function is **DelElem**.

Returns

The original time series minus the element deleted, if there was one.

Related reference:

"DelElem function" on page 7-33

"The ts_ins_elem() function" on page 9-33

"The ts_put_elem() function" on page 9-44

"The ts_upd_elem() function" on page 9-52

The ts_elem() function

The **ts_elem()** function returns an element from the time series at the given time.

Syntax

```
ts_tselem
ts_elem(ts_tsdesc   *tsdesc,
        mi_datetime *tstamp,
        mi_integer  *STATUS,
        mi_integer  *off)
```

tsdesc The time series descriptor returned by **ts_open()**.

tstamp A pointer to the time stamp for the desired element.

STATUS

Set on return to indicate whether the element is NULL or hidden. See "The ts_hide_elem() function" on page 9-31 for an explanation of the *isNull* argument.

off For regular time series, *off* is set to the offset on return. If the time series is irregular, or if the time stamp is not in the calendar, *off* is set to -1. The offset can be `NULL`.

Description

On return, *off* is filled in with the offset of the element for a regular time series or -1 for an irregular time series. The element is overwritten after two calls to fetch elements using this *tsdesc* (time series descriptor).

The equivalent SQL function is **GetElem**.

Returns

An element, its offset, and whether it is hidden, `NULL`, or both. This element must not be freed by the caller.

Related reference:

"GetElem function" on page 7-40
"DelElem function" on page 7-33
"The TS_ELEM_HIDDEN macro"
"The ts_hide_elem() function" on page 9-31
"The ts_last_elem() function" on page 9-34
"The ts_nth_elem() function" on page 9-41
"The TS_ELEM_NULL macro" on page 9-22
"The ts_first_elem() function" on page 9-23
"The ts_ins_elem() function" on page 9-33
"The ts_make_elem() function" on page 9-36
"The ts_put_elem() function" on page 9-44
"The ts_put_elem_no_dups() function" on page 9-45
"The ts_put_last_elem() function" on page 9-46
"The ts_put_nth_elem() function" on page 9-46
"The ts_upd_elem() function" on page 9-52

The TS_ELEM_HIDDEN macro

The TS_ELEM_HIDDEN macro determines whether the STATUS indicator returned by **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, and similar functions is set because the associated element was hidden.

Syntax

```
TS_ELEM_HIDDEN((mi_integer) STATUS)
```

STATUS
The *mi_integer* argument previously passed to **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, or a similar function.

Description

This macro returns a nonzero value if the associated element is hidden. This macro is often used in concert with TS_ELEM_NULL.

Returns

A nonzero value if the element associated with the *STATUS* argument was previously hidden by the **ts_hide_elem()** function.

Related reference:

“The ts_elem() function” on page 9-20

“The TS_ELEM_NULL macro”

“The ts_first_elem() function” on page 9-23

“The ts_hide_elem() function” on page 9-31

“The ts_last_elem() function” on page 9-34

“The ts_next() function” on page 9-39

“The ts_next_valid() function” on page 9-40

“The ts_nth_elem() function” on page 9-41

“The ts_previous_valid() function” on page 9-43

The TS_ELEM_NULL macro

The TS_ELEM_NULL macro determines whether the STATUS indicator returned by **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, or a similar function is `NULL` because the associated element is `NULL`.

Syntax

```
TS_ELEM_NULL((mi_integer) STATUS)
```

STATUS
: The *mi_integer* argument previously passed to **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, or a similar function.

Description

This macro returns a nonzero value if the associated element is `NULL`. This macro is often used in concert with TS_ELEM_HIDDEN.

Returns

A nonzero value if the element returned by **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, or similar function was `NULL`.

Related reference:

“The TS_ELEM_HIDDEN macro” on page 9-21

“The ts_elem() function” on page 9-20

“The ts_first_elem() function” on page 9-23

“The ts_hide_elem() function” on page 9-31

“The ts_last_elem() function” on page 9-34

“The ts_next() function” on page 9-39

“The ts_next_valid() function” on page 9-40

“The ts_nth_elem() function” on page 9-41

“The ts_previous_valid() function” on page 9-43

The ts_elem_to_row() function

The **ts_elem_to_row()** function converts a time series element into a new row.

Syntax

```
MI_ROW *
ts_elem_to_row(ts_tsdesc  *tsdesc,
                 ts_tselem  elem,
              mi_integer off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

elem A time series element. It must agree in type with the time series described by *tsdesc*.

off If the time series is regular and *off* is non-negative, *off* is used to compute the time stamp value placed in the first column of the returned row.

If the time series is regular and *off* is negative, column 0 of the resulting row will be taken from column 0 of the *elem* parameter (which will be `NULL` if the element was created for or extracted from a regular time series).

If the time series is irregular, the *off* parameter is ignored.

Returns

A row. The row must be freed by the caller using the **mi_row_free()** procedure.

Related reference:

"The ts_free_elem() procedure" on page 9-24

"The ts_make_elem() function" on page 9-36

"The ts_make_elem_with_buf() function" on page 9-37

"The ts_row_to_elem() function" on page 9-48

The ts_end_scan() procedure

The **ts_end_scan()** procedure ends a scan of a time series. It releases resources acquired by **ts_begin_scan()**. Upon return, no more elements can be retrieved using the given **ts_tscan** pointer.

Syntax

```
void
ts_end_scan(ts_tscan *scan)
```

scan The scan to be ended.

Example

See the **ts_interp()** function, Appendix A, "The Interp function example," on page A-1, for an example of **ts_end_scan()**.

Related reference:

"The ts_begin_scan() function" on page 9-6

The ts_first_elem() function

The **ts_first_elem()** function returns the first element in the time series.

Syntax

```
ts_tselem
ts_first_elem(ts_tsdesc  *tsdesc,
             mi_integer *STATUS)
```

tsdesc The time series descriptor returned by **ts_open()**.

STATUS
: A pointer to an **mi_integer** value. See "The ts_hide_elem() function" on page 9-31 for an explanation of the *STATUS* argument.

Description

If the time series is regular, the first element is always the origin of the time series. If the time series is irregular, the first element is the one with the earliest time stamp. The value must not be freed by the caller. The element is overwritten after two calls to fetch elements using this *tsdesc* (time series descriptor).

The equivalent SQL function is **GetFirstElem**.

Returns

The first element in the time series.

Related reference:

"GetFirstElem function" on page 7-42

"The TS_ELEM_HIDDEN macro" on page 9-21

"The TS_ELEM_NULL macro" on page 9-22

"GetElem function" on page 7-40

"The ts_begin_scan() function" on page 9-6

"The ts_elem() function" on page 9-20

"The ts_next() function" on page 9-39

"The ts_next_valid() function" on page 9-40

The ts_free() procedure

The **ts_free()** procedure frees all memory associated with the given time series argument. The time series argument must have been generated by a call to either **ts_create()** or **ts_copy()**.

Syntax

```
void
ts_free(ts_timeseries *ts)
```

ts The source time series.

Related reference:

"The ts_copy() function" on page 9-16

"The ts_create() function" on page 9-16

"The ts_create_with_metadata() function" on page 9-17

"The ts_get_ts() function" on page 9-30

"The ts_ins_elem() function" on page 9-33

The ts_free_elem() procedure

The **ts_free_elem()** procedure frees a time series element, releasing its resources. It is used to free elements created by **ts_make_elem()** or **ts_row_to_elem()**. It must not be called to free elements returned by **ts_elem()**, **ts_first_elem()**, **ts_last_elem()**, **ts_last_valid()**, **ts_next()**, **ts_next_valid()**, **ts_nth_elem()**, or **ts_previous_valid()**; those elements are overwritten with subsequent calls or freed when the corresponding scan or time series descriptor is closed.

Syntax

```
void
ts_free_elem(ts_tsdesc *tsdesc,
              ts_tselem elem)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

elem A time series element. It must agree in type with the time series described by *tsdesc*.

Related reference:

"The ts_elem_to_row() function" on page 9-22

"The ts_make_elem() function" on page 9-36

"The ts_make_elem_with_buf() function" on page 9-37

"The ts_row_to_elem() function" on page 9-48

The ts_get_all_cols() procedure

The **ts_get_all_cols()** procedure loads the values in the element into the *values* and *nulls* arrays.

Syntax

```
void
ts_get_all_cols(ts_tsdesc  *tsdesc,
                 ts_tselem  tselem,
                 MI_DATUM   *values,
                 mi_boolean *nulls,
                 mi_integer off)
```

tsdesc A time series pointer returned by **ts_open()**.

tselem The element to extract data from.

values The array to put the column data into. This array must be large enough to hold data for all the columns of the time series.

nulls An array that indicates null values.

off For a regular time series, *off* is the offset of the element. For an irregular time series, *off* is ignored.

Returns

None. The *values* and *nulls* arrays are filled in with data from the element. The *values* array is filled with values or pointers to values depending on whether the corresponding column is by reference or by value. The values in the *values* array must not be freed by the caller.

Related reference:

"The ts_datetime_cmp() function" on page 9-19

"The ts_col_cnt() function" on page 9-13

The ts_get_calname() function

The **ts_get_calname()** function returns the name of the calendar associated with the given time series.

Syntax

```
mi_string *
ts_get_calname(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetCalendarName**.

Returns

The name of the calendar. This value must be freed by the caller with **mi_free()**.

The ts_get_col_by_name() function

The **ts_get_col_by_name()** function pulls out the individual piece of data from an element in the column with the given name.

Syntax

```
MI_DATUM
ts_get_col_by_name(ts_tsdesc  *tsdesc,
                    ts_tselem  tselem,
                    mi_string  *colname,
                    mi_boolean *isNull,
                    mi_integer off)
```

tsdesc A pointer returned by **ts_open()**.

tselem An element to get column data from.

colname
The name of the column in the element.

isNull A pointer to a null indicator.

off For a regular time series, *off* is the offset of the element in the time series. For an irregular time series, *off* is ignored.

Returns

The data in the specified column. If the type of the column is by reference, a pointer is returned. If the type is by value, the data itself is returned. The caller cannot free this value. On return, *isNull* is set to indicate whether the column is NULL.

Related reference:

"The ts_datetime_cmp() function" on page 9-19

"The ts_get_col_by_number() function"

The ts_get_col_by_number() function

The **ts_get_col_by_number()** function pulls the individual pieces of data from an element. The column 0 (zero) is always the time stamp.

Syntax

```
MI_DATUM
ts_get_col_by_number(ts_tsdesc *tsdesc,
                      ts_tselem  tselem,
                      mi_integer colnumber,
                      mi_boolean *isNull,
                      mi_integer off)
```

tsdesc A pointer returned by **ts_open()**.

tselem An element to get column data from.

colnumber
The column number. Column numbers start at 0, which represents the time stamp.

isNull A pointer to a null indicator.

off For a regular time series, *off* is the offset of the element in the time series. For an irregular time series, *off* is ignored.

Returns

The data in the specified column. If the type of the column is by reference, a pointer is returned. If the type is by value, the data itself is returned. The caller cannot free this value. On return, *isNull* is set to indicate whether the column is `NULL`.

Example

See the **ts_interp()** function, Appendix A, "The Interp function example," on page A-1, for an example of **ts_get_col_by_number()**.

Related reference:

"The ts_datetime_cmp() function" on page 9-19

"The ts_get_col_by_name() function" on page 9-26

The ts_get_containername() function

The **ts_get_containername()** function gets the container name of the given time series.

Syntax

```
mi_string *
ts_get_containername(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetContainerName**.

Returns

The name of the container for the given time series. This value must not be freed by the user.

Related reference:

"GetContainerName function" on page 7-40

The ts_get_flags() function

The **ts_get_flags()** function returns the flags associated with the given time series.

Syntax

```
mi_integer
ts_get_flags(ts_timeseries *ts)
```

ts The source time series.

Description

The return value is a collection of flag bits. The possible flag bits set are TSFLAGS_IRR, TSFLAGS_INMEM, and TSFLAGS_ASSIGNED.

To check whether the time series is regular, use TS_IS_IRREGULAR.

Returns

An integer containing the flags for the given time series.

Related reference:

"IsRegular function" on page 7-61

"The TS_IS_IRREGULAR macro" on page 9-34

The ts_get_metadata() function

The **ts_get_metadata()** function returns the user-defined metadata and its type ID from the specified time series.

Syntax

```
mi_lvarchar *
ts_get_metadata(ts_timeseries *ts,
                MI_TYPEID     **metadata_typeid)
```

ts The time series to retrieve the metadata from.

metadata_typeid
The return parameter to hold the type ID of the user-defined metadata.

Description

The equivalent SQL function is **GetMetaData**.

Returns

The user-defined metadata contained in the specified time series. If the time series does not contain any user-defined metadata, then NULL is returned and the *metadata_typeid* pointer is set to NULL. This return value must be cast to the real user-defined type to be useful. The value returned can be freed by the caller with **mi_var_free()**.

Related reference:

"GetMetaData function" on page 7-47

"GetMetaTypeName function" on page 7-47

"UpdMetaData function" on page 7-114

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

"The ts_create_with_metadata() function" on page 9-17

"The ts_get_metadata() function"

"The ts_update_metadata() function" on page 9-52

The ts_get_origin() function

The **ts_get_origin()** function returns the origin of the given time series.

Syntax

```
mi_datetime *
ts_get_origin(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetOrigin**.

Returns

The origin of the given time series. This value must be freed by the caller using **mi_free()**.

Related reference:

"GetOrigin function" on page 7-52

The ts_get_stamp_fields() procedure

The **ts_get_stamp_fields()** procedure takes a pointer to an **mi_datetime** structure and returns the parameters with the year, month, day, hour, minute, second, and microsecond.

Syntax

```
void
ts_get_stamp_fields (MI_CONNECTION *conn,
                     mi_datetime   *dt,
                     mi_integer    *year,
                     mi_integer    *month,
                     mi_integer    *day,
                     mi_integer    *hour,
                     mi_integer    *minute,
                     mi_integer    *second,
                     mi_integer    *ms)
```

conn A valid DataBlade API connection.

dt The time stamp to convert.

year Pointer to year integer that the procedure sets. Can be NULL.

month Pointer to month integer that the procedure sets. Can be NULL.

day Pointer to day integer that the procedure sets. Can be NULL.

hour Pointer to hour integer that the procedure sets. Can be NULL.

minute Pointer to minute integer that the procedure sets. Can be NULL.

second Pointer to second integer that the procedure sets. Can be NULL.

ms Pointer to microsecond integer that the procedure sets. Can be NULL.

Returns

On return, the non-null year, month, day, hour, minute, second, and microsecond are set to the time that corresponds to the time indicated by the *dt* argument.

Related reference:

"The ts_make_stamp() function" on page 9-38

The ts_get_threshold() function

The **ts_get_threshold()** function returns the threshold of the specified time series.

Syntax

```
mi_integer
ts_get_threshold(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetThreshold**.

Returns

The threshold of the given time series.

Related reference:

"GetThreshold function" on page 7-54
"The ts_create() function" on page 9-16

The ts_get_ts() function

The **ts_get_ts()** function returns a pointer to the time series associated with the given time series descriptor.

Syntax

```
ts_timeseries *
ts_get_ts(ts_tsdesc *tsdesc)
```

tsdesc The time series descriptor from **ts_open()**.

Description

The **ts_get_ts()** function is useful when you must call a function that takes a time series argument (for example, **ts_get_calname()**), but you only have a *tsdesc* (time series descriptor).

Returns

A pointer to the time series associated with the given time series descriptor. This value can be freed by the caller after **ts_close()** has been called if the original time series was created by **ts_create()** or **ts_copy()**. To free it, use **ts_free()**.

Related reference:

"The ts_free() procedure" on page 9-24

"The ts_put_elem() function" on page 9-44

"The ts_put_elem_no_dups() function" on page 9-45

"The ts_put_last_elem() function" on page 9-46

"The ts_put_nth_elem() function" on page 9-46

The ts_get_typeid() function

The **ts_get_typeid()** function returns the type ID of the specified time series.

Syntax

```
mi_typeid *
ts_get_typeid(MI_CONNECTION *conn,
              ts_timeseries *ts)
```

conn A valid DataBlade API connection.

ts The source time series.

Description

This function returns the type ID of the specified time series. Usually, a time series type ID is located in an MI_FPARAM structure. This function is useful when there is no easy access to an MI_FPARAM structure.

Returns

A pointer to an MI_TYPEID structure that contains the type ID of the specified time series. You must not free this value after use.

Related reference:

"The ts_copy() function" on page 9-16

"The ts_create() function" on page 9-16

"The ts_create_with_metadata() function" on page 9-17

"The ts_open() function" on page 9-41

The ts_hide_elem() function

The **ts_hide_elem()** function marks the element at the given time stamp as invisible to a scan unless TS_SCAN_HIDDEN is set.

Syntax

```
ts_timeseries
ts_hide_elem(ts_tsdesc   *tsdesc,
             mi_datetime *tstamp)
```

tsdesc The time series descriptor returned by **ts_open()** for the source time series.

tstamp The time stamp to be made invisible to the scan.

Description

When an element is hidden, element retrieval API functions such as **ts_elem()** and **ts_nth_elem()** return the hidden element; however, their *STATUS* argument has the TS_NULL_HIDDEN bit set. The values for the element's *STATUS* argument are:

- If *STATUS* is TS_NULL_HIDDEN, the element is hidden.
- If *STATUS* is TS_NULL_NOTALLOCED, the element is NULL.
- If *STATUS* is both TS_NULL_HIDDEN and TS_NULL_NOTALLOCED, the element is both hidden and NULL.
- If *STATUS* is 0 (zero), the element is not hidden and is not NULL.

The TS_ELEM_HIDDEN and TS_ELEM_NULL macros are provided to check the value of *STATUS*.

Hidden elements cannot be modified; they must be revealed first using **ts_reveal_elem()**.

The equivalent SQL function is **HideElem**.

Returns

The modified time series. If there is no element at the given time stamp, an error is raised.

Related reference:

"HideElem function" on page 7-55
"The ts_elem() function" on page 9-20
"The TS_ELEM_HIDDEN macro" on page 9-21
"The TS_ELEM_NULL macro" on page 9-22
"The ts_reveal_elem() function" on page 9-48
"The ts_previous_valid() function" on page 9-43

The ts_index() function

The **ts_index()** function converts from a time stamp to an index (offset) for a regular time series.

Syntax

```
mi_integer
ts_index(ts_tsdesc   *tsdesc,
         mi_datetime *tstamp)
```

tsdesc The time series descriptor returned by **ts_open()**.

tstamp The time stamp to convert.

Description

Consider a time series that starts on Monday, January 1 and keeps track of weekdays. Calling **ts_index()** with a time stamp argument that corresponds to Monday, January 1, would return 0; a time stamp argument corresponding to Tuesday, January 2, would return 1; a time stamp argument corresponding to Monday, January 8, would return 5; and so on.

The equivalent SQL function is **GetIndex**.

Returns

An offset into the time series. If the time stamp falls before the time series origin, or if it is not a valid point in the calendar, -1 is returned; otherwise, the return value is always a positive integer.

Related reference:

"GetIndex function" on page 7-43

"The ts_cal_index() function" on page 9-8

"The ts_nth_elem() function" on page 9-41

"The ts_put_nth_elem() function" on page 9-46

"The ts_time() function" on page 9-49

The ts_ins_elem() function

The **ts_ins_elem()** function puts an element into an existing time series at a given timepoint.

Syntax

```
ts_timeseries *
ts_ins_elem(ts_tsdesc   *tsdesc,
            ts_tselem   tselem,
            mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be modified, returned by **ts_open()**.

tselem The element to add.

tstamp The timepoint at which to add the element. The time stamp column of the *tselem* is ignored.

Description

The equivalent SQL function is **InsElem**.

Returns

The original time series with the new element added. If the time stamp is not a valid timepoint in the time series, an error is raised. If there is already an element at the given time stamp, an error is raised.

Related reference:
"InsElem function" on page 7-57
"The ts_del_elem() function" on page 9-20
"The ts_elem() function" on page 9-20
"The ts_free() procedure" on page 9-24
"The ts_make_elem() function" on page 9-36
"The ts_make_elem_with_buf() function" on page 9-37
"The ts_put_elem() function" on page 9-44
"The ts_upd_elem() function" on page 9-52
"The ts_put_elem_no_dups() function" on page 9-45

The TS_IS_INCONTAINER macro

The TS_IS_INCONTAINER macro determines whether the time series data is stored in a container.

Syntax

```
TS_IS_INCONTAINER((ts_timeseries *) ts)
```

ts A pointer to a time series.

Returns

This function returns nonzero if the time series data is in a container, rather than in memory or in a row.

The TS_IS_IRREGULAR macro

The TS_IS_IRREGULAR macro determines whether the given time series is irregular.

Syntax

```
TS_IS_IRREGULAR((ts_timeseries *) ts)
```

ts A pointer to a time series.

Returns

A nonzero value if the given time series is irregular; otherwise, 0 is returned.

Related reference:
"The ts_get_flags() function" on page 9-27

The ts_last_elem() function

The **ts_last_elem()** function returns the last element from a time series.

Syntax

```
ts_tselem
ts_last_elem(ts_tsdesc *tsdesc,
             mi_integer *STATUS,
             mi_integer *off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

STATUS
: A pointer to a **mi_integer** value. See "The ts_hide_elem() function" on page 9-31 for a description of *STATUS*.

off
: If the time series is regular, *off* is set to the offset of the returned element. If the time series is irregular, or if the time series is empty, *off* is set to -1. This argument can be passed in as `NULL`.

Description

This function fills in *off* with the element's offset if *off* is not `NULL` and the time series is regular, and it sets *STATUS* to indicate if the element is `NULL` or hidden.

The equivalent SQL function is **GetLastElem**.

Returns

The last element of the specified time series, its offset, and whether it is `NULL` or hidden. If the time series is irregular, the offset is set to -1. This value must not be freed by the caller. The element is overwritten after two calls to fetch elements with this *tsdesc* (time series descriptor).

Related reference:

"GetLastElem function" on page 7-44
"The ts_elem() function" on page 9-20
"The TS_ELEM_HIDDEN macro" on page 9-21
"The TS_ELEM_NULL macro" on page 9-22
"The ts_nth_elem() function" on page 9-41
"The ts_upd_elem() function" on page 9-52

The ts_last_valid() function

The **ts_last_valid()** function extracts the entry for a particular timepoint.

Syntax

```
ts_tselem
ts_last_valid(ts_tsdesc   *tsdesc,
              mi_datetime *tstamp,
              mi_integer  *STATUS,
              mi_integer  *off)
```

tsdesc
: The descriptor for a time series returned by **ts_open()**.

tstamp
: The time stamp of interest.

STATUS
: A pointer to an **mi_integer** value. See "The ts_hide_elem() function" on page 9-31 for a description of *STATUS*.

off
: If the time series is regular, *off* is set to the offset of the returned element. If the time series is irregular, or if the time series is empty, *off* is set to -1. This argument can be passed as `NULL`.

Description

For regular time series, this function returns the first element with a time stamp less than or equal to *tstamp*. For irregular time series, it returns the latest element at or preceding the given time stamp.

Returns

The nearest element at or before the given time stamp. If there is no such element before the time stamp, NULL is returned.

NULL is returned if:
- The element at the timepoint is NULL and the time series is regular.
- The timepoint is before the origin.
- The time series is irregular and there are no elements at or before the given time stamp.

This element must not be freed by the caller; it is valid until the next element is fetched from the descriptor.

Related reference:

"GetLastValid function" on page 7-46

"The ts_previous_valid() function" on page 9-43

The ts_make_elem() function

The **ts_make_elem()** function makes an element from an array of values and nulls. Each array has one value for each column in the element.

Syntax

```
ts_tselem
ts_make_elem(ts_tsdesc *tsdesc,
             MI_DATUM   *values,
             mi_boolean *nulls,
             mi_integer *off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

values An array of data to be placed in the element. Data that is by value is placed in the array, and data that is by reference stores pointers.

nulls Stores columns in the element that should be NULL.

off For a regular time series, *off* contains the offset of the element on return. For an irregular time series, *off* is set to -1. This argument can be NULL.

Returns

An element and its offset. If *tsdesc* is a descriptor for a regular time series, the time stamp column in the element is set to NULL; if *tsdesc* is a descriptor for an irregular time series, the time stamp column is set to whatever was in *values*[0]. This element must be freed by the caller using **ts_free_elem()**.

Related reference:

"The ts_elem_to_row() function" on page 9-22
"The ts_free_elem() procedure" on page 9-24
"The ts_ins_elem() function" on page 9-33
"The ts_elem() function" on page 9-20
"The ts_make_elem_with_buf() function"
"The ts_put_elem() function" on page 9-44
"The ts_put_elem_no_dups() function" on page 9-45
"The ts_put_last_elem() function" on page 9-46
"The ts_put_nth_elem() function" on page 9-46
"The ts_row_to_elem() function" on page 9-48
"The ts_upd_elem() function" on page 9-52

The ts_make_elem_with_buf() function

The **ts_make_elem_with_buf()** function creates a time series element using the buffer in an existing time series element. The initial data in the element is overwritten.

Syntax

```
ts_tselem
ts_make_elem_with_buf(ts_tsdesc  *tsdesc,
                       MI_DATUM   *values,
                       mi_boolean *nulls,
                       mi_integer *off,
                       ts_tselem  elem)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

values An array of data to be placed in the element. Data that is by value is placed in the array, and data that is by reference stores pointers.

nulls Stores which columns in the element should be NULL.

off For a regular time series, *off* contains the offset of the element on return. For an irregular time series, *off* is set to -1. This argument can be NULL.

elem The time series element to be overwritten. It must agree in type with the subtype of the time series. If this argument is NULL, a new element is created.

Returns

A time series element. If the *elem* argument is non-null, that is returned containing the new values. If the *elem* argument is NULL, a new time series element is returned.

Related reference:
"The ts_elem_to_row() function" on page 9-22
"The ts_free_elem() procedure" on page 9-24
"The ts_ins_elem() function" on page 9-33
"The ts_make_elem() function" on page 9-36
"The ts_put_last_elem() function" on page 9-46
"The ts_upd_elem() function" on page 9-52

The ts_make_stamp() function

The **ts_make_stamp()** function constructs a time stamp from the year, month, day, hour, minute, second, and microsecond values and puts them into the **mi_datetime** pointed to by the *dt* argument.

Syntax

```
mi_datetime *
ts_make_stamp (MI_CONNECTION *conn,
               mi_datetime   *dt,
               mi_integer    year,
               mi_integer    month,
               mi_integer    day,
               mi_integer    hour,
               mi_integer    minute,
               mi_integer    second,
               mi_integer    ms)
```

conn A valid DataBlade API connection.

dt The time stamp to fill in. The caller should supply the buffer.

year The year to put into the returned **mi_datetime**.

month The month to put into the returned **mi_datetime**.

day The day to put into the returned **mi_datetime**.

hour The hour to put into the returned **mi_datetime**.

minute The minute to put into the returned **mi_datetime**.

second The second to put into the returned **mi_datetime**.

ms The microsecond to put into the returned **mi_datetime**.

Returns

A pointer to the same **mi_datetime** structure that was passed in.
Related reference:
"The ts_get_stamp_fields() procedure" on page 9-29

The ts_nelems() function

The **ts_nelems()** function returns the number of elements in the time series.

Syntax

```
mi_integer
ts_nelems(ts_tsdesc *tsdesc)
```

tsdesc The time series descriptor returned by **ts_open()**.

Description

The equivalent SQL function is **GetNelems**.

Returns

The number of elements in the time series.

Related reference:

"ClipGetCount function" on page 7-30

"GetNelems function" on page 7-48

The ts_next() function

After a scan has been started with **ts_begin_scan()**, elements can be retrieved from the time series with **ts_next()**.

Syntax

```
mi_integer
ts_next(ts_tscan  *tscan,
        ts_tselem *tselem)
```

tscan The specified scan.

tselem A pointer to an element that **ts_next()** fills in.

Description

On return, the **ts_tselem** contains the next element in the time series, if there is one.

When **ts_tselem** is valid, it can be passed to other routines in the time series API, such as **ts_put_elem()**, **ts_get_col_by_name()**, and **ts_get_col_by_number()**.

Returns

TS_SCAN_ELEM
: The *tselem* parameter contains a valid element.

TS_SCAN_NULL
: The value in the element was NULL or hidden; if *tselem* is not NULL, then the element was hidden; otherwise, the element was NULL.

TS_SCAN_EOS
: The scan has completed; *tselem* is not valid.

The return value must not be freed by the caller; it is freed when the scan is ended. It is overwritten after two **ts_next()** calls.

Example

See the **ts_interp()** function, Appendix A, "The Interp function example," on page A-1, for an example of **ts_next()**.

Related reference:
"The ts_begin_scan() function" on page 9-6
"The TS_ELEM_HIDDEN macro" on page 9-21
"The TS_ELEM_NULL macro" on page 9-22
"The ts_first_elem() function" on page 9-23
"The ts_next_valid() function"
"The ts_previous_valid() function" on page 9-43
"The ts_put_elem() function" on page 9-44
"The ts_put_elem_no_dups() function" on page 9-45
"The ts_put_last_elem() function" on page 9-46
"The ts_put_nth_elem() function" on page 9-46
"The ts_upd_elem() function" on page 9-52

The ts_next_valid() function

The **ts_next_valid()** function returns the nearest entry after a given time stamp.

Syntax

```
ts_tselem
ts_next_valid(ts_tsdesc    *tsdesc,
              mi_datetime  *tstamp,
              mi_integer   *STATUS,
              mi_integer   *off)
```

tsdesc The time series descriptor returned by **ts_open()**.

tstamp Points to the time stamp that precedes the element returned.

STATUS
Points to an **mi_integer** value that is filled in on return. See the discussion of **ts_hide_elem()** ("The ts_hide_elem() function" on page 9-31) for a description of *STATUS*.

off For regular time series, *off* points to an **mi_integer** value that is filled in on return with the offset of the returned element. For irregular time series, *off* is set to -1. Can be NULL.

Description

For regular time series, this function returns the element at the calendar's earliest valid timepoint following the given time stamp. For irregular time series, it returns the earliest element following the given time stamp.

Tip: The **ts_next_valid()** function is less efficient than **ts_next()**, so it is better to iterate through a time series using **ts_begin_scan()** and **ts_next()** rather than using **ts_first_elem()** and **ts_next_valid()**.

The equivalent SQL function is **GetNextValid**.

Returns

The element following the given time stamp. If no valid element exists or the time series is regular and the next valid interval contains a null element, NULL is returned. The value pointed to by *off* is either -1 if the time series is irregular or the offset of the element if the time series is regular. The element returned must not be freed by the caller. It is overwritten after two fetch calls.

See "The ts_hide_elem() function" on page 9-31 for an explanation of STATUS.

Related reference:

"GetNextValid function" on page 7-49

"The TS_ELEM_HIDDEN macro" on page 9-21

"The TS_ELEM_NULL macro" on page 9-22

"The ts_first_elem() function" on page 9-23

"GetLastValid function" on page 7-46

"The ts_next() function" on page 9-39

"The ts_previous_valid() function" on page 9-43

The ts_nth_elem() function

The **ts_nth_elem()** function returns the element at the *n*th position of the given time series.

Syntax

```
ts_tselem
ts_nth_elem(ts_tsdesc  *tsdesc,
            mi_integer N,
            mi_integer *STATUS)
```

tsdesc The descriptor returned by **ts_open()**.

N The time series offset or position to read the element from. This value must not be less than 0.

STATUS
A pointer to an **mi_integer** value that is set on return to indicate whether the element is NULL. See "The ts_hide_elem() function" on page 9-31 for a description of *STATUS*.

Description

The equivalent SQL function is **GetNthElem**.

Returns

The element at the *n*th position of the given time series, and whether it was NULL. This value must not be freed by the caller. It is overwritten after two fetch calls.

Related reference:

"GetNthElem function" on page 7-50

"The ts_elem() function" on page 9-20

"The TS_ELEM_HIDDEN macro" on page 9-21

"The TS_ELEM_NULL macro" on page 9-22

"The ts_index() function" on page 9-32

"The ts_last_elem() function" on page 9-34

The ts_open() function

The **ts_open()** function opens a time series.

Syntax

```
ts_tsdesc *
ts_open(MI_CONNECTION *conn,
       ts_timeseries *ts,
       MI_TYPEID     *type_id,
       mi_integer    flags)
```

conn A database connection. This argument is unused in the server.

ts The time series to open.

type_id The ID for the type of the time series to be opened. The ID is generally determined by looking in the MI_FPARAM structure.

flags Valid values for the *flags* parameter are defined in `tseries.h`.

The *flags* argument values

Valid values for the *flags* argument are defined in the file `tseries.h`. (the integer value you use for the *flags* argument is the sum of the desired values). Valid options are:

TSOPEN_RDWRITE
The default mode for opening a time series. Indicates that the time series can be read and written to.

TSOPEN_READ_HIDDEN
Indicates that hidden elements should be treated as if they are not hidden.

TSOPEN_WRITE_HIDDEN
Allows hidden elements to be written to without first revealing the element.

TSOPEN_WRITE_AND_HIDE
Causes any elements written to a time series also to be marked as hidden.

TSOPEN_WRITE_AND_REVEAL
Reveals any hidden element that is written.

TSOPEN_NO_NULLS
Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated, it is returned as `NULL`. If TSOPEN_NO_NULLS is set, an element that has each column set to `NULL` is returned instead.

These flags can be used in any combination except the following four combinations:

- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_REVEAL
- TSOPEN_WRITE_AND_REVEAL and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_HIDE, and TSOPEN_WRITE_AND_REVEAL

The TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_REVEAL, and TSOPEN_WRITE_AND_HIDE flags cannot be used with TSOPEN_READ_HIDDEN.

Description

Almost all other functions depend on this function being called first.

Use **ts_close** to close the time series.

Returns

A descriptor for the open time series.

Example

See the **ts_interp()** function, Appendix A, "The Interp function example," on page A-1, for an example of **ts_open()**.

Related reference:

"The ts_begin_scan() function" on page 9-6

"The ts_close() function" on page 9-12

"The ts_create() function" on page 9-16

"The ts_create_with_metadata() function" on page 9-17

"The ts_get_typeid() function" on page 9-31

The ts_previous_valid() function

The **ts_previous_valid()** function returns the last element preceding the given time stamp.

Syntax

```
ts_tselem
ts_previous_valid(ts_tsdesc   *tsdesc,
                  mi_datetime *tstamp,
                  mi_integer  *STATUS,
                  mi_integer  *off)
```

tsdesc The time series descriptor returned by **ts_open()**.

tstamp Points to the time stamp that follows the element returned.

STATUS
: Points to an **mi_integer** value that is filled in on return. If no element exists before the time stamp, or if the time stamp falls before the time series origin, *STATUS* is set to a nonzero value. See "The ts_hide_elem() function" on page 9-31 for a description of *STATUS*.

off For regular time series, *off* points to an **mi_integer** value that is filled in on return with the offset of the returned element. For irregular time series, *off* is set to -1. This argument can be passed as `NULL`.

Description

The equivalent SQL function is **GetPreviousValid**.

Returns

The element, if any, preceding the given time stamp. The element returned must not be freed by the caller. It is overwritten after two calls to fetch an element using this *tsdesc* (time series descriptor).

For irregular time series, if no valid element precedes the given time stamp, NULL is returned. NULL is also returned if the given time stamp is less than or equal to the origin of the time series.

Related reference:

"GetPreviousValid function" on page 7-53
"The TS_ELEM_HIDDEN macro" on page 9-21
"The TS_ELEM_NULL macro" on page 9-22
"The ts_last_valid() function" on page 9-35
"The ts_next_valid() function" on page 9-40
"The ts_hide_elem() function" on page 9-31
"The ts_next() function" on page 9-39

The ts_put_elem() function

The **ts_put_elem()** function puts new elements into an existing time series.

Syntax

```
ts_timeseries *
ts_put_elem(ts_tsdesc   *tsdesc,
            ts_tselem   tselem,
            mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be modified, returned by **ts_open()**.

tselem The element to add.

tstamp The time stamp at which to put the element. The time stamp column of *tselem* is ignored.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

For regular time series, if there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

For irregular time series, if there is no data at the given timepoint, the new data is inserted. If there is data at the given timepoint, then the following algorithm is used to determine where to place the data:

1. Round the time stamp up to the next second.
2. Search backward for the first element less than the new time stamp.
3. Insert the new data at this time stamp plus 10 microseconds.

The element passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The equivalent SQL function is **PutElem**.

Returns

The original time series with the element added.

Related reference:
"PutElem function" on page 7-65
"The ts_del_elem() function" on page 9-20
"The ts_get_ts() function" on page 9-30
"The ts_ins_elem() function" on page 9-33
"The ts_make_elem() function" on page 9-36
"The ts_elem() function" on page 9-20
"The ts_next() function" on page 9-39
"The ts_put_elem_no_dups() function"
"The ts_put_last_elem() function" on page 9-46
"The ts_upd_elem() function" on page 9-52
"The ts_put_ts() function" on page 9-47

The ts_put_elem_no_dups() function

The **ts_put_elem_no_dups()** function puts a new element into an existing time series. The element is inserted even if there is already an element with the given time stamp in the time series.

Syntax

```
ts_timeseries *
ts_put_elem_no_dups(ts_tsdesc   *tsdesc,
                    ts_tselem   tselem,
                    mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be modified, returned by **ts_open()**.

tselem The element to add.

tstamp The time stamp at which to put the element. The time stamp column of *tselem* is ignored.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

If there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

The element passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The equivalent SQL function is **PutElemNoDups**.

Returns

The original time series with the element added.

Related reference:

"PutElemNoDups function" on page 7-67

"The ts_put_elem() function" on page 9-44

"The ts_elem() function" on page 9-20

"The ts_get_ts() function" on page 9-30

"The ts_ins_elem() function" on page 9-33

"The ts_make_elem() function" on page 9-36

"The ts_next() function" on page 9-39

"The ts_put_last_elem() function"

"The ts_upd_elem() function" on page 9-52

The ts_put_last_elem() function

The **ts_put_last_elem()** function puts new elements at the end of an existing regular time series.

Syntax

```
ts_timeseries *
ts_put_last_elem(ts_tsdesc *tsdesc,
                 ts_tselem tselem)
```

tsdesc The time series to be updated.

tselem The element to add; any time stamp in the element is ignored.

Returns

The original time series with the element added. If the time series is irregular, an error is raised.

Related reference:

"The ts_put_elem() function" on page 9-44

"The ts_put_elem_no_dups() function" on page 9-45

"The ts_elem() function" on page 9-20

"The ts_get_ts() function" on page 9-30

"The ts_make_elem() function" on page 9-36

"The ts_make_elem_with_buf() function" on page 9-37

"The ts_next() function" on page 9-39

The ts_put_nth_elem() function

The **ts_put_nth_elem()** function puts new elements into an existing regular time series at a specified offset.

Syntax

```
ts_timeseries *
ts_put_nth_elem(ts_tsdesc  *tsdesc,
                ts_tselem  tselem,
                mi_integer N)
```

tsdesc The time series to be updated.

tselem The element to add; any time stamp in the element is ignored.

N The offset, indicating where the element to add should be placed. Offsets start at 0.

Returns

The original time series with the element added. If the time series is irregular, an error is raised.

Related reference:

"The ts_index() function" on page 9-32
"The ts_elem() function" on page 9-20
"The ts_get_ts() function" on page 9-30
"The ts_make_elem() function" on page 9-36
"The ts_next() function" on page 9-39

The ts_put_ts() function

The **ts_put_ts()** function updates a destination time series with the elements from the source time series.

Syntax

```
ts_timeseries *
ts_put_ts(ts_tsdesc  *src_tsdesc,
          ts_tsdesc  *dst_tsdesc,
          mi_boolean nodups)
```

src_tsdesc
: The source time series descriptor.

dst_tsdesc
: The destination time series descriptor.

nodups Determines whether to overwrite an element in the destination time series if there is an element at the same time stamp in the source time series. This argument is ignored if the destination time series is regular.

Description

The two descriptors must meet the following conditions:

- The origin of the source time series must be after or equal to that of the destination time series.
- The two time series must have the same calendar.

If *nodups* is MI_TRUE, the element from the source time series overwrites the element in the destination time series. For irregular time series, if *nodups* is MI_FALSE and there is already a value at the existing timepoint, the update is made at the next microsecond after the last element in the given second. If the last microsecond in the second already contains a value, an error is raised.

The equivalent SQL function is **PutTimeSeries**.

Returns

The time series associated with the destination time series descriptor.

Related reference:

"PutTimeSeries function" on page 7-70

"The ts_put_elem() function" on page 9-44

The ts_reveal_elem() function

The **ts_reveal_elem()** function makes the element at a given time stamp visible to a scan. It reverses the effect of **ts_hide_elem()**.

Syntax

```
ts_timeseries
ts_reveal_elem(ts_tsdesc   *tsdesc,
               mi_datetime *tstamp)
```

ts_desc The time series descriptor returned by **ts_open()** for the source time series.

tstamp The time stamp to be made visible to the scan.

Description

The equivalent SQL function is **RevealElem**.

Returns

The modified time series. No error is raised if there is no element at the given time stamp.

Related reference:

"HideElem function" on page 7-55

"The ts_hide_elem() function" on page 9-31

"RevealElem function" on page 7-72

The ts_row_to_elem() function

The **ts_row_to_elem()** function converts an MI_ROW structure into a new **ts_tselem** structure. The new element does not overwrite elements returned by any other time series API function.

Syntax

```
ts_tselem
ts_row_to_elem(ts_tsdesc  *tsdesc,
               MI_ROW     *row,
               mi_integer *offset_ptr)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

row A pointer to an MI_ROW structure. The row must have the same type as the subtype of the time series.

offset_ptr
If the time series is regular, the offset of the element in the time series is returned in *offset_ptr*. In this case, column 0 (the time stamp column) must not be NULL. If the time series is irregular, -1 is returned in *offset_ptr*.

The *offset_ptr* argument can be NULL. In this case, calendar computations are avoided and column 0 can be NULL.

Returns

An element and its offset. If the time series is regular, column 0 (the time stamp column) of the element is NULL.

The element must be freed by the caller using the **ts_free_elem()** procedure.

Related reference:

"The ts_elem_to_row() function" on page 9-22

"The ts_free_elem() procedure" on page 9-24

"The ts_make_elem() function" on page 9-36

The ts_time() function

The **ts_time()** function converts a regular time series offset to a time stamp.

Syntax

```
mi_datetime *
ts_time(ts_tsdesc  *tsdesc,
        mi_integer N)
```

ts_desc The time series descriptor returned by **ts_open()** for the source time series.

N The offset to convert. Negative values are allowed.

Description

For example, for a daily time series that starts on Monday, January 1, with a five-day-a-week pattern starting on Monday, this function returns Monday, January 1, when the argument is set to 0; Tuesday, January 2, when the argument is set to 1; Monday, January 8, when the argument is 5; and so on.

The equivalent SQL function is **GetStamp**.

Returns

The time stamp corresponding to the offset. This value must be freed by the user with **mi_free()**.

Related reference:

"GetStamp function" on page 7-54

"The ts_cal_range() function" on page 9-9

"The ts_cal_range_index() function" on page 9-10

"The ts_index() function" on page 9-32

The ts_tstamp_difference() function

The **ts_tstamp_difference()** function subtracts one date from another and returns the number of complete intervals between the two dates.

Syntax

```
mi_integer
ts_tstamp_difference(mi_datetime  *date1,
                     mi_datetime *date2,
                     mi_integer  interval)
```

date1 The first date.

date2 The date to subtract from the first date.

interval
The interval, as described next.

Description

Before the difference is calculated, both time stamps are truncated to the given interval. For example, if the interval is an hour and the first date is 2011-01-03 01:02:03.12345, its truncated value is 2011-01-03 01:00:00.00000.

Valid values for the *interval* parameter can be found in `tseries.h`. They are:
- TS_SECOND
- TS_MINUTE
- TS_HOUR
- TS_DAY
- TS_WEEK
- TS_MONTH
- TS_YEAR

Returns

The number of intervals of the type you specify between the two dates.

Example

For example, if the interval is *day* and the dates are 2011-01-01 00:00:00.00000 and 2011-01-01 00:00:00.00001, the result is 0. If the dates are 2011-01-01 00:00:00.0000 and 2011-01-02 00:10:00.12345, the result is 1.

Related reference:

"The ts_tstamp_minus() function"

"The ts_tstamp_plus() function" on page 9-51

The ts_tstamp_minus() function

The **ts_tstamp_minus()** function returns a time stamp at a specified number of intervals before a starting date you specify.

Syntax

```
mi_datetime *
ts_tstamp_minus(mi_datetime  *startdate,
                mi_integer   cnt,
                mi_integer   interval,
                mi_datetime  *result)
```

startdate
The date to start from.

cnt The number of intervals to subtract from the start date.

interval
The interval, as described next.

result The resulting date.

Description

Valid values for the *interval* parameter can be found in `tseries.h`. They are:
- TS_SECOND
- TS_MINUTE
- TS_HOUR
- TS_DAY
- TS_WEEK
- TS_MONTH
- TS_YEAR

If the *result* parameter is `NULL`, then a result **mi_datetime** structure is allocated and returned; otherwise, the return value is the given *result* parameter.

Returns

The time stamp at the specified number of intervals before the start date.

Related reference:

"The ts_tstamp_difference() function" on page 9-49

"The ts_tstamp_plus() function"

The ts_tstamp_plus() function

The **ts_tstamp_plus()** function returns a time stamp at a specified number of intervals after a starting date you specify.

Syntax

```
mi_datetime *
ts_tstamp_plus(mi_datetime  *startdate,
               mi_integer   cnt,
               mi_integer   interval,
               mi_datetime  *result)
```

startdate
: The date to start from.

cnt
: The number of intervals to add to the start date.

interval
: The interval, as described next.

result
: The resulting date.

Description

Valid values for the *interval* parameter can be found in `tseries.h`. They are:
- TS_SECOND
- TS_MINUTE
- TS_HOUR
- TS_DAY
- TS_WEEK
- TS_MONTH
- TS_YEAR

If the *result* parameter is NULL, then a result **mi_datetime** structure is allocated and returned; otherwise, the return value is the given *result* parameter.

Returns

The time stamp at the specified number of intervals after the start date.

Related reference:

"The ts_tstamp_difference() function" on page 9-49

"The ts_tstamp_minus() function" on page 9-50

The ts_update_metadata() function

The **ts_update_metadata()** function adds the supplied user-defined metadata to the specified time series.

Syntax

```
ts_timeseries *
ts_update_metadata(ts_timeseries *ts,
                   mi_lvarchar   *metadata,
                   MI_TYPEID     *metadata_typeid)
```

ts The time series for which to update metadata.

metadata
: The metadata to add to the time series. Can be NULL.

metadata_typeid
: The type ID of the metadata.

Description

The equivalent SQL function is **UpdMetaData**.

Returns

A copy of the specified time series updated to contain the supplied metadata, or if the *metadata* argument is NULL, a copy of the specified time series with the metadata removed.

Related reference:

"GetMetaData function" on page 7-47

"UpdMetaData function" on page 7-114

"GetMetaTypeName function" on page 7-47

"TSCreate function" on page 7-92

"TSCreateIrr function" on page 7-94

"The ts_create_with_metadata() function" on page 9-17

"The ts_get_metadata() function" on page 9-28

The ts_upd_elem() function

The **ts_upd_elem()** function updates an element in an existing time series at a given timepoint.

Syntax

```
ts_timeseries *
ts_upd_elem(ts_tsdesc   *tsdesc,
            ts_tselem   tselem,
            mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be updated, returned by **ts_open()**.

tselem The element to update.

tstamp The timepoint at which to update the element.

Description

There must already be an element at the given time stamp. For irregular time series, hidden elements cannot be updated.

The equivalent SQL function is **UpdElem**.

Returns

An updated copy of the original time series.

Related reference:

"The ts_del_elem() function" on page 9-20
"The ts_ins_elem() function" on page 9-33
"The ts_put_elem() function" on page 9-44
"The ts_put_elem_no_dups() function" on page 9-45
"UpdElem function" on page 7-113
"The ts_elem() function" on page 9-20
"The ts_last_elem() function" on page 9-34
"The ts_make_elem() function" on page 9-36
"The ts_make_elem_with_buf() function" on page 9-37
"The ts_next() function" on page 9-39

Appendix A. The Interp function example

The **Interp** function is an example of a server function that uses the time series API. This function interpolates between values of a regular time series to fill in null elements.

This function does not handle individual null columns. It assumes that all columns are of type FLOAT.

Interp might be used as follows:

```
select Interp(stock_data) from daily_stocks where stock_name = 'IBM';
```

This example, along with many others, is supplied in the `$INFORMIXDIR/extend/TimeSeries.`*`version`* directory.

To use the **Interp** function, create a server function:

```
create function Interp(TimeSeries) returns TimeSeries
external name '/tmp/Interpolate.bld(ts_interp)'
language c not variant;
```

You can now use the **Interp** function in a DB-Access statement. For example, consider the difference in output between the following two queries (the output has been reformatted; the actual output you would see would not be in tabular format):

```
select stock_data from daily_stocks where stock_name = 'IBM';


2011-01-03 00:00:00    1   1   1   1
2011-01-04 00:00:00    2   2   2   2
NULL
2011-01-06 00:00:00    3   3   3   3


select Interp(stock_data) from daily_stocks where stock_name = 'IBM';

2011-01-03 00:00:00    1   1   1   1
2011-01-04 00:00:00    2   2   2   2
2011-01-05 00:00:00    2.5 2.5 2.5 2.5
2011-01-06 00:00:00    3   3   3   3
```

```
/*
 * SETUP:
 * create function Interp(TimeSeries) returns TimeSeries
 * external name 'Interpolate.so(ts_interp)'
 * language c not variant;
 *
 *
 * USAGE:
 * select Interp(stock_data) from daily_stocks where stock_id = 901;
 */

#include <stdio.h>
#include <mi.h>
#include <tseries.h>

# define TS_MAX_COLS   100
# define DATATYPE   "smallfloat"

/*
```

```
 * This example interpolates between values to fill in null elements.
 * It assumes that all columns are of type smallfloat and that there
are
 * less than 100 columns in each element.
 */

ts_timeseries *
ts_interp(tsPtr, fParamPtr)
    ts_timeseries   *tsPtr;
    MI_FPARAM      *fParamPtr;
{
    ts_tsdesc      *descPtr;
    ts_tselem      tselem;
    ts_tscan      *scan;
    MI_CONNECTION       *conn;
    ts_typeinfo      *typeinfo;
    int         scancode;
    mi_real           *values[TS_MAX_COLS];
    mi_real           lastValues[TS_MAX_COLS], newValues[TS_MAX_COLS];
    mi_boolean      nulls[TS_MAX_COLS];
    mi_integer      minElem, curElem, elem;
    mi_integer      i;
    mi_boolean      noneYet;
    mi_integer      ncols;
    char      strbuf[100];

    /* get a connection for libmi */
    conn = mi_open(NULL,NULL,NULL);

    /* open a descriptor for the timeseries */
    descPtr = ts_open(conn, tsPtr, mi_fp_rettype(fParamPtr, 0), 0);

    if ((ncols = (mi_integer) mi_fp_funcstate(fParamPtr)) == 0) {
   ncols = ts_col_cnt(descPtr);

   if (ncols > TS_MAX_COLS) {
       sprintf(strbuf, "Timeseries elements have too many columns,
100 is
the max, got %d instead.", ncols);
       mi_db_error_raise(NULL, MI_FATAL, strbuf, 0);
   }

   for (i = 1; i < ncols; i++) {
       typeinfo = ts_colinfo_number(descPtr, i);

       if (strlen(typeinfo->ti_typename) != strlen(DATATYPE) &&
      memcmp(typeinfo->ti_typename, DATATYPE, strlen(DATATYPE)) !=
0){
      sprintf(strbuf, "column was not a %s, got %s instead.", DATATYPE,

typeinfo->ti_typename);
      mi_db_error_raise(NULL, MI_FATAL, strbuf, 0);
       }
   }

   mi_fp_setfuncstate(fParamPtr, (void *) ncols);
    }

    noneYet = MI_TRUE;
    minElem = -1;
    curElem = 0;
    /* begin a scan of the whole timeseries */
    scan = ts_begin_scan(descPtr, 0, NULL, NULL);
    while ((scancode = ts_next(scan, &tselem)) != TS_SCAN_EOS)
{
   switch(scancode) {
       case TS_SCAN_ELEM:
```

```
        /* if this element is not null expand its values */
        noneYet = MI_FALSE;
        ts_get_all_cols(descPtr, tselem, (void **) values, nulls, curElem);
        if (minElem == -1) {
            /* save each element */
            for (i = 1; i < ncols; i++)
           lastValues[i] = *values[i];
        }
        else {
            /* calculate the average */
            for (i = 1; i < ncols; i++) {
           newValues[i] = (*values[i] + lastValues[i])/2.0;
           lastValues[i] = *values[i];
           values[i] = &newValues[i];
            }

            /* update the missing elements */

            tselem = ts_make_elem(descPtr, (void **) values, nulls, &elem);
            for (elem = minElem; elem < curElem; elem++)
           ts_put_nth_elem(descPtr, tselem, elem);

            minElem = -1;
        }

        break;
         case TS_SCAN_NULL:
        if (noneYet)
            break;
        /* remember the first null element */
        if (minElem == -1)
            minElem = curElem;
        break;
    }

    curElem++;
     }
     ts_end_scan(scan);
     ts_close(descPtr);
     return(tsPtr);
}
```

Appendix B. The TSIncLoad procedure example

The **TSIncLoad** procedure loads data into a database containing a time series of corporate bond prices.

The **TSIncLoad** procedure loads time-variant data from a file into a table containing time series. It assumes that the table has already been populated with the time-invariant data. If the table already has time series data, the new data overwrites the old data or is appended to the existing time series, depending on the time stamps.

To set up the **TSIncLoad** example, create the procedure, the row subtype, and the database table:

```
create procedure TSIncLoad( table_name lvarchar,
                              file_name lvarchar,
                              calendar_name lvarchar,
                              origin datetime year to day,
                              threshold integer,
                              regular boolean,
                              container_name lvarchar,
                              nelems integer)
external name
'$INFORMIXDIR/extend/timeseries/Loader.bld(TSIncLoad)'
language C;

create row type day_info (
        ValueDate          datetime year to fraction(5),
        carryover          char(1),
        spread             integer,
        pricing_bmk_id     integer,
        price              float,
        yield              float,
        priority           char(1) );


execute procedure TSContainerCreate('ctnr_daily', 'rootdbs',
'day_info', 0, 0);

create table corporates (
        Secid              integer UNIQUE,
        series             TimeSeries(day_info));

insert into corporates values (
        25000006,
                'container(ctnr_daily), origin(2011-01-03
00:00:00.00000), calendar(daycal),
threshold(0)');
execute procedure TSIncLoad('corporates',
                            '/tmp/daily.dat',
                            'daycal',
                            '2011-01-03',
                            0,
                            't',
                            'ctnr_daily',
                            1);
```

Any name can be used for the **corporates** table. The **corporates** table can have any number of columns in addition to the **Secid** and **series** columns.

Each line of the data file has the following format:

```
Secid year-mon-day carryover spread pricing_bmk_id price yield priority
```

For example:

```
25000006 2010-1-7 m 2 12 2.2000000000 22.2 6
```

You can invoke the **TSIncLoad** procedure with an SQL statement like:

```
execute procedure TSIncLoad( 'corporates',
                                 'data_file_name',
                                 'cal_name',
                                 '2010-1-1',
                                 20,
                                 't',
                                 'container-name');
```

```
#include <ctype.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "datetime.h"
#include "mi.h"
#include "tseries.h"

#define DAY_INFO_TYPE_NAME "day_info"
#define DAILY_COL_COUNT 7

typedef struct
    {
    mi_integer   fd;
    mi_unsigned_integer flags;
#define LDBUF_LAST_CHAR_EOL 0x1

    mi_integer  buf_index;
    mi_integer  buf_len;
    mi_integer  line_no;
    mi_lvarchar *file_name;
    mi_string  data[2048];
    }
FILE_BUF;

#define STREAM_EOF (-1)

typedef struct sec_entry_s
    {
    mi_integer  sec_id;
    ts_tsdesc  *tsdesc;
    int  in_row; /* Indicates whether the time series is stored in
row. */
    struct  sec_entry_s *next;
    }
sec_entry_t;

typedef struct
    {
    mi_lvarchar *table_name;
    MI_TYPEID ts_typeid; /* The type id of timeseries(day_info) */
    mi_string *calendar_name;
    mi_datetime *origin;
    mi_integer threshold;
    mi_boolean regular;
    mi_string *container_name;
    mi_integer nelems; /* For created time series. */

    mi_integer hash_size;
```

```
    MI_CONNECTION *conn;
    sec_entry_t **hash;
    /* Value buffers -- only allocated once. */
    MI_DATUM col_data[ DAILY_COL_COUNT];
    mi_boolean col_is_null[ DAILY_COL_COUNT];
    char *carryover;
    char *priority;
    mi_double_precision price, yield;

    mi_integer instances_created;
    /* A count of the number of tsinstancetable entries added.  Used
to
     * decide when to update statistics on this table.
     */
    MI_SAVE_SET *save_set;
    }
loader_context_t;


/*
 ***************************************************************************
 * name:     init_context
 *
 * purpose:  Initialize the loader context structure.
 *
 * notes:
 ***************************************************************************
 */
static void
init_context( mi_lvarchar *table_name,
         mi_lvarchar *calendar_name,
         mi_datetime *origin,
         mi_integer threshold,
         mi_boolean regular,
         mi_lvarchar *container_name,
         mi_integer nelems,
         loader_context_t *context_ptr)
{
    mi_string buf[256];
    mi_integer table_name_len = mi_get_varlen( table_name);
    MI_ROW *row = NULL;
    MI_DATUM retbuf = 0;
    mi_integer retlen = 0;
    mi_lvarchar *typename = NULL;
    MI_TYPEID *typeid = NULL;
    mi_integer err = 0;


    if( table_name_len > IDENTSIZE)
   mi_db_error_raise( NULL, MI_EXCEPTION, "The table name is too long");

    memset( context_ptr, 0, sizeof( *context_ptr));
    context_ptr->conn = mi_open( NULL, NULL, NULL);

    typename = mi_string_to_lvarchar
              ( "timeseries(" DAY_INFO_TYPE_NAME ")");
    typeid = mi_typename_to_id( context_ptr->conn, typename);
    mi_var_free( typename);
    if( NULL == typeid)
   mi_db_error_raise( NULL, MI_EXCEPTION,
           "Type timeseries(" DAY_INFO_TYPE_NAME ") not defined.");
    context_ptr->ts_typeid = *typeid;

    context_ptr->table_name = table_name;

    context_ptr->calendar_name = mi_lvarchar_to_string( calendar_name);
    context_ptr->origin = origin;
```

```
    context_ptr->threshold = threshold;
    context_ptr->regular = regular;
    context_ptr->container_name = mi_lvarchar_to_string( container_name);
    context_ptr->nelems = nelems;

    /* Use the size (count) of the table as the hash table size. */
    sprintf( buf, "select count(*) from %.*s;",
        table_name_len,
        mi_get_vardata( table_name));
    if( MI_OK != mi_exec( context_ptr->conn, buf, MI_QUERY_BINARY))
   mi_db_error_raise( NULL, MI_EXCEPTION, "mi_exec failed");
    if( MI_ROWS != mi_get_result( context_ptr->conn))
   {
   sprintf( buf, "Could not get size of %.*s table.",
        table_name_len,
        mi_get_vardata( table_name));
   mi_db_error_raise( NULL, MI_EXCEPTION, buf);
   }
    if( NULL == (row = mi_next_row( context_ptr->conn, &err)))
   mi_db_error_raise( NULL, MI_EXCEPTION, "mi_next_row failed");
    if( MI_NORMAL_VALUE != mi_value( row, 0, &retbuf, &retlen)
   || 0 != dectoint( (mi_decimal *) retbuf, &context_ptr->hash_size))
   context_ptr->hash_size = 256;
    (void) mi_query_finish( context_ptr->conn);
    context_ptr->hash
   = mi_zalloc( context_ptr->hash_size*sizeof( *context_ptr->hash));

    context_ptr->col_data[1] = (MI_DATUM) mi_new_var(1); /* carryover
*/
    context_ptr->col_data[6] = (MI_DATUM) mi_new_var(1); /* priority
*/

    if( NULL == context_ptr->hash
   || NULL ==  context_ptr->col_data[1]
   || NULL ==  context_ptr->col_data[6])
   mi_db_error_raise( NULL, MI_EXCEPTION, "Not enough memory.");

    context_ptr->carryover
   = mi_get_vardata( (mi_lvarchar *) context_ptr->col_data[1]);
    context_ptr->col_data[4] = (MI_DATUM) &context_ptr->price;
    context_ptr->col_data[5] = (MI_DATUM) &context_ptr->yield;
    context_ptr->priority
   = mi_get_vardata( (mi_lvarchar *) context_ptr->col_data[6]);

    context_ptr->save_set = mi_save_set_create( context_ptr->conn);
} /* End of init_context. */


/*
 ***************************************************************************
 * name:     close_context
 *
 * purpose:  Close the context structure.  Free up all allocated memory.
 *
 ***************************************************************************
 */
static void
close_context( loader_context_t *context_ptr)
{
    mi_free( context_ptr->hash);
    context_ptr->hash = NULL;
    context_ptr->hash_size = 0;

    mi_var_free( (mi_lvarchar *) context_ptr->col_data[1]);
    mi_var_free( (mi_lvarchar *) context_ptr->col_data[6]);
    context_ptr->col_data[1] = context_ptr->col_data[6] = 0;
    context_ptr->carryover = context_ptr->priority = NULL;
```

```
    (void) mi_save_set_destroy( context_ptr->save_set);
    context_ptr->save_set = NULL;

    (void) mi_close( context_ptr->conn);

    mi_free( context_ptr->calendar_name);
    context_ptr->calendar_name = NULL;
    mi_free( context_ptr->container_name);
    context_ptr->container_name = NULL;

    context_ptr->conn = NULL;
} /* End of close_context. */


/*
 ***************************************************************************
 * name:     update_series
 *
 * purpose:  Update all the time series back into the table.
 *
 * returns:
 *
 * notes:
 ***************************************************************************
 */
static void
update_series( loader_context_t *context_ptr)
{
    mi_integer i = 0;
    register struct sec_entry_s *entry_ptr = NULL;
    struct sec_entry_s *next_entry_ptr = NULL;
    MI_STATEMENT *statement = NULL;
    char buf[256];
    mi_integer rc = 0;
    MI_DATUM values[2] = {0, 0};
    mi_integer lengths[2] = {-1, sizeof( mi_integer)};
    static const mi_integer nulls[2] = {0, 0};
    static const mi_string const *types[2]
   = {"timeseries(day_info)", "integer"};
    mi_unsigned_integer yield_count = 0;


    sprintf( buf, "update %.*s set series = ? where Secid = ?;",
        mi_get_varlen( context_ptr->table_name),
        mi_get_vardata( context_ptr->table_name));
    statement = mi_prepare( context_ptr->conn, buf, NULL);
    if( NULL == statement)
   mi_db_error_raise( NULL, MI_EXCEPTION, "mi_prepare failed");

    /* Look at all the entries in the hash table. */
    for( i = context_ptr->hash_size - 1; 0 <= i; i--)
   {
   for( entry_ptr = context_ptr->hash[i];
        NULL != entry_ptr;
        entry_ptr = next_entry_ptr)
       {
       if( NULL != entry_ptr->tsdesc)
      {
      yield_count++;
      if( 0 == (yield_count & 0x3f))
          {
          if( mi_interrupt_check())
         mi_db_error_raise( NULL, MI_EXCEPTION, "Load aborted.");
          mi_yield();
          }
```

```
      values[0] = ts_get_ts( entry_ptr->tsdesc);
      values[1] = (MI_DATUM) entry_ptr->sec_id;
      lengths[0] = mi_get_varlen( ts_get_ts( entry_ptr->tsdesc));

      if( mi_exec_prepared_statement( statement,
                  MI_BINARY,
                  1,
                  2,
                  values,
                  lengths,
                  (int *) nulls,
                  (char **) types,
                  0,
                  NULL)
          != MI_OK)
          mi_db_error_raise( NULL, MI_EXCEPTION,
                   "mi_exec_prepared_statement(update) failed");
      ts_close( entry_ptr->tsdesc);
      }
       next_entry_ptr = entry_ptr->next;
       mi_free( entry_ptr);
       }
   context_ptr->hash[i] = NULL;
   }
} /* End of update_series. */


/*
 ***************************************************************************
 * name:     open_buf
 *
 * purpose:  Open a file for reading and attach it to a buffer.
 *
 ***************************************************************************
 */
static void
open_buf( mi_lvarchar *file_name,
     FILE_BUF *buf_ptr)
{
    mi_string *file_name_str = mi_lvarchar_to_string( file_name);

    memset( buf_ptr, 0, sizeof( *buf_ptr));
    buf_ptr->fd = mi_file_open( file_name_str, O_RDONLY, 0);
    mi_free( file_name_str);
    buf_ptr->file_name = file_name;

    if( MI_ERROR == buf_ptr->fd)
   {
   char buf[356];
   mi_integer name_len = (256 < mi_get_varlen( file_name))
       ? 256 : mi_get_varlen( file_name);

   sprintf( buf, "mi_file_open(%.*s) failed",
       name_len, mi_get_vardata( file_name));

   mi_db_error_raise( NULL, MI_EXCEPTION, buf);
   }
    buf_ptr->buf_index = 0;
    buf_ptr->buf_len = 0;
    buf_ptr->line_no = 1;
} /* End of open_buf. */


/*
 ***************************************************************************
 * name:     get_char
 *
```

```
 * purpose:  Get the next character from a buffered file.
 *
 * returns:  The character or STREAM_EOF
 *
 ***************************************************************************
 */
static mi_integer
get_char( FILE_BUF *buf_ptr)
{
    register mi_integer c = STREAM_EOF;

    if( buf_ptr->buf_index >= buf_ptr->buf_len)
   {
   buf_ptr->buf_index = 0;
   buf_ptr->buf_len = mi_file_read( buf_ptr->fd,
                buf_ptr->data,
                sizeof( buf_ptr->data));
   if( MI_ERROR == buf_ptr->buf_len)
       {
       char buf[356];
       mi_integer name_len = (256 < mi_get_varlen( buf_ptr->file_name))
      ? 256 : mi_get_varlen( buf_ptr->file_name);

       sprintf( buf, "mi_file_read(%.*s) failed",
           name_len, mi_get_vardata(buf_ptr->file_name));

       mi_db_error_raise( NULL, MI_EXCEPTION, buf);
       }
   if( 0 == buf_ptr->buf_len)
       return( STREAM_EOF);
   }

    /* Increment buf_ptr->line_no until we have started on the next
line,
     * not when the newline character is seen.
     */
    if( buf_ptr->flags & LDBUF_LAST_CHAR_EOL)
   {
   buf_ptr->line_no++;
   buf_ptr->flags &= ~LDBUF_LAST_CHAR_EOL;
   }

    c = buf_ptr->data[ buf_ptr->buf_index++];
    if( '\n' == c)
   buf_ptr->flags |= LDBUF_LAST_CHAR_EOL;
    return( c);
} /* End of get_char. */


/*
 ***************************************************************************
 * name:     close_buf
 *
 * purpose:  Close a file attached to a buffer.
 *
 * notes:
 ***************************************************************************
 */
static void
close_buf( FILE_BUF *buf_ptr)
{
    mi_file_close( buf_ptr->fd);
    buf_ptr->fd = MI_ERROR;
    buf_ptr->buf_index = 0;
    buf_ptr->buf_len = 0;
    buf_ptr->file_name = NULL;
} /* End of close_buf. */
```

```
/*
 ***************************************************************************
 * name:     get_token
 *
 * purpose:  Get the next token from an input stream.
 *
 * returns:  The token in a buffer and the next character after the
buffer.
 *
 * notes:    Assumes that the tokens are separated by white space.
 ***************************************************************************
 */
static mi_integer
get_token( FILE_BUF *buf_ptr,
      mi_string *token,
      size_t token_buf_len)
{
    register mi_integer c = get_char( buf_ptr);
    register mi_integer i = 0;

    while( STREAM_EOF != c && isspace( c))
   c = get_char( buf_ptr);

    for( ;STREAM_EOF != c && ! isspace( c); c = get_char(
buf_ptr))
   {
   if( i >= token_buf_len - 1)
       {
       char err_buf[128];

       sprintf( err_buf, "Word is too long on line %d.", buf_ptr->line_no);
       mi_db_error_raise( NULL, MI_EXCEPTION, err_buf);
       }
   token[i++] = c;
   }
    token[i] = 0;

    return( c);
} /* End of get_token. */


/*
 ***************************************************************************
 * name:     increment_instances_created
 *
 * purpose:  Increment the instances_created field and update statistics
 *           when it crosses a threshold.  If the statistics for the
 *           time series instance table were never updated then the
server
 *           would not use the index on the instance table, and time
series
 *           opens would be very slow.
 *
 * returns:  nothing
 *
 * notes:
 ***************************************************************************
 */
static void
increment_instances_created( loader_context_t *context_ptr)
{
    context_ptr->instances_created++;
    if( 50 != context_ptr->instances_created)
   return;
```

```
    (void) mi_exec( context_ptr->conn,
          "update statistics high for table tsinstancetable( id);",
          MI_QUERY_BINARY);
} /* End of increment_instances_created. */


/*
 ***************************************************************************
 * name:     get_sec_entry
 *
 * purpose:  Get the security entry for a security ID
 *
 * returns:  A pointer to security entry
 *
 * notes:    If the entry is not found in the hash table then the
security
 *           is looked up in the table and a new entry made in the
hash
 *           table. A warning message will be emitted if the security
ID
 *           cannot be found.  In this case the security entry will
have
 *           a NULL tsdesc.
 ***************************************************************************
 */
static sec_entry_t *
get_sec_entry( loader_context_t *context_ptr,
          mi_integer sec_id,
          mi_integer line_no)
{
    mi_unsigned_integer i
   = ((mi_unsigned_integer) sec_id) % context_ptr->hash_size;
    sec_entry_t *entry_ptr = context_ptr->hash[i];
    mi_string buf[256];
    mi_integer rc = 0;

    /* Look the security ID up in the hash table. */
    for( ; NULL != entry_ptr; entry_ptr = entry_ptr->next)
   {
   if( sec_id == entry_ptr->sec_id)
       return( entry_ptr);
   }
    /* This is the first time this security ID has been seen. */
    entry_ptr = mi_zalloc( sizeof( *entry_ptr));
    entry_ptr->sec_id = sec_id;
    entry_ptr->next = context_ptr->hash[i];
    context_ptr->hash[i] = entry_ptr;

    /* Look up the security ID in the database table. */
    sprintf( buf,
        "select series from %.*s where Secid = %d;",
        mi_get_varlen( context_ptr->table_name),
        mi_get_vardata( context_ptr->table_name),
        sec_id);
    if( MI_OK != mi_exec( context_ptr->conn, buf, MI_QUERY_BINARY))
   mi_db_error_raise( NULL, MI_EXCEPTION, "mi_exec failed.");

    rc = mi_get_result( context_ptr->conn);
    if( MI_NO_MORE_RESULTS == rc)
   {
   sprintf( buf, "Security %d (line %d) not in %.*s.",
       sec_id, line_no,
       mi_get_varlen( context_ptr->table_name),
       mi_get_vardata( context_ptr->table_name));
   mi_db_error_raise( NULL, MI_MESSAGE, buf);
   /* Mi_db_error_raise returns after raising messages of type MI_MESSAGE.
    */
```

```
    }
     else if( MI_ROWS != rc)
    mi_db_error_raise( NULL, MI_EXCEPTION, "mi_get_result failed.");
     else
    {
    mi_integer err = 0;
    MI_ROW *row = mi_next_row( context_ptr->conn, &err);
    MI_DATUM ts_datum = 0;
    mi_integer retlen = 0;

    /* Save the row so that the time series column will not be erased
when
     * the query is finished.
     */
    if( NULL != row
        && MI_NORMAL_VALUE == mi_value( row, 0, &ts_datum, &retlen))
        {
        if( NULL == (row = mi_save_set_insert( context_ptr->save_set,
                      row)))
       mi_db_error_raise( NULL, MI_EXCEPTION,
                "mi_save_set_insert failed");
        }

    if( NULL != row)
        rc = mi_value( row, 0, &ts_datum, &retlen);
    else
        rc = MI_ERROR;
    if( MI_NORMAL_VALUE != rc && MI_NULL_VALUE != rc)
        {
        if( 0 != err)
       {
       sprintf( buf, "Look up of security ID %d in %.*s failed.",
           sec_id,
           mi_get_varlen( context_ptr->table_name),
           mi_get_vardata( context_ptr->table_name));
       mi_db_error_raise( NULL, MI_EXCEPTION, buf);
       }
        else
       {
       sprintf( buf, "Security %d (line %d) not in %.*s.",
           sec_id, line_no,
           mi_get_varlen( context_ptr->table_name),
           mi_get_vardata( context_ptr->table_name));
       mi_db_error_raise( NULL, MI_MESSAGE, buf);
       return( entry_ptr);
       }
        }
    if( MI_NULL_VALUE != rc)
        entry_ptr->in_row = (TS_IS_INCONTAINER( (ts_timeseries *) ts_datum)
              != 0);
    else
        {
        /* No time series has been created for this security yet.
         * Start one.
         */
        ts_datum = ts_create( context_ptr->conn,
               context_ptr->calendar_name,
               context_ptr->origin,
               context_ptr->threshold,
               context_ptr->regular ? 0 : TS_CREATE_IRR,
               &context_ptr->ts_typeid,
               context_ptr->nelems,
               context_ptr->container_name);
        entry_ptr->in_row = (TS_IS_INCONTAINER( (ts_timeseries *) ts_datum)
              == 0);
        if( entry_ptr->in_row)
       increment_instances_created( context_ptr);
```

```
        }
    entry_ptr->tsdesc = ts_open( context_ptr->conn,
                  ts_datum,
                  &context_ptr->ts_typeid,
                  0);
    }
     return( entry_ptr);
} /* End of get_sec_entry. */


/*
 ***************************************************************************
 * name:     is_null
 *
 * purpose:  Determine whether a token represents a null value.
 *
 * returns:  1 if so, 0 if not
 *
 ***************************************************************************
 */
static int
is_null( register mi_string *token)
{
    return( ('N' == token[0] || 'n' == token[0])
       && ('U' == token[1] || 'u' == token[1])
       && ('L' == token[2] || 'l' == token[2])
       && ('L' == token[3] || 'l' == token[3])
       && 0 == token[4]);
} /* End of is_null. */


/*
 ***************************************************************************
 * name:     read_day_data
 *
 * purpose:  Read in the daily data for one security.
 *
 * returns:  Fills in the timestamp structure, the col_data and col_is_null
 *           arrays.
 *
 * notes:    Assumes that the col_is_null array is initialized to
all TRUE.
 ***************************************************************************
 */
static void
read_day_data( loader_context_t *context_ptr,
          FILE_BUF *buf_ptr,
          mi_string *token,
          size_t token_buf_len,
          mi_datetime *tstamp_ptr)
{
    register mi_integer i = 0;
    register mi_integer c;

    /* ValueDate  DATETIME year to day*/
    c = get_token( buf_ptr, token, token_buf_len);
    if( STREAM_EOF== c && 0 == strlen( token)
   || '\n' == c)
   return;
    tstamp_ptr->dt_qual = TU_DTENCODE( TU_YEAR, TU_DAY);
    if( is_null( token))
   tstamp_ptr->dt_dec.dec_pos = DECPOSNULL;
    else
   {
   if( 0 == dtcvasc( token, tstamp_ptr))
       {
       context_ptr->col_is_null[0] = MI_FALSE;
```

```
        context_ptr->col_data[0] = (MI_DATUM) tstamp_ptr;
        }
    else
        {
        mi_string err_buf[128];

        sprintf( err_buf, "Illegal date on line %d", buf_ptr->line_no);
        mi_db_error_raise( NULL, MI_MESSAGE, err_buf);
        }
    }

     /* carryover char(1) */
     c = get_token( buf_ptr, token, token_buf_len);
     if( STREAM_EOF== c && 0 == strlen( token) || '\n' == c)
    return;
     if( ! is_null( token))
    {
    *(context_ptr->carryover) = token[0];
    context_ptr->col_is_null[1] = MI_FALSE;
    }

     /* spread integer,
      * pricing_bmk_id integer
      */
     for( i = 2; i < 4; i++)
    {
    c = get_token( buf_ptr, token, token_buf_len);
    if( STREAM_EOF== c && 0 == strlen( token)
        || '\n' == c)
        return;
    if( ! is_null( token))
        {
        context_ptr->col_data[i] = (MI_DATUM) atoi( token);
        context_ptr->col_is_null[i] = MI_FALSE;
        }
    }

     /* price float,
      * yield float
      */
     for( i = 4; i < 6; i++)
    {
    c = get_token( buf_ptr, token, token_buf_len);
    if( STREAM_EOF== c && 0 == strlen( token)
        || '\n' == c)
        return;
    if( ! is_null( token))
        {
        *((double *) context_ptr->col_data[i]) = atof( token);
        context_ptr->col_is_null[i] = MI_FALSE;
        }
    }

     /* priority char(1) */
     c = get_token( buf_ptr, token, token_buf_len);
     if( (STREAM_EOF == c || '\n' == c) && 0 == strlen( token))
    return;
     if( ! is_null( token))
    {
    *(context_ptr->priority) = token[0];
    context_ptr->col_is_null[6] = MI_FALSE;
    }
} /* End of read_day_data. */


/*
 ****************************************************************************
```

```
 * name:     read_line
 *
 * purpose:  Read a line from the file, fetch the time series descriptor
 *           corresponding to the Secid, create a time series element
for
 *           the line, and convert the date into an mi_datetime structure.
 *
 * returns:  1 if there was more data in the file,
 *           0 if the end of the file was found.
 *
 * notes:    Creates a new time series if the series column for the
Secid is
 *           NULL.
 ****************************************************************************
 */
int
read_line( loader_context_t *context_ptr,
      FILE_BUF *buf_ptr,
      ts_tsdesc **tsdesc_ptr,
      ts_tselem *day_elem_ptr,
      int *null_line,
      mi_datetime *tstamp_ptr,
      sec_entry_t **sec_entry_ptr_ptr)
{
    mi_integer sec_id = -1;
    sec_entry_t *sec_entry_ptr = NULL;
    mi_string token[256];
    mi_integer c = 0; /* Next character from file. */
    mi_integer i = 0;


    *sec_entry_ptr_ptr = NULL;
    *null_line = 1;
    for( i = 0; i < DAILY_COL_COUNT; i++)
   context_ptr->col_is_null[ i] = MI_TRUE;

    c = get_token( buf_ptr, token, sizeof( token));
    if( STREAM_EOF== c && 0 == strlen( token))
   return( 0);

    sec_id = atoi( token);

    *sec_entry_ptr_ptr = sec_entry_ptr
   = get_sec_entry( context_ptr, sec_id, buf_ptr->line_no);

    read_day_data( context_ptr,
         buf_ptr,
         token,
         sizeof( token),
         tstamp_ptr);

    *tsdesc_ptr = sec_entry_ptr->tsdesc;
    if( NULL == sec_entry_ptr->tsdesc)
   /* An invalid security ID. */
   return( 1);

    if( context_ptr->col_is_null[0]
   && TS_IS_IRREGULAR( ts_get_ts( sec_entry_ptr->tsdesc)))
   {
   mi_string err_buf[128];

   sprintf( err_buf, "Missing date on line %d.", buf_ptr->line_no);
   mi_db_error_raise( NULL, MI_MESSAGE, err_buf);
   return(1);
   }
    *null_line = 0;
    *day_elem_ptr = ts_make_elem_with_buf( sec_entry_ptr->tsdesc,
```

```
                  context_ptr->col_data,
                  context_ptr->col_is_null,
                  NULL,
                  *day_elem_ptr);
    return(1);
} /* End of read_line. */


/*
 ***************************************************************************
 * name:     TSIncLoad
 *
 * purpose:  UDR for incremental loading of timeseries from a file.
 *
 ***************************************************************************
 */
void
TSIncLoad( mi_lvarchar *table_name, /* the table that holds the time
series. */
      mi_lvarchar *file_name,
      /* The name of the file containing the data.  It must be accessible
       * on the server machine.
       */
      /*
       * The following parameters are only used to create new time
       * series.
       */
      mi_lvarchar *calendar_name,
      mi_datetime *origin,
      mi_integer threshold,
      mi_boolean regular,
      mi_lvarchar *container_name,
      mi_integer nelems,
      MI_FPARAM *fParamPtr)
{
    FILE_BUF buf = {0};
    ts_tselem day_elem = NULL;
    ts_tsdesc *tsdesc = NULL;
    ts_timeseries *ts = NULL;
    mi_datetime tstamp = {0};
    loader_context_t context = {0};
    mi_unsigned_integer yield_count = 0;
    sec_entry_t *sec_entry_ptr = NULL;
    int null_line = 0;


    init_context( table_name,
        calendar_name,
        origin,
        threshold,
        regular,
        container_name,
        nelems,
        &context);

    open_buf( file_name, &buf);

    while( read_line( &context,
            &buf,
            &tsdesc,
            &day_elem,
            &null_line,
            &tstamp,
            &sec_entry_ptr))
   {
   yield_count++;
   /* Periodically (once every 64 input lines) check for interrupts
```

```
and
    * yield the processor to other threads.
    */
   if( 0 == (yield_count & 0x3f))
       {
       if( mi_interrupt_check())
      mi_db_error_raise( NULL, MI_EXCEPTION, "Load aborted.");
       mi_yield();
       }

   if( null_line)
       continue;

   ts = ts_put_elem_no_dups( tsdesc, day_elem, &tstamp);
   if( sec_entry_ptr->in_row && TS_IS_INCONTAINER( ts))
       {
       sec_entry_ptr->in_row = 0;
       increment_instances_created( &context);
       }
   }

    if( NULL != day_elem)
   ts_free_elem( tsdesc, day_elem);

    close_buf( &buf);
    update_series( &context);
    close_context( &context);
} /* End of TSIncLoad. */
```

Appendix C. 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.

Tip: The information center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features by using the keyboard instead of the mouse.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

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

You can view the publications in Adobe Portable Document Format (PDF) by using the Adobe Acrobat Reader.

IBM and accessibility

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

Dotted decimal syntax diagrams

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

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

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

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

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

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

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

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 (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

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

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

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

Trademarks

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

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

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

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

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

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

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

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

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

Routines *(continued)*
SQL, time series *(continued)*

S

T

U

V

W

Printed in USA

SC27-3567-02

Spine information:

Informix Product Family Informix **Version 11.70** **IBM Informix TimeSeries Data User's Guide**