Informix TimeSeries DataBlade Module

User’s Guide

Version 3.1
April 1998
Part No. 000-5059

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Copyright © 1981-1998 by Informix Software, Inc. or its subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by "®," and in numerous other countries worldwide:

INFORMIX®; Illustra™; DataBlade®

All other marks or symbols are registered trademarks or trademarks of their respective owners.

ACKNOWLEDGMENTS

Documentation Team: Inge Halilovic, Joyce Simmonds

Contributors: Kevin Brown, Jack Klebanoff, Jun Luo

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

Chapter 1 Getting Started

Chapter 2 Overview of Time Series

Chapter 8 Time Series SQL Routines

Chapter 9 Time Series API Routines

Introduction

In This Chapter

This chapter introduces the *Informix TimeSeries DataBlade Module User's Guide*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout.

About This Manual

This manual contains information to assist you in using the Informix TimeSeries DataBlade module with Informix Dynamic Server with Universal Data Option. A DataBlade module adds custom data types and supporting routines to the server.

This section discusses the organization of the manual, the intended audience, and the associated software products that you must have to develop and use the Informix TimeSeries DataBlade module.

Organization of This Manual

The *Informix TimeSeries DataBlade Module User's Guide* contains chapters for using the Informix TimeSeries DataBlade module, its supplied data types and tables, its functions, and sample programs.

This manual includes the following chapters:

- This Introduction provides an overview of the manual, describes the documentation conventions used, and explains the generic style of this documentation.
- Chapter 1, "Getting Started," provides installation information and SQL compatibility issues.

- Chapter 2, "Overview of Time Series," describes time series architecture, organization, and concepts.
- Chapter 3, "Data Types and System Tables," describes the data types and tables provided with the Informix TimeSeries DataBlade module.
- Chapter 4, "Creating and Loading a Time Series," outlines the steps necessary to define and create a time series.
- Chapter 5, "Using the Virtual Table Interface," describes how you can create a virtual relational table that contains time series data without using a TimeSeries column.
- Chapter 6, "Calendar Pattern Routines," describes the SQL/C functions specific to calendar patterns.
- Chapter 7, "Calendar Routines," describes the SQL/C functions specific to calendars.
- Chapter 8, "Time Series SQL Routines," describes the SQL/C functions specific to time series.
- Chapter 9, "Time Series API Routines," describes the API functions.
- Appendix A and Appendix B provide sample programs.
- A glossary and an index appear at the end of this manual.

Types of Users

This manual is written for the following audience:

- Database administrators who install the Informix TimeSeries DataBlade module
- Developers who write applications to access time series information stored in Informix Dynamic Server with Universal Data Option databases

Software Dependencies

To use this manual, you must be using Informix Dynamic Server with Universal Data Option as your database server. See the on-line release notes for version numbers.

Documentation Conventions

This section describes the conventions that this manual uses:

- Typographical conventions
- Icon conventions

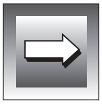
Typographical Conventions

This manual uses the following set of conventions to introduce new terms, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.

Icon Conventions

Throughout the documentation, you will find text that is identified by comment icons. Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
	The *warning* icon identifies vital instructions, cautions, or critical information.
	The *important* icon identifies significant information about the feature or operation that is being described.
	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

SQL Syntax Conventions

This guide uses the following conventions to specify DataBlade SQL syntax:

- Square brackets ([ ]) surround optional items.
- Ellipses (...) follow items that can be repeated one or more times.

Tip: *Standard DataBlade documentation uses curly braces to indicate repeatable items. However, the Informix TimeSeries DataBlade module uses curly braces as a part of calendar pattern syntax.*

- A vertical line (|) separates alternatives.
- Comma-separated lists of values are indicated by a parameter value with a "_commalist" suffix. For example, colorlist=*color_commalist* might indicate "colorlist=red, blue, green."
- SQL parameters are italicized; function arguments that must be specified as shown are not italicized.

These function syntax conventions used in context look like this:

```
myfunction([optionalarg], repeatablearg, ... set | noset,
myparam=my_commalist)
```

Although new line characters are allowed in most SQL syntax, they must not appear within quoted strings in SQL statements. The examples in this manual conform to this standard and wrap accordingly. For example:

```
insert into CalendarTable(c_name, c_calendar)
    values('my_cal',
        'startdate(1994-01-01 00:00:00.00000),
pattstart(1994-01-02 00:00:00.00000), pattern({24 off, 120
on, 24 off}, hour)');
```

Additional Documentation

This documentation set includes the printed manual *Informix TimeSeries DataBlade Module User's Guide* and on-line documents.

This section describes the following parts of the documentation set:

- On-line documentation
- Related reading

On-Line Documentation

The following table describes the on-line files that supplement the information in this manual.

On-Line File	Filename	Purpose
Release notes	TMSREL.TXT	Describes the following items: ■ Registration and upgrade procedures ■ Feature differences from earlier versions of Informix products and how these differences might affect current products ■ Information about known problems, their workarounds, and fixed bugs
Documentation notes	TMSDOC.TXT	Describes features not covered in the manuals or modified since publication.
Machine notes	TMSMAC.TXT	Describes platform-specific information regarding the release.

The on-line notes are located in the **$INFORMIXDIR/extend /TimeSeries.*version*** directory, where *version* is the DataBlade module version number plus a server version number in the form n.nn.UCn. For example, for the Version 3.1 release, the directory is **TimeSeries.3.10.UC1**.

Please examine the on-line files because they contain vital information about application and performance issues.

Related Reading

The following related Informix documents complement the information in this manual set:

- The *BladeManager User's Guide* describes how to register DataBlade modules.
- The *DataBlade API Programmer's Manual* provides a complete reference for the DataBlade API, which is used in the development of applications that interact with Informix Dynamic Server with Universal Data Option.

- The *INFORMIX-Universal Server Administrator's Guide* provides information on creating the dbspaces needed for creating the containers that hold time series data.
- For information about Informix products implement Structured Query Language (SQL), read the:
  - *Informix Guide to SQL: Tutorial*. It provides a tutorial on SQL and describes the fundamental ideas and terminology for planning and implementing a relational database.
  - *Informix Guide to SQL: Reference*. It includes details of the Informix system catalog tables, describes Informix and common environment variables that you should set, and describes the column data types that Informix database servers support.
  - *Informix Guide to SQL: Syntax* volumes. They list and describe standard Informix SQL and Stored Procedure Language (SPL) constructs.
- The *Extending INFORMIX-Universal Server: User-Defined Routines* manual explains how to define your own functions and procedures for use in an Informix Dynamic Server with Universal Data Option database. It defines common considerations for SPL routines and external routines.
- The *Extending INFORMIX-Universal Server: Data Types* manual explains how to create distinct and opaque data types, and how to write support functions for opaque data types.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

doc@informix.com

Or, send a facsimile to Technical Publications at:

650-926-6571

We appreciate your feedback.

Chapter

Getting Started

1

In This Chapter

This chapter describes the installation procedures, the location of the example files, and important restrictions to using SQL with the Informix TimeSeries DataBlade module.

Installation

To install the Informix TimeSeries DataBlade module, follow the instructions on the *Read Me First* sheet.

To register the Informix TimeSeries DataBlade module, follow the instructions in the on-line release notes file, **TMSREL.TXT** and the *BladeManager User's Guide*.

The **$INFORMIXDIR/extend** subdirectory of the Informix Dynamic Server with Universal Data Option installation directory contains a subdirectory for the Informix TimeSeries DataBlade module data type extension to the Informix database system called **TimeSeries.*version***, where *version* is the DataBlade module version number plus a server version number in the form n.nn.UCn. For example, for the Version 3.1 release, the directory is **TimeSeries.3.10.UC1**. This directory contains the installation scripts, on-line notes, executables, and libraries for the Informix TimeSeries DataBlade module.

Example Files

The SQL examples in this manual are based on the assumption that certain tables, types, and other database elements have been defined and that the examples were executed through the DB-Access interactive query processor. The setup script for the SQL examples in this manual is called **examples_setup.sql** and is located in the **examples** directory of the Informix TimeSeries DataBlade module installation. Sample query files are also located in the **examples** directory. The output of the examples is formatted to appear as it would in DB-Access.

To install the sample database schema (by running **examples_setup.sql**) and to compile the sample C programs, enter the following command from the **$INFORMIXDIR/extend/TimeSeries.3.2.UC1/examples** directory:

```
make -f Makefile.your_platform MY_DATABASE=yourdbname
```

where *your_platform* is the name of your operating system, and *yourdbname* is the name of your database.

To avoid locking conflicts, if you run any of the SQL examples in this manual, precede them with the BEGIN WORK statement and follow them with the ROLLBACK WORK statement.

NT and UNIX Versions of the TimeSeries DataBlade Module

The TimeSeries DataBlade module is available in an NT version and a UNIX version. These are the major differences in using these different versions:

- Different file separators are used in path or directory specifications. This manual shows the file separator for UNIX: a forward slash (/). If you are using the NT version of the TimeSeries DataBlade module, substitute the UNIX file separator with a backslash (\). For example, in the NT version of the TimeSeries DataBlade module, the examples directory is:

  $INFORMIXDIR\extend\TimeSeries.3.2.UC1\examples

- The **Makefile** file that installs the sample database schema has different contents.

SQL Limitations

Informix TimeSeries DataBlade module data can be used with most SQL constructs. The exceptions are:

- there is no **LessThan** (<) operator defined on time series data.
- you cannot use the SELECT UNIQUE statement on columns of type **TimeSeries**, since it uses **LessThan** to sort data. However, you can use the UNION ALL statement.
- you cannot use the ALTER TYPE statement.

Chapter

Overview of Time Series

2

In This Chapter

This chapter provides an overview of the Informix TimeSeries DataBlade module. It covers:

- the definition of a time series and how the Informix TimeSeries DataBlade module extends Informix Dynamic Server with Universal Data Option.
- the architecture of the Informix TimeSeries DataBlade module.
- how time series data is organized.
- the sorts of tasks Informix TimeSeries DataBlade module routines can perform.

Definition of a Time Series

A *time series* is a timestamped series of data entries, such as minute-by-minute reports of stock prices and trading volumes.

This kind of data is stored and analyzed by applications in many different industries, including manufacturing, journalism, science, and engineering. For example, a biotechnology company could take measurements of conditions in the tanks in which it grows cell cultures. The time series would record the temperature, pressure, acidity, and other parameters every minute. The company would use the data to monitor the health of the culture and analyze the data to determine the optimal growing conditions.

Time series data is also used in the financial world, for corporate financial reporting, stock prices, bond yields, and derivative securities. The examples in this manual are for stock trading.

Although relational database management systems can store time series for standard types by storing one row per timestamped data entry, performance is poor and storage is inefficient. Moreover, developing or changing applications using nonextensible relational systems is difficult; many aspects of business semantics must be managed by individual applications, not by the database server. Alternatively, nonrelational time series implementations suffer from limitations such as lack of generality and extensibility, predefined limits on the kind and structure of the data, and inability to combine time series data with other information.

In contrast, the Informix TimeSeries DataBlade module provides:

- high-performance storage and access architecture, consisting of containers to hold time series data outside of the database table.
- a rich set of time series analysis routines.
- full support for time series data as an object data type, using the **TimeSeries** data type.

With the Informix TimeSeries DataBlade module, the database management system is extended to "understand" time series and temporal data as a first-class type in a database. The time series entries are objects that the database can manipulate, rather than opaque large objects.

All entries for a particular time series are located in the same row of a database table. For example, all information on IBM's stock performance might be in a single row with two columns: one identifying the stock's name, and the other containing all the time series data. Thus, the entries are already identified by stock name, avoiding one of the sorting steps that conventional relational databases must perform. Within the time series column, entries are additionally indexed by timestamp. This makes retrieving values that are close in time particularly efficient.

Time series are efficiently stored using reusable business calendars, created with the **Calendar** and **CalendarPattern** data types provided by the Informix TimeSeries DataBlade module. Calendars determine the validity of data and organize time series data according to user-specified patterns.

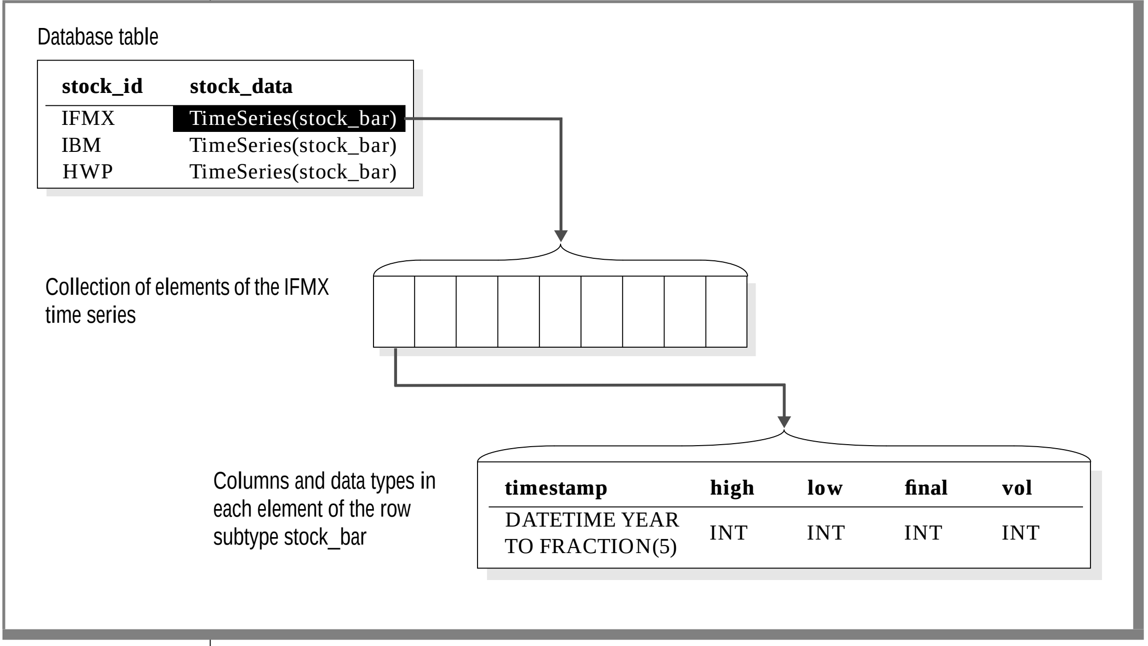
In addition, the Informix TimeSeries DataBlade module provides system tables to store information about time series, calendars, calendar patterns, and containers.

The Informix TimeSeries DataBlade module allows developers to focus on interpreting, rather than managing, the data. Sophisticated data evaluation models can be added (or modified) within the database system and executed in either the server or the client, thus supporting common use by all application developers.

Time Series Architecture

The **TimeSeries** data type acts as a type constructor for a time series subtype. The resulting **TimeSeries(*subtype*)** data type is a collection of row data types. A row data type consists of a group of named columns, of the same or different data types, within a single database column. You define the row subtype to suit your data requirements. Figure 2-1 illustrates the **TimeSeries(*subtype*)** data type.

Figure 2-1
TimeSeries Data Type Architecture



The database table in this example has two columns: a **stock_id** column containing the stock name, and a **stock_data** column containing the time series. Each row in the table contains a different time series. In this example, all three rows in the table have a time series of subtype **stock_bar**. The time series row subtype defines the structure of the elements.

The time series consists of elements, each representing a single row of data for a particular timestamp. The elements are ordered by timestamp.

Each element is a row data type containing columns, beginning with a timestamp column. The timestamp column must be of type DATETIME YEAR TO FRACTION(5), which specifies that the timestamp has the precision to store a time range from one year to 10 microseconds. Timestamps must be unique; multiple entries cannot have the same timestamp.

The Informix TimeSeries DataBlade module supports most of the data types allowed for row data types in Informix Dynamic Server with Universal Data Option, except data types that have an **Assign** or **Destroy** function defined. Other row data types or collections are allowed as columns in the row subtype. The number of the columns in an element is not restricted. The preceding example contains columns for the high, low, and final values of the stock, as well as the volume of stock traded.

In this example, each time series of type **stock_bar** necessarily has the same columns; however, each time series can have its own calendar and time series starting date.

Time series data can reside in the table column or, if the data is larger than a user-defined limit, in a container. See "Time Series Storage" on page 2-8 for more information on containers. Whether or not the time series data is in the table column, a column containing a time series also contains 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, the type of the time series, or exceptional conditions about the time series. The advantage of keeping this type of information with 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. When you create a time series, you can specify whether or not to include metadata. See "Creating a Time Series Containing Metadata" on page 4-10 for more information on metadata.

Kinds of Time Series

There are two kinds of time series: *regular* and *irregular.* A regular time series stores data for regularly spaced timepoints, with respect to a calendar, while an irregular time series stores data for arbitrary timepoints. Regular time series are appropriate for applications that record entries at predictable timepoints, such as stock summary data that is recorded every business day. Irregular time series are appropriate when the data arrives unpredictably, such as when the application records every stock trade as it happens.

Each element in a time series represents data associated with a time interval. The timestamp associated with an interval marks the beginning of the interval. Entries for a timestamp typically summarize events for that timestamp's interval (for example, high, low, and volume for a stock over a minute's time). In a regular time series, each element is the same length. Regular elements *persist* only for the length of an interval as defined by the calendar associated with the time series, and missing elements are null. In irregular time series, each element can be a different length. Irregular elements persist until the next element; there are no null elements.

A main distinction between the two kinds of time series is that irregular time series lack the concept of *offset*; they have no mapping between the timepoint associated with an element and its position relative to the start of the time series. Because of this, regular time series are stored more efficiently than irregular time series. The timestamps are not stored in regular time series, instead, they are computed from the element's offset. See "Offsets" on page 2-12 for a discussion of offsets.

In most respects, regular and irregular time series have the same characteristics. Therefore, most time series routines apply to both regular and irregular time series. Whether the time series is regular or irregular is specified when the time series is created. See "Creating and Populating a Time Series" on page 4-8.

Time Series Storage

Time series data may grow too large to fit into the row of a table, currently limited to approximately 2048 bytes. When this happens, the Informix TimeSeries DataBlade module moves the data portion of the time series into a *container*. A container is a structure that the Informix TimeSeries DataBlade module creates and maintains to hold time series data. Different time series can use the same container, as long as they are all regular or all irregular time series. A container exists in a dbspace, which is a named section of physical memory. You can create a dbspace using the ON-Monitor or onspaces utilities, or use the default dbspace, called **rootdbs**. See the *INFORMIX-Universal Server Administrator's Guide* for more information.

You create a container by running the SQL **TSContainerCreate** procedure call and remove it by running the **TSContainerDestroy** procedure (see "TSContainerCreate" on page 8-140 and "TSContainerDestroy" on page 8-142). You can specify the container to use for your time series when you create your time series. When a time series has data in a container, only a small header is left in the original table. This header, which is maintained internally by the Informix TimeSeries DataBlade module, contains information that identifies where in the container the Informix TimeSeries DataBlade module places the data.

You can control when a time series is promoted from in-row to in-container representation with the *threshold* parameter, which you set when creating a time series (see "TSCreate" on page 8-143). Once data is moved to a container, it cannot move back to a row, even if the number of elements drops below the threshold. A container is not required unless the data for the time series grows too large to fit in a single row of a table.

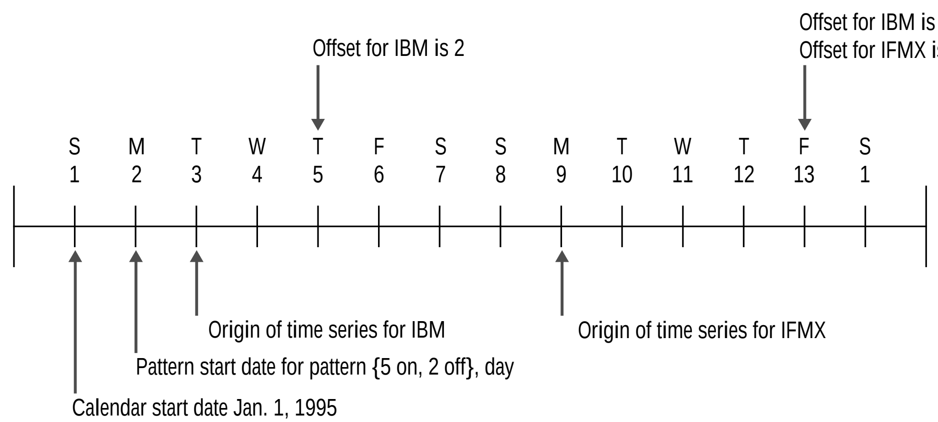
Time Series Organization

Time series are controlled by calendars using the **Calendar** and **CalendarPattern** data types. Figure 2-2 on page 2-9 illustrates the following concepts associated with calendars:

- Calendars
- Calendar patterns

- Time series origin
- Offset

Figure 2-2
Sample Time Series Time Line



This illustration covers the first two weeks of 1995. It shows:

- two regular time series (IBM and IFMX), their origins, and their offsets.
- a calendar with its start date, calendar pattern, and calendar pattern start date.

Calendars

Every time series is associated with a *calendar*. A calendar defines a set of valid times at which the time series can record data; it determines when and how often entries are accepted. Thus, calendars control the data in your database. For regular time series, calendars create the vector structure by converting timestamps into offsets (see "Offsets" on page 2-12 for more information). Although irregular time series do not have offsets, they still use calendars to define valid entry times.

Calendars are represented by the **Calendar** data type, which are composed of the following values:

start date The date the calendar begins. All other dates associated with the calendar and its time series must be the same as or later than this date.

pattern A regularly occurring set of valid and invalid time intervals. Calendar patterns are fully explained in the section "Calendar Patterns," next.

interval The calibration of the calendar pattern. Intervals are also explained in the section "Calendar Patterns."

pattern start date The date the calendar pattern starts. This date must be later than or equal to the calendar start date, and earlier than or the same as the time series origin (start date). This date must be less than one calendar pattern length after the calendar start date. See the section "Calendar Patterns" for more information.

For example, the calendar shown in Figure 2-2 on page 2-9 has the following characteristics:

- The calendar starts on January 1, 1995.
- The calendar pattern has five days on, two days off.
- The interval is days.
- The calendar pattern starts on Monday, January 2, 1995.

Regular and irregular time series can both use the same calendars.

Calendar information from the **Calendar** data type is stored in the **CalendarTable** system table. See Chapter 3, "Data Types and System Tables," for more information.

Calendar Patterns

Each calendar has a *calendar pattern* of time intervals 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. In the calendar pattern syntax, *on* indicates a valid period and *off* an invalid interval. The calendar pattern also indicates the time unit in which the interval is measured, for example, minute, hour, day, or month. Calendar patterns are represented by the **CalendarPattern** data type, which is specified as:

```
{pattern specification}, interval
```

For example, the calendar pattern shown in Figure 2-2 on page 2-9 is:

```
{5 on, 2 off}, day
```

Since the calendar pattern start date is Monday, January 2, 1995, the calendar pattern specifies that beginning on Mondays, five days are valid (weekdays) and two days are invalid (weekends). The interval for valid entries is a day; there are five entries per week.

The calendar must contain a calendar pattern starting time as well as a calendar starting time because calendar and calendar pattern starting times may not coincide. For example, most years do not begin on the first day of the week.

The calendar starting time and the calendar pattern starting time must be within a single calendar pattern length. A *calendar pattern length* is the number of intervals before the pattern repeats. For example, if the calendar pattern length is a week (its interval is a day), the calendar starting time can be at most a week before the calendar pattern starting time.

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. See “HideElem” on page 8-88 for information on hiding elements.

You can use the calendar pattern for a time series with the **WithinR** and **WithinC** functions to search for data around a specified timepoint. **WithinR** 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. **WithinC** performs a *calibrated* search. A calibrated search proceeds both forward and backward to the "natural" interval boundaries surrounding the given starting timepoint. See "WithinC, WithinR" on page 8-171 for more information.

Time Series Origin

Each time series has its own *timepoint of origin*, which must be at or after the starting timestamps of the calendar and the calendar pattern. This is the first timepoint of interest to the user. In a regular time series, it is treated as offset 0.

For example, the calendar in Figure 2-2 on page 2-9 starts on January 1, 1995, the calendar pattern starts on January 2, 1995, and the time series for IBM starts on January 3, 1995.

Timepoints before the origin in the calendar cannot be referred to without declaring an earlier timepoint of origin.

Offsets

A regular time series can be thought of as a vector of points. The number of intervals between the point and the origin is called the *offset*. Each offset is associated with a timestamp.

Calendars determine all the possible points on the vector. Each entry after the time series origin has an offset number assigned to it. Using an offset allows quicker calculations, and saves space in the database because the offset can be calculated instead of storing the timestamp.

For example, the vector for the time series for IBM, as taken from Figure 2-2 on page 2-9, is:

Day	T	W	T	F	M	T	W	T	F
Date	3	4	5	6	9	10	11	12	13
Offset	0	1	2	3	4	5	6	7	8

Days that are marked as invalid in the calendar pattern (Saturdays and Sundays) have no offset.

The vector for the time series for IFMX, as taken from Figure 2-2 on page 2-9, is:

Day	M	T	W	T	F
Date	9	10	11	12	13
Offset	0	1	2	3	4

Regular time series have nullable timestamps. When you insert an element without a timestamp into a regular time series, it is put into the next available offset on the vector. Any offsets you skip by entering an element with a timestamp more than one offset after the last one in the vector are marked as null. Skipped offsets can be updated later.

Both offsets and timestamps can be used as indexes to elements. The calendar associated with a regular time series automatically converts between timestamps and offsets.

Informix TimeSeries DataBlade Module Routines

The Informix TimeSeries DataBlade module provides routines to manipulate:

- calendar patterns.
- calendars.
- time series.

The Informix TimeSeries DataBlade module provides both SQL and API routines to manipulate time series, thus allowing the application developer the flexibility to call routines from the interactive query processing tool, DB-Access, or to call routines from within the application on either the client or the server computer.

Calendar and Calendar Pattern SQL Routines

Calendar and calendar pattern SQL routines can perform the following types of operations:

- Create the intersection of calendars or calendar patterns
- Create the union of calendars or calendar patterns

These routines are described in Chapter 6, "Calendar Pattern Routines," and Chapter 7, "Calendar Routines."

Time Series SQL Routines

Time series SQL routines can perform the following types of operations:

- Return information on the time series definition
- Manipulate elements, either as individuals or as sets
- Create and load time series
- Perform statistical calculations
- Perform arithmetic calculations
- Manage containers
- Convert between timestamps and offsets
- Extract periods of time series

See Chapter 8, "Time Series SQL Routines," for more information.

Time Series API Routines

The API routines perform many of the same tasks that the SQL routines do for time series.

See Chapter 9, “Time Series API Routines,” for more information.

Chapter

Data Types and System Tables

3

In This Chapter

The Informix TimeSeries DataBlade module adds specialized data types and system tables. This chapter explains how these data types and system tables are used.

Data Types

The Informix TimeSeries DataBlade module defines and supports the following data types in Informix Dynamic Server with Universal Data Option:

- **CalendarPattern**
- **Calendar**
- **TimeSeries**

The following sections describe each of the Informix TimeSeries DataBlade module data types.

The CalendarPattern Data Type

The **CalendarPattern** data type is an opaque data type that is the structure for a calendar pattern. A calendar pattern specifies an interval duration and the pattern of valid (on) and invalid (off) intervals. Intervals can be in the following time units:

- Second
- Minute
- Hour
- Day
- Week
- Month
- Year

The **CalendarPattern** data type has the following format:

```
{n on|off[, n on|off, ...]}, interval
```

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; once 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.

You can deal with exceptions to your calendar pattern by hiding elements for which there is no data. See "HideElem" on page 8-88 for more information.

The calendar pattern is stored in the **CalendarPatterns** table and can be used or reused in several calendars. See "The CalendarPatterns Table" on page 3-9 for more information on the **CalendarPatterns** table, and "Creating the Calendar Pattern" on page 4-3 for instructions on inserting values into it.

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. See Chapter 6, "Calendar Pattern Routines," for more information.

The Calendar Data Type

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

- a starting timestamp.
- a calendar pattern.
- a calendar pattern starting timestamp.

Calendars specify the times at which the time series can record data. 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(DATETIME YEAR TO FRACTION(5)),
pattstart(DATETIME YEAR TO FRACTION),
pattern(CalendarPattern) | pattname(varchar)'
```

The parameters and their values are shown in the following table.

Parameter	Data Type	Description
startdate	DATETIME YEAR TO FRACTION(5)	Calendar start date.
pattstart	DATETIME YEAR TO FRACTION(5)	Calendar pattern start date. Must be the same as or later than the calendar start date. The calendar and calendar pattern start dates must be less than one calendar pattern length apart.
pattern	**CalendarPattern**	Calendar pattern to use. The **pattern** and **pattname** keywords are mutually exclusive; one of them is required.
pattname	VARCHAR	Name of calendar pattern to use from CalendarPatterns table. The **pattern** and **pattname** keywords are mutually exclusive; one of them is required.

To create a calendar, insert the keywords and their values into the **CalendarTable** table. See "The CalendarTable Table" on page 3-10 for specific information on the **CalendarTable** table. See "Creating a Calendar" on page 4-4 for instructions on inserting 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. See Chapter 7, "Calendar Routines," for more information.

The TimeSeries Data Type

The **TimeSeries** data type is constructed from a row data type, and is a collection of row subtypes. See "Time Series Architecture" on page 2-5 for a description and illustration of the **TimeSeries(*subtype*)** data type.

Creating a TimeSeries Column

To create a **TimeSeries** column, first you create the TimeSeries subtype, using the CREATE ROW TYPE statement. See "Creating a Time Series Subtype" on page 4-5 for instructions.

All **TimeSeries** subtypes must have a DATETIME YEAR TO FRACTION(5) data type as the first column. The remaining columns of the row subtype can be any data type supported for row data types by Informix Dynamic Server with Universal Data Option, except data types that do not have an **Assign** or **Destroy** function (see "Restrictions on Row Data types," later in this section).

After you create the **TimeSeries** subtype, you create the table containing the **TimeSeries** column using the CREATE TABLE statement. See "Creating the Database Table" on page 4-6 for instructions.

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 (see "Creating and Populating a Time Series" on page 4-8).

The maximum allowable size for a single time series element is 1920 bytes.

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

Managing Performance

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.

Restrictions on Row Data types

The current Informix Dynamic Server with Universal Data Option restrictions on which data types can be included in a row type apply to the **TimeSeries** subtype. Row types cannot contain:

- SERIAL and SERIAL8 types.
- types that do not have **Assign** or **Destroy** functions assigned to them, including large object types and some user-defined types.

Time Series Return Types

Returned time series preserve calendar information, and, where possible, the threshold and container information.

Some functions that return a time series subtype require that the return value be cast to a particular time series type, and some do not. 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 needed.

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. Since the time series returned by these functions might not be able to use the container of the original time series, the resulting time series are not associated with a container until you run the **SetContainerName** function to specify the container to use.

System Tables

The system tables included in the Informix TimeSeries DataBlade module are:

- **CalendarPatterns.**
- **CalendarTable.**
- **TSInstanceTable.**
- **TSContainerTable.**

When a calendar is inserted into the **CalendarTable** table, it draws information from the **CalendarPatterns** table. The Informix TimeSeries DataBlade module refers only to **CalendarTable** for calendar and calendar pattern information; changes to the **CalendarPatterns** table have no effect unless **CalendarTable** is updated or re-created.

TSInstanceTable contains information about all time series.

The CalendarPatterns Table

The **CalendarPatterns** table included in the Informix TimeSeries DataBlade module 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. See "Creating the Calendar Pattern" on page 4-3 for instructions.

The CalendarTable Table

The **CalendarTable** table maintains information about the calendars used by the database. Its columns are described in the following table.

Column Name	Data Type	Description
c_version	INTEGER	Internal DataBlade module use only. The version of the calendar. Currently, only version 0 is supported.
c_refcount	INTEGER	Internal DataBlade module use only. 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 DataBlade module use only. The serial number of the calendar.

To insert a calendar into the **CalendarTable** table, use the INSERT statement. See "Creating a Calendar" on page 4-4 for instructions.

The 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.

The columns for the **TSInstanceTable** table are described in the following table.

Column Name	Date 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" on page 8-94).
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(18,1)	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 Informix TimeSeries DataBlade module, and users do not modify it directly, nor should they normally have to view it. Rows in this table are automatically inserted or deleted when large time series are created or destroyed.

The TSContainerTable Table

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

The columns for the **TSContainerTable** table are described in the following table.

Column Name	Date Type	Description
name	VARCHAR(18,1)	The name of the container of the time series. This is the primary key.
subtype	VARCHAR(18,1)	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 container is empty and always was empty. ■ if the time series is regular or irregular.

The **TSContainerTable** table is managed by the Informix TimeSeries DataBlade module, and users do not modify it directly, nor should they normally have to view it. Rows in this table are automatically inserted or deleted when containers are created or destroyed.

Containers are created and destroyed using the **TSContainerCreate** and **TSContainerDestroy** procedures, which insert and delete special rows in the **TSContainerTable** table. For more information, see "TSContainerCreate" on page 8-140 and "TSContainerDestroy" on page 8-142.

To determine which containers exist in the database, run the following query:

```
select name from TSContainerTable;
```

Chapter

Creating and Loading a Time Series

4

In This Chapter

This chapter describes the steps for creating and loading a time series:

1. Set up the calendar.
2. Create the time series column.
3. Create the times series container.
4. Define the time series.
5. Load the data.

Each of these steps is described in the following sections. The example time series defined in this chapter are used throughout this manual.

Setting Up a Calendar

This section explains how to create a calendar.

To set up a calendar

1. Create the calendar pattern by inserting values into the **CalendarPatterns** table.
2. Create the calendar by inserting values into the **CalendarTable** table.

Creating the Calendar Pattern

To create a calendar pattern, you must insert values into the **CalendarPatterns** table using the following syntax:

```
insert into CalendarPatterns
    values('pattern_name', 'calendar_pattern');
```

The *calendar_pattern* is of type **CalendarPattern** and has the following format:

```
{n on|off, ...}, interval
```

The *n* parameter is the number of intervals, `on` or `off` signify validity, and *interval* defines the duration of *n*. See "The CalendarPattern Data Type" on page 3-4 for more information.

The following example inserts a pattern into the **CalendarPattern** table:

```
insert into CalendarPatterns
    values ( 'workweek_day', '{1 off, 5 on, 1 off}, day');
```

The name of the pattern is **workweek_day**. Assuming that the pattern start date is a Sunday, the calendar pattern is in days, with Monday through Friday each having a valid entry.

Creating a Calendar

To create a calendar, you must insert values into the **CalendarTable** table using this syntax:

```
insert into CalendarTable(c_name, c_calendar)
    values('calendar_name', 'calendar');
```

The *calendar* is of type **Calendar** and has the format:

```
startdate(value), pattstart(value),
pattern(value)|pattname(value)
```

See "The Calendar Data Type" on page 3-5 for definitions of each of these parameters.

The following example inserts a calendar called **yearcal79** into the **CalendarTable** table:

```
insert into CalendarTable(c_name, c_calendar)
    values ('yearcal79 ',
    'startdate(1979-01-01 00:00:00.00000), pattstart(1979-
01-07 00:00:00.00000), pattname(workweek_day)');
```

This calendar starts on January 1, 1979; its pattern starts on January 7, 1979; and it uses the pattern **workweek_day**. See "The CalendarTable Table" on page 3-10 for specific information on the CalendarTable table.

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

```
insert into CalendarTable(c_name, c_calendar)
    values('my_cal',
        'startdate(1994-01-01 00:00:00.00000),
pattstart(1994-01-02 00:00:00.00000), pattern({24 off, 120
on, 24 off}, hour)');
```

Creating a Time Series Column

This section explains how to create a column for your time series.

To create a time series column

1. Create a time series subtype.
2. Create a table containing a time series column.

Creating a Time Series 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. An example of such a subtype is a daily stock bar holding a timestamp and the high, low, volume, and close for a day.

The only restriction on the data types in the row subtype that the Informix TimeSeries DataBlade module imposes is that the first column must be a timestamp of type DATETIME YEAR TO FRACTION(5). This data type specifies that the timestamp contain entries for all time intervals from one year to 10 microseconds, for example: 1995-01-01 12:00:05.00000. The number of columns in a subtype is not restricted.

To create the row subtype, use the SQL CREATE ROW TYPE statement. Subtypes for both regular and irregular time series are created in the same way. The syntax for creating a time series subtype is:

```
create row type subtype_name(
    col1    datetime year to fraction(5),
    col2    any_data_type,
    ...
);
```

Example: stock_bar Subtype for Regular Time Series

The following example creates a valid regular time series subtype, called **stock_bar**:

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

Example: stock_trade Subtype for Irregular Time Series

The following example creates a valid time series 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
);
```

Creating the Database Table

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

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)
);
```

Example: daily_stocks Table for Regular Time Series

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. This table is used to hold regular time series.

Example: activity_stocks Table for Irregular Time Series

The following example creates a table called **activity_stocks** that contains an 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. This table is used to hold irregular time series.

Creating a Time Series Container

You create a container by using the **TSContainerCreate** procedure. See "TSContainerCreate" on page 8-140 for a full description of **TSContainer-Create**. The **TSContainerCreate** procedure takes arguments for the container name, the name of the dbspace, the name of the time series type, the size of the container, and the size of the growth increments. The dbspace you want to use must already exist, or you can use the default dbspace, **rootdbs**. See the *INFORMIX-Universal Server Administrator's Guide* for more information on dbspaces.

The following example creates a container named **new_cont** for the time series type **stock_bar** in the space **rootdbs**:

```
execute procedure TSContainerCreate('new_cont',
'rootdbs','stock_bar', 0, 0);
```

Creating and Populating 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, what the format of that data is. The following table lists the options for creating and populating a time series.

Task	Function
Create an empty time series	■ **TSCreate** (regular time series), or **TSCreateIrr** (irregular time series)
Create an empty time series with metadata	■ **TSCreate** with the *metadata* argument (regular time series), or **TSCreateIrr** with the *metadata* argument (irregular time series)
Create and populate a time series	■ **TSCreate** with the *set_ts* argument (regular time series), or **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), or **TSCreateIrr** with the *set_ts* and *metadata* arguments (irregular time series)
Populate an existing time series	■ **BulkLoad** ■ Other functions, such as **PutElem**

Creating a Time Series with TSCreate or TSCreateIrr

You can use the **TSCreate** (for regular time series) or **TSCreateIrr** (for irregular time series) functions to create empty time series. Additionally, these functions have two optional arguments: a *set_ts* argument that allows you to populate the time series with data, and a *metadata* argument that allows you to add user-defined metadata to the time series. You can use either or both optional arguments. See "TSCreate" on page 8-143 and "TSCreateIrr" on page 8-147 for more information.

Creating an Empty Time Series

You can create an empty time series directly. The **TSCreate** and **TSCreateIrr** functions create an empty time series based on the calendar name, the origin timestamp, the threshold, the flags, the number of elements, and the container name.

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

```
insert into daily_stocks values(
    901,'IBM', TSCreate('daycal',
        '1994-01-03 00:00:00.00000',20,0,0, NULL));
```

Creating and Populating a Time Series

You can simultaneously create a time series and insert data into a table containing a time series column using the **TSCreate** or **TSCreateIrr** function with an additional argument that is a set of row types. This method is useful if the data to load resides in a separate table.

For example, if there is a table called **activity_load_tab** with a column **set_data** of type **SET(stock_trade)**, the following query could be used to create a time series and insert it into the **activity_stocks** table:

```
insert into activity_stocks
    select 1234,
       TSCreateIrr('daycal',
          '1994-01-03 00:00:00.00000'::datetime year to
fraction(5),
          20, 0, NULL,
          set_data)::timeseries(stock_trade)
    from activity_load_tab;
```

Creating a Time Series Containing Metadata

You can create an empty or populated time series that also contains user-defined metadata using the **TSCreate** or **TSCreateIrr** functions with an additional argument for the metadata data type. See "Time Series Architecture" on page 2-5 for a description of user-defined metadata.

To add metadata to a time series, you must create a distinct data type based on the **TimeSeriesMeta** data type with this SQL statement:

```
create distinct type MyMetaData as TimeSeriesMeta
```

When you create a distinct data type it inherits the representation and routines of the source data type. In this case, the **TimeSeriesMeta** data type is an opaque data type of variable length, up to a maximum length of 512 bytes. The **TimeSeriesMeta** data type is an argument in several routines that will automatically accept the distinct *MyMetaData* data type.

An opaque data type require support functions, such as input, output, send, receive, and so on. These functions are called when the time series functions are called with the opaque data type as an argument. However, by not providing support functions for the **TimeSeriesMeta** data type, the Informix TimeSeries DataBlade module allows you to tailor the support functions for *MyMetaData* to conform to your metadata requirements. See *Extending INFORMIX-Universal Server: Data Types* for instructions on creating support functions.

Once 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.

Creating 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 timestamp 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 *paramname* parameter can be any of these parameters, in any order:

- **calendar**
- **origin**
- **threshold**
- **container**
- **irregular**
- **datafile**

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

Parameter Name	Required	Value
calendar	Yes	Name of the calendar to use. There is no default name.
origin	No	Timestamp of the origin of the time series. The default origin is the calendar start date.
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.
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.

(1 of 2)

Parameter Name	Required	Value
irregular	Yes (for irregular)	No value, just the string “irregular”. This parameter must be included for an irregular times series, but cannot be included for a regular time series.
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.
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.

(2 of 2)

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 timestamp column. Each element is surrounded by parentheses and followed by an @ symbol and a timestamp. The timestamp is optional for regular time series but mandatory for irregular time series. Elements are separated by a comma. 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: Creating and Inserting into a Regular Time Series

The following is an example of an INSERT statement for a regular time series created in the table **daily_stocks**:

```
insert into daily_stocks values (1234, 'informix',
            'origin(1994-01-03 00:00:00.00000),
calendar(daycal), [(350, 310, 340, 1999), (362, 320, 350,
2500)]');
```

This INSERT statement creates a time series that starts on January 3, 1994, at the 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 Informix TimeSeries DataBlade module attempts to move the data to a container, but since there is no container specified, an error is raised.

Example: Creating and Inserting into an Irregular Time Series

The following is an example of 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(1994-10-06
00:00:00.00000), calendar(daycal),
[(6.25,1000,1,7,2,1)@1994-10-06 12:58:09.12345, (6.50, 2000,
1,8,3,1)@1994-10-06 12:58:09.23456]');
```

The INSERT statement creates a time series that starts on October 6, 1994, at the time of day specified by the calendar called **daycal**. Two rows of data are inserted with the specified timestamps.

Creating a Time Series with the Output of a Function

Many functions return time series. A time series created in this way is no different than one created by any of the previously mentioned methods.

The container for these time series 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.

For functions that take multiple time series arguments and return a time series (such as **Union**, **Intersect**, **Sum**, and **Apply**), the container of the result time series is set to NULL. No data is stored to disk unless the user sets a different container using **SetContainerName**.

For more information on these functions, see Chapter 8, "Time Series SQL Routines."

Loading Data into an Existing Time Series

Once a time series has been created, there are several ways to load data into it. The functions you can use are **BulkLoad**, **PutElem**, **PutSet**, **InsElem**, and **InsSet**.

Loading 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 filename as arguments. The filename 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.

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, that is, 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(1994-01-03 00:00:00.00000, 1.1, 2.2)
row(1994-01-04 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:

```
1994-01-03 00:00:00.00000   1.1     2.2
1994-01-04 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.

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 *INFORMIX-Universal Server Performance Guide* for more information on **onload**.

Example: Loading 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',
        '1994-01-03 00:00:00.00000',20,0,0, NULL));

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

Loading Data with Other Functions

You can also load data into a time series by using any of the following functions:

- **PutElem** updates a time series with a single element.
- **PutSet** updates a time series with a set of elements.
- **InsElem i**nserts an element into a time series.
- **InsSet i**nserts 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 resides. The *FunctionName* argument is the name of the function. The *data* argument is in the row type data element format (see "Creating a Time Series with Its Input Function" on page 4-10). 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 the section "Binary Arithmetic Functions" on page 8-37.

Chapter

5

Using the Virtual Table Interface

In This Chapter

This chapter describes the virtual table interface provided by the Informix TimeSeries DataBlade module.

About the Virtual Table Interface

The virtual table interface takes the data encapsulated in the **TimeSeries** data type and produces a virtual relational table containing the same data. The virtual table has the same schema as the base table, except for the **TimeSeries** column. The **TimeSeries** column is replaced with the columns of the **TimeSeries** row type.

For example, the table **daily_stocks** contains a **TimeSeries** column called **stock_data** that contains a row type with highest price, lowest price, closing price, and volume columns.

stock_id	stock_name	stock_data
900	IFMX	(*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), ...

The virtual table interface produces a virtual relational table that takes the **stock_data** column row elements and makes them individual columns.

stock_id	stock_name	timestamp*	high	low	final	vol
900	IFMX	*t1*	7.25	6.75	7	1000000
900	IFMX	*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.

Sometimes, it is simpler to write SQL queries for the virtual table than for the underlying base table. This is especially true for SQL queries with qualifications on a **TimeSeries** attribute.

An SQL SELECT statement against a virtual table returns data in ordinary data type format, rather than in the **TimeSeries** data type format.

When you insert data into a virtual table, the underlying base table is automatically updated; this gives you an alternative method of loading data into a **TimeSeries** column.

The virtual table is not a real table stored in the database, and so there is no duplicate storage of data. At any given moment, data visible in the virtual table is also visible in the base table.

Limitations in Using Virtual Tables

This section describes certain limitations that you need to be aware of when you use virtual tables.

Column Name Conflicts

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.

Creating Indexes on Virtual Tables

You cannot create an index on a time series virtual table.

The Size of Virtual Tables

The total length of a row in the virtual table (non-time-series and **TimeSeries** columns combined) is subject to the same limit as on any other database table. The server currently allows up to 32 KB.

Modifying Data in a Virtual 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 with the same timestamp into the virtual table.

Base Tables with Multiple TimeSeries Columns

You can only create a virtual table based on a single **TimeSeries** column. If your base table has more than one **TimeSeries** column, you must use the *TSColName* parameter of the **TSCreateVirtualTab** procedure to specify which **TimeSeries** column is used, as explained in "TSCreateVirtualTab" on page 5-8.

When to Use a Virtual Table

Many of the operations that time series SQL functions and API routines perform can be done using SQL statements against a virtual table. When do you use the virtual table interface and when do you use the time series routines? In many cases the answer is: use whichever interface you are more comfortable with.

This section provides information to help you in considering which interface to use.

The performance of the two interfaces is similar, although in some cases, one interface is slightly faster. For example, it is faster to use a query against a virtual table than to call the **Apply** or **Transpose** routines. However the **Clip** routine is faster than the equivalent operation using a virtual table.

Some operations are difficult or impossible in one interface but are easily accomplished in the other. For example, the aggregation in the following query against a virtual table is difficult to do using time series functions:

```
select avg(vol) from daily_stocks_no_ts
where stock_name = 'IBM'
and timestamp between datetime(1994-1-1) year to day
and datetime(1994-12-31) 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 to do using the virtual table interface.

Deleting elements is not supported by the virtual table interface. You must use the time series SQL functions or API routines to do this.

Creating a Time Series Virtual Table

To create a virtual table, you use the **TSCreateVirtualTab** procedure.

When you create the virtual table, you must specify how the virtual table handles data inserted into it with regard to updating the underlying base table. Two parameters of the **TSCreateVirtualTab** procedure specify this behavior:

- *NewTimeSeries*, described in "The NewTimeSeries Parameter" on page 5-9.
- *TSVTMode*, described in "The TSVTMode Parameter" on page 5-10.

The rest of this section describes the **TSCreateVirtualTab** procedure.

TSCreateVirtualTab

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)
returns integer;
```

VirtualTableName	The name of the new virtual table.
BaseTableName	The name of the base table.
NewTimeSeries	(optional) The new empty time series to create if inserts are allowed for a time series that does not yet exist. This option is explained in "The NewTimeSeries Parameter" on page 5-9.
TSVTMode	(optional) Sets the virtual table mode, as described in "The TSVTMode Parameter" on page 5-10.
TSColName	(optional) For base tables that have more than one **TimeSeries** column, use this parameter to specify the name of the **TimeSeries** column to be used to create the virtual table. The default value for *TSColName* is `NULL`, in which case the base table must have only one **TimeSeries** column.

Returns

TsCreateVirtualTab is a procedure; it does not return a value.

Example

The following example creates a virtual table called **daily_stocks_virt** based on the table **daily_stocks**. Since 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 TimeSeries DataBlade module creates a new empty time series that uses the calendar **daycal** and has an origin of January 3, 1994.

```
execute procedure TSCreateVirtualTab('daily_stocks_virt',
'daily_stocks',
'calendar(daycal), origin(1994-01-03 00:00:00.00000)'
);
```

For more information about how to build a string specifying a time series, as used for the *NewTimeSeries* parameter, refer to "Creating a Time Series with Its Input Function" on page 4-10.

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. The behavior in either case is described in the rest of this section.

NewTimeSeries Parameter Omitted

To prohibit inserts if a time series does not yet exist, omit the *NewTimeSeries* parameter when you create the virtual table.

You may insert a new row into the virtual table only if the row has a corresponding time series in the base table.

The new element is added to the corresponding time series as if a **PutElem** routine were used in an UPDATE statement on the base table.

If the new row does not have a corresponding time series, the attempt to insert results in an error.

NewTimeSeries Parameter Specified

To allow inserts if a time series does not yet exist, use the *NewTimeSeries* parameter to specify the time series input string.

You may insert a new row into the virtual table whether or not a corresponding time series exists in the base table.

If a time series does not exist, the TimeSeries DataBlade module creates a new time series, inserts the new element into the time series, and adds the new time series to the base table.

If a time series does exist, the new element is added to the corresponding time series as if a **PutElem** routine were used in an UPDATE statement on the base table.

The example shown in "TSCreateVirtualTab" on page 5-8, demonstrates the use of the *NewTimeSeries* parameter.

The TSVTMode Parameter

You use the *TSVTMode* parameter of the **TSCreateVirtualTab** procedure to control how data is updated in the base table when you perform an insert in the virtual table. *TSVTMode* has two values: 0 and 1.

TSVTMode = 0

The default value for *TSVTMode* is 0. By default, the TimeSeries DataBlade module uses **PutElemNoDups** to add an element to the underlying time series. If an element already exists at the same timepoint, the new element replaces the existing element. This enables you to perform bulk updates of the underlying time series. Refer to "PutElemNoDups" on page 8-111 for information about the **PutElemNoDups** routine.

TSVTMode = 1

The TimeSeries DataBlade module uses **PutElem** to add an element to the underlying time series. Refer to "PutElem" on page 8-109 for information about the **PutElem** routine.

Loading Data Using 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 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 need to perform incremental loading.

To load data via a virtual table

1. Put your input data in a single file.
2. Format the data according to the standard Informix load file format.
3. Use any of the Informix load utilities: **pload**, **onpload**, **dbload**, or the **load** command in DB-Access.

Refer to the *INFORMIX-Universal Server Administrator's Guide* for information about Informix load file formats and load utilities.

Refer to "Creating a Time Series Virtual Table" on page 5-7 for information about how the virtual table handles updates of the underlying base table.

Dropping 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.

Managing 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. For example:

```
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.

Tracing

Trace functions are available to help you debug your work with virtual tables. You should not use these trace functions unless you are working with an 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.

Detailed descriptions of these functions follows.

TSSetTraceFile

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 resides 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()**, refer to the *DataBlade API Programmer's Manual*.

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

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()**, refer to the *DataBlade API Programmer's Manual*.

Returns

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

Example

The following example turns tracing on:

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

Examples

The examples in this section show how to:

- create a time series virtual table.

- run queries against the virtual table and against its underlying base table.

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

Querying the Base Table

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

```
create row type stock_bar(
    timestampdatetime 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)
);
```

daily_stocks contains the following data.

stock_id	stock_name	stock_data
900	IFMX	(*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 the functions supplied with the TimeSeries DataBlade module. 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;
```

Creating 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 TimeSeries DataBlade module uses **PutElemNoDups** to add an element to the underlying time series in **daily_stocks**.

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

stock_id	stock_name	timestamp*	high	low	final	vol
900	IFMX	*t1*	7.25	6.75	7	1000000
900	IFMX	*t2*	7.5	6.875	7.125	1500000
...	...	...			...	...
901	IBM	*t1*	97	94.25	95	2000000

(1 of 2)

stock_id	stock_name	timestamp*	high	low	final	vol
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.					

(2 of 2)

Querying 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. For example:

```
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**.

Chapter

Calendar Pattern Routines

6

In This Chapter

This chapter, along with Chapter 7, "Calendar Routines," and Chapter 8, "Time Series SQL Routines," describes the routines provided by the Informix TimeSeries DataBlade module that are intended for execution by the user. These routines can be executed through the interactive query processor DB-Access or sent from an application using the DataBlade API function **mi_exec**.

In particular, the routines in this chapter allow the user to manipulate calendar patterns. For instance, there is a function that finds the intersection of two calendar patterns.

For information about DB-Access and the interactive query processor, see the *DB-Access User Manual*. For information about DataBlade API and **mi_exec**, see the *DataBlade API Programmer's Manual*.

AndOp

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 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 Topics

"AndOp" on page 7-4

CalPattStartDate

The **CalPattStartDate** function takes a calendar name and returns a DATETIME containing the startdate 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 startdate 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 Topics

"ts_cal_pattstartdate()" on page 9-20

"CalStartDate" on page 7-12

Collapse

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 Topics

"Expand" on page 6-9

Expand

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 Topics

"Collapse" on page 6-7

NotOp

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
```

OrOp

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 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 below 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 Topics

"OrOp" on page 7-13

Chapter

Calendar Routines

7

In This Chapter

This chapter, along with Chapter 6, "Calendar Pattern Routines," and Chapter 8, "Time Series SQL Routines," describes some of the Informix TimeSeries DataBlade module routines that are intended for execution by the user. These routines can be executed through the interactive query processor DB-Access or sent from an application using the DataBlade API function **mi_exec**.

The routines described in this chapter enable the user to manipulate calendars. For instance, there is a function that performs the union of two calendars and another that finds the intersection of two calendars.

For information about DB-Access and the interactive query processor, see the *DB-Access User Manual*. For information about DataBlade API and **mi_exec**, see the *DataBlade API Programmer's Manual*.

AndOp

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 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 others 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(1994-01-01 00:00:00), pattstart(1994-
              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(1994-04-01 00:00:00), pattstart(1994-
              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(1994-04-01 00:00:00), pattstart(19
                94-04-03 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 Topics

“AndOp” on page 6-4

“OrOp” on page 7-13

CalIndex

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

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 timestamps.

Example

The following query returns the number of intervals in the calendar **daycal** between 1994-01-03 and 1994-01-05:

```
select CalIndex('daycal',
        '1994-01-03 00:00:00.00000',
        '1994-01-05 00:00:00.00000')
    from systables
    where tabid = 1;
```

Related Topics

"ts_cal_range()" on page 9-21

"ts_cal_range_index()" on page 9-23

"ts_cal_stamp()" on page 9-25

CalRange

The **CalRange** function returns a set of valid timestamps 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 timestamp in the calendar.

end_stamp The end point of the range.

num_stamps The number of timestamps to return.

Description

The first syntax specifies the range as between two given timestamps. The second syntax specifies the number of valid timestamps to return after a given timestamp.

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

Returns

A list of timestamps.

Example

The following query returns a list of all the timestamps between 1994-01-03 and 1994-01-05 in the calendar **daycal**:

```
execute function CalRange('daycal',
        '1994-01-03 00:00:00.00000',
        '1994-01-05 00:00:00.00000'::datetime year
             to fraction(5));
```

The following query returns a list of the two timestamps following 1994-01-03 in the calendar **daycal**:

```
execute function CalRange('daycal',
    '1994-01-03 00:00:00.00000', 2);
```

Related Topics

"ts_cal_range()" on page 9-21

"ts_cal_range_index()" on page 9-23

"ts_cal_stamp()" on page 9-25

CalStamp

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

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 timestamp.

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

Description

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

Returns

The timestamp representing the given offset.

Example

The following example returns the timestamp that is two intervals after 1994-01-03:

```
execute function CalStamp('daycal',
    '1994-01-03 00:00:00.00000', 2);
```

Related Topics

"ts_cal_range()" on page 9-21

"ts_cal_range_index()" on page 9-23

"ts_cal_stamp()" on page 9-25

CalStartDate

The **CalStartDate** function takes a calendar name and returns a DATETIME containing the startdate 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 startdate 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 Topics

"ts_cal_startdate()" on page 9-26

"CalPattStartDate" on page 6-6

OrOp

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 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(1994-01-01 00:00:00), pattstart(1994-
             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(1994-04-01 00:00:00), pattstart(1994-
             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(1994-01-01 00:00:00), pattstart(19
                94-01-02 00:00:00), pattern({24 off,120 on,24
                off},hour)
```

Related Topics

"OrOp" on page 6-12

"AndOp" on page 7-4

Chapter

Time Series SQL Routines

8

In This Chapter

Time series SQL routines create instances of a particular time series type as well as add data to or change data in it. SQL routines are also provided to examine, analyze, manipulate, and aggregate the data within a time series.

The several data types and tables are used throughout the examples in this chapter are listed in the following table.

Type/Table	Description	Defined on
stock_bar	Type containing **timestamp**(DATETIME), **high**, **low**, **final**, and **vol** columns	page 4-6
daily_stocks	Table containing **stock_id**, **stock_name**, and **stock_data** columns	page 4-7
stock_trade	Type containing **timestamp**(DATETIME), **price**, **vol**, **trade**, **broker**, **buyer**, and **seller** columns	page 4-6
activity_stocks	Table containing **stock_id** and **activity_data** columns	page 4-7

The schema for these examples is in the **examples** subdirectory of the Informix TimeSeries DataBlade module installation.

Summary of Routines by Task Type

The following table shows the routines included in the Informix TimeSeries DataBlade module, arranged 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 timestamp and an offset	Return the offset given the timestamp	**GetIndex** (regular only)
	Return the timestamp given the offset	**GetStamp** (regular only)
Count the number of elements	Return the number of elements	**GetNelems**
	Get the number of elements between two timestamps	**ClipGetCount**

(1 of 6)

Task Type	Description	Routine Name
Select individual elements	Get the element associated with a given timestamp	**GetElem**
	Get the element at or before a timestamp	**GetLastValid**
	Get the element after a timestamp	**GetNextValid**
	Get the element before a timestamp	**GetPreviousValid**
	Get the element at a specified position	**GetNthElem** (regular only)
	Get the first element	**GetFirstElem**
	Get the last element	**GetLastElem**
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**
	Insert an element	**InsElem**
	Insert a set	**InsSet**
	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**

(2 of 6)

Task Type	Description	Routine Name
Make elements visible or invisible to a scan	Make an element invisible	**HideElem**
	Make an element visible	**RevealElem**
Extract and use part of a time series	Extract a period between two timestamps 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**
	Extract a period that includes a given time	**WithinC**
	Extract a period starting or ending at a given time	**WithinR**
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, or a regular populated time series, or a regular time series with metadata	**TSCreate**
	Create an irregular empty time series, or 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**

(3 of 6)

Task Type	Description	Routine Name
Transpose a time series	Convert time series data to tabular form	**Transpose**
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 SMALLFLOAT or DOUBLE PRECISION values	**TSRunningAvg**
	Compute a running sum over SMALLFLOAT or DOUBLE PRECISION values	**TSRunningSum**
	Compare SMALLFLOAT or DOUBLE PRECISION values	**TSCmp**
	Return a previously saved value	**TSPrevious**
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**

(4 of 6)

Task Type	Description	Routine Name
Perform an arithmetic operation on one or two time series (continued)	Get the modulus or remainder of a division of one time series by another	**Mod**
	Negate a time series	**Negate**
	Return 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**
	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 values in a time series	Aggregate values in a time series	**AggregateBy**

(5 of 6)

Task Type	Description	Routine Name
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**

(6 of 6)

The following routines are only used with regular time series:

- **GetIndex**
- **GetStamp**
- **GetNthElem**
- **Lag**
- **PutNthElem**
- **Sum**
- **TSCreate**

The **TSCreateIrr** function is only used with irregular time series.

Abs

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Acos

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

AggregateBy

The **AggregateBy** function aggregates the values in a time series using groups of values. This function is generally used to convert a time series with a small interval to a time series with a larger interval, for instance, to convert a daily time series to a weekly time series.

Syntax

```
AggregateBy(agg_express   lvarchar,
            cal_name       lvarchar,
            ts            TimeSeries)
returns TimeSeries;
```

agg_express A comma-separated list of these SQL aggregate operators: MIN, MAX, SUM, AVG, FIRST, LAST, or Nth (described below).

cal_name The name of a calendar that defines the aggregation period.

ts The time series to be aggregated.

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 timestamp column plus one column for each expression in the list.

An error is raised if the MIN, MAX, 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, 1994, to a weekly time series. The input calendar, named "days," starts at 12:00 AM, and the output calendar, named "weeks," starts at 12:00 AM, on Monday. The first output time is 00:00 Jan. 3, 1994; it is the aggregation of all input values from the input origin, Jan. 4, 1994, to 23:59:59.99999 Jan. 9, 1994. The second output time is 00:00 Jan. 10, 1994; it is the aggregation of all input values from 00:00 Jan 10, 1994 to 23:59:59.99999 Jan. 16, 1994.

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 may 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.

Returns

The aggregated time series, which is always regular.

Example

The following query aggregates the daily_stock time series to a weekly time series:

```
insert into weekly_stocks( stock_id, stock_name, stock_data)
select stock_id, stock_name,
AggregateBy( 'max($high), min($low),
last($final),sum($vol)',
'weeklycal', stock_data)::TimeSeries(daybar)
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 Topics

"Apply" on page 8-17

Apply

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,
      set_ts       set(TimeSeries))
returns TimeSeries;

Apply(sql_express  lvarchar,
      filter       lvarchar,
      ts           TimeSeries, ...)
returns TimeSeries;

Apply(sql_express lvarchar,
      filter      lvarchar,
      set_ts      set(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),
      set_ts      set(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),
      set_ts      set(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” on page 8-44 for more detail on range specifications.
end_stamp	The end point of the range. See “Clip” on page 8-44 for more detail on 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.
set_ts	A set 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 timestamp column. Do not specify the first timestamp as the first expression; the first timestamp 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 timestamp column, and **$0.1** is the column following the timestamp 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, the **$1.high** is equivalent to **$1.1**.

If **Apply** has only one time series argument, you can refer to 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 the example section on how to achieve this.

The following are examples of valid expressions for **Apply** to apply. Assume two argument time series with the same subtype **daybars(t** DATETIME YEAR TO FRACTION(5)**, high** REAL**, low** REAL**, close** REAL**, vol** REAL**)**. Then 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 daybars, arg2 daybars) returns (one_real)"
```

The syntax for the *filter* argument is similar to the above expression, except that it must evaluate to a single-column Boolean result. Only those elements that evaluate to TRUE are selected. For example:

```
"$0.vol > $1.vol and $0.close > ($0.high - $0.low)/2"
```

Apply with the *set_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 *set_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 *set_ts* argument to **Apply**. The C function can return the **TimeSeries** values in any order you want.
- Use ORDER BY in the *set_ts* expression.

Apply with the *set_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, that is, **Apply** considers all elements.

If both the clip time range and filter expression are given, then clipping is done before the 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” on page 8-161.

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(1994-01-01) year to day,
        datetime(1994-01-06) 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(1994-01-04) year to day,
    datetime(1994-01-05) 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',
    '1994-01-03 00:00:00.00000'::datetime year
        to fraction(5),
    '1994-01-03 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 above 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 Topics

"Clip" on page 8-44

"ClipCount" on page 8-47

"ClipGetCount" on page 8-50

"Intersect" on page 8-95

"TSAddPrevious" on page 8-136

"TSCmp" on page 8-138

"TSDecay" on page 8-151

"TSPrevious" on page 8-153

"TSRunningAvg" on page 8-155

"TSRunningSum" on page 8-157

"Union" on page 8-161

ApplyBinaryTsOp

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 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 time series, a row type, or a number. The second two arguments must be compatible under the function. See "Binary Arithmetic Functions" on page 8-37 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 8-37 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 8-37 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 **Apply-BinaryTSOp('plus', 1.data, h.data)**.

Related Topics

"ApplyOpToTsSet" on page 8-30

"Binary Arithmetic Functions" on page 8-37

ApplyCalendar

The **ApplyCalendar** function applies a new calendar to a time series.

Syntax

```
ApplyCalendar (ts        TimeSeries,
               cal_name  lvarchar)
returns TimeSeries;
```

ts The given time series from which specific timepoints will be projected.

cal_name The name of the calendar to apply.

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.

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

The **ApplyOpToTsSet** function applies a binary arithmetic function to a set of time series.

Syntax

```
ApplyOpToTsSet(func_name lvarchar,
               set_ts    set(TimeSeries))
returns TimeSeries;
```

func_name The name of a binary function. See "Binary Arithmetic Functions" on page 8-37 for more information.

set_ts A set of time series that are compatible with the function. All the time series in the set must have the same type.

Description

All the time series must have the same type. If the set is empty, then **ApplyOpToTsSet** returns NULL. If the set contains only one time series, then **ApplyOpToTsSet** returns a copy of that time series. If the set contains exactly two time series, **ts1** and **ts2**, then **ApplyOpToTsSet** returns **ApplyBinaryTsOp(*func_name*, ts1, ts2)**. If the set 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 set. 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 Topics

"ApplyBinaryTsOp" on page 8-25

"Binary Arithmetic Functions" on page 8-37

ApplyUnaryTsOp

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 8-159 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 Topics

"Unary Arithmetic Functions" on page 8-159

Asin

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Atan

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Atan2

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

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 timestamp, 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 timestamp of the resulting element is ignored; the element placed in the resulting time series has the later of the timestamps 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 timestamps, 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-timestamp 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',
         '1994-01-03 00:00:00.00000'
         ::datetime year to fraction(5),
         '1994-01-10 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',
         '1994-01-03 00:00:00.00000'
         ::datetime year to fraction(5),
         '1994-01-10 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 Topics

"Apply" on page 8-17

"ApplyBinaryTsOp" on page 8-25

"ApplyOpToTsSet" on page 8-30

"Unary Arithmetic Functions" on page 8-159

BulkLoad

The **BulkLoad** function loads data from a client file into an existing time series.

Syntax

```
BulkLoad (ts        TimeSeries,
          filename lvarchar)
returns TimeSeries;
```

ts The time series in which to load data.

filename The path and filename of the file to load.

Description

The file resides on the client and can be an absolute or relative pathname.

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, that is, 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(1994-01-03 00:00:00.00000, 1.1, 2.2)
row(1994-01-04 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:

```
1994-01-03 00:00:00.00000   1.1     2.2
1994-01-04 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 timestamps, the old values are replaced by the new values.

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

The **Clip** function extracts data between two timepoints in a time series, then creates and returns a new time series containing that data. This allows you to extract periods of interest from a large time series and 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)
  [, flag         integer])
returns TimeSeries;

Clip(ts           TimeSeries,
     begin_stamp  datetime year to fraction(5),
     end_offset   integer
  [, flag         integer])
returns TimeSeries;

Clip(ts           TimeSeries,
     begin_offset integer,
     end_stamp    datetime year to fraction(5)
  [, flag         integer])
returns TimeSeries;

Clip(ts           TimeSeries,
     begin_offset integer,
     end_offset   integer
  [, flag         integer])
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).
flag (optional)	An optional flag specifying how to determine the resulting time series origin. The possible values are 0 (not set) and 1.

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 timestamps. If the begin point is an integer, it is the absolute offset of an entry in the time series. If it is a timestamp, the **Clip** function uses the time series' calendar to find the offset that corresponds to the timestamp. If there is no entry in the time series exactly at the requested timestamp, **Clip** uses the calendar's timestamp that immediately follows the given timestamp 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 timestamps are permitted for the begin and end points.

Data at the beginning and ending offsets is included in the resulting time series.

If the *flag* argument is not set (has the default value of 0), then the origin of the resulting time series is the later of the begin point argument and the origin of the input time series. If the flag is set (has a value of 1), then the origin of the resulting time series is set to the earlier of the begin point argument and the origin of the input time series. In this case, timepoints before the origin of the time series are set to NULL.

Returns

A new time series value 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 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,
        '1994-01-03 00:00:00.00000'
         ::datetime year to fraction(5),
        '1994-01-07 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';
```

Related Topics

"Apply" on page 8-17

"ClipCount" on page 8-47

"ClipGetCount" on page 8-50

"GetElem" on page 8-62

"GetLastValid" on page 8-71

"GetNthElem" on page 8-79

"WithinC, WithinR" on page 8-171

ClipCount

The **ClipCount** function is a variation of **Clip** in which the first integer argument is interpreted as a count. If the count is positive, **ClipCount** begins with the first element at or after the timestamp and clips the next count entries. If the count is negative, **ClipCount** begins with the first element at or before the timestamp and clips the previous count entries.

Syntax

```
ClipCount(ts          TimeSeries,
          begin_stamp datetime year to fraction(5),
          num_stamps  integer
        [, flag        integer])
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 at or after (at or before if the integer is negative) the begin point to be included in the resultant time series.

flag (optional) An optional flag specifying how to determine the resulting time series origin. The possible values are 0 (not set) and 1.

Description

Begin points prior to the time series origin are permitted. Negative counts with such timestamps result in time series with no elements. Begin points prior to the calendar origin are not permitted.

If there is no entry in the calendar exactly at the requested timestamp, **ClipCount** uses the calendar's first valid timestamp that immediately follows the given timestamp as the begin point of the range. If the begin point is null, the origin of the time series is used.

If the *flag* argument is not set (has the default value of 0), then the origin of the resulting time series is the later of the begin point argument and the origin of the input time series. If the flag is set (has a value of 1), then the origin of the resulting time series is set to the earlier of the begin point argument and the origin of the input time series. In this case, timepoints before the origin of the time series are set to NULL.

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, 1994, at 9:30 A.M. for the stock with ID 600, and returns the entire resulting time series:

```
select ClipCount(activity_data,
      '1994-01-01 09:30:00.00000', 30)
    from activity_stocks
    where stock_id = 600;
```

The following example clips the previous 60 elements at or prior to August 22, 1994, at 12:00 midnight for the stock with ID 600:

```
select ClipCount(activity_data,
        '1994-08-22 00:00:00.00000', -60)
    from activity_stocks
    where stock_id = 600;
```

Related Topics

"Clip" on page 8-44

"ClipGetCount" on page 8-50

"GetElem" on page 8-62

"GetLastValid" on page 8-71

"GetNthElem" on page 8-79

ClipGetCount

The **ClipGetCount** function returns the number of elements in the current time series that occur in the time period delimited by the timestamps.

Syntax

```
ClipGetCount(ts          TimeSeries,
             begin_stamp datetime year to fraction(5),
             end_stamp   datetime year to fraction(5))
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.

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" on page 8-44 for more information on the beginning and ending points of the range.

Returns

The number of elements in the given time series that occur in the period delimited by the timestamps.

Example

The following statement returns the number of elements between 10:30 A.M. on March 14, 1994, and midnight on March 19, 1994, inclusive:

```
select ClipGetCount(activity_data,
    '1994-03-14 10:30:00.00000','1994-03-19 00:00:00.00000')
    from activity_stocks
    where stock_id = 600;
```

Related Topics

"Clip" on page 8-44

"GetIndex" on page 8-65

"GetNelems" on page 8-75

"GetNthElem" on page 8-79

"GetStamp" on page 8-85

"ts_nelems()" on page 9-85

Cos

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

DelClip

The **DelClip** function deletes all elements in the specified time range, including the delimiting timepoints.

Syntax

```
DelClip(ts          TimeSeries,
        begin_stamp datetime year to fraction(5),
        end_stamp   datetime year to fraction(5))
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.

Description

You can use **DelClip** to delete hidden elements.

If the begin or end point of the range falls outside the begin and end points of the calendar, 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 in a six-hour range on a given day:

```
update activity_stocks
set activity_data = DelClip(activity_data,
        '1994-01-05 00:00:00.00000'
      ::datetime year to fraction(5),
        '1994-01-06 00:00:00.00000'
      ::datetime year to fraction(5))
where stock_id = 600;
```

Related Topics

"Clip" on page 8-44

"DelElem" on page 8-55

"HideElem" on page 8-88

"InsSet" on page 8-92

"PutSet" on page 8-115

"UpdSet" on page 8-169

DelElem

The **DelElem** function deletes the element at a given timepoint.

Syntax

```
DelElem(ts     TimeSeries,
        tstamp datetime year to fraction(5))
returns TimeSeries;
```

ts The time series to act on.

tstamp The timestamp of the element to be deleted.

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 timestamps 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,
    '1994-01-05 12:58:09.23456'
    ::datetime year to fraction(5))
where stock_id = 600;
```

Related Topics

"DelClip" on page 8-53

"GetElem" on page 8-62

"HideElem" on page 8-88

"InsElem" on page 8-90

"PutElem" on page 8-109

"ts_del_elem()" on page 9-41

Divide

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

Exp

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

GetCalendar

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(1994-01-01 00:00:00),pattstart(1994-
             01-02 00:00:00),pattern({1 off,5 on,1 off},day)
```

Related Topics

"GetCalendarName" on page 8-60

"GetInterval" on page 8-67

"GetOrigin" on page 8-81

"TSCreate" on page 8-143

GetCalendarName

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 Topics

"GetCalendar" on page 8-59

"GetInterval" on page 8-67

"GetOrigin" on page 8-81

"TSCreate" on page 8-143

GetContainerName

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 does not reside 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 Topics

"ts_get_containername()" on page 9-58

GetElem

The **GetElem** function extracts the element for the given timestamp.

Syntax

```
GetElem(ts     TimeSeries,
        tstamp datetime year to fraction(5))
returns row;
```

ts The source time series.

tstamp The timestamp of the entry.

Description

If the timestamp 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 timestamp. For example, if the time series is set to hourly, 8:00 A.M. to 5:00 P.M., the timestamp 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 timestamp and the data from the time series at that timestamp. 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 particular day:

```
select GetElem(stock_data,'1994-01-04 00:00:00.00000')
    from daily_stocks
    where stock_name = 'IBM' or stock_name = 'HWP';
```

Related Topics

"DelElem" on page 8-55

"GetLastElem" on page 8-69

"GetLastValid" on page 8-71

"GetNextValid" on page 8-77

"GetNthElem" on page 8-79

"GetPreviousValid" on page 8-83

"InsElem" on page 8-90

"PutElem" on page 8-109

"Transpose" on page 8-133

"ts_elem()" on page 9-42

GetFirstElem

The **GetFirstElem** function returns the first element in a time series.

Syntax

```
GetFirstElem(ts TimeSeries)
returns row;
```

ts The source time series.

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 Topics

"GetLastElem" on page 8-69

"ts_first_elem()" on page 9-50

GetIndex

The **GetIndex** function returns the index (offset) of the regular time series entry associated with the supplied timestamp.

Syntax

```
GetIndex(ts     TimeSeries,
         tstamp datetime year to fraction(5))
returns integer;
```

ts The source time series.

tstamp The timestamp of the entry.

Description

The data extracted is associated with the time period that the timestamp is in. For example, if you have a time series set to hourly, 8:00 A.M. to 5:00 P.M., the timestamp 3:15 P.M. would return the index associated with 3:00 P.M.

The API equivalent of this function is **ts_index()**.

An error is raised if **GetIndex** is used with irregular time series.

Returns

The integer offset of the entry for the given timestamp in the time series.

NULL is returned if the timestamp 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 timestamp:

```
select stock_name, GetIndex(stock_data,
        '1994-01-05 00:00:00.00000')
 from daily_stocks;
```

Related Topics

"CalIndex" on page 7-6

"CalRange" on page 7-8

"GetElem" on page 8-62

"GetNelems" on page 8-75

"GetNthElem" on page 8-79

"GetStamp" on page 8-85

"ts_index()" on page 9-70

GetInterval

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 Topics

"The CalendarPattern Data Type" on page 3-4

"GetCalendar" on page 8-59

"GetCalendarName" on page 8-60

"GetOrigin" on page 8-81

"TSCreate" on page 8-143

GetLastElem

The **GetLastElem** function returns the final entry stored in a time series.

Syntax

```
GetLastElem(ts TimeSeries)
returns row;
```

ts The source time series.

Description

The API equivalent of this function is **ts_last_elem()**.

Returns

A row type value containing the time series data and timestamp 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 Topics

"GetElem" on page 8-62

"GetFirstElem" on page 8-64

"GetLastValid" on page 8-71

"GetNthElem" on page 8-79

"PutElem" on page 8-109

"ts_last_elem()" on page 9-76

GetLastValid

The **GetLastValid** function extracts the element for the given timestamp in a time series.

Syntax

```
GetLastValid(ts    TimeSeries,
             tstamp datetime year to fraction(5))
returns row;
```

ts The source time series.

tstamp The timestamp for the element.

Description

For regular time series, this function returns the element at the calendar's latest valid timepoint at or before the given timestamp. For irregular time series, it returns the latest element at or preceding the given timestamp.

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

Returns

A row type containing the nearest element at or before the given timestamp. The type of the row is the same as the time series subtype.

If the timestamp 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 timestamp:

```
select GetLastValid(stock_data, '1994-01-08 00:00:00.00000')
from daily_stocks
where stock_name = 'IBM';
```

Related Topics

"GetElem" on page 8-62

"GetLastElem" on page 8-69

"GetNextValid" on page 8-77

"GetNthElem" on page 8-79

"GetPreviousValid" on page 8-83

"PutElem" on page 8-109

"ts_last_valid()" on page 9-78

GetMetaData

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 Topics

"GetMetaTypeName" on page 8-74

"TSCreate" on page 8-143

"TSCreateIrr" on page 8-147

"ts_create_with_metadata()" on page 9-36

"ts_get_metadata()" on page 9-60

"ts_update_metadata" on page 9-106

"UpdMetaData" on page 8-167

GetMetaTypeName

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 Topics

"GetMetaData" on page 8-73

"TSCreate" on page 8-143

"TSCreateIrr" on page 8-147

"ts_create_with_metadata()" on page 9-36

"ts_get_metadata()" on page 9-60

"UpdMetaData" on page 8-167

GetNelems

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 Topics

"ClipGetCount" on page 8-50

"GetIndex" on page 8-65

"GetNthElem" on page 8-79

"GetStamp" on page 8-85

"ts_nelems()" on page 9-85

GetNextValid

The **GetNextValid** function returns the nearest entry after a given timestamp.

Syntax

```
GetNextValid(ts     TimeSeries,
             tstamp datetime year to fraction(5))
returns row;
```

ts	The source time series.
tstamp	The timestamp of the entry.

Description

For regular time series, **GetNextValid** returns the element at the calendar's earliest valid timepoint following the given timestamp. For irregular time series, it returns the earliest element following the given timestamp.

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

Returns

A row type containing the nearest element after the given timestamp. The type of the row is the same as the time series subtype.

NULL is returned if the timestamp is later than that of the last timestamp in the time series.

Example

The following example gets the first element that follows timestamp 1994-01-03 in a regular time series:

```
select GetNextValid(stock_data,'1994-01-03 00:00:00.00000')
    from daily_stocks
    where stock_name = 'IBM';
```

The following example gets the first element that follows timestamp 1994-01-03 in an irregular time series:

```
select GetNextValid(activity_data,
    '1994-01-03 00:00:00.00000')
    from activity_stocks
    where stock_id = 600;
```

Related Topics

"GetElem" on page 8-62

"GetLastValid" on page 8-71

"GetNthElem" on page 8-79

"GetPreviousValid" on page 8-83

"ts_next_valid()" on page 9-88

GetNthElem

The **GetNthElem** function extracts the entry at a particular offset in a regular time series.

Syntax

```
GetNthElem(ts TimeSeries,
           N integer)
returns row;
```

ts The source time series.

N The offset of an entry in the time series. Must be greater than or equal to 0.

Description

An error is raised if **GetNthElem** is used with irregular time series or if the integer in the argument is less than 0.

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 timestamp 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 timestamp (this could also be done with **GetElem**):

```
select GetNthElem(stock_data,GetIndex(stock_data,
               '1994-01-04 00:00:00.00000'))
    from daily_stocks
    where stock_name = 'IBM';
```

Related Topics

"GetElem" on page 8-62

"GetLastElem" on page 8-69

"GetLastValid" on page 8-71

"GetNextValid" on page 8-77

"GetPreviousValid" on page 8-83

"PutElem" on page 8-109

"Transpose" on page 8-133

"ts_nth_elem()" on page 9-90

GetOrigin

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 can be changed with **SetOrigin**.

Returns

The time series origin.

Example

The following example returns the timestamp of the origin of the time series for a given stock:

```
select GetOrigin(stock_data)
from daily_stocks
where stock_name = 'IBM';
```

Related Topics

"GetCalendar" on page 8-59

"GetInterval" on page 8-67

"GetCalendarName" on page 8-60

"TSCreate" on page 8-143

GetPreviousValid

The **GetPreviousValid** function returns the last element before the given timestamp.

Syntax

```
GetPreviousValid(ts     TimeSeries,
                 tstamp datetime year to fraction(5))
returns row;
```

ts The source time series.

tstamp The timestamp of interest.

Description

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

Returns

A row containing the last element before the given timestamp. The type of the row is the same as the time series subtype.

If the timestamp is less than or equal to the time series origin, NULL is returned.

Example

The following query gets the first element that precedes timestamp 1994-01-05 in a regular time series:

```
select GetPreviousValid(stock_data,
    '1994-01-05 00:00:00.00000')
    from daily_stocks
    where stock_name = 'IBM';
```

The following query gets the first element that precedes timestamp 1994-01-05 in an irregular time series:

```
select GetPreviousValid(activity_data,
    '1994-01-05 00:00:00.00000')
    from activity_stocks
    where stock_id = 600;
```

Related Topics

"GetElem" on page 8-62

"GetLastValid" on page 8-71

"GetNextValid" on page 8-77

"ts_previous_valid()" on page 9-93

GetStamp

The **GetStamp** function returns the timestamp associated with the supplied offset in a regular time series. Offsets begins at 0.

Syntax

```
GetStamp(ts     TimeSeries,
         offset integer)
returns datetime year to fraction(5);
```

ts	The source time series.
offset	The offset. Must be greater than or equal to 0.

Description

An error is raised if **GetStamp** is used with irregular time series or if the integer is less than 0.

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

Returns

The timestamp that begins the interval at the specified offset.

Example

The following query returns the timestamp of the beginning of a time series:

```
select GetStamp(stock_data,0)
    from daily_stocks
    where stock_name = 'IBM';
```

Related Topics

"CalIndex" on page 7-6

"CalRange" on page 7-8

"GetElem" on page 8-62

"GetIndex" on page 8-65

"GetNelems" on page 8-75

"GetNthElem" on page 8-79

"ts_time()" on page 9-105

GetThreshold

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 Topics

"ts_get_threshold()" on page 9-65

HideElem

The **HideElem** function marks an element, or a set of elements, at a given timestamp as invisible.

Syntax

```
HideElem(ts     TimeSeries,
          tstamp datetime year to fraction(5))
returns TimeSeries;

HideElem(ts TimeSeries,
        set_tstamps set(datetime year to fraction(5) not null))
returns TimeSeries;
```

ts	The source time series.
tstamp	The timestamp to be made invisible.
set_tstamps	The set of timestamps to be made invisible.

Description

Once 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 timestamp is not a valid timepoint in the time series' calendar, an error is raised.

Returns

The modified time series.

Example

The following example hides the element at 1994-01-03 in the time series for IBM:

```
select HideElem(stock_data, '1994-01-03 00:00:00.00000')
    from daily_stocks
    where stock_name = 'IBM';
```

Related Topics

"RevealElem" on page 8-119

"ts_begin_scan()" on page 9-16

"ts_hide_elem()" on page 9-68

"ts_reveal_elem()" on page 9-103

InsElem

The **InsElem** function inserts an element into a time series.

Syntax

```
InsElem(ts         TimeSeries,
        row_value  row)
returns TimeSeries;
```

ts The time series to act on.

row_value The row type value to be added to the time series.

Description

The element must be a row type of the correct type for the time series, beginning with a valid timestamp. If there is already an element with that timestamp in the time series, the insertion is void, and an error is raised. Once 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 timestamp 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('1994-10-06 08:06:56.00000', 6.50, 2000,
            1, 007, 3, 1)::stock_trade)
where stock_id = 600;
```

Related Topics

"DelElem" on page 8-55

"GetElem" on page 8-62

"InsSet" on page 8-92

"PutElem" on page 8-109

"ts_ins_elem()" on page 9-72

InsSet

The **InsSet** function inserts every element of a given set into a time series.

Syntax

```
InsSet(ts       TimeSeries,
       set_rows set)
returns TimeSeries;
```

ts	The time series to act on.
set_rows	The set of new row type values to store in the time series.

Description

The supplied row type values must have a timestamp as their first attribute. This timestamp 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 timestamp 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 timestamp that has been hidden.

Returns

The time series with the set 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 Topics

"DelClip" on page 8-53

"InsElem" on page 8-90

"PutSet" on page 8-115

"UpdSet" on page 8-169

InstanceId

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

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 timestamps. 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 element remain in the proper 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 timestamp 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 timestamps between 1994-01-03 and 1994-01-05:

```
select Intersect('1994-01-03 00:00:00.00000'
                ::datetime year to fraction(5),
                '1994-01-05 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 Topics

"Apply" on page 8-17

"Union" on page 8-161

IsRegular

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);
```

Lag

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

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Minus

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

Mod

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

Negate

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Plus

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

Positive

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Pow

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

PutElem

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)
returns TimeSeries;
```

ts The time series to act on.

row_value The new row type value to store in the time series.

Description

If the timestamp 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 timestamp up to the next second.
2. Search backwards for the first element less than the new timestamp.
3. Insert the new data at this timestamp 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('1994-08-25 09:06:00.00000',
        6.25, 1000, 1, 007, 2, 1)::stock_trade)
    where stock_id = 600;
```

Related Topics

"DelElem" on page 8-55

"GetElem" on page 8-62

"InsElem" on page 8-90

"PutElemNoDups" on page 8-111

"PutSet" on page 8-115

"TSCreate" on page 8-143

"ts_put_elem()" on page 9-95

PutElemNoDups

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)
returns TimeSeries;
```

ts The time series to act on.

row_value The new row type value to store in the time series.

Description

If the timestamp 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('1994-08-25 09:06:00.00000', 6.25,
        1000, 1, 007, 2, 1)::stock_trade)
    where stock_id = 600;
```

Related Topics

"PutElem" on page 8-109

"ts_put_elem_no_dups()" on page 9-97

PutNthElem

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)
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.

Description

This function is similar to **PutElem**, except **PutNthElem** takes an offset instead of a timestamp.

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 Topics

"PutElem" on page 8-109

PutSet

The **PutSet** function updates a time series with the supplied set of row type values.

Syntax

```
PutSet(ts     TimeSeries,
       set_ts set)
returns TimeSeries;
```

ts	The time series to act on.
set_ts	The set of new row type values to store in the time series.

Description

For each element in the set of rows, if the timestamp 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, then the following algorithm is used to determine where to place the data:

1. Round the timestamp up to the next second.
2. Search backward for the first element less than the new timestamp.
3. Insert the new data at this timestamp plus 10 microseconds.

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 updates a time series with a set:

```
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 Topics

"DelClip" on page 8-53

"InsSet" on page 8-92

"PutElem" on page 8-109

"TSCreate" on page 8-143

"UpdSet" on page 8-169

PutTimeSeries

The **PutTimeSeries** function puts every element, except hidden elements, of the first time series into the second time series.

Syntax

```
PutTimeSeries(ts1 TimeSeries,
              ts2 TimeSeries)
returns TimeSeries;
```

ts1 The time series to be inserted.

ts2 The time series into which the first time series is to be inserted.

Description

If both time series contain data at the same timepoint, the rule of **PutElem** is followed (see “PutElem” on page 8-109).

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 requires aggregation information, which can be provided using the* ***AggregateBy*** *function.*

Elements are added to the second time series by calling **ts_put_elem()**.

The API equivalent of this function is **ts_put_ts()**.

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 Topics

"AggregateBy" on page 8-14

"PutSet" on page 8-115

"ts_put_ts()" on page 9-101

RevealElem

The **RevealElem** function makes an element at a given timestamp 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 set(datetime year to fraction(5)))
returns TimeSeries;
```

ts	The time series to act on.
tstamp	The timestamp to be made visible to a scan.
set_stamps	The set of timestamps to be made visible to a scan.

Returns

The modified time series.

Example

The following example hides the element at 1994-01-03 in the IBM time series and then reveals it:

```
select HideElem(stock_data, '1994-01-03 00:00:00.00000')
    from daily_stocks
    where stock_name = 'IBM';

select RevealElem(stock_data, '1994-01-03 00:00:00.00000')
    from daily_stocks
    where stock_name = 'IBM';
```

Related Topics

"HideElem" on page 8-88

Round

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**.

See “Unary Arithmetic Functions” on page 8-159 for more information.

SetContainerName

The **SetContainerName** function sets the container name for a time series.

Syntax

```
SetContainerName(ts              TimeSeries,
                 container_name varchar(18,1))
returns TimeSeries;
```

ts	The time series to act on.
container_name	The name of the container.

Description

This function is needed when a function (such as **Apply**) returns a time series whose elements differ from the source time series. When this occurs, the returned time series cannot use the source time series' container. Therefore, if you want the returned time series to be inserted into a table, you should use **SetContainerName** to assign it a container.

Returns

The original time series with the new container.

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 = 'HWP';
```

Related Topics

"TSContainerCreate" on page 8-140

SetOrigin

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,
        '1994-01-02 00:00:00.00000');
```

Related Topics

"Apply" on page 8-17

"Clip" on page 8-44

"GetOrigin" on page 8-81

"PutTimeSeries" on page 8-117

Sin

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Sqrt

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Sum

The **Sum** function is the sum aggregate function for the **TimeSeries** data type for regular time series.

Syntax

```
Sum(ts TimeSeries)
returns TimeSeries;
```

ts The time series to act on.

Description

All time series summed must have the same calendar. They need not 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.

The **Sum** function is for regular time series only; using it on irregular time series raises an error.

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,'1994-01-04 00:00:00.00000')
        from daily_stocks;

select Apply('$final * $vol',
            '1994-01-04 00:00:00.00000'
            ::datetime year to fraction(5),
            '1994-12-30 00:00:00.00000'
            ::datetime year to fraction(5),
            stock_data)::TimeSeries(one_real)
from daily_stocks;
```

Tan

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**.

See "Unary Arithmetic Functions" on page 8-159 for more information.

Times

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**.

See "Binary Arithmetic Functions" on page 8-37 for more information.

TimeSeriesRelease

The **TimeSeriesRelease** function returns an lvarchar string containing the Informix TimeSeries DataBlade module version number and build date.

Syntax

```
TimeSeriesRelease()
returns lvarchar;
```

Returns

The version number and build date of the Informix TimeSeries DataBlade module.

Example

The following example shows how to get the Informix TimeSeries DataBlade module version using DB-Access:

```
execute function TimeSeriesRelease();
```

Related Topics

None.

Transpose

The **Transpose** function converts time series data for processing in a tabular format.

Syntax

```
Transpose (ts TimeSeries)
returns row;

Transpose (ts          TimeSeries,
           begin_stamp datetime year to fraction(5),
           end_stamp   datetime year to fraction(5))
returns row;

Transpose (ts          TimeSeries,
           begin_stamp datetime year to fraction(5),
           end_stamp   datetime year to fraction(5),
           flags       integer)
returns row;
```

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. The *flags* values are described in the section "The flags Argument Values," next.

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" on page 8-88).
TS_SCAN_EXACT_START	256	Return the element at the beginning timepoint, adding null elements if necessary.
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.

Description

The **Transpose** function is an iterator function and must therefore be used with the EXECUTE FUNCTION statement. See the *Informix Guide to SQL: Syntax* for more information on the EXECUTE FUNCTION statement.

If the beginning point is null, the scan starts at the first element of the time series. If the end point is null, the scan ends at the last element of the time series.

Returns

Multiple rows containing a timestamp and the other columns of the time series elements.

Example

The following example converts the data from **stock_bar** for IBM to a tabular form:

```
execute function Transpose((select stock_data
    from daily_stocks where stock_name = 'IBM'));
```

The following example shows transposed data for a clipped range:

```
execute function Transpose((select stock_data from
daily_stocks
                where stock_name = 'IBM'),
                datetime(1994-01-05) year to day,
                NULL::datetime year to fraction(5));


(expression)    ROW('1994-01-06 00:00:00.00000',99.00000
                000000,54.00000000000,66.00000000000,888
                .0000000000)
```

Related Topics

"GetElem" on page 8-62

"GetNthElem" on page 8-79

TSAddPrevious

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 money flow, 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))',
        '1994-01-03 00:00:00.00000'::datetime year to
fraction(5),
        '1994-01-08 00:00:00.00000'::datetime year to
fraction(5),
        stock_data)::TimeSeries(one_real)
from daily_stocks
where stock_name = 'IBM';
```

Related Topics

"Apply" on page 8-17

"TSCmp" on page 8-138

"TSDecay" on page 8-151

"TSPrevious" on page 8-153

"TSRunningAvg" on page 8-155

"TSRunningSum" on page 8-157

TSCmp

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 an **AggregateBy** or **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)',
    '1994-01-03 00:00:00.00000'::datetime year to
fraction(5),
    '1994-01-08 00:00:00.00000'::datetime year to
fraction(5),
    stock_data)::TimeSeries(one_real)
from daily_stocks
where stock_name = 'IBM';
```

Related Topics

"Apply" on page 8-17

"TSAddPrevious" on page 8-136

"TSDecay" on page 8-151

"TSPrevious" on page 8-153

"TSRunningAvg" on page 8-155

"TSRunningSum" on page 8-157

TSContainerCreate

The **TSContainerCreate** procedure creates a new container with the specified name for time series storage.

Syntax

```
TSContainerCreate(container_name  varchar(18,1),
                  dbspace_name    varchar(18,1),
                  ts_type         varchar(18,1),
                  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 type of the time series that will be placed in the container. This argument must be the name of an existing row type that begins with a timestamp.

container_size The initial size of the container, in kilobytes. If this argument is 0 or less, then a default size of 16 KB is used. If this parameter is positive, it must be at least four pages.

container_grow The increments by which the container grows, in kilobytes. If this argument is 0 or less, then a default size of 16 KB is used. If this parameter is positive, it must be at least four pages.

Description

As a result of **TSContainerCreate**, the Informix TimeSeries DataBlade module creates a container when the first time series is inserted into it. Both regular and irregular time series are stored in containers when they exceed a specified size, which is specified at time series creation.

A row is also inserted in the **TSContainerTable** table, as described in the section "The TSInstanceTable Table" on page 3-10.

Only users with update privileges on the **TSContainerTable** table can execute this procedure.

Returns

None.

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 Topics

"TSContainerDestroy" on page 8-142

"The TSContainerTable Table" on page 3-11

"The TSInstanceTable Table" on page 3-10

TSContainerDestroy

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(18,1));
```

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.

Returns

None.

Example

The following example destroys the container **ctnr_stock**:

```
execute procedure TSContainerDestroy('ctnr_stock');
```

Related Topics

"TSContainerCreate" on page 8-140

"The TSContainerTable Table" on page 3-11

"The TSInstanceTable Table" on page 3-10

TSCreate

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. Default is 20. The size of a row containing an in-row time series should not exceed 1500 bytes.
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 Containing Metadata" on page 4-10 for more information on 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 an Empty Time Series" on page 4-9 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',
        '1994-01-03 00:00:00.00000',20,0,0, NULL));
```

The following example creates a populated regular time series using **TSCreate**:

```
select TSCreate('daycal',
        '1994-01-05 00:00:00.00000',
        20,
        0,
        NULL,
        set_data)::TimeSeries(stock_trade)
    from activity_load_tab
    where stock_id = 600;
```

Related Topics

"GetCalendar" on page 8-59

"GetCalendarName" on page 8-60

"GetInterval" on page 8-67

"GetOrigin" on page 8-81

"TSCreateIrr" on page 8-147

TSCreateIrr

The **TSCreateIrr** function creates an empty irregular time series or an irregular time series populated with the given set 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,
            set_rows       set)
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,
            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. Default is 20. The size of a row containing an in-row time series should not exceed 1500 bytes.
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 Containing Metadata" on page 4-10 for more information on metadata.
set_rows	A set 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 an Empty Time Series" on page 4-9 for a description of how to use this function.

Returns

An irregular time series that is empty or populated with the given set and optionally contains user-defined metadata.

Example

The following example creates an empty irregular time series using **TSCreateIrr**:

```
select TSCreateIrr('daycal',
        '1994-01-05 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',
          '1994-01-03 00:00:00.00000'::datetime year to
fraction(5),
          20, 0, NULL,
          set_data)::timeseries(stock_trade)
    from activity_load_tab;
```

Related Topics

"GetCalendar" on page 8-59

"GetCalendarName" on page 8-60

"GetInterval" on page 8-67

"GetOrigin" on page 8-81

"TSCreate" on page 8-143

TSDecay

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 below).
initial_value	The initial value (`initial` in the sum below).
decay_factor	The decay factor (`decay` in the sum below).

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:

$$((\mathtt{decay}^{i})\mathtt{initial}) + \sum_{j=1}^{i} ((v_j)\mathtt{decay}^{i-j})$$

where `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 an **AggregateBy** or **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 Topics

"Apply" on page 8-17

"TSAddPrevious" on page 8-136

"TSCmp" on page 8-138

"TSPrevious" on page 8-153

"TSRunningAvg" on page 8-155

"TSRunningSum" on page 8-157

TSPrevious

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 datum in a time series with the immediately preceding value in the same series.

This function is useful only when used within an **AggregateBy** or **Apply** function.

Returns

The value previously saved. The first time **TSPrevious** is called, it returns NULL.

Example

See the example for "TSCmp" on page 8-138.

Related Topics

"Apply" on page 8-17

"TSAddPrevious" on page 8-136

"TSCmp" on page 8-138

"TSDecay" on page 8-151

"TSRunningAvg" on page 8-155

"TSRunningSum" on page 8-157

TSRunningAvg

The **TSRunningAvg** function computes a running average over SMALLFLOAT or DOUBLE PRECISION values.

Syntax

```
TSRunningAvg(value      smallfloat,
             num_values integer)
returns smallfloat;

TSRunningAvg(value      double precision,
             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 an **AggregateBy** or **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)',
    '1994-01-01 00:00:00.00000'::datetime year to
fraction(5),
    '1995-01-01 00:00:00.00000'::datetime year to
fraction(5),
    stock_data::TimeSeries(stock_bar))::
        TimeSeries(one_real)
from daily_stocks;
```

Related Topics

"Apply" on page 8-17

"TSAddPrevious" on page 8-136

"TSCmp" on page 8-138

"TSDecay" on page 8-151

"TSPrevious" on page 8-153

"TSRunningSum" on page 8-157

TSRunningSum

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 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 an **AggregateBy** or **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 Topics

"Apply" on page 8-17

"TSAddPrevious" on page 8-136

"TSCmp" on page 8-138

"TSDecay" on page 8-151

"TSPrevious" on page 8-153

"TSRunningAvg" on page 8-155

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 timestamps 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-timestamp 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 Topics

"Apply" on page 8-17

"ApplyUnaryTsOp" on page 8-32

"Binary Arithmetic Functions" on page 8-37

Union

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 proper 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 timestamp.

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 timestamps between 1994-01-03 and 1994-01-05:

```
select Union('1994-01-03 00:00:00.00000'
           ::datetime year to fraction(5),
           '1994-01-05 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 Topics

"Apply" on page 8-17

"Intersect" on page 8-95

UpdElem

The **UpdElem** function updates an existing element in a time series.

Syntax

```
UpdElem(ts         TimeSeries,
        row_value  row)
returns TimeSeries;
```

ts	The time series to update.
row_value	The new row data.

Description

The element must be a row type of the correct type for the time series, beginning with a timestamp. If there is no element in the time series with the given timestamp, 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('1994-01-04 12:58:09.12345', 6.75, 2000,
    2, 007, 3, 1)::stock_trade)
where stock_id = 600;
```

Related Topics

"DelElem" on page 8-55

"GetElem" on page 8-62

"InsElem" on page 8-90

"PutElem" on page 8-109

"ts_upd_elem()" on page 9-108

"UpdSet" on page 8-169

UpdMetaData

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 Topics

"GetMetaData" on page 8-73

"GetMetaTypeName" on page 8-74

"TSCreate" on page 8-143

"TSCreateIrr" on page 8-147

"ts_create_with_metadata()" on page 9-36

"ts_get_metadata()" on page 9-60

"ts_update_metadata" on page 9-106

UpdSet

The **UpdSet** function updates a set of existing elements in a time series.

Syntax

```
UpdSet(ts     TimeSeries,
       set_ts multiset)
returns TimeSeries;
```

ts The time series to update.

set_ts A set of rows that replace existing elements in the given time series, *ts*.

Description

The rows in *set_ts* must be of the correct type for the time series, beginning with a timestamp; otherwise, an error is raised. If the timestamp 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 Topics

"DelClip" on page 8-53

"InsSet" on page 8-92

"PutSet" on page 8-115

"UpdElem" on page 8-165

WithinC, WithinR

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, 1993," or "Show me the values in this hourly series for the week including December 27, 1991."

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 timestamp 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 timestamp and find the corresponding period (or periods) that include it.

For example, July 2, 1992, fell on a Thursday. 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 timestamp 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, 1994. **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 timestamp 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 timestamp 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 timestamp specified. Also, since irregular elements can appear at any point within the calendar time interval, **WithinR** returns elements with timestamps 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, '1994-07-11 07:37:18', 'day', 3,
'future')
[1994-07-11 07:37:18, 1994-07-14 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:

```
[1994-07-11 00:00:00, 1994-07-13 00:00:00]
```

Both functions take a time series, a timestamp, an interval name, a number of intervals, and a direction.

The supplied interval name need not 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 timestamp 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, 1994:

```
select WithinC(stock_data, '1994-01-04 00:00:00.00000',
        'week', 1, 'PAST')
    from daily_stocks
    where stock_name = 'IBM';
```

```
(expression)    origin(1994-01-03 00:00:00.00000),calend
                ar(daycal), container(),threshold(20),re
                gular,[(356.0000000000,310.0000000000,34
                0.000000000, 999.0000000000),(156.000000
                0000,110.0000000000,140.0000000000,111.0
                000000000), NULL, (99.00000000000,54.000
                00000000,66.00000000000, 888.0000000000)]
```

The following query returns two weeks' worth of stock trades starting on January 4, 1994, at 9:30 A.M.:

```
select WithinR(activity_data, '1994-01-04 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, '1994-02-01 00:00:00.00000',
        'month', 3, 'past')
    from activity_stocks
    where stock_id = 600;
```

Related Topics

"Clip" on page 8-44

Chapter

Time Series API Routines

9

In This Chapter

This chapter describes the Informix TimeSeries DataBlade module API routines.

Introducing the Time Series API Routines

The Informix TimeSeries DataBlade module 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 timestamp 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.

UNIX

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 Informix TimeSeries DataBlade module installation. ♦

Windows NT

On NT, 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 NT, **tsfeapi.lib**, **tsbeapi.lib**, and **tseries.h** are all in the **lib** directory in the Informix TimeSeries DataBlade module installation. ♦

Important: *Since values returned by **mi_value** are valid only until the next **mi_next_row** or **mi_query_finish**, it may be necessary to put time series in save sets, or to use **ts_copy** to access time series outside an **mi_get_results** loop.*

Differences Between Using Functions on the Server (tsbeapi) and on the Client (tsfeapi)

This section describes some important points to consider when you choose between using the client (**tsfeapi**) version of the time series API and the server (**tsbeapi**) version of the time series API.

First, the client and server interfaces do not behave in exactly the same way when updating a time series. This is due to the fact that **tsbeapi** operates directly on a time series, whereas **tsfeapi** operates on a private copy of a time series. This means that updates via **tsbeapi** are always reflected in the database while updates via **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.

Finally, 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 needs to 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 all at once. Depending on the size of the time series and the memory available, this may 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 may overwhelm the server, whereas if each client does their portion of the work, the load may be better balanced.

API Data Structures

The Informix TimeSeries DataBlade module includes the following data structures for its API routines:

- **ts_timeseries**
- **ts_tscan**
- **ts_tsdesc**
- **ts_tselem**

Each of these structures is briefly described in this section.

ts_timeseries

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.

ts_tscan

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**.

ts_tsdesc

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 need to 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.

ts_tselem

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 timestamp column in the element is left null, allowing you to avoid the expense of computing the timestamp if it is not needed. The timestamp 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 timestamp 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.

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()**

(1 of 5)

Task Type	Description	Routine Name
Scan a time series	Start a scan	**ts_begin_scan()**
	Retrieve the next element from a scan	**ts_next()**
	End a scan	**ts_end_scan()**
	Find the timestamp 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 associated with a given timestamp	**ts_elem()**
	Get the element at a specified position	**ts_nth_elem()** (regular only)
	Get the first element	**ts_first_elem()**
	Get the last element	**ts_last_elem()**
	Find the next element after a given timestamp	**ts_next_valid()**
	Find the last element before a given timestamp	**ts_previous_valid()**
	Find the last element at or before a given timestamp	**ts_last_valid()**

(2 of 5)

Task Type	Description	Routine Name
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 timestamp	**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 timestamp	Convert timestamp to index	**ts_index ()** (regular only)
	Convert index to timestamp	**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 to an element	**ts_row_to_elem()**
	Convert an element to an MI_ROW	**ts_elem_to_row()**
	Free memory from a time series element created by **ts_make_elem()** or **ts_row_to_elem()**.	**ts_free_elem()**

(3 of 5)

Task Type	Description	Routine Name
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()**
Create and retrieve timestamps	Create a timestamp	**ts_make_stamp()**
	Get fields from a timestamp	**ts_get_stamp_fields()**
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

(4 of 5)

Task Type	Description	Routine Name
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 timestamps	**ts_cal_index()**
	Return all valid timepoints between two timestamps	**ts_cal_range()**
	Return a specified number of timestamps starting at a given timestamp	**ts_cal_range_index()**
	Return the timestamp at a given number of intervals after a given timestamp	**ts_cal_stamp()**
Process timestamps	Compare two timestamps	**ts_datetime_cmp()**

(5 of 5)

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

(1 of 2)

API Routine	SQL Routine
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

(2 of 2)

ts_begin_scan()

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. The **mi_integer** values are described in the section "The flags Argument Values," next.
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" on page 8-44) 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.

The flags Argument Values

The *flags* argument determines how a scan should work on the returned set. The integer 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_hide_elem()" on page 9-68).
TS_SCAN_EXACT_START	256 (0x100)	Return NULL while 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_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.

Description

This function starts a scan of a time series between two timestamps.

The scan description is closed by calling **ts_end_scan()**.

Returns

An open scan description, 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, Appendix A, for an example of the **ts_begin_scan()** function.

Related Topics

"ts_current_offset()" on page 9-38

"ts_current_timestamp()" on page 9-39

"ts_end_scan()" on page 9-49

"ts_next()" on page 9-86

"ts_open()" on page 9-91

ts_cal_index()

The **ts_cal_index()** function returns the number of valid intervals in a calendar between two given timestamps.

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 timestamp. *begin_stamp* must not be earlier than the calendar origin.

end_stamp The timestamp whose offset from *begin_stamp* is to be determined. This timestamp 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 timestamps. If *stamp2* is earlier than *stamp1*, then the result is a negative number.

Related Topics

"ts_cal_range()" on page 9-21

"ts_cal_range_index()" on page 9-23

"ts_cal_stamp()" on page 9-25

ts_cal_pattstartdate()

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 Topics

"ts_cal_startdate()" on page 9-26

"CalPattStartDate" on page 6-6

ts_cal_range()

The **ts_cal_range()** function returns a list of timestamps containing all valid timepoints in a calendar between two timestamps (inclusive of the specified timestamps).

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 need to print out the timestamps of a series of regular time series elements. If the range is known, getting an array of all of the timestamps 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 timestamps.

Related Topics

"ts_cal_index()" on page 9-19

"ts_cal_range_index()" on page 9-23

"ts_time()" on page 9-105

ts_cal_range_index()

The **ts_cal_range_index()** function returns a list containing a specified number of timestamps starting at a given timestamp.

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 timestamps to return.

Description

This function is useful if you need to print out the timestamps of a series of regular time series elements. If the range is known, getting an array of all of the timestamps 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 timestamps.

Related Topics

"ts_cal_index()" on page 9-19

"ts_cal_range()" on page 9-21

"ts_cal_stamp()" on page 9-25

"ts_time()" on page 9-105

ts_cal_stamp()

The **ts_cal_stamp()** function returns the timestamp at a given number of calendar intervals after a given timestamp. The returned timestamp resides 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 timestamp.

offset The number of calendar intervals after the input timestamp.

Description

The equivalent SQL function is **CalStamp**.

Returns

The timestamp representing the given offset, which must be freed by the caller.

Related Topics

"ts_cal_range()" on page 9-21

"ts_cal_range_index()" on page 9-23

ts_cal_startdate()

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 Topics

"ts_cal_pattstartdate()" on page 9-20

"CalStartDate" on page 7-12

ts_close()

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.

Returns

None.

Example

See the **ts_interp()** function, Appendix A, for an example of **ts_close()**.

Related Topics

"ts_open()" on page 9-91

ts_col_cnt()

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.

ts_col_id()

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 timestamp column is always column 0.

Returns

The number of the column associated with *colname*.

Related Topics

"ts_colinfo_name()" on page 9-30

"ts_colinfo_number()" on page 9-31

ts_colinfo_name()

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 Topics

"ts_colinfo_number()" on page 9-31

ts_colinfo_number()

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 timestamp 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, for an example of **ts_colinfo_number()**.

Related Topics

"ts_colinfo_name()" on page 9-30

ts_copy()

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 Topics

"ts_free()" on page 9-51

ts_create()

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 Topics

"TSCreate" on page 8-143

"ts_free()" on page 9-51

"ts_open()" on page 9-91

ts_create_with_metadata()

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 Containing Metadata" on page 4-10 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 so, **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 Topics

"TSCreate" on page 8-143

"ts_create()" on page 9-34

"ts_free()" on page 9-51

"ts_open()" on page 9-91

ts_current_offset()

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 Topics

"ts_begin_scan()" on page 9-16

ts_current_timestamp()

The **ts_current_timestamp()** function finds the timestamp 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 timestamp of the first element. This value cannot be freed by the user with **mi_free()**.

Related Topics

"ts_begin_scan()" on page 9-16

ts_datetime_cmp()

The **ts_datetime_cmp()** function compares two timestamps 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 timestamp to compare.
tstamp2	The second timestamp to compare.

Returns

< 0	If *tstamp1* comes before *tstamp2*.
0	If *tstamp1* equals *tstamp2*.
> 0	If *tstamp1* comes after *tstamp2*.

Related Topics

"ts_get_all_cols()" on page 9-53

"ts_get_col_by_name()" on page 9-55

"ts_get_col_by_number()" on page 9-56

ts_del_elem()

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 Topics

"DelElem" on page 8-55

"ts_ins_elem()" on page 9-72

"ts_put_elem()" on page 9-95

"ts_upd_elem()" on page 9-108

ts_elem()

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 timestamp for the desired element.

STATUS Set on return to indicate whether the element is null or hidden. See "ts_hide_elem()" on page 9-68 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 timestamp 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 and / or null. This element must not be freed by the caller.

Related Topics

"DelElem" on page 8-55

"TS_ELEM_HIDDEN" on page 9-44

"ts_hide_elem()" on page 9-68

"ts_last_elem()" on page 9-76

"ts_nth_elem()" on page 9-90

TS_ELEM_HIDDEN

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 Topics

"ts_elem()" on page 9-42

"TS_ELEM_NULL" on page 9-46

"ts_first_elem()" on page 9-50

"ts_hide_elem()" on page 9-68

"ts_last_elem()" on page 9-76

"ts_next()" on page 9-86

"ts_next_valid()" on page 9-88

"ts_nth_elem()" on page 9-90

"ts_previous_valid()" on page 9-93

TS_ELEM_NULL

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 Topics

"ts_elem()" on page 9-42

"TS_ELEM_HIDDEN" on page 9-44

"ts_first_elem()" on page 9-50

"ts_hide_elem()" on page 9-68

"ts_last_elem()" on page 9-76

"ts_next()" on page 9-86

"ts_next_valid()" on page 9-88

"ts_nth_elem()" on page 9-90

"ts_previous_valid()" on page 9-93

ts_elem_to_row()

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 timestamp 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 Topics

"ts_free_elem()" on page 9-52

"ts_make_elem()" on page 9-80

"ts_make_elem_with_buf()" on page 9-82

"ts_row_to_elem()" on page 9-104

ts_end_scan()

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.

Returns

None.

Example

See the **ts_interp()** function, Appendix A, for an example of **ts_end_scan()**.

Related Topics

"ts_begin_scan()" on page 9-16

ts_first_elem()

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 "ts_hide_elem()" on page 9-68 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 timestamp. 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 Topics

"GetFirstElem" on page 8-64

"ts_begin_scan()" on page 9-16

"ts_elem()" on page 9-42

"ts_next()" on page 9-86

"ts_next_valid()" on page 9-88

ts_free()

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.

Returns

None.

Related Topics

"ts_copy()" on page 9-33

"ts_create()" on page 9-34

ts_free_elem()

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*.

Returns

None.

Related Topics

"ts_make_elem()" on page 9-80

"ts_make_elem_with_buf()" on page 9-82

"ts_row_to_elem()" on page 9-104

ts_get_all_cols()

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 Topics

"ts_col_cnt()" on page 9-28

ts_get_calname()

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()**.

ts_get_col_by_name()

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 Topics

"ts_get_col_by_number()" on page 9-56

ts_get_col_by_number()

The **ts_get_col_by_number()** function pulls the individual pieces of data from an element. The column 0 (zero) is always the timestamp.

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 timestamp.

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, for an example of **ts_get_col_by_number()**.

Related Topics

"ts_get_col_by_name()" on page 9-55

ts_get_containername()

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 Topics

"GetContainerName" on page 8-61

ts_get_flags()

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 Topics

"IsRegular" on page 8-99

"TS_IS_IRREGULAR" on page 9-75

ts_get_metadata()

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 Topics

"GetMetaData" on page 8-73

"GetMetaTypeName" on page 8-74

"TSCreate" on page 8-143

"TSCreateIrr" on page 8-147

"ts_create_with_metadata()" on page 9-36

"ts_get_metadata()" on page 9-60

ts_get_origin()

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 Topics

"GetOrigin" on page 8-81

ts_get_stamp_fields()

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 timestamp 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 mircosecond 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 Topics

"ts_make_stamp()" on page 9-83

ts_get_threshold()

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 Topics

"GetThreshold" on page 8-87

"ts_create()" on page 9-34

ts_get_ts()

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 need to 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 once **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 Topics

"ts_free()" on page 9-51

"ts_put_elem()" on page 9-95

ts_get_typeid()

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 Topics

"ts_copy()" on page 9-33

"ts_create()" on page 9-34

"ts_create_with_metadata()" on page 9-36

"ts_open()" on page 9-91

ts_hide_elem()

The **ts_hide_elem()** function marks the element at the given timestamp 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 timestamp 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* 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 timestamp, an error is raised.

Related Topics

"HideElem" on page 8-88

"ts_elem()" on page 9-42

"TS_ELEM_HIDDEN" on page 9-44

"TS_ELEM_NULL" on page 9-46

"ts_reveal_elem()" on page 9-103

ts_index()

The **ts_index()** function converts from a timestamp to a 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 timestamp to convert.

Description

Consider a time series that starts on Monday, January 1 and keeps track of week days. Calling **ts_index()** with a timestamp argument that corresponds to Monday, January 1, would return 0; a timestamp argument corresponding to Tuesday, January 2, would return 1; a timestamp 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 timestamp falls before the time series origin, or if it is not a valid point in the calendar, -1 is returned.

Related Topics

"GetIndex" on page 8-65

"ts_cal_index()" on page 9-19

"ts_nth_elem()" on page 9-90

"ts_put_nth_elem()" on page 9-100

"ts_time()" on page 9-105

ts_ins_elem()

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 timestamp 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 timestamp is not a valid timepoint in the time series' calendar, an error is raised. If there is already an element at the given timestamp, an error is raised.

Related Topics

"InsElem" on page 8-90

"ts_elem()" on page 9-42

"ts_free()" on page 9-51

"ts_make_elem()" on page 9-80

"ts_make_elem_with_buf()" on page 9-82

"ts_put_elem()" on page 9-95

"ts_upd_elem()" on page 9-108

TS_IS_INCONTAINER

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.

TS_IS_IRREGULAR

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.

ts_last_elem()

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 "ts_hide_elem()" on page 9-68 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 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 Topics

"GetLastElem" on page 8-69

"ts_elem()" on page 9-42

"ts_nth_elem()" on page 9-90

ts_last_valid()

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 timestamp of interest.

STATUS A pointer to an **mi_integer** value. See "ts_hide_elem()" on page 9-68 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 timestamp less than or equal to *tstamp*. For irregular time series, it returns the latest element at or preceding the given timestamp.

Returns

The nearest element at or before the given timestamp. If there is no such element before the timestamp, 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 timestamp.

This element must not be freed by the caller; it is valid until the next element is fetched from the descriptor.

Related Topics

"GetLastValid" on page 8-71

"ts_previous_valid()" on page 9-93

ts_make_elem()

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 timestamp column in the element will be set to null; if *tsdesc* is a descriptor for an irregular time series, the timestamp column is set to whatever was in *values*[0]. This element must be freed by the caller using **ts_free_elem()**.

Related Topics

"ts_elem()" on page 9-42

"ts_make_elem_with_buf()" on page 9-82

"ts_put_elem()" on page 9-95

"ts_free_elem()" on page 9-52

ts_make_elem_with_buf()

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 Topics

"ts_make_elem()" on page 9-80

ts_make_stamp()

The **ts_make_stamp()** function constructs a timestamp 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 timestamp 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 Topics

"ts_get_stamp_fields()" on page 9-63

ts_nelems()

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 Topics

"GetNelems" on page 8-75

ts_next()

Once 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, for an example of **ts_next()**.

Related Topics

"ts_begin_scan()" on page 9-16

ts_next_valid()

The **ts_next_valid()** function returns the nearest entry after a given timestamp.

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 timestamp 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()** ("ts_hide_elem()" on page 9-68) 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 timestamp. For irregular time series, it returns the earliest element following the given timestamp.

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 timestamp. 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 "ts_hide_elem()" on page 9-68 for an explanation of STATUS.

Related Topics

"GetLastValid" on page 8-71

"GetNextValid" on page 8-77

"ts_next()" on page 9-86

"ts_previous_valid()" on page 9-93

ts_nth_elem()

The **ts_nth_elem()** function returns the element at the *n*th position of the given regular 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 to read the element from. It is zero-based. If the offset is less than zero, an error is raised.

STATUS A pointer to an **mi_integer** value that is set on return to indicate whether the element is null. See "ts_hide_elem()" on page 9-68 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.

If the time series is irregular, an error is raised.

Related Topics

"GetNthElem" on page 8-79

"ts_elem()" on page 9-42

"ts_last_elem()" on page 9-76

ts_open()

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. It is generally determined by looking in the MI_FPARAM structure.

flags Must be 0.

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, for an example of **ts_open()**.

Related Topics

"ts_begin_scan()" on page 9-16

"ts_get_typeid()" on page 9-67

"ts_close()" on page 9-27

ts_previous_valid()

The **ts_previous_valid()** function returns the last element preceding the given timestamp.

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 timestamp that follows the element returned.

STATUS Points to an **mi_integer** value that is filled in on return. If no element exists before the timestamp, or if the timestamp falls before the time series origin, *STATUS* is set to a nonzero value. See "ts_hide_elem()" on page 9-68 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 timestamp. 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 timestamp, NULL is returned. NULL is also returned if the given timestamp is less than or equal to the origin of the time series.

Related Topics

"GetPreviousValid" on page 8-83

"ts_hide_elem()" on page 9-68

"ts_last_valid()" on page 9-78

"ts_next()" on page 9-86

"ts_next_valid()" on page 9-88

ts_put_elem()

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 timestamp at which to put the element. The timestamp column of the *tselem* is ignored.

Description

If the timestamp 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 timestamp up to the next second.
2. Search backward for the first element less than the new timestamp.
3. Insert the new data at this timestamp 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 Topics

"PutElem" on page 8-109

"ts_elem()" on page 9-42

"ts_get_ts()" on page 9-66

"ts_ins_elem()" on page 9-72

"ts_make_elem()" on page 9-80

"ts_next()" on page 9-86

"ts_put_elem_no_dups()" on page 9-97

"ts_put_last_elem()" on page 9-99

"ts_upd_elem()" on page 9-108

ts_put_elem_no_dups()

The **ts_put_elem_no_dups()** function puts a new element into an existing time series. The element is inserted whether or not there is already an element with the given timestamp 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 timestamp at which to put the element. The timestamp column of the *tselem* is ignored.

Description

If the timestamp 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 Topics

"PutElemNoDups" on page 8-111

"ts_elem()" on page 9-42

"ts_get_ts()" on page 9-66

"ts_ins_elem()" on page 9-72

"ts_make_elem()" on page 9-80

"ts_next()" on page 9-86

"ts_put_elem()" on page 9-95

"ts_put_last_elem()" on page 9-99

"ts_upd_elem()" on page 9-108

ts_put_last_elem()

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 timestamp in the element is ignored.

Returns

The original time series with the element added. If the time series is irregular, then an error is raised.

Related Topics

"ts_elem()" on page 9-42

"ts_get_ts()" on page 9-66

"ts_make_elem()" on page 9-80

"ts_make_elem_with_buf()" on page 9-82

"ts_next()" on page 9-86

"ts_put_elem()" on page 9-95

ts_put_nth_elem()

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 timestamp 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, then an error is raised.

Related Topics

"ts_elem()" on page 9-42

"ts_get_ts()" on page 9-66

"ts_make_elem()" on page 9-80

"ts_next()" on page 9-86

ts_put_ts()

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 timestamp 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 Topics

"PutTimeSeries" on page 8-117

"ts_put_elem()" on page 9-95

ts_reveal_elem()

The **ts_reveal_elem()** function makes the element at a given timestamp 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 timestamp 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 timestamp.

Related Topics

"RevealElem" on page 8-119

"ts_hide_elem()" on page 9-68

ts_row_to_elem()

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. 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 timestamp 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 timestamp column) of the element will be null.

The element must be freed by the caller using the **ts_free_elem()** procedure.

Related Topics

"ts_elem_to_row()" on page 9-48

"ts_free_elem()" on page 9-52

"ts_make_elem()" on page 9-80

ts_time()

The **ts_time()** function converts from a regular time series offset to a timestamp.

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. If *N* is less than 0, an error is raised.

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 timestamp corresponding to the offset. This value must be freed by the user with **mi_free()**.

Related Topics

"GetStamp" on page 8-85

"ts_index()" on page 9-70

ts_update_metadata

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, then a copy of the specified time series with the metadata removed.

Related Topics

"GetMetaData" on page 8-73

"GetMetaTypeName" on page 8-74

"TSCreate" on page 8-143

"TSCreateIrr" on page 8-147

"ts_create_with_metadata()" on page 9-36

"ts_get_metadata()" on page 9-60

"UpdMetaData" on page 8-167

ts_upd_elem()

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 add.

tstamp The timepoint at which to add the element.

Description

There must already be an element at the given timestamp. 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 Topics

"ts_elem()" on page 9-42

"ts_last_elem()" on page 9-76

"ts_make_elem()" on page 9-80

"ts_make_elem_with_buf()" on page 9-82

"ts_next()" on page 9-86

"ts_put_elem()" on page 9-95

"UpdElem" on page 8-165

The Interp Sample Function

The **Interp** function is an example of a server function that uses the Informix TimeSeries DataBlade module 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 subdirectory of the Informix TimeSeries DataBlade module installation.

The Interp Function Example

```
/*
 * 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_COLS100
# 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_tselemtselem;
    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_booleannulls[TS_MAX_COLS];
    mi_integerminElem, curElem, elem;
    mi_integeri;
    mi_booleannoneYet;
    mi_integerncols;
    charstrbuf[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);
}
```

Using the Interp Function

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';


1994-01-03 00:00:00    1   1   1   1
1994-01-04 00:00:00    2   2   2   2
NULL
1994-01-06 00:00:00    3   3   3   3


select Interp(stock_data) from daily_stocks where stock_name
= 'IBM';

1994-01-03 00:00:00    1   1   1   1
1994-01-04 00:00:00    2   2   2   2
1994-01-05 00:00:00    2.5 2.5 2.5 2.5
1994-01-06 00:00:00    3   3   3   3
```

Appendix

B

The TSIncLoad Sample Procedure

The **TSIncLoad** sample 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 will overwrite the old data, or be appended to the existing time series, depending on the timestamps.

Setting Up the TSIncLoad Example

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 day,
    carryover          char(1),
    spread             integer,
    pricing_bmk_id     integer,
    price              float,
    yield              float,
    priority           char(1) );

create table corporates (
    Secid              integer UNIQUE,
        .
        .
    .
    series     TimeSeries(day_info));

create index corporatesIdx on corporates( Secid);
```

Any name may be used for the **corporates** table. The **corporates** table may 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 1986-1-7 m 2 12 2.2000000000 22.2 6
```

Using the TSIncLoad Example

You can invoke the TSIncLoad procedure with an SQL statement like:

```
execute procedure TSIncLoad( ‘corporates’,
                                 ‘data_file_name’,
                                 ‘cal_name’,
                                 ‘1980-1-1’,
                                 20,
                                 ‘t’,
                                 ‘container-name’);
```

The TSIncLoad Procedure Example

```
#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 of the 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. */
```

Glossary

aggregate function
A function that performs a mathematical operation on a set of rows selected by a query and returns a single value that contains information about those rows: for example, sum, average, or count.

arithmetic function
A function that returns a value by performing a mathematical operation on one or more arguments.

See also: *binary arithmetic function*, *unary arithmetic function*.

binary arithmetic function
A function that performs a mathematical operation on two arguments.

built-in data type
A fundamental data type defined by the database server: for example, INTEGER, CHAR, or SERIAL8.

calendar
In the Informix TimeSeries DataBlade module, a user-specified starting point and pattern of valid entry times that govern time series data.

See also: *calendar pattern*.

calendar pattern
In the Informix TimeSeries DataBlade module, a pattern of time intervals that determine when time series data can be recorded.

calendar pattern interval
In the Informix TimeSeries DataBlade module, the calibration of the calendar pattern, for example, day, month, or year.

calendar pattern length
In the Informix TimeSeries DataBlade module, the amount of time before the calendar pattern repeats.

calibrated search
In the Informix TimeSeries DataBlade module, a search to the natural interval boundaries around a given timepoint.

cast A mechanism that the database server uses to convert data from one data type to another. The server provides built-in casts that it performs automatically. Users can create both implicit and explicit casts.

See also: *explicit cast*, *implicit cast*.

chunk The largest contiguous section of disk space available. A group of chunks defines a dbspace or sbspace.

See also: *dbspace*.

collection An instance of a collection data type; a group of elements of the same data type stored in a SET, MULTISET, or LIST.

See also: *collection data type*.

collection data type A complex data type that groups values, called elements, of a single data type in a column. Collection data types consist of the SET, MULTISET, or LIST type constructor and an element type, which can be any data type, including a complex data type.

complex data type A data type that is built from a combination of other data types using an SQL type constructor or the CREATE ROW TYPE statement, and whose components can be accessed through SQL statements. Complex data types include collection data types and row data types.

constructed type A complex data type created with a type constructor, for example, a collection data type or an unnamed row data type.

constructor type See *type constructor*.

container In the Informix TimeSeries DataBlade module, a structure that the Informix TimeSeries DataBlade module creates and maintains to hold time series data. It exists in a user-specified *dbspace*.

data file A flat file containing data to be loaded into the database.

data type See *built-in data type*, *extended data type*.

DataBlade API (DBAPI) The C application programming interface (API) for Informix Dynamic Server with Universal Data Option. DBAPI is used for the development of DataBlade module applications that access data stored in an Informix Dynamic Server with Universal Data Option database. DBAPI sends SQL command strings to the server for execution and processes results returned by the server to the application.

DataBlade module A collection of database objects and supporting code that extends an object-relational database to manage new kinds of data or add new features. A DataBlade module can include new data types, routines, casts, access methods, SQL code, client code, and installation programs.

DATETIME A built-in data type to hold date and time information. For the Informix TimeSeries DataBlade module, DATETIME YEAR TO FRACTION(5) is used to store a time range from a year to ten microseconds.

dbspace A logical collection of one or more chunks of contiguous disk space. Because chunks represent specific regions of disk space, the creators of databases and tables can control where their data is physically located by placing databases or tables in specific dbspaces.

distinct data type A data type that is created with the CREATE DISTINCT TYPE statement. A distinct data type is based on an existing opaque, built-in, distinct, or named row data type, known as its source type. The distinct data type has the same internal storage representation as its source type, but it has a different name. To compare a distinct data type with its source type requires an explicit cast. A distinct data type inherits all routines that are defined on its source type.

element A member of a collection. In the Informix TimeSeries DataBlade module, an element is the data stored for a particular timepoint.

See also: *collection data type*.

element persistence In the Informix TimeSeries DataBlade module, the period of time for which an element is valid. Regular elements are valid for one interval. Irregular elements are valid until the next element.

element data type The data type of the elements in a collection.

explicit cast A cast that requires a user to specify the CAST AS keyword or cast operator (::) to convert data from one data type to another. An explicit cast requires a function if the internal storage representations of the two data types are not equivalent.

extended data type A term used to refer to data types that are not built-in; namely, complex data types, opaque data types, and distinct data types.

external routine A routine written in a language external to the database (for example, C), whose body is stored outside the database but whose name and parameters are registered in the system catalog tables.

field A component of a named row data type. A field has a name and a data type and is accessed in an SQL statement by using dot notation, for example, **row_type_name.field_name**.

function A routine that can accept arguments and returns one or more values.

See also: *built-in function*, *routine*, *user-defined function*.

fundamental data type A data type that cannot be broken into smaller pieces by the database server, for example, built-in data types and opaque data types.

Global Language Support (GLS) An application environment that allows Informix application-programming interfaces (APIs) and database servers to handle different languages, cultural conventions, and code sets. Developers use the GLS libraries to manage all string, currency, date, and time data types in their code. Using GLS, you can add support for a new language, character set, and encoding by editing resource files, without access to the original source code, and without rebuilding the DataBlade module or client software.

implicit cast A cast that the database server automatically performs to convert data from one data type to another.

See also: *explicit cast*.

inheritance The property that allows a named row data type or a typed table to inherit representation (data fields and columns) and behavior (routines, operators, and rules) from a named row data type or typed table superior to it in a defined hierarchy. Inheritance allows for incremental modification, so that an object can inherit a general set of properties and then add properties that are specific to itself.

intersection A data set consisting of the data that is common to two or more sets of data. Typically, an intersection is the set of rows that are common to two or more specified tables.

interval See *calendar pattern interval*.

irregular time series A time series that stores data associated with arbitrary timepoints.

iterator function An external function that is invoked repeatedly by the database server. A function is an iterator function if the CREATE FUNCTION statement that created it uses the ITERATOR function modifier in the WITH clause.

keyword A word that has meaning to a program. For example, the word SELECT is a keyword in SQL.

large object A data object that exceeds 255 bytes in length. A large object is logically stored in a table column but physically stored independently of the column, because of its size. Large objects can contain non-ASCII data. Informix Dynamic Server with Universal Data Option recognizes two kinds of large objects: simple large objects (TEXT, BYTE) and smart large objects (CLOB, BLOB).

library A collection of precompiled routines.

LIST constructor A type constructor used to create a LIST data type.

LIST data type A collection data type in which elements are ordered and duplicates are allowed.

See also: *collection data type.*

lock A claim, or reservation, that a program places on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it.

There are locks on databases, tables, disk pages, rows, and index-key values.

Locks are necessary to manage concurrency. Informix Dynamic Server with Universal Data Option provides several levels of locking: shared, exclusive, and promotable.

metadata Data about data. Metadata provides information about data in the database or used in the application. Metadata can be data attributes, such as name, size, and data type, or descriptive information about data.

In the Informix TimeSeries DataBlade module, metadata is user-defined data stored in the time series header file. Metadata allows the time series to be self-describing.

multi-representational type A type that intelligently switches between normal in-row data and large objects. For the Informix TimeSeries DataBlade module, time series data is stored in-row until it reaches a user-specified threshold, then it moves to a container.

MULTISET constructor A type constructor used to create a MULTISET data type.

MULTISET data type A collection data type in which elements are not ordered and duplicates are allowed.

See also: *collection type*.

named row data type A row data type that is created with the CREATE ROW TYPE statement and has a name. A named row data type can be used to construct a typed table and can be part of a type or table hierarchy.

See also: *row data type, unnamed row data type.*

offset In the Informix TimeSeries DataBlade module, the number of valid possible timepoints away from the time series origin in a regular time series. It is used as an index for regular time series.

opaque data type A fundamental data type of a predefined fixed or variable length whose internal structure is hidden. Opaque data types are created with the SQL statement CREATE OPAQUE TYPE. Support functions must always be defined for opaque types.

origin In the Informix TimeSeries DataBlade module, the first possible valid timepoint in a time series. It must be at or after the calendar and calendar pattern starting dates.

parameter A variable to which a value can be assigned in a specific application. In a routine, a parameter is the placeholder for the argument values passed to the subroutine at runtime.

procedure A routine that can accept arguments but does not return a value.

registration Loading a DataBlade module's objects into a database. Registration makes a DataBlade module available for use by client applications that open that database.

regular time series A time series that stores data for regularly spaced timepoints, with respect to a calendar. Elements in a regular time series are associated with an offset.

See also: *offset*.

relative search In the Informix TimeSeries DataBlade module, a search a given number of intervals around a given timepoint.

routine A collection of program statements that perform a particular task. Routines include functions, which return one or more values, and procedures, which do not return values.

See also: *function*, *procedure*.

routine overloading Defining more than one routine with the same name but different parameter lists.

ROW constructor Type constructor used to construct unnamed row data types.

row data type A complex data type consisting of a group of ordered data elements (fields) of the same or different data types. The fields of a row type can be of any supported built-in or extended data type, including complex data types, except SERIAL and SERIAL8 and, in certain situations, TEXT and BYTE.

There are two kinds of row data types:

- Named row types, created using the CREATE ROW TYPE statement
- Unnamed row types, created using the ROW constructor

See also: *named row types*, *unnamed row types*.

SET constructor A type constructor used to create a SET data type.

SET data type A collection data type in which elements are not ordered and duplicates are not allowed.

See also: *collection data type.*

set function See *aggregate function*.

shared library A library that contains routines that are used by applications, are loaded into memory by the operating system as they are needed, and are shared with other applications.

subtype A named row type that inherits all representation (data fields) and behavior (routines) from a supertype above it in the type hierarchy and can add additional fields and routines.

For the Informix TimeSeries DataBlade module, the subtype you create is of type **TimeSeries**.

support functions The functions that Informix Dynamic Server with Universal Data Option automatically invokes to process a data type.

The database server uses a support function to perform operations (such as converting to and from the internal, external, and binary representations of the type) on opaque data types.

system catalog A group of database tables that contain information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, the information about indexes and database privileges, and so on.

threshold In the Informix TimeSeries DataBlade module, a user-specified limit for the number of time series elements stored in-row, beyond which the elements are moved to a container.

time series In the Informix TimeSeries DataBlade module, a timestamped series of data entries.

timestamp In the Informix TimeSeries DataBlade module, the date and time portion of a time series element that indicates the beginning of that element's validity.

See also: *DATETIME*.

type constructor An SQL keyword that indicates to the database server the type of complex data to create.

See also: *ROW, SET, MULTISET, LIST*.

unary arithmetic function A function that returns a value by performing a mathematical operation on one argument.

union A data set consisting of the combination of that data from two or more sets of data.

unnamed row data type A row type created with the ROW constructor that has no defined name and no inheritance properties. Two unnamed row types are equivalent if they have the same number of fields and if corresponding fields have the same data type, even if the fields have different names.

user-defined routine A routine, written in one of the languages that Informix Dynamic Server with Universal Data Option supports, that provides added functionality for data types or encapsulates application logic.

Index

A

B

C

D

E

F

G

H

I

S

T

U

V

W

Symbols