

Informix Product Family
Informix
Version 12.10

*IBM Informix TimeSeries Data
User's Guide*



Informix Product Family
Informix
Version 12.10

*IBM Informix TimeSeries Data
User's Guide*



Note

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

This edition replaces SC27-4535-02.

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

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

© **Copyright IBM Corporation 2006, 2014.**

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

Contents

Introduction	xi
About this publication	xi
Types of users	xi
Assumptions about your locale	xi
What's new in TimeSeries data for Informix, Version 12.10	xii
Example code conventions	xvii
Additional documentation	xviii
Compliance with industry standards	xviii
Syntax diagrams	xviii
How to read a command-line syntax diagram	xix
Keywords and punctuation	xx
Identifiers and names	xxi
How to provide documentation feedback	xxi
 Chapter 1. Informix TimeSeries solution	 1-1
Informix TimeSeries solution architecture	1-3
Time series concepts	1-4
TimeSeries data type technical overview	1-5
Regular time series	1-7
Irregular time series	1-7
Packed time series	1-8
JSON time series	1-12
Calendar	1-13
Time series storage	1-14
Getting started with the Informix TimeSeries solution	1-19
Planning for creating a time series	1-19
Planning for data storage	1-20
Planning for loading time series data	1-23
Planning for replication of time series data	1-23
Planning for accessing time series data	1-24
Hardware and software requirements	1-25
Installing the IBM Informix TimeSeries Plug-in for Data Studio	1-25
Database requirements for time series data	1-26
SQL restrictions for time series data	1-26
Time series global language support	1-26
Sample smart meter data	1-27
Setting up stock data examples	1-28
 Chapter 2. Data types and system tables	 2-1
CalendarPattern data type	2-1
Calendar data type	2-4
TimeSeries data type	2-6
Time series return types	2-8
CalendarPatterns table	2-8
CalendarTable table	2-8
TSContainerTable table	2-9
TSContainerWindowTable	2-10
TSContainerUsageActiveWindowVTI Table	2-11
TSContainerUsageDormantWindowVTI Table	2-12
TSInstanceTable table	2-12
 Chapter 3. Create and manage a time series through SQL	 3-1
Example: Create and load a regular time series	3-1
Creating a TimeSeries data type and table	3-2
Creating regular, empty time series	3-2

Creating the data load file	3-3
Loading the time series data	3-3
Accessing time series data through a virtual table	3-4
Example: Create and load a hertz time series	3-5
Example: Create and load a compressed time series.	3-7
Example: Create and load a time series with JSON data	3-10
Defining a calendar	3-12
Predefined calendars.	3-13
Create a time series column	3-13
Creating a TimeSeries subtype	3-13
Create the database table	3-14
Creating containers	3-15
Rules for rolling window containers	3-16
Monitor containers	3-18
Manage container pools	3-19
Create a time series	3-22
Creating a time series with metadata	3-23
Time series input function	3-24
Create a time series with the output of a function	3-27
Load data into an existing time series	3-27
IBM Informix TimeSeries Plug-in for Data Studio	3-27
Writing a loader program	3-31
Loading JSON data	3-33
Loading data from a file into a virtual table	3-34
Load data with the BulkLoad function.	3-35
Load small amounts of data with SQL functions	3-36
Delete time series data	3-37
Manage packed data.	3-37

Chapter 4. Virtual tables for time series data 4-1

Performance of queries on virtual tables	4-2
The structure of virtual tables	4-2
The display of data in virtual tables	4-3
Insert data through virtual tables	4-4
Creating a time series virtual table	4-5
TSCreateVirtualTab procedure	4-5
Example of creating a virtual table	4-8
Example of creating a fragmented virtual table	4-10
TSCreateExpressionVirtualTab procedure	4-13
The TSVTMode parameter.	4-16
Drop a virtual table	4-26
Trace functions	4-26
The TSSetTraceFile function	4-26
TSSetTraceLevel function	4-27

Chapter 5. Calendar pattern routines 5-1

AndOp function	5-1
CalPattStartDate function	5-2
Collapse function	5-3
Expand function	5-4
NotOp function.	5-4
OrOp function	5-5

Chapter 6. Calendar routines 6-1

AndOp function	6-1
CalIndex function	6-2
CalRange function	6-3
CalStamp function	6-4
CalStartDate function	6-5
OrOp function	6-5

Chapter 7. Time series SQL routines	7-1
Time series SQL routines sorted by task.	7-2
Time series routines that run in parallel.	7-7
The flags argument values	7-9
Abs function	7-11
Acos function	7-11
AggregateBy function	7-11
AggregateRange function	7-15
Apply function	7-18
ApplyBinaryTsOp function	7-23
ApplyCalendar function	7-24
ApplyOpToTsSet function	7-25
ApplyUnaryTsOp function.	7-26
Asin function	7-27
Atan function	7-27
Atan2 function.	7-27
Binary arithmetic functions	7-27
BulkLoad function	7-30
Clip function	7-31
ClipCount function	7-35
ClipGetCount function	7-37
Cos function	7-38
CountIf function	7-38
DelClip function	7-42
DelElem function	7-43
DelRange function	7-44
DelTrim function	7-45
Divide function	7-46
ElemIsHidden function	7-47
ElemIsNull function	7-47
Exp function	7-48
FindHidden function	7-48
GetCalendar function	7-48
GetCalendarName function	7-49
GetClosestElem function	7-49
GetCompression function	7-50
GetContainerName function	7-51
GetElem function.	7-52
GetFirstElem function	7-53
GetFirstElementStamp function	7-53
GetHertz function	7-54
GetIndex function	7-55
GetInterval function	7-55
GetLastElem function	7-56
GetLastElementStamp function	7-57
GetLastNonNull function	7-57
GetLastValid function	7-58
GetMetaData function	7-59
GetMetaTypeName function	7-59
GetNelems function	7-60
GetNextNonNull function	7-61
GetNextValid function	7-61
GetNthElem function	7-62
GetOrigin function	7-64
GetPacked function	7-64
GetPreviousValid function	7-65
GetStamp function	7-66
GetThreshold function	7-67
HideElem function	7-67
HideRange function	7-68
InsElem function	7-69

InsSet function	7-70
InstanceId function	7-71
Intersect function	7-71
IsRegular function	7-73
Lag function	7-73
Logn function	7-74
Minus function	7-74
Mod function	7-75
Negate function	7-75
NullCleanup function	7-75
Plus function	7-77
Positive function	7-77
Pow function	7-77
PutElem function	7-77
PutElemNoDups function	7-79
PutNthElem function	7-80
PutSet function	7-80
PutTimeSeries function	7-82
RevealElem function	7-83
RevealRange function	7-83
Round function	7-84
SetContainerName function	7-84
SetOrigin function	7-85
Sin function	7-85
Sqrt function	7-85
Tan function	7-86
Times function	7-86
TimeSeriesRelease function	7-86
Transpose function	7-86
TSAddPrevious function	7-90
TSCmp function	7-90
TSColNameToList function	7-91
TSColNumToList function	7-92
TSContainerCreate procedure	7-93
TSContainerDestroy procedure	7-98
TSContainerLock procedure	7-99
TSContainerManage function	7-99
TSContainerNElems function	7-104
TSContainerPctUsed function	7-105
TSContainerPoolRoundRobin function	7-107
TSContainerPurge function	7-108
TSContainerSetPool procedure	7-111
TSContainerTotalPages function	7-112
TSContainerTotalUsed function	7-113
TSContainerUsage function	7-114
TSCreate function	7-116
TSCreateIrr function	7-118
TSDecay function	7-122
TSL_Attach function	7-123
TSL_Commit function	7-124
TSL_Flush function	7-126
TSL_FlushAll function	7-128
TSL_FlushInfo function	7-129
TSL_FlushStatus function	7-131
TSL_GetKeyContainer function	7-131
TSL_GetLogMessage function	7-132
TSL_Init function	7-133
TSL_Put function	7-135
TSL_PutRow function	7-137
TSL_PutSQL function	7-138
TSL_SessionClose function	7-139

TSL_SetLogMode function	7-140
TSL_Shutdown procedure	7-141
TSPrevious function	7-141
TSRollup function	7-142
TSRowNameToList function	7-145
TSRowNumToList function	7-146
TSRowToList function	7-147
TSRunningAvg function	7-147
TSRunningCor function	7-149
TSRunningMed function	7-150
TSRunningSum function	7-151
TSRunningVar function	7-152
TSSetToList function	7-153
TSToXML function	7-154
Unary arithmetic functions	7-156
Union function	7-157
UpdElem function	7-159
UpdMetaData function	7-159
UpdSet function	7-160
WithinC and WithinR functions.	7-161

Chapter 8. Time series Java class library. 8-1

Java class files and sample programs	8-3
Preparing the server for Java classes	8-3
Mapping time series data types	8-4
Querying time series data with the IfmxTimeSeries object.	8-4
Obtaining the time series Java class version	8-5

Chapter 9. Time series API routines 9-1

Differences in using functions on the server and on the client	9-1
Data structures for the time series API	9-2
The ts_timeseries structure	9-2
The ts_tscan structure.	9-2
The ts_tsdesc structure	9-2
The ts_tselem structure	9-3
Time series API routines sorted by task	9-3
The ts_begin_scan() function	9-7
The ts_cal_index() function	9-9
The ts_cal_pattstartdate() function	9-9
The ts_cal_range() function	9-10
The ts_cal_range_index() function	9-11
The ts_cal_stamp() function	9-11
The ts_cal_startdate() function	9-12
The ts_close() function	9-12
The ts_closest_elem() function	9-13
The ts_col_cnt() function	9-14
The ts_col_id() function.	9-14
The ts_colinfo_name() function	9-15
The ts_colinfo_number() function	9-15
The ts_copy() function	9-16
The ts_create() function.	9-17
The ts_create_with_metadata() function	9-18
The ts_current_offset() function	9-20
The ts_current_timestamp() function	9-21
The ts_datetime_cmp() function	9-21
The ts_del_elem() function.	9-22
The ts_elem() function	9-22
The TS_ELEM_HIDDEN macro	9-23
The TS_ELEM_NULL macro	9-24
The ts_elem_to_row() function	9-24

The ts_end_scan() procedure	9-25
The ts_first_elem() function	9-25
The ts_free() procedure	9-26
The ts_free_elem() procedure	9-26
The ts_get_all_cols() procedure	9-27
The ts_get_calname() function	9-27
The ts_get_col_by_name() function	9-28
The ts_get_col_by_number() function	9-28
The ts_get_compressed() function	9-29
The ts_get_containername() function	9-29
The ts_get_flags() function	9-30
The ts_get_hertz() function	9-30
The ts_get_metadata() function	9-31
The ts_get_origin() function	9-31
The ts_get_packed() function	9-32
The ts_get_stamp_fields() procedure	9-32
The ts_get_threshold() function	9-33
The ts_get_ts() function	9-33
The ts_get_typeid() function	9-34
The ts_hide_elem() function	9-34
The ts_index() function	9-35
The ts_ins_elem() function	9-36
The TS_IS_INCONTAINER macro	9-36
The TS_IS_IRREGULAR macro	9-37
The ts_last_elem() function	9-37
The ts_last_valid() function	9-38
The ts_make_elem() function	9-38
The ts_make_elem_with_buf() function	9-39
The ts_make_stamp() function	9-40
The ts_nelems() function	9-41
The ts_next() function	9-41
The ts_next_valid() function	9-42
The ts_nth_elem() function	9-43
The ts_open() function	9-44
The ts_previous_valid() function	9-45
The ts_put_elem() function	9-46
The ts_put_elem_no_dups() function	9-47
The ts_put_last_elem() function	9-48
The ts_put_nth_elem() function	9-48
The ts_put_ts() function	9-49
The ts_reveal_elem() function	9-50
The ts_row_to_elem() function	9-50
The ts_time() function	9-51
The ts_tstamp_difference() function	9-51
The ts_tstamp_minus() function	9-52
The ts_tstamp_plus() function	9-53
The ts_update_metadata() function	9-54
The ts_upd_elem() function	9-54

Appendix A. The Interp function example A-1

Appendix B. The TSIncLoad procedure example B-1

Appendix C. Accessibility C-1

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

Notices	D-1
Privacy policy considerations	D-3
Trademarks	D-3
Index	X-1

Introduction

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

About this publication

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

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

Types of users

This publication is written for the following audience:

- Developers who write applications to access time series information stored in IBM® Informix® databases

Assumptions about your locale

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

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

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

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

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

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

What's new in TimeSeries data for Informix, Version 12.10

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

For a complete list of what's new in this release, go to http://pic.dhe.ibm.com/infocenter/informix/v121/topic/com.ibm.po.doc/new_features_ce.htm.

Table 1. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC4

Overview	Reference
<p>Enhancements to the time series Java™ class library</p> <p>When you write a Java application with the time series Java class library, now you can define time series objects with the new builder classes. Previously, you defined time series objects with string representations of SQL statements. Builder classes reduce the possibility of errors and improve usability. The methods in the Java class library run faster than in previous releases.</p> <p>The time series Java class library has the following enhancements for creating time series objects:</p> <ul style="list-style-type: none">• You can now determine whether the definitions of two calendars or calendar patterns are the same.• You can create calendar patterns and calendars with new <code>IfmxCalendarPattern.Builder</code> and <code>IfmxCalendar.Builder</code> classes.• You can create and manage containers with the new <code>TimeSeriesContainer</code> and <code>TimeSeriesContainer.Builder</code> classes.• You can create TimeSeries row types with the new <code>TimeSeriesRowType</code> and <code>TimeSeriesRowType.Builder</code> classes.• You can create a simpler custom type map that uses a <code>PatternClassMap</code> instead of individual entries for each data type with the new <code>TimeSeriesTypeMap</code> and <code>TimeSeriesTypeMap.Builder</code> classes. <p>The <code>IfmxTimeSeries</code> class has the following enhancements for managing time series data:</p> <ul style="list-style-type: none">• You can insert data into a time series with the new <code>IfmxTimeSeries.Builder</code> class.• You can easily modify data and process query results because the results of queries on time series data are now JDBC updatable result sets.• You can distinguish between case sensitive and case insensitive databases and make multiple updates within a row.• You can convert the time series data to the appropriate time zone on the client.• You can select and update data by specifying similar data types instead of the exact data types. Data is implicitly cast during read and write operations. Previously, transactions that did not specify the exact data types failed.	<p>Chapter 8, “Time series Java class library,” on page 8-1</p>

Table 1. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC4 (continued)

Overview	Reference
<p>Include JSON documents in time series</p> <p>You can include JSON documents that are associated with timestamps in time series. For example, weather monitoring sensors that return 2 - 50 values in JSON documents through the REST API every 10 minutes. You store JSON documents with time series data as BSON documents in a BSON column in the TimeSeries data type.</p>	<p>"JSON time series" on page 1-12</p> <p>"Example: Create and load a time series with JSON data" on page 3-10</p>
<p>Create a time series with the REST API or the MongoDB API</p> <p>If you have applications that handle time series data, you can now create and manage a time series with the REST API or the MongoDB API. Previously, you created a time series by running SQL statements. For example, you can program sensor devices that do not have client drivers to load time series data directly into the database with HTTP commands from the REST API.</p> <p>You create time series objects by adding definitions to time series collections. You interact with time series data through a virtual table.</p>	<p>This feature is documented in the <i>IBM Informix JSON Compatibility Guide</i>.</p> <p>Create time series through the wire listener (JSON compatibility)</p>
<p>Replicate hertz and compressed time series data</p> <p>You can now replicate hertz and compressed time series data with Enterprise Replication.</p>	

Table 2. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC3

Overview	Reference
<p>Efficient storage for hertz and numeric time series data</p> <p>You can save disk space by packing multiple time series records in each element. If your data is recorded with a regular subsecond frequency, you can define a hertz time series to pack records for a second of data in each time series element. If all the columns in your TimeSeries data type are numeric, you can define a compressed time series to pack and compress up to 4 KB of records in each time series element.</p>	<p>"Packed time series" on page 1-8</p> <p>"Example: Create and load a hertz time series" on page 3-5</p> <p>"Example: Create and load a compressed time series" on page 3-7</p>
<p>Faster queries by running time series routines in parallel</p> <p>Time series SQL routines that you include in the WHERE clause of SELECT statements return results faster when they run in parallel. If you fragment the table that contains the time series data and enable PDQ, time series SQL routines run in parallel.</p>	<p>"Time series routines that run in parallel" on page 7-7</p>

Table 2. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC3 (continued)

Overview	Reference
<p>Control the destroy behavior for rolling window containers</p> <p>You can limit the number of partitions of a rolling window container that can be destroyed in an operation. You control how many partitions are destroyed and whether active partitions can be destroyed when the number of partitions that must be detached is greater than the size of the dormant window. When you create a rolling window container, set the <i>destroy_count</i> parameter to a positive integer and the <i>window_control</i> parameter to 2 or 3 in the TSCreateContainer function. You can change the destroy behavior of an existing rolling window container by including the <i>wcontrol</i> parameter in the TSTContainerManage function.</p>	<p>"TSContainerCreate procedure" on page 7-93</p> <p>"TSContainerManage function" on page 7-99</p>
<p>Monitor groups of containers with wildcard characters</p> <p>You can monitor groups of containers that have similar names. Include the wildcard characters for the MATCHES operator in the parameter for the container name in the TSContainerUsage, TSContainerTotalPages, TSContainerTotalUsed, TSContainerPctUsed, TSContainer, and TSContainerNElems functions.</p>	<p>"TSContainerUsage function" on page 7-114</p> <p>"TSContainerTotalPages function" on page 7-112</p> <p>"TSContainerTotalUsed function" on page 7-113</p> <p>"TSContainerPctUsed function" on page 7-105</p> <p>"TSContainerNElems function" on page 7-104</p>
<p>Faster queries with IN conditions through virtual tables</p> <p>Access methods that are created through the virtual table interface now process IN conditions in query predicates that operate on simple columns. Processing through a virtual table interface is generally faster than SQL processing. For example, queries with IN conditions that you run on time series virtual tables now run faster than in previous releases.</p>	

Table 3. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC2

Overview	Reference
<p>Accelerate queries on time series data</p> <p>You accelerate queries on time series data by creating data marts that are based on time series virtual tables.</p> <p>You can define virtual partitions so that you can quickly refresh the data in part of the data mart or continuously refresh the data. You can make queries faster by limiting the amount of data in the data mart to specific time intervals.</p>	<p>"Planning for accessing time series data" on page 1-24</p> <p>"Performance of queries on virtual tables" on page 4-2</p>
<p>Faster queries on time series virtual tables</p> <p>You can run queries in parallel on a virtual table that is fragmented. The virtual table must be based on a time series table that is fragmented by expression. Include the fragment flag in the <i>TSVTMode</i> parameter when you create the virtual table.</p> <p>You can include the flags for the <i>TSVTMode</i> parameter as a set of strings instead of as a number.</p>	<p>"Example of creating a fragmented virtual table" on page 4-10</p> <p>"The TSVTMode parameter" on page 4-16</p>

Table 3. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC2 (continued)

Overview	Reference
<p>Replicate time series data with all high-availability clusters</p> <p>You can now replicate time series data with all types of high-availability clusters. Previously, you could replicate time series data only with High-Availability Data Replication (HDR) clusters, and not with shared-disk secondary and remote stand-alone secondary clusters. Secondary servers must be read-only.</p>	<p>"Planning for replication of time series data" on page 1-23</p>
<p>Order TimeSeries columns in query results</p> <p>You can include a TimeSeries column in an ORDER BY clause of an SQL query. The ORDER BY clause sorts the results from the TimeSeries column by the time series instance ID.</p>	<p>"TSInstanceTable table" on page 2-12</p>
<p>Improvements for time series loader programs</p> <p>You have new options for how you flush time series data to disk when you write a loader program. You can flush time series elements for all containers to disk in a single transaction or in multiple transactions. If you want your client application to control transactions, run the TSL_FlushAll function. The TSL_FlushAll function flushes time series elements to disk in one transaction. If you want the loader program to control the size of your transactions, run the TSL_Commit function. The TSL_Commit function flushes time series elements to disk in multiple transactions, based on the commit interval that you specify.</p> <p>You can view the results of the data flushing function by running the TSL_FlushInfo function.</p> <p>You can specify that no duplicate elements are allowed when you flush time series data to disk.</p>	<p>"TSL_FlushAll function" on page 7-128</p> <p>"TSL_Commit function" on page 7-124</p> <p>"TSL_FlushInfo function" on page 7-129</p>
<p>Faster aggregation of an interval of time series data</p> <p>You can aggregate an interval of time series data faster by including start and end dates in the TSRollup function. Previously, you selected an interval of time series data with the Clip or similar function and passed the results to the TSRollup function.</p>	<p>"TSRollup function" on page 7-142</p>

Table 4. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC1

Overview	Reference
<p>Manage time-series data in rolling window containers</p> <p>You can control the amount of time-series data that is stored in containers by specifying when to delete obsolete data. You create a rolling window container that has multiple partitions that are stored in multiple dbspaces. You configure a rolling window container to define the time interval for each partition and how many partitions are allowed: for example, 12 partitions that each store a month of data. When you insert data for a new month, a new partition is created, and if the number of partitions exceed the maximum that is allowed, the oldest partition becomes dormant. You specify when to destroy dormant partitions. Previously, you had to delete obsolete data manually.</p>	<p>"Time series storage" on page 1-14</p> <p>"Rules for rolling window containers" on page 3-16</p>
<p>Load time-series data faster by reducing logging</p> <p>If you load time-series elements into containers in a single transaction, you can save time by specifying a reduced amount of logging. By default, every time-series element that you insert generates two log records: one for the inserted element and one for the page header update. However, you can specify that page header updates are logged for each transaction instead. For example, you can insert a set of daily meter readings for a meter in one transaction and reduce the amount of logging by almost half.</p> <p>Run one or more of the PutElem, PutElemNoDups, PutNthElem, InsElem, BulkLoad, or PutTimeSeries functions with the TSOPEN_REDUCED_LOG (256) flag or the TSL_Flush function with the 257 flag within a transaction without other functions or SQL statements. If you insert data through a virtual table, run the TSCreateVirtualTab procedure with the TS_VTI_REDUCED_LOG (256) flag, and then insert data within a transaction without other types of statements.</p>	<p>"The flags argument values" on page 7-9</p> <p>"The TSVTMode parameter" on page 4-16</p>
<p>Replicate time-series data</p> <p>You can replicate time-series data with Enterprise Replication. For example, if you collect time-series data in multiple locations, you can consolidate the data to a central server.</p>	<p>"Planning for replication of time series data" on page 1-23</p>
<p>Faster writing to time-series containers</p> <p>By default, multiple sessions can now write to a time-series container simultaneously. However, you can limit the number of sessions to one. Data is loaded faster if only one session writes to the container. Use the TSContainerLock procedure to control whether multiple sessions are allowed. Previously, you wrote your application to prevent more than one session from writing to a container at one time.</p>	<p>"TSContainerLock procedure" on page 7-99</p>

Table 4. What's New in IBM Informix TimeSeries Data User's Guide for 12.10.xC1 (continued)

Overview	Reference
<p>Write a custom program to load time-series data</p> <p>You can use time-series SQL routines to write a custom program that loads time-series data into the Informix database. You can load data in parallel in a highly efficient manner by controlling what data is loaded into which containers. You can include a custom loader program in your application.</p>	<p>"Writing a loader program" on page 3-31</p>
<p>Enhancements to the Informix TimeSeries Plug-in for Data Studio</p> <p>When you use the Informix TimeSeries Plug-in for Data Studio, you can load time-series data into an Informix database directly from another database. You do not have to export the data into a file. When you create a table definition, specify a connection to a database and a query to return the data that you want to load. You can preview the returned data to validate the query. You can also set other properties of the load job within the plug-in.</p>	<p>"Create a load job to load data from a database" on page 3-29</p>
<p>Return the timestamp of the first or last time-series element</p> <p>You can return the timestamp of the first or last element in a time series by running the GetFirstElementStamp function or the GetLastElementStamp function. You can choose whether the element can be null or must contain data. For example, you can return the first element that has data to determine the number of null elements between the origin and the first element that has data.</p>	<p>"GetFirstElementStamp function" on page 7-53</p> <p>"GetLastElementStamp function" on page 7-57</p>
<p>Faster queries through virtual tables</p> <p>Queries on time series virtual tables now run faster because qualifiers to a WHERE clause that contain multiple column, constant, or expression parameters are processed through the virtual table interface instead of through SQL processing.</p>	

Example code conventions

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

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

```
CONNECT TO stores_demo
...

DELETE FROM customer
  WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL

at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

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

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

Additional documentation

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

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

Compliance with industry standards







IBM Informix products are compliant with various standards.

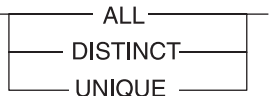
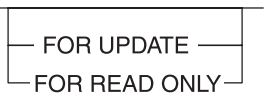
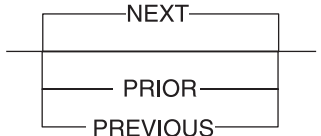
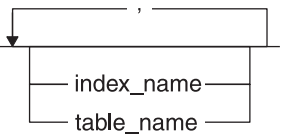
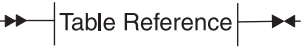
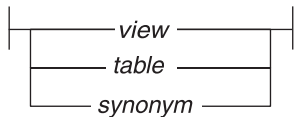
IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment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

Syntax diagrams

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

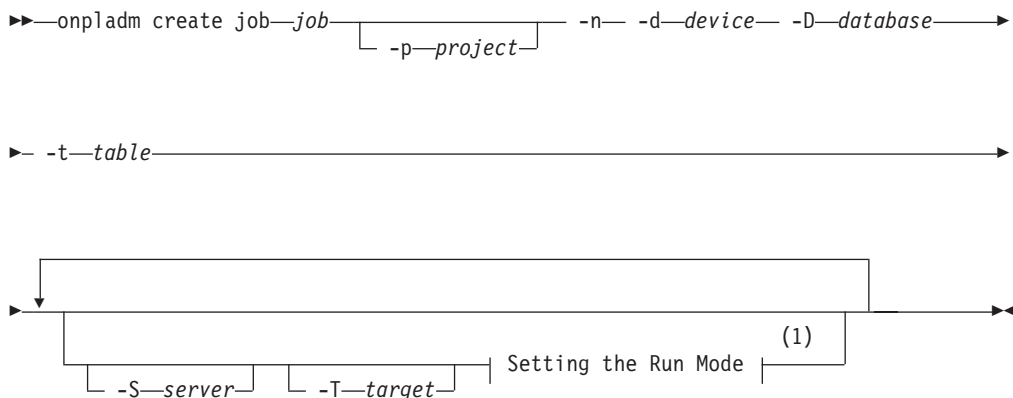
Table 5. Syntax Diagram Components

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

Component represented in PDF	Component represented in HTML	Meaning
	<pre> ---+---ALL-----+--- +--DISTINCT-----+ '---UNIQUE-----'</pre>	Required item with choice. Only one item must be present.
	<pre> ---+-----+--- +--FOR UPDATE---+ '--FOR READ ONLY--'</pre>	Optional items with choice are shown below the main line, one of which you might specify.
	<pre> .---NEXT-----. ---+-----+--- +---PRIOR-----+ '---PREVIOUS-----'</pre>	The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line is used by default.
	<pre> .-----,----- v----- ---+-----+--- +---index_name---+ '---table_name---'</pre>	Optional items. Several items are allowed; a comma must precede each repetition.
	<pre>>>- Table Reference -<<</pre>	Reference to a syntax segment.
<p>Table Reference</p> 	<pre> -----view-----+--- +-----table-----+ '-----synonym-----'</pre>	Syntax segment.

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

Creating a no-conversion job

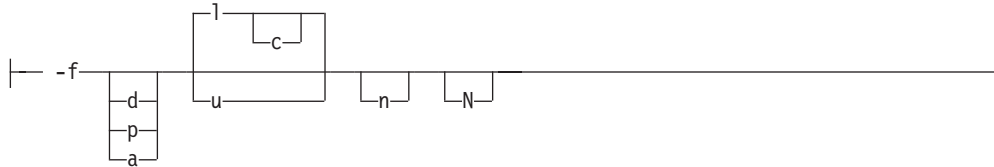


Notes:

- 1 See page Z-1

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

Setting the run mode:



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

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

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

Keywords and punctuation

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

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

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

Identifiers and names

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

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

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

►►—SELECT—*column_name*—FROM—*table_name*—◄◄

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

How to provide documentation feedback

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

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- In the Informix information center, which is available online at <http://www.ibm.com/software/data/sw-library/>, open the topic that you want to comment on. Click the feedback link at the bottom of the page, complete the form, and submit your feedback.
- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

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

We appreciate your suggestions.

Chapter 1. Informix TimeSeries solution

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

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

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

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

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

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

The Informix TimeSeries solution loads and queries time stamped data faster, requires less storage space, and provides more analytical capability than a standard relational table implementation. Although relational database management systems can store time series data for standard types by storing one row per time-stamped

data entry, performance is poor and storage is inefficient. The Informix TimeSeries solution saves disk space by not storing duplicate information from the columns that do not contain the time-based data. The Informix TimeSeries solution loads and queries time series data quickly because the data is stored on disk in order by time stamp and by source.

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

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

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

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

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

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

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

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

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

Informix TimeSeries solution architecture

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

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

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

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

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

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

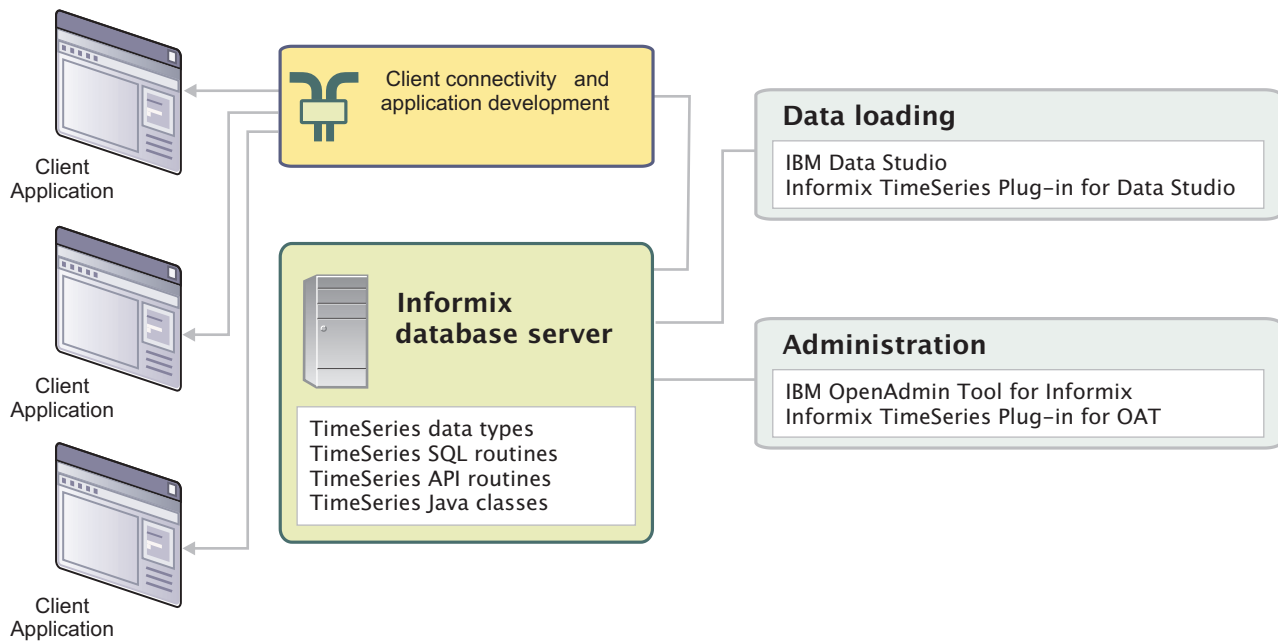


Figure 1-1. Informix TimeSeries architecture

Time series concepts

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

Understand the following concepts when you create a time series:

TimeSeries data type

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

Element

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

Packed element

An element in which records for multiple timestamps are stored to save storage space. Packed elements can store hertz data that is recorded at a subsecond frequency and compressed numeric data.

Timepoint

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

Origin

The earliest time stamp that is allowed. Data that has a timestamp before the origin is not allowed.

time series instance

For each **TimeSeries** data type value, the set of elements that is stored in a container. Each instance has a unique identifier that is stored in the **TSInstanceTable** table. The time series instance ID is used by some time series routines and SQL statements. The time series instance ID is used by some time series routines.

Calendar

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

Calendar pattern

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

Container

A named portion of a dbspace that contains the time series data for a specific **TimeSeries** data type and regularity. The data is ordered by time stamp. You can control in which containers your time series data is stored.

Regularity

Whether a time series has regularly spaced timepoints or irregularly spaced timepoints.

Virtual table

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

TimeSeries data type technical overview

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

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

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

Database table **ts_data**

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

Time series data for ID 1000111

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

Time series data for ID 1046021

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

Time series data for ID 1090954

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

Figure 1-2. TimeSeries data type architecture

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

Related concepts:

“TimeSeries data type” on page 2-6

Related tasks:

“Creating a TimeSeries subtype” on page 3-13

Regular time series

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

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

Table 1-4. Offsets for a daily time series

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

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

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

Related reference:

“Create a time series” on page 3-22

Irregular time series

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

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

Hertz data and compressed numeric data are stored in irregular time series because multiple records are packed into each element. However, hertz and compressed time series require regularly spaced data and behave differently from standard irregular time series.

Related reference:

“Create a time series” on page 3-22

Packed time series

A packed time series stores records for multiple timepoints in each element to reduce storage space. You can create a packed time series if you have hertz data or you have numeric data that you want to compress. Both hertz and compressed numeric data must be recorded at regular intervals. Packed elements save approximately 4 bytes per record as compared to a regular time series, not including the savings for compressing the data.

For a hertz time series, each time series element is packed with records for one second. Hertz data is recorded at a regular subsecond frequency. For example, an electrical grid might have phasor measurement units that measure electrical waves at 50 hertz.

For a compressed time series, each time series element is packed with compressed records until the size of the element approaches 4 KB. All of the columns in the **TimeSeries** subtype must be numeric. You define compression separately for each column, except the first timestamp column, which is compressed by default. For example, a weather station might measure the wind speed, air temperature, air pressure, and precipitation for each weather sensor every 15 minutes.

Packed data is loaded faster than data that is not packed because packed data generates fewer log records. Each element that you insert into the database generates one or two log records, depending on whether logging is reduced. Packed data requires fewer elements.

When you create a virtual table or run a query, packed data is indistinguishable from time series data that is not packed. Each subelement that is shown in a virtual table has a row. Each subelement that is returned by a query is shown as an individual element.

Hertz time series

If your time series data is recorded at a regular subsecond frequency, you can define a hertz time series to store the data efficiently.

In each element, a hertz time series stores an 11-byte timestamp for the first record and a 1-byte timestamp for each of the other records. An element contains records for one second. For example, if data is recorded 5 times a second, then each element contains 5 sets of values. If the **TimeSeries** subtype contains a timestamp column and two other columns, the following table shows how the values are stored in each element.

Table 1-5. How hertz data is packed in time series elements

Timestamp of element	Values for .00000 seconds	Values for .20000 seconds	Values for .40000 seconds	Values for .60000 seconds	Values for .80000 seconds
2014-01-01 00:00:00.00000	1.01, 0.25	1.93, 0.11	1.74, 0.02	1.03, 0.45	1.85, 0.44

Table 1-5. How hertz data is packed in time series elements (continued)

Timestamp of element	Values for .00000 seconds	Values for .20000 seconds	Values for .40000 seconds	Values for .60000 seconds	Values for .80000 seconds
2014-01-01 00:00:01.00000	2.00, 0.02	1.99, 0.05	1.53, 0.03	NULL, NULL	1.76, 0.01

Data requirements

Hertz records must have timestamps that conform to the hertz subsecond boundaries within approximately 0.3% tolerance. Only one record per subsecond boundary is allowed. For example, if the hertz value is 5, then there are 5 valid timestamps per second. For the first second after midnight, the following timestamps are valid:

- 00:00:00.00000
- 00:00:00.20000
- 00:00:00.40000
- 00:00:00.60000
- 00:00:00.80000

The tolerance means that a timestamp has a margin of error of plus or minus 0.00003 seconds. For example, for the second record, a subsecond value of 0.20003 is accepted and stored as 0.2, but a subsecond value of 0.20004 is rejected.

If the value of a subsecond boundary has more than 5 significant digits, the value is rounded down. For example, the 127th subsecond value for a hertz of 255 has the value 0.498039216, which is rounded down to 0.49803.

NULL values are allowed.

The size of the data for 1 second cannot exceed 32769 bytes.

Time series definition

The **TimeSeries** subtype that you define for hertz data must have columns of only the following data types: SMALLINT, INT, BIGINT, SMALLFLOAT, FLOAT, DATE, INT8, CHAR, VARCHAR, NCHAR, NVCHAR, LVARCHAR, DATETIME, DECIMAL, and MONEY.

You can use a calendar with any interval size because the interval is significant only in defining on and off periods. The interval size for a hertz time series is defined by the *hertz* parameter when you create the time series.

You define a hertz time series by running the **TSCreateIrr** function with the *hertz* parameter.

Hertz data must be stored in containers that are not rolling window containers.

Related concepts:

“Manage packed data” on page 3-37

“TimeSeries data type” on page 2-6

Related tasks:

“Example: Create and load a hertz time series” on page 3-5

Related reference:

"TSCreateIrr function" on page 7-118

Compressed numeric time series

If your time series data is recorded at a regular frequency and all the time series values are numeric, you can define a compressed time series to store the data efficiently.

In each element, a compressed time series stores an 11-byte timestamp for the first record and a 2-byte timestamp for each of the other records. The compression ratio of the rest of the time series data varies depending on the type of data and the compression definitions. For example, you can compress an 8-byte BIGINT value down to 1 byte, with some loss of precision.

Time series definition

You define a compressed time series by running the **TSCreateIrr** function with the *compression* parameter.

You must include a compression definition for every column in the **TimeSeries** subtype, except the first timestamp column. The compression definitions are associated with the columns in the same order. If you do not want to compress a particular column, include a compression definition of no compression for that column. If you specify that none of the columns are compressed, only the first timestamp column is compressed.

Besides the first timestamp column, the **TimeSeries** subtype columns must have only the following data types: SMALLINT, INTEGER, BIGINT, SMALLFLOAT, and FLOAT.

The calendar that you specify in the time series definition defines the size of the interval, however, off periods are not allowed. One record per interval is accepted and the timestamp must be on the interval boundary. For example, if the calendar has an interval of minute, a timestamp that has seconds values other than 00.00000, such as 2013-01-01 01:52:15.00000, is rejected.

Compressed records must be stored in containers. However, a compressed time series cannot be stored in rolling window containers.

Compression types

You compress data with the following types of compression algorithms:

Quantization

The quantization compression algorithm divides continuous values into discrete grids. Each grid represents a range of values. Fewer bytes are needed to represent a grid than a numeric value. The quantization algorithm can be lossy. The quantization algorithm allows NULL values.

The quantization compression algorithm is suitable when records are frequent and the values are highly variable.

You specify the upper and lower boundaries of the values of the data and the number of bytes to store for each value. The larger the difference between the upper and lower boundaries and the smaller the compressed size that you specify, the more compact and possibly lossy the data becomes.

Linear

The linear compression algorithm represents values as line segments, which are defined by two end points. If the values are within the supplied deviation, the values are not recorded. The linear compression algorithm records a value only when a new value deviates too much from the last recorded value. The linear compression algorithm does not allow NULL values.

The linear compression algorithm is suitable when values vary little.

The larger the maximum deviation that you specify, the more compact and possibly lossy the data becomes.

Use the boxcar variant if you need fast reading and writing performance.

Use the swing door variant if you need a higher compression ratio.

You can combine compression types and choose the quantization linear boxcar or quantization linear swing door compression algorithm.

You can choose not to compress a column. Columns that are not compressed allow NULL values.

Lossiness

The following equation describes margin of error that is allowed between the original and the compressed values for the different compression types:

Quantization type:

$$\text{margin of error} = \frac{(\text{upper_bound} - \text{lower_bound})}{(2^{(\text{compress_size} * 8)})}$$

Linear types:

$$\text{margin of error} = \text{maximum_deviation}$$

Combination of quantization and linear types:

$$\text{margin of error} = \frac{(\text{upper_bound} - \text{lower_bound})}{(2^{(\text{compress_size} * 8)} + \text{maximum_deviation})}$$

compress_size

The size of the compressed data, in bytes.

lower_bound

The lowest acceptable value.

maximum_deviation

The absolute value of the margin of error.

upper_bound

The highest acceptable value.

For example, if the compression definition for quantization is q(1,1,100), the compression size is 1 byte, the lower boundary is 1, and the upper boundary is 100. The following equation calculates the margin of error:

$$(100-1)/256 = 0.387$$

The maximum difference between the original value and the compressed value is plus or minus 0.387.

For the linear compression type, the margin of error is equal to the maximum deviation value. For example, if the original value is 20 and the maximum deviation value is 0.1, then the compressed value is in the range 19.9 - 20.1.

If the compression definition for quantization linear boxcar is `qlb(1,1,100,100000)`, the compression size is 1 byte, the maximum deviation is 1, the lower boundary is 100, and the upper boundary is 100000. The following equation calculates the margin of error:

$$(100000-100)/256 = 390.235 + 1 = 391.235$$

The maximum difference between the original value and the compressed value is plus or minus 391.235.

Related concepts:

"Manage packed data" on page 3-37

"TimeSeries data type" on page 2-6

Related tasks:

"Example: Create and load a compressed time series" on page 3-7

Related reference:

"TSCreateIrr function" on page 7-118

JSON time series

You can create a time series that contains JSON documents. Because JSON documents do not have a rigid structure, you avoid schema changes when the structure of the data changes. You can easily load and query the data.

The advantage of having time-based data unstructured is that the schema of the **TimeSeries** data type is simple and does not need to be altered as the data changes.

A JSON time series has the following advantages over creating a column for every type of value in the **TimeSeries** data type:

MongoDB and REST API clients

You can load JSON documents directly from MongoDB and REST API clients without formatting the data.

Application compatibility

The changes that you need to make to your application when you move your data into time series is minimized because you do not need to perform schema migration.

Variable schema

If the structure of your time-based data is likely to change, storing that data as unstructured data in JSON documents prevents the need to update your schema or your application. For example, if you have sensors that monitor every machine in a factory, when you add a machine, the new sensors might collect different types of data than existing sensors.

Simplified schema

If your schema for time-based data includes more columns than each record typically uses, or your records typically contain many NULL values, you can easily load data as unstructured JSON documents. For example, if you have 50 different measurements but each sensor collects only 5 of those measurements, each record has 45 NULL values.

Storing data as JSON documents might require more storage space and result in slower queries than storing data in individual columns in the **TimeSeries** row type.

Time series definition

Although you load JSON documents into your time series table, internally the database server stores the JSON documents as BSON. Therefore, when you create a **TimeSeries** row type, you include a BSON column to store the JSON documents. You can include only one BSON column in a **TimeSeries** data type. You can also include columns that have other data types.

The maximum size of a document in a BSON column in a **TimeSeries** data type is 4 KB.

Loading and querying JSON time series

You can load JSON documents into a time series with the time series input statement, through a virtual table, or by writing a loader program.

When you query a time series that contains JSON documents, the results are automatically cast from BSON to JSON. However, when you select data from a virtual table on a JSON time series, you must cast the BSON column to JSON to view the documents.

You can aggregate individual fields within JSON documents by specifying the field name in the aggregate operation. You cannot aggregate an entire BSON column. When you run the **TSRollup**, **AggregateBy**, or **AggregateRange** function, you must cast the results to a **TimeSeries** row type that has the appropriate types of columns for the results of the aggregate operation.

Related tasks:

“Example: Create and load a time series with JSON data” on page 3-10

“Loading JSON data” on page 3-33

Calendar

Every time series is associated with a calendar. A calendar defines a set of valid times for elements in a time series.

Each calendar has a calendar pattern of time periods during which data is allowed or prohibited. Data can be recorded during on periods but cannot be recorded during off periods. The calendar pattern is based on a time interval; for example, second, minute, hour, day, or month. A start date specifies when the pattern starts.

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

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

Table 1-6. When data collection is on or off

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
OFF: data is prohibited	ON: data is allowed	ON: data is allowed	ON: data is allowed	ON: data is allowed	ON: data is allowed	OFF: data is prohibited

Table 1-6. When data collection is on or off (continued)

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
OFF: data is prohibited	ON: data is allowed	ON: data is allowed	ON: data is allowed	ON: data is allowed	ON: data is allowed	OFF: data is prohibited

For regular time series, an interval represents the range of time in which one element is allowed. You cannot enter more than one element into an interval. The value of an element persists for the length of one interval. If you query for a value in an off period, you receive an error. Intervals that are missing data are null. You can query for null elements to find which elements are missing.

For irregular time series, the interval is only relevant for designating off periods when data is not allowed. The interval size for valid periods does not affect the number of elements that you can insert into a specific time range. For example, although the smallest interval size is a second, you can enter subsecond frequency elements into an irregular time series. The value of an element persists until the next element. If you query for a value in an off period, you receive the value of the last element. Irregular time series have no null elements.

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

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

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

Related reference:

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

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

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

Time series storage

Time series data is stored in a *container* unless the data remains small enough to fit in a single row of a table. When a time series is stored in a container, the data is stored contiguously and is retrieved with a minimum number of disk reads. If you do not create time series containers before you insert time series data, the containers are created automatically as needed. You can manage storage by controlling in which dbspaces the time series data is stored and by deleting old time series data.

A container is mapped to a disk partition in a dbspace. A dbspace is a logical grouping of physical storage (chunks). The following illustration shows the architecture of containers in the database. A database usually contains multiple dbspaces. A dbspace can contain multiple containers along with tables and free space. A container can contain data for one or more sources, for example, electricity meters. The time series data for a particular source is stored on pages in time stamp order.

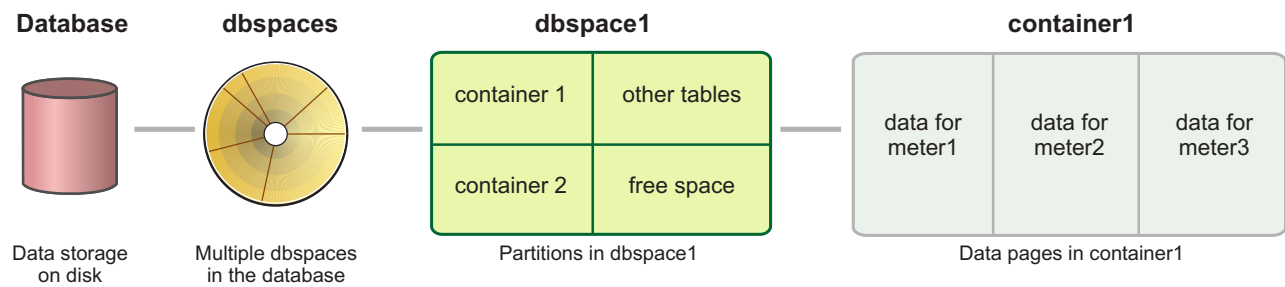


Figure 1-3. Architecture of the default configuration of a container in a database

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

All containers that are created automatically by the database server belong to the default container pool, called **autopool**. A container pool is a group of containers. You can create one or more container pools in which to include containers. You can assign containers to container pools. Alternatively, you can create your own container pool policy function.

Strategies for using multiple dbspaces

Strategies for storing time series data in multiple dbspaces depend on how the data is distributed, how data is inserted into the appropriate container, and how you delete old data.

- Time series data is stored in multiple dbspaces in the following situations:
- The table is fragmented over multiple dbspaces and containers are created automatically in the same dbspaces as the table fragments.
 - You create multiple containers in multiple dbspaces.
 - You create a rolling window container that stores time series data in multiple dbspaces.

The following table compares the different strategies.

Table 1-7. Comparison of container strategies

Strategy	Data distribution method	Inserting data	Deleting data by date range
The table is fragmented among multiple dbspaces and containers are created automatically.	The time series data is distributed among the same dbspaces as the table fragments.	The data is stored in containers in round robin order.	You run the TSContainerPurge function to delete data by date range in all containers.
You create multiple containers in multiple dbspaces.	You decide how to distribute the data. You can distribute the time series data by primary key values. You can decide on a specific set of primary key values for each container.	You specify the appropriate container name for the primary key values when you insert data.	You run the TSContainerPurge function to delete data by date range in all containers.
You create a rolling window container.	The time series data is distributed by date interval. Each date interval is stored in its own partition. Partitions are stored in multiple dbspaces.	You specify the rolling window container name when you insert data. The container controls in which dspace the data is stored.	You configure an automatic purge policy to delete data by date range, or you manually destroy partitions.

Multiple dbspaces for multiple containers

The following illustration shows multiple containers in multiple dbspaces. The time series data is distributed by the primary key value.

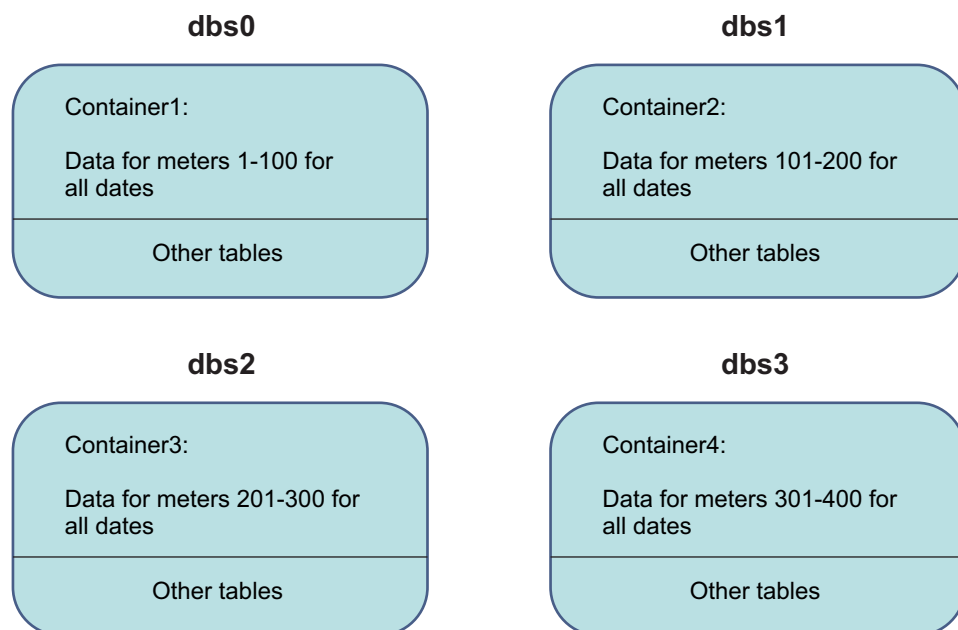


Figure 1-4. Architecture of dbspaces for multiple containers

Each container stores the data for all dates for a specific set of meter IDs. Each dspace stores a container and other tables. If the containers were created

automatically, each dbspace contains the table fragment that stores the same primary key values as the container.

Dbspaces for a rolling window container

When you create a rolling window container, you specify a time interval by which to store the data and the list of dbspaces in which to store the data. A rolling window container stores time series data by the specified time interval in separate partitions. Partitions are stored in the specified dbspaces in round robin order. The container dbspace stores the information about what data is in which partition. The following illustration shows a container that stores data in four dbspaces. The time interval for each partition is one month.

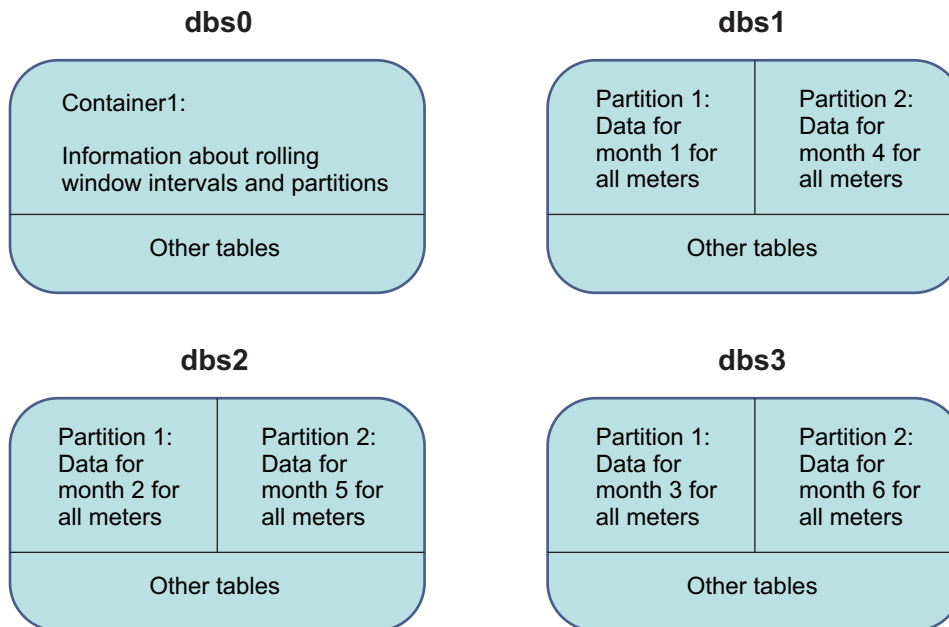


Figure 1-5. Architecture of dbspaces for a rolling window container

In this illustration, the dbspace named **dbs0** contains the container, **Container1**, and other tables. **Container1** stores the information about the time interval of the data in each partition and the location of each partition. The dbspaces named **dbs1**, **dbs2**, and **dbs3** store time series elements in partitions and other tables. Each partition stores the data for one month for all meter IDs.

Active and dormant windows

Partitions make it easy to remove old data. You can configure a rolling window container to automatically delete old data after a specified amount of data is stored. You specify the number of partitions to keep in the active window and in the dormant window. The active window contains the partitions into which you can insert data. The dormant window contains partitions that you no longer need to query, but are not yet ready to delete. The active window moves ahead in time when you insert data for the next time interval. When you insert data that is after the latest partition, a new partition is added and the active window moves ahead. When the active window exceeds the maximum number of partitions, the oldest partition is moved to the dormant window. When the dormant window exceeds the maximum number of partitions, the oldest partition is destroyed.

The following illustration shows how an active window and a dormant window grow and move over time. In this example, the maximum size of both windows is two months.

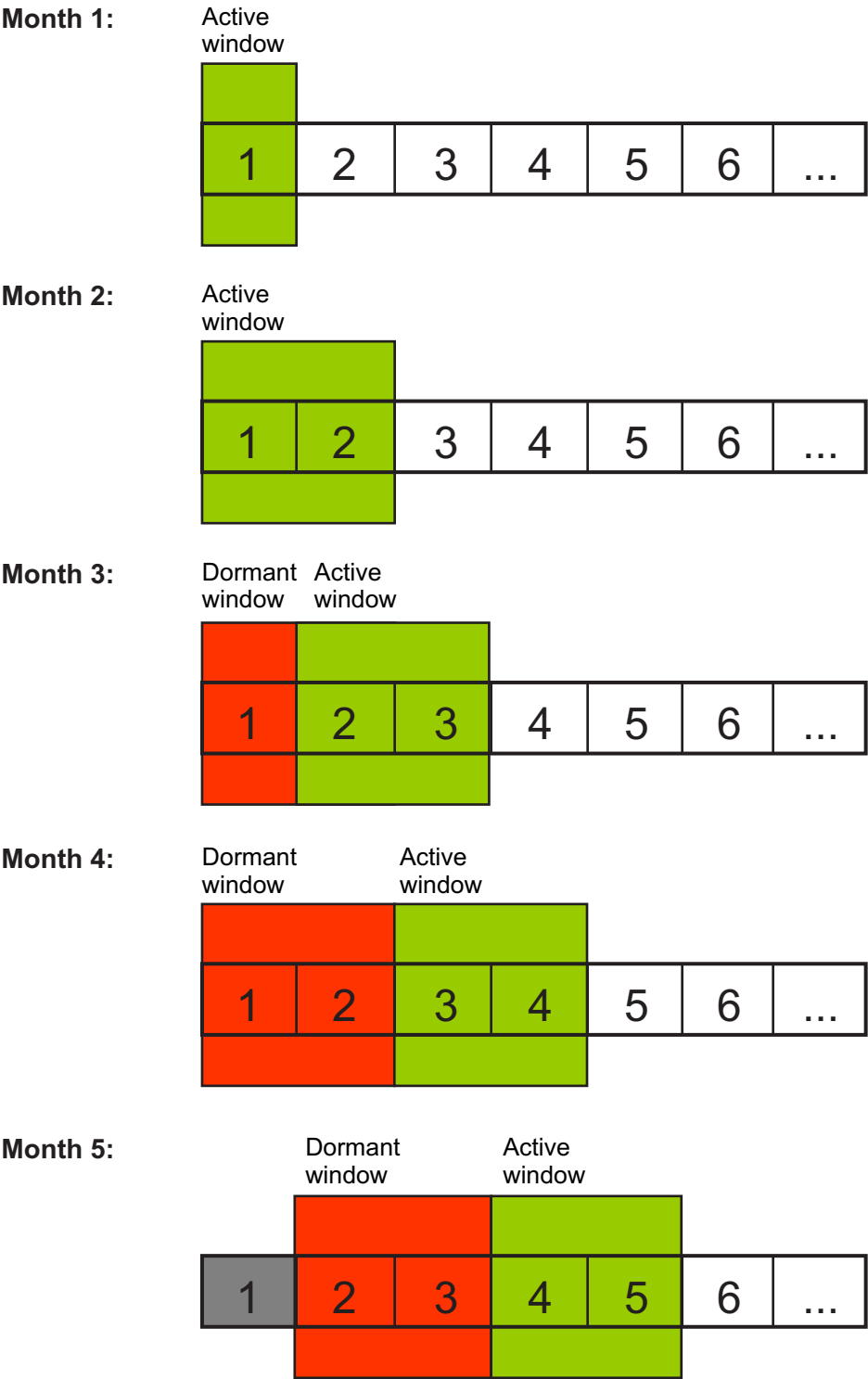


Figure 1-6. Example of an active window and a dormant window, which move over time

This illustration shows how the windows grow and move as data is inserted for each month:

- When data is added for month 1, a partition is created in the active window.
- When data is added for month 2, a second partition is created in the active window.
- When data is added for month 3, the partition for month 1 moves out of the active window and into the dormant window.
- When data is added for month 4, the active window adds a partition for month 4 and moves the partition for month 2 into the dormant window.
- When data is added for month 5, the active window and the dormant window move forward. The partition for month 1 is destroyed.

You can move partitions between the active and dormant windows and change the size of the windows.

Related concepts:

“Monitor containers” on page 3-18

Related tasks:

“Creating containers” on page 3-15

Related reference:

“Rules for rolling window containers” on page 3-16

“TSContainerUsage function” on page 7-114

“TSContainerTotalPages function” on page 7-112

“TSContainerTotalUsed function” on page 7-113

“TSContainerPctUsed function” on page 7-105

“TSContainerNElems function” on page 7-104

“Planning for data storage” on page 1-20

Getting started with the Informix TimeSeries solution

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

Planning for creating a time series

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

You can perform the necessary tasks for creating a time series in the following ways:

- Running SQL commands
- Writing an application with the time series Java class library
- Writing an application with the REST API or the MongoDB API that runs through a wire listener

The following table lists the properties of a time series.

Table 1-8. Properties of a time series

Time series property	Description	How to define
Timepoint size	For a regular time series, how long a timepoint lasts.	Define a calendar pattern.
When timepoints are valid	The times when elements can be accepted.	Define a calendar pattern.

Table 1-8. Properties of a time series (continued)

Time series property	Description	How to define
Data in the time series	The time stamp and the other data that is collected for each time stamp.	Create a TimeSeries data type.
Time series table	The table that contains the TimeSeries data type column.	Create a table with a TimeSeries column.
Location	Where the time series data is stored	Create one or more containers.
Origin	The earliest timestamp of any element	Create a time series.
Regularity	Whether the timepoints are evenly spaced or arbitrarily spaced.	Create a regular or an irregular time series.
Metadata	Optional information included with the time series that can be retrieved by routines.	Create a time series with metadata.
Hertz	The data is recorded at a regularly spaced subsecond frequency.	Create an irregular time series with a hertz value
Compression	Optional compression of time series data that is only numeric.	Create an irregular time series with compression definitions

Related concepts:



Create time series through the wire listener (JSON compatibility)

Related reference:

Chapter 3, “Create and manage a time series through SQL,” on page 3-1

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

Planning for data storage

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

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

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

Regular and irregular time series:

$$\text{space in bytes} = [\text{primary_key} + \text{index_entry} + (\text{timestamp} + \text{ts_columns} * \text{elements}) + 4 * \text{elements}] * (\text{table_rows}) + \text{B-tree_size}$$

Hertz time series:

$$\text{space in bytes} = [\text{primary_key} + \text{index_entry} + (\text{timestamp} + \text{ts_columns} * \text{records}) + (4 + 11) * \text{elements}] * (\text{table_rows}) + \text{B-tree_size}$$

Compressed time series:

$$\begin{aligned} \text{space in bytes} = & [\text{primary_key} + \text{index_entry} \\ & + (\text{timestamp} + \text{ts_columns} * \text{records}) \\ & + (4 + 11) * \text{elements}] * (\text{table_rows}) + \text{B-tree_size} \end{aligned}$$

B-tree_size

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

elements

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

For hertz time series, each element represents one second of data.

For compressed time series, each element represents approximately 4 KB of data.

index_entry

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

primary_key

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

records For hertz and compressed time series, the number of records.

table_rows

The number of rows in the time series table.

ts_columns

The size of the data types of the columns in the **TimeSeries** data type, except the timestamp column. The CHAR data type requires an additional 4 bytes when it is included in a **TimeSeries** data type.

timestamp

The size of the timestamp per element:

Regular time series = 0

Irregular time series = 11 bytes

The size of the timestamp per record:

Hertz time series = 1 byte

Compressed time series = 2 bytes

The 4 bytes per element is a slot entry.

For hertz and compressed time series, each element has an 11-byte timestamp in addition to the 1-byte or 2-byte timestamp, respectively, for each record.

The equation is a guideline. The amount of required space can be affected by other factors, such as the small amount of space that is needed for the slot table and the null bitmap for each element. The equation might underestimate the amount of required space if the row size of your time series data size is small. The maximum

number of elements that are allowed on a data page is 254. If the row size of your time series data is small, the page might contain the maximum number of elements but have unused space, especially if you are not using a 2 KB page size.

Rolling window container storage requirements

Rolling window containers allow you to limit the amount of current data to a specific time range.

Rolling window containers have two different types of partitions with different storage requirements: the container partition and the window partitions. The container partition contains information about the rolling window intervals and partitions. The window partitions store time series elements. The container partition typically requires much less space than the window partitions. To avoid allocating unnecessary space for the container partition, store the container partition and the window partitions in different dbspaces that have different extent sizes.

A rolling window container has one container partition. Use the following formula to estimate the size of the container partition:

$$\text{Space} = (\text{container_name_length} + \text{dbspace_name_length} + 48) * (\text{active_window_size} + \text{dormant_window_size}) * 2$$

active_window_size

The maximum number of partitions in the active window. If you do not intend to set a limit to the number of partitions, estimate the maximum number of partitions you expect.

container_name_length

The length of the container name, in bytes.

dbspace_name_length

The length of the dbspace name for the container partition, in bytes.

dormant_window_size

The maximum number of partitions in the dormant window. If you do not intend to set a limit to the number of partitions, estimate the maximum number of partitions you expect.

A rolling window container has multiple window partitions. You can allocate multiple dbspaces for window partitions. Use the following formula to estimate the number of partitions in each dbspace:

$$\text{Approximate number of partitions in each dbspace} = \text{CEIL}((\text{active_window_size} + \text{dormant_window_size}) / \text{number_dbspaces}) + 1$$

active_window_size

The maximum number of partitions in the active window. If you do not intend to set a limit to the number of partitions, estimate the maximum number of partitions you expect.

dormant_window_size

The maximum number of partitions in the dormant window. If you do not intend to set a limit to the number of partitions, estimate the maximum number of partitions you expect.

number_dbspaces

The number of dbspaces that are allocated for the window partitions.

Related tasks:

“Creating containers” on page 3-15

Related reference:

“Time series storage” on page 1-14

“TSContainerCreate procedure” on page 7-93

Planning for loading time series data

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

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

Table 1-9. Data loading methods

Data to load	Methods
Bulk data from a file that is created by your data collection application	<p>Use IBM Data Studio and the IBM Informix TimeSeries Plug-in for Data Studio to create a load job for a delimited file.</p> <p>Write a custom loader program that runs time series SQL routines.</p> <p>Create a virtual table and load data that is in standard relational format.</p> <p>Use the BulkLoad SQL function. The file must be formatted according to the BulkLoad function requirements.</p>
Data in a database, including databases in database servers other than Informix	<p>Use IBM Data Studio and the IBM Informix TimeSeries Plug-in for Data Studio to create a load job for data in a database.</p> <p>Write a custom loader program that runs time series SQL routines.</p>
Alter or add one or more elements to edit incorrect data or insert missing values	<p>Use the InsElem SQL function to insert an element or the PutElem SQL function to update an element. Use the InsSet SQL function to insert multiple elements or the PutSet SQL function to update multiple elements.</p> <p>Create a virtual table and use a standard SQL INSERT statement. You can add or update elements.</p>

Related concepts:

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

“IBM Informix TimeSeries Plug-in for Data Studio” on page 3-27

Related tasks:

“Loading data from a file into a virtual table” on page 3-34

“Writing a loader program” on page 3-31

Related reference:

“Load data with the BulkLoad function” on page 3-35

“Load small amounts of data with SQL functions” on page 3-36

Planning for replication of time series data

You can replicate time series data with high-availability clusters and Enterprise Replication. Review the restrictions and requirements before you set up replication.

Restrictions

You cannot replicate time series data with high-availability secondary servers that allow updates.

You cannot replicate time series data with the Change Data Capture API.

High-availability clusters

You can replicate time series data between the following types of high-availability clusters that have read-only secondary servers:

- High-Availability Data Replication (HDR)
- Shared-disk secondary servers
- Remote stand-alone secondary servers


High-availability clusters do not require any prerequisites or have any other restrictions for replicating time series data.

Because some time series calendar and container information is kept in memory, stop replication before you drop and then re-create your calendar or container definitions with the same names but different definitions.

Enterprise Replication

You can replicate time series data with Enterprise Replication. You must follow the requirements for setting up Enterprise Replication for **TimeSeries** data types.

Related concepts:

 Replication of TimeSeries data types (Enterprise Replication Guide)

 High-availability cluster configuration (Administrator's Guide)

Related tasks:

“Creating containers” on page 3-15

Planning for accessing time series data

You use SQL routines, Java classes and methods, and C API routines to access and manipulate time series data directly. You can also transform time series data into virtual tables that you can query with standard SQL statements. You can create a data mart for time series data to query on other dimensions than time.

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

Use TimeSeries SQL routines, Java classes and methods and API routines, and API routines that are written in C to perform the following types of operations to access or manipulate time series data:

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


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

If you want to query time series data on dimensions other than time, such as customer or location, or run analytic functions, you can create data marts with Informix Warehouse Accelerator. Informix Warehouse Accelerator can efficiently run queries on the typically large amounts of time series data. You can control the time range of time series values that you load into the accelerator.

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

Related concepts:

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

 Data marts for time series data (Informix Warehouse Accelerator Guide)

Related reference:

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

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

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

“TSToXML function” on page 7-154

Hardware and software requirements

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

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

Installing the IBM Informix TimeSeries Plug-in for Data Studio

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

IBM Data Studio version or IBM Optim Developer Studio, must be installed and running. IBM Data Studio is included with the Informix product.

To install the TimeSeries plug-in:

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

Related concepts:

"IBM Informix TimeSeries Plug-in for Data Studio" on page 3-27

Database requirements for time series data

To implement the Informix TimeSeries solution, the Scheduler must be running and the database must conform to requirements.


The Scheduler must be running in the database server. If the Scheduler is not running when you create the **TimeSeries** data type or run a time series routine, a message that the data type is not found or the routine cannot be resolved is returned.

The database that contains the time series data must meet the following requirements:

- The database must be logged.
- The database must not be defined as an ANSI database.
- Table and column names cannot be delimited identifiers. The DELIMIDENT environment variable must be not set or set to n.

If you attempt to create a **TimeSeries** data type or run a time series routine in an unlogged or ANSI database, the message DataBlade registration failed is printed in the online message log.

Related reference:

 scheduler argument: Stop or start the scheduler (SQL administration API) (Administrator's Reference)

 DELIMIDENT environment variable (SQL Reference)

 ondblog: Change Logging Mode (Administrator's Reference)

SQL restrictions for time series data

Some SQL statements cannot operate on time series data.

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

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

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

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

Time series global language support

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

Datetime data

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

`"yyyy-mm-dd hh:mm:ss:ffffff"`

<i>yyyy</i>	Year, expressed in digits
<i>mo</i>	Month of year, expressed in digits
<i>dd</i>	Day of month, expressed in digits
<i>hh</i>	Hour of day, expressed in digits
<i>mm</i>	Minute of hour, expressed in digits
<i>ss</i>	Seconds of minute, in digits
<i>fffff</i>	Fraction of a second, in digits

Character data

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

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

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

Numeric data

Floating point data must use the default U.S. format:

- The ASCII period (.) is the decimal separator.
- The ASCII plus (+) and minus (-) signs must be used.

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

Sample smart meter data

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

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

Customer_ts_data

Contains customer numbers and location references.

ts_data_location

Contains spatial location information.

ts_data

Contains location references and smart meter time series data.

ts_data_v

Contains the data in the **ts_data** table in relational format as a virtual table.

The time series data is stored in a container named **raw_container**.

Related concepts:

 dbaccessdemo command: Create demonstration databases (DB-Access Guide)

Related reference:

 The stores_demo Database Map (SQL Reference)

Setting up stock data examples

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

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

1. Set the following environment variables:

- MACHINE=*machine*
- PROD_VERSION=*version*
- USERFUNCDIR=\$INFORMIXDIR/extend/TimeSeries.*version*/examples

The *version* is the internal TimeSeries version number, for example 5.00.UC1. Check the installation directory for the correct version number. The *machine* is the name of the operating system, as listed in the \$INFORMIXDIR/inc1/dbdk/makeinc file, for example, linux.

2. Run the **examples_setup.sql** command from the \$INFORMIXDIR/extend/TimeSeries.*version*/examples directory: make -f Makefile
MY_DATABASE=*dbname* The *dbname* is the name of a database.

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

Chapter 2. Data types and system tables

Specialized data types and system tables handle time series data.

You create time series objects that have the following data types:

- Calendar
- CalendarPattern
- TimeSeries

You can also create a user-defined data type to store time series metadata. Other time series data types store information in the time series system tables.

The time series system tables are created in the database in which you create time series objects.

Important: Do not delete time series system tables unless you do not have any time series data.

Related concepts:

“Calendar” on page 1-13

CalendarPattern data type

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

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

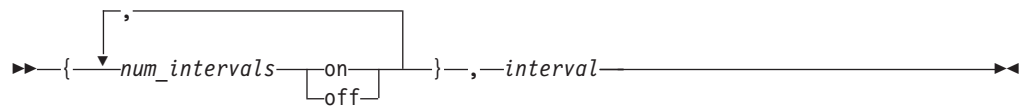


Table 2-1. CalendarPattern data type parameter values

Value	Description
<i>interval</i>	<p>For regular time series, defines the size of one element. One of the following interval names:</p> <ul style="list-style-type: none"> • Second • Minute • Hour • Day • Week • Month • Year <p>For irregular time series and hertz time series, the interval is significant only in defining on and off periods. The interval does not determine the number of elements. For hertz time series, the effective interval for recording data is the value of the <i>hertz</i> parameter when the time series is created.</p>
<i>num_intervals</i>	<p>A positive integer that represents the number of interval units. Interval units are either valid intervals for time series data, if followed by on, or invalid intervals for time series data, if followed by off. The maximum number of interval units, either on or off, in a calendar pattern is 2035. Internal calculations take longer to perform if you use a long calendar pattern.</p> <p>For compressed time series, the off keyword is not allowed.</p>

Usage

The information inside the braces is the pattern specification. The pattern specification has one or more elements that consist of *n*, the number of interval units, and either **on** or **off**, to signify valid or invalid intervals. Elements are separated by commas. The information after the braces is the interval size.

The calendar pattern length is how many intervals before the calendar pattern starts over; after all timepoints in the pattern specification are exhausted, the pattern is repeated. For this reason, a weekly calendar pattern with daily intervals must contain exactly seven intervals and a daily calendar pattern with hourly intervals must contain exactly 24 intervals. If your calendar pattern length is not correct, your time series data might not match your requirements. For example, the pattern {1 off, 4 on, 1 off} appears to repeat every week, but the pattern repeats every six days because there are only six intervals. When the calendar pattern begins is specified by the calendar pattern start date.

Calendars that have the same calendar pattern length but different interval sizes are not equivalent. For example, a calendar can 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 syntax for this calendar pattern is:

```
INSERT INTO CalendarPatterns
VALUES('workweek_day',
      '{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:

```
INSERT INTO CalendarPatterns
VALUES('workweek_hour',
      '{ 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. However, the number of allowed records for regular time series and the on and off periods are different between the calendar patterns. The **workweek_day** calendar pattern allows a maximum of five records per week for regular time series and prohibits records on weekends. The **workweek_hour** calendar pattern allows 45 records per week for regular time series and prohibits records on weekends and 15 hours per day. For irregular time series, the number of on periods and the number of records do not correspond.

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

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

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

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

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

Examples

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

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

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

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

The following statement creates a pattern that is named **fourday_day** that has a weekly pattern of four days on and three days off:

```
INSERT INTO CalendarPatterns
VALUES('fourday_day',
      '{1 off, 4 on, 2 off}, day');
```

Related concepts:

“Calendar data type” on page 2-4

“GetInterval function” on page 7-55

Table 2-2. Calendar data type parameter values (continued)

Value	Data type	Description
<i>pattern_date</i>	DATETIME YEAR TO FRACTION(5)	<p>Defines when the calendar pattern starts.</p> <p>If both the calendar start date and the pattern start date are included, the pattern start date must be the same as or later than the calendar start data by a number of intervals that is less than or equal to the number of interval lengths in the pattern length.</p> <p>If you do not specify a calendar pattern start date, the calendar start date is used.</p>
<i>pattern_name</i>	VARCHAR	Name of calendar pattern to use from CalendarPatterns table.

Usage

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

Set the calendar start date and calendar pattern start dates at the logical beginning of an interval. For example, if the interval size is a day, specify a start date time of midnight.

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

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

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

Examples

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

```
INSERT INTO CalendarTable(c_name, c_calendar)
VALUES ('weekcal',
       'startdate(2011-01-02 00:00:00.000000),
       pattstart(2011-01-02 00:00:00.000000),
       pattname(workweek_day)');
```

This calendar starts on 2011-01-02 and uses a pattern named **workweek_day**.

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

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

Value	Description
<i>data_type</i>	<p>Can be any data type except the following data types:</p> <ul style="list-style-type: none"> SERIAL, SERIAL8, or BIGSERIAL data types Types that have Assign or Destroy functions assigned to them, including large object types and some user-defined types JSON <p>A hertz time series must have columns of only the following data types: SMALLINT, INT, BIGINT, SMALLFLOAT, FLOAT, DATE, INT8, CHAR, VARCHAR, NCHAR, NVCHAR, LVARCHAR, DATETIME, DECIMAL, and MONEY.</p> <p>A compressed time series must have only the following data types: SMALLINT, INTEGER, BIGINT, SMALLFLOAT, and FLOAT.</p> <p>You can include only one BSON column. A BSON document cannot exceed 4 KB in size.</p>
<i>subtype_name</i>	<p>The name of the TimeSeries subtype. Can be a maximum of 128 bytes.</p> <p>Must follow the Identifier syntax. For more information, see Identifier (SQL Syntax).</p>
<i>timestamp_field</i>	<p>The name of the field that contains the time stamp. Must be unique for the row data type.</p> <p>Must follow the Identifier syntax. For more information, see Identifier (SQL Syntax).</p>

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

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

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

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

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

```
update statistics high for table tsinstancetable;
```

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

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

Related concepts:

“Hertz time series” on page 1-8

“Compressed numeric time series” on page 1-10

“TimeSeries data type technical overview” on page 1-5

Related tasks:

“Creating a TimeSeries subtype” on page 3-13

Related reference:

“Create the database table” on page 3-14

Time series return types

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

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

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

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

CalendarPatterns table

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

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

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

Related tasks:

“Defining a calendar” on page 3-12

Related reference:

“CalendarPattern data type” on page 2-1

CalendarTable table

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

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

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

Table 2-4. The CalendarTable table

Column name	Data type	Description
c_version	INTEGER	Internal. The version of the calendar. Currently, only version 0 is supported.

Table 2-4. The *CalendarTable* table (continued)

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

Related concepts:

“Calendar data type” on page 2-4

Related tasks:

“Defining a calendar” on page 3-12

Related reference:

“Predefined calendars” on page 3-13

TSContainerTable table

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

Table 2-5. The columns in the *TSContainerTable* table

Column name	Data type	Description
name	VARCHAR(128,1)	The name of the container of the time series. The primary key of the table. Containers that are created automatically are named autopool <i>nnnnnnnn</i> , where <i>n</i> is a positive integer eight digits long with leading zeros.
subtype	VARCHAR(128,1)	The name of the time series subtype.
partitionDesc	tsPartitionDesc_t	The description of the partition for the container.
flags	INTEGER	Stores flags to indicate: <ul style="list-style-type: none"> • Whether the container is empty and always was empty. • Whether the time series is regular or irregular. • Whether the container is enabled for Enterprise Replication. • Whether the container is a rolling window container.
pool	VARCHAR(128,1) DEFAULT NULL	The name of the container pool to which the container belongs. NULL indicates that the container does not belong to a container pool. The default container pool is named autopool .

The **TSContainerTable** table is managed by the database server. Do not modify it directly. Rows in this table are automatically inserted or deleted when containers are created or destroyed.

You can create or destroy containers by using the **TSContainerCreate** and **TSContainerDestroy** procedures, which insert and delete rows in the **TSContainerTable** table.

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

```
SELECT NAME FROM TSContainerTable;
```

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

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

Related concepts:

“Monitor containers” on page 3-18

Related reference:

“TSContainerWindowTable”

“TSContainerCreate procedure” on page 7-93

“TSContainerDestroy procedure” on page 7-98

TSContainerWindowTable

The **TSContainerWindowTable** table has one row for each rolling window container.

Two virtual tables are based on the **TSContainerWindowTable** table. The virtual tables contain a row for each partition:

- The **TSContainerUsageActiveWindowVTI** contains information about the partitions in the active window.
- The **TSContainerUsageDormantWindowVTI** contains information about the partitions in the dormant window.

Table 2-6. The columns in the TSContainerWindowTable table

Column name	Data type	Description
name	VARCHAR(128)	The name of the rolling window container.
windowinterval	VARCHAR(8)	The size of the interval of each partition: year, month, week, or day.
activewindowsize	INTEGER	The number of partitions in the active window.
dormantwindowsize	INTEGER	The number of partitions in the dormant window.
windowspaces	LVARCHAR(4096)	The list of dbspaces in which partitions are stored.
activewindow	TimeSeries(TSContainerWindow_r)	A row type that contains interval information for the partitions that are in the active window. To view the starting timestamp and location information about the partitions, query the TSContainerUsageActiveWindowVTI table.

Table 2-6. The columns in the *TSContainerWindowTable* table (continued)

Column name	Data type	Description
dormantwindow	TimeSeries(TSContainerWindow_r)	A row type that contains interval information for the partitions that are in the dormant window. To view the starting timestamp and location information about the partitions, query the TSContainerUsageDormantWindowVTI table.
container_size	INTEGER	The first extent size of the partitions.
container_grow	INTEGER	The next extent size of the partitions.
windowcontrol	INTEGER	A flag that indicates how active partitions are handled when they are manually or automatically detached: <ul style="list-style-type: none"> • 0 = Default. Partitions are not automatically destroyed if they cannot fit into the dormant window. • 1 = Partitions are automatically destroyed if they cannot fit into the dormant window.

Related concepts:

“TSContainerTable table” on page 2-9

“Monitor containers” on page 3-18

TSContainerUsageActiveWindowVTI Table

The **TSContainerUsageActiveWindowVTI** table is a virtual table that is based on the **TSContainerWindowTable** table. The **TSContainerUsageActiveWindowVTI** table has one row for each partition in the active window of a rolling window container.

Table 2-7. The columns in the *TSContainerUsageActiveWindowVTI* table

Column name	Data type	Description
name	VARCHAR(128)	The name of the rolling window container.
windowinterval	VARCHAR(8)	The size of the interval of each partition: year, month, week, or day.
activewindowsize	INTEGER	The number of partitions in the active window.
dormantwindowsize	INTEGER	The number of partitions in the dormant window.
windowspaces	LVARCHAR(4096)	The list of dbspaces in which partitions are stored.
tstamp	DATETIME YEAR TO FRACTION(5)	The starting timestamp of the time range for the partition.
partition	tsPartitionDesc_t	The description of the partition.
container_size	INTEGER	The first extent size of the partitions.
container_grow	INTEGER	The next extent size of the partitions.
windowcontrol	INTEGER	A flag that indicates how active partitions are handled when they are manually or automatically detached: <ul style="list-style-type: none"> • 0 = Default. Partitions are not automatically destroyed if they cannot fit into the dormant window. • 1 = Partitions are automatically destroyed if they cannot fit into the dormant window.

Related concepts:

“Monitor containers” on page 3-18

TSContainerUsageDormantWindowVTI Table

The **TSContainerUsageDormantWindowVTI** table is a virtual table that is based on the **TSContainerWindowTable** table. The

TSContainerUsageDormantWindowVTI table has one row for each partition in the dormant window of a rolling window container.

Table 2-8. The columns in the TSContainerUsageDormantWindowVTI table

Column name	Data type	Description
name	VARCHAR(128)	The name of the rolling window container.
windowinterval	VARCHAR(8)	The size of the interval of each partition: year, month, week, or day.
activewindowsize	INTEGER	The number of partitions in the active window.
dormantwindowsize	INTEGER	The number of partitions in the dormant window.
windowspaces	LVARCHAR(4096)	The list of dbspaces in which partitions are stored.
tstamp	DATETIME YEAR TO FRACTION(5)	The starting timestamp of the time range for the partition.
partition	tsPartitionDesc_t	The description of the partition.
container_size	INTEGER	The first extent size of the partitions.
container_grow	INTEGER	The next extent size of the partitions.
windowcontrol	INTEGER	A flag that indicates how active partitions are handled when they are manually or automatically detached: <ul style="list-style-type: none">• 0 = Default. Partitions are not automatically destroyed if they cannot fit into the dormant window.• 1 = Partitions are automatically destroyed if they cannot fit into the dormant window.

Related concepts:

“Monitor containers” on page 3-18

TSInstanceTable table

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

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

Table 2-9. The columns in the *TSInstanceTable* table

Column name	Data type	Description
id	BIGSERIAL	<p>The serial number of the time series and the primary key for this table. You can use the InstanceId function to return this number.</p> <p>The instance ID is a required argument in some time series routines.</p> <p>The instance ID is also used to order results if an SQL query includes an ORDER BY clause on a TimeSeries column.</p>
cal_id	INTEGER	The identification of the CalendarTable row for the time series.
flags	SMALLINT	<p>Stores various flags for the time series.</p> <ul style="list-style-type: none"> • 0x01 = TSFLAGS_IRR. The time series is irregular. • 0x10 = TSFLAGS_PACKED. The time series contains packed elements. • 0x20 = TSFLAGS_COMPRESSED. The time series is compressed. • 0x40 = TSFLAGS_HERTZ. The time series contains hertz data.
vers	SMALLINT	The version of the time series.
container_name	VARCHAR(128,1)	The name of the container of the time series. This is a reference to the primary key of the TSContainerTable table.
ref_count	INTEGER	The number of different references to the same time series instance.

The **TSInstanceTable** table is managed by the database server. Do not modify it directly. Rows in this table are automatically inserted or deleted when large time series are created or destroyed.

Related reference:

“InstanceId function” on page 7-71

“TSContainerCreate procedure” on page 7-93

“TSContainerSetPool procedure” on page 7-111

“TSContainerDestroy procedure” on page 7-98

Chapter 3. Create and manage a time series through SQL

Before you can load time series data into the database, you must configure database objects specific to your time series. You can manage data storage and remove data as necessary. You can run SQL statements to create and manage time series.

To create and load a time series:

1. Create a calendar or choose a predefined calendar.
2. Create a time series column.
3. If necessary, create containers.
4. Create the time series.
5. Load data into the time series.

Related concepts:

“Planning for creating a time series” on page 1-19

Example: Create and load a regular time series

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

Prerequisites:

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

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

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

Time series property	Definition
Timepoint size	15 minutes
When timepoints are valid	Every 15 minutes with no invalid times
Data in the time series	The following data: <ul style="list-style-type: none">• Timestamp• A decimal value that represents electricity usage
Time series table	The following columns: <ul style="list-style-type: none">• A meter ID column of type BIGINT• A TimeSeries data type column
Origin	All meter IDs have an origin of 2010-11-10 00:00:00.00000
Regularity	Regular

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

Time series property	Definition
Metadata	No metadata
Amount of storage space	Approximately 1 MB (8640 timepoints for each of the 28 rows)
Where to store the data	In an automatically created container in the same dbspace as the stores_demo database, which is in the root dbspace by default
How to load the data	The TimeSeries plug-in
How to access the data	A virtual table

Creating a TimeSeries data type and table

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

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

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

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

The **timestamp** column contains the time of the meter reading and the **data** column contains the reading value.

2. Create a time series table named **my_ts_data** by running the following SQL statement:

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

Related tasks:

“Creating a TimeSeries subtype” on page 3-13

Related reference:

“Create the database table” on page 3-14

Creating regular, empty time series

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

To create regular, empty time series:

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

```
UNLOAD TO "my_meter_id.unl" SELECT loc_esi_id FROM ts_data;
```

2. Create a temporary table named **my_tmp** and load the meter IDs into it by running the following SQL statements:

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

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

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

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

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

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

Related reference:

“Time series input function” on page 3-24

Creating the data load file

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

To create the data load file:

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

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

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

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

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

Related reference:

“TSCreateVirtualTab procedure” on page 4-5

Loading the time series data

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

To load time series data:

1. If you are using Data Studio or Optim Developer Studio on a different computer, move the **\$INFORMIXDIR\my_meter_data.unl** file to that computer and start Data Studio or Optim Developer Studio.
2. From the main menu, choose **Help > Cheat Sheets**, expand the **TimeSeries Data** category, choose **Loading from a File**, and click **OK**.
3. Open the TimeSeries perspective.

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

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

Related concepts:

"IBM Informix TimeSeries Plug-in for Data Studio" on page 3-27

Accessing time series data through a virtual table

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

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

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

```
EXECUTE PROCEDURE TSCreateVirtualTab("my_vt2", "my_ts_data",
  "calendar(ts_15min), origin(2010-11-10 00:00:00.000000)");
```

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

```
SELECT * FROM my_vt2 WHERE timestamp = "2010-11-10 00:00:00.000000";
```

Related reference:

"TSCreateVirtualTab procedure" on page 4-5

Example: Create and load a hertz time series

This example shows how to create, load, and query a time series that stores hertz data.

In this example, you create a time series that contains hertz data that is recorded 50 times a second. You can create the time series table in any database that you choose. The following table lists the time series properties that are used in this example.

Table 3-2. Time series properties used in this example

Time series property	Definition
Element size	1 second
When elements are valid	Always
Data in the time series	The following data: <ul style="list-style-type: none">• Timestamp• A SMALLINT value• An INTEGER value• A BIGINT value
Time series table	The following columns: <ul style="list-style-type: none">• An ID column of type INTEGER• A TimeSeries data type column
Origin	2014-01-01 00:00:00.00000
Regularity	Irregular
Hertz	50 records per second
Metadata	No metadata
Where to store the data	In a container that you create
How to load the data	The InsElem function
How to access the data	A virtual table

To create, load, and view a hertz time series:

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

```
CREATE ROW TYPE ts_data_h(  
    tstamp      datetime year to fraction(5),  
    tssmallint  smallint,  
    tsint       int,  
    tsbigint    bigint  
);
```

You can include only certain data types in your **TimeSeries** subtype.

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

```
CREATE TABLE IF NOT EXISTS tstable_h(  
    id int not null primary key,  
    ts timeseries(ts_data_h)  
) LOCK MODE ROW;
```

3. Create a container that is named **container_h** by running the following SQL statement:

```
EXECUTE PROCEDURE  
    TSContainerCreate('container_h', 'rootdbs', 'ts_data_h', 512, 512);
```

You can choose to create the container in a different dbspace than the root dbspace.

4. Create a calendar by running the following SQL statement:

```
INSERT INTO CalendarTable(c_name, c_calendar)
VALUES('ts_1sec',
'startdate(2014-01-01 00:00:00.00000),
pattern({1 on}, second)');
```

You cannot specify a subsecond interval for a calendar, however, the hertz time series definition overrides the calendar interval.

5. Create a hertz time series with a hertz value of 50 by running the following SQL statement in an explicit transaction:

```
BEGIN;
Started transaction.

INSERT INTO tstable_h VALUES(50,
TSCreateIrr('ts_1sec', '2014-01-01 00:00:00.00000', 0, 50, 0, 'container_h')
);
1 row(s) inserted.

COMMIT;
```

The *threshold* and *nelems* parameters must be set to 0.

6. Insert five records for the same second into the time series by running the following SQL statements:

```
BEGIN;
Started transaction.

UPDATE tstable_h SET ts = InsElem(ts, row('2014-01-01 00:00:00.00000',
1, 201, 99991)::ts_data_h)
WHERE id = 50;
1 row(s) updated.

UPDATE tstable_h SET ts = InsElem(ts, row('2014-01-01 00:00:00.02000',
2, 202, 99992)::ts_data_h)
WHERE id = 50;
1 row(s) updated.

UPDATE tstable_h SET ts = InsElem(ts, row('2014-01-01 00:00:00.04000',
3, 203, 99993)::ts_data_h)
WHERE id = 50;
1 row(s) updated.

UPDATE tstable_h SET ts = InsElem(ts, row('2014-01-01 00:00:00.06000',
4, 204, 99994)::ts_data_h)
WHERE id = 50;
1 row(s) updated.

UPDATE tstable_h SET ts = InsElem(ts, row('2014-01-01 00:00:00.08000',
5, 205, 99995)::ts_data_h)
WHERE id = 50;
1 row(s) updated.

COMMIT;
Data committed.
```

You must insert records in chronological order.

7. Create a virtual table that is named **vt_tstable_h_h** that is based on the hertz time series by running the following SQL statement:


```
EXECUTE PROCEDURE
  TSCreateVirtualTab('vt_tstable_h_h', 'tstable_h', 4096, 'ts');
```

8. Query the virtual table to view the hertz data by running the following SQL statement:

```
SELECT * FROM vt_tstable_h_h;
```

```
id          50
tstamp      2014-01-01 00:00:00.00000
tssmallint  1
tsint       201
tsbigint     99991
```

```
id          50
tstamp      2014-01-01 00:00:00.02000
tssmallint  2
tsint       202
tsbigint     99992
```

```
id          50
tstamp      2014-01-01 00:00:00.04000
tssmallint  3
tsint       203
tsbigint     99993
```

```
id          50
tstamp      2014-01-01 00:00:00.06000
tssmallint  4
tsint       204
tsbigint     99994
```

```
id          50
tstamp      2014-01-01 00:00:00.08000
tssmallint  5
tsint       205
tsbigint     99995
```

5 row(s) retrieved.

Related concepts:

“TimeSeries data type” on page 2-6

“Hertz time series” on page 1-8

Related reference:

“Create the database table” on page 3-14

“TSContainerCreate procedure” on page 7-93

“TSCreateIrr function” on page 7-118

“InsElem function” on page 7-69

“TSCreateVirtualTab procedure” on page 4-5

Example: Create and load a compressed time series

This example shows how to create, load, and query a time series that stores compressed numeric data.

In this example, you create a time series that contains numeric data that is compressed. You can create the time series table in any database that you choose. The following table lists the time series properties that are used in this example.

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

Time series property	Definition
Calendar interval	Second
When elements are valid	Always
Data in the time series	The following data: <ul style="list-style-type: none"> • Timestamp • Three columns with INTEGER values
Time series table	The following columns: <ul style="list-style-type: none"> • An ID column of type INTEGER • A TimeSeries data type column
Origin	2014-01-01 00:00:00.00000
Regularity	Irregular
Compression	Three different compression definitions
Metadata	No metadata
Where to store the data	In a container that you create
How to load the data	The InsElem function
How to access the data	A virtual table

To create, load, and view a compressed time series:

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

```
CREATE ROW TYPE ts_data_c(
    tstamp    datetime year to fraction(5),
    ts1       int,
    ts2       int,
    ts3       int
);
```

You can include only certain numeric data types in your **TimeSeries** subtype.

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

```
CREATE TABLE IF NOT EXISTS tstable_c(
    id int not null primary key,
    ts timeseries(ts_data_c)
) LOCK MODE ROW;
```

3. Create a container that is named **container_c** by running the following SQL statement:

```
EXECUTE PROCEDURE
    TSContainerCreate('container_c', 'rootdbs', 'ts_data_c', 512, 512);
```

You can choose to create the container in a different dbspace than the root dbspace.

4. Create a calendar by running the following SQL statement:

```
INSERT INTO CalendarTable(c_name, c_calendar)
VALUES('ts_1sec',
    'startdate(2014-01-01 00:00:00.00000),
    pattern({1 on}, second)');
```

You cannot include off periods in the calendar for a compressed time series.

5. Create a compressed time series with compression definitions for the three data columns in the **ts_data_c** data type by running the following SQL statement in an explicit transaction:

```
BEGIN;
Started transaction.

INSERT INTO tstable_c VALUES(50,
    TSCreateIrr('ts_1sec', '2014-01-01 00:00:00.00000', 'container_c',
        'q(2,1,100),lb(20),ls(20)'));
1 row(s) inserted.

COMMIT;
```

The first INTEGER column has a compression type of quantization with a compression size of 2 bytes, a lower bound of 1 and an upper bound of 100. The second INTEGER column has a compression type of linear boxcar and a maximum deviation of 20. The third INTEGER column has a compression type of linear swing door and a maximum deviation of 20.

6. Insert five records for the same element into the time series by running the following SQL statements:

```
BEGIN;
Started transaction.

UPDATE tstable_c SET ts = InsElem(ts, row('2014-01-01 00:00:00.00000',
    1, 201, 99991)::ts_data_c) WHERE id = 50;
1 row(s) updated.

UPDATE tstable_c SET ts = InsElem(ts, row('2014-01-01 00:00:01.00000',
    2, 202, 99992)::ts_data_c) WHERE id = 50;
1 row(s) updated.

UPDATE tstable_c SET ts = InsElem(ts, row('2014-01-01 00:00:02.00000',
    3, 203, 99993)::ts_data_c) WHERE id = 50;
1 row(s) updated.

UPDATE tstable_c SET ts = InsElem(ts, row('2014-01-01 00:00:03.00000',
    4, 204, 99994)::ts_data_c) WHERE id = 50;
1 row(s) updated.

UPDATE tstable_c SET ts = InsElem(ts, row('2014-01-01 00:00:04.00000',
    5, 205, 99995)::ts_data_c) WHERE id = 50;
1 row(s) updated.

COMMIT;
Data committed.
```

You must insert records in chronological order.

7. Create a virtual table that is named **vt_tstable_c** that is based on the compressed time series by running the following SQL statement:

```
EXECUTE PROCEDURE
    TSCreateVirtualTab('vt_tstable_c', 'tstable_c', 4096, 'ts');
```

8. Query the virtual table to view the compressed data by running the following SQL statement:

```
SELECT * FROM vt_tstable_c;
```

id	tstamp	ts1	ts2	ts3
50	2014-01-01 00:00:00.00000	1	201	99991
50	2014-01-01 00:00:01.00000	2	202	99992
50	2014-01-01 00:00:02.00000	3	203	99993

50	2014-01-01 00:00:03.00000	4	204	99994
50	2014-01-01 00:00:04.00000	5	205	99995

5 row(s) retrieved.

Related concepts:

“TimeSeries data type” on page 2-6

“Compressed numeric time series” on page 1-10

Related reference:

“Create the database table” on page 3-14

“TSContainerCreate procedure” on page 7-93

“TSCreateIrr function” on page 7-118

“InsElem function” on page 7-69

“TSCreateVirtualTab procedure” on page 4-5

Example: Create and load a time series with JSON data

This example shows how to create, load, and query a time series that stores JSON data.

In this example, you create a time series that contains meter readings in JSON documents. Readings are taken every 15 minutes. The following table lists the time series properties that are used in this example.

Table 3-4. Time series properties used in this example

Time series property	Definition
Timepoint size	15 minutes
When timepoints are valid	Every 15 minutes with no invalid times
Data in the time series	<ul style="list-style-type: none"> Timestamp JSON documents that are stored in a BSON column in the TimeSeries subtype
Time series table	<ul style="list-style-type: none"> A meter ID column of type INTEGER A TimeSeries data type column
Origin	2014-01-01 00:00:00.00000
Regularity	Regular
Metadata	No metadata
Where to store the data	In a container that you create
How to load the data	Through an external table and a loader program
How to access the data	Through a virtual table

To create, load, and query a time series that contains JSON data:

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

```
CREATE ROW TYPE ts_data_j(
    timestamp    datetime year to fraction(5),
    sensor_data  BSON
);
```

Because the JSON documents are stored in the database as BSON documents, the **sensor_data** column has a BSON data type.

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

```
CREATE TABLE IF NOT EXISTS tstable_j(  
    id int not null primary key,  
    ts timeseries(ts_data_j)  
) LOCK MODE ROW;
```

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

```
EXECUTE PROCEDURE  
    TSContainerCreate('container_b', 'dbspace1', 'ts_data_j', 512, 512);
```

4. Create a time series with a JSON document by running the following SQL statement:

```
INSERT INTO tstable_j VALUES(1, 'origin(2014-01-01 00:00:00.000000),  
    calendar(ts_15min), container(container_b),  
    [{"v1":1.5, "v2":20.5}]]');
```

A predefined calendar with 15 minute intervals is specified. The JSON document contains two values.

5. Create a pipe-delimited file in any directory with the name **json.unl** that contains the time series data to load:

```
1|2014-01-01 00:15:00.000000|{"v1":2.0, "v2":17.4}  
1|2014-01-01 00:30:00.000000|{"v1":1.9, "v2":20.2}  
1|2014-01-01 00:45:00.000000|{"v1":1.8, "v2":19.7}
```

6. Create an external table and load it with time series data from the **json.unl** file:

```
CREATE EXTERNAL TABLE ext_tstable_j  
(  
    id          INT,  
    tstamp      DATETIME YEAR TO FRACTION(5),  
    json_doc    JSON  
)  
USING(  
    FORMAT 'DELIMITED',  
    DATAFILES  
    (  
        "DISK:path/json.unl"  
    )  
);
```

Substitute *path* with the directory for the **json.unl** file.

7. Initialize a global context and open a database session by running the **TSL_Init** function:

```
EXECUTE FUNCTION TSL_Init('tstable_j','ts');
```

8. Load the data by running the **TSL_PutSQL** function with an SQL statement that selects the data from the external table and casts the JSON column to BSON:

```
EXECUTE FUNCTION TSL_PutSQL('tstable_j|ts',  
    "SELECT id, tstamp, sensor_data::bson FROM ext_tstable_j");
```

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

```
BEGIN;  
EXECUTE FUNCTION TSL_FlushAll('tstable_j|ts');  
COMMIT WORK;
```

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

```
EXECUTE FUNCTION TSL_SessionClose('tstable_j|ts');  
EXECUTE FUNCTION TSL_Shutdown('tstable_j|ts');
```

11. Create a virtual table that is named **virt_tstable_j** by running the **TSCreateVirtualTab** procedure:
EXECUTE PROCEDURE TSCreateVirtualTab(virt_tstable_j, tstable_j);
12. View the virtual table by running a SELECT statement. Cast the **sensor_data** column to JSON so that you can view the data:
SELECT id, tstamp, sensor_data::json FROM virt_tstable_j;

(expression)	id	tstamp	sensor_data
	1	2014-01-01 00:00:00.00000	{"v1":1.5, "v2":20.5}
	1	2014-01-01 00:15:00.00000	{"v1":2.0, "v2":17.4}
	1	2014-01-01 00:30:00.00000	{"v1":1.9, "v2":20.2}
	1	2014-01-01 00:45:00.00000	{"v1":1.8, "v2":19.7}

4 row(s) retrieved.
13. Insert a row through the virtual table. You must explicitly cast the JSON data to the JSON data type, and then cast the data to the BSON data type:
INSERT INTO virt_tstable_j values(1, "2014-01-01 01:00:00.00000",
('{"v1":2.1, "v2":20.1}'::json)::bson);

Related concepts:

“JSON time series” on page 1-12

Related tasks:

“Creating a TimeSeries subtype” on page 3-13

“Writing a loader program” on page 3-31

“Loading JSON data” on page 3-33

Related reference:

“Create the database table” on page 3-14

“Time series input function” on page 3-24

“TSCreateVirtualTab procedure” on page 4-5

 CREATE EXTERNAL TABLE Statement (SQL Syntax)

Defining a calendar

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

To create a calendar:

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

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

Related concepts:

“Calendar data type” on page 2-4

“CalendarPatterns table” on page 2-8

“CalendarTable table” on page 2-8

Related reference:

“CalendarPattern data type” on page 2-1

Predefined calendars

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

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

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

The following table contains the properties of predefined calendars.

Table 3-5. Predefined calendars

Calendar name	Interval duration	Start date and time
ts_1min	Once a minute	2011-01-01 00:00:00.00000
ts_15min	Once every 15 minutes	2011-01-01 00:00:00.00000
ts_30min	Once every 30 minutes	2011-01-01 00:00:00.00000
ts_1hour	Once an hour	2011-01-01 00:00:00.00000
ts_1day	Once a day	2011-01-01 00:00:00.00000
ts_1week	Once a week	2011-01-02 00:00:00.00000
ts_1month	Once a month	2011-01-01 00:00:00.00000
ts_1year	Once a year	2011-01-01 00:00:00.00000

Related concepts:

“CalendarTable table” on page 2-8

Create a time series column

To create a time series column:

Creating a TimeSeries subtype

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

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

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

Examples

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

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

```

        low          real,
        final        real,
        vol          real
    );

```

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

```

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

```


Related concepts:

“TimeSeries data type” on page 2-6

“TimeSeries data type technical overview” on page 1-5

Related reference:

 CREATE ROW TYPE statement (SQL Syntax)

 DATETIME data type (SQL Reference)

Create the database table

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

You have the following options and restrictions when you create a table with at TimeSeries column:

- You can create the table in a dbspace that uses non-default page size.
- You cannot use delimited identifiers for table or column names.
- If you plan to replicate time series data with Enterprise Replication, the primary key column must not be an opaque data type.
- If you plan to write a loader program, the name of the table and the name of the TimeSeries column must not contain uppercase letters.
- You can fragment the table. When you fragment the table and enable PDQ, certain queries can run faster:
 - TimeSeries routines that select time series data can run in parallel. The table can be fragmented by any method.
 - Queries on a fragmented virtual table that is based on the table can run in parallel. The table must be fragmented by expression.
- You can include other options of the CREATE TABLE statement.

The basic 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)
);

```

List the data type of the TimeSeries column as **TimeSeries**(*subtype_name*), where *subtype_name* is the name of the subtype that you created.

Examples

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

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

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

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

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

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

Related concepts:

"TimeSeries data type" on page 2-6

Related reference:

"Time series routines that run in parallel" on page 7-7

 [CREATE TABLE statement \(SQL Syntax\)](#)

Creating containers

Containers are created automatically when they are needed, in the same dbspaces in which the table is stored. If you want to store your time series data in other dbspaces, you can create containers. You can move containers between container pools. You must create containers if you want to replicate time series data with Enterprise Replication or create rolling window containers.

To create a container, run the **TSContainerCreate** procedure. For Enterprise Replication, you must create containers with the same names on every replication server before you start replication.

To control whether multiple sessions write to a container at one time, run the **TSContainerLock** procedure.

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

Related concepts:

"Planning for replication of time series data" on page 1-23

Related reference:

"TSContainerCreate procedure" on page 7-93

"TSContainerDestroy procedure" on page 7-98

"TSContainerSetPool procedure" on page 7-111

"Time series storage" on page 1-14

"TSContainerPurge function" on page 7-108

"Planning for data storage" on page 1-20

“TSContainerLock procedure” on page 7-99

Rules for rolling window containers

Rolling window containers store data in partitions by date intervals and can automatically delete old data. The active window and the dormant window have a set of behaviors that affect how the data is handled and how you can interact with the data.

To create a rolling window container, run the **TSContainerCreate** function. Specify the size of a partition. Specify maximum sizes for the active and dormant windows. Enable the automatic deletion of old partitions. Specify multiple dbspaces in which to store partitions.

To change the properties or the layout of rolling window containers, run the **TSContainerManage** function. You can change the window sizes, attach or detach partitions, destroy partitions, change whether partitions are destroyed automatically, and change the extent sizes of partitions.

To take advantage of rolling windows, set the sizes of the active and dormant windows to positive integers. If you set the size of the active window to 0, the active window size is unlimited and you must manually detach partitions into the dormant window. If you set the size of the dormant window to 0, you must manually destroy partitions, regardless of whether you enable the automatic destroying of partitions.

Set the active window size to the same or larger than the time range of data that you plan to load. If the range of data is larger than the active window size, the load might fail. For example, suppose that you are storing meter data and choose an interval of one week and an active window size of four weeks. When you add a month of data for the first meter ID, five partitions are created because a month is usually longer than four weeks. The partition that stores the data for the first week of the month is moved to the dormant window. When you attempt to insert a month of data for the next meter ID, the insert fails because some of the data does not fit into the active window.

In general, set the size of the dormant window larger than the size of the active window. The dormant window provides a staging area for data that you no longer need but that you do not yet want to delete.

If you enable the automatic destroying of partitions, test your system under realistic conditions before you implement it. Consider the size of your active and dormant windows and how you insert data. For example, the active and dormant windows each have a maximum size of three intervals. If you insert data that has a timestamp of seven intervals later than the most recent original data, all of your original data is destroyed.

The following list of rules describe the behavior of the active window and dormant windows if you set both the window sizes to positive numbers. The examples that are provided assume that the active window has a monthly interval, has a maximum size of 6 partitions, and contains partitions for January, February, and March. The dormant window has a maximum size of 3 partitions.

Rules for querying, inserting, and modifying data:

- You cannot query, insert, update, or delete data from a partition in a dormant window.

- You can query, insert, update, and delete the data in the partitions that are in the active window. For example, you can insert, update, or delete an element for February 28.
- If you add data with timestamps that are after the latest partition, new partitions are created automatically in the active window. For example, you insert an element for March 2. A new partition is created for March.
- You can add data that is earlier than the oldest partition if both of the following conditions are met:
 - The data does not have timestamps that are before the origin of the rolling window container.
 - The active window has space for the new partition, which is created automatically.
 - The dormant window does not have data that has the same or a more recent timestamp than the data that is being added. For example, the dormant window is empty, and you insert an element for December 31 of the previous year. A new partition is created for December in the active window.
- If you add data that is more than one interval later than the latest partition, intermediate empty partitions are also created. Empty partitions are counted in the window size. For example, you insert an element for May 15. New partitions are created for April and May. The April partition remains empty until you insert data for April. The active window has 6 partitions.
- When you add data for a new partition and the active window is full, the oldest partition automatically moves to the dormant window. For example, you add data for July. Partitions are created automatically for June and July. The partitions for December and January are moved to the dormant window.

Rules for managing partitions:

- You can manually detach partitions from the active window to the dormant window. Any partitions older than the partition you choose to detach are also detached. For example, if you detach the March partition to the dormant window, the February partition is detached automatically. However, if the dormant window is full and partitions are not destroyed automatically, that detach operation fails. You must first manually destroy the appropriate number of partitions.
- You can attach partitions into the active window from the dormant window. The active window must have room for that partition and any partitions that belong in between that partition and the oldest partition in the active window. For example, you attach the January partition to the active window. The February partition is attached automatically to the active window.
- You can change the size of the active window. If you make the size of the active window larger, you can attach partitions from the dormant window. For example, you change the active window size to 9 and attach the December partition to the active window. The January partition also attaches to the active window. If you make the size of the active window smaller, the oldest partitions that exceed the new maximum size are detached to the dormant window. However, if the dormant window is full and partitions are not destroyed automatically, the resizing operation fails. You must first manually destroy the appropriate number of partitions.

Rules for destroying data:

- When you add data and one or more new active partitions are created so that the number of active partitions exceeds the active window size, the appropriate number of older active partitions are detached and moved to the dormant

window. How the partitions that are being moved to the dormant window are handled depends on the number of partitions that are being moved and the *window_control* parameter:

- If the number of active partitions that are being detached is less than or equal to the dormant window size, then the active partitions are moved to the dormant window. If the number of dormant partitions exceeds the dormant window size, the older partitions are destroyed.
- If the number of partitions that must be detached is greater than the dormant windows size, you specify the action taken by setting the *window_control* parameter:
 - To allow as any dormant partitions as necessary to be destroyed but prevent any active partitions from being destroyed, set the *window_control* parameter to 0. An operation fails if it requires active partitions to be destroyed.
 - To allow as many as necessary active and dormant partitions to be destroyed, set the *window_control* parameter to 1.
 - To prevent active partitions from being destroyed and limit the number of dormant partitions that can be destroyed, set the window control flag to 2 and set the *destroy_count* parameter to a positive integer.
 - To allow a specific number of active and dormant partitions to be destroyed, set the *window_control* parameter to 3 and the *destroy_count* parameter to a positive integer.
- You can manually destroy partitions in the dormant window. Any partitions that are older than the date that you specify are destroyed. For example, the dormant window contains partitions for January, February, and March. You specify to destroy partitions before March. The February and January partitions are destroyed.
- You can change the size of the dormant window. If you make it smaller, the action that is taken depends on the *window_control* parameter. The behavior is the same as when you add partitions.
- You can change the type and number of partitions that can be destroyed.

Related concepts:

"Delete time series data" on page 3-37

Related reference:

"Time series storage" on page 1-14

"TSContainerCreate procedure" on page 7-93

"TSContainerManage function" on page 7-99

Monitor containers

You can view information about time series containers, such as the properties of containers, how large containers are, and how full containers are. If you monitor the containers over time, you can predict how quickly containers fill and how much data fits into each container.

Use the IBM OpenAdmin Tool (OAT) for Informix to monitor containers. Alternatively, query system tables and run time series SQL routines.

To view information about the properties of containers, query the **TSContainerTable** table. For rolling window containers, view information about partitions in the **TSContainerUsageActiveWindowVTI** and **TSContainerUsageDormantWindowVTI** tables.

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

- The **TSContainerUsage** function returns the number of pages that contain time series data, the number of elements, and the number of pages that are allocated to the container.
- The **TSContainerTotalPages** function returns the number of pages that are allocated to the container.
- The **TSContainerTotalUsed** function returns the number of pages that contain time series data.
- The **TSContainerPctUsed** function returns what percent of the container is full.
- The **TSContainerNElems** function returns the number of time series data elements that are stored in the container.

If you specify NULL instead of a container name, the functions return information about all containers in the database. If you include wildcard characters for the MATCHES operator in the container name, the functions return information about all containers that have matching names. For example, if your containers have a naming convention, you can monitor groups of containers that have similar names. For rolling window containers, you can specify which sets of partitions to view information about.

Related concepts:

"TSContainerTable table" on page 2-9

Related reference:

"TSContainerWindowTable" on page 2-10

"TSContainerUsageDormantWindowVTI Table" on page 2-12

"TSContainerUsageActiveWindowVTI Table" on page 2-11

"TSContainerUsage function" on page 7-114

"TSContainerTotalPages function" on page 7-112

"TSContainerTotalUsed function" on page 7-113

"TSContainerPctUsed function" on page 7-105

"TSContainerNElems function" on page 7-104

"Time series storage" on page 1-14

 oncheck -pt and -pT: Display tblspaces for a Table or Fragment (Administrator's Reference)

Manage container pools

By default, containers that are automatically created are added to the default container pool, named **autopool**.

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

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

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

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

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

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

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

Example 2: Removing a container from the default container pool

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

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

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

Configuring additional container pools

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

If you want to use a container pool policy other than round-robin order, you must write the user-defined container pool policy function before you insert data into the container pool. For more information, see “User-defined container pool policy” on page 3-21.

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

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

Example

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

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

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

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

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

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

```
INSERT INTO smartmeters(meter_id,rawreadings)
VALUES('met00001','origin(2006-01-01 00:00:00.000000),
calendar(smartmeter),regular,threshold(0),
container(TSContainerPoolRoundRobin(readings)),
[(33070,-13.00,100.00,9.98e+34),
(19347,-4.00,100.00,1.007e+35),
(17782,-18.00,100.00,9.83e+34)]');
```

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

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

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

Related reference:

“TSContainerCreate procedure” on page 7-93

“TSContainerPoolRoundRobin function” on page 7-107

“TSContainerSetPool procedure” on page 7-111

User-defined container pool policy:

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

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

Syntax

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



```

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

```

PolicyName

The name of the user-defined function.

table_name

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

column_name

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

subtype

The name of the **TimeSeries** subtype.

irregular

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

user_data

Optional argument for the name of the container pool.

Description

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

Returns

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

Related reference:

“TSContainerPoolRoundRobin function” on page 7-107

Create a time series

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

The following table lists the methods for creating and populating a time series.

Table 3-6. Methods for creating time series

Task	Function
Create an empty time series	<ul style="list-style-type: none"> • TSCreate (regular time series) • TSCreateIrr (irregular time series)

Table 3-6. Methods for creating time series (continued)

Task	Function
Create a time series with metadata	<ul style="list-style-type: none"> • TSCreate with the <i>metadata</i> parameter (regular time series) • TSCreateIrr with the <i>metadata</i> parameter (irregular time series)
Create and populate a time series	<ul style="list-style-type: none"> • TSCreate with the <i>set_ts</i> argument (regular time series) • TSCreateIrr with the <i>set_ts</i> argument (irregular time series) • The time series input function • The output of a function
Create a hertz time series	<ul style="list-style-type: none"> • TSCreateIrr with the <i>hertz</i> parameter • The time series input function
Create a compressed time series	<ul style="list-style-type: none"> • TSCreateIrr with the <i>compression</i> parameter • The time series input function

Related concepts:

“TimeSeries data type” on page 2-6

“Regular time series” on page 1-7

“Irregular time series” on page 1-7

Related reference:

“TSCreate function” on page 7-116

“TSCreateIrr function” on page 7-118

Creating a time series with metadata

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

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

To create a time series with metadata:

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

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

Related reference:

- “TSCreate function” on page 7-116
- “TSCreateIrr function” on page 7-118
- “UpdMetaData function” on page 7-159
- “GetMetaData function” on page 7-59

Time series input function

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

Use the following syntax for the time series input function.

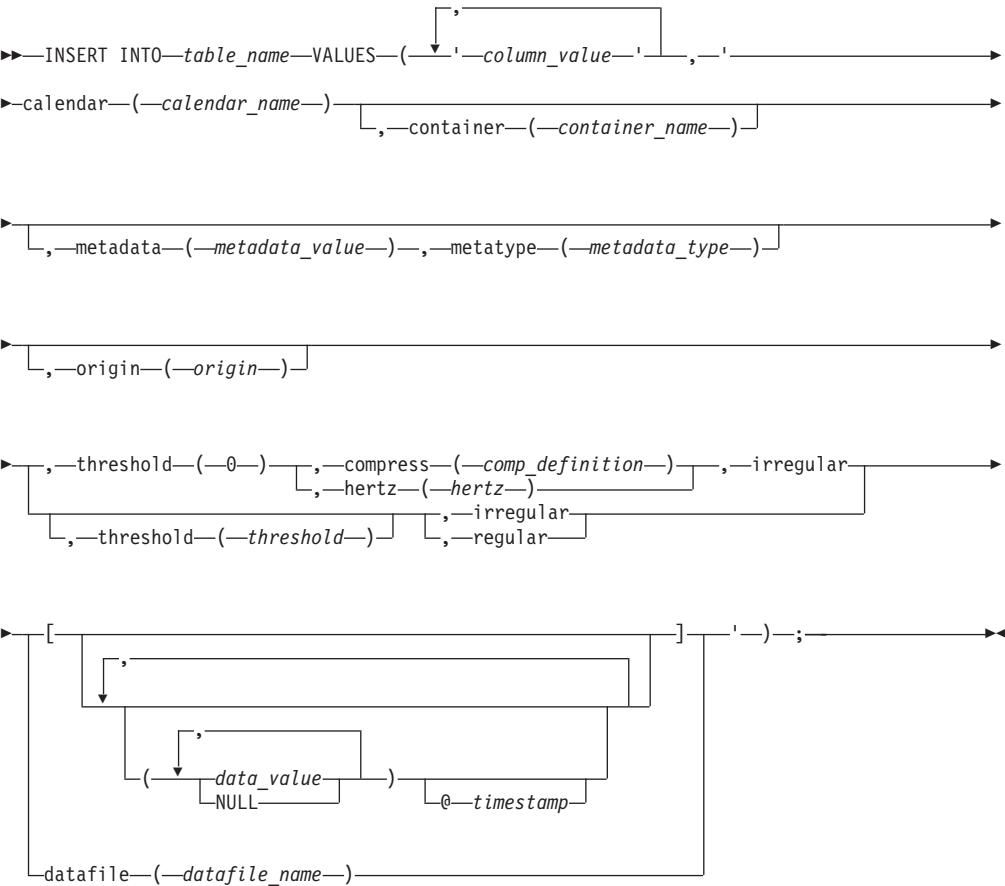


Table 3-7. Time series input function elements

Element	Description
calendar_name	The name of the calendar.

Table 3-7. Time series input function elements (continued)

Element	Description
<i>comp_definition</i>	A string that includes a compression definition for each column in the TimeSeries row type except the first column. For the syntax of the compression definition, see “TSCreateIrr function” on page 7-118. The irregular keyword must be included and the value of the <i>threshold</i> parameter must be 0.
<i>column_value</i>	A value of any column in the table except the TimeSeries column.
<i>container_name</i>	The name of an existing container.
<i>data_value</i>	The value of a column in a time series element, except the timestamp column.
<i>datafile_name</i>	The name of a file that contains time series data. For the format of the file, see “BulkLoad function” on page 7-30.
<i>hertz</i>	An integer 1-255 that specifies the number of records per second. The irregular keyword must be included and the value of the <i>threshold</i> parameter must be 0.
<i>metadata_type</i>	The user-defined metadata type. For more information, see “Creating a time series with metadata” on page 3-23.
<i>metadata_value</i>	The metadata values. Can be NULL. For more information, see “Creating a time series with metadata” on page 3-23.
<i>origin</i>	The origin of the time series. The default origin is the calendar start date.
<i>table_name</i>	The name of the time series table.
<i>threshold</i>	The threshold for the time series. If the time series stores more than this number of elements, it is converted to a container. Otherwise, it is stored directly in the row that contains it. The default is 20. The size of a row that contains an in-row time series cannot exceed 1500 bytes.
<i>timestamp</i>	The timestamp of the element. The time stamp is optional for regular time series but mandatory for irregular time series.

All data types have an associated input function that is automatically run when ASCII data is inserted into the column. For the **TimeSeries** data type, the input has several pieces of data that is embedded in the text. This information is used to convey the name of the calendar, the time stamp of the origin, the threshold, the container, the regularity, and the initial time series data. A time series is regular by default; the **regular** keyword is optional. To define an irregular time series, you must include the **irregular** keyword.

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

If you include the **hertz** or **compress** keywords, you must run the input function in an explicit transaction.

Example: Create a regular time series

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

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

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

Example: Create an irregular time series

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

```
insert into activity_stocks values (
    600, 'irregular, container(ctnr_stock), origin(2005-10-06 00:00:00.000000),
    calendar(daycal), [(6.25,1000,1,7,2,1)@2005-10-06 12:58:09.12345, (6.50, 2000,
    1,8,3,1)@2005-10-06 12:58:09.23456]');
```

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

Example: Create a time series for hertz data

The following statement creates an empty irregular time series that stores hertz data with a frequency of 50 records per second:

```
BEGIN;
INSERT INTO tstable VALUES(0, 'origin(2013-01-01 00:00:00.000000),
                                calendar(ts_1sec), container(container_2k),
                                threshold(0), hertz(50), irregular, []^T)
COMMIT;
```

Example: Create a time series for compressed data

The following statement creates an empty irregular time series that compresses the time series records for a TimeSeries subtype that has six numeric columns in addition to the timestamp column:

```
BEGIN;
INSERT INTO tstable VALUES(0, "origin(2013-01-01 00:00:00.000000),
                           calendar(ts_1sec), container(container_4k), threshold(0),
                           compress(n(),q(1,1,100),ls(0.10), lb(0.10),qls(2,0.15,100,100000)),
                           qlb(2,0.25,100,100000)), irregular, [])
COMMIT;
```

Related tasks:

“Loading JSON data” on page 3-33

Related reference:

“Load small amounts of data with SQL functions” on page 3-36

Create a time series with the output of a function

Many functions return a time series.

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

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

Load data into an existing time series

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

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

IBM Informix TimeSeries Plug-in for Data Studio

You can load time-based data into existing time series instances by creating load jobs in the IBM Informix TimeSeries Plug-in for Data Studio.

You must have the following prerequisites to create load jobs in the Informix TimeSeries Plug-in for Data Studio:

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

A load job consists of the following definitions:

- Record reader definition that describes the source of the data. The data can be in a file or in a database, including a database other than Informix.
- Table definition that describes the schema of the Informix table that has the **TimeSeries** column.
- Mapping definition that maps the source data to the time series table. If you change your table definition, you must also update the corresponding mapping definition.
- A connection profile for the Informix database.
- A connection profile for the source database, if you are loading data from another database.
- Load properties that describe how the data is loaded.
Load jobs have the following default properties:
 - Existing records can be updated.
 - The timestamp date format is yyyy-MM-dd HH:mm:ss.
 - Missing rows are created and populated with NULL values.
 - Data loading is distributed among five threads.
 - The data is saved to disk after every KB of data is loaded.
 - The data is attempted to be inserted 10 times before returning a failure.
 You can change these values and set other load job properties, such as logging load errors, by editing load properties in the plug-in.

You can reuse the definitions that you created in other load jobs.

After you create a load job, you can run it from the command line with the command-line loader application.

The maximum number of load jobs you can run simultaneously is limited by the capabilities of your computer.

The TimeSeries plug-in includes cheat sheets that provide detailed instructions for creating load jobs and loading data.

Related concepts:

“Planning for loading time series data” on page 1-23

Related tasks:

“Installing the IBM Informix TimeSeries Plug-in for Data Studio” on page 1-25

Creating a load job to load data from a file

Use the IBM Informix TimeSeries Plug-in for Data Studio to create a load job that loads time-based data from a file into existing time series instances.

Loading data from a file has the following additional requirements:

- The data can be formatted as a single-delimited, double-delimited, fixed-width, or LSE formatted data stream, for example: one or more files or URLs.
 - The data must be ASCII characters.
 - The pipe symbol (|) is interpreted as a delimiter.
 - Field delimiters can be any ASCII character or regular expression representable in Java.
 - The maximum size of the input file is set by your Java implementation.
1. Open the appropriate cheat sheet by choosing **Help > Cheat Sheets**, expand the **TimeSeries Data** category, choose **Loading from a File**, and click **OK**.

2. Follow the instructions in the cheat sheet to create the load job.

Related tasks:

“Running a load job from the command line”

Create a load job to load data from a database

Use the IBM Informix TimeSeries Plug-in for Data Studio to create a load job that loads time-based data from a database into existing time series instances.

Loading data from databases has the following additional requirements:

- The data is in a database, including databases on database servers other than Informix.
 - Connectivity information for the database server that contains the source data. The database server must support the Informix or DRDA[®] connectivity protocols. The connection is created through JDBC.
1. Open the appropriate cheat sheet by choosing **Help > Cheat Sheets**, expand the **TimeSeries Data** category, choose **Loading from a Database**, and click **OK**.
 2. Follow the instructions in the cheat sheet to create the load job.

Related tasks:

“Running a load job from the command line”

Running a load job from the command line

After you create a load job in the IBM Informix TimeSeries Plug-in for Data Studio, you can run the load job from the command line with the command-line loader application. The command-line loader application is useful if you want to load data without using the IBM Data Studio or Eclipse user interface. The command-line loader requires a minimal Eclipse platform.

The following software is required on the computer on which you run the command-line loader:

- The Eclipse Platform Runtime Binary, or a larger Eclipse or Data Studio installation, with all necessary dependencies
- TimeSeries plug-in

To load data by running the command-line loader on a computer with a minimal Eclipse platform:

1. Create load job definition files by running the TimeSeries plug-in through Data Studio or the Eclipse IDE.
2. Copy the five definition files:
 - Record reader definition file
 - Table definition file
 - Mapping definition file
 - The connection profile file for the Informix database that has the **TimeSeries** column
 - Load properties file. If you did not edit the load properties and want to use the default properties, you can create an empty file with a **.tslp** extension.
3. Move the copies of the five definition files to the computer on which you want to load the data.
4. Extract the TimeSeries plug-in files into the top-level Eclipse directory.
5. Start Eclipse with the **-clean** and **-initialize** flags to install the plug-in: **./eclipse -clean -initialize**
6. Run the command-line loader application.

Related tasks:

“Creating a load job to load data from a file” on page 3-28

“Create a load job to load data from a database” on page 3-29

Related reference:

“Command-line loader application”

Command-line loader application

You run the command-line loader application as an argument to the **eclipse** command. The command-line loader application uses the parameter files from a load job that you create with the IBM Informix TimeSeries Plug-in for Data Studio.

Syntax

```
./eclipse -application com.ibm.informix.timeseries.loader \  
-recordReader=RecordReader_file \  
-table=Table_file.tbl \  
-map=Map_file.fcmap \  
-connection=Connection_file.xml \  
-properties=LoadProps_file.ts1p
```

RecordReader_file

The name of the file that contains the record reader definition, including the database connection information, if necessary. The file extension depends on from where the data is loaded:

.udrf = User-defined record format. The data is loaded from a file.

.drr = Database record reader. The data is loaded from a database.

Table_file

The name of the file that contains the table definition.

Map_file

The name of the file that contains the mapping definition.

Connection_file

The name of the file that contains the Informix database connection profile.

LoadProps_file

The name of the file that contains the load properties.

Usage

Use the command-line loader application to load time series data into a **TimeSeries** column. The Eclipse Platform Runtime Binary, or a larger Eclipse program, the TimeSeries plug-in, and the load files that are generated by the TimeSeries plug-in are required.

You can load time series data in parallel by running multiple loader applications simultaneously.

As the data is loaded, statistics about the data are printed to the console. When the load job is complete, cumulative statistics are printed.

Example

The following command starts a load job that specifies the files that are named for **loadjob1**:


```
./eclipse -application com.ibm.informix.timeseries.loader \
        -recordReader=customer1.udrf \
        -table=loadjob1_table.tbl \
        -map=loadjob1_map.fcmap \
        -connection=loadjob1_conn.xml \
        -properties=loadjob1_prop.ts1p
```

Related tasks:

“Running a load job from the command line” on page 3-29

Writing a loader program

You can write a program to load time series data by using time series SQL routines.

You must have the following prerequisites before you load data:

- An existing table with a **TimeSeries** column. The name of the table and the name of the **TimeSeries** column must not contain uppercase letters.
- Primary key values in your table and a time series instance that is defined for each row. The time series definition must include a threshold value of 0, which means that all elements are stored in containers.
- A container that is associated with the **TimeSeries** column.
- Data that consists of a primary key and time-based data. The data can have the form of a buffer, a file as a CLOB data type, a ROW data type that is compatible with the **TimeSeries** data type, or an SQL query to extract the data from the source database.
- The data must be compatible with Informix data types.

A loader program creates a loader session. A loader session loads data into a specific **TimeSeries** column. You must use separate loader sessions for every **TimeSeries** column. Opening a loader session takes time. Leave a session open instead of repeatedly opening and closing the session.

Within a loader session, you can open multiple database sessions so that you can load data in parallel.

To write a loader program that uses one database session:

1. Initialize a global context and open a database session by running the **TSL_Init** function.
2. Copy data from a file or input stream into the database server by running the **TSL_Put**, **TSL_PutRow**, or **TSL_PutSQL** function. You run this function many times while you load data.
3. Save data to disk by running the **TSL_FlushAll** or **TSL_Commit** function. You can view information about the last flush operation by running the **TSL_FlushInfo** function.
4. If necessary, change the logging mode by running the **TSL_SetLogMode** function.
5. Monitor the progress of loaded and saved data by running the **TSL_GetLogMessage** function.
6. Close the database session by running the **TSL_SessionClose** function.
7. Remove the global context and shut down the loader by running the **TSL_Shutdown** procedure.

To write a loader program that uses multiple database sessions:

1. Initialize a global context and open a database session by running the **TSL_Init** function.
2. Open additional database sessions by running the **TSL_Attach** function.
3. Determine how to distribute the data among database sessions by running the **TSL_GetKeyContainer** function to find into which container each primary key value belongs.
Loading is faster if each database session loads data into a different container.
4. Within the context of each database session, run the **TSL_Put**, **TSL_PutRow**, or **TSL_PutSQL** function and the **TSL_FlushAll** or **TSL_Commit** function to load and save data.
5. Within the context of each database session, monitor the progress of loaded and saved data by running the **TSL_GetLogMessage** function.
6. If necessary, change the logging mode of all database sessions by running the **TSL_SetLogMode** function.
7. Within the context of each database session, close the database session by running the **TSL_SessionClose** function.
8. Remove the global context and shut down the loader by running the **TSL_Shutdown** procedure.

Examples

The following loader session uses one database session to load data into the **ts_data** table in the **stores_demo** database:

```
EXECUTE PROCEDURE ifx_allow_newline ('t');

EXECUTE FUNCTION TSL_Init ('ts_data','raw_reads',
                           3,4, NULL, '%Y-%m-%d %H:%M:%S',
                           '/tmp/rejects.log',NULL);

EXECUTE FUNCTION TSL_Put ('ts_data|raw_reads',
'4727354321000111|KWH|P|2010-11-10 00:00:00.00000|0.092|
4727354321000111|KWH|P|2010-11-10 00:15:00.00000|0.084|
4727354321000111|KWH|P|2010-11-10 00:30:00.00000|0.09|
4727354321000111|KWH|P|2010-11-10 00:45:00.00000|0.085|
4727354321000111|KWH|P|2010-11-10 01:00:00.00000|0.088|
4727354321000111|KWH|P|2010-11-10 01:15:00.00000|0.088|
4727354321000111|KWH|P|2010-11-10 01:30:00.00000|0.085|
4727354321000111|KWH|P|2010-11-10 01:45:00.00000|0.091|
4727354321046021|KWH|P|2010-11-10 00:00:00.00000|0.041|
4727354321046021|KWH|P|2010-11-10 00:15:00.00000|0.041|
4727354321046021|KWH|P|2010-11-10 00:30:00.00000|0.04|
4727354321046021|KWH|P|2010-11-10 00:45:00.00000|0.041|
4727354321046021|KWH|P|2010-11-10 01:00:00.00000|0.041|
4727354321046021|KWH|P|2010-11-10 01:15:00.00000|0.041|
4727354321046021|KWH|P|2010-11-10 01:30:00.00000|0.055|
4727354321046021|KWH|P|2010-11-10 01:45:00.00000|0.073|
4727354321046021|KWH|P|2010-11-10 02:00:00.00000|0.071|
4727354321046021|KWH|P|2010-11-10 02:15:00.00000|0.068|
4727354321046021|KWH|P|2010-11-10 02:30:00.00000|0.07|
');

EXECUTE FUNCTION TSL_Put ('ts_data|raw_reads',
'4727354321090954|KWH|P|2010-11-10 00:00:00.00000|0.026|
4727354321090954|KWH|P|2010-11-10 00:15:00.00000|0.035|
4727354321090954|KWH|P|2010-11-10 00:30:00.00000|0.062|
4727354321090954|KWH|P|2010-11-10 00:45:00.00000|0.092|
4727354321090954|KWH|P|2010-11-10 01:00:00.00000|0.016|
4727354321090954|KWH|P|2010-11-10 01:15:00.00000|0.043|
4727354321090954|KWH|P|2010-11-10 01:30:00.00000|0.038|
4727354321090954|KWH|P|2010-11-10 01:45:00.00000|0.037|
```

```

4727354321090954|KWH|P|2010-11-10 02:00:00.00000|0.034|
4727354321090954|KWH|P|2010-11-10 02:15:00.00000|0.023|
4727354321090954|KWH|P|2010-11-10 02:30:00.00000|0.03|
4727354321090954|KWH|P|2010-11-10 02:45:00.00000|0.05|
4727354321090954|KWH|P|2010-11-10 03:00:00.00000|0.048|
4727354321090954|KWH|P|2010-11-10 03:15:00.00000|0.047|
');

```

```

begin;
EXECUTE FUNCTION TSL_FlushAll ('ts_data|raw_reads');
commit;

EXECUTE FUNCTION TSL_SessionClose ('ts_data|raw_reads');

EXECUTE PROCEDURE TSL_Shutdown ('ts_data|raw_reads');

```

Related concepts:

“Planning for loading time series data” on page 1-23

Related tasks:

“Loading JSON data”

Related reference:

“TSL_Put function” on page 7-135

“TSL_Shutdown procedure” on page 7-141

“TSL_SessionClose function” on page 7-139

“TSL_Init function” on page 7-133

“TSL_SetLogMode function” on page 7-140

“TSL_GetLogMessage function” on page 7-132

“TSL_GetKeyContainer function” on page 7-131

“TSL_PutRow function” on page 7-137

“TSL_PutSQL function” on page 7-138

“TSL_FlushInfo function” on page 7-129

“TSL_Commit function” on page 7-124

“TSL_FlushAll function” on page 7-128

Loading JSON data

You can load JSON data into a time series with the time series input statement, through a virtual table, or by writing a loader program.

When you insert time series data that contains JSON documents with the time series input function as an INSERT statement, the database server automatically casts the JSON documents to BSON.

When you create a virtual table that is based on a time series that contains JSON documents, you can insert rows through the virtual table. You must explicitly cast the JSON data to the JSON data type, and then cast the data to the BSON data type. When you query a virtual table, you must cast the contents of the BSON column to JSON to view the data.

To write a loader program to load JSON data into a time series:

1. Do the prerequisite tasks that are necessary for a loader program, including creating a **TimeSeries** subtype, creating a table, creating a container, and instantiating a time series.
2. Create an external table and load time series data from a file or a pipe. For example, use the following format for loading from a file:

```

CREATE EXTERNAL TABLE ext_table_name
(
  primary_key_col    data_type,
  tstamp_col        DATETIME YEAR TO FRACTION(5),
  json_col          JSON
)
USING(
  FORMAT 'DELIMITED',
  DATAFILES
  (
    "DISK: path/filename"
  )
);

```

The *ext_table_name* is the name that you give to the external table.

The *primary_key_col* is the name of the primary key column of the time series table. The primary key can consist of multiple columns.

The *tstamp_col* is the name of the timestamp column.

The *data_type* is the data type of the primary key column.

The *json_col* is the name of the column that contains JSON documents.

The *path* is the directory for the data file.

The *filename* is the name of the data file.

- Write a loader program that loads the time series data from the external table. Run the **TSL_PutSQL** function with a SELECT statement that returns data from the external table and casts the JSON column to BSON. For example, use the following format to run the **TSL_PutSQL** function:

```

EXECUTE FUNCTION TSL_PutSQL('ts_table_name|ts_col',
  "SELECT primary_key_col, tstamp_col,
    json_col::bson FROM ext_table_name");

```

You cannot load JSON data with the **TSL_Put** or **TSL_PutRow** functions.

Related concepts:

“JSON time series” on page 1-12

Related tasks:

“Example: Create and load a time series with JSON data” on page 3-10

“Writing a loader program” on page 3-31

Related reference:

 CREATE EXTERNAL TABLE Statement (SQL Syntax)

“Time series input function” on page 3-24

Loading data from a file into a virtual table

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

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

To load relational data through a virtual table:

- Create a virtual table that is based on a time series table.
- Put your input data in a single file.

3. Format the data according to the standard IBM Informix load file format.
4. Use any of the Informix load utilities: **pload**, **onpload**, **dbload**, or the **load** command in DB-Access, to load the file into the virtual table.

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

Related concepts:

"Planning for loading time series data" on page 1-23

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

Related reference:

"TSCreateVirtualTab procedure" on page 4-5

Load data with the BulkLoad function

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

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

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

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

Related concepts:

"Planning for loading time series data" on page 1-23

Related reference:

"BulkLoad function" on page 7-30

Data file formats for BulkLoad

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

- Using type constructors
- Using tabs

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

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

```
row(2011-01-03 00:00:00.000000, 1.1, 2.2)
row(2011-01-04 00:00:00.000000, 10.1, 20.2)
```

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

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

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

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

The spaces between entries represent a tab.

In both formats, NULL indicates a null entry.

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

Example: Load data with BulkLoad

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

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

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

Load small amounts of data with SQL functions

You can load individual elements or sets of elements by using time series SQL functions.

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

PutElem

Updates a time series with a single element.

PutSet Updates a time series with a set of elements.

InsElem

Inserts an element into a time series.

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

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

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

The *TimeSeries_col* argument is the name of the column in which the time series is located. The *FunctionName* argument is the name of the function. The *data* argument is in the row type data element format. The WHERE clause specifies which row in the table to update.

The following example appends an element to a time series by running the **PutElem** function:

```
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 example, by including binary arithmetic functions.

Related concepts:

“Planning for loading time series data” on page 1-23

Related reference:

“Time series input function” on page 3-24

“Binary arithmetic functions” on page 7-27

Delete time series data

You can delete time series data to remove incorrect data or to remove old data.

The easiest and fastest way to delete old time series data is to create a rolling window container, from which aging data in partitions can be manually or automatically detached or destroyed.

You can remove the oldest time series data through an end date in one for more containers for multiple time series instances by running the **TSContainerPurge** function.

You can delete data from a single time series instance in the following ways:

- Delete a single element by running the **DelElem** function.
- Delete elements in a time range and free any resulting empty pages by running the **DelRange** function.
- Free any empty pages in a time series instance by running the **NullCleanup** function.

Related reference:

“Rules for rolling window containers” on page 3-16

“DelRange function” on page 7-44

“TSContainerPurge function” on page 7-108

“DelElem function” on page 7-43

“NullCleanup function” on page 7-75

Manage packed data

You can insert, delete, and select packed data.

You must follow these rules when you insert or delete packed data:

- Records must be inserted in chronological order. Existing records cannot be replaced or updated. Missing records are allowed but cannot be updated.
- Functions that insert or delete compressed data must be run in an explicit transaction.

You can determine whether a time series contains packed data by running the **GetPacked** function. You can determine the frequency of hertz data by running the **GetHertz** function. You can determine the compression definition of compressed data by running the **GetCompression** function.

Insert packed data

You can insert data into an existing hertz or compressed time series by running the following functions:

- **InsElem**
- **InsSet**
- **TSL_Put**
- **TSL_PutRow**
- **TSL_PutSQL**

You can also insert data through a virtual table.

The data is saved to disk when an element reaches maximum size or the transaction is committed.

Delete packed data

You can delete elements of hertz data by running the following functions:

- **DelClip**
- **DelItem**
- **DelRange**
- **DelTrim**

You cannot delete individual records from an element. Regardless of the time range you specify, whole elements are deleted.

You cannot delete elements from a compressed time series. To delete compressed data, you must delete the time series instance. For example, you can delete a row in the table that contains packed data.

Select packed data

You can run any time series function that selects data on packed data. You do not need to know that the data is packed. When packed data is returned by a time series function, every packed record appears as an individual element.

The following example shows a returned value from a hertz time series:

```
SELECT GetElem(ts_2k, '2014-01-01 00:00:02.50000') FROM tstable50

(expression) ROW('2014-01-01 00:00:02.50000',2      ,50      ,5050
```

Similarly, when you create a virtual table that is based on packed data, each packed record is a row in the virtual table. The following example shows four values in a virtual table that are from the same element in a hertz time series:

```
SELECT * FROM vt_tstable_2k WHERE tstamp < '2014-01-01 00:00:00.08000'
```

```
id          50
tstamp      2014-01-01 00:00:00.00000
tssmallint  2
tsint       50
tsbigint    5050

id          50
tstamp      2014-01-01 00:00:00.02000
tssmallint  2
tsint       50
tsbigint    5050

id          50
tstamp      2014-01-01 00:00:00.04000
tssmallint  2
```



```
tsint      50
tsbigint   5050

id         50
tstamp     2014-01-01 00:00:00.060000
tssmallint 2
tsint      50
tsbigint   5050
```

4 row(s) retrieved.

Related concepts:

“Hertz time series” on page 1-8

“Compressed numeric time series” on page 1-10

Related reference:

“GetPacked function” on page 7-64

“GetHertz function” on page 7-54

“GetCompression function” on page 7-50

Chapter 4. Virtual tables for time series data

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

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

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

You can use a virtual table as the basis of a data mart and accelerate queries on time series data. When you accelerate queries, you can analyze large amounts of data faster than directly querying the base table or the virtual table.

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

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

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

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

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

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

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

Related concepts:

“Planning for accessing time series data” on page 1-24

“Planning for loading time series data” on page 1-23

Related tasks:

“Loading data from a file into a virtual table” on page 3-34

Performance of queries on virtual tables

The performance of queries on virtual tables depends on the type of query and whether the query is accelerated.

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

If you want to query large amounts of time series data, the performance of queries on virtual tables that are accelerated by Informix Warehouse Accelerator is significantly faster than any other type of query. You can control how much of the virtual table is loaded into the data mart to further improve query performance. You can easily change which parts of the virtual table are in the data mart. For example, you create a virtual table that contains three years of time series data. You create a data mart that is based on the virtual table. You define virtual partitions for the data mart so that you can quickly refresh the data mart. You define time windows for the first three months of each year of data and load that data into the data mart. You run analytic queries on the data through the accelerator. Then you change the time window to the second three months of each year and run analytic queries on that data.

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

- Create the virtual table as a fragmented table and enable PDQ so that queries are run in parallel. The base table must be fragmented by expression.
- 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 that consists of all key columns.
- Run UPDATE STATISTICS on the base table and on its key columns after any load or delete operation:

```
UPDATE STATISTICS HIGH FOR TABLE daily_stocks;
```

```
UPDATE STATISTICS HIGH FOR TABLE daily_stocks (stock_id);
```

By default, statistics are automatically updated once a week.

Related concepts:

 [Data marts for time series data \(Informix Warehouse Accelerator Guide\)](#)

The structure of virtual tables

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

For example, the table **ts_data** contains a **TimeSeries** column called **raw_reads** that contains a row type with **timestamp** and **value** columns. The following table displays

part of the **ts_data** table. The actual time stamp values are shown for clarity, although the time stamp values are calculated instead of stored in regular time series.

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

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

The virtual table that is based on the **ts_data** table converts the **raw_reads** column elements into individual columns. The rows are ordered by timestamp, starting with the earliest timestamp. The following table displays part of the virtual table that is based on the **ts_data** table.

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

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

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

The display of data in virtual tables

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

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

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

When you select data from a virtual table by timestamps, the rows whose timestamps are closest to being equal to or earlier than the timestamps specified in the query are returned. If the time series is irregular, the returned rows show the

same timestamps as specified in the query, regardless if the actual timestamps are the same. You can specify that when you select data from a virtual table by timestamps, only rows whose timestamps are exactly equal to the timestamps specified in the query are returned.

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

Related concepts:

“The TSVTMode parameter” on page 4-16

Related reference:

“TSCreateVirtualTab procedure” on page 4-5

“TSCreateExpressionVirtualTab procedure” on page 4-13

Insert data through virtual tables

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

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

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

If the base table has a primary key, the primary key is used to find the row to update and updates to the base table do not require accurate values for columns that are not part of the primary key.

If the base table does not have a primary key, all columns in the table except the **TimeSeries** column are used to identify the row to be updated and updates to the base table require accurate values for every column in the base table other than the **TimeSeries** column. You can only update the values in the **TimeSeries** column.

You can specify the rules for the INSERT statement and which columns can be updated:

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

You can speed data insertion by reducing the amount of logging. If you reduce the amount of logging, INSERT statements must be run in a transaction without other types of SQL statements and the elements that are inserted are not visible until the transaction commits.

Control the rules for inserting data by setting the *NewTimeSeries* parameter and the *TSVTMode* parameter in the **TSCreateVirtualTab** procedure.

Related concepts:

“The TSVTMode parameter” on page 4-16

Related reference:

“TSCreateVirtualTab procedure”

Creating a time series virtual table

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

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

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

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

If you alter the base table, you must drop and re-create the virtual table.

Related reference:

“TSCreateVirtualTab procedure”

“TSCreateExpressionVirtualTab procedure” on page 4-13

TSCreateVirtualTab procedure

The **TSCreateVirtualTab** procedure creates a virtual table that is based on a table that contains a **TimeSeries** column.

Syntax

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

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

VirtualTableName

The name of the new virtual table.

BaseTableName

The name of the base table.

NewTimeSeries (**optional**)

The definition of the new time series to create.

TSVTMode (**optional**)

Sets the virtual table mode, as described in “The TSVTMode parameter” on page 4-16

page 4-16. Can be an integer or a string of one or more flag names that are separated by one of the following delimiters: plus sign (+), pipe (|), or comma (,).

***TSColName* (optional)**

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

Usage

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

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

The NewTimeSeries parameter

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

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

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

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

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

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

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

Example

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

```
EXECUTE PROCEDURE TSCreateVirtualTab('daily_stocks_virt',
    'daily_stocks', 'calendar(daycal)',
    origin('2011-01-03 00:00:00.000000'));
```

The following statement creates a virtual table with the same characteristics as the previous statement, except that the *TSVTMode* parameter specifies to allow duplicate timepoints and to reduce logging:

```
EXECUTE PROCEDURE TSCreateVirtualTab('daily_stocks_virt',
    'daily_stocks', 'calendar(daycal)',
    origin('2011-01-03 00:00:00.000000'),
    'put_elem+reduced_log');
```

Related concepts:

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

“Insert data through virtual tables” on page 4-4

Related tasks:

“Creating a time series virtual table” on page 4-5

“Loading data from a file into a virtual table” on page 3-34

Example of creating a virtual table

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

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

Query the base table

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

```
create row type stock_bar(  
    timestamp      datetime year to fraction(5),  
    high          real,  
    low           real,  
    final         real,  
    vol          real  
);  
  
create table daily_stocks (  
    stock_id      int,  
    stock_name    varchar,  
    stock_data    TimeSeries(stock_bar)  
);
```

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

Table 4-4. The **daily_stocks** base table

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

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

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

In this query, *one_real* is a row type that is 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, use a query that has the following format:

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

Create the virtual table

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

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

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

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

The following table illustrates the virtual table, **daily_stocks_no_ts**.

Table 4-5. The *daily_stocks_no_ts* virtual table

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

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

Query the virtual table

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

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

The query to obtain price and volume within a specific time range is:

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

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

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

Inserting data into the virtual table is also simple. To add an 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**.

Related concepts:

“The TSVTMode parameter” on page 4-16

Example of creating a fragmented virtual table

This example shows how to create a fragmented virtual table that is based on a table that contains time series data and that is fragmented by expression.

Prerequisites

Before you run the statements in this example, create three dbspaces named **db1**, **db2**, and **db3**. Otherwise, substitute your dbspace names in the example.

About this example

When you create the **stores_demo** database, all the setup tasks for creating and loading a time series table are complete. The **ts_data** table in the **stores_demo** database contains time series data. The **ts_data_v** table is a virtual table that is based on the **ts_data** table. The **ts_data** table and the **ts_data_v** virtual table are not fragmented.

In this example, you alter the **ts_data** table to fragment it by expression into three dbspaces. You re-create the **ts_data_v** virtual table as a fragmented virtual table. Then, you run queries in parallel against the **ts_data_v** virtual table.

Preparing for parallel queries

Run the SQL statements in these steps from an SQL editor, such as DB-Access or IBM OpenAdmin Tool (OAT) for Informix.

To prepare for running parallel queries on the **ts_data_v** virtual table:

1. If necessary, create the **stores_demo** database by running the following command:

```
dbaccessdemo
```
2. If necessary, set the **PDQPRIORITY** environment variable to a value other than OFF to enable parallel database queries. For example, run the following SQL statement:

```
SET PDQPRIORITY 100
```
3. Specify an explain output file and enable explain output so that you can determine whether your queries run in parallel by running the following SQL statements:

```
SET EXPLAIN FILE TO 'c:/test/parallelvtq.out';  
SET EXPLAIN ON;
```

You can specify a different directory and file name for the explain output file.
4. Drop the existing virtual table on time series data, **ts_data_v**, in the **stores_demo** database by running the following SQL statement:

```
DROP TABLE ts_data_v;
```
5. Alter the **ts_data** table to fragment it by expression by running the following SQL statement:

```
ALTER FRAGMENT ON TABLE ts_data init  
  FRAGMENT BY EXPRESSION  
    PARTITION part1 (loc_esl_id <= "4727354321355594") in db1,  
    PARTITION part2 (loc_esl_id <= "4727354321510846") in db2,  
    REMAINDER IN db3;
```
6. Create the fragmented virtual table, **ts_data_v**, by running the following SQL statement:

```
EXECUTE PROCEDURE TSCreateVirtualTab(
    'ts_data_v',
    'ts_data',
    'origin(2010-11-10 00:00:00.000000),calendar(call15min),
    container(raw_container),threshold(0),regular',
    'fragment',
    'raw_reads'
);
```

The difference between this definition of the **ts_data_v** virtual table and the original definition is the value of the fourth parameter, the *TSVTMode* parameter. In this definition, the *TSVTMode* parameter is set to **fragment** to fragment the virtual table. In the original definition of the **ts_data_v** virtual table, the *TSVTMode* parameter is set to 0, which indicates the default behavior.

7. Set the isolation level to DIRTY READ by running the following SQL statement:

```
SET ISOLATION TO DIRTY READ;
```

The DIRTY READ isolation level ensures that a parallel query on a virtual table succeeds if the data in the base table is being modified at the same time.

Run parallel queries

The following SQL statement selects the number of values, the sum of the values, and the average of the values for a 15-minute period on February 7, 2011 for each customer who lives in the state of Arizona:

```
SELECT state,
       COUNT(value) num_values,
       SUM(value) sum,
       AVG(value) average
FROM ts_data_v v, customer_ts_data l, customer c
WHERE
    v.loc_esl_id=l.loc_esl_id AND l.customer_num=c.customer_num
    AND state = "AZ"
    AND v.tstamp BETWEEN '2011-02-07 23:30:00.000000'
                      AND '2011-02-07 23:45:00.000000'
GROUP BY state, value
ORDER BY state, value;
```

The result of the query displays the information for the three qualifying customers:

state	(count)	(sum)	(avg)
AZ	1	0.011	0.011
AZ	1	0.012	0.012
AZ	2	0.050	0.025

The explain output file, `parallelvtq.out`, describes the query plan for this query. The information (Parallel, fragments: ALL) for the second and third scans indicates that the scans ran in parallel and accessed all fragments:

...

```
Estimated Cost: 14
Estimated # of Rows Returned: 1
Maximum Threads: 5
Temporary Files Required For: Order By Group By
```

```
1) informix.c: SEQUENTIAL SCAN
```

```
Filters: informix.c.state = 'AZ'
```

```
2) informix.l: INDEX PATH
```

```
(1) Index Keys: customer_num (Parallel, fragments: ALL)
```

```
Lower Index Filter: informix.l.customer_num = informix.c.customer_num
```

```
NESTED LOOP JOIN
```

3) informix.v: VTI SCAN (Parallel, fragments: ALL)

```
VTI Filters: (informix.lessthanorequal(informix.v.tstamp,datetime
(2011-02-07 23:45:00.00000) year to fraction(5) ) AND
informix.greaterthanorequal(informix.v.tstamp,datetime
(2011-02-07 23:30:00.00000) year to fraction(5) ))
```

```
Filters: informix.v.loc_esl_id = informix.l.loc_esl_id
NESTED LOOP JOIN
```

The following SQL statement selects the number of values, the sum of the values, and the average of the values after 11:30 PM on February 7, 2011 for each customer:

```
SELECT state,
        COUNT(value) num_values,
        SUM(value) sum,
        AVG(value) average
FROM ts_data_v v, customer_ts_data l, customer c
WHERE
        v.loc_esl_id=l.loc_esl_id
        AND l.customer_num=c.customer_num
        AND greaterthan(v.tstamp,'2011-02-07 23:30:00.00000')
GROUP BY state, value
ORDER BY state, value;
```

The result of the query displays the information for all the qualifying customers:

state	(count)	(sum)	(avg)
AZ	1	0.011	0.011
AZ	1	0.025	0.025
CA	1	0.020	0.02
CA	1	0.023	0.023
CA	1	0.056	0.056
CA	1	0.065	0.065
CA	1	0.071	0.071
CA	1	0.073	0.073
CA	1	0.088	0.088
CA	1	0.162	0.162
CA	1	0.204	0.204
CA	1	0.226	0.226
CA	1	0.246	0.246
CA	1	0.277	0.277
CA	1	0.323	0.323
CA	1	0.340	0.34
CA	1	0.415	0.415
CA	1	0.469	0.469
CA	1	0.670	0.67
CA	1	1.412	1.412
CO	1	0.118	0.118
DE	1	0.256	0.256
FL	1	3.470	3.47
MA	1	4.388	4.388
NJ	2	0.374	0.187
NY	1	0.239	0.239
OK	1	0.086	0.086

The GROUP BY and ORDER BY clauses order the results by state.

The explain output for this query shows that the query ran in parallel.

...

```
Estimated Cost: 63
Estimated # of Rows Returned: 1
Maximum Threads: 5
```

Temporary Files Required For: Order By Group By

1) informix.l: SEQUENTIAL SCAN

2) informix.v: VTI SCAN (Parallel, fragments: ALL)

VTI Filters: informix.greaterthan(informix.v.tstamp,datetime
(2011-02-07 23:30:00.00000) year to fraction(5))

Filters: informix.v.loc_esl_id = informix.l.loc_esl_id

NESTED LOOP JOIN

3) informix.c: INDEX PATH

(1) Index Keys: customer_num (Parallel, fragments: ALL)

Lower Index Filter: informix.l.customer_num = informix.c.customer_num

NESTED LOOP JOIN

Related concepts:

“The TSVTMode parameter” on page 4-16

➡ dbaccessdemo command: Create demonstration databases (DB-Access Guide)

Related reference:

➡ ALTER FRAGMENT statement (SQL Syntax)

➡ SET PDQPRIORITY statement (SQL Syntax)

➡ SET ISOLATION statement (SQL Syntax)

➡ SET EXPLAIN statement (SQL Syntax)

TSCreateExpressionVirtualTab procedure

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

Syntax

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

VirtualTableName

The name of the new virtual table.

BaseTableName

The name of the base table.

expression

The expression to be evaluated on time series data. The expression must result in a time series value that has a **TimeSeries** subtype that is specified by the *subtype* parameter.

subtype

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

TSVTMode (optional)

Sets the virtual table mode, as described in “The TSVTMode parameter” on page 4-16.

TSColName (optional)

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

Usage

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

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

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

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

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

You can use the following variables in the expression:

- **\$ts_column_name**: If the base table has multiple **TimeSeries** columns, instead of specifying the name of the **TimeSeries** column in the expression, you can use the **\$ts_column_name** variable to substitute the value of the *TSColName* parameter in the **TSCreateExpressionVirtualTab** procedure. Because the column name is a variable, you can use the same expression for each of the **TimeSeries** columns in the table.
- **\$ts_begin_time**: Instead of specifying a DATETIME value, you can use this variable and specify the beginning time point of the time series in the WHERE clause of the SELECT statement when you query the virtual table. If the WHERE clause does not contain the beginning timepoint, the first timepoint in the time series is used.
- **\$ts_end_time**: Instead of specifying a DATETIME value, you can use this variable and specify the ending time point of the time series in the WHERE clause of the SELECT statement when you query the virtual table. If the WHERE clause does not contain the ending timepoint, the last timepoint in the time series is used.

You can test whether the subtype that you create for the results of the expression is valid by running the following statement against your base table with the

expression and *subtype* parameters that you plan to use in the **TSCreateExpressionVirtualTab** procedure:

```
SELECT expression::timeseries(subtype) FROM BaseTableName;
```

If the statement fails, you cannot create the virtual table.

Examples

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

Example 1: Find the daily maximum and minimum values

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

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

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

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

meter_id	t	value1	value2
met00000	2011-01-01 00:00:00.00000	37.000000000000	9.000000000000
met00000	2011-01-02 00:00:00.00000	34.000000000000	8.000000000000
met00001	2011-01-01 00:00:00.00000	36.000000000000	9.000000000000
met00001	2011-01-02 00:00:00.00000	36.000000000000	10.000000000000
met00002	2011-01-01 00:00:00.00000	34.000000000000	9.000000000000
met00002	2011-01-02 00:00:00.00000	36.000000000000	10.000000000000

6 row(s) retrieved.

Example 2: Find the daily maximum of a running average

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

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

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

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

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

meter_id	t	value
met00000	2011-01-01 00:00:00.00000	30.250000000000
met00000	2011-01-02 00:00:00.00000	29.500000000000
met00001	2011-01-01 00:00:00.00000	29.750000000000
met00001	2011-01-02 00:00:00.00000	31.000000000000
met00002	2011-01-01 00:00:00.00000	31.250000000000
met00002	2011-01-02 00:00:00.00000	28.750000000000

6 row(s) retrieved.

Related concepts:

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

Related tasks:

“Creating a time series virtual table” on page 4-5

The TSVTMode parameter

The *TSVTMode* parameter configures the behavior and display of the virtual table for time series data.

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

- How data is updated in the base table when you insert data into the virtual table
- Whether NULL time series elements are displayed in a virtual table
- Whether to fragment the virtual table so that queries can be run on the virtual table in parallel
- Whether updates to existing rows in the base table require accurate values for columns that are not part of the primary key
- Whether existing values in columns other than the **TimeSeries** column or the primary key columns can be updated.
- Whether NULL values can be used in the INSERT statement for columns other than the primary key columns.
- Whether hidden time series elements are displayed in a virtual table
- Whether data selected by time stamp exactly matches the specified timestamps or includes the last rows that are equal to or earlier than the specified timestamps.
- Whether to quickly insert elements into existing time series instances that are stored in containers.
- Whether to reduce how many log records are generated when you insert data.

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

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

The default value of the *TSVTMode* parameter, 0, sets the default behavior of the virtual table. Each of the other values of the *TSVTMode* parameter reverses one aspect of the default behavior.

You can set the *TSVTMode* parameter to a combination of the values. You can specify values for the *TSVTMode* parameter in the following formats:

- **Numeric:** Sum the numeric values of the flags that you want to include. For example, if you want both null and hidden elements to be displayed in the virtual table, set the *TSVTMode* parameter to 514 (512 + 2). You can also specify the numeric value as a hexadecimal number.
- **String:** List the flag names that you want to include, separated by one of the following delimiters: plus sign (+), pipe (|), or comma (.). For example, if you want both null and hidden elements to be displayed in the virtual table, set the *TSVTMode* parameter to 'scan_hidden+show_nulls'.

Table 4-6. Settings for the *TSVTMode* parameter

Flag name	Value	Description
putelemnodups	0	<p>Default. The virtual table has the following behavior:</p> <ul style="list-style-type: none"> • Multiple elements for the same timepoint are not allowed. Updates to the underlying time series update existing elements for the same timepoint. Uses the PutElemNoDups function. • Null elements are not included in the virtual table. • The virtual table is not fragmented. • If the base table has a primary key, the primary key is used to find the row to update and updates to the base table do not require accurate values for columns that are not part of the primary key. If the base table does not have a primary key, all columns in the table except the TimeSeries column are used to identify the row to be updated and updates to the base table require accurate values for every column in the base table other than the TimeSeries column. NOT NULL constraints are included in the virtual table for the primary key columns and other columns that have NOT NULL constraints in the base table. • For updates to existing rows, only the TimeSeries column can be updated. • Hidden elements are not included in the virtual table. • When you select data from a virtual table by timestamps, the rows whose timestamps are closest to being equal to or earlier than the timestamps specified in the query are returned. If the time series is irregular, the returned rows show the same timestamps as specified in the query, regardless if the actual timestamps are the same. <p>See “Default behavior” on page 4-20</p>
putelem	1	<p>Multiple elements for the same timepoint are allowed. Updates to the underlying time series insert elements even if elements exist for the timepoints. Uses the PutElem function.</p> <p>See “Duplicate timepoints” on page 4-23.</p>

Table 4-6. Settings for the TSVTMode parameter (continued)

Flag name	Value	Description
show_nulls	2	<p>Null elements are displayed in the virtual table. Hidden elements are displayed as null elements, unless the value 512 is also set.</p> <p>See “Null and hidden elements” on page 4-23.</p>
fragment	4	<p>This setting is only valid if the base table is fragmented by expression.</p> <p>The virtual table is fragmented by expression using the same distribution scheme as the base table. Queries on the virtual table are run in parallel if PDQ is enabled.</p>
disable_not_null_constraints	16	<p>For existing rows, you can specify NULL values for columns that are not part of the primary key, regardless if those columns have NOT NULL constraints in the base table. NOT NULL constraints are not included in the virtual table, but are enforced in the base table.</p> <p>For new rows, you can specify null values for columns that are not part of the primary key and do not have NOT NULL constraints.</p>
update_nonkey_not_nulls	32	<p>This setting is valid only if the base table has a primary key.</p> <p>You can update the value of columns in an existing row that are not part of the primary key. You can specify NULL for non-primary key columns that you do not want to update. All columns that have non-NULL values in the INSERT statement are updated in the base table, except the primary key columns.</p> <p>See “Update values that are not in the primary key” on page 4-21.</p>
update_nonkey_include_nulls	64	<p>This setting is valid only if the base table has a primary key.</p> <p>You can update the value of all the columns in an existing row that are not part of the primary key, including setting null values for columns that allow null values. Columns that are not part of the primary key are updated to the value included in the INSERT statement. Columns that allow null values can be set to NULL.</p> <p>See “Update values that are not in the primary key and allow null values” on page 4-22.</p>

Table 4-6. Settings for the *TSVTMode* parameter (continued)

Flag name	Value	Description
elem_insert	128	<p>You can quickly insert elements directly into containers given the following constraints:</p> <ul style="list-style-type: none"> • The base table has a primary key • The time series instances exist and are stored in containers • Base table columns are not being updated <p>Cannot be combined with the settings 16, 32, or 64. Cannot be combined with the <i>NewTimeSeries</i> parameter.</p>
reduced_log	256	<p>Reduces how many log records are generated when you insert elements into containers. By default, every element that you insert generates two log records: one for the inserted element and one for the page header update. If this flag is set, page header updates are logged per transaction instead of per element.</p> <p>The INSERT statements must be run within a transaction without other types of SQL statements. The elements that are inserted are not visible by dirty reads until after the transaction commits.</p>
scan_hidden	512	<p>Hidden elements are displayed in the virtual table.</p> <p>See “Null and hidden elements” on page 4-23.</p>
scan_discreet	1024	<p>When you select data from a virtual table by timestamps, only rows whose timestamps are exactly equal to the timestamps specified in the query are returned.</p>

Update columns in the base table

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

The following table describes how to control updating columns in the base table, assuming that the base table has a primary key. Whether the *NewTimeSeries* parameter is specified also affects the behavior of inserting data into the base table. For information about the effect of the *NewTimeSeries* parameter, see “TSCreateVirtualTab procedure” on page 4-5.

Table 4-7. *TSVTMode* parameter settings that affect which columns are updated in the base table

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

Table 4-7. *TSVTMode* parameter settings that affect which columns are updated in the base table (continued)

Columns to update	TSVTMode parameter setting
Update the TimeSeries column and all other non-primary key columns. You can set columns that do not have NOT NULL constraints to null values.	64 or 80 (64 + 16)
Update the TimeSeries column and all other non-primary key columns that have NOT NULL constraints. You can specify null values for columns that have NOT NULL constraints.	48 (32 + 16)

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

```
acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000
energy       33070
temperature  -13.00000000000
owner        John
address      5 Nowhere Place
```

1 row(s) retrieved.

Default behavior

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

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

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

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

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

```
SELECT * FROM smartmeters_vti_nn;
```

```
acct_no      6546
meter_id     234
```

```

t          2011-01-01 00:00:00.00000
energy     33070
temperature -13.0000000000
owner      John
address    5 Nowhere Place

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

```

2 row(s) retrieved.

Fragment the virtual table

When you run queries in parallel on fragmented virtual tables, set the isolation level to Dirty Read. Otherwise, when a parallel query on the virtual table and a query that modifies data on the base table coincide, the parallel query might fail with a -244 error. The Dirty Read isolation level can result in returning phantom rows from other sessions that are never committed. If you do not want to set the Dirty Read isolation level and you want to run a parallel query at a time when you know that the base table is being updated, you can disable PDQ for the transaction.

The results of parallel queries on fragmented virtual tables are not necessarily ordered by the primary key values. You can ensure that the results are ordered properly by including an ORDER BY clause in the query that specifies the primary key and the timestamp column. If you do not include an ORDER BY clause, the results are grouped in batches of up to 128 rows per primary key value. Within each batch, the results are ordered by timestamp. The batches for each primary key value are ordered chronologically, but interspersed with batches for different primary key values.

For example, suppose that the results of a query include 396 values for the primary key value meter1, 347 values for meter2, and 280 values for meter3. The results might be ordered in the following way:

```

128 rows of meter1
128 rows of meter2
128 rows of meter3
128 rows of meter1
128 rows of meter3
128 rows of meter1
128 rows of meter3
12 rows of meter1
24 rows of meter3
91 rows of meter2

```

Update values that are not in the primary key

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

```

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

```

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

```

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

```

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

```

SELECT * FROM smartmeters_vti_nn_nk_nn;

```

```

acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000
energy       33070
temperature  -13.00000000000
owner        Jim
address      5 Nowhere Place

```

```

acct_no      6546
meter_id     234
t            2011-01-01 00:45:00.00000
energy       3234
temperature  -12.00000000000
owner        Jim
address      5 Nowhere Place

```

2 row(s) retrieved.

Update values that are not in the primary key and allow null values

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

```

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

```

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

```

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

```

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

```

SELECT * FROM smartmeters_vti_nn_nk_in;

```

```

acct_no      6546
meter_id     234
t            2011-01-01 00:00:00.00000

```



```

energy      33070
temperature -13.0000000000
owner       Jim
address

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

```

2 row(s) retrieved.

Duplicate timepoints

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

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

Null and hidden elements

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

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

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

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

```

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

```

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

```

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

```

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

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

```
code AA
t      2011-01-14 00:00:00.00000
high   69.25000000000
low    68.37500000000
final  68.62500000000
vol    462.00000000000
```

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

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

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

```
code AA
t      2011-01-14 00:00:00.00000
high   69.25000000000
low    68.37500000000
final  68.62500000000
vol    462.00000000000
```

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

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

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

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

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

```

order by t;

code AA
t      2011-01-14 00:00:00.00000
high  69.250000000000
low    68.375000000000
final  68.625000000000
vol    462.000000000000

code AA
t      2011-01-18 00:00:00.00000
high  69.750000000000
low    68.750000000000
final  69.625000000000
vol    281.2000100000

code AA
t      2011-01-19 00:00:00.00000
high  69.625000000000
low    69.125000000000
final  69.625000000000
vol    96.699997000000
3 row(s) retrieved.

```

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

```

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

```

```

code AA
t      2011-01-14 00:00:00.00000
high  69.250000000000
low    68.375000000000
final  68.625000000000
vol    462.000000000000

```

```

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

```

```

code AA
t      2011-01-18 00:00:00.00000
high  69.750000000000
low    68.750000000000
final  69.625000000000
vol    281.2000100000

```

```

code AA
t      2011-01-19 00:00:00.00000
high  69.625000000000
low    69.125000000000
final  69.625000000000
vol    96.699997000000
4 row(s) retrieved.

```

Related concepts:

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

“Insert data through virtual tables” on page 4-4

Related reference:

“Example of creating a fragmented virtual table” on page 4-10
“PutElemNoDups function” on page 7-79
“PutElem function” on page 7-77
“Example of creating a virtual table” on page 4-8

Drop a virtual table

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

Trace functions

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

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

The functions are:

TSSetTraceFile

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

TSSetTraceLevel

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

The TSSetTraceFile function

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

Syntax

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

traceFileName

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

Description

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

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

Returns

Returns 0 on success, -1 on failure.

Example

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

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

TSSetTraceLevel function

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

Syntax

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

traceLevelSpec

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

Description

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

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

Returns

Returns 0 on success, -1 on failure.

Example

The following example turns tracing on:

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

Chapter 5. Calendar pattern routines

You can use calendar pattern routines to manipulate calendar patterns.

Calendar pattern routines can perform the following types of operations:

- Create the intersection of calendar patterns
- Create the union of calendar patterns
- Alter a calendar pattern

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

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

Related concepts:

“Calendar” on page 1-13

Related reference:

“CalendarPattern data type” on page 2-1

AndOp function

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

Syntax

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

cal_patt1

The first calendar pattern.

cal_patt2

The second calendar pattern.

Description

This function returns a calendar pattern that has every interval on that was on in both calendar patterns; the rest of the intervals are off. If the specified 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 that are combined by the AND operator.

Example

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

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

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

select * from CalendarPatterns
      where cp_name = 'fourday_day';

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

select * from CalendarPatterns
      where cp_name = 'workweek_hour';

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

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

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

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

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

Related reference:

“AndOp function” on page 6-1

CalPattStartDate function

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

Syntax

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

calname

The name of the source calendar.

Description

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

Returns

The start date of the pattern for the specified calendar.

Example

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

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

Related reference:

“CalStartDate function” on page 6-5

“The ts_cal_pattstartdate() function” on page 9-9

Collapse function

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

Syntax

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

cal_patt

The calendar pattern to be collapsed.

interval

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

Description

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

Returns

The collapsed calendar pattern.

Example

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

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

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

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

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

Related reference:

“Expand function” on page 5-4

Expand function

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

Syntax

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

cal_patt

The calendar pattern to expand.

interval

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

Description

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

Returns

The expanded calendar pattern.

Example

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

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

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

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

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

Related reference:

“Collapse function” on page 5-3

NotOp function

The **NotOp** function turns all on intervals off and all off intervals on in the specified 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 function

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

Syntax

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

cal_patt1

The first calendar pattern.

cal_patt2

The second calendar pattern.

Description

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

Returns

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

Example

The examples use the following three calendar pattern definitions:

```
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

```

The following **OrOp** statement returns the union of two daily calendar patterns:

```

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

```

The following **OrOp** statement returns the union of one hourly and one daily calendar pattern:

```

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

```

```

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

```

Related reference:

“OrOp function” on page 6-5

Chapter 6. Calendar routines

You can use calendar routines to manipulate calendars.

Calendar routines can perform the following types of operations:

- Create the intersection of calendars
- Create the union of calendars
- Return information about the calendar

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

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

Related concepts:

“Calendar” on page 1-13

“Calendar data type” on page 2-4

AndOp function

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

Syntax

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

cal1 The first calendar.

cal2 The second calendar.

Description

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

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

Returns

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

Example

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

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

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

select c_calendar from CalendarTable
       where c_name = 'daycal';

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

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

The query returns the following results:

(expression)

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

Related reference:

“AndOp function” on page 5-1

“OrOp function” on page 6-5

CalIndex function

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

Syntax

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

cal_name

The name of the calendar.

begin_stamp

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

end_stamp

The end point of the range.

Description

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

Returns

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

Example

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

```
select CalIndex('daycal',
               '2011-01-03 00:00:00.000000',
               '2011-01-05 00:00:00.000000')
from systables
where tabid = 1;
```

Related reference:

“The `ts_cal_range()` function” on page 9-10

“The `ts_cal_range_index()` function” on page 9-11

“The `ts_cal_stamp()` function” on page 9-11

“GetIndex function” on page 7-55

“GetStamp function” on page 7-66

CalRange function

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

Syntax

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

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

cal_name

The name of the calendar.

begin_stamp

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

end_stamp

The end point of the range.

num_stamps

The number of time stamps to return.

Description

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

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

Returns

A list of time stamps.

Example

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

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

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

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

Related reference:

“The `ts_cal_range()` function” on page 9-10

“The `ts_cal_range_index()` function” on page 9-11

“The `ts_cal_stamp()` function” on page 9-11

“GetIndex function” on page 7-55

“GetStamp function” on page 7-66

CalStamp function

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

Syntax

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

cal_name

The name of the calendar.

tstamp The input time stamp.

num_stamps

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

Description

The equivalent API function is `ts_cal_stamp()`.

Returns

The time stamp that represents the specified offset.

Example

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


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

Related reference:

“Time series routines that run in parallel” on page 7-7

“The ts_cal_range() function” on page 9-10

“The ts_cal_range_index() function” on page 9-11

“The ts_cal_stamp() function” on page 9-11

CalStartDate function

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

Syntax

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

cal_name

The name of the calendar.

Description

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

Returns

The start date of the specified calendar.

Example

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

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

Related reference:

“CalPattStartDate function” on page 5-2

“The ts_cal_startdate() function” on page 9-12

OrOp function

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

Syntax

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

cal1 The first calendar to be combined.

cal2 The second calendar to be combined.

Description

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

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

Returns

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

Example

The following **OrOp** function returns the union of an hourly calendar with a daily calendar that has a different start date:

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

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

select c_calendar from CalendarTable
      where c_name = 'daycal';

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

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

The query returns the following result:

(expression)

```
startdate(2011-01-01 00:00:00), pattstart(2011-01-02
00:00:00), pattern({24 off,120 on,24 off},hour)
```

Related reference:

“OrOp function” on page 5-5

“AndOp function” on page 6-1

Chapter 7. Time series SQL routines

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

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

Type/Table	Description
stock_bar	Type containing timestamp (DATETIME), high , low , final , and vol columns
daily_stocks	Table containing stock_id , stock_name , and stock_data columns
stock_trade	Type containing timestamp (DATETIME), price , vol , trade , broker , buyer , and seller columns
activity_stocks	Table containing stock_id and activity_data columns

For more information about these data types and tables, see “Creating a TimeSeries subtype” on page 3-13 and “Create the database table” on page 3-14.

The schema for these examples is in the \$INFORMIXDIR/TimeSeries.*version*/examples directory.

Related concepts:

“Planning for accessing time series data” on page 1-24

Time series SQL routines sorted by task

Time series SQL routines are sorted into logical areas that are based on the type of task.

Table 7-1. Time series SQL routines by task type

Task type	Description
Get information from a time series	Get the origin: "GetOrigin function" on page 7-64
	Get the interval: "GetInterval function" on page 7-55
	Get the calendar: "GetCalendar function" on page 7-48
	Get the calendar name: "GetCalendarName function" on page 7-49
	Get the container name: "GetContainerName function" on page 7-51
	Get user-defined metadata: "GetMetaData function" on page 7-59
	Get the metadata type: "GetMetaTypeName function" on page 7-59
	Determine whether a time series is regular: "IsRegular function" on page 7-73
	Get the instance ID if the time series is stored in a container: "InstanceId function" on page 7-71
	Determine whether a time series contains packed data: "GetPacked function" on page 7-64
Convert between a time stamp and an offset	Get the frequency of hertz data: "GetHertz function" on page 7-54
	Get the compression type of compressed data: "GetCompression function" on page 7-50
Count the number of elements	Return the offset for the specified timestamp: "GetIndex function" on page 7-55
	Return the time stamp for the specified offset: "GetStamp function" on page 7-66
Select individual elements	Return the number of elements: "GetNelems function" on page 7-60
	Get the number of elements between two time stamps: "ClipGetCount function" on page 7-37
	Get the number of elements that match the criteria of an arithmetic expression: "CountIf function" on page 7-38
Select individual elements	Get the element associated with the specified time stamp "GetElem function" on page 7-52
	Get the element at or before a time stamp: "GetLastValid function" on page 7-58
	Get the element after a time stamp: "GetNextValid function" on page 7-61
	Get the element before a time stamp: "GetPreviousValid function" on page 7-65
	Get the element at a specified position: "GetNthElem function" on page 7-62
	Get the first element: "GetFirstElem function" on page 7-53
	Get the timestamp of the first element: "GetFirstElementStamp function" on page 7-53
	Get the last element: "GetLastElem function" on page 7-56
	Get the timestamp of the last element: "GetLastElementStamp function" on page 7-57
	Get the last non-null element: "GetLastNonNull function" on page 7-57
	Get the next non-null element: "GetNextNonNull function" on page 7-61

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description
Modify elements or a set of elements	Add or update a single element: “PutElem function” on page 7-77
	Add or update a single element: “PutElemNoDups function” on page 7-79
	Add or update a single element at a specified offset (regular only): “PutNthElem function” on page 7-80
	Add or update an entire set: “PutSet function” on page 7-80
	Insert an element: “InsElem function” on page 7-69
	Insert a set: “InsSet function” on page 7-70
	Update an element: “UpdElem function” on page 7-159
	Update a set: “UpdSet function” on page 7-160
	Put every element of one time series into another time series: “PutTimeSeries function” on page 7-82
Delete elements	Delete an element at the specified timepoint: “DelElem function” on page 7-43
	Delete all elements in a time series instance for a specified time range: “DelClip function” on page 7-42
	Delete all elements and free space in a time series instance for a specified time range in any part of a time series: “DelRange function” on page 7-44
	Delete all elements and free space in a time series instance for a specified time range at the end of a time series: “DelTrim function” on page 7-45
	Free empty pages in a specified time range or throughout the time series instance: “NullCleanup function” on page 7-75
	Delete elements through a specified timestamp from one or more containers for one or more time series instances: “TSContainerPurge function” on page 7-108
Modify metadata	Update user-defined metadata: “UpdMetaData function” on page 7-159
Make elements visible or invisible to a scan	Make an element invisible “HideElem function” on page 7-67
	Make a range of elements invisible: “HideRange function” on page 7-68
	Make an element visible: “RevealElem function” on page 7-83
	Make a range of elements visible: “RevealRange function” on page 7-83
Check for null or hidden elements	Determine whether an element is hidden: “ElemIsHidden function” on page 7-47
	Determine whether an element is null: “ElemIsNull function” on page 7-47
Extract and use part of a time series	Extract a period between two time stamps or corresponding to a set of values and run an expression or function on every entry: “Apply function” on page 7-18
	Extract data between two timepoints: “Clip function” on page 7-31
	Clip some elements: “ClipCount function” on page 7-35
	Output values in XML format: “TSToXML function” on page 7-154
	Extract a period that includes the specified time or starts or ends at the specified time: “WithinC and WithinR functions” on page 7-161
Apply a new calendar to a time series	Apply a calendar: “ApplyCalendar function” on page 7-24

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description
Create and load a time series	<p>Load data from a client file: "BulkLoad function" on page 7-30</p> <p>Create a regular empty time series, a regular populated time series, or a regular time series with metadata: "TSCreate function" on page 7-116</p> <p>Create an irregular empty time series, an irregular populated time series, or an irregular time series with metadata: "TSCreateIrr function" on page 7-118</p>
Load time series data through a loader program	<p>Initialize a loader session: "TSL_Init function" on page 7-133</p> <p>Open a database session: "TSL_Attach function" on page 7-123</p> <p>Load data into the database server: "TSL_Put function" on page 7-135</p> <p>Load data from a table into the database server: "TSL_PutSQL function" on page 7-138</p> <p>Load a row of data into the database server: "TSL_PutRow function" on page 7-137</p> <p>Flush loaded data to disk in a single transaction: "TSL_FlushAll function" on page 7-128</p> <p>Flush loaded data to disk in multiple transactions: "TSL_Commit function" on page 7-124</p> <p>View information about the last data flush operation: "TSL_FlushInfo function" on page 7-129</p> <p>Reset the logging mode: "TSL_SetLogMode function" on page 7-140</p> <p>Monitor loading and saving data: "TSL_GetLogMessage function" on page 7-132</p> <p>Get the container name for a specific primary key: "TSL_GetKeyContainer function" on page 7-131</p> <p>Close a database session: "TSL_SessionClose function" on page 7-139</p> <p>Shut down the loader session: "TSL_Shutdown procedure" on page 7-141</p>
Find the intersection or union of time series	<p>Build the intersection of multiple time series and optionally clip the result: "Intersect function" on page 7-71</p> <p>Build the union of multiple time series and optionally clip the result: "Union function" on page 7-157</p>
Iterator functions	Convert time series data to tabular form: "Transpose function" on page 7-86
Aggregate functions	<p>Return a list (collection of rows) containing all elements in a time series: "TSSetToList function" on page 7-153</p> <p>Return a list of values from the specified column name in the time series: "TSColNameToList function" on page 7-91</p> <p>Return a list of values from the specified column number in the time series: "TSColNumToList function" on page 7-92</p> <p>Return a list of values that contains the columns of the time series plus non-time-series columns: "TSRowToList function" on page 7-147</p> <p>Return a list of values from the specified column name of the time series plus non-time-series columns: "TSRowNameToList function" on page 7-145</p> <p>Return a list of values from the specified column number of the time series plus non-time-series columns: "TSRowNumToList function" on page 7-146</p>

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description
Used within the Apply function to perform statistical calculations on a time series	Sum SMALLFLOAT or DOUBLE PRECISION values: "TSAddPrevious function" on page 7-90
	Compute the decay function: "TSDecay function" on page 7-122
	Compute a running average over a specified number of values: "TSRunningAvg function" on page 7-147
	Compute a running correlation between two time series over a specified number of values: "TSRunningCor function" on page 7-149
	Compute a running median over a specified number of values: "TSRunningMed function" on page 7-150
	Compute a running sum over a specified number of values: "TSRunningSum function" on page 7-151
	Compute a running variance over a specified number of values: "TSRunningVar function" on page 7-152
	Compare SMALLFLOAT or DOUBLE PRECISION values: "TSCmp function" on page 7-90
	Return a previously saved value: "TSPrevious function" on page 7-141

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description
Perform an arithmetic operation on one or two time series	Add two time series together: "Plus function" on page 7-77
	Subtract one time series from another: "Minus function" on page 7-74
	Multiply one time series by another: "Times function" on page 7-86
	Divide one time series by another: "Divide function" on page 7-46
	Raise the first argument to the power of the second: "Pow function" on page 7-77
	Get the absolute value: "Abs function" on page 7-11
	Exponentiate the time series: "Exp function" on page 7-48
	Get the natural logarithm of a time series: "Logn function" on page 7-74
	Get the modulus or remainder of a division of one time series by another: "Mod function" on page 7-75
	Negate a time series: "Negate function" on page 7-75
	Return the argument and the argument is bound to the unary + operator: "Positive function" on page 7-77
	Round the time series to the nearest whole number: "Round function" on page 7-84
	Get the square root of the time series: "Sqrt function" on page 7-85
	Get the cosine of the time series: "Cos function" on page 7-38
	Get the sine of the time series: "Sin function" on page 7-85
	Get the tangent of the time series: "Tan function" on page 7-86
	Get the arc cosine of the time series: "Acos function" on page 7-11
	Get the arc sine of the time series: "Asin function" on page 7-27
	Get the arc tangent of the time series: "Atan function" on page 7-27
	Get the arc tangent for two time series: "Atan2 function" on page 7-27
Apply an arithmetic operation on one or more time series	Apply a binary function to a pair of time series, or to a time series and a compatible row type or number: "ApplyBinaryTsOp function" on page 7-23
	Apply a unary function to a time series: "ApplyUnaryTsOp function" on page 7-26
	Apply another function to a set of time series: "ApplyOpToTsSet function" on page 7-25
Aggregate time series values	Aggregate values in a time series from a single row: "AggregateBy function" on page 7-11
	Aggregate values in a time series from a single row over a specified time range: "AggregateRange function" on page 7-15
	Aggregate time series values across multiple rows: "TSRollup function" on page 7-142
Create a time series that lags	Create a time series that lags the source time series by a specified offset (regular only): "Lag function" on page 7-73
Reset the origin	Reset the origin: "SetOrigin function" on page 7-85

Table 7-1. Time series SQL routines by task type (continued)

Task type	Description
Manage containers	Create a container: “TSContainerCreate procedure” on page 7-93
	Delete a container: “TSContainerDestroy procedure” on page 7-98
	Set the container name: “SetContainerName function” on page 7-84
	Specify the container pool for inserting data into a time series: “TSContainerPoolRoundRobin function” on page 7-107
	Add a container into a container pool or remove a container from a container pool: “TSContainerSetPool procedure” on page 7-111
	Delete elements through a specified timestamp from one or more containers: “TSContainerPurge function” on page 7-108
	Control whether multiple sessions write to the container at the same time: “TSContainerLock procedure” on page 7-99
	Change the properties of rolling windows containers: “TSContainerManage function” on page 7-99
Monitor containers	Change the extent sizes of containers: “TSContainerManage function” on page 7-99
	Return the number of elements in one or all containers: “TSContainerNElems function” on page 7-104
	Return the percentage of space that is used in one or all containers: “TSContainerPctUsed function” on page 7-105
	Return the total number of pages that are allocated to one or all containers: “TSContainerTotalPages function” on page 7-112
	Return the number of pages that are used by one or all containers: “TSContainerTotalUsed function” on page 7-113
	Return the number of elements, the number of pages that are used, and the total number of pages that are allocated for one or all containers: “TSContainerUsage function” on page 7-114

Time series routines that run in parallel

Some time series routines can run in parallel. Running in parallel is faster than running serially.

For time series routines to run in parallel, the following conditions are required:

- The table that contains the time series data is fragmented, by any fragmentation method.
- Parallel database queries are enabled: the PDQPRIORITY environment variable is set to a value other than OFF and the MAX_PDQPRIORITY configuration parameter is set to a value other than 0 or 1.
- The routine is referenced in the WHERE clause of a SELECT statement. Routines that are referenced in the Projection clause are not run in parallel.
- The routine must select data from a table that contains time series data or a time series system table. Routines that insert, update, or delete data do not run in parallel.

- The routine cannot call another routine that cannot run in parallel. If a routine that can run in parallel calls another routine that cannot run in parallel, neither routine runs in parallel.
- The routine cannot include a collection data type as an argument or return value. For example, if the **Intersect** function includes a parameter that has a SET data type, the function is not run in parallel.

The routines in the following list can run in parallel.

Related reference:

 MAX_PDQPPRIORITY configuration parameter (Administrator's Reference)

 PDQPPRIORITY environment variable (SQL Reference)

"Create the database table" on page 3-14

"Abs function" on page 7-11

"Acos function" on page 7-11

"AggregateBy function" on page 7-11

"AggregateRange function" on page 7-15

"ApplyBinaryTsOp function" on page 7-23

"ApplyCalendar function" on page 7-24

"ApplyUnaryTsOp function" on page 7-26

"Asin function" on page 7-27

"Atan function" on page 7-27

"Atan2 function" on page 7-27

"CalStamp function" on page 6-4

"Clip function" on page 7-31

"ClipCount function" on page 7-35

"ClipGetCount function" on page 7-37

"Cos function" on page 7-38

"CountIf function" on page 7-38

"DelClip function" on page 7-42

"DelElem function" on page 7-43

"DelRange function" on page 7-44

"DelTrim function" on page 7-45

"Divide function" on page 7-46

"ElemIsHidden function" on page 7-47

"ElemIsNull function" on page 7-47

"Exp function" on page 7-48

"GetCalendar function" on page 7-48

"GetCalendarName function" on page 7-49

"GetClosestElem function" on page 7-49

"GetContainerName function" on page 7-51

"GetElem function" on page 7-52

"GetFirstElem function" on page 7-53

"GetFirstElementStamp function" on page 7-53

"GetIndex function" on page 7-55

"GetInterval function" on page 7-55

"GetLastElem function" on page 7-56

"GetLastElementStamp function" on page 7-57
 "GetLastNonNull function" on page 7-57
 "GetLastValid function" on page 7-58
 "GetNelems function" on page 7-60
 "GetNextNonNull function" on page 7-61
 "GetNextValid function" on page 7-61
 "GetNthElem function" on page 7-62
 "GetOrigin function" on page 7-64
 "GetPreviousValid function" on page 7-65
 "GetStamp function" on page 7-66
 "GetThreshold function" on page 7-67
 "HideElem function" on page 7-67
 "HideRange function" on page 7-68
 "Intersect function" on page 7-71
 "IsRegular function" on page 7-73
 "Lag function" on page 7-73
 "Logn function" on page 7-74
 "Minus function" on page 7-74
 "Mod function" on page 7-75
 "Negate function" on page 7-75
 "NullCleanup function" on page 7-75
 "Plus function" on page 7-77
 "Positive function" on page 7-77
 "Pow function" on page 7-77
 "RevealElem function" on page 7-83
 "RevealRange function" on page 7-83
 "Round function" on page 7-84
 "Sin function" on page 7-85
 "Sqrt function" on page 7-85
 "Tan function" on page 7-86
 "Times function" on page 7-86
 "TSToXML function" on page 7-154
 "Union function" on page 7-157
 "WithinC and WithinR functions" on page 7-161

The flags argument values

The time series SQL functions that insert data have a *flags* argument to determine how elements are inserted.

The value of the *flags* argument is the sum of the flag values that you want to use.

Table 7-2. The values of the *flags* argument

Flag	Value	Meaning	Restrictions
TSOPEN_RDWRITE	0	(Default) Indicates that the time series can be read and written to.	

Table 7-2. The values of the flags argument (continued)

Flag	Value	Meaning	Restrictions
TSOPEN_READ_HIDDEN	1	Indicates that hidden elements are treated as if they are not hidden.	Cannot be used in combination with the following flag values: <ul style="list-style-type: none"> • TSOPEN_WRITE_AND_HIDE • TSOPEN_WRITE_AND_REVEAL • TSOPEN_WRITE_HIDDEN
TSOPEN_WRITE_HIDDEN	2	Allows hidden elements to be written to without first revealing them. The element remains hidden afterward.	Cannot be used in combination with the following flag values: <ul style="list-style-type: none"> • TSOPEN_WRITE_AND_HIDE • TSOPEN_WRITE_AND_REVEAL • TSOPEN_READ_HIDDEN
TSOPEN_WRITE_AND_HIDE	4	Marks as hidden any elements that are written to a time series.	Cannot be used in combination with the following flag values: <ul style="list-style-type: none"> • TSOPEN_WRITE_HIDDEN • TSOPEN_WRITE_AND_REVEAL • TSOPEN_READ_HIDDEN
TSWRITE_AND_REVEAL	8	Reveals any hidden element that is written to.	
TSOPEN_NO_NULLS	32	Prevents elements that were never allocated (TS_NULL_NOTALLOCATED) from being returned as NULL. By default, if an element is not allocated, it is returned as NULL. If this flag is set, an element that has each column set to NULL is returned instead.	
TS_PUTELEM_NO_DUPS	64	Prevents duplicate elements. By default, the function adds elements with the PutElem function. If this flag is set, the function uses the PutElemNoDups function.	Can be used only in the following functions: <ul style="list-style-type: none"> • BulkLoad • PutTimeSeries • PutSet
TSOPEN_REDUCED_LOG	256	Reduces how many log records are generated when you insert data. By default, every element that you insert generates two log records: one for the inserted element and one for the page header update. If this flag is set, page header updates are logged per transaction instead of per element, which improves performance.	Can be used only in the following functions: <ul style="list-style-type: none"> • BulkLoad • InsElem • PutElem • PutElemNoDups • PutNthElem • PutTimeSeries <p>Functions that include this flag must be run within a transaction. The transaction can include other functions that use this flag. The transaction cannot include functions that do not use this flag or other SQL statements. The elements that are inserted are not visible by dirty reads until after the transaction commits.</p>

Abs function

The **Abs** function returns the absolute value of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt** and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Acos function

The **Acos** function returns the arc cosine of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

AggregateBy function

The **AggregateBy** function aggregates the values in a time series using a new time interval that you specify by providing a calendar.

This function can be used to convert a time series with a small interval to a time series with a larger interval: for instance, to produce a weekly time series from a daily time series.

If you supply the optional *start* and *end* DATETIME parameters, just that part of the time series is aggregated to the new time interval.

Syntax

```
AggregateBy(agg_express    lvarchar,  
           cal_name       lvarchar,  
           ts             TimeSeries  
           flags          integer default 0  
           start datetime year to fraction(5) default NULL,  
           end datetime year to fraction(5) default NULL  
)  
returns TimeSeries;
```

agg_express

A comma-separated list of these SQL aggregate operators: MIN, MAX, MEDIAN, SUM, AVG, FIRST, LAST, or Nth. The MIN, MAX, MEDIAN, SUM, and AVG expressions can operate only on numeric columns.

cal_name

The name of a calendar that defines the aggregation period.

ts

The time series to be aggregated.

flags (**optional**)

Determines how data points in off periods of calendars are handled during aggregation. See “The flags argument values” on page 7-13.

start (**optional**)

The date and time at which to start aggregation.

end (optional)

The date and time at which to end aggregation.

Description

The **AggregateBy** function converts the input time series to a regular time series with a calendar given by the *cal_name* argument.

The *agg_express* expressions operate on a column of the input time series, which is specified by one of the following column identifiers:

\$colname

The *colname* is the name of the column to aggregate in the **TimeSeries** data type. For example, if the column name is **high**, the column identifier is *\$high*.

\$colnumber

The *colnumber* is the position of the column to aggregate in the **TimeSeries** data type. For example if the column number is 1, the column identifier is *\$1*.

\$bson_field_name

The *bson_field_name* is the name of a field in at least one BSON document in the BSON column in the **TimeSeries** data type. For example, if the field name is **v1**, the column identifier is *\$v1*. If the BSON field name is the same as another column in the **TimeSeries** data type, you must qualify the field name in one of the following ways:

- *\$colname.bson_field_name*

For example, if the BSON column name is **b_data** and the field name is **v1**, the column identifier is *\$b_data.v1*.

- *\$colnumber.bson_field_name*

For example, if the BSON column number is 1 and the field name is **v1**, the column identifier is *\$1.v1*.

You must cast the results of the **AggregateBy** function on a BSON field to a **TimeSeries** data type that has the appropriate type of columns for the result of the expression.

The *Nth* expression returns the value of a column for the specified aggregation period, using the following syntax:

Nth(\$col, n)

\$col The column identifier.

n A positive or negative number that indicates the position of the **TimeSeries** row within the aggregation period. Positive values of *n* begin at the first row in the aggregation period; therefore, *Nth(\$col, 1)* is equivalent to *FIRST(\$col)*. Negative values of *n* begin with the last row in the aggregation period; therefore, *Nth(\$col, -1)* is equivalent to *LAST(\$col)*.

If an aggregation period does not have a value for the *n*th row, then the *Nth* function returns a null value for that period. The *Nth* function is more efficient for positive values of the *n* argument than for negative values.

An aggregation time period is denoted by the start date and time of the period.

The origin of the aggregated output time series is the first period on or before the origin of the input time series. Each output period is the aggregation of all input periods from the start of the output period up to, but not including, the start of the next output period.

For instance, suppose you want to aggregate a daily time series that starts on Tuesday, Jan. 4, 2011, to a weekly time series. The input calendar, named “days,” starts at 12:00 a.m., and the output calendar, named “weeks,” starts at 12:00 a.m., on Monday.

The first output time is 00:00 Jan. 3, 2011; it is the aggregation of all input values from the input origin, Jan. 4, 2011, to 23:59:59.99999 Jan. 9, 2011. The second output time is 00:00 Jan. 10, 2011; it is the aggregation of all input values from 00:00 Jan 10, 2011 to 23:59:59.99999 Jan. 16, 2011.

Typically, the **AggregateBy** function is used to aggregate from a fine-grained regular time series to a coarser-grained one. However, the following scenarios are also supported:

- Converting from a regular time series to a time series with a calendar of the same granularity. In this case, **AggregateBy** shifts the times back to accommodate differences in the calendar start times: for example, 00:00 from 8:00. Elements can be removed or null elements added to accommodate differences in the on/off pattern.
- Converting from a regular time series to one with a calendar of finer granularity. In this case, **AggregateBy** replicates values.
- The input time series is irregular. Because the granularity of an irregular time series does not depend on the granularity of the calendar, this case is treated like aggregation from a fine-grained time series to a coarser-grained one. This type of aggregation always produces a regular time series.

The flags argument values

The *flags* argument determines how data points in the off periods of calendars are handled during aggregation and how hidden elements are managed. It can have the following values.

- | | |
|---|--|
| 0 | (Default) Data in off periods is aggregated with the next output period. |
| 1 | Data in off periods is aggregated with the previous output period. |
| 2 | Indicates that the scan runs with the TS_SCAN_HIDDEN flag set (hidden elements are returned). |
| 4 | Indicates that the scan runs with the TS_SCAN_SKIP_HIDDEN flag set (hidden elements are not returned). |

For example, consider an input time series that has a daily calendar with no off days: it has data from weekdays and weekends. If you aggregate this data by a business-day calendar (5 days on, 2 days off, starting on a Monday), a *flags* argument of 0 causes weekend data to be aggregated with the next Monday's data, and a *flags* argument of 1 causes weekend data to be aggregated with the previous Friday's data.

For another example, consider a quarterly calendar that is defined as:

```
'startdate(2010-1-1 00:00:00.00000), pattstart(2010-1-1 00:00:00.00000),  
pattern({1 on, 2 off}, month'
```

If you aggregate this calendar with either a *flags* argument of 0 or no *flags* argument, all input points up to, but not including, 2010-2-1 00:00:00.00000 are aggregated into the first output element. All points from 2010-2-1 00:00:00.00000 up to, but not including, 2010-5-1 00:00:00.00000 are aggregated into the second output element, and so on.

If the *flags* argument is 1, all input points up to but not including 2010- 4-1 00:00:00.00000 are aggregated into the first output element. All points from 2010-4-1 00:00:00.00000 up to, but not including, 2010-7-1 00:00:00.00000 are aggregated into the second output element, and so on. The **AggregateBy** clause might look like this:

```
AggregateBy('max($high)', 'quarterlycal', ts, 1);
```

Returns

The aggregated time series, which is always regular, if you are aggregating to a new time interval. The resulting time series has a time stamp column plus one column for each expression in the list.

Examples: Stock data

The following query aggregates the **daily_stocks** time series to a weekly time series:

```
insert into daily_stocks( stock_id, stock_name, stock_data)
select stock_id, stock_name,
AggregateBy('max($high), min($low),last($final),sum($vol)',
'weekcal', stock_data)::TimeSeries(stock_bar)
from daily_stocks;
```

The following query clause selects the second price from each week:

```
AggregateBy( 'Nth($price, 2)', 'weekly', ts)
```

This query clause selects the second to the last price from each week:

```
AggregateBy( 'Nth($price, -2)', 'weekly', ts)
```

Examples: BSON data

This example is based on the following row type and time series definition. The **TimeSeries** row type contains an **INTEGER** column that is named **v1** and the **BSON** column contains a field that is also named **v1**.

```
CREATE ROW TYPE rb(timestamp datetime year to fraction(5), data bson, v1 int);
```

```
INSERT INTO tj VALUES(1,'origin(2011-01-01 00:00:00.00000), calendar(ts_15min),
container(kontainer),threshold(0), regular,[{"v1":99},20]');
```

The following statement creates a **TimeSeries** data type to hold the results of the aggregation on the **BSON** field in an **INTEGER** column:

```
CREATE ROW TYPE outrow(timestamp datetime year to fraction(5), x int);
```

If a column and a **BSON** field have the same name, the column takes precedence. The following statement returns the maximum value from the **v1** **INTEGER** column:

```
SELECT AggregateBy('max($v1)', 'ts_1year', tsdata, 0
"2011-01-01 00:00:00.00000"::datetime year to fraction(5),
"2012-01-01 00:00:00.00000"::datetime year to fraction(5))
FROM tj;
```


The following two equivalent statements return the maximum value from the **v1** field in the **data** BSON column, which is column 1 in the **TimeSeries** row type:

```
SELECT AggregateBy('max($data.v1)', 'ts_1year', tsdata, 0
    "2011-01-01 00:00:00.000000"::datetime year to fraction(5),
    "2012-01-01 00:00:00.000000"::datetime year to fraction(5))
    ::timeseries(outrow)
FROM tj;

SELECT AggregateBy('max($1.v1)', 'ts_1year', tsdata, 0
    "2011-01-01 00:00:00.000000"::datetime year to fraction(5),
    "2012-01-01 00:00:00.000000"::datetime year to fraction(5))
    ::timeseries(outrow)
FROM tj;
```

The aggregated time series that is returned has the **TimeSeries** data type **outrow**. If you do not cast the result to a row type that has appropriate columns for the results, the statement fails.

Related reference:

“Time series routines that run in parallel” on page 7-7
 “TSRollup function” on page 7-142
 “AggregateRange function”
 “Apply function” on page 7-18
 “PutTimeSeries function” on page 7-82

AggregateRange function

The **AggregateRange** function produces an aggregate over each element for a time range that is specified by *start* and *end* DATETIME parameters.

Syntax

```
AggregateRange(agg_express lvarchar,
               ts           TimeSeries
               flags       integer default 0
               start       datetime year to fraction(5) default NULL,
               end         datetime year to fraction(5) default NULL
               )
returns row;
```

agg_express

A comma-separated list of these SQL aggregate operators: MIN, MAX, MEDIAN, SUM, AVG, FIRST, LAST, or Nth. The MIN, MAX, MEDIAN, SUM, and AVG expressions can operate only on numeric columns.

ts The time series to be aggregated.

flags (optional)

See “The flags argument values” on page 7-16.

You cannot use a *flags* argument value of 1 with this function.

start (optional)

The date and time at which to start aggregation.

end (optional)

The date and time at which to end aggregation.

Description

The **AggregateRange** function converts the input section of a time series to a row of aggregate values.

The *agg_express* expressions operate on a column of the input time series, which is specified by one of the following column identifiers:

\$colname

The *colname* is the name of the column to aggregate in the **TimeSeries** data type. For example, if the column name is **high**, the column identifier is **\$high**.

\$colnumber

The *colnumber* is the position of the column to aggregate in the **TimeSeries** data type. For example, if the column number is 1, the column identifier is **\$1**.

\$bson_field_name

The *bson_field_name* is the name of a field in at least one BSON document in the BSON column in the **TimeSeries** data type. For example, if the field name is **v1**, the column identifier is **\$v1**. If the BSON field name is the same as another column in the **TimeSeries** data type, you must qualify the field name in one of the following ways:

- *\$colname.bson_field_name*

For example, if the BSON column name is **b_data** and the field name is **v1**, the column identifier is **\$b_data.v1**.

- *\$colnumber.bson_field_name*

For example, if the BSON column number is 1 and the field name is **v1**, the column identifier is **\$1.v1**.

You must cast the results of the **AggregateRange** function on a BSON field to a **TimeSeries** data type the appropriate type of columns for the result of the expression, for example a timestamp column and an INTEGER column.

The **Nth** expression returns the value of a column for the specified aggregation period, using the following syntax:

Nth(\$col, n)

\$col The column identifier.

n A positive or negative number indicating the position of the TimeSeries row within the aggregation period. Positive values of *n* begin at the first row in the aggregation period; therefore, **Nth(\$col, 1)** is equivalent to **FIRST(\$col)**. Negative values of *n* begin with the last row in the aggregation period; therefore, **Nth(\$col, -1)** is equivalent to **LAST(\$col)**.

If an aggregation period does not have a value for the *n*th row, then the **Nth** function returns a null value for that period. The **Nth** function is more efficient for positive values of the *n* argument than for negative values.

An aggregation time period is denoted by the start date and time of the period.

The flags argument values

The *flags* argument determines how data points in the off periods of calendars are handled during aggregation and how hidden elements are managed. It can have the following values.

0 (default)

Data in off periods is aggregated with the next output period.

- 2 Indicates that the scan runs with the TS_SCAN_HIDDEN flag set (hidden elements are returned).
- 4 Indicates that the scan runs with the TS_SCAN_SKIP_HIDDEN flag set (hidden elements are not returned).

Returns

A single element (row).

Example: Stock data

The following example produces an average of the values in the column **high** of the time series called **stock_data**. First, the example creates the row type, *elemval*, as a cast for the result.

```
create row type elemval (timestamp datetime year to fraction(5),
                        high double precision);
```

```
select
  AggregateRange('avg($high)', stock_data)::elemval
from daily_stocks;
```

Examples: BSON data

This example is based on the following row type and time series definition. The **TimeSeries** row type contains an INTEGER column that is named **v1** and the BSON column contains a field that is also named **v1**.

```
CREATE ROW TYPE rb(timestamp datetime year to fraction(5), data bson, v1 int);
```

```
INSERT INTO tj VALUES(1,'origin(2011-01-01 00:00:00.000000), calendar(ts_15min),
container(kontainer),threshold(0), regular,[{"v1":99},20]');
```

The following statement creates a **TimeSeries** data type for the results:

```
CREATE ROW TYPE outrow(timestamp datetime year to fraction(5), x int);
```

If a column and a BSON field have the same name, the column takes precedence. The following statement returns the maximum value from the **v1** INTEGER column:

```
SELECT AggregateRange('max($v1)', 'ts_1year', tsdata, 0
  "2011-01-01 00:00:00.000000"::datetime year to fraction(5),
  "2012-01-01 00:00:00.000000"::datetime year to fraction(5))
FROM tj;
```

The following two equivalent statements return the maximum value from the **v1** field in the **data** BSON column, which is column 1 in the **TimeSeries** row type:

```
SELECT AggregateRange('max($data.v1)', 'ts_1year', tsdata, 0
  "2011-01-01 00:00:00.000000"::datetime year to fraction(5),
  "2012-01-01 00:00:00.000000"::datetime year to fraction(5))
::outrow
FROM tj;
```

```
SELECT AggregateRange('max($1.v1)', 'ts_1year', tsdata, 0
  "2011-01-01 00:00:00.000000"::datetime year to fraction(5),
  "2012-01-01 00:00:00.000000"::datetime year to fraction(5))
::outrow
FROM tj;
```

The aggregated time series that is returned has the **TimeSeries** data type **outrow**. If you do not cast the result to a row type that has the appropriate columns for the results, the statement fails.

Related reference:

“Time series routines that run in parallel” on page 7-7

“AggregateBy function” on page 7-11

“Apply function”

Apply function

The **Apply** function queries one or more time series and applies a user-specified SQL expression or function to the selected time series elements.

Syntax

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

```
Apply(sql_express lvarchar,  
      multiset_ts multiset(TimeSeries))  
returns TimeSeries;
```

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

```
Apply(sql_express lvarchar,  
      filter lvarchar,  
      multiset_ts multiset(TimeSeries))  
returns TimeSeries;
```

```
Apply(sql_express lvarchar,  
      begin_stamp datetime year to fraction(5),  
      end_stamp datetime year to fraction(5),  
      ts TimeSeries, ...)  
returns TimeSeries with (handlesnulls);
```

```
Apply(sql_express lvarchar,  
      begin_stamp datetime year to fraction(5),  
      end_stamp datetime year to fraction(5),  
      multiset_ts multiset(TimeSeries))  
returns TimeSeries with (handlesnulls);
```

```
Apply(sql_express lvarchar,  
      filter lvarchar,  
      begin_stamp datetime year to fraction(5),  
      end_stamp datetime year to fraction(5),  
      ts TimeSeries, ...)  
returns TimeSeries with (handlesnulls);
```

```
Apply(sql_express lvarchar,  
      filter lvarchar,  
      begin_stamp datetime year to fraction(5),  
      end_stamp datetime year to fraction(5),  
      multiset_ts multiset(TimeSeries))  
returns TimeSeries with (handlesnulls);
```

sql_express

The SQL expression or function to evaluate.

filter The filter expression used to select time series elements.

begin_stamp

The begin point of the range. See “Clip function” on page 7-31 for more detail about range specifications.

end_stamp

The end point of the range. See “Clip function” on page 7-31 for more detail about range specifications.

ts

The first *ts* argument is the first series, the second *ts* argument is the second series, and so on. This function can take up to eight *ts* arguments. The order of the arguments must correspond to the desired order in the SQL expression or function. There is no limit to the number of \$ parameters in the expression.

multiset_ts

A multiset of time series.

Description

This function runs a user-specified SQL expression on the given time series and produces a new time series containing the result of the expression at each qualifying element of the input time series.

You can qualify the elements from the input time series by specifying a time period to clip and by using a filter expression.

The *sql_express* argument is a comma-separated list of expressions to run for each selected element. There is no limit to the number of expressions you can run. The results of the expressions must match the corresponding columns of the result time series minus the first time stamp column. Do not specify the first time stamp as the first expression; the first time stamp is generated for each expression result.

The parameters to the expression can be an input element or any column of an input time series. You should use \$, followed by the position of a given time series on the input time series list to represent its data element, plus a dot, then the number of the column. Both the position number and column number are zero-based.

For example, **\$0** means the element of the first input time series, **\$0.0** represents its time stamp column, and **\$0.1** is the column following the time stamp column. Another way to refer to a column is to use the column name directly, instead of the column number. Suppose the second time series has a column called **high** then you can use **\$1.high** to refer to it. If the **high** column is the second column in the element, **\$1.high** is equivalent to **\$1.1**.

If **Apply** has only one time series argument, you can refer to the column name without the time series position part; hence, **\$0.high** is the same as **\$high**. Notice that **\$0** always means the whole element of the first time series. It does *not* mean the first column of the time series, even if there is only one time series argument.

If you use a function as your expression, then it must take the subtype of each input time series in that order as its arguments and return a row type that corresponds to the subtype of the result time series of **Apply**. In most cases, it is faster to evaluate a function than to evaluate a generic expression. If performance is critical, you should implement the calculation to be performed in a function and use the function syntax. See “Example” on page 7-21 for how to achieve this.

The following examples show valid expressions for **Apply** to apply. Assume two argument time series with the same subtype **daybar**(t DATETIME YEAR TO FRACTION(5), **high** REAL, **low** REAL, **close** REAL, **vol** REAL). The expression could be any of:

- "\$0.high + \$1.high)/2, (\$0.low + \$1.low)/2"
- "(\$0.1 + \$1.1)/2, (\$0.2 + \$1.2)/2"
- "\$0.high, \$1.high"
- "avghigh"

The signature of **avghigh** is:

"avghigh(arg1 daybar, arg2 daybar) returns (one_real)"

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

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

Apply with the *multiset_ts* argument assigns parameter numbers by fetching **TimeSeries** values from the set and processing them in the order in which they are returned by the set management code. Since sets are unordered, parameters might not be assigned numbers predictably. **Apply** with the *multiset_ts* argument is useful only if you can guarantee that the **TimeSeries** values are returned in a fixed order. There are two ways to guarantee this:

- Write a C function that creates the set and use the function as the *multiset_ts* argument to **Apply**. The C function can return the **TimeSeries** values in any order you want.
- Use ORDER BY in the *multiset_ts* expression

Apply with the *multiset_ts* argument evaluates the expression once for every timepoint in the resulting union of time series values. When all the data in the clipped period has been exhausted, **Apply** returns the resulting series.

Apply uses the optional clip time range to restrict the data to a particular time period. If the beginning timepoint is NULL, then **Apply** uses the earliest valid timepoint of all the input time series. If the ending timepoint is NULL, then **Apply** uses the latest valid timepoint of all the input time series. When the optional clip time range is not used, it is equivalent to both the beginning and ending timepoints being NULL: **Apply** considers all elements.

If both the clip time range and filter expression are given, then clipping is done before filtering.

If you use a string literal or NULL for the clip time range, you should cast to DATETIME YEAR TO FRACTION(5) on at least the beginning timepoint to avoid ambiguity in function resolution.

When more than one input time series is specified, a union of all input time series is performed to produce the source of data to be filtered and evaluated by **Apply**. Hence, **Apply** acts as a union function, with extra filtering and manipulation of union results. For details on how the **Union** function works, see "Union function" on page 7-157.

Returns

A new time series with the results of evaluating the expression on every selected element from the source time series.

Example

The following example uses **Apply** without a filter argument and without a clipped range:

```
select Apply('$high-$low',
             datetime(2011-01-01) year to day,
             datetime(2011-01-06) year to day,
             stock_data)::TimeSeries(one_real)
from daily_stocks
where stock_name = 'IBM';
```

The following example shows **Apply** without a filter and with a clipped range:

```
select Apply(
    '($0.high+$1.high)/2, ($0.low+$1.low)/2, ($0.final+$1.final)/2,
    ($0.vol+$1.vol)/2',
    datetime(2011-01-04) year to day,
    datetime(2011-01-05) year to day,
    t1.stock_data, t2.stock_data)
::TimeSeries(stock_bar)
from daily_stocks t1, daily_stocks t2
where t1.stock_name = 'IBM' and t2.stock_name = 'HWP';
```

The following example shows **Apply** with a filter and without a clip range. The resulting time series contains the closing price of the days that the trading range is more than 10% of the low:

```
create function ts_sum(a stock_bar)
returns one_real;
return row(null::datetime year to fraction(5),
(a.high + a.low + a.final + a.vol))::one_real;
end function;

select Apply('ts_sum',
             '2011-01-03 00:00:00.000000'::datetime year
             to fraction(5),
             '2011-01-03 00:00:00.000000'::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;

    rowdesc = row->desc;
    result = row->data;
    values[0] = rowdesc->high;
    values[1] = rowdesc->low;
    nulls[0] = rowdesc->highnull;
    nulls[1] = rowdesc->lownull;
    r = 0;
    len = rowdesc->len;
    conn = row->conn;
    rc = 0;
    for (i = 0; i < len; i++)
    {
        if (nulls[0] && nulls[1])
            continue;
        r = (mi_real) *values[0] - (mi_real) *values[1];
        if (r > 0.1 * (mi_real) *values[1])
            result[i] = r;
    }
    return result;
}
```

```

    nulls[0] = MI_TRUE;
    nulls[1] = MI_FALSE;
    conn = mi_open(NULL,NULL,NULL);
    if ((rc = mi_value(row, 1, (MI_DATUM *) &high,
        &len)) == MI_ERROR)
    mi_db_error_raise(conn, MI_EXCEPTION,
        "ts_test_float_sql: corrupted argument row");
    if (rc == MI_NULL_VALUE)
    goto retisnull;

    if ((rc = mi_value(row, 2, (MI_DATUM *) &low,
        &len)) == MI_ERROR)
    mi_db_error_raise(conn, MI_EXCEPTION,
        "ts_test_float_sql: corrupted argument row");
    if (rc == MI_NULL_VALUE)
    goto retisnull;

    r = *high - *low;
    values[1] = (void *) &r;
    rowdesc = mi_row_desc_create(mi_typestring_to_id(conn,
        "one_real"));
    result = mi_row_create(conn, rowdesc, (MI_DATUM *)
        values, nulls);
    mi_close(conn);
    return (result);
retisnull:
    mi_fp_setreturnisnull(fp, 0, MI_TRUE);
    return (MI_ROW *) NULL;
}
/* end of applyfunc.c */

```

Then create the following SQL function:

```

create function HighLowDiff(arg stock_bar) returns one_real
external name '/tmp/applyfunc.bld(high_low_diff)'
language C;

```

```

select stock_name, Apply('HighLowDiff',
    stock_data)::TimeSeries(one_real)
from daily_stocks;

```

The following query is equivalent to the previous query, but it does not have the performance advantages of using a function as the expression to evaluate:

```

select stock_name, Apply('$high - $low',
    stock_data)::TimeSeries(one_real)
from daily_stocks;

```

Related reference:

- “AggregateBy function” on page 7-11
- “AggregateRange function” on page 7-15
- “Clip function” on page 7-31
- “ClipCount function” on page 7-35
- “ClipGetCount function” on page 7-37
- “Intersect function” on page 7-71
- “TSAddPrevious function” on page 7-90
- “TSCmp function” on page 7-90
- “TSDecay function” on page 7-122
- “TSPrevious function” on page 7-141
- “TSRunningAvg function” on page 7-147

“TSRunningSum function” on page 7-151
 “Union function” on page 7-157
 “Binary arithmetic functions” on page 7-27
 “SetOrigin function” on page 7-85
 “TSRunningCor function” on page 7-149
 “TSRunningMed function” on page 7-150
 “TSRunningVar function” on page 7-152
 “Unary arithmetic functions” on page 7-156

ApplyBinaryTsOp function

The **ApplyBinaryTsOp** function applies a binary arithmetic function to a pair of time series or to a time series and a compatible row type or number.

Syntax

```
ApplyBinaryTsOp(func_name  lvarchar,
                ts          TimeSeries,
                ts          TimeSeries)
returns TimeSeries;
```

```
ApplyBinaryTsOp(func_name  lvarchar,
                number_or_row scalar|row,
                ts          TimeSeries)
returns TimeSeries;
```

```
ApplyBinaryTsOp(func_name  lvarchar,
                ts          TimeSeries,
                number_or_row scalar|row)
returns TimeSeries;
```

func_name

The name of a binary arithmetic function.

ts

The time series to use in the operation. The second and third arguments can be a time series, a row type, or a number. At least one of the two must be a time series.

number_or_row

A number or a row type to use in the operation. The second and third arguments can be a time series, a row type, or a number. The second two arguments must be compatible under the function. See “Binary arithmetic functions” on page 7-27 for a description of the compatibility requirements.

Description

These functions operate in an analogous fashion to the arithmetic functions that have been overloaded to operate on time series. See the description of these functions in “Binary arithmetic functions” on page 7-27 for more information. For example, **Plus(ts1, ts2)** is equivalent to **ApplyBinaryTsOp(‘Plus’, ts1, ts2)**.

Returns

A time series of the same type as the first time series argument, which can result in a loss of precision. The return type can be explicitly cast to a compatible time series type with more precision to avoid this problem. See “Binary arithmetic functions” on page 7-27 for more information.

Example

The following example uses **ApplyBinaryTSOp** to implement the **Plus** function:

```
create row type simple_series( stock_id int, data TimeSeries(one_real));
create table daily_high of type simple_series;
insert into daily_high
select stock_id,
       Apply( '$0.high',
              NULL::datetime year to fraction(5),
              NULL::datetime year to fraction(5),
              stock_data)::TimeSeries(one_real)
from daily_stocks;
create table daily_low of type simple_series;
insert into daily_low
select stock_id,
       Apply( '$0.low',
              NULL::datetime year to fraction(5),
              NULL::datetime year to fraction(5),
              stock_data)::TimeSeries(one_real)
from daily_stocks;
create table daily_avg of type simple_series;
insert into daily_avg
select l.stock_id, ApplyBinaryTSOp("plus", l.data, h.data)/2
from daily_low l, daily_high h
where l.stock_id = h.stock_id;
```

You can receive the same results by substituting **(l.data + h.data)** for **ApplyBinaryTSOp('plus', l.data, h.data)**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“ApplyOpToTsSet function” on page 7-25

“Binary arithmetic functions” on page 7-27

ApplyCalendar function

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

Syntax

```
ApplyCalendar (ts      TimeSeries,
               cal_name lvarchar,
               flags    integer default 0)
returns TimeSeries;
```

ts The specified time series from which specific timepoints are projected.

cal_name The name of the calendar to apply.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

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';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

ApplyOpToTsSet function

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

Syntax

```
ApplyOpToTsSet(func_name   lvarchar,
               multiset_ts multiset(TimeSeries))
returns TimeSeries;
```

func_name

The name of a binary function. See “Binary arithmetic functions” on page 7-27 for more information.

multiset_ts

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

Description

All the time series must have the same type. If the multiset is empty, then **ApplyOpToTsSet** returns NULL. If the multiset contains only one time series, then **ApplyOpToTsSet** returns a copy of that time series. If the multiset contains exactly two time series, **ts1** and **ts2**, then **ApplyOpToTsSet** returns **ApplyBinaryTsOp(func_name, ts1, ts2)**. If the multiset contains three time series, **ts1**, **ts2**, and **ts3**, then **ApplyOpToTsSet** returns **ApplyBinaryTsOp(func_name, ApplyBinaryTsOp(func_name, ts1, ts2), ts3)**, and so on.

Returns

A time series of the same type as the time series in the multiset. The calendar of the resulting time series is the union of the calendars of the input time series. The resulting time series is regular if all the input time series are regular and irregular if any of the inputs are irregular.

Related reference:

“ApplyBinaryTsOp function” on page 7-23

“Binary arithmetic functions” on page 7-27

ApplyUnaryTsOp function

The **ApplyUnaryTsOp** function applies a unary arithmetic function to a time series.

Syntax

```
ApplyUnaryTsOp(func_name lvarchar,  
              ts          TimeSeries)  
returns TimeSeries;
```

func_name

The name of the unary arithmetic function.

ts

The time series to act on.

Description

This function operates in an analogous fashion to the unary arithmetic functions that have been overloaded to operate on time series. See the description of these functions in the section “Unary arithmetic functions” on page 7-156 for more information. For example, **Logn(ts1)** is equivalent to **ApplyUnaryTsOp('Logn', ts1)**.

Returns

A time series of the same type as the supplied time series.

Example

The following example uses **ApplyUnaryTSOp** with the **Logn** function:

```
create row type simple_series( stock_id int, data TimeSeries(one_real));  
create table daily_high of type simple_series;  
insert into daily_high  
  select stock_id,  
         Apply( '$0.high',  
               NULL::datetime year to fraction(5),  
               NULL::datetime year to fraction(5),  
               stock_data)::TimeSeries(one_real)  
  from daily_stocks;  
create table daily_low of type simple_series;  
insert into daily_low  
  select stock_id,  
         Apply( '$0.low',  
               NULL::datetime year to fraction(5),  
               NULL::datetime year to fraction(5),  
               stock_data)::TimeSeries(one_real)  
  from daily_stocks;  
create table daily_avg of type simple_series;  
insert into daily_avg  
  select l.stock_id, ApplyBinaryTSOp("plus", l.data, h.data)/2  
  from daily_low l, daily_high h  
  where l.stock_id = h.stock_id;  
create table log_high of type simple_series;  
insert into log_high  
  select stock_id, ApplyUnaryTsOp( "logn",  
                                   data) from daily_avg;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Asin function

The **Asin** function returns the arc sine of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Atan function

The **Atan** function returns the arc tangent of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Atan2 function

The **Atan2** function returns the arc tangent of corresponding elements from two time series.

It is one of the binary arithmetic functions that work on time series. The others are **Divide**, **Minus**, **Mod**, **Plus**, **Pow**, and **Times**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions”

Binary arithmetic functions

The standard binary arithmetic functions **Atan2**, **Plus**, **Minus**, **Times**, **Divide**, **Mod**, and **Pow** can operate on time series data. The **Plus**, **Minus**, **Times**, and **Divide** functions can also be denoted by their standard operators **+**, **-**, *****, and **/**.

Syntax

```
Function(ts TimeSeries,  
        ts TimeSeries)  
returns TimeSeries;  
  
Function(number scalar,  
        ts TimeSeries)  
returns TimeSeries;  
  
Function(ts TimeSeries,  
        number scalar)  
returns TimeSeries;  
  
Function(row row,  
        ts TimeSeries)  
returns TimeSeries;
```

```
Function(ts           TimeSeries,
        row 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 A scalar number. Must be compatible with the source time series.

row A row type. Must be compatible with the source time series.

Description

In the first format, both arguments are time series. The result is a time series that starts at the later of the starting times of the inputs. The end point of the result is the later of the two input end points if both inputs are irregular. The result end point is the earlier of the input regular time series end points if one or more of the inputs is a regular time series. The result time series has one time point for each input time point in the interval.

The element at time *t* in the resulting time series is formed from the last elements at or before time *t* in the two input time series. Normally the function is applied column by column to the input columns, except for the time stamp, to produce the output element. In this case, the two input row types must have the same number of columns, and the corresponding columns must be compatible under the function.

However, if there is a variant of the function that operates directly on the row types of the two input time series, then that variant is used. Then the input row types can have different numbers of columns and the columns might be incompatible. The time stamp of the resulting element is ignored; the element placed in the resulting time series has the later of the time stamps of the input elements.

The resulting calendar is the union of the calendars of the input time series. If the input calendars are the same, then the resulting calendar is the same as the input calendar. Otherwise, a new calendar is made. The name of the resulting calendar is a string that contains the names of the calendars of the input time series, separated by a vertical line (|). For example, if two time series are joined, and **mycal** and **yourcal** are the names of their corresponding calendars, the resulting calendar is named **mycal|yourcal**.

The resulting time series is regular if both the input time series are regular and irregular if either of the inputs is irregular.

One of the inputs can be a scalar number or a row type. In this case, the resulting time series has the same calendar, sequence of time stamps, and regularity as the input time series. If one of the inputs is a scalar number, then the function is applied to the scalar number and to each non-time stamp column of each element of the input time series.

If an input is a row type, then that row type must be compatible with the time series row type. The function is applied to the input row type and each element of the input time series. It is applied column by column or directly to the two row

types, depending on whether there is a variant of the function that handles the row types directly.

Returns

The same type of time series as the first time series input, unless the 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 wanted effect. **Plus(tsi, tsr)::TimeSeries(cr)** has type **TimeSeries(cr)** and does not discard the fractional parts of the resulting numbers.

Example

Suppose that you want to know the average daily value of stock prices. The following statements separate the daily high and low values for the stocks into separate time series in a **daily_high** table and a **daily_low** table:

```
create row type price( timestamp datetime year to fraction(5),
    val real);
create row type simple_series( stock_id int, data
    TimeSeries(price));
```

```
create table daily_high of type simple_series;
```

```
$insert into daily_high
select stock_id,
    Apply('$high',
        '2011-01-03 00:00:00.00000'
        ::datetime year to fraction(5),
        '2011-01-10 00:00:00.00000'
        ::datetime year to fraction(5),
        stock_data)::TimeSeries(one_real)
from daily_stocks;
```

```
create table daily_low of type simple_series;
```

```
insert into daily_low
select stock_id,
    Apply('$low',
        '2011-01-03 00:00:00.00000'
        ::datetime year to fraction(5),
        '2011-01-10 00:00:00.00000'
        ::datetime year to fraction(5),
        stock_data)::TimeSeries(price)
from daily_stocks;
```

The following query uses the symbol form of the **Plus** and **Divide** functions to produce a time series of daily average stock prices in the **daily_avg** table:

```
create table daily_avg of type simple_series;
```

```
insert into daily_avg
select l.stock_id, (l.data + h.data)/2
from daily_low l, daily_high h
where l.stock_id = h.stock_id;
```

Related reference:

“Load small amounts of data with SQL functions” on page 3-36

“ApplyBinaryIsOp function” on page 7-23

“ApplyOpToTsSet function” on page 7-25
 “Atan2 function” on page 7-27
 “Apply function” on page 7-18
 “Unary arithmetic functions” on page 7-156
 “Divide function” on page 7-46
 “Minus function” on page 7-74
 “Mod function” on page 7-75
 “Plus function” on page 7-77
 “Pow function” on page 7-77
 “Times function” on page 7-86

BulkLoad function

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

Syntax

```

BulkLoad (ts          TimeSeries,
          filename lvarchar,
          flags      integer default 0)
returns TimeSeries;
```

ts The time series in which to load data.

filename The path and file name of the file to load.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

The file is located on the client and can be an absolute or relative path name.

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

- Using type constructors
- Using tabs

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

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

```

row(2011-01-03 00:00:00.000000, 1.1, 2.2)
row(2011-01-04 00:00:00.000000, 10.1, 20.2)
```

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

```

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

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

```

2011-01-03 00:00:00.000000    1.1    2.2
2011-01-04 00:00:00.000000   10.1   20.2
```


The spaces between entries represent a tab.

In both formats, the word NULL indicates a null entry.

When **BulkLoad** encounters data with duplicate time stamps in a regular time series, the old values are replaced by the new values. In an irregular time series, when **BulkLoad** encounters data with duplicate time stamps, the following algorithm is used to determine where to place the data belonging to the duplicate time stamp:

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

This is the same algorithm as used by the **PutElem** function, described in “PutElem function” on page 7-77.

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';
```

Related reference:

“Load data with the BulkLoad function” on page 3-35

Clip function

The **Clip** function extracts data between two timepoints in a time series and returns a new time series that contains that data. You can extract periods of interest from a large time series and to store or operate on them separately from the large series.

Syntax

```
Clip(ts           TimeSeries,
     begin_stamp datetime year to fraction(5),
     end_stamp   datetime year to fraction(5),
     flag         integer default 0)
returns TimeSeries;
```

```
Clip(ts           TimeSeries,
     begin_stamp datetime year to fraction(5),
     end_offset   integer,
     flag         integer default 0)
returns TimeSeries;
```

```
Clip(ts           TimeSeries,
     begin_offset integer,
     end_stamp   datetime year to fraction(5),
     flag         integer default 0)
returns TimeSeries;
```

```
Clip(ts           TimeSeries,
```

```

    begin_offset integer,
    end_offset   integer,
    flag         integer default 0)
returns TimeSeries;

```

ts The time series to clip.

begin_stamp

The begin point of the range. Can be NULL.

end_stamp

The end point of the range. Can be NULL.

begin_offset

The begin offset of the range (regular time series only).

end_offset

The end offset of the range (regular time series only).

***flag* (optional)**

The configuration of the resulting time series. Each flag value other than 0 reverses one aspect of the default behavior. The value of the *flag* argument is the sum of the flag values that you want to use.

0 = Default behavior:

- The origin of the resulting time series is the later of the begin point and the origin of the input time series.
- Hidden elements are not included in the resulting time series.
- The resulting time series has the same regularity as the input time series.
- The first record in a resulting irregular time series has the timestamp of the begin point and the value of the first record from the input time series that is equal to or earlier than the begin point.

1 = The origin of the resulting time series is the earlier of the begin point and the origin of the input time series. For regular time series, timepoints that are before the origin of the time series are set to NULL. For irregular time series, has no effect.

2 = Hidden elements are included and kept hidden in the resulting time series.

4 = Hidden elements are included and revealed in the resulting time series.

8 = The resulting time series is irregular regardless of whether the input time series is irregular.

16 = For irregular time series, the resulting time series begins with the first record that is equal to or later than the begin point. For regular time series, has no effect.

Description

The **Clip** functions all take a time series, a begin point, and an end point for the range.

For regular time series, the begin and end points can be either integers or time stamps. If the begin point is an integer, it is the absolute offset of an entry in the time series. Data at the beginning and ending offsets is included in the resulting time series. If the begin point is a time stamp, the **Clip** function uses the calendar of the input time series to find the offset that corresponds to the time stamp. If

there is no entry in the time series exactly at the requested time stamp, **Clip** uses the time stamp that immediately follows the specified time stamp as the begin point of the range.

The end point is used in the same way as the begin point, except that it specifies the end of the range, rather than its beginning. The begin and end points can be NULL, in which case the beginning or end of the time series is used.

For irregular time series, only time stamps are allowed for the begin and end points. The timestamp of the first record in a resulting irregular time series is later than or equal to the begin point. However, the value of a record in an irregular time series persists until the next record. Therefore, by default, the first record in the resulting time series can have a value that corresponds to an earlier timestamp than the begin point. You can specify that the first record in the resulting time series is the first record whose timestamp is equal to or after the begin point by including the *flag* argument value of 16.

You can specify that the resulting time series is irregular by including the *flag* argument value of 8.

You can choose whether the origin of the resulting time series can be earlier than the origin of the input time series by setting the *flag* argument. By default, the origin of the resulting time series cannot be earlier than the origin of the input time series. You can also control how hidden elements are handled with the *flag* argument. By default, hidden elements from the input time series are not included in the resulting time series. You can include hidden element in the resulting time series and specify whether those elements remain hidden or are revealed in the resulting time series.

Returns

A new time series that contains 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.

Examples

The results of the **Clip** function are slightly different for regular and irregular time series.

Example 1: Regular time series

The following query extracts data from a time series and creates a table that contains the specified stock data for a single week:

```
create table week_1_analysis (stock_id int, stock_data
    TimeSeries(stock_bar));
insert into week_1_analysis
select stock_id,
    Clip(stock_data,
        '2011-01-03 00:00:00.00000'
        ::datetime year to fraction(5),
        '2011-01-07 00:00:00.00000'
        ::datetime year to fraction(5))
from daily_stocks
where stock_name = 'IBM';
```

The following query returns the first six entries for a specified stock in a time series:

```
select Clip(stock_data, 0, 5)
from daily_stocks
where stock_name = 'IBM';
```

Example 2: Irregular time series

An irregular time series has the following values:

```
2005-12-17 10:23:00.00000 26.46
2006-01-03 13:19:00.00000 27.30
2006-01-04 13:19:00.00000 28.67
2006-01-09 13:19:00.00000 30.56
```

The following statement extracts data from a time series over a five day period:

```
EXECUTE FUNCTION Transpose ((
  select Clip(
    tsdata,
    "2006-01-01 00:00:00.00000"::datetime year to fraction (5),
    "2006-01-05 00:00:00.00000"::datetime year to fraction (5),
    0)
  from ts_tab
  where station_id = 228820)) ;
```

The resulting irregular time series is as follows:

```
2006-01-01 00:00:00.00000 26.46
2006-01-03 13:19:00.00000 27.30
2006-01-04 13:19:00.00000 28.67
```

The first record has a time stamp equal to the begin point of the clip and the value of the first original value. Because the time series is irregular, a record persists until the next record. Therefore, the value of 26.46 is still valid on 2006-01-01.

However, if the **Clip** function includes the *flag* argument value of 16, the first value of the resulting time series is later than the begin point of the clip. The following statement extracts data that is after the begin point:

```
EXECUTE FUNCTION Transpose ((
  select Clip(
    tsdata,
    "2006-01-01 00:00:00.00000"::datetime year to fraction (5),
    "2006-01-05 00:00:00.00000"::datetime year to fraction (5),
    16)
  from ts_tab
  where station_id = 228820)) ;
```

The resulting irregular time series is as follows:

```
2006-01-03 13:19:00.00000 27.30
2006-01-04 13:19:00.00000 28.67
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Apply function” on page 7-18

“ClipCount function” on page 7-35

“ClipGetCount function” on page 7-37

“GetElem function” on page 7-52

“GetLastValid function” on page 7-58

“GetNthElem function” on page 7-62

“WithinC and WithinR functions” on page 7-161

“DelClip function” on page 7-42

“DelTrim function” on page 7-45

“SetOrigin function” on page 7-85

ClipCount function

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

Syntax

```
ClipCount(ts           TimeSeries,  
         begin_stamp datetime year to fraction(5),  
         num_stamps  integer,  
         flag        integer default 0)  
returns TimeSeries;
```

ts The time series to clip.

begin_stamp
 The begin point of the range. Can be NULL.

num_stamps
 The number of elements to be included in the resultant time series.

flag (optional)

The configuration of the resulting time series. Each flag value other than 0 reverses one aspect of the default behavior. The value of the *flag* argument is the sum of the flag values that you want to use.

0 = Default behavior:

- The origin of the resulting time series is the later of the begin point and the origin of the input time series.
- Hidden elements are not included in the resulting time series.
- The resulting time series has the same regularity as the input time series.
- The first record in a resulting irregular time series has the timestamp of the begin point and the value of the first record from the input time series that is equal to or earlier than the begin point.

1 = The origin of the resulting time series is the earlier of the begin point and the origin of the input time series. For regular time series, timepoints that are before the origin of the time series are set to NULL. For irregular time series, has no effect.

2 = Hidden elements are included and kept hidden in the resulting time series.

4 = Hidden elements are included and revealed in the resulting time series.

8 = The resulting time series is irregular regardless of whether the input time series is irregular.

16 = For irregular time series, the resulting time series begins with the first record that is equal to or later than the begin point. For regular time series, has no effect.

Description

Begin points before the time series origin are permitted. Negative counts with such time stamps result in time series with no elements. Begin points before the calendar origin are not permitted.

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

The timestamp of the first record in a resulting irregular time series is later than or equal to the begin point. However, the value of a record in an irregular time series persists until the next record. Therefore, by default, the first record in the resulting time series can have a value that corresponds to an earlier timestamp than the begin point. You can specify that the first record in the resulting time series is the first record whose timestamp is equal to or after the begin point by including the *flag* argument value of 16.

You can specify that the resulting time series is irregular by including the *flag* argument value of 8.

You can choose whether the origin of the resulting time series can be earlier than the origin of the input time series by setting the *flag* argument. By default, the origin of the resulting time series cannot be earlier than the origin of the input time series. You can also control how hidden elements are handled with the *flag* argument. By default, hidden elements from the input time series are not included in the resulting time series. You can include hidden element in the resulting time series and specify whether those elements remain hidden or are revealed in the resulting time series.

Returns

A new time series containing only data from the requested range. The new series has the same calendar as the original, but it can have a different origin and number of entries.

Example

The following example clips the first 30 elements at or after March 14, 2011, at 9:30 a.m. for the stock with ID 600, and it returns the entire resulting time series:

```
select ClipCount(activity_data,
  '2011-01-01 09:30:00.00000', 30)
  from activity_stocks
 where stock_id = 600;
```

The following example clips the previous 60 elements at or before August 22, 2011, at 12:00 midnight for the stock with ID 600:

```
select ClipCount(activity_data,
  '2011-08-22 00:00:00.00000', -60)
  from activity_stocks
 where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Apply function” on page 7-18

“Clip function” on page 7-31

“ClipCount function” on page 7-35

“ClipGetCount function”

“GetElem function” on page 7-52

“GetLastValid function” on page 7-58

“GetNthElem function” on page 7-62

ClipGetCount function

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

Syntax

```
ClipGetCount(ts TimeSeries,  
            begin_stamp datetime year to fraction(5) default NULL,  
            end_stamp   datetime year to fraction(5) default NULL,  
            flags       integer default 0)  
returns integer;
```

ts The source time series.

begin_stamp
 The begin point of the range. Can be NULL.

end_stamp
 The end point of the range. Can be NULL.

flags Valid values for the *flags* argument are described later in this topic.

Description

For an irregular time series, deleted elements are not counted. For a regular time series, only entries that are non-null are counted, so **ClipGetCount** might return a different value than **GetNelems**.

If the begin point is NULL, the time series origin is used. If the end point is NULL, the end of the time series is used.

See “Clip function” on page 7-31 for more information about the begin and end points of the range.

The flags argument values

The *flags* argument determines how a scan should work on the returned set. If you set the *flags* argument to 0 (the default), null and hidden elements are not part of the count. If the *flags* argument has a value of 512 (0x200) (the TS_SCAN_HIDDEN bit is set), all non-null elements are counted whether they are hidden or not.

Flag	Value	Meaning
TSOPEN_RDWRITE	0	(Default) Hidden elements are not included in the count.
TS_SCAN_HIDDEN	512	Hidden elements marked by HideElem are included in the count (see “HideElem function” on page 7-67).

Returns

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

Example

The following statement returns the number of elements between 10:30 a.m. on March 14, 2011, and midnight on March 19, 2011, inclusive:

```
select ClipGetCount(activity_data,
    '2011-03-14 10:30:00.000000','2011-03-19 00:00:00.000000')
    from activity_stocks
    where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Apply function” on page 7-18

“Clip function” on page 7-31

“ClipCount function” on page 7-35

“GetIndex function” on page 7-55

“GetNelems function” on page 7-60

“GetNthElem function” on page 7-62

“GetStamp function” on page 7-66

“The ts_nelems() function” on page 9-41

Cos function

The **Cos** function returns the cosine of its argument.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

CountIf function

The **CountIf** function counts the number of elements that match the criteria of a simple arithmetic expression.

Syntax

```
CountIf (
    ts      TimeSeries,
    expr     lvarchar,
    begin_stamp datetime year to fraction(5) default null,
    end_stamp datetime year to fraction(5) default null)
returns integer
```

```
CountIf (
    ts      TimeSeries,
    col     lvarchar,
    op      lvarchar,
    value   lvarchar,
    begin_stamp datetime year to fraction(5) default null,
    end_stamp datetime year to fraction(5) default null)
returns integer
```

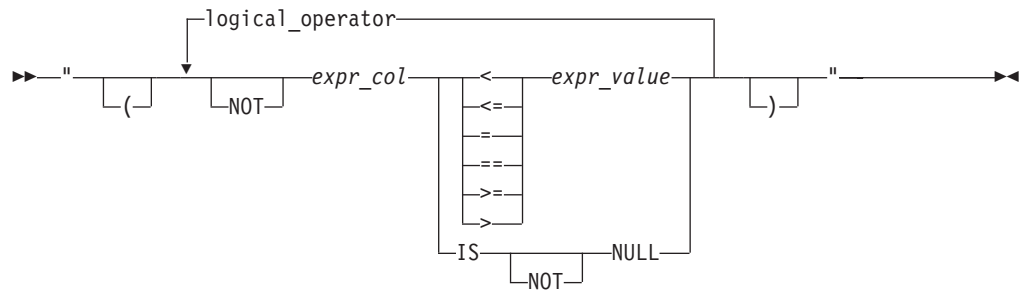


```

CountIf (
  ts      TimeSeries,
  col      lvarchar,
  op      lvarchar,
  value    decimal,
  begin_stamp datetime year to fraction(5) default null,
  end_stamp datetime year to fraction(5) default null)
returns integer

```

Syntax of expr



ts The time series to count.

expr An expression to filter elements by comparing element values to a number or string. You can combine multiple expressions with the AND or the OR operator and use parentheses to nest multiple expressions. Use the following arguments within an expression:

expr_col

The name of the column within a **TimeSeries** data type. If the column is of type BSON, the *expr_col* must be the name of a field in one or more BSON documents in the BSON column. If the BSON field name is the same as another column in the **TimeSeries** data type, you must precede the field name with the column name and a dot: *bson_column_name.bson_field_name*.

expr_value

The value that is used in the comparison. Can be either a number, a string, or NULL.

logical_operator

The AND or the OR operator.

begin_stamp **(optional)**

The begin point of the range. Can be NULL. By default, *begin_stamp* is the beginning of the time series.

end_stamp **(optional)**

The end point of the range. Can be NULL. By default, *end_stamp* is the end of the time series.

col

The name of the column within a **TimeSeries** data type. Can be prefixed with the words IS NULL OR. Must be surrounded by quotation marks. If the column is of type BSON, the *col* must be the name of a field in one or more BSON documents in the BSON column. If the BSON field name is the same as another column in the **TimeSeries** data type, you must precede the field name with the column name and a dot: *bson_column_name.bson_field_name*.

- op* An operator. Can be <, <=, =, !=, >=, or >. Must be surrounded by quotation marks.
- value* The value that is used in the comparison. Can be either a number, a string, or NULL. Sting values must be surrounded by quotation marks.

Usage

Use the **CountIf** function to determine how many elements fit criteria that are based on the values of the columns within the **TimeSeries** subtype. For example, you can apply criteria on multiple columns or determine whether a column has any null values. You can select a time range or query the entire time series.

Returns

An integer that represents the number of elements that fit the criteria.

Examples

The examples are based on the following time series:

```
INSERT INTO CalendarTable(c_name, c_calendar)
VALUES ('sm_15min',
        'startdate(2011-07-11 00:00:00.000000),
        pattstart(2011-07-11 00:00:00.000000),
        pattern('{1 on,14 off}', minute)');
1 row(s) inserted.

EXECUTE PROCEDURE TSContainerCreate('sm0', 'tsspace0', 'sm_row', 0, 0);
Routine executed.

CREATE ROW TYPE sm_row
(
    t      datetime year to fraction(5),
    energy smallint,
    ind    smallint
);
Row type created.

CREATE TABLE sm (
    meter_id varchar(255) primary key,
    readings TimeSeries(sm_row)
) IN tsspace;
Table created.

INSERT INTO sm VALUES ('met0', 'origin(2011-07-11 00:00:00.000000),
                           calendar(sm_15min),container(sm0),threshold(0),
                           regular,[(1,0),(2,1),(3,0),(4,2),(5,3),(6,9),
                           (7,3),(8,0),(9,0),(-123,0),(NULL,0),(NULL,0),
                           (400,3)]');
1 row(s) inserted.
```

Example: Count elements when a column is null

The following statement counts the number of elements where the **energy** column has a null value:

```
SELECT CountIf(readings,'energy IS NULL')
FROM sm;

(expression)
```

2

1 row(s) retrieved.

Two elements contain null values for the **energy** column.

Example: Count elements that match a value in one of two columns

The following statement counts the number of elements where either the value of the **energy** column is equal to 1 or the value of the **ind** column is equal to 0:

```
SELECT CountIf(readings,'energy = 1 or ind = 0')  
FROM sm;
```

(expression)

5

1 row(s) retrieved.

Five elements meet the criteria.

Example: Count elements in a specific time range

The following statement counts the number of elements where the value of the **energy** column is greater than or equal to 5, from 2011-07-11 01:00:00.00000 until the end of the time series:

```
SELECT CountIf(readings,'energy >= 5','2011-07-11 01:00:00.00000'::datetime  
year to fraction(5))  
FROM sm;
```

(expression)

6

1 row(s) retrieved.

Six elements meet the criteria.

Example: Count elements greater than a value

The following statement counts the number of elements where the value of the **energy** column is greater than -128:

```
SELECT CountIf(readings,'energy > -128')  
FROM sm;
```

(expression)

11

1 row(s) retrieved.

The following statement is equivalent to the previous statement, except that the format uses separate arguments for the column name, the operator, and the comparison value instead of a single expression argument:

```
SELECT CountIf(readings,'energy', '>', -128)  
FROM sm;
```

(expression)

11

1 row(s) retrieved.

Example: Count elements in a BSON column

This example is based on the following row type and time series definition. The **TimeSeries** row type contains an **INTEGER** column that is named **v1** and the **BSON** column contains a field that is also named **v1**.

```
CREATE ROW TYPE rb(timestamp datetime year to fraction(5), data bson, v1 int);
```

```
INSERT INTO tj VALUES(1,'origin(2011-01-01 00:00:00.000000), calendar(ts_15min),  
container(kontainer),threshold(0), regular,[{"v1":99},20]');
```

If a column and a BSON field have the same name, the column takes precedence. The following statement counts the number of elements that have a value less than 50 for the **v1** **INTEGER** column:

```
SELECT CountIf(tsddata,'v1 < 50')  
FROM tj;
```

(expression)

1

1 row(s) retrieved.

The following statement counts the number of elements that have a value less than 100 for the **v1** field in the **BSON data** column:

```
SELECT CountIf(tsddata,'data.v1 < 100')  
FROM tj;
```

(expression)

1

1 row(s) retrieved.

Related reference:

“Time series routines that run in parallel” on page 7-7

DelClip function

The **DelClip** function deletes all elements in the specified time range, including the delimiting timepoints, for the specified time series instance. The **DelClip** function differs from the **DelTrim** function in its handling of deletions from the end of a regular time series. **DelTrim** shortens the time series and reclaims space, whereas **DelClip** replaces elements with null values.

Syntax

```
DelClip(ts           TimeSeries,  
       begin_stamp datetime year to fraction(5),  
       end_stamp   datetime year to fraction(5)  
       flags       integer default 0  
)  
returns TimeSeries;
```

ts The time series to act on.

begin_stamp
 The begin point of the range.

end_stamp

The end point of the range.

flags

Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default value is 0.

Description

You can use **DelClip** to delete hidden elements from a time series instance.

If the begin or end point of the range falls before the origin of the time series or after the last element in the time series, an error is raised.

When **DelClip** operates on a regular time series instance, it replaces elements with null elements; it never changes the number of elements in a regular time series.

Returns

A time series with all elements in the range between the specified timepoints deleted.

Example

The following example removes all elements on the specified day for the specified time series instance:

```
update activity_stocks
set activity_data = DelClip(activity_data,
    '2011-01-05 00:00:00.00000'
    ::datetime year to fraction(5),
    '2011-01-06 00:00:00.00000'
    ::datetime year to fraction(5))
where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Clip function” on page 7-31

“DelElem function”

“DelTrim function” on page 7-45

“HideElem function” on page 7-67

“InsSet function” on page 7-70

“PutSet function” on page 7-80

“UpdSet function” on page 7-160

DelElem function

The **DelElem** function deletes the element at the specified timepoint in the specified time series instance.

Syntax

```
DelElem(ts      TimeSeries,
        tstamp datetime year to fraction(5),
        flags integer default 0)
returns TimeSeries;
```

ts The time series to act on.

tstamp The time stamp of the element to be deleted.

flags Valid values for the *flags* parameter are described in “The flags argument values” on page 7-9. The default is 0.

Description

If there is no element at the specified timepoint, no elements are deleted and no error is raised.

The API equivalent of **DelElem** is **ts_del_elem()**.

Hidden time stamps cannot be deleted.

Returns

A time series with one element deleted.

Example

The following example deletes an element from a time series instance:

```
update activity_stocks
set activity_data = DelElem(activity_data,
    '2011-01-05 12:58:09.23456'
    ::datetime year to fraction(5))
where stock_id = 600;
```

Related concepts:

“Delete time series data” on page 3-37

Related reference:

“Time series routines that run in parallel” on page 7-7

“DelClip function” on page 7-42

“DelTrim function” on page 7-45

“GetElem function” on page 7-52

“HideElem function” on page 7-67

“InsElem function” on page 7-69

“PutElem function” on page 7-77

“The ts_del_elem() function” on page 9-22

“UpdElem function” on page 7-159

“The ts_elem() function” on page 9-22

DelRange function

The **DelRange** function deletes all elements in the specified time range in the specified time series instance, including the delimiting timepoints. The **DelRange** function is similar to the **DelTrim** function except that the **DelRange** function deletes elements and reclaims space from any part of a regular time series.

Syntax

```
DelRange(ts           TimeSeries,
        begin_stamp datetime year to fraction(5),
        end_stamp   datetime year to fraction(5),
        flags       integer default 0)
returns TimeSeries;
```

ts The time series to act on.

begin_stamp

The begin point of the range.

end_stamp

The end point of the range.

flags

Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

Use the **DelRange** function to delete elements in a time series instance from a specified time range and free any resulting empty pages. For example, you can remove data from the beginning of a time series instance to archive the data.

If you use the **DelRange** function to delete hidden elements, or if the begin point of the range falls before the origin of the time series, an error is raised.

Returns

A time series with all elements in the range between the specified timepoints deleted.

Example

The following example removes all elements in a one-day range on the specified day for the specified time series instance:

```
UPDATE ts_data
SET meter_data = DelRange(meter_data,
    '2010-11-11 00:00:00.00000'
    ::datetime year to fraction(5),
    '2010-11-11 00:00:00.00000'
    ::datetime year to fraction(5))
WHERE loc_esl_id = 4727354321000111;
```

Related concepts:

“Delete time series data” on page 3-37

Related reference:

“Time series routines that run in parallel” on page 7-7

DelTrim function

The **DelTrim** function deletes all elements in the specified time range in a time series instance, including the delimiting timepoints. The **DelTrim** function is similar to the **DelClip** function except that the **DelTrim** function deletes elements and reclaims space from the end of a regular time series instance, whereas the **DelClip** function replaces elements with null values. The **DelTrim** function is also similar to the **DelRange** function except that the **DelRange** function deletes elements and reclaims space from any part of a regular time series instance.

Syntax

```
DelTrim(ts           TimeSeries,
       begin_stamp datetime year to fraction(5),
       end_stamp   datetime year to fraction(5),
       flags       integer default 0)
returns TimeSeries;
```

ts The time series to act on.

begin_stamp

The begin point of the range.

end_stamp

The end point of the range.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

If you use the **DelTrim** function to delete elements from the end of a time series instance, **DelTrim** trims off all null elements from the end of the time series and thus reduces the number of elements in the time series.

If you use the **DelTrim** function to delete hidden elements, or if the begin point of the range falls before the origin of the time series instance, an error is raised.

Returns

A time series with all elements in the range between the specified timepoints deleted.

Example

The following example removes all elements in a one-day range on the specified day for the specified time series instance:

```
update activity_stocks
set activity_data = DelTrim(activity_data,
    '2011-01-05 00:00:00.00000'
    ::datetime year to fraction(5),
    '2011-01-06 00:00:00.00000'
    ::datetime year to fraction(5))
where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“DelClip function” on page 7-42

“DelElem function” on page 7-43

“Clip function” on page 7-31

“HideElem function” on page 7-67

“InsSet function” on page 7-70

“PutSet function” on page 7-80

“UpdSet function” on page 7-160

Divide function

The **Divide** function divides one time series by another.

It is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Minus**, **Mod**, **Plus**, **Pow**, and **Times**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions” on page 7-27

ElemIsHidden function

The **ElemIsHidden** function determines if an element is hidden.

Syntax

```
ElemIsHidden(ts      TimeSeries,  
             offset integer)  
returns Boolean;  
  
ElemIsHidden(ts      TimeSeries,  
             tstamp datetime year to fraction(5))  
returns Boolean;
```

ts The time series to act on.

offset The offset of the element to examine.

tstamp The time stamp of the element to examine.

Description

Use either offset or time stamp to locate the element you want to examine.

Returns

Returns TRUE if the element is hidden and FALSE if it is not.

Related reference:

“Time series routines that run in parallel” on page 7-7

“ElemIsNull function”

“FindHidden function” on page 7-48

ElemIsNull function

The **ElemIsNull** function determines if an element contains no data.

Syntax

```
ElemIsNull(ts      TimeSeries,  
           offset integer)  
returns Boolean;  
  
ElemIsNull(ts      TimeSeries,  
           tstamp datetime year to fraction(5))  
returns Boolean;
```

ts The time series to act on.

offset The offset of the element to examine.

tstamp The time stamp of the element to examine.

Description

Use either offset or time stamp to locate the element you want to examine.

Returns

Returns TRUE if the element has never been written to or was written to and the data has since been deleted; returns FALSE if the element contains data or is hidden.

Related reference:

“Time series routines that run in parallel” on page 7-7

“ElemIsHidden function” on page 7-47

“FindHidden function”

Exp function

The **Exp** function exponentiates the time series.

It is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

FindHidden function

The **FindHidden** function scans a time series and returns all elements that are hidden.

Syntax

```
FindHidden(ts TimeSeries,  
          start datetime year to fraction(5) default NULL,  
          end   datetime year to fraction(5) default NULL)  
returns multiset;
```

ts The time series to act on.

start (**optional**)

The date from which to start the scan.

end (**optional**)

The date at which to end the scan.

Description

You can scan the whole time series or specify a start date and an end date for the scan.

Returns

A multiset containing all the hidden elements in the date range you specify.

Related reference:

“ElemIsHidden function” on page 7-47

“ElemIsNull function” on page 7-47

GetCalendar function

The **GetCalendar** function returns the calendar associated with the given time series.

Syntax

```
GetCalendar(ts TimeSeries)  
returns Calendar;
```

ts The time series from which to obtain a calendar.

Returns

The calendar used by the time series.

Example

The following example returns the calendar used by the time series for IBM:

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

```
(expression) startdate(2011-01-01 00:00:00),pattstart(2011-
01-02 00:00:00),pattern({1 off,5 on,1 off},day)
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetClosestElem function”

“GetInterval function” on page 7-55

“GetOrigin function” on page 7-64

“TSCreate function” on page 7-116

“GetCalendarName function”

GetCalendarName function

The **GetCalendarName** function returns the name of the calendar used by the given time series.

Syntax

```
GetCalendarName(ts TimeSeries)
returns lvarchar;
```

ts The time series from which to obtain a calendar name.

Returns

The name of the calendar used by the time series.

Example

The following example returns the name of the calendar used by the time series for IBM:

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

```
(expression) daycal
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetCalendar function” on page 7-48

GetClosestElem function

The **GetClosestElem** function returns the first element that is non-null and closest to the given time stamp. Optionally, you can specify which column within the time series element must be non-null to satisfy the search.

Syntax

```
GetClosestElem(ts          TimeSeries,  
               tstamp      datetime year to fraction(5),  
               cmp         lvarchar,  
               column_list lvarchar default NULL,  
               flags       integer default 0)
```

returns ROW

ts The time series to act on.

tstamp The time stamp to start searching from.

cmp A comparison operator used with *tstamp* to determine where to start the search. Valid values for *cmp* are <, <=, =, ==, >=, and >.

column_list

To search for an element with one or more columns non-null, specify a list of column names separated by a vertical bar (|). An error is raised if any of the column names does not exist in the time series type

To search for a null element, set *column_list* to NULL.

flags Determines whether hidden elements should be returned. Valid for the *flags* parameter values are defined in *tseries.h*. They are:

- TS_CLOSEST_NO_FLAGS (no special flags)
- TS_CLOSEST_RETNULLS_FLAGS (return hidden elements)

Description

The search algorithm **ts_closest_elem** is as follows:

- If *cmp* is any of : <=, =, ==, or >=, the search starts at *tstamp*.
- If *cmp* is <, the search starts at the first element before *tstamp*.
- If *cmp* is >, the search starts at the first element after *tstamp*.

The *tstamp* and *cmp* parameters are used to determine where to start the search. The search continues in the direction indicated by *cmp* until an element is found that qualifies. If no element qualifies, the return value is NULL.

Important: For irregular time series, values in an irregular element persist until the next element. This means that any of the “equals” operations on an irregular time series look for <= first. If *cmp* is >= and the <= operation fails, the operation then looks forward for the next element; otherwise, NULL is returned.

Returns

An element meeting the described criteria that is non-null and closest to the given time stamp.

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetCalendar function” on page 7-48

“GetInterval function” on page 7-55

“GetOrigin function” on page 7-64

GetCompression function

The **GetCompression** function returns the compression string if the time series data is compressed.

Syntax

`GetCompression(ts TimeSeries)`
returns string;

ts The name of the time series.

Description

Use the **GetCompression** function to determine the type of compression that is used in a time series that contains compressed numeric data.

Returns

Returns a string that represents the compression type if the time series contains compressed data; returns NULL if the time series does not contain compressed data.

Example

The following statement indicates that the time series that is named **compress_test** uses the compression type Quantization:

```
SELECT GetCompression(compress_test) FROM tstable;
```

```
(expression)    n(),q(1,1,100),ls(0.10), lb(0.10),qls(2,0.15,100,100000),  
                 qlb(2,0.25,100,100000)
```

1 row(s) retrieved.

Related concepts:

“Manage packed data” on page 3-37

Related reference:

“The `ts_get_compressed()` function” on page 9-29

GetContainerName function

The **GetContainerName** function returns the name of the container for the given time series.

Syntax

`GetContainerName(ts TimeSeries)`
returns lvarchar;

ts The time series from which to obtain the container name.

Description

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

Returns

The name of the container for the given time series.

An empty string is returned if the time series is not located in a container.

Example

The following example gets the name of the container holding the stock with ID 600:

```
select GetContainerName(activity_data)
  from activity_stocks
 where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“The `ts_get_containername()` function” on page 9-29

GetElem function

The **GetElem** function extracts the element for the given time stamp.

Syntax

```
GetElem(ts      TimeSeries,
        tstamp datetime year to fraction(5),
        flags  integer default 0)
returns row;
```

ts The source time series.

tstamp The time stamp of the entry.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

If the time stamp is for a time that is not part of the calendar, or if it falls before the origin of the given time series, NULL is returned. In some cases, **GetLastValid**, **GetNextValid**, or **GetPreviousValid** might be more appropriate.

For a regular time series, the data extracted is associated with the time period containing the time stamp. For example, if the time series is set to hourly, 8:00 a.m. to 5:00 p.m., the time stamp 3:15 p.m. would return 3:00 p.m. and the data associated with that time.

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

Returns

A row type containing the time stamp and the data from the time series at that time stamp. The type of the row is the same as the time series subtype.

Example

The following query retrieves the stock data of two stocks for a particular day:

```
select GetElem(stock_data,'2011-01-04 00:00:00.00000')
  from daily_stocks
 where stock_name = 'IBM' or stock_name = 'HWP';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Clip function” on page 7-31

“ClipCount function” on page 7-35

“DelElem function” on page 7-43

“GetLastElem function” on page 7-56

“GetLastValid function” on page 7-58

“GetNextValid function” on page 7-61

“GetNthElem function” on page 7-62
“GetPreviousValid function” on page 7-65
“InsElem function” on page 7-69
“PutElem function” on page 7-77
“Transpose function” on page 7-86
“The ts_elem() function” on page 9-22
“GetIndex function” on page 7-55
“GetStamp function” on page 7-66
“UpdElem function” on page 7-159
“The ts_first_elem() function” on page 9-25

GetFirstElem function

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

Syntax

```
GetFirstElem(ts      TimeSeries,  
            flags integer default 0)  
returns row;
```

ts The source time series.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

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

Returns

A row type containing the first element of the time series, or NULL if there are no elements. The type of the row is the same as the time series subtype.

Example

The following example gets the first element in the time series for the stock with ID 600:

```
select GetFirstElem(activity_data)  
  from activity_stocks  
 where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7
“GetLastElem function” on page 7-56
“The ts_first_elem() function” on page 9-25

GetFirstElementStamp function

The **GetFirstElementStamp** function returns the timestamp of the first element in the time series.

Syntax

```
GetFirstElementStamp(ts TimeSeries,  
                     flag integer DEFAULT 0)  
returns lvarchar;
```

ts The source time series.

flag 0 = Default. Return the timestamp of the first element, regardless of whether the element is null.

 1 = Return the timestamp of the first element that has data, which is stored on the first page of the time series in a container.

Returns

The timestamp of the first element in the time series, as a DATETIME YEAR TO FRACTION(5) value.

Related reference:

“Time series routines that run in parallel” on page 7-7

GetHertz function

The **GetHertz** function returns the frequency for packed hertz data.

Syntax

```
GetHertz(ts TimeSeries)  
returns integer;
```

ts The name of the time series.

Description

Use the **GetHertz** function to determine how many records per second the time series can store.

Returns

Returns an integer 1-255 if the time series contains packed hertz data; returns 0 if the time series does not contain packed hertz data.

Example

The following statement indicates that the time series that is named **hertz_test** can store 60 records per second:

```
EXECUTE FUNCTION GetHertz(hertz_test);
```

(expression)

60

Related concepts:

“Manage packed data” on page 3-37

Related reference:

“The ts_get_hertz() function” on page 9-30

GetIndex function

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

Syntax

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

ts The source time series.

tstamp The time stamp of the entry.

Description

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

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

Returns

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

NULL is returned if the time stamp is not a valid day in the calendar, or if it falls before the origin of the time series.

Example

The following example returns the offset for the supplied time stamp:

```
select stock_name, GetIndex(stock_data,  
  '2011-01-05 00:00:00.00000')  
  from daily_stocks;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“ClipGetCount function” on page 7-37

“CalIndex function” on page 6-2

“CalRange function” on page 6-3

“GetElem function” on page 7-52

“GetNelems function” on page 7-60

“GetNthElem function” on page 7-62

“GetStamp function” on page 7-66

“The ts_index() function” on page 9-35

GetInterval function

The **GetInterval** function returns the interval used by a time series.

Syntax

```
GetInterval(ts TimeSeries)  
returns lvarchar;
```

ts The source time series.

Description

The calendars used by time series values can record intervals of one second, minute, hour, day, week, month, or year. The underlying interval of the calendar describes how often a time series records data.

Returns

An LVARCHAR string that describes the time series interval.

Example

The following query finds all stocks that are not traded on a daily basis:

```
select stock_name
from daily_stocks
where GetInterval(stock_data) <> 'day';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetCalendar function” on page 7-48

“GetClosestElem function” on page 7-49

“CalendarPattern data type” on page 2-1

“GetOrigin function” on page 7-64

“TSCreate function” on page 7-116

GetLastElem function

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

Syntax

```
GetLastElem(ts TimeSeries,
            flags integer default 0)
returns row;
```

ts The source time series.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

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

Returns

A row-type value containing the time series data and time stamp of the last entry in the time series. If the time series is empty, NULL is returned. The type of the row is the same as the time series subtype.

Example

The following query returns the final entry in a time series:

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

The following query retrieves the final entries on a daily stocks table:

```
select GetLastElem(stock_data) from daily_stocks;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetElem function” on page 7-52

“GetFirstElem function” on page 7-53

“GetLastValid function” on page 7-58

“GetNthElem function” on page 7-62

“PutElem function” on page 7-77

“The ts_last_elem() function” on page 9-37

“GetPreviousValid function” on page 7-65

GetLastElementStamp function

The **GetLastElementStamp** function returns the timestamp of the last element in the time series.

Syntax

```
GetLastElementStamp(ts    TimeSeries,  
                   flag integer DEFAULT 0)  
returns lvarchar;
```

ts The source time series.

flag 0 = Default. Return the timestamp of the last element, regardless of whether the element is null.

 1 = Return the timestamp of the last element that has data, which is stored on the last page of the time series in a container.

Returns

The timestamp of the last element in the time series, as a DATETIME YEAR TO FRACTION(5) value.

Related reference:

“Time series routines that run in parallel” on page 7-7

GetLastNonNull function

The **GetLastNonNull** function returns the last non-null element on or before the date you specify.

Syntax

```
GetLastNonNull(ts          TimeSeries,  
              tstamp       datetime year to fraction(5),  
              column_name lvarchar default null,  
              flags        integer default 0  
)  
returns row;
```

ts The source time series.

tstamp The time stamp for the element you specify.

column_name (optional)

If you specify a column using the *column_name* argument, the **GetLastNonNull** function returns the last non-null element on or before the specified date that has a non-null value in the specified column.

If you do not specify the *column_name* argument, the **GetLastNonNull** function returns the last non-null element on or before the date. It is possible that all the columns except the time stamp could be NULL.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

There are no null elements in an irregular time series. Therefore, when you use the **GetLastNonNull** function on an irregular time series, always specify a column name. If you use the **GetLastNonNull** function on an irregular time series without specifying a column name, its effect is equivalent to that of the **GetLastValid** function.

Returns

A non-null element of the time series.

Related reference:

“Time series routines that run in parallel” on page 7-7

GetLastValid function

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

Syntax

```
GetLastValid(ts      TimeSeries,  
            tstamp datetime year to fraction(5),  
            flags integer default 0)  
returns row;
```

ts The source time series.

tstamp The time stamp for the element.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

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

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

Returns

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

If the time stamp is earlier than the origin of the time series, NULL is returned.

Example

The following query returns the last valid entry in a time series at or before a given time stamp:

```
select GetLastValid(stock_data, '2011-01-08 00:00:00.000000')
from daily_stocks
where stock_name = 'IBM';
```

Related reference:

“Time series routines that run in parallel” on page 7-7
 “Clip function” on page 7-31
 “ClipCount function” on page 7-35
 “GetElem function” on page 7-52
 “GetLastElem function” on page 7-56
 “GetNextValid function” on page 7-61
 “GetNthElem function” on page 7-62
 “GetPreviousValid function” on page 7-65
 “PutElem function” on page 7-77
 “The ts_last_valid() function” on page 9-38
 “The ts_next_valid() function” on page 9-42

GetMetaData function

The **GetMetaData** function returns the user-defined metadata from the given time series.

Syntax

```
create function GetMetaData(ts TimeSeries)
returns TimeSeriesMeta;
```

ts The time series to retrieve metadata from.

Returns

This function returns the user-defined metadata contained in the given time series. If the time series does not contain user-defined metadata, then NULL is returned. This return value must be cast to the source data type to be useful.

Related tasks:

“Creating a time series with metadata” on page 3-23

Related reference:

“GetMetaTypeName function”
 “TSCreate function” on page 7-116
 “TSCreateIrr function” on page 7-118
 “UpdMetaData function” on page 7-159
 “The ts_create_with_metadata() function” on page 9-18
 “The ts_get_metadata() function” on page 9-31
 “The ts_update_metadata() function” on page 9-54

GetMetaTypeName function

The **GetMetaTypeName** function returns the type name of the user-defined metadata type stored in the given time series.

Syntax

```
create function GetMetaTypeName(ts TimeSeries)
returns lvarchar;
```

ts The time series to retrieve the metadata from.

Returns

The type name of the user-defined metadata type stored in the given time series. Returns NULL if the given time series does not have user-defined metadata.

Related reference:

“GetMetaData function” on page 7-59

“TSCreate function” on page 7-116

“TSCreateIrr function” on page 7-118

“UpdMetaData function” on page 7-159

“The ts_create_with_metadata() function” on page 9-18

“The ts_get_metadata() function” on page 9-31

“The ts_update_metadata() function” on page 9-54

GetNelems function

The **GetNelems** function returns the number of elements stored in a time series.

Syntax

```
GetNelems(ts TimeSeries)  
returns integer;
```

ts The source time series.

Description

For regular time series, **GetNelems** also counts null elements before the last non-null element, so **GetNelems** might not return the same results as **ClipGetCount**, which does not count null elements.

Returns

The number of elements in the time series.

Example

The following query returns all stocks containing fewer than 355 elements:

```
select stock_name from daily_stocks  
where GetNelems(stock_data) < 355;
```

The following query returns the last five elements of each time series:

```
select Clip(stock_data, GetNelems(stock_data) - 4,  
           GetNelems(stock_data))  
from daily_stocks where stock_name = 'IBM';
```

This example only works if the time series has more than four elements.

Related reference:

“Time series routines that run in parallel” on page 7-7

“ClipGetCount function” on page 7-37

“GetIndex function” on page 7-55

“GetNthElem function” on page 7-62

“GetStamp function” on page 7-66

GetNextNonNull function

The **GetNextNonNull** function returns the next non-null element on or after the date you specify.

Syntax

```
GetNextNonNull(ts          TimeSeries,  
               tstamp      datetime year to fraction(5),  
               column_name lvarchar default null  
               flags       integer default 0  
)  
returns row;
```

ts The source time series.

tstamp The time stamp for the element.

column_name (optional)

If you specify a column using the *column_name* argument, the **GetNextNonNull** function returns the next non-null element on or after the specified date that has a non-null value in the specified column.

If you do not specify the *column_name* argument, the **GetLastNonNull** function returns the next non-null element on or after the date specified by *tstamp*. It is possible that all the columns except the time stamp could be NULL.

flags Valid values for the *flags* parameter are described in “The flags argument values” on page 7-9. The default is 0.

Description

There are no null elements in an irregular time series. Therefore, when you use the **GetNextNonNull** function on an irregular time series, always specify a column name. If you use the **GetNextNonNull** function on an irregular time series without specifying a column name, the function's effect is equivalent to that of the **GetNextValid** function.

Returns

A non-null element of the time series.

Related reference:

“Time series routines that run in parallel” on page 7-7

GetNextValid function

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

Syntax

```
GetNextValid(ts          TimeSeries,  
             tstamp      datetime year to fraction(5),  
             flags       integer default 0)  
returns row;
```

ts The source time series.

tstamp The time stamp of the entry.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

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

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

Returns

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

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

Example

The following example gets the first element that follows time stamp 2011-01-03 in a regular time series:

```
select GetNextValid(stock_data,'2011-01-03 00:00:00.00000')
  from daily_stocks
 where stock_name = 'IBM';
```

The following example gets the first element that follows time stamp 2011-01-03 in an irregular time series:

```
select GetNextValid(activity_data,
  '2011-01-03 00:00:00.00000')
  from activity_stocks
 where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetElem function” on page 7-52

“GetLastValid function” on page 7-58

“GetNthElem function”

“GetPreviousValid function” on page 7-65

“The ts_next_valid() function” on page 9-42

GetNthElem function

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

Syntax

```
GetNthElem(ts      TimeSeries,
          N        integer,
          flags    integer default 0)
returns row;
```

ts The source time series.

N The offset or position of an entry in the time series. This value cannot be less than 0.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

For irregular time series, the **GetNthElem** function returns the *N*th element that is found. For regular time series, the *N*th element is also the *N*th interval from the beginning of the time series.

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

Returns

A row value for the requested offset, including all the time series data at that timepoint and the time stamp of the entry in the time series' calendar. The type of the row is the same as the time series subtype.

If the offset is greater than the offset of the last element in the time series, NULL is returned.

Example

The following query returns the last element in a time series:

```
select GetNthElem(stock_data,GetNelems(stock_data)-1)
  from daily_stocks
  where stock_name = 'IBM';
```

The following query returns the element in a time series at a certain time stamp (this could also be done with **GetElem**):

```
select GetNthElem(stock_data,GetIndex(stock_data,
  '2011-01-04 00:00:00.00000'))
  from daily_stocks
  where stock_name = 'IBM';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Clip function” on page 7-31

“ClipCount function” on page 7-35

“ClipGetCount function” on page 7-37

“GetElem function” on page 7-52

“GetIndex function” on page 7-55

“GetLastElem function” on page 7-56

“GetLastValid function” on page 7-58

“GetNelems function” on page 7-60

“GetNextValid function” on page 7-61

“GetPreviousValid function” on page 7-65

“PutElem function” on page 7-77

“Transpose function” on page 7-86

“The ts_nth_elem() function” on page 9-43

“GetStamp function” on page 7-66

GetOrigin function

The **GetOrigin** function returns the origin of the time series.

Syntax

```
GetOrigin(ts TimeSeries)  
returns datetime year to fraction(5);
```

ts The source time series.

Description

Every time series value has a corresponding calendar and an origin within the calendar. The calendar describes how often data values appear in the time series. The origin of the time series is the first timepoint within the calendar for which the time series can contain data; however, the time series does not necessarily have data for that timepoint. The origin is set when the time series is created, and it can be changed with **SetOrigin**.

Returns

The time series origin.

Example

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

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

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetCalendar function” on page 7-48

“GetInterval function” on page 7-55

“GetClosestElem function” on page 7-49

“TSCreate function” on page 7-116

“SetOrigin function” on page 7-85

“The ts_get_origin() function” on page 9-31

GetPacked function

The **GetPacked** function returns whether the specified time series contains packed data.

Syntax

```
GetPacked(ts        TimeSeries)  
returns integer;
```

ts The name of the time series.

Description

Use the **GetPacked** function to determine whether a time series stores either hertz data or compressed numeric data in packed elements.

Returns

Returns 1 if the time series contains packed data; returns 0 if the time series does not contain packed data.

Example

The following statement indicates that the time series that is named **historic_measure** contains packed data:

```
EXECUTE FUNCTION GetPacked(historic_measure);
```

(expression)

1

Related concepts:

“Manage packed data” on page 3-37

Related reference:

“The `ts_get_packed()` function” on page 9-32

GetPreviousValid function

The **GetPreviousValid** function returns the last element before the given time stamp.

Syntax

```
GetPreviousValid(ts      TimeSeries,  
                tstamp datetime year to fraction(5),  
                flags integer default 0)  
returns row;
```

ts The source time series.

tstamp The time stamp of interest.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

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

Returns

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

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

Example

The following query gets the first element that precedes time stamp 2011-01-05 in a regular time series:

```
select GetPreviousValid(stock_data,  
    '2011-01-05 00:00:00.00000')  
from daily_stocks  
where stock_name = 'IBM';
```

The following query gets the first element that precedes time stamp 2011-01-05 in an irregular time series:

```
select GetPreviousValid(activity_data,  
    '2011-01-05 00:00:00.00000')  
    from activity_stocks  
    where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“GetElem function” on page 7-52

“GetLastElem function” on page 7-56

“GetLastValid function” on page 7-58

“GetNextValid function” on page 7-61

“GetNthElem function” on page 7-62

“The ts_previous_valid() function” on page 9-45

GetStamp function

The **GetStamp** function returns the time stamp associated with the supplied offset in a time series. Offsets can be positive or negative integers.

Syntax

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

ts The source time series.

offset The offset.

Description

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

Returns

The time stamp that begins the interval at the specified offset.

Example

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

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

Related reference:

“Time series routines that run in parallel” on page 7-7

“ClipGetCount function” on page 7-37

“GetIndex function” on page 7-55

“GetNelems function” on page 7-60

“CalIndex function” on page 6-2

“CalRange function” on page 6-3

“GetElem function” on page 7-52

“GetNthElem function” on page 7-62

“The ts_time() function” on page 9-51

GetThreshold function

The **GetThreshold** function returns the threshold associated with the specified time series.

Syntax

```
GetThreshold(ts      TimeSeries)
returns integer;
```

ts The source time series.

Description

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

Returns

The threshold of the supplied time series.

Example

The following query returns the threshold of the specified time series:

```
select GetThreshold(stock_data) from daily_stocks;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“The ts_get_threshold() function” on page 9-33

HideElem function

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

Syntax

```
HideElem(ts      TimeSeries,
        tstamp datetime year to fraction(5),
        flags integer default 0)
returns TimeSeries;
```

```
HideElem(ts      TimeSeries,
        multiset_tstamps multiset(datetime year to fraction(5) not null),
        flags integer default 0)
returns TimeSeries;
```

ts The source time series.

tstamp The time stamp to be made invisible.

multiset_tstamps
 The multiset of time stamps to be made invisible.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

After an element is hidden, reading that element returns NULL and writing it results in an error message. It is, however, possible to use **ts_begin_scan()** to read hidden elements.

The API equivalent to this function is **ts_hide_elem()**.

If the time stamp is not a valid timepoint in the time series, an error is raised.

Returns

The modified time series.

Example

The following example hides the element at 2011-01-03 in the time series for IBM:

```
select HideElem(stock_data, '2011-01-03 00:00:00.00000')
  from daily_stocks
 where stock_name = 'IBM';
```

Related concepts:

“Calendar data type” on page 2-4

Related reference:

“Time series routines that run in parallel” on page 7-7

“CalendarPattern data type” on page 2-1

“DelClip function” on page 7-42

“DelElem function” on page 7-43

“DelTrim function” on page 7-45

“RevealElem function” on page 7-83

“The ts_begin_scan() function” on page 9-7

“The ts_hide_elem() function” on page 9-34

“The ts_reveal_elem() function” on page 9-50

“HideRange function”

HideRange function

The **HideRange** function marks as invisible a range of elements between a starting time stamp and an ending time stamp.

Syntax

```
HideRange(ts      TimeSeries,
         start   datetime year to fraction(5),
         end     datetime year to fraction(5),
         flags   integer default 0
       )
returns TimeSeries;
```

ts The time series to act on.

start The starting time stamp.

end The ending time stamp.

flags Valid values for the flags parameter are described in “The flags argument values” on page 7-9. The default is 0.

Description

After an element is hidden, reading that element returns NULL and writing it results in an error message. It is, however, possible to use **ts_begin_scan()** to read hidden elements, as described in “The ts_begin_scan() function” on page 9-7.

If the time stamp is not a valid timepoint in the time series, an error is raised.

Returns

The modified time series.

Related reference:

“Time series routines that run in parallel” on page 7-7

“HideElem function” on page 7-67

“RevealRange function” on page 7-83

InsElem function

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

Syntax

```
InsElem(ts           TimeSeries,  
        row_value   row,  
        flags       integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

row_value

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

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

The element must be a row type of the correct type for the time series, beginning with a valid time stamp. If there is already an element with that time stamp in the time series, the insertion is void, and an error is raised. After the insertion is done, the time series must be assigned to a row in a table, or the insertion is lost.

InsElem should be used only within UPDATE and INSERT statements. If it is used within a SELECT statement or a qualification, unpredictable results can occur.

You cannot insert an element at a time stamp that is hidden.

The API equivalent of **InsElem** is **ts_ins_elem()**.

Returns

The new time series with the element inserted.

Example

The following example inserts an element into a time series:

```
update activity_stocks  
set activity_data =  
    InsElem(activity_data,  
            row('2011-10-06 08:06:56.000000', 6.50, 2000,  
                1, 007, 3, 1)::stock_trade)  
where stock_id = 600;
```

Related reference:

“DelElem function” on page 7-43

“GetElem function” on page 7-52
“InsSet function”
“PutElem function” on page 7-77
“The ts_ins_elem() function” on page 9-36
“UpdElem function” on page 7-159

InsSet function

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

Syntax

```
InsSet(ts           TimeSeries,  
      multiset_rows multiset,  
      flags         integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

multiset_rows

The multiset of new row type values to store in the time series.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

The supplied row type values must have a time stamp as their first attribute. This time stamp is used to determine where in the time series the insertions are to be performed. For example, to insert into a time series that stores a single double-precision value, the row type values passed to **InsSet** would have to contain a time stamp and a double-precision value.

If there is already an element at the specified timepoint, the entire insertion is void, and an error is raised.

You cannot insert an element at a time stamp that has been hidden.

Returns

The time series with the multiset inserted.

Example

The following example inserts a set of **stock_trade** items into a time series:

```
update activity_stocks  
set activity_data = (select InsSet(activity_data, set_data)  
                    from activity_load_tab where stock_id = 600)  
where stock_id = 600;
```

Related reference:

“DelClip function” on page 7-42
“DelTrim function” on page 7-45
“InsElem function” on page 7-69
“PutSet function” on page 7-80
“UpdSet function” on page 7-160

InstanceId function

The **InstanceId** function determines if the time series is stored in a container and, if it is, returns the instance ID of that time series.

Syntax

```
InstanceId(ts TimeSeries)  
returns bigint;
```

ts The source time series.

Description

The instance ID is used as an index in the container. It can also be used to lookup information from 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;
```

Related concepts:

“TSInstanceTable table” on page 2-12

Intersect function

The **Intersect** function performs an intersection of the specified time series over the entire length of each time series or over a clipped portion of each time series.

Syntax

```
Intersect(ts TimeSeries,  
          ts TimeSeries,...)  
returns TimeSeries;
```

```
Intersect(set_ts set(TimeSeries))  
returns TimeSeries;
```

```
Intersect(begin_stamp datetime year to fraction(5),  
          end_stamp  datetime year to fraction(5),  
          ts          TimeSeries,  
          ts          TimeSeries,...)  
returns TimeSeries;
```

```
Intersect(begin_stamp datetime year to fraction(5),  
          end_stamp  datetime year to fraction(5),  
          set_ts     set(TimeSeries))  
returns TimeSeries;
```

ts The time series that form the intersection. **Intersect** can take from two to eight time series arguments.

set_ts Indicates the intersection of a set of time series.

begin_stamp
 The begin point of the clip.

end_stamp

The end point of the clip.

Description

The second and fourth forms of the function **Intersect** intersect a set of time series. The resulting time series has one DATETIME YEAR TO FRACTION(5) column followed by each column in each time series in order, not including the other time stamps. When using the second or fourth form, it is important to ensure that the order of the time series in the set is deterministic so that elements remain in the correct order.

Since the resulting time series is a different type from the input time series, the result of the intersection must be cast.

Intersect can be thought of as a join on the time stamp columns.

If any of the input time series is irregular, the resulting time series is irregular.

For the purposes of **Intersect**, the value at a specified timepoint is that of the most recent valid element. For regular time series, this is the value corresponding to the current interval, which can be NULL; it is not necessarily the most recent non-null value. For irregular time series, this condition never occurs, because irregular time series do not have null intervals.

For example, consider the intersection of two irregular time series, one containing bid prices for a certain stock, and one containing asking prices. The intersection of the two time series contains bid and ask values for each timepoint at which a price was either bid or asked. Now consider a timepoint at which a bid was made but no price was asked. The intersection at that timepoint contains the bid price offered at that timepoint, along with the most recent asking price.

If an intersection involves one or more regular time series, the resulting time series starts at the latest of the start points of the input time series and ends at the earliest of the end points of the regular input time series. If all the input time series are irregular, the resulting irregular time series starts at the latest of the start points of the input time series and ends at the latest of the end points. If a union involves one or more time series, the resulting time series starts at the first of the start points of the input time series and ends at the latest of the end points of the input time series. Other than this difference in start and end points, and of the resulting calendar, there is no difference between union and intersection involving time series.

In an intersection, the resulting time series has a calendar that is the combination of the calendars of the input time series with the AND operator. The resulting calendar is stored in the **CalendarTable** table. The name of the resulting calendar is a string containing the names of the calendars of the input time series joined by an ampersand (&). For example, if two time series are intersected, and **mycal** and **yourcal** are the names of their corresponding calendars, the resulting calendar is named **mycal&yourcal**.

To be certain of the order of the columns in the resultant time series when using **Intersect** with the *set_ts* argument, use the ORDER BY clause.

Apply also combines multiple time series into a single time series. Therefore, using **Intersect** within **Apply** is often unnecessary.

Returns

The time series that results from the intersection.

Example

The following example returns the intersection of two time series:

```
select Intersect(d1.stock_data,
                d2.stock_data)::TimeSeries(stock_bar_union)
   from daily_stocks d1, daily_stocks d2
  where d1.stock_name='IBM' and d2.stock_name='HWP';
```

The following query intersects two time series and returns data only for time stamps between 2011-01-03 and 2011-01-05:

```
select Intersect('2011-01-03 00:00:00.00000'
                ::datetime year to fraction(5),
                '2011-01-05 00:00:00.00000'
                ::datetime year to fraction(5),
                d1.stock_data,
                d2.stock_data
                )::TimeSeries(stock_bar_union)
   from daily_stocks d1, daily_stocks d2
  where d1.stock_name = 'IBM' and d2.stock_name = 'HWP';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Apply function” on page 7-18

“Union function” on page 7-157

IsRegular function

The **IsRegular** function tells whether a specified time series is regular.

Syntax

```
IsRegular(ts TimeSeries)
returns boolean;
```

ts The source time series.

Returns

TRUE if the time series is regular; otherwise FALSE.

Example

The following query gets stock IDs for all stocks in irregular time series:

```
select stock_id
   from activity_stocks
  where not IsRegular(activity_data);
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“The `ts_get_flags()` function” on page 9-30

Lag function

The **Lag** function creates a new regular time series in which the data values lag the source time series by a fixed offset.

Syntax

```
Lag(ts      TimeSeries,  
    nelems integer)  
returns TimeSeries;
```

ts The source time series.

nelems The number of elements to lag the series by. Positive values lag the result behind the argument, and negative values lead the result ahead.

Description

Lag shifts only offsets, not the source time series. Therefore, a lag of -2 eliminates the first two elements. For example, if there is a daily time series, Monday to Friday, and a one-day lag (an argument of -1) is imposed, then there is no first Monday, the first Tuesday is Monday, and the next Monday is Friday. It would be more typical of a daily time series to lag a full week.

For example, this function allows the user to create a hypothetical time series, with closing stock prices for each day moved two days ahead on the calendar.

Lag is valid only for regular time series.

Returns

A new time series with the same calendar and origin as the source time series but that has its elements assigned to different offsets.

Example

The following query creates a new time series that lags the original time series by three days:

```
select Lag(stock_data,3)  
from daily_stocks  
where stock_name = 'IBM';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

Logn function

The **Logn** function returns the natural logarithm of a time series.

The **Logn** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Minus function

The **Minus** function subtracts one time series from another.

The **Minus** function is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Mod**, **Plus**, **Pow**, and **Times**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions” on page 7-27

Mod function

The **Mod** function computes the modulus or remainder of a division of one time series by another.

The **Mod** function is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Minus**, **Plus**, **Pow**, and **Times**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions” on page 7-27

Negate function

The **Negate** function negates a time series.

The **Negate** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

NullCleanup function

The **NullCleanup** function frees any pages in a time series instance that contain only null elements in a range or for the whole time series instance.

Syntax

```
NullCleanup(ts           TimeSeries,  
           begin_stamp datetime year to fraction(5),  
           end_stamp   datetime year to fraction(5),  
           flags       integer default 0)  
returns TimeSeries;
```

```
NullCleanup(ts           TimeSeries,  
           flags       integer default 0)  
returns TimeSeries;
```

```
NullCleanup(ts           TimeSeries,  
           begin_stamp datetime year to fraction(5),  
           flags       integer default 0)  
returns TimeSeries;
```

```
NullCleanup(ts           TimeSeries,  
           NULL,  
           end_stamp   datetime year to fraction(5),  
           flags       integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

begin_stamp
 The begin point of the range.

end_stamp

The end point of the range.

flags

Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

Use the **NullCleanup** function to free empty pages from a time series instance in one of the following time ranges:

- A specified begin point and a specified end point
- The whole time series instance
- A specified begin point and the end of the time series instance
- The beginning of the time series instance and a specified end point

If the begin point of the range falls before the origin of the time series instance, an error is raised.

Returns

A time series with all the empty pages in the range freed.

Examples

Example 1: Free empty pages between specified begin and end points

The following example frees the empty pages in a one-day range on the specified day in the time series instance for the location ID of 4727354321000111:

```
UPDATE ts_data
SET meter_data = NullCleanup(meter_data,
    '2010-11-11 00:00:00.00000'
    ::datetime year to fraction(5),
    '2010-11-11 00:00:00.00000'
    ::datetime year to fraction(5))
WHERE loc_esl_id = 4727354321000111;
```

Example 2: Free all empty pages in the time series instance

The following example frees all empty pages in the time series instance for the location ID of 4727354321000111:

```
UPDATE ts_data
SET meter_data = NullCleanup(meter_data)
WHERE loc_esl_id = 4727354321000111;
```

Example 3: Free empty pages from the beginning of the time series instance to a specified date

The following example frees empty pages from the beginning of the time series instance to the specified end point in the time series instance for the location ID of 4727354321000111:

```
UPDATE ts_data
SET meter_data = NullCleanup(meter_data, NULL,
    '2010-11-11 00:00:00.00000'
    ::datetime year to fraction(5))
WHERE loc_esl_id = 4727354321000111;
```

Related concepts:

“Delete time series data” on page 3-37

Related reference:

“Time series routines that run in parallel” on page 7-7

Plus function

The **Plus** function adds two time series together.

The **Plus** function is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Minus**, **Mod**, **Pow**, and **Times**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions” on page 7-27

Positive function

The **Positive** function returns the argument. It is bound to the unary “+” operator.

The **Positive** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Round**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Pow function

The **Pow** function raises the first argument to the power of the second.

The **Pow** function is one of the binary arithmetic functions that work on time series. The others are **Atan**, **Divide**, **Minus**, **Mod**, **Plus**, and **Times**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions” on page 7-27

PutElem function

The **PutElem** function adds an element to a time series at the timepoint indicated in the supplied row type.

Syntax

```
PutElem(ts           TimeSeries,  
        row_value row,  
        flags       integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

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

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

For regular time series, if there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

For irregular time series, if there is no data at the given timepoint, the new data is inserted. If there is data at the given timepoint, the following algorithm is used to determine where to place the data:

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

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The API equivalent of **PutElem** is **ts_put_elem()**.

Returns

A modified time series that includes the new values.

Example

The following example appends an element to a time series:

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

The following example updates a time series:

```
update activity_stocks
set activity_data = PutElem(activity_data,
    row('2011-08-25 09:06:00.000000',
        6.25, 1000, 1, 007, 2, 1)::stock_trade)
where stock_id = 600;
```

Related concepts:

“The TSVTMode parameter” on page 4-16

Related reference:

“DelElem function” on page 7-43

“GetElem function” on page 7-52

“GetLastElem function” on page 7-56

“GetLastValid function” on page 7-58

“GetNthElem function” on page 7-62

“InsElem function” on page 7-69

“PutElemNoDups function” on page 7-79

“PutSet function” on page 7-80

“TSCreate function” on page 7-116

“The ts_put_elem() function” on page 9-46

“PutNthElem function” on page 7-80

“UpdElem function” on page 7-159

PutElemNoDups function

The **PutElemNoDups** function inserts a single element into a time series. If there is already an element at the specified timepoint, it is replaced by the new element.

Syntax

```
PutElemNoDups(ts           TimeSeries,  
              row_value row,  
              flags       integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

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

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

If there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The API equivalent of **PutElemNoDups** is **ts_put_elem_no_dups()**.

Returns

A modified time series that includes the new values.

Example

The following example updates a time series:

```
update activity_stocks  
set activity_data = PutElemNoDups(activity_data,  
  row('2011-08-25 09:06:00.000000', 6.25,  
      1000, 1, 007, 2, 1)::stock_trade)  
where stock_id = 600;
```

Related concepts:

“The TSVTMode parameter” on page 4-16

Related reference:

“PutElem function” on page 7-77

“The ts_put_elem_no_dups() function” on page 9-47

PutNthElem function

The **PutNthElem** function puts the supplied row at the supplied offset in a regular time series.

Syntax

```
PutNthElem(ts          TimeSeries,  
          row_value   row,  
          N           integer,  
          flags       integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

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

N The offset. Must be greater than or equal to 0.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

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

If there is data at the given offset, it is updated with the new data; otherwise, the new data is inserted.

The row type passed in must match the subtype of the time series.

Hidden elements cannot be updated.

Returns

A modified time series that includes the new values.

Example

The following example puts data in the first element of the IBM time series:

```
update daily_stocks  
set stock_data =  
    PutNthElem(stock_data,  
          row(NULL::dateTime year to fraction(5), 355, 309,  
          341, 999)::stock_bar, 0)  
where stock_name = 'IBM';
```

Related reference:

“PutElem function” on page 7-77

PutSet function

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

Syntax

```
PutSet(ts           TimeSeries,  
      multiset_ts set,  
      flags        integer default 0)  
returns TimeSeries;
```

ts The time series to act on.

multiset_ts
 The multiset of new row type values to store in the time series.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

For each element in the multiset of rows, if the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

For regular time series, if there is data at a specified 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 specified timepoint, the new data is inserted. If there is data at the specified timepoint, the following algorithm is used to determine where to place the data:

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

The row type that is 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 multiset:

```
update activity_stocks  
set activity_data = (select PutSet(activity_data, set_data)  
                      from activity_load_tab where stock_id = 600)  
where stock_id = 600;
```

Related reference:

“DelClip function” on page 7-42

“DelTrim function” on page 7-45

“InsSet function” on page 7-70

“PutElem function” on page 7-77

“TSCreate function” on page 7-116

“UpdSet function” on page 7-160

“PutTimeSeries function” on page 7-82

PutTimeSeries function

The **PutTimeSeries** function puts every element of the first time series into the second time series.

Syntax

```
PutTimeSeries(ts1   TimeSeries,  
              ts2   TimeSeries,  
              flags integer default 0)  
returns TimeSeries;
```

ts1 The time series to be inserted.

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

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

If both time series contain data at the same timepoint, the rule of **PutElem** is followed (see “PutElem function” on page 7-77), unless the TS_PUTELEM_NO_DUPS value of the *flags* parameter is set.

Both time series must have the same calendar. Also, the origin of the time series that is specified by the first argument must be later than or equal to the origin of the time series that is specified by the second argument.

This function can be used to convert a regular time series to an irregular one.

Important: Converting an irregular time series to regular often requires aggregation information, which can be provided by the **AggregateBy** function.

Elements are added to the second time series by calling **ts_put_elem()** (if the TS_PUTELEM_NO_DUPS value of the *flags* parameter is not set).

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

Returns

A version of the second time series into which the first time series was inserted.

Example

The following example converts a regular time series to an irregular one. The **daily_stocks** table holds regular time series data, and the **activity_stocks** table holds irregular time series data. Additionally, the elements in the **daily_stocks** time series are converted from **stock_bar** to **stock_trade**:

```
update activity_stocks  
  set activity_data = PutTimeSeries(activity_data, 'calendar(daycal),  
irregular':::TimeSeries(stock_trade))  
  where stock_id = 600;
```

Related reference:

“AggregateBy function” on page 7-11

“PutSet function” on page 7-80

“The ts_put_ts() function” on page 9-49

“SetOrigin function” on page 7-85

RevealElem function

The **RevealElem** function makes an element at a specified time stamp available for a scan. It reverses the effect of **HideElem**.

Syntax

```
RevealElem(ts      TimeSeries,  
          tstamp datetime year to fraction(5))  
returns TimeSeries;  
  
RevealElem(ts      TimeSeries,  
          set_stamps multiset(datetime year to fraction(5)))  
returns TimeSeries;
```

ts The time series to act on.

tstamp The time stamp to be made visible to a scan.

set_stamps
 The multiset of time stamps to be made visible to a scan.

Returns

The modified time series.

Example

The following example hides the element at 2011-01-03 in the IBM time series and then reveals it:

```
select HideElem(stock_data, '2011-01-03 00:00:00.00000')  
  from daily_stocks  
 where stock_name = 'IBM';  
  
select RevealElem(stock_data, '2011-01-03 00:00:00.00000')  
  from daily_stocks  
 where stock_name = 'IBM';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“HideElem function” on page 7-67

“The ts_reveal_elem() function” on page 9-50

RevealRange function

The **RevealRange** function makes hidden elements in a specified date range visible. It reverses the effect of **HideRange**.

Syntax

```
RevealRange(ts      TimeSeries,  
           start    datetime year to fraction(5),  
           end      datetime year to fraction(5),  
           )  
returns TimeSeries;
```

ts The time series to act on.

start The time stamp at the start of the range.

end The time stamp at the end of the range.

Returns

The modified time series.

Related reference:

“Time series routines that run in parallel” on page 7-7

“HideRange function” on page 7-68

Round function

The **Round** function rounds a time series to the nearest whole number.

The **Round** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Sin**, **Sqrt**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

SetContainerName function

The **SetContainerName** function sets the container name for a time series, even if the time series already has a container name.

Syntax

```
SetContainerName(ts           TimeSeries,  
                container_name varchar(128,1))  
returns TimeSeries;
```

ts The time series to act on.

container_name
 The name of the container.

Description

If a time series is stored in a container, you can use the **SetContainerName** function to copy the time series from one container to another. The time series is copied to the container that you specify with the *container_name* parameter. The original time series is unaffected.

Returns

A time series with a new container set.

Example

The following example creates the container **tsirr** and sets a time series to it:

```
execute procedure TSContainerCreate('tsirr', 'rootdbs',  
    'stock_bar_union', 0, 0);
```

```
select SetContainerName(Union(s1.stock_data,  
    s2.stock_data)::TimeSeries(stock_bar_union),  
    'tsirr')  
from daily_stocks s1, daily_stocks s2  
where s1.stock_name = 'IBM' and s2.stock_name = 'AA02';
```

Related reference:

SetOrigin function

The **SetOrigin** function moves the origin of a time series back in time.

Syntax

```
SetOrigin(ts      TimeSeries,  
         origin datetime year to fraction(5))  
returns TimeSeries;
```

ts The time series to act on.

origin The new origin of the time series.

Description

If the supplied origin is not a valid timepoint in the given time series calendar, the first valid timepoint following the supplied origin becomes the new origin. The new origin must be earlier than the current origin. To move the origin forward, use the **Clip** function.

Returns

The time series with the new origin.

Example

The following example sets the origin of the **stock_data** time series:

```
update daily_stocks  
  set stock_data = SetOrigin(stock_data,  
    '2011-01-02 00:00:00.00000');
```

Related reference:

“Apply function” on page 7-18

“Clip function” on page 7-31

“GetOrigin function” on page 7-64

“PutTimeSeries function” on page 7-82

Sin function

The **Sin** function returns the sine of its argument.

The **Sin** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Sqrt function

The **Sqrt** function returns the square root of its argument.

The **Sqrt** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, and **Tan**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Tan function

The **Tan** function returns the tangent of its argument.

The **Tan** function is one of the unary arithmetic functions that work on time series. The others are **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, and **Sin**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Unary arithmetic functions” on page 7-156

Times function

The **Times** function multiplies one time series by another.

The **Times** function is one of the binary arithmetic functions that work on time series. The others are **Atan2**, **Divide**, **Minus**, **Mod**, **Plus**, and **Pow**.

Related reference:

“Time series routines that run in parallel” on page 7-7

“Binary arithmetic functions” on page 7-27

TimeSeriesRelease function

The **TimeSeriesRelease** function returns an LVARCHAR string containing the TimeSeries extension version number and build date.

Syntax

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

Returns

The version number and build date.

Example

The following example shows how to get the version number using DB-Access:

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

Transpose function

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

Syntax

```
Transpose (ts          TimeSeries,  
          begin_stamp datetime year to fraction(5) default NULL,  
          end_stamp   datetime year to fraction(5) default NULL,  
          flags       integer default 0)  
returns row;
```

```
Transpose (query      lvarchar,  
          dummy        row,  
          begin_stamp datetime year to fraction(5) default NULL,  
          end_stamp   datetime year to fraction(5) default NULL,  
          col_name    lvarchar default NULL,  
          flags       integer default 0)  
returns row with (iterator);
```

ts The time series to transpose.

begin_stamp
 The begin point of the range. Can be NULL.

end_stamp
 The end point of the range. Can be NULL.

flags Determines how a scan works on the returned set.

query A string that contains a SELECT statement that can return multiple columns but only one time series column. The non-time-series columns are concatenated with each time series element in the returned rows.

dummy A row type that must be passed in as NULL and cast to the expected return type of each row that is returned by the query string version of the **Transpose** function.

col_name
 If *col_name* is not NULL, only the column that is specified with this parameter is used from the time series element, plus the non-time-series columns.

Description

The **Transpose** function is an iterator function. You can run the **Transpose** function with the EXECUTE FUNCTION statement or in a table expression.

Normally the transpose function skips NULL elements when returning the rows found in a time series. If the TS_SCAN_NULLS_OK (0x40) bit of the *flags* parameter is set, the **Transpose** function returns NULL elements.

If the beginning point is NULL, the scan starts at the first element of the time series, unless the TS_SCAN_EXACT_START value of the *flags* parameter is set.

If the end point is NULL, the scan ends at the last element of the time series, unless the TS_SCAN_EXACT_END value of the *flags* parameter is set.

The flags argument values

The *flags* argument determines how a scan works on the returned set. The value of *flags* is the sum of the wanted flag values from the following table.

Table 7-3. The flags argument values

Flag	Value	Meaning
TS_SCAN_HIDDEN	512	Return hidden elements marked by HideElem (see “HideElem function” on page 7-67).
TS_SCAN_EXACT_START	256	Return the element at the beginning timepoint, adding null elements if necessary.
TS_SCAN_EXACT_END	128	Return elements up to the end point (return NULL if necessary).
TS_SCAN_NULLS_OK	64	Return null time series elements (by default, time series elements that are NULL are not returned).
TS_SCAN_NO_NULLS	32	Instead of returning a null row, return a row with the time stamp set and the other columns set to NULL.
TS_SCAN_SKIP_END	16	Skip the element at the end timepoint of the scan range.
TS_SCAN_SKIP_BEGIN	8	Skip the element at the beginning timepoint of the scan range.
TS_SCAN_SKIP_HIDDEN	4	Used by ts_begin_scan() to tell ts_next() not to return hidden elements.

Returns

Multiple rows that contain a time stamp and the other columns of the time series elements.

Example 1: Convert time series data to a table

The following statement converts the data from **stock_data** for IBM to tabular form:

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

Example 2: Transpose clipped data

The following statement converts data for a clipped range into tabular form:

```
execute function Transpose((select stock_data from daily_stocks
    where stock_name = 'IBM'),
    datetime(2011-01-05) year to day,
    NULL::datetime year to fraction(5));
```

The statement returns the following data in the form of a row data type:

```
ROW('2011-01-06 00:00:00.000000',99.00000
000000,54.000000000000,66.000000000000,888.00000000000)
```

Example 3: Convert time series and other data into tabular format

The following example returns the time series columns together with the non-time-series columns in tabular form:

```
execute function Transpose ('select * from daily_stocks', NULL::row(stock_id
int, stock_name lvarchar,
    t datetime year to fraction(5), high real, low real, final real, volume real));
```

Example 4: Display specific data as multiple fields within a single column

The following statement selects the time and energy readings from a time series:

```
SELECT mr.t,mr.energy
FROM TABLE(transpose
              ((SELECT readings FROM smartmeters
                 WHERE meter_id = 13243))::smartmeter_row)
AS tab(mr);
```

The statements returns a table named **tab** that contains one column, named **mr**. The **mr** column is an unnamed row type that has the same fields as the **TimeSeries** subtype named **smartmeter_row**. The output has a field for time and a field for energy:

t	energy
2011-01-01 00:00:00.00000	29
2011-01-01 00:15:00.00000	18
2011-01-01 00:30:00.00000	13
2011-01-01 00:45:00.00000	26
2011-01-01 01:00:00.00000	21
2011-01-01 01:15:00.00000	15
2011-01-01 01:30:00.00000	20
2011-01-01 01:45:00.00000	24
2011-01-01 02:00:00.00000	30
2011-01-01 02:15:00.00000	30
2011-01-01 02:30:00.00000	29
2011-01-01 02:45:00.00000	32
2011-01-01 03:00:00.00000	29

Example 5: Display specific data in a table with multiple columns

The following statement uses the statement from the previous example inside a table expression in the FROM clause:

```
SELECT * FROM (
    SELECT mr.t,mr.energy,mr.temperature
    FROM TABLE(transpose
                  ((SELECT readings FROM smartmeters
                     WHERE meter_id = 13243))::smartmeter_row)
    AS tab(mr)
) AS sm(t,energy,temp)
WHERE temp < -10;
```

The statement returns the following data in the form of a table named **sm** that contains three columns:

t	energy	temp
2011-01-01 00:00:00.00000	29	-13.0000000000
2011-01-01 00:30:00.00000	13	-18.0000000000
2011-01-01 01:00:00.00000	21	-13.0000000000
2011-01-01 01:15:00.00000	15	-11.0000000000
2011-01-01 03:15:00.00000	22	-19.0000000000
2011-01-01 03:45:00.00000	28	-14.0000000000
2011-01-01 04:00:00.00000	19	-14.0000000000
2011-01-01 04:30:00.00000	27	-14.0000000000
2011-01-01 04:45:00.00000	27	-15.0000000000
2011-01-01 05:00:00.00000	28	-11.0000000000

Related reference:

“GetElem function” on page 7-52

“GetNthElem function” on page 7-62

“TSColNameToList function” on page 7-91
“TSColNumToList function” on page 7-92
“TSRowNameToList function” on page 7-145
“TSRowNumToList function” on page 7-146
“TSRowToList function” on page 7-147
“TSSetToList function” on page 7-153

TSAddPrevious function

The **TSAddPrevious** function sums all the values it is called with and returns the current sum every time it is called. The current argument is not included in the sum.

Syntax

```
TSAddPrevious(current_value smallfloat)  
returns smallfloat;  
  
TSAddPrevious(current_value double precision)  
returns double precision;  
  
current_value  
    The current value.
```

Description

Use the **TSAddPrevious** function within an **AggregateBy** or **Apply** function. The **TSAddPrevious** function can take parameters that are columns of a time series. Use the same parameter format as the **AggregateBy** or **Apply** function accepts.

Returns

The sum of all previous values returned by this function.

Example

The following example uses the **TSAddPrevious** function to calculate the summation of the average dollars into or out of a market or equity:

```
select Apply('TSAddPrevious($vol * (($final - $low) - ($high - $final) / (.0001  
+ $high - $low)) * (($high + $low + $final) / 3))',  
            '2011-01-03 00:00:00.00000'::datetime year to fraction(5),  
            '2011-01-08 00:00:00.00000'::datetime year to fraction(5),  
            stock_data)::TimeSeries(one_real)  
from daily_stocks  
where stock_name = 'IBM';
```

Related reference:

“Apply function” on page 7-18
“TSCmp function”
“TSDecay function” on page 7-122
“TSPrevious function” on page 7-141
“TSRunningAvg function” on page 7-147
“TSRunningSum function” on page 7-151

TSCmp function

The **TSCmp** function compares two values.

Syntax

```
TSCmp(value1 smallfloat,  
      value2 smallfloat)  
returns int;
```

```
TSCmp(value1 double precision,  
      value2 double precision)  
returns int;
```

value1 The first value to be compared.

value2 The second value to be compared.

Description

Use the **TSCmp** function within the **Apply** function.

The **TSCmp** function takes either two SMALLFLOAT values or two DOUBLE PRECISION values; both values must be the same type. The **TSCmp** function can take parameters that are columns of a time series. Use the same parameter format that the **Apply** function accepts.

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 the **TSCmp** function to calculate the on-balance volume, a continuous summation that adds the daily volume to the running total if the stock or index advances and subtracts the volume if it declines:

```
select Apply  
  ('TSAddPrevious(TSCmp($final, TSPrevious($final)) * $vol)',  
   '2011-01-03 00:00:00.00000'::datetime year to fraction(5),  
   '2011-01-08 00:00:00.00000'::datetime year to fraction(5),  
   stock_data)::TimeSeries(one_real)  
from daily_stocks  
where stock_name = 'IBM';
```

Related reference:

“Apply function” on page 7-18

“TSAddPrevious function” on page 7-90

“TSDecay function” on page 7-122

“TSPrevious function” on page 7-141

“TSRunningAvg function” on page 7-147

“TSRunningSum function” on page 7-151

TSColNameToList function

The **TSColNameToList** function takes a TimeSeries column and returns a list (collection of rows) containing the values of one of the columns in the elements of the time series. Null elements are not added to the list.

Syntax

```
TSColNameToList(ts      TimeSeries,  
                colname lvarchar)  
returns list
```

ts The time series to act on.

colname The column to return.

Description

Because this aggregate function can return rows of any type, the return value must be explicitly cast at runtime.

Returns

A list (collection of rows).

Example

This query returns a list of all values in the column **high**:

```
select * from table((select  
    TSColNameToList(stock_data, 'high')::list(real  
    not null) from daily_stocks));
```

Related reference:

“Transpose function” on page 7-86

“TSColNumToList function”

“TSRowNameToList function” on page 7-145

“TSRowNumToList function” on page 7-146

“TSSetToList function” on page 7-153

“TSRowToList function” on page 7-147

TSColNumToList function

The **TSColNumToList** function takes a **TimeSeries** column and returns a list (collection of rows) containing the values of one of the columns in the elements of the time series. Null elements are not added to the list.

Syntax

```
TSColNumToList(ts      TimeSeries,  
                colnum integer)  
returns list
```

ts The time series to act on.

colnum The column to return.

Description

The column is specified by its number; column numbering starts at 1, with the first column following the time stamp column.

Because this aggregate function can return rows of any type, the return value must be explicitly cast at runtime.

Returns

A list (collection of rows).

Example

This query returns a list of all values in the column **high**:

```
select * from table((select
    TSColNumToList(stock_data, 1)::list(real
    not null) from daily_stocks));
```

Related reference:

“TSColNameToList function” on page 7-91

“Transpose function” on page 7-86

“TSRowNameToList function” on page 7-145

“TSRowNumToList function” on page 7-146

“TSSetToList function” on page 7-153

“TSRowToList function” on page 7-147

TSContainerCreate procedure

The **TSContainerCreate** procedure creates a container to store the time series data for the specified **TimeSeries** subtype. You can create a container in one dbspace, a container that spans multiple partitions, or a rolling window container, which controls the amount of data that is stored.

Only users with update privileges on the **TSContainerTable** table and the **TSContainerWindowTable** can run this procedure.

Rolling window containers are a special type of container that requires additional arguments. See “Syntax for rolling window containers” on page 7-94. Use the rolling window container syntax to create a container that spans multiple partitions.

Syntax

```
TSContainerCreate(container_name varchar(128,1),
                  dbspace_name   varchar(128,1),
                  ts_type         varchar(128,1),
                  container_size integer,
                  container_grow integer);
```

container_name

The name of the new container. The container name must be unique.

dbspace_name

The name of the dbspace that holds the container.

ts_type

The name of the **TimeSeries** subtype that is stored in the container. This argument must be the name of an existing row type that begins with a time stamp.

container_size

The first extent size of the container, in KB.

The value must be equivalent to at least 4 pages. If you specify 0 or a negative number, 16 KB is used. The maximum size of a container depends on the page size:

- For 2-KB pages, the maximum size is 32 GB.

- For 4-KB pages, the maximum size is 64 GB.
- For 8-KB pages, the maximum size is 128 GB.
- For 16-KB pages, the maximum size is 256 GB.

container_grow

The increments by which the container grows, in KB. The value must be equivalent to at least 4 pages. If you specify 0 or a negative number, 16 KB is used.

Usage

By default, containers are created automatically as needed when you insert data into a time series. However, you can create additional containers by using the **TSContainerCreate** procedure.

You can create multiple containers in the same dbspace.

When you create a container, a row is inserted in the **TSContainerTable** table.

Example

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

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

Syntax for rolling window containers

```
TSContainerCreate(container_name    varchar(128,1),
                  dbspace_name     varchar(128,1),
                  ts_type           varchar(128,1),
                  container_size    integer,
                  container_grow    integer,
                  window_origin     datetime year to fraction(5));
```

```
TSContainerCreate(container_name    varchar(128,1),
                  dbspace_name     varchar(128,1),
                  ts_type           varchar(128,1),
                  container_size    integer,
                  container_grow    integer,
                  window_origin     datetime year to fraction(5),
                  window_interval   lvARCHAR default 'month',
                  active_windowsize integer default 0,
                  dormant_windowsize integer default 0,
                  window_spaces     lvARCHAR(4096) default null,
                  window_control    integer default 0,
                  rwi_firsttextsize integer default 16,
                  rwi_nexttextsize  integer default 16,
                  destroy_count     integer default 0);
```

container_name

The name of the new container. The container name must be unique.

dbspace_name

The name of the dbspace that contains the container partition. If you do not specify additional dbspaces with the *window_spaces* argument, this dbspace also contains the partitions for time series elements.

ts_type The name of the **TimeSeries** subtype that is stored by the container. This argument must be the name of an existing row type that begins with a time stamp.

container_size

The first extent size for partitions, in KB.

The value must be equivalent to at least 4 pages. If you specify 0 or a negative number, 16 KB is used. The maximum size of a partition depends on the page size:

- For 2 KB pages, the maximum size is 32 GB.
- For 4 KB pages, the maximum size is 64 GB.
- For 8 KB pages, the maximum size is 128 GB.
- For 16 KB pages, the maximum size is 256 GB.

container_grow

The next extent size for partitions, in KB. The value must be equivalent to at least 4 pages. If you specify 0 or a negative number, 16 KB is used.

window_origin

The first timestamp that is allowed for the rolling window container. The rolling window container rejects time series values if the origin of the time series is before the origin of the container.

window_interval

The range of time for which data is stored in each partition. By default, the window interval is one month, which means that each window partition contains data from one calendar month. Possible values are:

- **day** = The partitions contain data from one day. The container uses the **ts_1day** calendar.
- **week** = The partitions contain data from one week. The container uses the **ts_1week** calendar.
- **month** = Default. The partitions contain data from one calendar month. The container uses the **ts_1month** calendar.
- **year** = The partitions contain data from one calendar year. The container uses the **ts_1year** calendar.

active_windowsize (Optional)

The maximum number of partitions in the active window:

- 0 = Default. No size limit.
- Positive integer = The maximum number of partitions in the active window.

dormant_windowsize (Optional)

The maximum number of partitions in the dormant window:

- 0 = Default. No size limit.
- Positive integer = The maximum number of partitions in the dormant window. Start with a value that is equal to or greater than the size of the active window.

window_spaces (Optional)

The dbspaces in which partitions are stored:

- **NULL** = Default. Partitions are created in the dbspace that is specified by the *dbspace_name* argument.
- A comma-separated list of dbspace names = Partitions are created in the listed dbspaces in round-robin order. The list cannot include temporary dbspaces or sbspaces.

window_control (Optional)

A flag that indicates how many partitions are destroyed and whether

active partitions can be destroyed when the number of partitions that must be detached is greater than the dormant windows size:

- 0 = Default. As many as necessary dormant partitions are destroyed. If the operation requires more new active partitions than the value of the *active_windowsize* parameter, the operation fails.
- 1 = As many as necessary existing dormant partitions and older active partitions are destroyed. Use this setting with caution. Destroyed data cannot be recovered.
- 2 = Dormant partitions are destroyed, but limited to the number specified by the *destroy_count* parameter. If the number of partitions that must be destroyed for an operation exceeds the value of the *destroy_count* parameter, the operation fails.
- 3 (2 + 1) = Existing dormant partitions and older active partitions are destroyed, but limited to the number specified by the *destroy_count* parameter. If the number of partitions that must be destroyed for an operation exceeds the value of the *destroy_count* parameter, the operation fails.

When you destroy a partition, the data that is stored in the partition is deleted.

rwi_firstextsize (Optional)

The first extent size, in KB, for the container partition. The default size is 16 KB.

rwi_nextextsize (Optional)

The next extent size, in KB, for the container partition. The default size is 16 KB.

destroy_count (Optional)

How many partitions can be destroyed in an operation. Valid if the value of the *window_control* parameter is 2 or 3.

0 = Default. No dormant partitions are destroyed.

A positive integer = The maximum number of partitions that can be destroyed in an operation. If the number of partitions that must be destroyed for an operation exceeds the value of the *destroy_count* parameter, the operation fails.

Usage for rolling window containers

The rolling window container stores information about the properties of the windows and information about the partitions. The partitions store the time series data for specific date ranges. The **TSContainerCreate** procedure creates the rolling window container when the procedure completes. Partitions are created as needed when you insert time series elements.

To create a rolling window container that stores data in multiple dbspaces and automatically deletes old data, set the following arguments to non-default values:

- *active_windowsize*: Set to a positive value to limit to the size of the active window.
- *dormant_windowsize*: Set to a positive value to limit to the size of the dormant window.
- *window_spaces*: Set to a list of dbspaces.

If you use the default size of 0 for the active window, you create a container that grows until you manually detach partitions into the dormant window and manually destroy partitions from the dormant window.

The container partitions and the partitions in the active and dormant windows can require significantly different amounts of storage space. Plan the storage for rolling window containers carefully.

When you create a rolling window container, a row is inserted in the **TSContainerTable** and the **TSContainerWindowTable** table. As partitions are added for time series data, rows are added to the **TSContainerUsageActiveWindowVTI** and the **TSContainerUsageDormantWindowVTI** tables.

Example 1: Create a rolling window container

The following example creates a rolling window container:

```
execute procedure TSContainerCreate('readings_container',
                                   'containerdbs', 'rt_raw_intvl', 25600, 12800,
                                   '2011-01-01 00:00:00.000000'::datetime year to fraction(5),
                                   'month', 4, 10, 'dbs0, dbs1, dbs2, dbs3, dbs4', 1, 16, 8);
```

The example configures a rolling window container that has the following properties:

- The container name is **readings_container**.
- The dbspace for the container partition is named **containerdbs**.
- The name of the time series is **rt_raw_intvl**.
- The first extent size of the partitions is 25600 KB.
- The next extent size of the partitions is 12800 KB.
- The active window contains up to 4 partitions.
- The dormant window contains up to 10 partitions.
- The origin of the container is 2011-01-01 00:00:00.00000.
- Partitions each hold a month of data.
- The dbspaces for partitions are named **dbs0**, **dbs1**, **dbs2**, **dbs3**, and **dbs4**.
- Partitions that no longer fit into the dormant window are automatically destroyed.
- The first extent size of the dbspace **containerdbs** is 16 KB.
- The next extent size of the dbspace **containerdbs** is 8 KB.

Example 2: Create a container with multiple dbspaces

The following example creates a container that stores data in multiple dbspaces but does not use a purging policy:

```
execute procedure TSContainerCreate('readings_container',
                                   'containerdbs', 'rt_raw_intvl', 25600, 12800,
                                   '2011-01-01 00:00:00.000000'::datetime year to fraction(5),
                                   'month', 0, 0, 'dbs0, dbs1, dbs2, dbs3, dbs4', 0, 16, 8);
```

The example configures a container that has the following properties:

- The container name is **readings_container**.
- The dbspace for the container partition is named **containerdbs**.
- The name of the time series is **rt_raw_intvl**.

- The first extent size of the partitions is 25600 KB.
- The next extent size of the partitions is 12800 KB.
- The active window size is unlimited.
- The dormant window size is unlimited.
- The origin of the container is 2011-01-01 00:00:00.00000.
- Partitions each hold a month of data.
- The dbspaces for partitions are named **db0**, **db1**, **db2**, **db3**, and **db4**.
- Partitions are not automatically destroyed.
- The first extent size of the dbspace **containerdb** is 16 KB.
- The next extent size of the dbspace **containerdb** is 8 KB.

The container stores each month of data in a partition in one of the five dbspaces for partitions. Because the active window size is unlimited, all partitions are active until they are manually detached and then destroyed.

Related concepts:

"TSInstanceTable table" on page 2-12

"TSContainerTable table" on page 2-9

Related tasks:

"Creating containers" on page 3-15

"Configuring additional container pools" on page 3-20

Related reference:

"Rules for rolling window containers" on page 3-16

"SetContainerName function" on page 7-84

"TSContainerDestroy procedure"

"Planning for data storage" on page 1-20

TSContainerDestroy procedure

The **TSContainerDestroy** procedure deletes the container row from the **TSContainerTable** table and removes the container and its corresponding system catalog rows.

Syntax

```
TSContainerDestroy(container_name varchar(128,1));
```

container_name

The name of the container to destroy.

Description

You can destroy a container only if 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 run this procedure.

Example

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

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

Related concepts:

"TSInstanceTable table" on page 2-12

"TSContainerTable table" on page 2-9

Related tasks:

"Creating containers" on page 3-15

Related reference:

"TSContainerCreate procedure" on page 7-93

TSContainerLock procedure

The **TSContainerLock** procedure controls whether multiple sessions can write to a container at one time.

Syntax

```
TSContainerLock(  
    container_name  varchar(128),  
    flag            integer);
```

container_name

The name of the container. Must be an existing container name.

flag

Controls whether multiple sessions can write to the container:

0 = Multiple sessions can write to the container at the same time. Multiple locks are available for the container.

1 = Only one session at a time can write to the container. One lock is available for the container.

Usage

By default, multiple sessions can write to a container at the same time. You can prevent more than one session from writing to a container by setting the *flags* argument to 1. Data is loaded faster when a single session writes to a container and the *flags* argument is set to 1. If your application enforces that one session writes to a container at a time, set the *flags* argument to 1 to improve performance.

Example

The following statement restricts the number of sessions that can write to the container named **ctn_sm0** to 1:

```
EXECUTE PROCEDURE TSContainerLock('ctn_sm0',1);
```

Related tasks:

"Creating containers" on page 3-15

TSContainerManage function

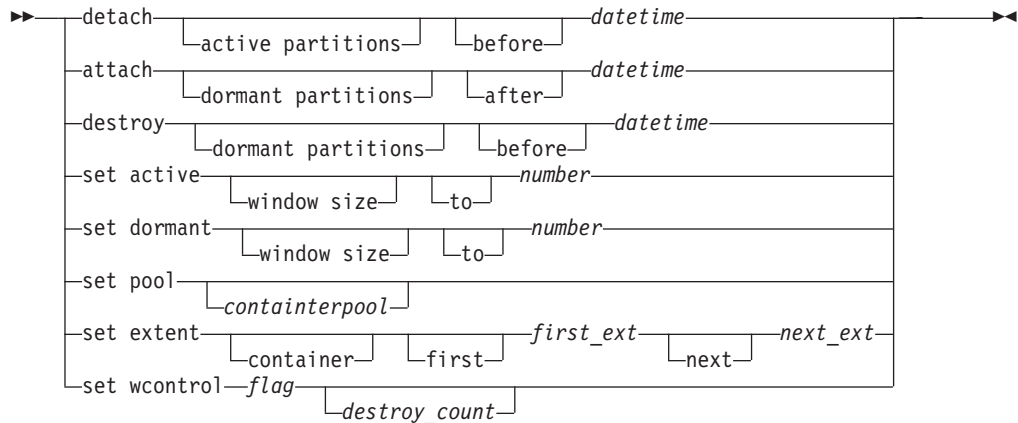
The **TSContainerManage** function changes the properties of containers.

Only users with update privileges on the **TSContainerTable** table and the **TSContainerWindowTable** table can run this function.

Syntax

```
TSContainerManage(container_name  lvarchar,  
                 command          lvarchar);
```

Syntax of command



container_name

The name of the container. The container name must exist.

command

A command that changes the properties of the container:

detach active partitions before *datetime*

For rolling window containers, detaches any active partitions that are before the specified timestamp to the dormant window.

The keywords **active partitions** and **before** are optional and do not change the command.

attach dormant partitions after *datetime*

For rolling window containers, attaches any dormant partitions that are after the specified timestamp into the active window. However, the maximum number of partitions in the active window is not exceeded, even if more partitions fit the timestamp criteria.

The keywords **dormant partitions** and **after** are optional and do not change the command.

destroy dormant partitions before *datetime*

For rolling window containers, destroys any dormant partitions that are before the specified timestamp. The partitions can contain data. Any active partitions that meet the timestamp criteria are not destroyed.

The keywords **dormant partitions** and **before** are optional and do not change the command.

set active window size to *number*

For rolling window containers, sets the number of partitions in the active window:

- 0 = No size limit.
- Positive integer = The maximum number of partitions in the active window. If you decrease the size, the number of oldest active partitions that exceed the active window size are detached to the dormant window. If you increase the size, partitions in the dormant window are not attached into the active window.

The keywords **window size** and **to** are optional and do not change the command.

set dormant window size to *number*

For rolling window containers, sets the number of partitions in the dormant window:

- 0 = Default. No size limit.
- Positive integer = The maximum number of partitions in the dormant window. If you decrease the size, the number of dormant partitions that exceed the dormant window size are destroyed.

The keywords **window size** and **to** are optional and do not change the command.

set pool *containerpool*

Moves the container into the specified container pool.

set pool

Removes the container from its container pool. Same as the **TSContainerSetPool** procedure.

set extent container first *first_ext* **next** *next_ext*

Sets the size, in KB, of the first and next extents for the container partition.

The keywords **first** and **next** are optional and do not change the command.

set extent first *first_ext* **next** *next_ext*

Sets the sizes, in KB, of the first and next extents for the partitions that contain time series elements. For rolling window containers, changes the sizes of the existing partitions in the active and dormant windows and sets the size of new partitions.

The keywords **first** and **next** are optional and do not change the command.

set wcontrol *flagdestroy_count*

Sets a flag that indicates whether active partitions can be destroyed when the number of partitions that must be detached is greater than the dormant windows size and optionally sets how many partitions can be destroyed. The values for *flag* are:

- 0 = Default. As many as necessary dormant partitions are destroyed. If the operation requires more new active partitions than the value of the *active_windowsize* parameter, the operation fails.
- 1 = As many as necessary existing dormant partitions and older active partitions are destroyed. Use this setting with caution. Destroyed data cannot be recovered.
- 2 = Dormant partitions are destroyed, but limited to the number specified by the *destroy_count* value. If the number of partitions that must be destroyed for an operation exceeds the value of the *destroy_count* value, the operation fails.
- 3 (2 + 1) = Existing dormant partitions and older active partitions are destroyed, but limited to the number specified by the *destroy_count* value. If the number of partitions that must be destroyed for an operation exceeds the value of the *destroy_count* value, the operation fails.

The *destroy_count* value is valid if the value of the *flag* parameter is 2 or 3:

- 0 = Default. No dormant partitions are destroyed.
- A positive integer = The maximum number of partitions that can be destroyed in an operation. If the number of partitions that must be destroyed for an operation exceeds the value of the *destroy_count* value, the operation fails.

Usage

For all containers, you can change the container pool and the extent sizes of the partitions. For rolling window containers, you can also change the window sizes and attach, detach, or destroy partitions.

Returns

A message that describes the result of the command.

Examples

The following examples are based on a rolling window container named **readings_container** that has a day interval, an active window size of 5, and a dormant window size of 10. The time series elements use a 15-minute calendar with the range from 2012-01-01 00:00:00.00000 to 2012-01-10 23:45:00.00000.

The partitions are distributed between the active and the dormant windows in the following way:

- Partitions that are in the active window:
 - P6: elements for 2012-01-06
 - P7: elements for 2012-01-07
 - P8: elements for 2012-01-08
 - P9: elements for 2012-01-09
 - P10: elements for 2012-01-10
- Partitions that are in the dormant window:
 - P1: elements for 2012-01-01
 - P2: elements for 2012-01-02
 - P3: elements for 2012-01-03
 - P4: elements for 2012-01-04
 - P5: elements for 2012-01-05

Example 1: Detach partitions

The following example moves partitions from the active window to the dormant window:

```
execute function TSContainerManage(
  "readings_container",
  "detach active partitions before 2012-01-08")
```

The following message describes the result:

```
detach succeeded: 2 partitions moved
```

The partitions are now distributed between the active and the dormant windows in the following way:

- Partitions that are in the active window:

- P8: elements for 2012-01-08
- P9: elements for 2012-01-09
- P10: elements for 2012-01-10
- Partitions that are in the dormant window:
 - P1: elements for 2012-01-01
 - P2: elements for 2012-01-02
 - P3: elements for 2012-01-03
 - P4: elements for 2012-01-04
 - P5: elements for 2012-01-05
 - P6: elements for 2012-01-06
 - P7: elements for 2012-01-07

Partitions P6 and P7 moved into the dormant window.

Example 2: Attach partitions

The following example moves a partition from the dormant window to the active window:

```
execute function TSContainerManage(
  "readings_container",
  "attach dormant partitions after 2012-01-06")
```

The following message describes the result:

attach succeeded: 1 partition moved

The partitions are now distributed between the active and the dormant windows in the following way:

- Partitions that are in the active window:
 - P7: elements for 2012-01-07
 - P8: elements for 2012-01-08
 - P9: elements for 2012-01-09
 - P10: elements for 2012-01-10
- Partitions that are in the dormant window:
 - P1: elements for 2012-01-01
 - P2: elements for 2012-01-02
 - P3: elements for 2012-01-03
 - P4: elements for 2012-01-04
 - P5: elements for 2012-01-05
 - P6: elements for 2012-01-06

Partition P7 moved into the active window.

Example 3: Increase the active window size

The following example increases the size of the active window to 10:

```
execute function TSContainerManage(
  "readings_container",
  "set active window size to 10");
```

The following message describes the result:

Set active window size succeeded: 0 partitions moved

Although the active window size is larger, partitions in the dormant window are not moved back into the active window.

Example 4: Destroy partitions

The following example destroys partitions in the dormant window:

```
execute function TSContainerManage(  
  "readings_container",  
  "destroy dormant partitions before 2012-01-08");
```

The following message describes the result:

destroy succeeded: 6 partitions destroyed

The partitions are now distributed between the active and the dormant windows in the following way:

- Partitions that are in the active window:
 - P7: elements for 2012-01-07
 - P8: elements for 2012-01-08
 - P9: elements for 2012-01-09
 - P10: elements for 2012-01-10
- No partitions in the dormant window.

Although some of the active partitions are before 2012-01-08, they are not destroyed.

Example 5: Change the destroy behavior

The following statement changes the behavior when partitions are destroyed to allow up to seven dormant and active partitions to be destroyed in an operation:

```
execute function TSContainerManage('readings_container','set wcontrol 3 7');
```

Related reference:

“Rules for rolling window containers” on page 3-16

TSContainerNElems function

The **TSContainerNElems** function returns the number of time series data elements stored in the specified container or in all containers.

Syntax

```
TSContainerNElems(container_name varchar(128,1));  
TSContainerNElems(container_name varchar(128,1)  
                  rw_flag integer default 0);
```

container_name

Specifies which container to return information about. Must be an existing container name. You can include wildcard characters from the MATCHES operator: *, ?, [...], \, ^. The function returns information for all containers that have names that match the expression. See MATCHES operator.

The value NULL returns information about all containers for the database.

rw_flag

For rolling window containers, specifies for which partitions to return the sum of number of elements:

0 = The partitions in the active window

1 = The partitions in the dormant window

3 = The container partition. The number of elements equals the number of intervals in the active and dormant windows.

4 = All partitions.

Description

Use the **TSContainerNElems** function to view the number of elements stored in a container. For rolling window containers, the **TSContainerNElems** function returns the sum of the number of elements in the specified set of partitions.

Returns

The number of elements.

Example

The following statement returns the number of elements stored in the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerNElems("mult_container");
```

elements

26

1 row(s) retrieved.

The following statement returns the number of elements stored in all containers:

```
EXECUTE FUNCTION TSContainerNElems(NULL);
```

elements

241907

1 row(s) retrieved.

Related concepts:

"Monitor containers" on page 3-18

Related reference:

"TSContainerUsage function" on page 7-114

"TSContainerTotalPages function" on page 7-112

"TSContainerTotalUsed function" on page 7-113

"TSContainerPctUsed function"

"Time series storage" on page 1-14

TSContainerPctUsed function

The **TSContainerPctUsed** function returns the percentage of space that is used in the specified container or in all containers.

Syntax

```
TSContainerPctUsed(container_name varchar(128,1));  
TSContainerPctUsed(container_name varchar(128,1)  
                  rw_flag integer default 0);
```

container_name

Specifies which container to return information about. Must be an existing container name. You can include wildcard characters from the MATCHES operator: *, ?, [...], \, ^. The function returns information for all containers that have names that match the expression. See MATCHES operator.

The value NULL returns information about all containers for the database.

rw_flag

For rolling window containers, specifies for which partitions to return the percentage of the used space:

0 = The partitions in the active window

1 = The partitions in the dormant window

2 = The container partition.

3 = All partitions.

Description

Use the **TSContainerPctUsed** function to view the percentage of used space in a container or in all containers. For rolling window containers, the **TSContainerPctUsed** function returns the percentage of the space in the specified set of partitions that is used.

Returns

The percentage of used space.

Example

The following statement returns the percentage of used space in the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerPctUsed("mult_container");
```

percent

60.000

1 row(s) retrieved.

The following statement returns the percentage of used space in all containers:

```
EXECUTE FUNCTION TSContainerPctUsed(NULL);
```

percent

93.545

1 row(s) retrieved.

Related concepts:

“Monitor containers” on page 3-18

Related reference:

“TSContainerUsage function” on page 7-114

“TSContainerTotalPages function” on page 7-112

“TSContainerTotalUsed function” on page 7-113

“TSContainerNElems function” on page 7-104

“Time series storage” on page 1-14

TSContainerPoolRoundRobin function

The **TSContainerPoolRoundRobin** function provides a round-robin policy for inserting time series data into containers in the specified container pool.

Syntax

```
TSContainerPoolRoundRobin(  
    table_name lvarchar,  
    column_name lvarchar,  
    subtype lvarchar,  
    irregular integer,  
    pool_name lvarchar)  
returns lvarchar;
```

table_name

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

column_name

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

subtype

The name of the **TimeSeries** subtype.

irregular

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

pool_name

The name of the container pool.

Description

Use the **TSContainerPoolRoundRobin** function to select containers in which to insert time series data from the specified container pool. The container pool must exist before you can insert data into it, and at least one container within the container pool must be configured for the same **TimeSeries** subtype as used by the data being inserted. Set the **TSContainerPoolRoundRobin** function to a container pool name and use it as the value for the **container** argument in the VALUES clause of an INSERT statement. The **TSContainerPoolRoundRobin** function returns container names to the INSERT statements in round-robin order.

Returns

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

Example

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

```
INSERT INTO smartmeters(meter_id,rawreadings)  
VALUES('met00001','origin(2006-01-01 00:00:00.00000),  
calendar(smarmeter),regular,threshold(0),  
container(TSContainerPoolRoundRobin(readings)),
```

```
[(33070,-13.00,100.00,9.98e+34),
 (19347,-4.00,100.00,1.007e+35),
 (17782,-18.00,100.00,9.83e+34)]');
```

When the INSERT statement runs, the **TSContainerPoolRoundRobin** function runs with the following values:

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

The **TSContainerPoolRoundRobin** function sorts the container names alphabetically and returns the first container name to the INSERT statement. The next time an INSERT statement is run, the **TSContainerPoolRoundRobin** function returns the second container name, and so on.

Related tasks:

“Configuring additional container pools” on page 3-20

Related reference:

“User-defined container pool policy” on page 3-21

TSContainerPurge function

The **TSContainerPurge** function deletes time series data through a specified timestamp from one or more containers.

Syntax

```
TSContainerPurge(
    control_file  lvarchar,
    location      lvarchar default 'client',
    flags         integer default 0);
returns lvarchar
```

control_file

The name of the text file that contains information about which elements to delete from which containers. The file must have one or more lines in the following format:

```
container_name|instance_id|end_range|
```

container_name

The name of the container from which to delete elements.

instance_id

The unique identifier of a time series instance. An instance is a row in a table that includes a **TimeSeries** column.

end_range

The ending time of the deletion range. For a regular time series, the index of the last timestamp to delete. For irregular time series, the last timestamp to delete.

location **(Optional)**

The location of the control file. Can be either of the following values:

'client'

Default. The control file is on the client computer.

'server'

The control file is on the same computer as the database server.

flags **(Optional)**

Determines delete behavior. Can be either of the following values:

- 0 Elements that match the delete criteria are deleted only if all elements on a page match the criteria. The resulting empty pages are freed.
- 1 Elements on pages where all the elements match the delete criteria are deleted and the pages are freed. Remaining elements that match the delete criteria are set to NULL.

Usage

Use the **TSContainerPurge** function to remove old data from containers. The **TSContainerPurge** function deletes pages where all elements have a timestamp that is equal to or older than the specified ending time in the specified containers for the specified time series instances. The resulting empty pages are freed.

You can use the **TSContainerPurge** function to remove data from row that use a different **TimeSeries** subtype than the subtype specified in the container definition. However, the **TimeSeries** subtype that is named in the container definition must exist.

You can create a control file by unloading the results of a SELECT statement that defines the delete criteria into a file. Use the following TimeSeries functions in the SELECT statement to populate the control file with the container names, instance IDs, and, for regular time series, the indexes of the end range for reach instance:

- **GetContainerName** function
- **InstanceId** function
- **GetIndex** function (regular time series only)

If you intend to delete a large amount of data at one time, running multiple **TSContainerPurge** functions to delete data from different containers might be faster than running a single **TSContainerPurge** function.

Returns

An LVARCHAR string that describes how many containers were affected, how many pages were freed, and how many elements were deleted. For example:
"containers(4) deleted_pages(2043) deleted_slots(260300)"

Example 1: Delete regular time series data from multiple containers

The following statement creates a control file named `regular_purge.unl` to delete elements from 10 regular time series instances in all containers that store those instances:

```
UNLOAD TO 'regular_purge.unl'
SELECT GetContainerName(readings),InstanceId(readings),
       GetIndex(readings,'2011-10-01 23:45:00.00000'::datetime year to
                  fraction(5))::varchar(25)
FROM sm
WHERE meter_id IN ('met0','met1','met11','met4','met5','met6',
                  'met61','met7','met8','met9');
```

The resulting control file has the following contents:

```
sm0|1|7871|
sm0|8|7295|
sm0|13|6911|
sm1|2|7775|
```

sm1	9	7199
sm1	14	6815
sm2	3	7679
sm2	10	7103
sm3	7	7391
sm3	12	7007

The 10 time series instances that are specified in the WHERE clause are stored in the four different containers, which are listed in the first column. The second column lists the ID for each time series instance. The third column lists the element index number that corresponds to the timestamp 2011-10-01 23:45:00.00000.

The following statement deletes all elements at and before 2011-10-01 23:45:00.00000 for the 10 time series instances:

```
EXECUTE FUNCTION TSContainerPurge('regular_purge.unl',1);
```

Any deleted elements that remain are marked as NULL.

Example 2: Delete elements from a specific container

The following statement creates a control file named regular_purge2.unl to delete elements from all time series instances in the container named **sm0**:

```
UNLOAD TO regular_purge2.unl
  SELECT GetContainerName(readings),InstanceId(readings),
         GetIndex(readings,'2011-10-01 23:00:00.00000'::datetime
                   year to fraction(5))::varchar(25)
  FROM sm
  WHERE GetContainerName(readings) = 'sm0';
```

The resulting control file has entries for a single container:

sm0	1	7871
sm0	8	7295
sm0	13	6911

Example 3: Deleting irregular time series data

The following statement creates a control file named irregular_purge.unl to delete elements from four irregular time series instances:

```
UNLOAD TO irregular_purge.unl
  SELECT GetContainerName(readings),InstanceId(readings),
         '2011-10-01 23:00:00.00000'::varchar(25)
  FROM sm
  WHERE meter_id IN ('met12','met2','met3','met62');
```

The resulting control file includes the ending timestamp instead of the element index number, for example:

sm4	4	2011-10-01 23:45:00.00000
sm4	6	2011-10-01 23:45:00.00000
sm5	5	2011-10-01 23:45:00.00000
sm5	11	2011-10-01 23:45:00.00000

Related concepts:

“Delete time series data” on page 3-37

Related tasks:

“Creating containers” on page 3-15

Related reference:

“GetIndex function” on page 7-55

“GetContainerName function” on page 7-51

TSContainerSetPool procedure

The **TSContainerSetPool** procedure moves the specified container into the specified container pool.

Syntax

```
TSContainerSetPool(  
    container_name varchar(128,1),  
    pool_name varchar(128,1) default null);
```

```
TSContainerSetPool(  
    container_name varchar(128,1));
```

container_name

The name of the container to move.

pool_name

The name of the container pool in which to move the container.

Description

You can use the **TSContainerSetPool** procedure to move a container into a container pool, move a container from one container pool to another, or remove a container from a container pool. Containers that created automatically are in the container pool named **autopool** by default. If you create a container with the **TSContainerCreate** procedure, the container does not belong to a container pool until you run the **TSContainerSetPool** procedure to move it into a container pool.

If the container pool specified in the **TSContainerSetPool** procedure does not exist, the procedure creates it.

To move a container from one container pool to another, run the **TSContainerSetPool** procedure and specify the destination container pool name.

To move a container out of a container pool, run the **TSContainerSetPool** procedure without a container pool name.

The **TSContainerTable** table contains a row for each container and the container pool to which the container belongs.

Example 1: Move a container into a container pool

The following statement moves a container named **ctn_1** into a container pool that is named **smartmeter_pool**:

```
EXECUTE PROCEDURE TSContainerSetPool  
    ('ctn_1', 'smartmeter_pool');
```

Example 2: Remove a container from a container pool

The following statement removes a container named **ctn_1** from its container pool:

```
EXECUTE PROCEDURE TSContainerSetPool  
    ('ctn_1');
```

Related concepts:

“TSInstanceTable table” on page 2-12

Related tasks:

“Creating containers” on page 3-15

“Configuring additional container pools” on page 3-20

TSContainerTotalPages function

The **TSContainerTotalPages** function returns the total number of pages that are allocated to the specified container or in all containers.

Syntax

```
TSContainerTotalPages(container_name  varchar(128,1));  
TSContainerTotalPages(container_name  varchar(128,1),  
                      rw_flag          integer default 0);
```

container_name

Specifies which container to return information about. Must be an existing container name. You can include wildcard characters from the MATCHES operator: *, ?, [...], \, ^. The function returns information for all containers that have names that match the expression. See MATCHES operator.

The value NULL returns information about all containers for the database.

rw_flag

For rolling window containers, specifies for which partitions to return the sum of the total pages that are allocated:

0 = The partitions in the active window

1 = The partitions in the dormant window

2 = The container partition.

3 = All partitions.

Description

Use the **TSContainerTotalPages** function to view the size of a container or all containers. For rolling window containers, the **TSContainerTotalPages** function returns the sum of size of the specified set of partitions.

Returns

The number of allocated pages.

Example

The following statement returns the number of pages that are allocated to the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerTotalPages("mult_container");
```

```
total
```

```
50
```

```
1 row(s) retrieved.
```

The following statement returns the number of pages allocated to all the containers:

```
EXECUTE FUNCTION TSContainerTotalPages(NULL);
```

```
total
```

1 row(s) retrieved.

Related concepts:

“Monitor containers” on page 3-18

Related reference:

“TSContainerUsage function” on page 7-114

“TSContainerTotalUsed function”

“TSContainerPctUsed function” on page 7-105

“TSContainerNElems function” on page 7-104

“Time series storage” on page 1-14

TSContainerTotalUsed function

The **TSContainerTotalUsed** function returns the total number of pages that contain time series data in the specified container or in all containers.

Syntax

```
TSContainerTotalUsed(container_name varchar(128,1));
TSContainerTotalUsed(container_name varchar(128,1),
                    rw_flag      integer default 0);
```

container_name

Specifies which container to return information about. Must be an existing container name. You can include wildcard characters from the MATCHES operator: *, ?, [...], \, ^. The function returns information for all containers that have names that match the expression. See MATCHES operator.

The value NULL returns information about all containers for the database.

rw_flag

For rolling window containers, specifies for which partitions to return the sum of the total number of pages that contain time series elements:

0 = The partitions in the active window

1 = The partitions in the dormant window

2 = The container partition.

3 = All partitions.

Description

Use the **TSContainerTotalUsed** function to view the amount of data in a container or in all containers. For rolling window containers, the **TSContainerTotalUsed** function returns the sum of the amount of data in the specified set of partitions.

Returns

The number of pages that contain time series data.

Example

The following statement returns the number of pages that are used by time series data in the container named **mult_container**:

```
EXECUTE FUNCTION TSContainerTotalUsed("mult_container");
```

```
pages
```

```
30
```

1 row(s) retrieved.

The following statement returns the number of pages that are used by time series data in all containers:

```
EXECUTE FUNCTION TSContainerTotalUsed(NULL);
```

```
pages
```

```
2029
```

1 row(s) retrieved.

Related concepts:

"Monitor containers" on page 3-18

Related reference:

"TSContainerUsage function"

"TSContainerTotalPages function" on page 7-112

"TSContainerPctUsed function" on page 7-105

"TSContainerNElems function" on page 7-104

"Time series storage" on page 1-14

TSContainerUsage function

The **TSContainerUsage** function returns information about the size and capacity of the specified container or of all containers.

Syntax

```
TSContainerUsage(container_name varchar(128,1));
```

```
TSContainerUsage(container_name varchar(128,1),  
                 rw_flag integer default 0);
```

container_name

Specifies which container to return information about. Must be an existing container name. You can include wildcard characters from the MATCHES operator: *, ?, [...], \, ^. The function returns information for all containers that have names that match the expression. See MATCHES operator.

The value NULL returns information about all containers for the database.

rw_flag

For rolling window containers, specifies for which partitions to return the sum of storage space usage:

0 = The partitions in the active window

1 = The partitions in the dormant window

2 = The container partition.

3 = All partitions.

Description

Use the **TSContainerUsage** function to monitor how full the specified container is. For rolling window containers, the **TSContainerTotalUsage** function returns summary values for how full the specified set of partitions is. You can use the information from this function to determine how quickly your containers are filling and whether you must allocate more storage space.

Returns

The number of pages that contain time series data in the `pages` column, the number of elements in the `slots` column, and the number of pages that are allocated to the container in the `total` column.

Example: Monitor a container

The following statement returns the information for the container that is named **mult_container**:

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

pages	slots	total
30	26	50

1 row(s) retrieved.

This container has 26 time series data elements that use 30 pages out of the total 50 pages of space. Although the container is almost half empty, the container can probably accommodate fewer than 20 more time series elements.

Example: Monitor all containers

The following statement returns the information for all containers:

```
EXECUTE FUNCTION TSContainerUsage(NULL);
```

pages	slots	total
2029	241907	2169

1 row(s) retrieved.

The containers have only 140 pages of available space.

Example: Monitor a group of containers

Suppose that you have containers that have the following names:

- **active_cnt1**
- **active_cnt2**
- **historical_cnt1**

The following statement returns the information for the two containers that have names that begin with **active**:

```
EXECUTE FUNCTION TSContainerUsage(active*);
```

pages	slots	total
-------	-------	-------

1 row(s) retrieved.

Related concepts:

“Monitor containers” on page 3-18

Related reference:

“TSContainerTotalPages function” on page 7-112

“TSContainerTotalUsed function” on page 7-113

“TSContainerPctUsed function” on page 7-105

“TSContainerNElems function” on page 7-104

“Time series storage” on page 1-14

TSCreate function

The **TSCreate** function creates an empty regular time series or a regular time series populated with the given set of data. The new time series can also have user-defined metadata attached to it.

Syntax

```
TSCreate(cal_name      lvvarchar,
        origin       datetime year to fraction(5),
        threshold    integer,
        zero         integer,
        nelems       integer,
        container_name lvvarchar)
returns TimeSeries with (handlesnulls);
```

```
TSCreate(cal_name      lvvarchar,
        origin       datetime year to fraction(5),
        threshold    integer,
        zero         integer,
        nelems       integer,
        container_name lvvarchar,
        set_rows     set)
returns TimeSeries with (handlesnulls);
```

```
TSCreate(cal_name      lvvarchar,
        origin       datetime year to fraction(5),
        threshold    integer,
        zero         integer,
        nelems       integer,
        container_name lvvarchar,
        metadata     TimeSeriesMeta)
returns TimeSeries with (handlesnulls);
```

```
TSCreate(cal_name      lvvarchar,
        origin       datetime year to fraction(5),
        threshold    integer,
        zero         integer,
        nelems       integer,
        container_name lvvarchar,
        metadata     TimeSeriesMeta,
        set_rows     set)
returns TimeSeries with (handlesnulls);
```

cal_name

The name of the calendar for the time series.

origin The origin of the time series. This is the first valid date from the calendar for which data can be stored in the series.

threshold

The threshold for the time series. If the time series stores more than this number of elements, it is converted to a container. Otherwise, it is stored directly in the row that contains it, not in a container. The default is 20. The size of a row containing an in-row time series should not exceed 1500 bytes.

If a time series has too many bytes to fit in a row before this threshold is reached, the time series is put into a container at that point.

zero Must be 0.

nelems The number of elements allocated for the resultant time series. If the number of elements exceeds this value, the time series is expanded through reallocation.

container_name

The name of the container used to store the time series. Can be NULL.

metadata

The user-defined metadata to be put into the time series. See “Creating a time series with metadata” on page 3-23 for more information about metadata.

set_rows

A set of row type values used to populate the time series. The type of these rows must be the same as the subtype of the time series.

Description

If **TSCreate** is called with a *metadata* argument, then the metadata is saved in the time series.

Returns

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

Example

The following example creates an empty time series using **TSCreate**:

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

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

```
select TSCreate('daycal',  
  '2011-01-05 00:00:00.00000',  
  20,  
  0,  
  NULL,  
  set_data)::TimeSeries(stock_trade)  
from activity_load_tab  
where stock_id = 600;
```

Related tasks:

“Creating a time series with metadata” on page 3-23

Related reference:

“Create a time series” on page 3-22

“GetCalendar function” on page 7-48

“GetInterval function” on page 7-55
 “GetMetaData function” on page 7-59
 “GetMetaTypeName function” on page 7-59
 “GetOrigin function” on page 7-64
 “PutElem function” on page 7-77
 “PutSet function” on page 7-80
 “TSCreateIrr function”
 “UpdMetaData function” on page 7-159
 “The ts_create() function” on page 9-17
 “The ts_create_with_metadata() function” on page 9-18
 “The ts_get_metadata() function” on page 9-31
 “The ts_update_metadata() function” on page 9-54

TSCreateIrr function

The **TSCreateIrr** function creates an empty irregular time series or an irregular time series that is populated with the specified multiset of data. The new time series can also have user-defined metadata that is attached to it.

Syntax

```
TSCreateIrr(cal_name      lvarchar,
            origin        datetime year to fraction(5),
            threshold     integer,
            hertz         integer,
            nelems        integer,
            container_name lvarchar)
returns TimeSeries with (handlesnulls);
```

```
TSCreateIrr(cal_name      lvarchar,
            origin        datetime year to fraction(5),
            threshold     integer,
            hertz         integer,
            container_name lvarchar,
            multiset_rows multiset)
returns TimeSeries with (handlesnulls);
```

```
TSCreateIrr(cal_name      lvarchar,
            origin        datetime year to fraction(5),
            threshold     integer,
            hertz         integer,
            container_name lvarchar,
            metadata      TimeSeriesMeta)
returns TimeSeries with (handlesnulls);
```

```
TSCreateIrr(cal_name      lvarchar,
            origin        datetime year to fraction(5),
            threshold     integer,
            hertz         integer,
            container_name lvarchar,
            metadata      TimeSeriesMeta,
            multiset_rows multiset)
returns TimeSeries with (handlesnulls);
```

```
TSCreateIrr(cal_name      lvarchar,
            origin        datetime year to fraction(5),
            container_name lvarchar,
            compression  lvarchar)
returns TimeSeries with (handlesnulls);
```



```
TSCreateIrr(cal_name    lvarchar,
            origin      datetime year to fraction(5),
            container_name lvarchar,
            compression  lvarchar,
            multiset_rows multiset)
returns TimeSeries with (handlesnulls);
```

```
TSCreateIrr(cal_name    lvarchar,
            origin      datetime year to fraction(5),
            container_name lvarchar,
            compression  lvarchar,
            metadata     TimeSeriesMeta
        )
returns TimeSeries with (handlesnulls);
```

```
TSCreateIrr(cal_name    lvarchar,
            origin      datetime year to fraction(5),
            container_name lvarchar,
            compression  lvarchar,
            metadata     TimeSeriesMeta,
            multiset_rows multiset
        )
returns TimeSeries with (handlesnulls);
```

cal_name

The name of the calendar for the time series.

origin The origin of the time series, which is the first valid date from the calendar for which data can be stored in the series.

threshold

The threshold for the time series. If the time series stores more than this number of elements, it is converted to a container. Otherwise, it is stored directly in the row that contains it. The default is 20. The size of a row that contains an in-row time series cannot exceed 1500 bytes.

If a time series has too many bytes to fit in a row before this threshold is reached, the time series is put into a container.

hertz (**Optional**)

An integer that specifies whether the times series stores hertz data:

0 = The time series does not contain hertz data.

1 - 255 = The number of records per second.

If you set the *hertz* parameter to a value other than 0, the values of the *threshold* parameter must be 0.

nelems (**Optional**)

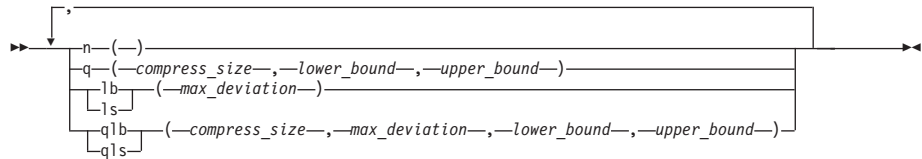
The number of elements that are allocated for the resultant time series. If the number of elements exceeds this value, the time series is expanded through reallocation.

container_name

The name of the container that is used to store the time series. Can be NULL.

compression (**Optional**)

Syntax of the compression parameter



A string that includes a compression definition for each column in the **TimeSeries** subtype except the first column. List the compression definition for each field in the order of the fields in the subtype, which are separated by commas. The compression definition consists of the compression type and the corresponding compression attributes:

Compression types:

- n = None. The column is not compressed.
- q = Quantization
- lb = Linear boxcar
- ls = Linear swing door
- qlb = Quantization linear boxcar
- qls = Quantization linear swing door

Compression attributes:

- *compress_size* = The number of bytes to store: 1, 2, or 4. The value must be smaller than the size of the associated column.
- *lower_bound* = A number that represents the lower boundary of acceptable values. The range of values is the same as the data type of the associated column.
- *upper_bound* = A number that represents the upper boundary of acceptable values. The range of values is the same as the data type of the associated column.
- *max_deviation* = A positive floating point number that represents the maximum deviation between the actual value and the compressed value. The range of values is 0 through the largest value of the data type of the associated column. The value 0 indicates that no deviation is allowed.

See "Compressed numeric time series" on page 1-10.

metadata (Optional)

The user-defined metadata to be put into the time series.

multiset_rows (Optional)

A multiset of rows to populate the time series. The type of these rows must be the same as the subtype of the time series.

Description

If the **TSCreateIrr** function is called with the *metadata* parameter, then metadata is saved in the time series.

If you include the *compression* parameter, each time series element is packed with compressed records until the size of the element approaches 4 KB or the transaction is committed. You must run the **TSCreateIrr** function within an explicit transaction.

If you include the *hertz* parameter, each time series element is packed with the number of records that are specified by the *hertz* parameter. An element is saved to disk after a record for the last subsecond boundary is inserted or the transaction is

committed. You must run the **TSCreateIrr** function within an explicit transaction.

Returns

An irregular, hertz, or compressed time series.

Example: Create an empty time series

The following example creates an empty irregular time series:

```
select TSCreateIrr('daycal',
    '2011-01-05 00:00:00.00000',
    20,
    0,
    NULL,
    set_data)::TimeSeries(stock_trade)
from activity_load_tab
where stock_id = 600;
```

Example: Create a populated time series

The following example creates a populated irregular time series:

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

Example: Create a compressed time series

When you create a compressed time series, you need to know the structure of the **TimeSeries** subtype and characteristics about the data in each column. The following statement creates a **TimeSeries** row type that has a timestamp column and six other columns of numeric data:

```
CREATE ROW TYPE irregular_t
(
    tstamp DATETIME YEAR TO FRACTION(5),
    key1 smallint,
    key2 int,
    key3 bigint,
    key4 smallfloat,
    key5 float,
    key6 int
);
```

The following statement creates a compressed time series instance in the table **tstable**:

```
BEGIN;
INSERT INTO tstable VALUES(1,
    TSCreateIrr('ts_1sec', '2013-01-01 00:00:00.00000',
        'container 2k', 'n()',q(1,1,100),ls(0.10),
        lb(0.10),qls(2,0.15,100,100000),qlb(2,0.25,100,100000)'))
COMMIT;
```

The columns in the **irregular_t** row type are compressed in the following ways:

- The **key1** column is not compressed.
- The **key2** column is compressed by the quantization compression type with a compression size of 1 byte, a lower bound of 1 and an upper bound of 100.

- The **key3** column is compressed by the linear swing door compression type with a maximum deviation of 0.10.
- The **key4** column is compressed by the linear boxcar compression type with a maximum deviation of 0.10.
- The **key5** column is compressed by the quantization linear swing door compression type with a compression size of 2 bytes, a maximum deviation of 0.15, a lower bound of 100, and an upper bound of 100000.
- The **key6** column is compressed by quantization linear boxcar with a compression size of 2 bytes, a maximum deviation of 0.25, a lower bound of 100, and an upper bound of 100000.

Example: Create a hertz time series

The following statement creates a time series that stores 50 records per second in each element:

```
BEGIN;
INSERT INTO tstable VALUES(1,
                           TSCreateIrr('ts_1sec', '2014-01-01 00:00:00.00000',
                                         0, 50, 0, 'container1'))
COMMIT;
```

Related concepts:

“Hertz time series” on page 1-8

“Compressed numeric time series” on page 1-10

Related tasks:

“Creating a time series with metadata” on page 3-23

Related reference:

“Create a time series” on page 3-22

“GetMetaData function” on page 7-59

“GetMetaTypeName function” on page 7-59

“TSCreate function” on page 7-116

“The ts_create_with_metadata() function” on page 9-18

“The ts_create() function” on page 9-17

“UpdMetaData function” on page 7-159

“The ts_get_metadata() function” on page 9-31

“The ts_update_metadata() function” on page 9-54

TSDecay function

The **TSDecay** function computes a decay function over its arguments.

Syntax

```
TSDecay(current_value smallfloat,
       initial_value  smallfloat,
       decay_factor   smallfloat)
returns smallfloat;
```

```
TSDecay(current_value double precision,
       initial_value  double precision,
       decay_factor   double precision)
returns double precision;
```

current_value

The current datum (v_j in the sum shown next).

initial_value

The initial value (*initial* in the sum shown next).

decay_factor

The decay factor (*decay* in the sum shown next).

Description

All three arguments must be of the same type.

The function maintains a sum of all the arguments it has been called with so far. Every time it is called, the sum is multiplied by the supplied decay factor. Given a decay factor between 0 and 1, this causes the importance of older arguments to fall off over time. The first time that **TSDecay** is called, it includes the supplied initial value in the running sum. The actual function that **TSDecay** computes is:

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

In this computation, i is the number of times the function has been called so far, and v_j is the value it was called with in its j th invocation.

This function is useful only when used within the **Apply** function.

Returns

The result of the decay function.

Example

The following example computes the decay:

```
create function ESA18(a smallfloat) returns smallfloat;  
return (.18 * a) + TSDecay(.18 * a, a, .82);  
end function;
```

Related reference:

“Apply function” on page 7-18

“TSAddPrevious function” on page 7-90

“TSCmp function” on page 7-90

“TSPrevious function” on page 7-141

“TSRunningAvg function” on page 7-147

“TSRunningSum function” on page 7-151

TSL_Attach function

The **TSL_Attach** function opens a database session for loading data.

Syntax

```
TSL_Attach(  
    table_name varchar(128),  
    column_name varchar(128),  
    reject_file varchar(255) default NULL)  
returns lvarchar
```

```
TSL_Attach(
    table_name varchar(128),
    column_name varchar(128),
returns lvarchar
```

table_name

The name of the time series table. Must not contain uppercase letters. Must match the table name supplied to the **TSL_Init** function to initialize the loader session.

column_name

The name of the **TimeSeries** column. Must not contain uppercase letters. Must match the column name supplied to the **TSL_Init** function to initialize the loader session.

reject_file (optional)

The path and file name for storing records that were not applied. For example, records with an incorrect number of fields or a formatting error are not applied. By default, only the number of rejected records is recorded. Overwrites the reject file specified in the **TSL_Init** function.

Usage

Use the **TSL_Attach** function to open an additional database session to load time series data in parallel. You must run the **TSL_Attach** function in the context of a loader session that was initialized by the **TSL_Init** function.

Returns

- A session handle that consists of a table name and a **TimeSeries** column name.
- An exception or NULL if the session was not initialized.

Example

The following statement opens a database session for the table **ts_data** and the **TimeSeries** column **raw_reads**:

```
EXECUTE FUNCTION TSL_Attach('ts_data','raw_reads');
```

TSL_Commit function

The **TSL_Commit** function flushes data for all containers to disk in multiple transactions.

Syntax

```
TSL_Commit(
    handle          lvarchar,
    writeflag       integer DEFAULT 1,
    commit_interval integer DEFAULT NULL)
returns integer
```

handle The table and column name combination that is returned by the **TSL_Attach** or the **TSL_Init** function.

writeflag (Optional)

An integer that represents whether the duplicate elements are allowed for irregular time series and whether logging is reduced. You must supply a value for the duplicate element behavior (1 or 5) and can optionally add the value for reduced logging (256).

1 = Default. Duplicate elements are allowed.

5 = Duplicate elements replace existing elements.

(1 + 256 = 257) = Duplicate elements are allowed. Logging is reduced. See the description of the `TSOPEN_REDUCED_LOG` flag in “The flags argument values” on page 7-9.

(5 + 256 = 261) = Duplicate elements replace existing elements. Logging is reduced.

commit_interval (**Optional**)

A positive integer that represents the maximum number of elements to insert per transaction. The default value if the argument is omitted or set to NULL is 10000.

Usage

Use the **TSL_Commit** function to write time series data to disk as part of a loader program. You must run the **TSL_Commit** function in the context of a loader session that was initialized by the **TSL_Init** function. You run the **TSL_Commit** function to save data that is loaded by the **TSL_Put**, **TSL_PutSQL**, or **TSL_PutRow** function. The **TSL_Commit** function is useful when the amount of data that you are loading is too large for your logical log to flush as one transaction.

The **TSL_Commit** function commits transactions after flushing the specified number of elements, or when all elements for a container are flushed. For example, suppose that you set the *commit_interval* argument to 5000. If you load 10 000 elements into a container, the **TSL_Commit** function commits two transactions. If you insert 10 000 elements in equal amounts into five containers, the **TSL_Commit** function commits five transactions of 2000 elements. If you run the **TSL_Commit** function within an explicit transaction, then the **TSL_Commit** function does not commit transactions.

If you specify that duplicate elements are allowed for irregular time series with the *writeflag* argument value of 1 (the default) or 257 (1 + 256), the **TSL_Commit** function inserts data in the same way as the **PutElem** function. You can specify that duplicate elements replace existing elements that have the same timestamps with the *writeflag* argument value of 5 or 261 (5 + 256), so that the **TSL_Commit** function inserts data in the same way as the **PutElemNoDups** function.

If you specify reduced logging with the *writeflag* argument value of 257 (1 + 256) or 261 (5 + 256), you must run **TSL_Commit** function in a transaction that can include only other functions that use reduced logging with the `TSOPEN_REDUCED_LOG` flag. The elements that are saved are not visible by dirty reads until after the transaction commits.

The **TSL_FlushInfo** function returns eight categories of information about the last flush operation that saved data to disk.

Returns

An integer that indicates the status of the function:

- A positive integer = The number of elements that were inserted.
- -1 = Process was interrupted or encountered a severe error.

Example: Run the TSL_Commit function

The following statement saves the data to disk after every 5000 elements are inserted for the table **ts_data** and the **TimeSeries** column **raw_reads**:

```
EXECUTE FUNCTION TSL_Commit('ts_data|raw_reads', 1, 5000);
```

Example: Run the TSL_Commit function with reduced logging

The following statement saves the data to disk with reduced logging after every 20 000 elements are inserted:

```
EXECUTE FUNCTION TSL_Commit('ts_data|raw_reads', 257, 20000);
```

Example: Run the TSL_Commit function with reduced logging and no duplicate elements

The following statement saves the data to disk with reduced logging and replaces existing elements that have the same timestamps as new elements:

```
EXECUTE FUNCTION TSL_Commit('ts_data|raw_reads', 261, 200000);
```

Related tasks:

“Writing a loader program” on page 3-31

Related reference:

“TSL_FlushInfo function” on page 7-129

TSL_Flush function

The **TSL_Flush** function flushes data for one or all containers to disk in a single transaction.

Syntax

```
TSL_Flush(  
    handle          lvarchar,  
    container       integer DEFAULT NULL,  
    writeflag       integer DEFAULT 1)  
returns integer
```

handle The table and column name combination that is returned by the **TSL_Attach** or the **TSL_Init** function.

container (Optional)

The container identifier. Default is NULL, which indicates all containers.

writeflag (Optional)

An integer that represents whether the duplicate elements are allowed for irregular time series and whether logging is reduced. You must supply a value for the duplicate element behavior (1 or 5) and can optionally add the value for reduced logging (256).

1 = Default. Duplicate elements are allowed.

5 = Duplicate elements replace existing elements.

(1 + 256 = 257) = Duplicate elements are allowed. Logging is reduced. See the description of the TSOPEN_REDUCED_LOG flag in “The flags argument values” on page 7-9.

(5 + 256 = 261) = Duplicate elements replace existing elements. Logging is reduced.

Usage

Use the **TSL_Flush** function to write time series data to disk as part of a loader program. You must run the **TSL_Flush** function in a transaction in the context of a loader session that was initialized by the **TSL_Init** function. You run the **TSL_Flush** function to save data that is loaded by the **TSL_Put**, **TSL_PutSQL**, or **TSL_PutRow** function.

If you specify that duplicate elements are allowed for irregular time series with the *writeflag* argument value of 1 (the default) or 257 (1 + 256), the **TSL_Flush** function inserts data in the same way as the **PutElem** function. You can specify that duplicate elements replace existing elements that have the same timestamps with the *writeflag* argument value of 5 or 261 (5 + 256), so that the **TSL_Flush** function inserts data in the same way as the **PutElemNoDups** function.

If you specify reduced logging with the *writeflag* argument value of 257 (1 + 256) or 261 (5 + 256), you must run **TSL_Flush** function in a transaction that can include only other functions that use reduced logging with the **TSOPEN_REDUCED_LOG** flag. The elements that are saved are not visible by dirty reads until after the transaction commits.

The **TSL_FlushInfo** function returns eight categories of information about the last flush operation that saved data to disk.

Returns

An integer that indicates the status of the function:

- Positive 4-byte number = The top 16 bits represent the number of time series instances that were affected. The lower 16 bits represent the number of elements that were inserted. The returned number of elements that were inserted does not exceed 65535, even if more than 65535 elements were successfully inserted. You can run the **TSL_Flush** function through the **TSL_FlushStatus** function to return the number of containers that are affected and the number of elements that are inserted as integers.
- -1 = Process was interrupted or encountered a severe error.

Example: Run the TSL_Flush function

The following statement saves the data to disk for the table **ts_data** and the **TimeSeries** column **raw_reads**:

```
BEGIN WORK;  
EXECUTE FUNCTION TSL_Flush('ts_data|raw_reads');  
COMMIT WORK;
```

Example: Run the TSL_Flush function with reduced logging

The following statement saves the data to disk with reduced logging:

```
BEGIN WORK;  
EXECUTE FUNCTION TSL_Flush('ts_data|raw_reads', NULL, 257);  
COMMIT WORK;
```

Example: Run the TSL_Flush function with reduced logging and no duplicate elements

The following statement saves the data to disk with reduced logging and replaces existing elements that have the same timestamps as new elements:

```
BEGIN WORK;
EXECUTE FUNCTION TSL_Flush('ts_data|raw_reads', NULL, 261);
COMMIT WORK;
```

Related reference:

“TSL_FlushInfo function” on page 7-129

“TSL_FlushStatus function” on page 7-131

TSL_FlushAll function

The **TSL_FlushAll** function flushes data for all containers to disk in a single transaction.

Syntax

```
TSL_FlushAll(
    handle          lvarchar,
    writeflag       integer DEFAULT 1)
returns integer
```

handle The table and column name combination that is returned by the **TSL_Attach** or the **TSL_Init** function.

writeflag **(Optional)**

An integer that represents whether the duplicate elements are allowed for irregular time series and whether logging is reduced. You must supply a value for the duplicate element behavior (1 or 5) and can optionally add the value for reduced logging (256).

1 = Default. Duplicate elements are allowed.

5 = Duplicate elements replace existing elements.

(1 + 256 = 257) = Duplicate elements are allowed. Logging is reduced. See the description of the TSOPEN_REDUCED_LOG flag in “The flags argument values” on page 7-9.

(5 + 256 = 261) = Duplicate elements replace existing elements. Logging is reduced.

Usage

Use the **TSL_FlushAll** function to write time series data to disk as part of a loader program. You must run the **TSL_FlushAll** function in a transaction in the context of a loader session that was initialized by the **TSL_Init** function. You run the **TSL_FlushAll** function to save data that is loaded by the **TSL_Put**, **TSL_PutSQL**, or **TSL_PutRow** function.

If you specify that duplicate elements are allowed for irregular time series with the *writeflag* argument value of 1 (the default) or 257 (1 + 256), the **TSL_FlushAll** function inserts data in the same way as the **PutElem** function. You can specify that duplicate elements replace existing elements that have the same timestamps with the *writeflag* argument value of 5 or 261 (5 + 256), so that the **TSL_FlushAll** function inserts data in the same way as the **PutElemNoDups** function.

If you specify reduced logging with the *writeflag* argument value of 257 (1 + 256) or 261 (5 + 256), you must run **TSL_FlushAll** function in a transaction that can include only other functions that use reduced logging with the TSOPEN_REDUCED_LOG flag. The elements that are saved are not visible by dirty reads until after the transaction commits.

The **TSL_FlushInfo** function returns eight categories of information about the last flush operation that saved data to disk.

Returns

An integer that indicates the status of the function:

- A positive integer = The number of elements that were inserted.
- -1 = Process was interrupted or encountered a severe error.

Example: Run the TSL_FlushAll function

The following statement saves the data to disk for the table **ts_data** and the **TimeSeries** column **raw_reads**:

```
BEGIN WORK;  
EXECUTE FUNCTION TSL_FlushAll('ts_data|raw_reads');  
COMMIT WORK;
```

Example: Run the TSL_FlushAll function with reduced logging

The following statement saves the data to disk with reduced logging:

```
BEGIN WORK;  
EXECUTE FUNCTION TSL_FlushAll('ts_data|raw_reads', 257);  
COMMIT WORK;
```

Example: Run the TSL_FlushAll function with reduced logging and no duplicate elements

The following statement saves the data to disk with reduced logging and replaces existing elements that have the same timestamps as new elements:

```
BEGIN WORK;  
EXECUTE FUNCTION TSL_FlushAll('ts_data|raw_reads', 261);  
COMMIT WORK;
```

Related tasks:

“Writing a loader program” on page 3-31

Related reference:

“TSL_FlushInfo function”

TSL_FlushInfo function

The **TSL_FlushInfo** function returns information about the last flush operation that saved data to disk.

Syntax

```
TSL_FlushInfo(  
             handle          lvarchar)  
returns TSL_FlushInfo_r
```

handle The table and column name combination that was included in a **TSL_Flush**, **TSL_FlushAll**, or **TSL_Commit** function.

Usage

Use the **TSL_FlushInfo** function to view information about the last time that time series data for the specified table and column name was saved to disk as part of a loader program. You can flush data to disk with the **TSL_Flush**, **TSL_FlushAll**, or **TSL_Commit** function. You must run the **TSL_FlushInfo** function in the context of

a loader session that was initialized by the **TSL_Init** function.

Returns

A row type that has the following format:

```
TSL_FlushInfo_r
(
  containers    integer,
  elements      integer,
  duplicates    integer,
  instance_ids  integer,
  commits       integer,
  rollbacks     integer,
  exceptions    integer,
  errors        integer
);
```

containers

The number of containers that were affected by the flush.

elements

The number of elements that were affected by the flush.

duplicates

The number of duplicate values that were inserted for irregular time series, if the *writeflag* value was 1 or 257.

instance_ids

The number of time series instances that were affected by the flush.

commits

The number of committed transactions, for the **TSL_Commit** function. For the **TSL_Flush** and **TSL_FlushAll** functions, this number is always 0.

rollbacks

For the **TSL_Commit** function, whether transactions were rolled back:

0 = no rolled back transactions

1 = a transaction was interrupted by the user

exceptions

The number of exceptions that were raised during the flush. For example, attempting to insert an element that has a timestamp that is before the origin results in an exception. Attempting to save data into a container that does not have enough available space also results in an exception.

errors The number of errors in the load operation. For example, incorrectly formatted input data results in an error.

Example

The following statement returns the results of the **TSL_FlushInfo** function as a table:

```
SELECT mr.*
FROM TABLE(TSL_FlushInfo('ts_data|raw_reads')) AS TAB(mr);

containers    4
elements      198458
duplicates    0
instance_ids  522
```

commits 4
rollbacks 0
exceptions 0
errors 0

Related tasks:

“Writing a loader program” on page 3-31

Related reference:

“TSL_Commit function” on page 7-124

“TSL_FlushAll function” on page 7-128

“TSL_Flush function” on page 7-126

TSL_FlushStatus function

The **TSL_FlushStatus** function presents the return value of the **TSL_Flush** function in a format that is easier to read.

Syntax

```
TSL_FlushStatus(  
    value integer)  
returns integer, integer  
  
value    The TSL_Flush function.
```

Returns

Two 16-bit integers that indicate the status of the **TSL_Flush** function:

- The number of containers into which elements are inserted.
- The total number of elements that are inserted.

A return value of -1 indicates that the process was interrupted or encountered a severe error.

Example

The following statement runs the **TSL_Flush** function and returns the number of containers that are affected and the number of elements that are inserted as integers:

```
EXECUTE FUNCTION TSL_FlushStatus(TSL_Flush('ts_data|raw_reads'));
```

Related reference:

“TSL_Flush function” on page 7-126

TSL_GetKeyContainer function

The **TSL_GetKeyContainer** function returns the name of the container that stores the time series values for the specified primary key value.

Syntax

```
TSL_GetKeyContainer(  
    handle        lvarchar,  
    key_value    lvarchar)  
returns varchar(128)  
  
handle    The table and column name combination returned by the TSL_Attach or  
          the TSL_Init function.
```

key_value

The primary key value. If the primary key contains multiple columns, separate each value by a pipe character.

Usage

Use the **TSL_GetKeyContainer** function to determine into which container the specified data is loaded. If you load data in multiple sessions, you can configure the sessions to load into different containers. You must run the **TSL_GetKeyContainer** function in the context of a loader session that was initialized by the **TSL_Init** function.

Returns

The name of the container.

Examples

The following statement returns the name of the container associated with the time series data for the primary key value of 1 in the **meter** table:

```
EXECUTE FUNCTION TSL_GetKeyContainer('meter|readings', '1');
```

Related tasks:

“Writing a loader program” on page 3-31

TSL_GetLogMessage function

The **TSL_GetLogMessage** function specifies how many messages are retrieved from the loader message queue.

Syntax

```
TSL_GetLogMessage(  
    handle          lvarchar,  
    max_messages    integer)  
returns lvarchar
```

handle The table and column name combination returned by the **TSL_Attach** function.

max_messages

0 = All messages are retrieved.

A positive integer = The number of messages to retrieve. If the number is greater than the total number of messages in the queue, all messages are retrieved.

Usage

Use the **TSL_GetLogMessage** function to retrieve loader messages as part of a loader program. To retrieve messages, the log mode must be set to 2 so that messages are sent to the loader program. The log mode is set by the **TSL_Init** function or reset by the **TSL_SetLogMode** function. You must run the **TSL_GetLogMessage** function in the context of a loader session that was initialized by the **TSL_Init** function. You can use the **TSL_GetLogMessage** function to monitor that data that is loaded by the **TSL_Put** function and saved to disk by the **TSL_Flush** function.

The **TSL_GetLogMessage** function is an iterator function. You can retrieve the messages with an SQL cursor through a virtual table.

Returns

The specified number of messages from the loader message queue. If NULL is returned, the loader message queue is empty.

Examples

In the context of a loader program, the following statement retrieves 50 messages from the loader message log:

```
EXECUTE FUNCTION TSL_GetLogMessage('ts_data-raw_reads',50);
```

Related tasks:

“Writing a loader program” on page 3-31

TSL_Init function

The **TSL_Init** function initializes a session for loading data.

Syntax

```
TSL_Init(  
    table_name      varchar(128),  
    column_name     varchar(128))  
returns lvarchar;
```

```
TSL_Init(  
    table_name      varchar(128),  
    column_name     varchar(128),  
    tstamp_format   varchar(25))  
returns lvarchar;
```

```
TSL_Init(  
    table_name      varchar(128),  
    column_name     varchar(128),  
    tstamp_format   varchar(25),  
    reject_file     varchar(255))  
returns lvarchar;
```

```
TSL_Init(  
    table_name      varchar(128),  
    column_name     varchar(128),  
    log_type        integer,  
    log_level       integer,  
    logfile         lvarchar)  
returns lvarchar;
```

```
TSL_Init(  
    table_name      varchar(128),  
    column_name     varchar(128),  
    log_type        integer,  
    log_level       integer,  
    logfile         lvarchar,  
    tstamp_format   lvarchar)  
returns lvarchar;
```

```
TSL_Init(  
    table_name      varchar(128),  
    column_name     varchar(128),  
    log_type        integer,  
    log_level       integer,  
    logfile         lvarchar,
```

```

        tstamp_format  lvarchar,
        reject_file   lvarchar)
returns lvarchar;

```

```

TSL_Init(
    table_name      varchar(128),
    column_name     varchar(128),
    log_type        integer,
    log_level       integer,
    logfile         lvarchar,
    tstamp_format   lvarchar,
    reject_file     lvarchar,
    where_clause    lvarchar)
returns lvarchar;

```

table_name

The name of the time series table. Must not contain uppercase letters. The table must contain a **TimeSeries** column and have a primary key.

column_name

The name of the **TimeSeries** column. Must not contain uppercase letters.

log_type (Optional)

The type of message log:

0 = No message log

1 = Log all messages to the specified file

2 = Log all messages in a queue for the loader program for retrieval by the **TSL_GetLogMessage** function

3 = Default. Log all messages to the server message log

log_level (Optional)

The severity of information that is included in the message log file:

2 = Warning messages and error messages.

4 = Default. Error messages.

logfile (Optional)

If the value of the *log_type* argument is set to 1, the path and file name of the loader message log file. If the *log_type* argument is set to a value other than 1, the *logfile* argument is ignored.

tstamp_format (Optional)

The string that describes the format of the timestamp. The default format is: %Y-%m-%d %H:%M:%S. The string must conform to the format specified by the **DBTIME** environment variable, but can include a pipe character (|) between the date portion and the time portion. The pipe character indicates that the date and the time are in separate fields. For example, you can use a two-field format: %Y-%m-%d| %H:%M:%S.

reject_file (Optional)

The path and file name for storing records that were not applied. For example, records with an incorrect number of fields or a formatting error are not applied. By default, only the number of rejected records is recorded.

where_clause (Optional)

Additional predicate to append to the WHERE clause that is generated by loader. The predicate can identify a subset of time series data to load. Because the WHERE clause limits the entire loader session to a subset of

time series data, use caution when you include the WHERE clause. The predicate must start with the keyword AND. For example: AND meter_id = "A100"

Usage

Use the **TSL_Init** function to initialize a session as part of a loader program. The **TSL_Init** function must be the first function that you run in your loader program. The **TSL_Init** function creates a global context for the loader and opens a database session. You can open additional database sessions by running the **TSL_Attach** function. You can close a database session by running the **TSL_SessionClose** function. The global context remains in effect until you run the **TSL_Shutdown** function or restart the database server.

You must run the **TSL_Init** function within an EXECUTE FUNCTION statement. You cannot run the **TSL_Init** function in a SELECT statement.

Returns

- A session handle that consists of a table name and a **TimeSeries** column name.
- An exception or NULL if the session was not initialized.

Example

The following statement initializes a loader session for a table named **ts_data** and a **TimeSeries** column named **raw_reads**:


```
EXECUTE FUNCTION TSL_Init('ts_data','raw_reads',
                          '%Y-%m-%d %H:%M:%S','/tmp/rejects.log',NULL);
```

The input data uses a single-column timestamp format. The rejected records are saved in a file.

Related tasks:

“Writing a loader program” on page 3-31

Related reference:

 DBTIME environment variable (SQL Reference)

TSL_Put function

The **TSL_Put** function loads time series data.

Syntax

```
TSL_Put(
    handle          lvarchar,
    elementlist lvarchar)
returns integer
```

```
TSL_Put(
    handle          lvarchar,
    elementlist CLOB)
returns integer
```

handle The table and column name combination that is returned by the **TSL_Attach** or the **TSL_Init** function.

elementlist

A buffer that contains the time series data to load. That maximum size of a buffer is 32 KB. Can be a file as a CLOB data type.

The element list must have the following format:

primary_key | timestamp | values |

primary_key

One or more column values that comprise the primary key, which are separated by pipe characters. If this field is empty, the value from the previous line is used.

timestamp

The format of the timestamp is specified by the *timestamp_format* argument of the **TSL_Init** function. The default format of the timestamp is: *YYYY-mm-dd HH:MM:SS* (year-month-day hour:minute:seconds).

values

Values for the columns in the **TimeSeries** subtype. Separate multiple values with pipe characters and end the list of values with a pipe character.

Each element must be on a separate line. Each line must end with a newline character (\n). If necessary, enable newline characters in quoted strings by setting the **ALLOW_NEWLINE** configuration parameter or running the **IFX_ALLOW_NEWLINE()** procedure.

Usage

Use the **TSL_Put** function to load time series data as part of a loader program. You must run the **TSL_Put** function in the context of a loader session that was initialized by the **TSL_Init** function. You can run the **TSL_Put** function multiple times in the same session. The data is stored in the database server until you run the **TSL_Flush** function to write the data to disk.

Returns

- An integer that indicates the number of records that were inserted.
- An exception if no records were inserted.

Examples

These examples run in the context of an initialized loader session.

Example: Load an element

The following statement loads one element into a table named **tsdata** that has a primary key column named **pkcol**:

```
EXECUTE FUNCTION TSL_Put('tsdata|pkcol','MX230001|2011-01-01|00:00:00|23.4|56.7|');
```

Example 2: Load a file

The following statement loads data from a file name **tsdata.unl** that is converted to a CLOB data type:

```
EXECUTE FUNCTION TSL_Put('tsdata|pkcol',FileToClob('/data/tsdata.unl','server'));
```

Example 3: Element list format

The following element list contains a value for every primary key and date field, even if the values are the same as the previous element:

```

MX230001|2011-01-01|00:00:00|23.4|56.7|
MX230001|2011-01-01|01:00:00|34.7|57.8|
MX230001|2011-01-01|02:00:00|12.8|58.3|
MX230001|2011-01-01|03:00:00|18.4|59.1|
MX672382|2011-01-01|00:00:00|3.2|0.0|
MX672382|2011-01-01|01:00:00|4.7|0.0|
MX672382|2011-01-01|02:00:00|5.8|0.0|
MX672382|2011-01-01|03:00:00|1.3|0.0|

```

The following element list is equivalent to the previous list, but requires less room in the input buffer because duplicate primary key and date values are omitted:

```

MX230001|2011-01-01|00:00:00|23.4|56.7|
|01:00:00|34.7|57.8|
|02:00:00|12.8|58.8|
|03:00:00|18.4|59.1|
MX672382|2011-01-01|00:00:00|3.2|0.0|
|01:00:00|4.7|0.0|
|02:00:00|5.8|0.0|
|03:00:00|1.3|0.0|

```

The following element list has a primary key that has multiple columns. The values in the primary key that repeat are omitted:

```


MX23001|AQ74D|2011-01-01|00:00:00|23.11|98.43|
|AQ74E||22.71|97.65|
||00:01:00|22.69|94.56|
MX23002|AV90A|2011-01-01|00:00:00|23.12|91.43|

```

Related tasks:

“Writing a loader program” on page 3-31

Related reference:

 [ALLOW_NEWLINE configuration parameter \(Administrator's Reference\)](#)

 [IFX_ALLOW_NEWLINE Function \(SQL Syntax\)](#)

TSL_PutRow function

The **TSL_PutRow** function loads a row of time series data.

Syntax

```

TSL_PutRow(
    handle          lvvarchar,
    primary_key    lvvarchar,
    row             ROW)
returns integer

```

handle The table and column name combination returned by the **TSL_Attach** or the **TSL_Init** function.

primary_key

A primary key value. Can be a composite of multiple values separated by a pipe symbol. For example, if a primary key in the source table consists of two columns, **id1** and **id2**, a primary key value where **id1** is 5 and **id2** is 2 is represented as: '5|2'.

row

A ROW data type with values that are compatible with the **TimeSeries** column in the handle.

Usage

Use the **TSL_PutRow** function to load a row of time series data as part of a loader program. You must run the **TSL_PutRow** function in the context of a loader session that was initialized by the **TSL_Init** function.

You can run the **TSL_PutRow** function multiple times in the same session. The data is stored in the database server until you run the **TSL_Flush** function to write the data to disk.

Returns

- An integer that indicates the number of records that were inserted.
- An exception if no records were inserted.

Examples

This example runs in the context of an initialized loader session.

The following statement loads a row of data from a ROW data type that is cast to the **TimeSeries** data type named **readings**:

```
EXECUTE FUNCTION TSL_PutRow('meter|readings', '5',  
    row( datetime(2011-01-01 00:15:00.000000) year to fraction(5),  
        1.0)::reading);
```

Related tasks:

“Writing a loader program” on page 3-31

TSL_PutSQL function

The **TSL_PutSQL** function loads time series data from a table.

Syntax

```
TSL_PutSQL(  
    handle          lvarchar,  
    statement       lvarchar)  
returns integer
```

handle The table and column name combination that is returned by the **TSL_Attach** or the **TSL_Init** function.

statement

An SQL statement that selects data from an existing database table. The projection clause of the statement must consist of a primary key and the time series data. The time series data can be multiple columns or a ROW data type and must be compatible with the data type of the **TimeSeries** column in the handle. The project clause can be one of the following formats:

primary_key, timestamp, values

primary_key, row(timestamp, values)

primary_key

The primary key column, represented in character format. Can consist of multiple column names concatenated and separated by a pipe symbol. Cast to an LVARCHAR data type, if necessary. For example, if the primary key in the source table consists of two columns, named **id1** and **id2**, the primary key column in the projection clause is `id1 || '|' || id2`.

timestamp

The format of the timestamp is specified by the *timestamp_format* argument of the **TSL_Init** function. The default format of the timestamp is: *YYYY-mm-dd HH:MM:SS* (year-month-day hour:minute:seconds).

values

Values for the columns in the **TimeSeries** subtype. Separate multiple values with commas.

Usage

Use the **TSL_PutSQL** function to load time series data from another table as part of a loader program. You must run the **TSL_PutSQL** function in the context of a loader session that was initialized by the **TSL_Init** function.

You can run the **TSL_PutSQL** function multiple times in the same session. The data is stored in the database server until you run the **TSL_Flush** function to write the data to disk.

Returns

- An integer that indicates the number of records that were inserted.
- An exception if no records were inserted.

Examples

These examples run in the context of an initialized loader session.

Example 1: Load a primary key and multiple columns

The following example selects data from a table named **dataload** from a primary key column, a timestamp column, and a column with other values:

```
EXECUTE FUNCTION TSL_PutSQL('meter|readings',  
  'SELECT id::lvarchar, tstamp, value FROM dataload');
```

Example 2: Load a primary key and a ROW data type

The following example selects data from a table named **dataload** from a primary key column and a ROW data type that consists of a timestamp field and a value field:

```
EXECUTE FUNCTION TSL_PutSQL('meter|readings',  
  'SELECT id::lvarchar, row(tstamp, value) FROM dataload');
```

Related tasks:

“Writing a loader program” on page 3-31

TSL_SessionClose function

The **TSL_SessionClose** function closes the current database session.

Syntax

```
TSL_SessionClose(  
  handle lvarchar)  
returns int
```

handle The table and column name combination returned by the **TSL_Attach** or the **TSL_Init** function.

Usage

Use the **TSL_SessionClose** function to close the current database session. The **TSL_SessionClose** function does not affect the global context. Call the **TSL_Flush** function to flush data to disk before calling the **TSL_SessionClose** function.

Returns

- 0 = The session was closed.
- 1 = An error occurred.

Example

The following statement closes a database session for the table **ts_data** and the **TimeSeries** column **raw_reads**:

```
EXECUTE FUNCTION TSL_SessionClose('ts_data|raw_reads');
```

Related tasks:

“Writing a loader program” on page 3-31

TSL_SetLogMode function

The **TSL_SetLogMode** function specifies how messages about the loader are logged.

Syntax

```
TSL_SetLogMode(  
    handle          lvarchar,  
    log_type        integer,  
    log_level       integer,  
    logfile         lvarchar)  
returns integer
```

```
TSL_SetLogMode(  
    handle          lvarchar,  
    log_type        integer,  
    log_level       integer)  
returns integer
```

```
TSL_SetLogMode(  
    handle          lvarchar,  
    log_level       integer)  
returns integer
```

handle The table and column name combination returned by the **TSL_Attach** function.

log_type (Optional)

The type of message log:

0 = No message log

1 = Log all messages to the specified file

2 = Log all messages in a queue for the loader program for retrieval by the **TSL_GetLogMessage** function

3 = Default. Log all messages to the server message log

log_level

The severity of information that is included in the message log file:

2 = Warning messages and error messages.

4 = Default. Error messages.

logfile (Optional)

If the value of the *log_type* argument is set to 1, the path and file name of the loader message log file. If the *log_type* argument is set to a value other than 1, the *logfile* argument is ignored.

Usage

Use the **TSL_SetLogMode** function in a loader program to specify the severity of messages that are saved in the loader message log. You must run the **TSL_SetLogMode** function in the context of a loader session that was initialized by the **TSL_Init** function. The **TSL_SetLogMode** function changes the log mode that was specified by the **TSL_Init** function for all loading sessions.

Returns

- 0 = The log mode was set.
- An error if the function failed.

Examples

The following statement saves loader warning and error messages to a file:

```
EXECUTE FUNCTION TSL_SetLogMode('tsdata|raw_reads',1,2,'/tmp/messag.log');
```

Related tasks:

“Writing a loader program” on page 3-31

TSL_Shutdown procedure

The **TSL_Shutdown** procedure shuts down the loader application and releases all resources.

Syntax

```
TSL_Shutdown(  
            handle    lvarchar)
```

handle The table and column name combination returned by the **TSL_Init** function.

Usage

Use the **TSL_Shutdown** procedure when data loading is complete. The **TSL_Shutdown** procedure closes the global context and releases the associated resources that are used by the loader application. Any virtual tables that were created during the session are removed. Run the **TSL_SessionClose** function to close each database session before running the **TSL_Shutdown** procedure.

Example

```
EXECUTE PROCEDURE TSL_Shutdown('ts_data|raw_reads');
```

Related tasks:

“Writing a loader program” on page 3-31

TSPrevious function

The **TSPrevious** function records the supplied argument and returns the last argument it was passed.

Syntax

`TSPrevious(value int)`
returns int;

`TSPrevious(value smallfloat)`
returns smallfloat;

`TSPrevious(value double precision)`
returns double precision;

value The value to save.

Description

Use the **TSPrevious** function within the **Apply** function.

TSPrevious function is useful in comparing a value in a time series with the value immediately preceding it. The **TSPrevious** function can take parameters that are columns of a time series. Use the same parameter format that the **Apply** function accepts.

Returns

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

Example

See the example for the “TSCmp function” on page 7-90.

Related reference:

“Apply function” on page 7-18

“TSAddPrevious function” on page 7-90

“TSCmp function” on page 7-90

“TSDecay function” on page 7-122

“TSRunningAvg function” on page 7-147

“TSRunningSum function” on page 7-151

TSRollup function

The **TSRollup** function aggregates time series values by time for multiple rows in the table.

Syntax

```
TSRollup(  
    ts TimeSeries,  
    'agg_express' lvarchar)  
RETURNS TimeSeries;
```

agg_express

A comma-separated list of the following elements, in any order:

SQL aggregate operator = AVG, COUNT, MIN, MAX, SUM, or the FIRST and LAST operators. The FIRST operator returns a time series that contains the first element that was entered into the database for each timestamp.

The LAST operator returns the last element that is entered for each timestamp. You can include multiple operators. Each operator requires an argument that is the column of the input time series, which is specified by one of the following column identifiers:

\$colname

The *colname* is the name of the column to aggregate in the **TimeSeries** data type. For example, if the column name is **high**, the column identifier is **\$high**.

\$colnumber

The *colnumber* is the position of the column to aggregate in the **TimeSeries** data type. For example if the column number is 1, the column identifier is **\$1**.

\$bson_field_name

The *bson_field_name* is the name of a field in at least one BSON document in the BSON column in the **TimeSeries** data type. For example, if the field name is **v1**, the column identifier is **\$v1**. If the BSON field name is the same as another column in the **TimeSeries** data type, you must qualify the field name in one of the following ways:

- *\$colname.bson_field_name*

For example, if the BSON column name is **b_data** and the field name is **v1**, the column identifier is **\$b_data.v1**.

- *\$colnumber.bson_field_name*

For example, if the BSON column number is 1 and the field name is **v1**, the column identifier is **\$1.v1**.

You must cast the results of the **TSRollup** function on a BSON field to a **TimeSeries** data type that has the appropriate type of columns for the result of the expression.

start(*start_time*) = Optional. Specifies the start of the time range on which to operate. The value of *start_time* must be in the form of a DATETIME value. No value indicates that the time range starts at the origin of the time series. If you include the **start** argument more than once in the aggregate expression, only the last **start** argument is used.

end(*end_time*) = Optional. Specifies the end of the time range on which to operate. The value of *end_time* must be in the form of a DATETIME value. No value indicates that the time range ends at the last element in the time series. If you include the **end** argument more than once in the aggregate expression, only the last **end** argument is used.

ts The name of the **TimeSeries** data type or a function that returns a **TimeSeries** data type, such as **AggregateBy**.

Description

Use the **TSRollup** function to run one or more aggregate operators on multiple rows of time series data in a table.

You can specify a time range on which to run the aggregate expression by including the **start** and **end** arguments in the aggregate expression. Specifying the time range in the **TSRollup** function is faster than first clipping the data by running the **Clip** or **AggregateBy** function.

Returns

A **TimeSeries** data type that is the result of the expression or expressions.

Example: Sum of all electricity usage in a postal code

The following statement adds all the electricity usage values for each time stamp in the **ts_data** table in the **stores_demo** database for the customers that have a postal code of 94063:

```
SELECT TSRollup(raw_reads, "sum($value)")
  FROM ts_data, customer, customer_ts_data
 WHERE customer.zipcode = "94063"
    AND customer_ts_data.customer_num = customer.customer_num
    AND customer_ts_data.loc_es_i_id = ts_data.loc_es_i_id;
```

Example: Sum of daily electricity usage by postal code

Suppose that you have a table named **ts_table** that contains a user ID, the postal code of the user, and the electricity usage data for each customer, which is collected every 15 minutes and stored in a column named **value** in a time series named **ts_col**. The following query returns the total amounts of electricity used daily for each postal code for one month:

```
SELECT zipcode,
  TSRollup(
    AggregateBy('SUM($value)', 'callday', ts_col, 0,
      '2011-01-01 00:00:00.00000', '2011-01-31 23:45:00:00.00000'),
    'SUM($value)'
  )
FROM ts_table
GROUP BY zipcode;
```

The first argument to the **TSRollup** function is an **AggregateBy** function, which sums the electricity usage for each customer for each day of January 2011. The second argument is a SUM operator that sums the daily electricity usage by postal code.

The resulting table contains a row for each postal code. Each row has a time series that contains the sum of the electricity that is used by customers who live in that postal code for each day in January 2011.

Example: Sum of electricity usage for one hour

The following example adds all the electricity usage values for the specified time range of one hour in the **ts_data** table in the **stores_demo** database for the customers that have a postal code of 94063:

```
SELECT TSRollup(raw_reads, "SUM($value), start(2010-11-10 11:45:00),
  end(2010-11-10 12:45:00)")
  FROM ts_data, customer, customer_ts_data
 WHERE customer.zipcode = "94063"
    AND customer_ts_data.customer_num = customer.customer_num
    AND customer_ts_data.loc_es_i_id = ts_data.loc_es_i_id;
```

Example: Maximum value of a field in a BSON column

This example is based on the following row type and time series definition. The **TimeSeries** row type contains an **INTEGER** column that is named **v1** and the **BSON** column contains a field that is also named **v1**.

```
CREATE ROW TYPE rb(timestamp datetime year to fraction(5), data bson, v1 int);

INSERT INTO tj VALUES(1,'origin(2011-01-01 00:00:00.00000), calendar(ts_15min),
container(kontainer),threshold(0), regular,[[{"v1":99},20]]');
```

The following statement creates a **TimeSeries** data type to hold the results of the aggregation on the BSON field in an INTEGER column:

```
CREATE ROW TYPE outrow(timestamp datetime year to fraction(5), x int);
```

If a column and a BSON field have the same name, the column takes precedence. The following statement returns the maximum value from the **v1** INTEGER column:

```
SELECT TSRollup(tsdata, 'max($v1)') FROM tj;
```

```
TSRollup origin(2011-01-01 00:00:00.00000), calendar(ts_15min), container(),  
threshold(0), regular, [(20      )]
```

1 row(s) retrieved.

The following two equivalent statements return the maximum value from the **v1** field in the **data** BSON column, which is column 1 in the **TimeSeries** row type:

```
SELECT TSRollup(tsdata, 'max($data.v1)::timeseries(outrow) FROM tj;  
SELECT TSRollup(tsdata, 'max($1.v1)::timeseries(outrow) FROM tj;
```

```
TSRollup origin(2011-01-01 00:00:00.00000), calendar(ts_15min), container(),  
threshold(0), regular, [(99.000000000000)]
```

1 row(s) retrieved.

The aggregated time series that is returned has the **TimeSeries** data type **outrow**. If you do not cast the result to a row type that has appropriate columns for the results, the statement fails.

Related reference:

“AggregateBy function” on page 7-11

TSRowNameToList function

The **TSRowNameToList** function returns a list (collection of rows) containing one individual column from a time series column plus the non-time-series columns of a table. Null elements are not added to the list.

Syntax

```
TSRowNameToList(ts_row      row,  
                colname    lvarchar)  
returns list (row not null)
```

ts_row The time series to act on.

colname

The time series column to return.

Description

The **TSRowNameToList** function can only be used on rows with one **TimeSeries** column.

You must cast the return variable to match the names and types of the columns being returned exactly.

Returns

A list (collection of rows).

Example

The query returns a list of rows, each containing the **ID** and **high** columns.

```
select
  TSRowNameToList(d, 'high')::list(
    row(id integer, name lvarchar, high real) not null)
  from daily_stocks d;
```

Related reference:

“TSColNameToList function” on page 7-91

“TSColNumToList function” on page 7-92

“Transpose function” on page 7-86

“TSRowNumToList function”

“TSRowToList function” on page 7-147

“TSSetToList function” on page 7-153

TSRowNumToList function

The **TSRowNumToList** function returns a list (collection of rows) containing one individual column from a time series column plus the non-time-series columns of a table. Null elements are not added to the list.

Syntax

```
TSRowNumToList(ts_row      row,
               colnum      integer)
returns list (row not null)
```

ts_row The time series to act on.

colnum The number of the time series column to return.

Description

The **TSRowNumToList** function can only be used on rows with one **TimeSeries** column.

The column is specified by its number; column numbering starts at 1, with the first column following the time stamp column.

You must cast the return variable to match the names and types of the columns being returned exactly.

Returns

A list (collection of rows).

Example

The query returns a list of rows, each containing the **ID**, **name**, and **high** columns.

```
select
  TSRowNumToList(d, 1)::list(row
    (id integer, name lvarchar, high real) not null)
  from daily_stocks d;
```

Related reference:

“TSColNameToList function” on page 7-91

“TSColNumToList function” on page 7-92

“TSRowNameToList function” on page 7-145

“Transpose function” on page 7-86

“TSRowToList function”

“TSSetToList function” on page 7-153

TSRowToList function

The **TSRowToList** function returns a list (collection of rows) containing the individual columns from a time series column plus the non-time-series columns of a table. Null elements are not added to the list.

Syntax

```
TSRowToList(ts_row    row)  
returns list (row not null)
```

ts_row A row value that contains a time series as one of its columns.

Description

The **TSRowToList** function can only be used on rows with one **TimeSeries** column.

You must cast the return variable to match the names and types of the columns being returned exactly.

Returns

A list (collection of rows).

Example

The query returns a list of rows, each containing the following columns: **stock_id**, **stock_name**, **t**, **high**, **low**, **final**, **vol**.

```
select TSRowToList(d)::list(row(stock_id integer,  
                                stock_name lvarchar,  
                                t datetime year to fraction(5),  
                                high real,  
                                low real,  
                                final real,  
                                vol real) not null)  
from daily_stocks d;
```

Related reference:

“TSRowNameToList function” on page 7-145

“TSRowNumToList function” on page 7-146

“Transpose function” on page 7-86

“TSColNameToList function” on page 7-91

“TSColNumToList function” on page 7-92

“TSSetToList function” on page 7-153

TSRunningAvg function

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

Syntax

`TSRunningAvg(value double precision,
 num_values integer)
returns double precision;`

`TSRunningAvg(value real,
 num_values integer)
returns double precision;`

value The value to include in the running average.

num_values
 The number of values to include in the running average, *k*.

Description

Use the **TSRunningAvg** function within the **Apply** function.

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.

The **TSRunningAvg** function can take parameters that are columns of a time series. Use the same parameter format that the **Apply** function accepts.

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.

Returns

A SMALLFLOAT or DOUBLE PRECISION running average of the last *k* values.

Example

The example is based on the following row type:

```
create row type if not exists stock_bar (  
  timestamp datetime year to fraction(5),  
  high real,  
  low real,  
  final real,  
  vol real);
```

The example uses the following input data:

2011-01-03 00:00:00.00000	3	2	1	3
2011-01-04 00:00:00.00000	2	2	2	3
2011-01-05 00:00:00.00000	2	2	3	3
2011-01-06 00:00:00.00000	2	2		3

Notice the null value for the **final** column on 2011-01-06.

The SELECT query in the following example returns the closing price from the **final** column and the 4-day moving average from the stocks in the time series:

```
select stock_name, Apply('TSRunningAvg($final,4)',  
  '2011-01-03 00:00:00.00000'::datetime year to fraction(5),  
  '2011-01-06 00:00:00.00000'::datetime year to fraction(5),  
  stock_data::TimeSeries(stock_bar))::TimeSeries(one_real)  
from first_stocks;
```

The query returns the following result:

```
stock_name    IBM
(expression)  origin(2011-01-03 00:00:00.00000), calendar(daycal), container(),
              threshold(20), regular, [(1.000000000000), (1.500000000000), (2.
              000000000000), (2.000000000000)]
```

The fourth result is the same as the third result because the fourth value in the **final** column is null.

Related reference:

“Apply function” on page 7-18

“TSAddPrevious function” on page 7-90

“TSCmp function” on page 7-90

“TSDecay function” on page 7-122

“TSPrevious function” on page 7-141

“TSRunningSum function” on page 7-151

“TSRunningCor function”

“TSRunningMed function” on page 7-150

“TSRunningVar function” on page 7-152

TSRunningCor function

The **TSRunningCor** function computes the running correlation of two time series over a running window. The **TSRunningCor** function returns NULL if the variance of either input is zero or NULL over the window.

Syntax

```
TSRunningCor(value1    double precision,
             value2    double precision,
             num_values integer)
returns double precision;
```

```
TSRunningCor(value1    real,
             value2    real,
             num_values integer)
returns double precision;
```

value1 The column of the first time series to use to calculate the running correlation.

value2 The column of the second time series to use to calculate the running correlation.

num_values
The number of values to include in the running correlation, *k*.

Description

Use the **TSRunningCor** function within the **Apply** function.

The **TSRunningCor** function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

The first set of (*num_values* - 1) outputs result from shorter windows (the first output is derived from the first input time, the second output is derived from the first two input times, and so on). Null elements in the input also result in shortened windows.

The **TSRunningCor** function can take parameters that are columns of a time series. Use the same parameter format that the **Apply** function accepts.

Returns

A DOUBLE PRECISION running correlation of the last k values.

Example

This statement finds the running correlation between stock data for IBM and AA1 over a 20 element window. Again, the first 19 output elements are exceptions because they result from windows of fewer than 20 elements. The first is NULL because correlation is undefined for just one element.

```
select Apply('TSRunningCor($0.high, $1.high, 20)',
            ds1.stock_data::TimeSeries(stock_bar),
            ds1.stock_data::TimeSeries(stock_bar))::TimeSeries(one_real)
from daily_stocks ds1, daily_stocks ds2
where ds1.stock_name = 'IBM'
and ds2.stock_name = 'AA1';
```

Tip: When a start date is supplied to the **Apply** function, the first ($num_values - 1$) output elements are still formed from incomplete windows. The **Apply** function never looks at data before the specified start date.

Related reference:

“Apply function” on page 7-18

“TSRunningAvg function” on page 7-147

“TSRunningMed function”

“TSRunningSum function” on page 7-151

“TSRunningVar function” on page 7-152

TSRunningMed function

The **TSRunningMed** function computes the median of a time series over a running window. This function is useful only when used within the **Apply** function.

Syntax

```
TSRunningMed(value          double precision,
            num_values integer)
returns double precision;
```

```
TSRunningMed(value          real,
            num_values integer)
returns double precision;
```

value The first input value to use to calculate the running median. Typically, the name of a DOUBLE, FLOAT, or REAL column in your time series.

num_values

The number of values to include in the running median, k .

Description

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

The first ($num_values - 1$) outputs result from shorter windows (the first output is derived from the first input time, the second output is derived from the first two

input times, and so on). Null elements in the input also result in shortened windows.

Returns

A DOUBLE PRECISION running median of the last k values.

Example

This statement produces a time series from the running median over a 10-element window of the column **high** of *stock_data*. You can refer to the columns of a time series as **\$colname** or **\$colnumber**: for example, **\$high**, or **\$1**.

```
select stock_name, Apply('TSRunningMed($high, 10)',
                        stock_data::TimeSeries(stock_bar))::
    TimeSeries(one_real)
from daily_stocks;
```

Related reference:

“TSRunningCor function” on page 7-149

“Apply function” on page 7-18

“TSRunningAvg function” on page 7-147

“TSRunningSum function”

“TSRunningVar function” on page 7-152

TSRunningSum function

The **TSRunningSum** function computes a running sum over SMALLFLOAT or DOUBLE PRECISION values.

Syntax

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

```
TSRunningSum(value          double precision,
             num_values integer)
returns double precision;
```

value The input value to include in the running sum.

num_values
 The number of values to include in the running sum, k .

Description

A running sum is the sum of the last k values, where k is supplied by the user. If a value is NULL, the previous value is used.

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

This function is useful only when used within the **Apply** function.

Returns

A SMALLFLOAT or DOUBLE PRECISION running sum of the last k values.

Example

The following function calculates the *volume accumulation percentage*. The columns represented by **a** through **e** are: **high**, **low**, **close**, **volume**, and **number_of_days**, respectively:

```
create function VAP(a float, b float, c float, d float, e int) returns int;
return cast(100 * TSRunningSum(d * ((c - b) - (a - c)) /
(.0001 + a - b), e) / (.0001 + TSRunningSum(d, e)) as int);
end function;
```

Related reference:

“Apply function” on page 7-18

“TSAddPrevious function” on page 7-90

“TSCmp function” on page 7-90

“TSDecay function” on page 7-122

“TSPrevious function” on page 7-141

“TSRunningAvg function” on page 7-147

“TSRunningCor function” on page 7-149

“TSRunningMed function” on page 7-150

“TSRunningVar function”

TSRunningVar function

The **TSRunningVar** function computes the variance of a time series over a running window.

Syntax

```
TSRunningVar(value double precision,
             num_values integer)
returns double precision;
```

```
TSRunningVar(value real,
             num_values integer)
returns double precision;
```

value The first input value to use to calculate the running correlation.

num_values

The number of values to include in the running variance, *k*.

Description

Use the **TSRunningVar** function within the **Apply** function.

This function runs over a fixed number of elements, not over a fixed length of time; therefore, it might not be appropriate for irregular time series.

The first (*num_values* - 1) outputs are exceptions because they result from shorter windows (the first output is derived from the first input time, the second output is derived from the first two input times, and so on). Null elements in the input also result in shortened windows.

The **TSRunningVar** function can take parameters that are columns of a time series. Use the same parameter format that the **Apply** function accepts.

Returns

A DOUBLE PRECISION running variance of the last k values.

Example

This statement produces a time series with the same length and calendar as **stock_data** but with one data column other than the time stamp. Element n of the output is the variance of column 1 of **stock_bar** elements $n-19, n-18, \dots n$. The first 19 elements of the output are a bit different: the first element is NULL, because variance is undefined for a series of 1. The second output element is the variance of the first two input elements, and so on.

If element i of **stock_data** is NULL, or if column 1 of element i of **stock_data** is NULL, output elements $i, i + 1, \dots i + 19$, are variances of just 19 numbers (assuming that there are no other null values in the input window).

```
select stock_name, Apply('TSRunningVar($0.high, 20)',
                        stock_data::TimeSeries(stock_bar))::
                        TimeSeries(one_real)
from daily_stocks;
```

Related reference:

“Apply function” on page 7-18

“TSRunningAvg function” on page 7-147

“TSRunningCor function” on page 7-149

“TSRunningMed function” on page 7-150

“TSRunningSum function” on page 7-151

TSSetToList function

The **TSSetToList** function takes a **TimeSeries** column and returns a list (collection of rows) containing all the elements in the time series. Null elements are not added to the list.

Syntax

```
TSSetToList(ts      TimeSeries)
returns list (row not null)
```

ts The time series to act on.

Description

Because this aggregate function can return rows of any type, the return value must be explicitly cast at runtime.

Returns

A list (collection of rows).

Example

The following query collects all the elements in all the time series in the **stock_data** column into a list and then selects out the **high** column from each element.

```
select high from table((select
  TsSetToList(stock_data)::list(stock_bar
  not null) from daily_stocks));
```

Related reference:

"TSColNameToList function" on page 7-91

"TSColNumToList function" on page 7-92

"TSRowNameToList function" on page 7-145

"TSRowNumToList function" on page 7-146

"Transpose function" on page 7-86

"TSRowToList function" on page 7-147

TSToXML function

The **TSToXML** function returns an XML representation of a time series.

Syntax

```
TSToXML(doctype    lvarchar,
        id         lvarchar,
        ts         timeseries,
        output_max integer default 0)
returns lvarchar;
```

```
TSToXML(doctype    lvarchar,
        id         lvarchar,
        ts         timeseries)
returns lvarchar;
```

doctype

The name of the topmost XML element.

id

The primary key value in the time series table that uniquely identifies the time series.

ts

The name of the **TimeSeries** subtype.

output_max

The maximum size, in bytes, of the XML output. If the parameter is absent, the default value is 32 768. The following table describes the results for each possible value of the *output_max* parameter.

Value	Result
no value	32 768 bytes
negative integer	$2^{32}-1$ bytes
1 through 4096	4096 bytes
4096 through $2^{32}-1$	the specified number of bytes

Description

Use the **TSToXML** function to provide a standard representation for information exchange in XML format for small amounts of data.

The top-level tag in the XML output is the first argument to the **TSToXML** function.

The *id* tag must uniquely identify the time series and refer the XML output to the row on which it is based.

The *AllData* tag indicates whether all the data was returned or the data was truncated because it exceeded the size set by the *output_max* parameter.

The remaining XML tags represent the TimeSeries subtype and its columns, including the time stamp.

The special characters <, >, &, ', and " are replaced by their XML predefined entities.

Returns

The specified time series in XML format, up to the size set by the *output_max* parameter. The AllData tag indicates whether all the data was returned (1) or whether the data was truncated (0).

Example

The following query selects the time series data for one hour by using the **Clip** function from the **TimeSeries** subtype named **actual** to return in XML format:

```
SELECT TSToXML('meterdata', esi_id,
              Clip(actual, '2010-09-08 12:00:00'::datetime year to second,
                      '2010-09-08 13:00:00'::datetime year to second ) )
FROM ts_data
WHERE esi_id = '2250561334';
```

The following XML data is returned:

```
<meterdata>
  <id>2250561334</id>
  <AllData>1</AllData>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>0.9170000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>0.4610000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>4.1570000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>6.3280000000</value>
  </meter_data>
  <meter_data>
    <tstamp>2010-09-08 12:15:00.00000</tstamp>
    <value>2.6690000000</value>
  </meter_data>
</meterdata>
```

The name of the TimeSeries subtype is meter_data and its columns are tstamp and value.

The value of 1 in the AllData tag indicates that for this example, all data was returned.

Related concepts:

“Planning for accessing time series data” on page 1-24

Related reference:

“Time series routines that run in parallel” on page 7-7

Unary arithmetic functions

The standard unary functions **Abs**, **Acos**, **Asin**, **Atan**, **Cos**, **Exp**, **Logn**, **Negate**, **Positive**, **Round**, **Sin**, **Sqrt**, and **Tan** are extended to operate on time series.

Syntax

```
Function(ts TimeSeries)  
returns TimeSeries;
```

ts The time series to act on.

Description

The resulting time series has the same regularity, calendar, and sequence of time stamps as the input time series. It is derived by applying the function to each element of the input time series.

If there is a variant of the function that operates directly on the input element type, then that variant is applied to each element. Otherwise, the function is applied to each non-time stamp column of the input time series.

Returns

The same type of time series as the input; unless it is cast, then it returns the type of time series to which it is cast.

Example

The following query converts the daily stock price and volume data into log space:

```
create table log_stock (stock_id int, data TimeSeries(stock_bar));  
insert into log_stock  
    select stock_id, Logn(stock_data)  
    from daily_stocks;
```

Related reference:

- "Abs function" on page 7-11
- "Acos function" on page 7-11
- "ApplyUnaryTsOp function" on page 7-26
- "Asin function" on page 7-27
- "Atan function" on page 7-27
- "Binary arithmetic functions" on page 7-27
- "Cos function" on page 7-38
- "Exp function" on page 7-48
- "Logn function" on page 7-74
- "Negate function" on page 7-75
- "Positive function" on page 7-77
- "Round function" on page 7-84
- "Sin function" on page 7-85
- "Sqrt function" on page 7-85
- "Tan function" on page 7-86
- "Apply function" on page 7-18

Union function

The **Union** function performs a union of multiple time series, either over the entire length of each time series or over a clipped portion of each time series.

Syntax

```
Union(ts TimeSeries,...)  
returns TimeSeries;
```

```
Union(set_ts set(TimeSeries))  
returns TimeSeries;
```

```
Union(begin_stamp datetime year to fraction(5),  
      end_stamp   datetime year to fraction(5),  
      ts          TimeSeries,...)  
returns TimeSeries;
```

```
Union(begin_stamp datetime year to fraction(5),  
      end_stamp   datetime year to fraction(5),  
      set_ts      set(TimeSeries))  
returns TimeSeries;
```

ts The time series that form the union. **Union** can take from two to eight time series arguments.

set_ts A set of time series.

begin_stamp
 The begin point of the clip.

end_stamp
 The end point of the clip.

Description

The second and fourth forms of the function perform a union of a set of time series. The resulting time series has one DATETIME YEAR TO FRACTION(5) column, followed by each column in each time series, in order. When using the second or fourth form, it is important to ensure that the order of the time series in the set is deterministic so that the elements remain in the correct order.

Since the type of the resulting time series is different from that of the input time series, the result of the union must be cast.

Union can be thought of as an outer join on the time stamp.

In a union, the resulting time series has a calendar that is the combination of the calendars of the input time series with the OR operator. The resulting calendar is stored in the **CalendarTable** table. The name of the resulting calendar is a string containing the names of the calendars of the input time series, separated by a vertical bar (|). For example, if two time series are combined, and **mycal** and **yourcal** are the names of their corresponding calendars, the resulting calendar is named **mycal|yourcal**. If all the time series have the same calendar, then **Union** does not create a new calendar.

For a regular time series, if a time series does not have a valid element at a timepoint of the resulting calendar, the value for that time series element is NULL.

To be certain of the order of the columns in the resultant time series when using **Union** over a set, use the ORDER BY clause.

For the purposes of **Union**, the value at a given timepoint is that of the most recent valid element. For regular time series, this is the value corresponding to the current interval, which can be NULL; it is not necessarily the most recent non-null value. For irregular time series, this condition never occurs since irregular time series do not have null intervals.

For example, consider the union of two irregular time series, one containing bid prices for a certain stock, and one containing asking prices. The union of the two time series contains bid and ask values for each timepoint at which a price was either bid or asked. Now consider a timepoint at which a bid was made but no price was asked. The union at that timepoint contains the bid price offered at that timepoint, along with the most recent asking price.

If an intersection involves one or more regular time series, the resulting time series starts at the latest of the start points of the input time series and ends at the earliest of the end points of the regular input time series. If all the input time series are irregular, the resulting irregular time series starts at the latest of the start points of the input time series and ends at the latest of the end points. If a union involves one or more time series, the resulting time series starts at the first of the start points of the input time series and ends at the latest of the end points of the input time series. Other than this difference in start and end points, and of the resulting calendar, there is no difference between union and intersection involving time series.

Apply also combines multiple time series into a single time series. Therefore, using **Union** within **Apply** is often unnecessary.

Returns

The time series that results from the union.

Example

The following query constructs the union of time series for two different stocks:

```
select Union(s1.stock_data,
            s2.stock_data)::TimeSeries(stock_bar_union)
from daily_stocks s1, daily_stocks s2
where s1.stock_name = 'IBM' and s2.stock_name = 'HWP';
```

The following example finds the union of two time series and returns data only for time stamps between 2011-01-03 and 2011-01-05:

```
select Union('2011-01-03 00:00:00.00000'
            ::datetime year to fraction(5),
            '2011-01-05 00:00:00.00000'
            ::datetime year to fraction(5),
            s1.stock_data,
            s2.stock_data)::TimeSeries(stock_bar_union)
from daily_stocks s1, daily_stocks s2
where s1.stock_name = 'IBM' and s2.stock_name = 'HWP';
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“Apply function” on page 7-18

“Intersect function” on page 7-71

UpdElem function

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

Syntax

```
UpdElem(ts           TimeSeries,  
        row_value  row,  
        flags       integer default 0)  
returns TimeSeries;
```

ts The time series to update.

row_value
 The new row data.

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

The element must be a row type of the correct type for the time series, beginning with a time stamp. If there is no element in the time series with the given time stamp, an error is raised.

Hidden elements cannot be updated.

The API equivalent of **UpdElem** is **ts_upd_elem()**.

Returns

A new time series containing the updated element.

Example

The following example updates a single element in an irregular time series:

```
update activity_stocks  
set activity_data = UpdElem(activity_data,  
    row('2011-01-04 12:58:09.12345', 6.75, 2000,  
        2, 007, 3, 1)::stock_trade)  
where stock_id = 600;
```

Related tasks:

“Mapping time series data types” on page 8-4

Related reference:

“DelElem function” on page 7-43

“GetElem function” on page 7-52

“InsElem function” on page 7-69

“PutElem function” on page 7-77

“UpdSet function” on page 7-160

“The ts_upd_elem() function” on page 9-54

UpdMetaData function

The **UpdMetaData** function updates the user-defined metadata in the specified time series.

Syntax

```
create function UpdMetaData(ts      TimeSeries,  
                           metadata TimeSeriesMeta)  
returns TimeSeries;
```

ts The time series for which to update metadata.

metadata

The metadata to be added to the time series. Can be NULL.

Description

This function adds the supplied user-defined metadata to the specified time series. If the *metadata* argument is NULL, then the time series is updated to contain no metadata. If it is not NULL, then the user-defined metadata is stored in the time series.

Returns

The time series updated to contain the supplied metadata, or the time series with metadata removed, if the *metadata* argument is NULL.

Related tasks:

“Creating a time series with metadata” on page 3-23

Related reference:

“GetMetaData function” on page 7-59

“GetMetaTypeName function” on page 7-59

“TSCreate function” on page 7-116

“TSCreateIrr function” on page 7-118

“The ts_create_with_metadata() function” on page 9-18

“The ts_get_metadata() function” on page 9-31

“The ts_update_metadata() function” on page 9-54

UpdSet function

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

Syntax

```
UpdSet(ts      TimeSeries,  
      set_ts multiset,  
      flags integer default 0)  
returns TimeSeries;
```

ts The time series to update.

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

flags Valid values for the *flags* argument are described in “The flags argument values” on page 7-9. The default is 0.

Description

The rows in *set_ts* must be of the correct type for the time series, beginning with a time stamp; otherwise, an error is raised. If the time stamp of any element does not correspond to an element already in the time series, an error is raised, and the entire update is void.

Hidden elements cannot be updated.

Returns

The updated time series.

Example

The following example updates elements in a time series:

```
update activity_stocks
set activity_data = (select UpdSet(activity_data, set_data)
                    from activity_load_tab where stock_id = 600)
where stock_id = 600;
```

Related reference:

“DelClip function” on page 7-42

“DelTrim function” on page 7-45

“InsSet function” on page 7-70

“PutSet function” on page 7-80

“UpdElem function” on page 7-159

WithinC and WithinR functions

The **WithinC** and **WithinR** functions perform calendar-based queries, converting among time units and doing the calendar math to extract periods of interest from a time series value.

Syntax

```
WithinC(ts           TimeSeries,
        tstamp      datetime year to fraction(5),
        interval    lvarchar,
        num_intervals integer,
        direction   lvarchar)
returns TimeSeries;
```

```
WithinR(ts           TimeSeries,
        tstamp      datetime year to fraction(5),
        interval    lvarchar,
        num_intervals integer,
        direction   lvarchar)
returns TimeSeries;
```

ts The source time series.

tstamp The timepoint of interest.

interval

The name of an interval: second, minute, hour, day, week, month, or year.

num_intervals

The number of intervals to include in the output.

direction

The direction in time to include intervals. Possible values are:

- FUTURE, or F, or f
- PAST, or P, or p

Description

Every time series has a calendar that describes the active and inactive periods for the time series and how often they occur. A regular time series records one value for every active period of the calendar. Calendars can have periods of a second, a minute, an hour, a day, a week, a month, or a year. Given a time series, you might want to pose calendar-based queries on it, such as, “Show me all the values in this daily series for six years beginning on May 31, 2004,” or “Show me the values in this hourly series for the week including December 27, 2010.”

The **Within** functions are the primary mechanism for queries of this form. They convert among time units and do the calendar math to extract periods of interest from a time series value. There are two fundamental varieties of **Within** queries: calibrated (**WithinC**) and relative (**WithinR**).

WithinC, or within calibrated, takes a time stamp and finds the period that includes that time. Weeks have natural boundaries (Sunday through Saturday), as do years (January 1 through December 31), months (first day of the month through the last), 24-hour days, 60-minute hours, and 60-second minutes. **WithinC** allows you to specify a time stamp and find the corresponding period (or periods) that include it.

For example, July 2, 2010, fell on a Friday. Given an hourly time series, **WithinC** allows you to ask for all the hourly values in the series beginning on Sunday morning at midnight of that week and ending on Saturday night at 11:59:59. Of course, the calendar might not mark all of those hours as active; only data from active periods is returned by the **Within** functions.

WithinR, or within relative, takes a time stamp from the user and finds the period beginning or ending at that time. For example, given a weekly time series, **WithinR** can extract all the weekly values for two years beginning on June 3, 2008. **WithinR** is able to convert weeks to years and count forward or backward from the supplied date for the number of intervals requested. Relative means that you supply the exact time stamp of interest as the begin point or end point of the range.

WithinR behaves slightly differently for irregular than for regular time series. With regular time series, the time stamp argument is always mapped to a timepoint in accordance with the argument time series calendar interval. Relative offsetting is then performed starting with that point.

In irregular time series, the corresponding calendar interval does not indicate where time series elements are, and therefore offsetting begins at exactly the time stamp specified. Also, since irregular elements can appear at any point within the calendar time interval, **WithinR** returns elements with time stamps up to the last instant of the argument interval.

For example, assume an irregular time series with a daily calendar turning on all weekdays. The following function returns elements in the following interval (excluding the endpoint):

```
WithinR(stock_data, '2010-07-11 07:37:18', 'day', 3, 'future')  
[2010-07-11 07:37:18, 2010-07-14 07:37:18]
```

In a regular time series, the interval is as follows, since each timepoint corresponds to the period containing the entire following day:

```
[2010-07-11 00:00:00, 2010-07-13 00:00:00]
```

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

The supplied interval name is not required to be the same as the interval stored by the time series calendar, but it cannot be smaller than that interval. For example, given an hourly time series, the **Within** functions can count forward or backward for hours, days, weeks, months, or years, but not for minutes or seconds.

The direction argument indicates which periods other than the period containing the time stamp should be included; if there is only one period, the direction argument is moot.

For both **WithinC** and **WithinR**, the requested timepoint is included in the output.

Returns

A new time series with the same calendar as the original, but containing only the requested values.

Example

The following query retrieves data from the calendar week that includes Friday, January 4, 2011:

```
select WithinC(stock_data, '2011-01-04 00:00:00.00000',
               'week', 1, 'PAST')
   from daily_stocks
  where stock_name = 'IBM';
```

The query returns the following results:

(expression)

```
origin(2011-01-03 00:00:00.00000),calendar(daycal),
container(),threshold(20),regular,[(356.0000000000,310.0000000000,340.0000000000,
999.0000000000),(156.000000
0000,110.0000000000,140.0000000000,111.0000000000), NULL,
(99.0000000000,54.000 00000000,66.0000000000,
888.0000000000)]
```

The following query returns two weeks' worth of stock trades starting on January 4, 2011, at 9:30 a.m.:

```
select WithinR(activity_data, '2011-01-04 09:30:00.00000', 'week', 2, 'future')
   from activity_stocks
  where stock_id = 600;
```

The following query returns the preceding three months' worth of stock trades:

```
select WithinR(activity_data, '2011-02-01 00:00:00.00000',
               'month', 3, 'past')
   from activity_stocks
  where stock_id = 600;
```

Related reference:

“Time series routines that run in parallel” on page 7-7

“CalendarPattern data type” on page 2-1

“Clip function” on page 7-31

Chapter 8. Time series Java class library

You can use the time series Java class library to create and manage a time series from within Java applications or applets.

The time series Java class library uses the JDBC 2.0 specification for supporting user-defined data types in Java.

When you write a Java application for time series data, you use the IBM Informix JDBC Driver to connect to an IBM Informix database, as shown in the following figure. See your *IBM Informix JDBC Driver Programmer's Guide* for information about how to set up your Java programs to connect to Informix databases.

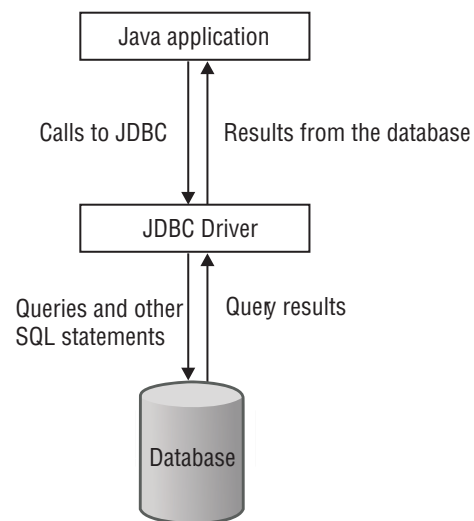


Figure 8-1. Runtime architecture for Java programs that connect to a database

The Java application makes calls to the JDBC driver, which sends queries and other SQL statements to the IBM Informix database. The database sends query results to the JDBC driver, which sends them on to the Java application.

You can also use the time series Java classes in Java applets and servlets, as shown in the following figures.

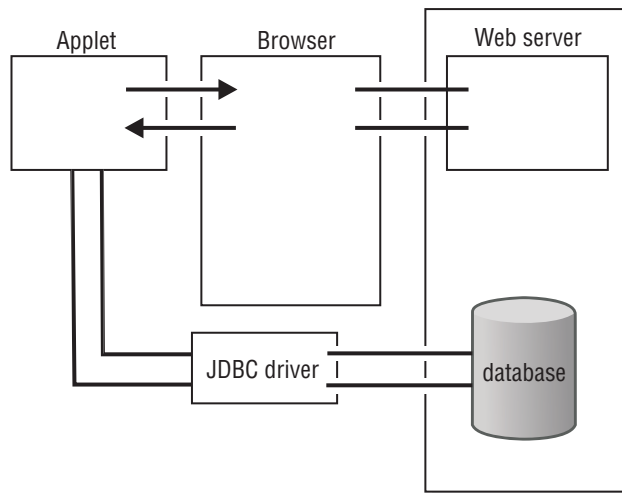


Figure 8-2. Runtime architecture for a Java applet

The database server is connected to the JDBC driver, which is connected to the applet. The applet is also connected to a browser, which is connected to a web server that communicates with the database.

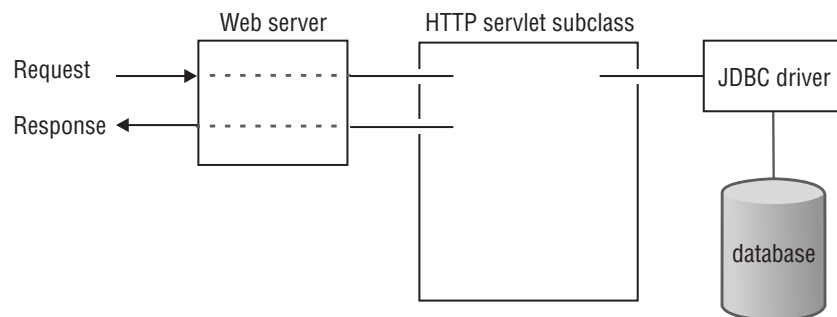


Figure 8-3. Runtime architecture for a Java servlet

A request from an application goes through a web server, an HTTP servlet subclass, and the JDBC driver to the database. The database sends responses back along the same path.

Summary of time series classes

The Java class library contains classes that you use to complete all necessary tasks for creating and managing time series. You can create time series objects with the Builder classes that are provided for each class. The following table lists the Java class that you need for each of the tasks in creating and managing a time series.


Table 8-1. Time series tasks and the corresponding Java classes

Tasks	Java classes
Create, query, and manage a custom type map for time series data types	TimeSeriesTypeMap TimeSeriesTypeMap.Builder
Create, query, and manage calendar patterns	IfmxCalendarPattern IfmxCalendarPattern.Builder

Table 8-1. Time series tasks and the corresponding Java classes (continued)

Tasks	Java classes
Create, query, and manage calendars	IfmxCalendar IfmxCalendar.Builder
Create, query, and manage containers	TimeSeriesContainer TimeSeriesContainer.Builder
Create, query, and manage TimeSeries row types	TimeSeriesRowType TimeSeriesRowType.Builder
Instantiate time series	IfmxTimeSeries IfmxTimeSeries.Builder
Query, insert, update, or delete time series data	IfmxTimeSeries

Related concepts:

-  [Connect to the database \(JDBC Driver Guide\)](#)
- [“Planning for accessing time series data” on page 1-24](#)
- [“Planning for creating a time series” on page 1-19](#)

Java class files and sample programs

The time series Java class .jar file, Javadoc, and sample programs are included with the database server.

The Java class file, `IfmxTimeSeries.jar`, is in the `$INFORMIXDIR/extend/TimeSeries.version/java/lib` directory.

The Javadoc files are in the `$INFORMIXDIR/extend/TimeSeries.version/java/doc` directory.

To access the sample programs, run the `jar -xvf IfmxTimeSeries.jar` command to expand the `IfmxTimeSeries.jar` file. The examples are in the resulting `com/informix/docExamples` directory. The examples include the SQL scripts `setup.sql` and `clean.sql` to set up data for the examples and clean it up afterward.

The `TimeSeriesExample.java` file provides a comprehensive example of using Builders to create, load, and query time series.

Preparing the server for Java classes

You must prepare the database server so that you can include the time series Java classes in your application by setting the `CLASSPATH` environment variable to the location of the Java class files.

Your Java program must use the following versions of Java and JDBC:

- IBM Java Developer Kit 1.6 or later
- IBM Informix JDBC Driver, Version 4.10.JC4 or later

To prepare the database server for using Java classes:

Set the CLASSPATH environment variable to the location of the .jar file or its contents. For example, if you leave the .jar file in the default location, use the following setting:

```
CLASSPATH=$INFORMIXDIR/extend/TimeSeries.version  
/java/lib/IfmxTimeSeries.jar:$CLASSPATH;export CLASSPATH
```

If you move the .jar file or expand it, set the CLASSPATH environment variable to the appropriate directory.

Mapping time series data types

To enable your Java program to retrieve or send a **Calendar**, **CalendarPattern**, or **TimeSeries** data type to or from an IBM Informix database, you must create a custom type map.

Create a type map with one of the following methods:

- Create an entry for each time series data type in the type map for each database connection. Entries are valid only for the current connection. The following example makes an entry in the type map of a connection for handling **TimeSeries** data:

```
java.util.Map customTypeMap;  
customTypeMap = conn.getTypeMap();  
  
customTypeMap.put("timeseries(stock_bar)",  
    Class.forName("com.informix.timeseries.IfmxTimeSeries"));
```

The readSQL method extracts the time series data from the database result set. There must be an entry in the type map for every **TimeSeries** type that your program uses. You must also add entries for the **Calendar** and **CalendarPattern** data types if your program selects those types from the database, for example:

```
customTypeMap.put("calendarpattern",  
    Class.forName("com.informix.timeseries.IfmxCalendarPattern"));  
  
customTypeMap.put("calendar",  
    Class.forName("com.informix.timeseries.IfmxCalendar"));
```

- Create a custom map for the time series data types and reference the map in your application. Use the Builder pattern from the TimeSeriesTypeMap class to create a PatternClassMap for the **TimeSeries**, **Calendar**, and **CalendarPattern** data types. For example, the following map provides case-insensitive access to time series data types:

```
Map<String,Class<?>> typeMap = TimeSeriesTypeMap.builder().caseSensitive(false)  
    .build();  
connection.setTypeMap(typeMap);
```

You can include the type map in your getObject method call, for example:

```
ts = (IfmxTimeSeries)rSet.getObject(1, typeMap);
```

Related reference:

“UpdElem function” on page 7-159

Querying time series data with the IfmxTimeSeries object

You can use a SELECT statement your Java program to retrieve **TimeSeries** data with the **IfmxTimeSeries** object.

For example, the following code selected time series data into an **IfmxTimeSeries** object:

```
String sqlCmd = "SELECT ts FROM test WHERE id = 1";
PreparedStatement pstmt = conn.prepareStatement(sqlCmd);
ResultSet rSet = pstmt.executeQuery();

com.informix.timeseries.IfmxTimeSeries ts;

rSet.next()
ts = (IfmxTimeSeries)rSet.getObject(1);
```

In this example, **rSet** is a valid **java.sql.ResultSet** object. After running the **SELECT** statement, the **getObject** method is used to put the time series data into the variable **ts** (an **IfmxTimeSeries** object). The **TimeSeries** type is at column 1 in the result set. The example assumes that an entry has been made into the **conn** object type map for the **TimeSeries** column, **ts**.

Because the **IfmxTimeSeries** class implements the JDBC **ResultSet** interface, you can treat an **IfmxTimeSeries** object as if it is an ordinary result set. For example, you can use the **next** method to iterate through the elements of the time series, as shown in the following example:

```
ts.beforeFirst();
while (ts.next())
{
    java.sql.Timestamp tStamp = ts.getTimestamp(1);
    int col1 = ts.getInt(2);
    int col2 = ts.getInt(3);
}
```

The example shows that you use the **beforeFirst** method to position the time series cursor before the beginning of the time series and then the **next** method to iterate through the elements. While looping through the elements, the program uses the **getTimestamp** method to extract the time stamp into the variable **tStamp** and the **getInt** method to extract the first column of data into **col1** and the second column into **col2**. The columns of a time series element are numbered, starting with the time stamp column as column 1.

When you clip a time series, the resultant time series is read-only.

The sample programs demonstrate how to retrieve and update time series data.

Obtaining the time series Java class version

If requested by IBM Software Support, you can obtain the version stamp for the time series Java classes.

To retrieve the version stamp for the time series Java classes, use one of the following methods:

- Run the following command from the command line:

```
java com.informix.timeseries.TimeSeriesBuildInformation
```
- Construct an instance of **com.informix.timeseries.TimeSeriesBuildInformation** from within an application:

```
import com.informix.TimeSeries.TimeSeriesBuildInformation;
TimeSeriesBuildInformation buildInfo = new TimeSeriesBuildInformation();
String version = buildInfo.getVersion();
```

Chapter 9. Time series API routines

The time series application programming interface routines allow application programmers to directly access a time series datum.

You can scan and update a set of time series elements, or a single element referenced by either a time stamp or a time series index. These routines can be used in client programs that fetch time series data in binary mode or in registered server or client routines that have an argument or return value of a time series type.

If there is a failure, these routines raise an error condition and do not return a value.

On UNIX, these routines exist in two archives: `tsfeapi.a` and `tsbeapi.a`. To use any of these routines, include the `tsbeapi.a` file when producing a shared library for the server, or use `tsfeapi.a` when compiling a client application.

The `tseries.h` header file must be included when there are calls to any of the time series interface routines.

On UNIX, `tsfeapi.a`, `tsbeapi.a`, and `tseries.h` are all in the `lib` directory in the database server installation.

On Windows, these routines exist in two archives: `tsfeapi.lib` and `tsbeapi.lib`. To use any of these routines, include the `tsbeapi.lib` file when producing a shared library for the server, or use `tsfeapi.lib` when compiling a client application.

The `tseries.h` header file must be included when there are calls to any of the time series interface routines.

On Windows, `tsfeapi.lib`, `tsbeapi.lib`, and `tseries.h` are all in the `lib` directory in the database server installation.

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

Related concepts:

“Planning for accessing time series data” on page 1-24

Differences in using functions on the server and on the client

There are significant differences between using the client version of the time series API (`tsfeapi`) and the server version of the time series API (`tsbeapi`).

The client and server interfaces do not behave in exactly the same way when updating a time series. This is because `tsbeapi` operates directly on a time series, whereas `tsfeapi` operates on a private copy of a time series. This means that updates through `tsbeapi` are always reflected in the database, while updates through `tsfeapi` are not. For changes made by `tsfeapi` to become permanent, the client must write the updated time series back into the database.

Another difference between the two interfaces is in how time series are passed as arguments to the **mi_exec_prepare_statement()** function. On the server, no special steps are required: a time series can be passed as is to this function. However, on the client you must make a copy of the time series with **ts_copy** and pass the copy as an argument to the **mi_exec_prepare_statement()** function.

There can be a difference in efficiency between the client and the server APIs. Functions built to run on the server take advantage of the underlying paging mechanism. For instance, if a function must scan across 20 years worth of data, the **tsbeapi** interface keeps only a few pages in memory at any one time. For a client program to do this, the entire time series must be brought over to the client and kept in memory. Depending on the size of the time series and the memory available, this might cause swapping problems on the client. However, performance depends on many factors, including the pattern of usage and distribution of your hardware. If hundreds of users are performing complex analysis in the server, it can overwhelm the server, whereas if each client does their portion of the work, the load can be better balanced.

Data structures for the time series API

The time series API uses four data structures.

The **ts_timeseries** structure

A **ts_timeseries** structure is the header for a time series. It can be stored in and retrieved from a time series column of a table.

The **ts_timeseries** structure contains pointers, so it cannot be copied directly. Use the **ts_copy()** function to copy a time series.

When you pass a binary time series value, *ts*, of type **ts_timeseries**, to **mi_exec_prepared_statement()**, you must pass *ts* in the values array and 0 in the lengths array.

The **ts_tscan** structure

A **ts_tscan** structure allows you to look at no more than two time series elements at a time. It maintains a current scan position in the time series and has two element buffers for creating elements. An element fetched from a scan is overwritten after two **ts_next()** calls.

A **ts_tscan** structure is created with the **ts_begin_scan()** function and destroyed with the **ts_end_scan()** procedure.

The **ts_tsdesc** structure

A **ts_tsdesc** structure contains a time series (**ts_timeseries**) and data structures for working with it. Among other things, **ts_tsdesc** tracks the current element and holds two element buffers for creating two elements.

Important: The two element buffers are shared by the element-fetching functions. An element that is fetched is overwritten two fetch calls later. Elements fetched by functions like **ts_elem()** should not be explicitly freed. They are freed when the **ts_tsdesc** is closed.

If you must look at more than two elements at a time, open a scan or use the **ts_make_elem()** or **ts_make_elem_with_buf()** routines to make a copy of one of your elements.

A **ts_tsdesc** structure is created by the **ts_open()** function and destroyed by the **ts_close()** procedure. It is used by most of the time series API routines.

The **ts_tselem** structure

A **ts_tselem** structure is a pointer to one element (row) of a time series.

When you use **ts_tselem** with a regular time series, the time stamp column in the element is left as NULL, allowing you to avoid the expense of computing the time stamp if it is not required. The time stamp is computed on demand in the **ts_get_col_by_name()**, **ts_get_col_by_number()**, and **ts_get_all_cols()** routines. For irregular time series, the time stamp column is never NULL.

You can convert a **ts_tselem** structure to and from an **MI_ROW** structure with the **ts_row_to_elem()** and **ts_elem_to_row()** routines.

If the element was created by the **ts_make_elem()** or **ts_make_elem_with_buf()** procedure, you must use the **ts_free_elem()** procedure to free the memory allocated for a **ts_tselem** structure.

Time series API routines sorted by task

Time series API routines are sorted into logical areas based on the type of task.

The following table shows the time series interface routines listed by task type. An uppercase routine name, such as **TS_ELEM_NULL**, denotes a macro.

Table 9-1. Time series API routines sorted by task

Task type	Description
Open and close a time series	Open a time series: “The ts_open() function” on page 9-44
	Close a time series: “The ts_close() function” on page 9-12
	Return a pointer to the time series associated with the specified time series descriptor: “The ts_get_ts() function” on page 9-33
Create and copy a time series	Create a time series: “The ts_create() function” on page 9-17
	Create a time series with metadata: “The ts_create_with_metadata() function” on page 9-18
	Copy a time series: “The ts_copy() function” on page 9-16
	Free all memory associated with a time series created with ts_copy() or ts_create() : “The ts_free() procedure” on page 9-26
	Copy all elements of one time series into another: “The ts_put_ts() function” on page 9-49

Table 9-1. Time series API routines sorted by task (continued)

Task type	Description
Scan a time series	Start a scan: “The ts_begin_scan() function” on page 9-7
	Retrieve the next element from a scan: “The ts_next() function” on page 9-41
	End a scan: “The ts_end_scan() procedure” on page 9-25
	Find the time stamp of the last element retrieved from a scan: “The ts_current_timestamp() function” on page 9-21
	Return the offset for the last element returned by ts_next() (regular time series): “The ts_current_offset() function” on page 9-20
Make elements visible or invisible to a scan	Make an element invisible: “The ts_hide_elem() function” on page 9-34
	Make an element visible: “The ts_reveal_elem() function” on page 9-50
Select individual elements from a time series	Get the element closest to a specified time stamp: “The ts_closest_elem() function” on page 9-13
	Get the element associated with a specified time stamp: “The ts_elem() function” on page 9-22
	Get the element at a specified position: “The ts_nth_elem() function” on page 9-43
	Get the first element: “The ts_first_elem() function” on page 9-25
	Get the last element: “The ts_last_elem() function” on page 9-37
	Find the next element after a specified time stamp: “The ts_next_valid() function” on page 9-42
	Find the last element before a specified time stamp: “The ts_previous_valid() function” on page 9-45
	Find the last element at or before a specified time stamp: “The ts_last_valid() function” on page 9-38
Update a time series	Insert an element: “The ts_ins_elem() function” on page 9-36
	Update an element: “The ts_upd_elem() function” on page 9-54
	Delete an element: “The ts_del_elem() function” on page 9-22
	Put an element in a place specified by a time stamp: “The ts_put_elem() function” on page 9-46 and “The ts_put_elem_no_dups() function” on page 9-47
	Append an element (regular time series): “The ts_put_last_elem() function” on page 9-48
	Put an element in a place specified by an offset (regular time series): “The ts_put_nth_elem() function” on page 9-48
Modify metadata	Update metadata: “The ts_update_metadata() function” on page 9-54
Convert between an index and a time stamp	Convert time stamp to index (regular time series): “The ts_index() function” on page 9-35
	Convert index to time stamp (regular time series): “The ts_time() function” on page 9-51

Table 9-1. Time series API routines sorted by task (continued)

Task type	Description
Transform an element	Create an element from an array of values and nulls: “The <code>ts_make_elem()</code> function” on page 9-38 and “The <code>ts_make_elem_with_buf()</code> function” on page 9-39
	Convert an <code>MI_ROW</code> value to an element: “The <code>ts_row_to_elem()</code> function” on page 9-50
	Convert an element to an <code>MI_ROW</code> value: “The <code>ts_elem_to_row()</code> function” on page 9-24
	Free memory from a time series element created by <code>ts_make_elem()</code> or <code>ts_row_to_elem()</code> : “The <code>ts_free_elem()</code> procedure” on page 9-26
Extract column data from an element	Get a column from an element by name: “The <code>ts_colinfo_name()</code> function” on page 9-15
	Get a column from an element by number: “The <code>ts_colinfo_number()</code> function” on page 9-15
	Pull columns from an element into <i>values</i> and <i>nulls</i> arrays: “The <code>ts_get_all_cols()</code> procedure” on page 9-27
Create and perform calculations with time stamps	Compare two time stamps: “The <code>ts_datetime_cmp()</code> function” on page 9-21
	Get fields from a time stamp: “The <code>ts_get_stamp_fields()</code> procedure” on page 9-32
	Create a time stamp: “The <code>ts_make_stamp()</code> function” on page 9-40
	Calculate the number of intervals between two time stamps: “The <code>ts_tstamp_difference()</code> function” on page 9-51
	Subtract <i>N</i> intervals from a time stamp: “The <code>ts_tstamp_minus()</code> function” on page 9-52
Get information about element data	Add <i>N</i> intervals to a time stamp: “The <code>ts_tstamp_plus()</code> function” on page 9-53
	Find the number of a column: “The <code>ts_col_id()</code> function” on page 9-14
	Return the number of columns contained in each element: “The <code>ts_col_cnt()</code> function” on page 9-14
	Get type information for a column specified by number: “The <code>ts_colinfo_number()</code> function” on page 9-15
	Get type information for a column specified by name: “The <code>ts_colinfo_name()</code> function” on page 9-15
	Determine whether an element is hidden: “The <code>TS_ELEM_HIDDEN</code> macro” on page 9-23
	Determine whether an element is NULL: “The <code>TS_ELEM_NULL</code> macro” on page 9-24

Table 9-1. Time series API routines sorted by task (continued)

Task type	Description
Get information about a time series	Get the name of a calendar associated with a time series: "The ts_get_calname() function" on page 9-27
	Return the number of elements in a time series: "The ts_nelems() function" on page 9-41
	Return the flags associated with the time series: "The ts_get_flags() function" on page 9-30
	Get the name of the container: "The ts_get_containername() function" on page 9-29
	Determine whether the time series is in a container: "The TS_IS_INCONTAINER macro" on page 9-36
	Get the origin of the time series: "The ts_get_origin() function" on page 9-31
	Get the metadata associated with the time series: "The ts_get_metadata() function" on page 9-31
	Determine whether the time series is irregular: "The TS_IS_IRREGULAR macro" on page 9-37
	Determine whether a time series contains packed data: "The ts_get_packed() function" on page 9-32
	Get the frequency of hertz data: "The ts_get_hertz() function" on page 9-30
	Get the compression type of compressed data: "The ts_get_compressed() function" on page 9-29
Get information about a calendar	Return the number of valid intervals between two time stamps: "The ts_cal_index() function" on page 9-9
	Return all valid timepoints between two time stamps: "The ts_cal_range() function" on page 9-10
	Return a specified number of time stamps starting at a specified time stamp: "The ts_cal_range_index() function" on page 9-11
	Return the time stamp at a specified number of intervals after a specified time stamp: "The ts_cal_stamp() function" on page 9-11

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

API routine	SQL routine
<code>ts_cal_stamp()</code>	<code>CalStamp</code>
<code>ts_create()</code>	<code>TSCreate, TSCreateIrr</code>
<code>ts_create_with_metadata()</code>	<code>TSCreate, TSCreateIrr</code>
<code>ts_del_elem()</code>	<code>DelElem</code>
<code>ts_elem()</code>	<code>GetElem</code>
<code>ts_first_elem()</code>	<code>GetFirstElem</code>
<code>ts_get_calname()</code>	<code>GetCalendarName</code>
<code>ts_get_containername()</code>	<code>GetContainerName</code>
<code>ts_get_metadata()</code>	<code>GetMetaData</code>
<code>ts_get_origin()</code>	<code>GetOrigin</code>
<code>ts_hide_elem()</code>	<code>HideElem</code>
<code>ts_index()</code>	<code>GetIndex</code>
<code>ts_ins_elem()</code>	<code>InsElem</code>
<code>ts_last_elem()</code>	<code>GetLastElem</code>
<code>ts_nelems()</code>	<code>GetNelems</code>
<code>ts_next_valid()</code>	<code>GetNextValid</code>
<code>ts_nth_elem()</code>	<code>GetNthElem</code>
<code>ts_previous_valid()</code>	<code>GetPreviousValid</code>
<code>ts_put_elem()</code>	<code>PutElem</code>
<code>ts_put_elem_no_dups()</code>	<code>PutElemNoDups</code>
<code>ts_put_ts()</code>	<code>PutTimeSeries</code>
<code>ts_reveal_elem()</code>	<code>RevealElem</code>
<code>ts_time()</code>	<code>GetStamp</code>
<code>ts_update_metadata()</code>	<code>UpdMetaData</code>
<code>ts_upd_elem()</code>	<code>UpdElem</code>

The `ts_begin_scan()` function

The `ts_begin_scan()` function begins a scan of elements in a time series.

Syntax

```
ts_tscan *
ts_begin_scan(ts_tsdesc  *tsdesc,
              mi_integer  flags,
              mi_datetime *begin_stamp,
              mi_datetime *end_stamp)
```

tsdesc Returned by `ts_open()`.

flags Determines how a scan should work on the returned set.

begin_stamp

Pointer to **mi_datetime**, to specify where the scan should start. If *begin_stamp* is NULL, the scan starts at the beginning of the time series. The *begin_stamp* argument acts much like the *begin_stamp* argument to the **Clip** function (“Clip function” on page 7-31) unless `TS_SCAN_EXACT_START` is set.

end_stamp

Pointer to **mi_datetime**, to specify where the scan should stop. If *end_stamp* is NULL, the scan stops at the end of the time series. When *end_stamp* is set, the scan stops after the data at *end_stamp* is returned.

Description

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

The scan descriptor is closed by calling `ts_end_scan()`.

The *flags* argument values

The *flags* argument determines how a scan should work on the returned set. Valid values for the *flags* argument are defined in `tseries.h`. The integer value is the sum of the desired values from the following table.

Flag	Value	Meaning
TS_SCAN_HIDDEN	512 (0x200)	Return hidden elements marked by <code>ts_hide_elem()</code>
TS_SCAN_EXACT_START	256 (0x100)	Return NULL if the begin point is earlier than the time series origin. (Normally a scan does not start before the time series origin.)
TS_SCAN_EXACT_END	128 (0x80)	Return NULL until the end timepoint of the scan is reached, even if the end timepoint is beyond the end of the time series.
TS_SCAN_NO_NULLS	32 (0x20)	Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated it is returned as NULL. If TS_SCAN_NO_NULLS is set, an element is returned that has each column set to NULL instead.
TS_SCAN_SKIP_END	16 (0x10)	Skip the element at the end timepoint of the scan range.
TS_SCAN_SKIP_BEGIN	8 (0x08)	Skip the element at the beginning timepoint of the scan range.
TS_SCAN_SKIP_HIDDEN	4 (0x04)	Skip hidden elements.

Returns

An open scan descriptor, or NULL if the scan times are both before the origin of the time series or if the end time is before the start time.

Example

See the `ts_interp()` function, in Appendix A, “The Interp function example,” on page A-1, for an example of the `ts_begin_scan()` function.

Related reference:

“HideElem function” on page 7-67

“The `ts_current_offset()` function” on page 9-20

“The `ts_current_timestamp()` function” on page 9-21

“The `ts_end_scan()` procedure” on page 9-25

“The `ts_next()` function” on page 9-41

“The `ts_open()` function” on page 9-44

“The `ts_first_elem()` function” on page 9-25

The `ts_cal_index()` function

The `ts_cal_index()` function returns the number of valid intervals in a calendar between two given time stamps.

Syntax

```
mi_integer *  
ts_cal_index (MI_CONNECTION *conn,  
              mi_string      *cal_name,  
              mi_datetime    *begin_stamp,  
              mi_datetime    *end_stamp)
```

conn A valid DataBlade API connection.

cal_name
 The name of the calendar.

begin_stamp
 The beginning time stamp. *begin_stamp* must not be earlier than the calendar origin.

end_stamp
 The time stamp whose offset from *begin_stamp* is to be determined. This time stamp can be earlier than *begin_stamp*.

Description

The equivalent SQL function is **CalIndex**.

Returns

The number of valid intervals in the given calendar between the two time stamps. If *end_stamp* is earlier than *begin_stamp*, then the result is a negative number.

Related reference:

“The `ts_cal_range()` function” on page 9-10

“The `ts_cal_range_index()` function” on page 9-11

“The `ts_cal_stamp()` function” on page 9-11

“The `ts_index()` function” on page 9-35

The `ts_cal_pattstartdate()` function

The `ts_cal_pattstartdate()` function takes a calendar name and returns the start date of the pattern for that calendar.

Syntax

```
mi_datetime *  
ts_cal_pattstartdate (MI_CONNECTION *conn,  
                      mi_string      *cal_name)
```

conn A pointer to a valid DataBlade API connection structure.

cal_name
 The name of the calendar.

Description

The equivalent SQL function is **CalPattStartDate**.

Returns

An `mi_datetime` pointer that points to the start date of a calendar pattern. You must free this value after use.

Related reference:

“CalPattStartDate function” on page 5-2

“The `ts_cal_startdate()` function” on page 9-12

The `ts_cal_range()` function

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

Syntax

```
MI_COLLECTION *  
ts_cal_range (MI_CONNECTION *conn,  
              mi_string      *cal_name,  
              mi_datetime    *begin_stamp,  
              mi_datetime    *end_stamp)
```

conn A valid DataBlade API connection.

cal_name
 The name of the calendar.

begin_stamp
 The begin point of the range. It must not be earlier than the calendar origin.

end_stamp
 The end point of the range.

Description

This function is useful if you must print out the time stamps of a series of regular time series elements. If the range is known, getting an array of all of the time stamps is more efficient than using `ts_time()` on each element.

The caller is responsible for freeing the result of this function.

The equivalent SQL function is **CalRange**.

Returns

A list of time stamps.

Related reference:

“CalIndex function” on page 6-2

“CalRange function” on page 6-3

“CalStamp function” on page 6-4

“The `ts_cal_index()` function” on page 9-9

“The `ts_cal_range_index()` function” on page 9-11

“The `ts_time()` function” on page 9-51

“The `ts_cal_stamp()` function” on page 9-11

The `ts_cal_range_index()` function

The `ts_cal_range_index()` function returns a list containing a specified number of time stamps starting at a given time stamp.

Syntax

```
MI_COLLECTION *  
ts_cal_range_index (MI_CONNECTION, *conn,  
                   mi_string      *cal_name,  
                   mi_datetime    *begin_stamp,  
                   mi_integer     num_stamps)
```

conn A valid DataBlade API connection.

cal_name
 The name of the calendar.

begin_stamp
 The beginning of the range. It must be greater than or equal to the calendar origin.

num_stamps
 The number of time stamps to return.

Description

This function is useful if you must print out the time stamps of a series of regular time series elements. If the range is known, getting an array of all of the time stamps is more efficient than using `ts_time()` on each element.

The caller is responsible for freeing the result of this function.

Returns

A list of time stamps.

Related reference:

“CalIndex function” on page 6-2

“CalRange function” on page 6-3

“CalStamp function” on page 6-4

“The `ts_cal_index()` function” on page 9-9

“The `ts_cal_range()` function” on page 9-10

“The `ts_cal_stamp()` function”

“The `ts_time()` function” on page 9-51

The `ts_cal_stamp()` function

The `ts_cal_stamp()` function returns the time stamp at a given number of calendar intervals before or after a given time stamp. The returned time stamp is located in allocated memory, so the caller should free it using `mi_free()`.

Syntax

```
mi_datetime *  
ts_cal_stamp (MI_CONNECTION *conn,  
             mi_string      *cal_name,  
             mi_datetime    *tstamp,  
             mi_integer     offset)
```

conn A valid DataBlade API connection.

cal_name
The name of the calendar.

tstamp The input time stamp.

offset The number of calendar intervals before or after the input time stamp. Use a negative number to indicate an offset before the specified time stamp and a positive number to indicate an offset after the specified time stamp.

Description

The equivalent SQL function is **CalStamp**.

Returns

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

Related reference:

“CalIndex function” on page 6-2

“CalRange function” on page 6-3

“CalStamp function” on page 6-4

“The ts_cal_index() function” on page 9-9

“The ts_cal_range_index() function” on page 9-11

“The ts_cal_range() function” on page 9-10

The ts_cal_startdate() function

The **ts_cal_startdate()** function returns the start date of a calendar.

Syntax

```
mi_datetime *  
ts_cal_startdate (MI_CONNECTION *conn,  
                  mi_string      *cal_name)
```

conn A pointer to a valid DataBlade API connection structure.

cal_name
The name of the calendar.

Description

The equivalent SQL function is **CalStartDate**.

Returns

An *mi_datetime* pointer that points to the start date of a calendar. You must free this value after use.

Related reference:

“CalStartDate function” on page 6-5

“The ts_cal_pattstartdate() function” on page 9-9

The ts_close() function

The **ts_close()** procedure closes the associated time series.

Syntax

```
void  
ts_close(ts_tsdesc *tsdesc)
```

tsdesc A time series descriptor returned by **ts_open**.

Description

After a call to this procedure, *tsdesc* is no longer valid and so should not be passed to any routine requiring the *tsdesc* argument.

Example

See the **ts_interp()** function, Appendix A, “The Interp function example,” on page A-1, for an example of **ts_close()**.

Related reference:

“The **ts_open()** function” on page 9-44

The **ts_closest_elem()** function

The **ts_closest_elem()** function returns the first element, or column(s) of an element, that is non-null and closest to the given time stamp.

Syntax

```
ts_tselem  
ts_closest_elem(ts_tsdesc *tsdesc,  
                mi_datetime  *tstamp,  
                mi_string    *cmp,  
                mi_string    *col_list,  
                mi_integer    flags, mi_integer *isNull,  
                mi_integer    *off)
```

tsdesc A time series descriptor returned by **ts_open**.

tstamp The time stamp to start searching from.

cmp A comparison operator. Valid values for *cmp* are <, <=, =, ==, '>=, and >.

col_list To search for an element with a particular set of columns non-null, specify a list of column names separated by a vertical bar (|). An error is raised if any of the column names do not exist in the time series sub-rowtype.

To search for a non-null element, set *col_list* to NULL.

flags Determines whether hidden elements should be returned. Valid values for the *flags* parameter are defined in *tseries.h*. They are:

- TS_CLOSEST_NO_FLAGS (no special flags)
- TS_CLOSEST_RETNULLS_FLAGS (return hidden elements)

isNull The *isNull* parameter must not be NULL. On return, it is set with the null indicator bits found in *tseries.h*. These are:

- 0 (element is not hidden and is allocated)
- TS_NULL_NOTALLOCED (element has not been written to)
- TS_NULL_HIDDEN (element is hidden)

off If the time series is regular, the offset of the returned element will be returned in the *off* parameter, if *off* is not NULL.

Description

The search algorithm that **ts_closest_elem** uses is as follows:

- If *cmp* is any of <=, =, ==, or >=, the search starts at *tstamp*.
- If *cmp* is <, the search starts at the first element before *tstamp*.
- If *cmp* is >, the search starts at the first element after *tstamp*.

The *tstamp* and *cmp* parameters are used to determine where to start the search. The search continues in the direction indicated by *cmp* until an element is found that qualifies. If no element qualifies, then the return value is NULL.

Important: For irregular time series, values in an irregular element persist until the next element. This means that any of the previous “equals” operations on an irregular time series will look for <= first. If *cmp* is >= and the <= operations fails, the operation then looks forward for the next element; otherwise, NULL is returned.

Returns

An element that meets the criteria described.

The ts_col_cnt() function

The **ts_col_cnt()** function returns the number of columns contained in each element of a time series.

Syntax

```
mi_integer  
ts_col_cnt (ts_tsdesc *tsdesc)
```

tsdesc A time series descriptor returned by **ts_open**.

Returns

The number of columns.

Related reference:

“The ts_get_all_cols() procedure” on page 9-27

The ts_col_id() function

The **ts_col_id()** function takes a column name and returns the associated column number.

Syntax

```
mi_integer  
ts_col_id(ts_tsdesc *tsdesc,  
          mi_string *colname)
```

tsdesc A time series descriptor returned by **ts_open()**.

colname

The name of the column.

Description

Column numbers start at 0; therefore, the first time stamp column is always column 0.

Returns

The number of the column associated with *colname*.

Related reference:

“The `ts_colinfo_name()` function”

“The `ts_colinfo_number()` function”

The `ts_colinfo_name()` function

The `ts_colinfo_name()` function gets type information for a column in a time series.

Syntax

```
ts_typeinfo *
ts_colinfo_name (ts_tsdesc *tsdesc,
                 mi_string *colname)
```

tsdesc A time series descriptor returned by `ts_open()`.

colname

The name of the column to return information for.

Description

The resulting `typeinfo` structure and its `ti_typename` field must be freed by the caller.

Returns

A pointer to a `ts_typeinfo` structure. This structure is defined as follows:

```
typedef struct _ts_typeinfo
{
    MI_TYPEID      *ti_typeid; /* type id */
    mi_integer      ti_typelen; /* internal length */
    mi_smallint     ti_typealign; /* internal alignment */
    mi_smallint     ti_typebyvalue; /* internal byvalue flag */
    mi_integer      ti_typebound; /* internal bound */
    mi_integer      ti_typeparameter; /* internal parameter */
    mi_string       *ti_typename; /* type name of the column */
} ts_typeinfo;
```

Related reference:

“The `ts_col_id()` function” on page 9-14

“The `ts_colinfo_number()` function”

The `ts_colinfo_number()` function

The `ts_colinfo_number()` function gets type information for a column in a time series.

Syntax

```
ts_typeinfo *
ts_colinfo_number (ts_tsdesc *tsdesc,
                   mi_integer id)
```

tsdesc A time series descriptor returned by `ts_open()`.

id The column number to return information for. The *id* argument must be greater than or equal to 0 and less than the number of columns in a time series element. An *id* of 0 corresponds to the time stamp column.

Description

The resulting **typeinfo** structure and its **ti_typename** field must be freed by the caller.

Returns

A pointer to a **ts_typeinfo** structure. This structure is defined as follows:

```
typedef struct _ts_typeinfo
{
    MI_TYPEID      *ti_typeid; /* type id */
    mi_integer      ti_typelen; /* internal length */
    mi_smallint     ti_typealign; /* internal alignment */
    mi_smallint     ti_typebyvalue; /* internal byvalue flag */
    mi_integer      ti_typebound; /* internal bound */
    mi_integer      ti_typeparameter; /* internal parameter */
    mi_string       *ti_typename; /* type name of the column */
} ts_typeinfo;
```

Example

See the **ts_interp()** function, Appendix A, “The Interp function example,” on page A-1, for an example of **ts_colinfo_number()**.

Related reference:

“The **ts_col_id()** function” on page 9-14

“The **ts_colinfo_name()** function” on page 9-15

The **ts_copy()** function

The **ts_copy()** function makes and returns a copy of the given time series of the type in the *type_id* argument.

Syntax

```
ts_timeseries *
ts_copy(MI_CONNECTION *conn,
        ts_timeseries *ts,
        MI_TYPEID      *typeid)
```

conn A valid DataBlade API connection.

ts The time series to be copied.

typeid The ID of the row type of the time series to be copied.

Description

Since values returned by **mi_value()** are valid only until the next **mi_next_row()** or **mi_query_finish()** call, it is sometimes necessary to use **ts_copy()** to access a time series outside an **mi_get_result()** loop.

On the client, you must use the **ts_copy()** function to make a copy of a time series before you pass the time series as an argument to the **mi_exec_prepare()** statement.

Returns

A copy of the given time series. This value must be freed by the user by calling **ts_free()**.

Related reference:

“The `ts_free()` procedure” on page 9-26

“The `ts_get_typeid()` function” on page 9-34

The `ts_create()` function

The `ts_create()` function creates a time series.

Syntax

```
ts_timeseries *
ts_create(MI_CONNECTION *conn,
          mi_string      *calname,
          mi_datetime     *origin,
          mi_integer      threshold,
          mi_integer      flags,
          MI_TYPEID       *typeid,
          mi_integer      nelem,
          mi_string       *container)
```

```
ts_timeseries *
ts_create(MI_CONNECTION *conn,
          mi_string      *calname,
          mi_datetime     *origin,
          mi_integer      threshold,
          MI_TYPEID       *typeid,
          mi_integer      nelem,
          mi_string       *container,
          mi_integer      hertz)
```

```
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,
          mi_string       compression)
```

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 0 = Regular time series
 1 = Irregular time series
 3 = Hertz time series
 5 = Compressed 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.

hertz (**Optional**)

An integer 1-255 that specifies the number of records per second. Implicitly creates an irregular time series.

The value of the *threshold* parameter must be 0. The value of the *flags* parameter must be 3.

compression (**Optional**)

A string that includes a compression definition for each column in the **TimeSeries** subtype except the first column. For the syntax of the compression parameter and descriptions of the compression types and attributes, see "TSCreateIrr function" on page 7-118.

The value of the *threshold* parameter must be 0. The value of the *flags* parameter must be 5.

Description

The equivalent SQL function is **TSCreate** or **TSCreateIrr**.

If you include the *hertz* or *compression* parameter, you must run the **ts_create()** function within an explicit transaction.

Returns

A pointer to a new time series. The user can free this value by calling **ts_free()**.

Related reference:

"TSCreateIrr function" on page 7-118

"TSCreate function" on page 7-116

"The ts_free() procedure" on page 9-26

"The ts_open() function" on page 9-44

"The ts_get_threshold() function" on page 9-33

"The ts_get_typeid() function" on page 9-34

The ts_create_with_metadata() function

The **ts_create_with_metadata()** function creates a time series with user-defined metadata attached.

Syntax

```
ts_timeseries *
ts_create_with_metadata(MI_CONNECTION *conn,
                        mi_string      *calname,
                        mi_datetime    *origin,
                        mi_integer     threshold,
                        mi_integer     flags,
                        MI_TYPEID      *typeid,
                        mi_integer     nelem,
                        mi_string      *container,
                        mi_lvarchar    *metadata,
                        MI_TYPEID      *metadata_typeid)

ts_timeseries *
ts_create_with_metadata(MI_CONNECTION *conn,
```

```

        mi_string      *calname,
        mi_datetime    *origin,
        mi_integer     threshold,
        MI_TYPEID      *typeid,
        mi_integer     nelem,
        mi_string       *container,
        mi_lvarchar     *metadata,
        MI_TYPEID      *metadata_typeid,
        mi_integer      hertz)

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,
        mi_string      compression)

```

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 0 = Regular time series
 1 = Irregular time series
 3 = Hertz time series
 5 = Compressed 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. For more information about metadata, see “Creating a time series with metadata” on page 3-23. Can be NULL.

metadata_typeid The type ID of the metadata. Can be NULL if the metadata argument is NULL.

***hertz* (Optional)**

An integer 1-255 that specifies the number of records per second. Implicitly creates an irregular time series.

The value of the *threshold* parameter must be 0. The value of the *flags* parameter must be 3.

compression (Optional)

A string that includes a compression definition for each column in the **TimeSeries** subtype except the first column. For the syntax of the compression parameter and descriptions of the compression types and attributes, see “TSCreateIrr function” on page 7-118.

The value of the *threshold* parameter must be 0. The value of the *flags* parameter must be 5.

Description

This function behaves the same as **ts_create()**, plus it saves the supplied metadata in the time series. The metadata can be NULL or a zero-length LVARCHAR; if either, **ts_create_with_metadata()** acts exactly like **ts_create()**. If the *metadata* pointer points to valid data, the *metadata_typeid* parameter must be a valid pointer to a valid type ID for a user-defined type.

If you include the *hertz* or *compression* parameter, you must run the **ts_create()** function within an explicit transaction.

The equivalent SQL function is **TSCreate** or **TSCreateIrr**.

Returns

A pointer to a new time series. The user can free this value by calling **ts_free()**.

Related reference:

“GetMetaData function” on page 7-59

“GetMetaTypeName function” on page 7-59

“TSCreateIrr function” on page 7-118

“UpdMetaData function” on page 7-159

“TSCreate function” on page 7-116

“The ts_free() procedure” on page 9-26

“The ts_open() function” on page 9-44

“The ts_get_metadata() function” on page 9-31

“The ts_get_typeid() function” on page 9-34

“The ts_update_metadata() function” on page 9-54

The ts_current_offset() function

The **ts_current_offset()** function returns the offset for the last element returned by **ts_next()**.

Syntax

mi_integer
ts_current_offset(*ts_tscan *tscan*)

tscan The scan descriptor returned by **ts_begin_scan()**.

Returns

The offset of the last element returned. If no element has been returned yet, the offset of the first element is returned. For irregular time series, **ts_current_offset()** always returns -1.

Related reference:

“The **ts_begin_scan()** function” on page 9-7

The **ts_current_timestamp()** function

The **ts_current_timestamp()** function finds the time stamp that corresponds to the current element retrieved from the scan.

Syntax

```
mi_datetime *  
ts_current_timestamp(ts_tscan *scan)
```

scan The scan descriptor returned by **ts_begin_scan()**.

Returns

If no elements have been retrieved, the value returned is the time stamp of the first element. This value cannot be freed by the user with **mi_free()**.

Related reference:

“The **ts_begin_scan()** function” on page 9-7

The **ts_datetime_cmp()** function

The **ts_datetime_cmp()** function compares two time stamps and returns a value that indicates whether *tstamp1* is before, equal to, or after *tstamp2*.

Syntax

```
mi_integer  
ts_datetime_cmp(mi_datetime *tstamp1,  
                mi_datetime *tstamp2)
```

tstamp1

The first time stamp to compare.

tstamp2

The second time stamp to compare.

Returns

< 0 If *tstamp1* comes before *tstamp2*.

0 If *tstamp1* equals *tstamp2*.

> 0 If *tstamp1* comes after *tstamp2*.

Related reference:

“The **ts_get_all_cols()** procedure” on page 9-27

“The **ts_get_col_by_name()** function” on page 9-28

“The **ts_get_col_by_number()** function” on page 9-28

The `ts_del_elem()` function

The `ts_del_elem()` function deletes an element from a time series at a given timepoint.

Syntax

```
ts_timeseries *  
ts_del_elem(ts_tsdesc  *tsdesc,  
            mi_datetime *tstamp)
```

tsdesc The time series descriptor returned by `ts_open()`.

tstamp The timepoint from which to delete the element.

Description

If there is no element at the timepoint, no error is raised, and no change is made to the time series. It is an error to delete a hidden element.

The equivalent SQL function is **DelElem**.

Returns

The original time series minus the element deleted, if there was one.

Related reference:

“DelElem function” on page 7-43

“The `ts_ins_elem()` function” on page 9-36

“The `ts_put_elem()` function” on page 9-46

“The `ts_upd_elem()` function” on page 9-54

The `ts_elem()` function

The `ts_elem()` function returns an element from the time series at the given time.

Syntax

```
ts_tselem  
ts_elem(ts_tsdesc  *tsdesc,  
        mi_datetime *tstamp,  
        mi_integer  *STATUS,  
        mi_integer  *off)
```

tsdesc The time series descriptor returned by `ts_open()`.

tstamp A pointer to the time stamp for the desired element.

STATUS

Set on return to indicate whether the element is NULL or hidden. See “The `ts_hide_elem()` function” on page 9-34 for an explanation of the *isNull* argument.

off For regular time series, *off* is set to the offset on return. If the time series is irregular, or if the time stamp is not in the calendar, *off* is set to -1. The offset can be NULL.

Description

On return, *off* is filled in with the offset of the element for a regular time series or -1 for an irregular time series. The element is overwritten after two calls to fetch elements using this *tsdesc* (time series descriptor).

The equivalent SQL function is **GetElem**.

Returns

An element, its offset, and whether it is hidden, NULL, or both. This element must not be freed by the caller.

Related reference:

"GetElem function" on page 7-52

"DelElem function" on page 7-43

"The TS_ELEM_HIDDEN macro"

"The ts_hide_elem() function" on page 9-34

"The ts_last_elem() function" on page 9-37

"The ts_nth_elem() function" on page 9-43

"The TS_ELEM_NULL macro" on page 9-24

"The ts_first_elem() function" on page 9-25

"The ts_ins_elem() function" on page 9-36

"The ts_make_elem() function" on page 9-38

"The ts_put_elem() function" on page 9-46

"The ts_put_elem_no_dups() function" on page 9-47

"The ts_put_last_elem() function" on page 9-48

"The ts_put_nth_elem() function" on page 9-48

"The ts_upd_elem() function" on page 9-54

The TS_ELEM_HIDDEN macro

The TS_ELEM_HIDDEN macro determines whether the STATUS indicator returned by **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, and similar functions is set because the associated element was hidden.

Syntax

TS_ELEM_HIDDEN(*(mi_integer) STATUS*)

STATUS

The *mi_integer* argument previously passed to **ts_elem()**, **ts_nth_elem()**, **ts_first_elem()**, or a similar function.

Description

This macro returns a nonzero value if the associated element is hidden. This macro is often used in concert with TS_ELEM_NULL.

Returns

A nonzero value if the element associated with the *STATUS* argument was previously hidden by the **ts_hide_elem()** function.

Related reference:

“The `ts_elem()` function” on page 9-22
“The `TS_ELEM_NULL` macro”
“The `ts_first_elem()` function” on page 9-25
“The `ts_hide_elem()` function” on page 9-34
“The `ts_last_elem()` function” on page 9-37
“The `ts_next()` function” on page 9-41
“The `ts_next_valid()` function” on page 9-42
“The `ts_nth_elem()` function” on page 9-43
“The `ts_previous_valid()` function” on page 9-45

The `TS_ELEM_NULL` macro

The `TS_ELEM_NULL` macro determines whether the `STATUS` indicator returned by `ts_elem()`, `ts_nth_elem()`, `ts_first_elem()`, or a similar function is `NULL` because the associated element is `NULL`.

Syntax

```
TS_ELEM_NULL((mi_integer) STATUS)
```

STATUS

The *mi_integer* argument previously passed to `ts_elem()`, `ts_nth_elem()`, `ts_first_elem()`, or a similar function.

Description

This macro returns a nonzero value if the associated element is `NULL`. This macro is often used in concert with `TS_ELEM_HIDDEN`.

Returns

A nonzero value if the element returned by `ts_elem()`, `ts_nth_elem()`, `ts_first_elem()`, or similar function was `NULL`.

Related reference:

“The `TS_ELEM_HIDDEN` macro” on page 9-23
“The `ts_elem()` function” on page 9-22
“The `ts_first_elem()` function” on page 9-25
“The `ts_hide_elem()` function” on page 9-34
“The `ts_last_elem()` function” on page 9-37
“The `ts_next()` function” on page 9-41
“The `ts_next_valid()` function” on page 9-42
“The `ts_nth_elem()` function” on page 9-43
“The `ts_previous_valid()` function” on page 9-45

The `ts_elem_to_row()` function

The `ts_elem_to_row()` function converts a time series element into a new row.

Syntax

```
MI_ROW *  
ts_elem_to_row(ts_tsdesc *tsdesc,  
               ts_tselem elem,  
               mi_integer off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

elem A time series element. It must agree in type with the time series described by *tsdesc*.

off If the time series is regular and *off* is non-negative, *off* is used to compute the time stamp value placed in the first column of the returned row.

If the time series is regular and *off* is negative, column 0 of the resulting row will be taken from column 0 of the *elem* parameter (which will be NULL if the element was created for or extracted from a regular time series).

If the time series is irregular, the *off* parameter is ignored.

Returns

A row. The row must be freed by the caller using the **mi_row_free()** procedure.

Related reference:

“The **ts_free_elem()** procedure” on page 9-26

“The **ts_make_elem()** function” on page 9-38

“The **ts_make_elem_with_buf()** function” on page 9-39

“The **ts_row_to_elem()** function” on page 9-50

The **ts_end_scan()** procedure

The **ts_end_scan()** procedure ends a scan of a time series. It releases resources acquired by **ts_begin_scan()**. Upon return, no more elements can be retrieved using the given **ts_tscan** pointer.

Syntax

```
void
ts_end_scan(ts_tscan *scan)
```

scan The scan to be ended.

Example

See the **ts_interp()** function, Appendix A, “The Interp function example,” on page A-1, for an example of **ts_end_scan()**.

Related reference:

“The **ts_begin_scan()** function” on page 9-7

The **ts_first_elem()** function

The **ts_first_elem()** function returns the first element in the time series.

Syntax

```
ts_tselem
ts_first_elem(ts_tsdesc *tsdesc,
              mi_integer *STATUS)
```

tsdesc The time series descriptor returned by **ts_open()**.

STATUS

A pointer to an **mi_integer** value. See “The **ts_hide_elem()** function” on page 9-34 for an explanation of the *STATUS* argument.

Description

If the time series is regular, the first element is always the origin of the time series. If the time series is irregular, the first element is the one with the earliest time stamp. The value must not be freed by the caller. The element is overwritten after two calls to fetch elements using this *tsdesc* (time series descriptor).

The equivalent SQL function is **GetFirstElem**.

Returns

The first element in the time series.

Related reference:

“GetFirstElem function” on page 7-53

“The TS_ELEM_HIDDEN macro” on page 9-23

“The TS_ELEM_NULL macro” on page 9-24

“GetElem function” on page 7-52

“The ts_begin_scan() function” on page 9-7

“The ts_elem() function” on page 9-22

“The ts_next() function” on page 9-41

“The ts_next_valid() function” on page 9-42

The ts_free() procedure

The **ts_free()** procedure frees all memory associated with the given time series argument. The time series argument must have been generated by a call to either **ts_create()** or **ts_copy()**.

Syntax

```
void  
ts_free(ts_timeseries *ts)
```

ts The source time series.

Related reference:

“The ts_copy() function” on page 9-16

“The ts_create() function” on page 9-17

“The ts_create_with_metadata() function” on page 9-18

“The ts_get_ts() function” on page 9-33

“The ts_ins_elem() function” on page 9-36

The ts_free_elem() procedure

The **ts_free_elem()** procedure frees a time series element, releasing its resources. It is used to free elements created by **ts_make_elem()** or **ts_row_to_elem()**. It must not be called to free elements returned by **ts_elem()**, **ts_first_elem()**, **ts_last_elem()**, **ts_last_valid()**, **ts_next()**, **ts_next_valid()**, **ts_nth_elem()**, or **ts_previous_valid()**; those elements are overwritten with subsequent calls or freed when the corresponding scan or time series descriptor is closed.

Syntax

```
void  
ts_free_elem(ts_tsdesc *tsdesc,  
             ts_tselem elem)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

elem A time series element. It must agree in type with the time series described by *tsdesc*.

Related reference:

“The **ts_elem_to_row()** function” on page 9-24

“The **ts_make_elem()** function” on page 9-38

“The **ts_make_elem_with_buf()** function” on page 9-39

“The **ts_row_to_elem()** function” on page 9-50

The **ts_get_all_cols()** procedure

The **ts_get_all_cols()** procedure loads the values in the element into the *values* and *nulls* arrays.

Syntax

```
void  
ts_get_all_cols(ts_tsdesc *tsdesc,  
                ts_tselem tselem,  
                MI_DATUM *values,  
                mi_boolean *nulls,  
                mi_integer off)
```

tsdesc A time series pointer returned by **ts_open()**.

tselem The element to extract data from.

values The array to put the column data into. This array must be large enough to hold data for all the columns of the time series.

nulls An array that indicates null values.

off For a regular time series, *off* is the offset of the element. For an irregular time series, *off* is ignored.

Returns

None. The *values* and *nulls* arrays are filled in with data from the element. The *values* array is filled with values or pointers to values depending on whether the corresponding column is by reference or by value. The values in the *values* array must not be freed by the caller.

Related reference:

“The **ts_datetime_cmp()** function” on page 9-21

“The **ts_col_cnt()** function” on page 9-14

The **ts_get_calname()** function

The **ts_get_calname()** function returns the name of the calendar associated with the given time series.

Syntax

```
mi_string *  
ts_get_calname(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetCalendarName**.

Returns

The name of the calendar. This value must be freed by the caller with **mi_free()**.

The **ts_get_col_by_name()** function

The **ts_get_col_by_name()** function pulls out the individual piece of data from an element in the column with the given name.

Syntax

```
MI_DATUM
ts_get_col_by_name(ts_tsdesc *tsdesc,
                  ts_tselem tselem,
                  mi_string *colname,
                  mi_boolean *isNull,
                  mi_integer off)
```

tsdesc A pointer returned by **ts_open()**.

tselem An element to get column data from.

colname
The name of the column in the element.

isNull A pointer to a null indicator.

off For a regular time series, *off* is the offset of the element in the time series.
For an irregular time series, *off* is ignored.

Returns

The data in the specified column. If the type of the column is by reference, a pointer is returned. If the type is by value, the data itself is returned. The caller cannot free this value. On return, *isNull* is set to indicate whether the column is NULL.

Related reference:

“The **ts_datetime_cmp()** function” on page 9-21

“The **ts_get_col_by_number()** function”

The **ts_get_col_by_number()** function

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

Syntax

```
MI_DATUM
ts_get_col_by_number(ts_tsdesc *tsdesc,
                    ts_tselem tselem,
                    mi_integer colnumber,
                    mi_boolean *isNull,
                    mi_integer off)
```

tsdesc A pointer returned by **ts_open()**.

tselem An element to get column data from.

colnumber

The column number. Column numbers start at 0, which represents the time stamp.

isNull A pointer to a null indicator.

off For a regular time series, *off* is the offset of the element in the time series. For an irregular time series, *off* is ignored.

Returns

The data in the specified column. If the type of the column is by reference, a pointer is returned. If the type is by value, the data itself is returned. The caller cannot free this value. On return, *isNull* is set to indicate whether the column is NULL.

Example

See the `ts_interp()` function, Appendix A, “The Interp function example,” on page A-1, for an example of `ts_get_col_by_number()`.

Related reference:

“The `ts_datetime_cmp()` function” on page 9-21

“The `ts_get_col_by_name()` function” on page 9-28

The `ts_get_compressed()` function

The `ts_get_compressed()` function returns the compression string if the time series data is compressed.

Syntax

```
mi_string  
ts_get_compressed(ts_timeseries *ts)
```

ts The name of the time series.

Description

Use the `ts_get_compressed()` function to determine the type of compression that is used in a time series that contains compressed numeric data.

Returns

Returns a string that represents the compression type if the time series contains compressed data; returns NULL if the time series does not contain compressed data.

Related reference:

“GetCompression function” on page 7-50

The `ts_get_containername()` function

The `ts_get_containername()` function gets the container name of the given time series.

Syntax

```
mi_string *  
ts_get_containername(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetContainerName**.

Returns

The name of the container for the given time series. This value must not be freed by the user.

Related reference:

“GetContainerName function” on page 7-51

The **ts_get_flags()** function

The **ts_get_flags()** function returns the flags associated with the given time series.

Syntax

```
mi_integer  
ts_get_flags(ts_timeseries *ts)
```

ts The source time series.

Description

The return value is a collection of flag bits. The possible flag bits set are TSFLAGS_IRR, TSFLAGS_INMEM, and TSFLAGS_ASSIGNED.

To check whether the time series is regular, use TS_IS_IRREGULAR.

Returns

An integer containing the flags for the given time series.

Related reference:

“IsRegular function” on page 7-73

“The TS_IS_IRREGULAR macro” on page 9-37

The **ts_get_hertz()** function

The **ts_get_hertz()** function returns the frequency for packed hertz data.

Syntax

```
mi_integer  
ts_get_hertz(ts_timeseries *ts)
```

ts The name of the time series.

Description

Use the **ts_get_hertz()** function to determine how many records per second the time series can store.

Returns

Returns an integer 1-255 if the time series contains packed hertz data; returns 0 if the time series does not contain packed hertz data.

Related reference:

“GetHertz function” on page 7-54

The `ts_get_metadata()` function

The `ts_get_metadata()` function returns the user-defined metadata and its type ID from the specified time series.

Syntax

```
mi_lvarchar *  
ts_get_metadata(ts_timeseries *ts,  
                MI_TYPEID    **metadata_typeid)
```

ts The time series to retrieve the metadata from.

metadata_typeid

The return parameter to hold the type ID of the user-defined metadata.

Description

The equivalent SQL function is **GetMetaData**.

Returns

The user-defined metadata contained in the specified time series. If the time series does not contain any user-defined metadata, then NULL is returned and the *metadata_typeid* pointer is set to NULL. This return value must be cast to the real user-defined type to be useful. The value returned can be freed by the caller with **mi_var_free()**.

Related reference:

“GetMetaData function” on page 7-59

“GetMetaTypeName function” on page 7-59

“UpdMetaData function” on page 7-159

“TSCreate function” on page 7-116

“TSCreateIrr function” on page 7-118

“The `ts_create_with_metadata()` function” on page 9-18

“The `ts_get_metadata()` function”

“The `ts_update_metadata()` function” on page 9-54

The `ts_get_origin()` function

The `ts_get_origin()` function returns the origin of the given time series.

Syntax

```
mi_datetime *  
ts_get_origin(ts_timeseries *ts)
```

ts The source time series.

Description

The equivalent SQL function is **GetOrigin**.

Returns

The origin of the given time series. This value must be freed by the caller using `mi_free()`.

Related reference:

"GetOrigin function" on page 7-64

The `ts_get_packed()` function

The `ts_get_packed()` function returns whether the specified time series contains packed data.

Syntax

```
mi_integer
ts_get_packed(ts_timeseries *ts)

ts          The name of the time series.
```

Description

Use the `ts_get_packed()` function to determine whether a time series stores either hertz data or compressed numeric data in packed elements.

Returns

Returns 1 if the time series contains packed data; returns 0 if the time series does not contain packed data.

Related reference:

"GetPacked function" on page 7-64

The `ts_get_stamp_fields()` procedure

The `ts_get_stamp_fields()` procedure takes a pointer to an `mi_datetime` structure and returns the parameters with the year, month, day, hour, minute, second, and microsecond.

Syntax

```
void
ts_get_stamp_fields (MI_CONNECTION *conn,
                    mi_datetime *dt,
                    mi_integer *year,
                    mi_integer *month,
                    mi_integer *day,
                    mi_integer *hour,
                    mi_integer *minute,
                    mi_integer *second,
                    mi_integer *ms)
```

conn A valid DataBlade API connection.

dt The time stamp to convert.

year Pointer to year integer that the procedure sets. Can be NULL.

month Pointer to month integer that the procedure sets. Can be NULL.

day Pointer to day integer that the procedure sets. Can be NULL.

hour Pointer to hour integer that the procedure sets. Can be NULL.

minute Pointer to minute integer that the procedure sets. Can be NULL.
second Pointer to second integer that the procedure sets. Can be NULL.
ms Pointer to microsecond integer that the procedure sets. Can be NULL.

Returns

On return, the non-null year, month, day, hour, minute, second, and microsecond are set to the time that corresponds to the time indicated by the *dt* argument.

Related reference:

"The `ts_make_stamp()` function" on page 9-40

The `ts_get_threshold()` function

The `ts_get_threshold()` function returns the threshold of the specified time series.

Syntax

```
mi_integer  
ts_get_threshold(ts_timeseries *ts)  
  
ts          The source time series.
```

Description

The equivalent SQL function is **GetThreshold**.

Returns

The threshold of the given time series.

Related reference:

"GetThreshold function" on page 7-67

"The `ts_create()` function" on page 9-17

The `ts_get_ts()` function

The `ts_get_ts()` function returns a pointer to the time series associated with the given time series descriptor.

Syntax

```
ts_timeseries *  
ts_get_ts(ts_tsdesc *tsdesc)  
  
tsdesc      The time series descriptor from ts_open().
```

Description

The `ts_get_ts()` function is useful when you must call a function that takes a time series argument (for example, `ts_get_calname()`), but you only have a *tsdesc* (time series descriptor).

Returns

A pointer to the time series associated with the given time series descriptor. This value can be freed by the caller after `ts_close()` has been called if the original time series was created by `ts_create()` or `ts_copy()`. To free it, use `ts_free()`.

Related reference:

“The `ts_free()` procedure” on page 9-26

“The `ts_put_elem()` function” on page 9-46

“The `ts_put_elem_no_dups()` function” on page 9-47

“The `ts_put_last_elem()` function” on page 9-48

“The `ts_put_nth_elem()` function” on page 9-48

The `ts_get_typeid()` function

The `ts_get_typeid()` function returns the type ID of the specified time series.

Syntax

```
mi_typeid *  
ts_get_typeid(MI_CONNECTION *conn,  
              ts_timeseries *ts)
```

conn A valid DataBlade API connection.

ts The source time series.

Description

This function returns the type ID of the specified time series. Usually, a time series type ID is located in an `MI_FPARAM` structure. This function is useful when there is no easy access to an `MI_FPARAM` structure.

Returns

A pointer to an `MI_TYPEID` structure that contains the type ID of the specified time series. You must not free this value after use.

Related reference:

“The `ts_copy()` function” on page 9-16

“The `ts_create()` function” on page 9-17

“The `ts_create_with_metadata()` function” on page 9-18

“The `ts_open()` function” on page 9-44

The `ts_hide_elem()` function

The `ts_hide_elem()` function marks the element at the given time stamp as invisible to a scan unless `TS_SCAN_HIDDEN` is set.

Syntax

```
ts_timeseries  
ts_hide_elem(ts_tsdesc *tsdesc,  
            mi_datetime *tstamp)
```

tsdesc The time series descriptor returned by `ts_open()` for the source time series.

tstamp The time stamp to be made invisible to the scan.

Description

When an element is hidden, element retrieval API functions such as `ts_elem()` and `ts_nth_elem()` return the hidden element; however, their *STATUS* argument has the `TS_NULL_HIDDEN` bit set. The values for the element's *STATUS* argument are:

- If *STATUS* is `TS_NULL_HIDDEN`, the element is hidden.
- If *STATUS* is `TS_NULL_NOTALLOCED`, the element is `NULL`.
- If *STATUS* is both `TS_NULL_HIDDEN` and `TS_NULL_NOTALLOCED`, the element is both hidden and `NULL`.
- If *STATUS* is 0 (zero), the element is not hidden and is not `NULL`.

The `TS_ELEM_HIDDEN` and `TS_ELEM_NULL` macros are provided to check the value of *STATUS*.

Hidden elements cannot be modified; they must be revealed first using **`ts_reveal_elem()`**.

The equivalent SQL function is **HideElem**.

Returns

The modified time series. If there is no element at the given time stamp, an error is raised.

Related reference:

“HideElem function” on page 7-67

“The `ts_elem()` function” on page 9-22

“The `TS_ELEM_HIDDEN` macro” on page 9-23

“The `TS_ELEM_NULL` macro” on page 9-24

“The `ts_reveal_elem()` function” on page 9-50

“The `ts_previous_valid()` function” on page 9-45

The `ts_index()` function

The **`ts_index()`** function converts from a time stamp to an index (offset) for a regular time series.

Syntax

```
mi_integer
ts_index(ts_tsdesc *tsdesc,
         mi_datetime *tstamp)
```

tsdesc The time series descriptor returned by **`ts_open()`**.

tstamp The time stamp to convert.

Description

Consider a time series that starts on Monday, January 1 and keeps track of weekdays. Calling **`ts_index()`** with a time stamp argument that corresponds to Monday, January 1, would return 0; a time stamp argument corresponding to Tuesday, January 2, would return 1; a time stamp argument corresponding to Monday, January 8, would return 5; and so on.

The equivalent SQL function is **GetIndex**.

Returns

An offset into the time series. If the time stamp falls before the time series origin, or if it is not a valid point in the calendar, -1 is returned; otherwise, the return value is always a positive integer.

Related reference:

"GetIndex function" on page 7-55

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

"The ts_nth_elem() function" on page 9-43

"The ts_put_nth_elem() function" on page 9-48

"The ts_time() function" on page 9-51

The ts_ins_elem() function

The **ts_ins_elem()** function puts an element into an existing time series at a given timepoint.

Syntax

```
ts_timeseries *
ts_ins_elem(ts_tsdesc  *tsdesc,
            ts_tselem  tselem,
            mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be modified, returned by **ts_open()**.

tselem The element to add.

tstamp The timepoint at which to add the element. The time stamp column of the *tselem* is ignored.

Description

The equivalent SQL function is **InsElem**.

Returns

The original time series with the new element added. If the time stamp is not a valid timepoint in the time series, an error is raised. If there is already an element at the given time stamp, an error is raised.

Related reference:

"InsElem function" on page 7-69

"The ts_del_elem() function" on page 9-22

"The ts_elem() function" on page 9-22

"The ts_free() procedure" on page 9-26

"The ts_make_elem() function" on page 9-38

"The ts_make_elem_with_buf() function" on page 9-39

"The ts_put_elem() function" on page 9-46

"The ts_upd_elem() function" on page 9-54

"The ts_put_elem_no_dups() function" on page 9-47

The TS_IS_INCONTAINER macro

The **TS_IS_INCONTAINER** macro determines whether the time series data is stored in a container.

Syntax

```
TS_IS_INCONTAINER((ts_timeseries *) ts)
```

ts A pointer to a time series.

Returns

This function returns nonzero if the time series data is in a container, rather than in memory or in a row.

The TS_IS_IRREGULAR macro

The TS_IS_IRREGULAR macro determines whether the given time series is irregular.

Syntax

```
TS_IS_IRREGULAR((ts_timeseries *) ts)
```

ts A pointer to a time series.

Returns

A nonzero value if the given time series is irregular; otherwise, 0 is returned.

Related reference:

“The ts_get_flags() function” on page 9-30

The ts_last_elem() function

The **ts_last_elem()** function returns the last element from a time series.

Syntax

```
ts_tselem  
ts_last_elem(ts_tsdesc *tsdesc,  
             mi_integer *STATUS,  
             mi_integer *off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

STATUS

A pointer to a **mi_integer** value. See “The ts_hide_elem() function” on page 9-34 for a description of *STATUS*.

off If the time series is regular, *off* is set to the offset of the returned element. If the time series is irregular, or if the time series is empty, *off* is set to -1. This argument can be passed in as NULL.

Description

This function fills in *off* with the element's offset if *off* is not NULL and the time series is regular, and it sets *STATUS* to indicate if the element is NULL or hidden.

The equivalent SQL function is **GetLastElem**.

Returns

The last element of the specified time series, its offset, and whether it is NULL or hidden. If the time series is irregular, the offset is set to -1. This value must not be freed by the caller. The element is overwritten after two calls to fetch elements with this *tsdesc* (time series descriptor).

Related reference:

“GetLastElem function” on page 7-56

“The ts_elem() function” on page 9-22

“The TS_ELEM_HIDDEN macro” on page 9-23
“The TS_ELEM_NULL macro” on page 9-24
“The ts_nth_elem() function” on page 9-43
“The ts_upd_elem() function” on page 9-54

The ts_last_valid() function

The **ts_last_valid()** function extracts the entry for a particular timepoint.

Syntax

```
ts_tselem  
ts_last_valid(ts_tsdesc *tsdesc,  
              mi_datetime *tstamp,  
              mi_integer *STATUS,  
              mi_integer *off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

tstamp The time stamp of interest.

STATUS

A pointer to an **mi_integer** value. See “The ts_hide_elem() function” on page 9-34 for a description of *STATUS*.

off If the time series is regular, *off* is set to the offset of the returned element. If the time series is irregular, or if the time series is empty, *off* is set to -1. This argument can be passed as NULL.

Description

For regular time series, this function returns the first element with a time stamp less than or equal to *tstamp*. For irregular time series, it returns the latest element at or preceding the given time stamp.

Returns

The nearest element at or before the given time stamp. If there is no such element before the time stamp, NULL is returned.

NULL is returned if:

- The element at the timepoint is NULL and the time series is regular.
- The timepoint is before the origin.
- The time series is irregular and there are no elements at or before the given time stamp.

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

Related reference:

“GetLastValid function” on page 7-58
“The ts_previous_valid() function” on page 9-45

The ts_make_elem() function

The **ts_make_elem()** function makes an element from an array of values and nulls. Each array has one value for each column in the element.

Syntax

```
ts_tselem  
ts_make_elem(ts_tsdesc *tsdesc,  
             MI_DATUM  *values,  
             mi_boolean *nulls,  
             mi_integer *off)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

values An array of data to be placed in the element. Data that is by value is placed in the array, and data that is by reference stores pointers.

nulls Stores columns in the element that should be NULL.

off For a regular time series, *off* contains the offset of the element on return. For an irregular time series, *off* is set to -1. This argument can be NULL.

Returns

An element and its offset. If *tsdesc* is a descriptor for a regular time series, the time stamp column in the element is set to NULL; if *tsdesc* is a descriptor for an irregular time series, the time stamp column is set to whatever was in *values*[0]. This element must be freed by the caller using **ts_free_elem()**.

Related reference:

"The **ts_elem_to_row()** function" on page 9-24

"The **ts_free_elem()** procedure" on page 9-26

"The **ts_ins_elem()** function" on page 9-36

"The **ts_elem()** function" on page 9-22

"The **ts_make_elem_with_buf()** function"

"The **ts_put_elem()** function" on page 9-46

"The **ts_put_elem_no_dups()** function" on page 9-47

"The **ts_put_last_elem()** function" on page 9-48

"The **ts_put_nth_elem()** function" on page 9-48

"The **ts_row_to_elem()** function" on page 9-50

"The **ts_upd_elem()** function" on page 9-54

The **ts_make_elem_with_buf()** function

The **ts_make_elem_with_buf()** function creates a time series element using the buffer in an existing time series element. The initial data in the element is overwritten.

Syntax

```
ts_tselem  
ts_make_elem_with_buf(ts_tsdesc *tsdesc,  
                     MI_DATUM  *values,  
                     mi_boolean *nulls,  
                     mi_integer *off,  
                     ts_tselem elem)
```

tsdesc The descriptor for a time series returned by **ts_open()**.

values An array of data to be placed in the element. Data that is by value is placed in the array, and data that is by reference stores pointers.

nulls Stores which columns in the element should be NULL.

- off* For a regular time series, *off* contains the offset of the element on return. For an irregular time series, *off* is set to -1. This argument can be NULL.
- elem* The time series element to be overwritten. It must agree in type with the subtype of the time series. If this argument is NULL, a new element is created.

Returns

A time series element. If the *elem* argument is non-null, that is returned containing the new values. If the *elem* argument is NULL, a new time series element is returned.

Related reference:

"The `ts_elem_to_row()` function" on page 9-24

"The `ts_free_elem()` procedure" on page 9-26

"The `ts_ins_elem()` function" on page 9-36

"The `ts_make_elem()` function" on page 9-38

"The `ts_put_last_elem()` function" on page 9-48

"The `ts_upd_elem()` function" on page 9-54

The `ts_make_stamp()` function

The `ts_make_stamp()` function constructs a time stamp from the year, month, day, hour, minute, second, and microsecond values and puts them into the `mi_datetime` pointed to by the *dt* argument.

Syntax

```
mi_datetime *
ts_make_stamp (MI_CONNECTION *conn,
               mi_datetime *dt,
               mi_integer  year,
               mi_integer  month,
               mi_integer  day,
               mi_integer  hour,
               mi_integer  minute,
               mi_integer  second,
               mi_integer  ms)
```

conn A valid DataBlade API connection.

dt The time stamp to fill in. The caller should supply the buffer.

year The year to put into the returned `mi_datetime`.

month The month to put into the returned `mi_datetime`.

day The day to put into the returned `mi_datetime`.

hour The hour to put into the returned `mi_datetime`.

minute The minute to put into the returned `mi_datetime`.

second The second to put into the returned `mi_datetime`.

ms The microsecond to put into the returned `mi_datetime`.

Returns

A pointer to the same `mi_datetime` structure that was passed in.

Related reference:

"The `ts_get_stamp_fields()` procedure" on page 9-32

The `ts_nelems()` function

The `ts_nelems()` function returns the number of elements in the time series.

Syntax

```
mi_integer  
ts_nelems(ts_tsdesc *tsdesc)
```

tsdesc The time series descriptor returned by `ts_open()`.

Description

The equivalent SQL function is `GetNelems`.

Returns

The number of elements in the time series.

Related reference:

“ClipGetCount function” on page 7-37

“GetNelems function” on page 7-60

The `ts_next()` function

After a scan has been started with `ts_begin_scan()`, elements can be retrieved from the time series with `ts_next()`.

Syntax

```
mi_integer  
ts_next(ts_tscan *tscan,  
        ts_tselem *tselem)
```

tscan The specified scan.

tselem A pointer to an element that `ts_next()` fills in.

Description

On return, the `ts_tselem` contains the next element in the time series, if there is one.

When `ts_tselem` is valid, it can be passed to other routines in the time series API, such as `ts_put_elem()`, `ts_get_col_by_name()`, and `ts_get_col_by_number()`.

Returns

TS_SCAN_ELEM

The *tselem* parameter contains a valid element.

TS_SCAN_NULL

The value in the element was NULL or hidden; if *tselem* is not NULL, then the element was hidden; otherwise, the element was NULL.

TS_SCAN_EOS

The scan has completed; *tselem* is not valid.

The return value must not be freed by the caller; it is freed when the scan is ended. It is overwritten after two `ts_next()` calls.

Example

See the `ts_interp()` function, Appendix A, “The Interp function example,” on page A-1, for an example of `ts_next()`.

Related reference:

- “The `ts_begin_scan()` function” on page 9-7
- “The `TS_ELEM_HIDDEN` macro” on page 9-23
- “The `TS_ELEM_NULL` macro” on page 9-24
- “The `ts_first_elem()` function” on page 9-25
- “The `ts_next_valid()` function”
- “The `ts_previous_valid()` function” on page 9-45
- “The `ts_put_elem()` function” on page 9-46
- “The `ts_put_elem_no_dups()` function” on page 9-47
- “The `ts_put_last_elem()` function” on page 9-48
- “The `ts_put_nth_elem()` function” on page 9-48
- “The `ts_upd_elem()` function” on page 9-54

The `ts_next_valid()` function

The `ts_next_valid()` function returns the nearest entry after a given time stamp.

Syntax

```
ts_tselem  
ts_next_valid(ts_tsdesc  *tsdesc,  
              mi_datetime *tstamp,  
              mi_integer  *STATUS,  
              mi_integer  *off)
```

tsdesc The time series descriptor returned by `ts_open()`.

tstamp Points to the time stamp that precedes the element returned.

STATUS

Points to an **mi_integer** value that is filled in on return. See the discussion of `ts_hide_elem()` (“The `ts_hide_elem()` function” on page 9-34) for a description of *STATUS*.

off For regular time series, *off* points to an **mi_integer** value that is filled in on return with the offset of the returned element. For irregular time series, *off* is set to -1. Can be NULL.

Description

For regular time series, this function returns the element at the calendar's earliest valid timepoint following the given time stamp. For irregular time series, it returns the earliest element following the given time stamp.

Tip: The `ts_next_valid()` function is less efficient than `ts_next()`, so it is better to iterate through a time series using `ts_begin_scan()` and `ts_next()` rather than using `ts_first_elem()` and `ts_next_valid()`.

The equivalent SQL function is **GetNextValid**.

Returns

The element following the given time stamp. If no valid element exists or the time series is regular and the next valid interval contains a null element, NULL is returned. The value pointed to by *off* is either -1 if the time series is irregular or the offset of the element if the time series is regular. The element returned must not be freed by the caller. It is overwritten after two fetch calls.

See “The `ts_hide_elem()` function” on page 9-34 for an explanation of `STATUS`.

Related reference:

“`GetNextValid` function” on page 7-61

“The `TS_ELEM_HIDDEN` macro” on page 9-23

“The `TS_ELEM_NULL` macro” on page 9-24

“The `ts_first_elem()` function” on page 9-25

“`GetLastValid` function” on page 7-58

“The `ts_next()` function” on page 9-41

“The `ts_previous_valid()` function” on page 9-45

The `ts_nth_elem()` function

The `ts_nth_elem()` function returns the element at the *n*th position of the given time series.

Syntax

```
ts_tselem  
ts_nth_elem(ts_tsdesc *tsdesc,  
            mi_integer N,  
            mi_integer *STATUS)
```

tsdesc The descriptor returned by `ts_open()`.

N The time series offset or position to read the element from. This value must not be less than 0.

STATUS

A pointer to an `mi_integer` value that is set on return to indicate whether the element is NULL. See “The `ts_hide_elem()` function” on page 9-34 for a description of *STATUS*.

Description

The equivalent SQL function is `GetNthElem`.

Returns

The element at the *n*th position of the given time series, and whether it was NULL. This value must not be freed by the caller. It is overwritten after two fetch calls.

Related reference:

“`GetNthElem` function” on page 7-62

“The `ts_elem()` function” on page 9-22

“The `TS_ELEM_HIDDEN` macro” on page 9-23

“The `TS_ELEM_NULL` macro” on page 9-24

“The `ts_index()` function” on page 9-35

“The `ts_last_elem()` function” on page 9-37

The `ts_open()` function

The `ts_open()` function opens a time series.

Syntax

```
ts_tsdesc *  
ts_open(MI_CONNECTION *conn,  
        ts_timeseries *ts,  
        MI_TYPEID      *type_id,  
        mi_integer      flags)
```

conn A database connection. This argument is unused in the server.

ts The time series to open.

type_id The ID for the type of the time series to be opened. The ID is generally determined by looking in the MI_FPARAM structure.

flags Valid values for the *flags* parameter are defined in `tseries.h`.

The *flags* argument values

Valid values for the *flags* argument are defined in the file `tseries.h`. (the integer value you use for the *flags* argument is the sum of the desired values). Valid options are:

TSOPEN_RDWRITE

The default mode for opening a time series. Indicates that the time series can be read and written to.

TSOPEN_READ_HIDDEN

Indicates that hidden elements should be treated as if they are not hidden.

TSOPEN_READ_ONLY

Indicates that the time series can only be read.

TSOPEN_WRITE_HIDDEN

Allows hidden elements to be written to without first revealing the element.

TSOPEN_WRITE_AND_HIDE

Causes any elements written to a time series also to be marked as hidden.

TSOPEN_WRITE_AND_REVEAL

Reveals any hidden element that is written.

TSOPEN_NO_NULLS

Affects the way elements are returned that have never been allocated (TS_NULL_NOTALLOCATED). Usually, if an element has not been allocated, it is returned as NULL. If TSOPEN_NO_NULLS is set, an element that has each column set to NULL is returned instead.

These flags can be used in any combination except the following four combinations:

- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN and TSOPEN_WRITE_AND_REVEAL
- TSOPEN_WRITE_AND_REVEAL and TSOPEN_WRITE_AND_HIDE
- TSOPEN_WRITE_HIDDEN, TSOPEN_WRITE_AND_HIDE, and TSOPEN_WRITE_AND_REVEAL

The `TSOPEN_WRITE_HIDDEN`, `TSOPEN_WRITE_AND_REVEAL`, and `TSOPEN_WRITE_AND_HIDE` flags cannot be used with `TSOPEN_READ_HIDDEN`.

Description

Almost all other functions depend on this function being called first.

Use `ts_close` to close the time series.

Returns

A descriptor for the open time series.

Example

See the `ts_interp()` function, Appendix A, “The Interp function example,” on page A-1, for an example of `ts_open()`.

Related reference:

“The `ts_begin_scan()` function” on page 9-7

“The `ts_close()` function” on page 9-12

“The `ts_create()` function” on page 9-17

“The `ts_create_with_metadata()` function” on page 9-18

“The `ts_get_typeid()` function” on page 9-34

The `ts_previous_valid()` function

The `ts_previous_valid()` function returns the last element preceding the given time stamp.

Syntax

```
ts_tselem  
ts_previous_valid(ts_tsdsc  *tsdesc,  
                 mi_datetime *tstamp,  
                 mi_integer  *STATUS,  
                 mi_integer  *off)
```

tsdesc The time series descriptor returned by `ts_open()`.

tstamp Points to the time stamp that follows the element returned.

STATUS

Points to an **mi_integer** value that is filled in on return. If no element exists before the time stamp, or if the time stamp falls before the time series origin, *STATUS* is set to a nonzero value. See “The `ts_hide_elem()` function” on page 9-34 for a description of *STATUS*.

off For regular time series, *off* points to an **mi_integer** value that is filled in on return with the offset of the returned element. For irregular time series, *off* is set to -1. This argument can be passed as `NULL`.

Description

The equivalent SQL function is **GetPreviousValid**.

Returns

The element, if any, preceding the given time stamp. The element returned must not be freed by the caller. It is overwritten after two calls to fetch an element using this *tsdesc* (time series descriptor).

For irregular time series, if no valid element precedes the given time stamp, NULL is returned. NULL is also returned if the given time stamp is less than or equal to the origin of the time series.

Related reference:

"GetPreviousValid function" on page 7-65

"The TS_ELEM_HIDDEN macro" on page 9-23

"The TS_ELEM_NULL macro" on page 9-24

"The ts_last_valid() function" on page 9-38

"The ts_next_valid() function" on page 9-42

"The ts_hide_elem() function" on page 9-34

"The ts_next() function" on page 9-41

The ts_put_elem() function

The **ts_put_elem()** function puts new elements into an existing time series.

Syntax

```
ts_timeseries *  
ts_put_elem(ts_tsdesc  *tsdesc,  
            ts_tselem  tselem,  
            mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be modified, returned by **ts_open()**.

tselem The element to add.

tstamp The time stamp at which to put the element. The time stamp column of *tselem* is ignored.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

For regular time series, if there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

For irregular time series, if there is no data at the given timepoint, the new data is inserted. If there is data at the given timepoint, then the following algorithm is used to determine where to place the data:

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

The element passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The equivalent SQL function is **PutElem**.

Returns

The original time series with the element added.

Related reference:

“PutElem function” on page 7-77

“The ts_del_elem() function” on page 9-22

“The ts_get_ts() function” on page 9-33

“The ts_ins_elem() function” on page 9-36

“The ts_make_elem() function” on page 9-38

“The ts_elem() function” on page 9-22

“The ts_next() function” on page 9-41

“The ts_put_elem_no_dups() function”

“The ts_put_last_elem() function” on page 9-48

“The ts_upd_elem() function” on page 9-54

“The ts_put_ts() function” on page 9-49

The ts_put_elem_no_dups() function

The **ts_put_elem_no_dups()** function puts a new element into an existing time series. The element is inserted even if there is already an element with the given time stamp in the time series.

Syntax

```
ts_timeseries *  
ts_put_elem_no_dups(ts_tsdesc  *tsdesc,  
                   ts_tselem  tselem,  
                   mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be modified, returned by **ts_open()**.

tselem The element to add.

tstamp The time stamp at which to put the element. The time stamp column of *tselem* is ignored.

Description

If the time stamp is NULL, the data is appended to the time series (for regular time series) or an error is raised (for irregular time series).

If there is data at the given timepoint, it is updated with the new data; otherwise, the new data is inserted.

The element passed in must match the subtype of the time series.

Hidden elements cannot be updated.

The equivalent SQL function is **PutElemNoDups**.

Returns

The original time series with the element added.

Related reference:

“PutElemNoDups function” on page 7-79

"The `ts_put_elem()` function" on page 9-46
"The `ts_elem()` function" on page 9-22
"The `ts_get_ts()` function" on page 9-33
"The `ts_ins_elem()` function" on page 9-36
"The `ts_make_elem()` function" on page 9-38
"The `ts_next()` function" on page 9-41
"The `ts_put_last_elem()` function"
"The `ts_upd_elem()` function" on page 9-54

The `ts_put_last_elem()` function

The **`ts_put_last_elem()`** function puts new elements at the end of an existing regular time series.

Syntax

```
ts_timeseries *  
ts_put_last_elem(ts_tsdesc *tsdesc,  
                 ts_tselem tselem)
```

tsdesc The time series to be updated.

tselem The element to add; any time stamp in the element is ignored.

Returns

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

Related reference:

"The `ts_put_elem()` function" on page 9-46
"The `ts_put_elem_no_dups()` function" on page 9-47
"The `ts_elem()` function" on page 9-22
"The `ts_get_ts()` function" on page 9-33
"The `ts_make_elem()` function" on page 9-38
"The `ts_make_elem_with_buf()` function" on page 9-39
"The `ts_next()` function" on page 9-41

The `ts_put_nth_elem()` function

The **`ts_put_nth_elem()`** function puts new elements into an existing regular time series at a specified offset.

Syntax

```
ts_timeseries *  
ts_put_nth_elem(ts_tsdesc *tsdesc,  
                ts_tselem tselem,  
                mi_integer N)
```

tsdesc The time series to be updated.

tselem The element to add; any time stamp in the element is ignored.

N The offset, indicating where the element to add should be placed. Offsets start at 0.

Returns

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

Related reference:

“The `ts_index()` function” on page 9-35

“The `ts_elem()` function” on page 9-22

“The `ts_get_ts()` function” on page 9-33

“The `ts_make_elem()` function” on page 9-38

“The `ts_next()` function” on page 9-41

The `ts_put_ts()` function

The `ts_put_ts()` function updates a destination time series with the elements from the source time series.

Syntax

```
ts_timeseries *
ts_put_ts(ts_tsdesc *src_tsdesc,
          ts_tsdesc *dst_tsdesc,
          mi_boolean nodups)
```

src_tsdesc

The source time series descriptor.

dst_tsdesc

The destination time series descriptor.

nodups Determines whether to overwrite an element in the destination time series if there is an element at the same time stamp in the source time series. This argument is ignored if the destination time series is regular.

Description

The two descriptors must meet the following conditions:

- The origin of the source time series must be after or equal to that of the destination time series.
- The two time series must have the same calendar.

If *nodups* is `MI_TRUE`, the element from the source time series overwrites the element in the destination time series. For irregular time series, if *nodups* is `MI_FALSE` and there is already a value at the existing timepoint, the update is made at the next microsecond after the last element in the given second. If the last microsecond in the second already contains a value, an error is raised.

The equivalent SQL function is **PutTimeSeries**.

Returns

The time series associated with the destination time series descriptor.

Related reference:

“PutTimeSeries function” on page 7-82

“The `ts_put_elem()` function” on page 9-46

The `ts_reveal_elem()` function

The `ts_reveal_elem()` function makes the element at a given time stamp visible to a scan. It reverses the effect of `ts_hide_elem()`.

Syntax

```
ts_timeseries  
ts_reveal_elem(ts_tsdesc  *tsdesc,  
               mi_datetime *tstamp)
```

ts_desc The time series descriptor returned by `ts_open()` for the source time series.

tstamp The time stamp to be made visible to the scan.

Description

The equivalent SQL function is **RevealElem**.

Returns

The modified time series. No error is raised if there is no element at the given time stamp.

Related reference:

“HideElem function” on page 7-67

“The `ts_hide_elem()` function” on page 9-34

“RevealElem function” on page 7-83

The `ts_row_to_elem()` function

The `ts_row_to_elem()` function converts an `MI_ROW` structure into a new `ts_tselem` structure. The new element does not overwrite elements returned by any other time series API function.

Syntax

```
ts_tselem  
ts_row_to_elem(ts_tsdesc  *tsdesc,  
               MI_ROW      *row,  
               mi_integer  *offset_ptr)
```

tsdesc The descriptor for a time series returned by `ts_open()`.

row A pointer to an `MI_ROW` structure. The row must have the same type as the subtype of the time series.

offset_ptr

If the time series is regular, the offset of the element in the time series is returned in *offset_ptr*. In this case, column 0 (the time stamp column) must not be NULL. If the time series is irregular, -1 is returned in *offset_ptr*.

The *offset_ptr* argument can be NULL. In this case, calendar computations are avoided and column 0 can be NULL.

Returns

An element and its offset. If the time series is regular, column 0 (the time stamp column) of the element is NULL.

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

Related reference:

“The `ts_elem_to_row()` function” on page 9-24

“The `ts_free_elem()` procedure” on page 9-26

“The `ts_make_elem()` function” on page 9-38

The `ts_time()` function

The `ts_time()` function converts a regular time series offset to a time stamp.

Syntax

```
mi_datetime *  
ts_time(ts_tsdesc *tsdesc,  
        mi_integer N)
```

ts_desc The time series descriptor returned by `ts_open()` for the source time series.

N The offset to convert. Negative values are allowed.

Description

For example, for a daily time series that starts on Monday, January 1, with a five-day-a-week pattern starting on Monday, this function returns Monday, January 1, when the argument is set to 0; Tuesday, January 2, when the argument is set to 1; Monday, January 8, when the argument is 5; and so on.

The equivalent SQL function is **GetStamp**.

Returns

The time stamp corresponding to the offset. This value must be freed by the user with `mi_free()`.

Related reference:

“GetStamp function” on page 7-66

“The `ts_cal_range()` function” on page 9-10

“The `ts_cal_range_index()` function” on page 9-11

“The `ts_index()` function” on page 9-35

The `ts_tstamp_difference()` function

The `ts_tstamp_difference()` function subtracts one date from another and returns the number of complete intervals between the two dates.

Syntax

```
mi_integer  
ts_tstamp_difference(mi_datetime *date1,  
                    mi_datetime *date2,  
                    mi_integer interval)
```

date1 The first date.

date2 The date to subtract from the first date.

interval
The interval, as described next.

Description

Before the difference is calculated, both time stamps are truncated to the given interval. For example, if the interval is an hour and the first date is 2011-01-03 01:02:03.12345, its truncated value is 2011-01-03 01:00:00.00000.

Valid values for the *interval* parameter can be found in `tseries.h`. They are:

- TS_SECOND
- TS_MINUTE
- TS_HOUR
- TS_DAY
- TS_WEEK
- TS_MONTH
- TS_YEAR

Returns

The number of intervals of the type you specify between the two dates.

Example

For example, if the interval is *day* and the dates are 2011-01-01 00:00:00.00000 and 2011-01-01 00:00:00.00001, the result is 0. If the dates are 2011-01-01 00:00:00.00000 and 2011-01-02 00:10:00.12345, the result is 1.

Related reference:

“The `ts_tstamp_minus()` function”

“The `ts_tstamp_plus()` function” on page 9-53

The `ts_tstamp_minus()` function

The `ts_tstamp_minus()` function returns a time stamp at a specified number of intervals before a starting date you specify.

Syntax

```
mi_datetime *  
ts_tstamp_minus(mi_datetime *startdate,  
                mi_integer  cnt,  
                mi_integer  interval,  
                mi_datetime *result)
```

startdate

The date to start from.

cnt

The number of intervals to subtract from the start date.

interval

The interval, as described next.

result

The resulting date.

Description

Valid values for the *interval* parameter can be found in `tseries.h`. They are:

- TS_SECOND
- TS_MINUTE

- TS_HOUR
- TS_DAY
- TS_WEEK
- TS_MONTH
- TS_YEAR

If the *result* parameter is NULL, then a result **mi_datetime** structure is allocated and returned; otherwise, the return value is the given *result* parameter.

Returns

The time stamp at the specified number of intervals before the start date.

Related reference:

“The `ts_tstamp_difference()` function” on page 9-51

“The `ts_tstamp_plus()` function”

The `ts_tstamp_plus()` function

The **`ts_tstamp_plus()`** function returns a time stamp at a specified number of intervals after a starting date you specify.

Syntax

```
mi_datetime *
ts_tstamp_plus(mi_datetime  *startdate,
               mi_integer   cnt,
               mi_integer   interval,
               mi_datetime  *result)
```

startdate

The date to start from.

cnt

The number of intervals to add to the start date.

interval

The interval, as described next.

result

The resulting date.

Description

Valid values for the *interval* parameter can be found in `tseries.h`. They are:

- TS_SECOND
- TS_MINUTE
- TS_HOUR
- TS_DAY
- TS_WEEK
- TS_MONTH
- TS_YEAR

If the *result* parameter is NULL, then a result **mi_datetime** structure is allocated and returned; otherwise, the return value is the given *result* parameter.

Returns

The time stamp at the specified number of intervals after the start date.

Related reference:

“The `ts_tstamp_difference()` function” on page 9-51

“The `ts_tstamp_minus()` function” on page 9-52

The `ts_update_metadata()` function

The `ts_update_metadata()` function adds the supplied user-defined metadata to the specified time series.

Syntax

```
ts_timeseries *  
ts_update_metadata(ts_timeseries *ts,  
                  mi_lvarchar  *metadata,  
                  MI_TYPEID    *metadata_typeid)
```

ts The time series for which to update metadata.

metadata

 The metadata to add to the time series. Can be NULL.

metadata_typeid

 The type ID of the metadata.

Description

The equivalent SQL function is **UpdMetaData**.

Returns

A copy of the specified time series updated to contain the supplied metadata, or if the *metadata* argument is NULL, a copy of the specified time series with the metadata removed.

Related reference:

“GetMetaData function” on page 7-59

“UpdMetaData function” on page 7-159

“GetMetaTypeName function” on page 7-59

“TSCreate function” on page 7-116

“TSCreateIrr function” on page 7-118

“The `ts_create_with_metadata()` function” on page 9-18

“The `ts_get_metadata()` function” on page 9-31

The `ts_upd_elem()` function

The `ts_upd_elem()` function updates an element in an existing time series at a given timepoint.

Syntax

```
ts_timeseries *  
ts_upd_elem(ts_tsdesc  *tsdesc,  
            ts_tselem  tselem,  
            mi_datetime *tstamp)
```

tsdesc A descriptor of the time series to be updated, returned by `ts_open()`.

tselem The element to update.

tstamp The timepoint at which to update the element.

Description

There must already be an element at the given time stamp. For irregular time series, hidden elements cannot be updated.

The equivalent SQL function is **UpdElem**.

Returns

An updated copy of the original time series.

Related reference:

“The `ts_del_elem()` function” on page 9-22

“The `ts_ins_elem()` function” on page 9-36

“The `ts_put_elem()` function” on page 9-46

“The `ts_put_elem_no_dups()` function” on page 9-47

“UpdElem function” on page 7-159

“The `ts_elem()` function” on page 9-22

“The `ts_last_elem()` function” on page 9-37

“The `ts_make_elem()` function” on page 9-38

“The `ts_make_elem_with_buf()` function” on page 9-39

“The `ts_next()` function” on page 9-41

Appendix A. The Interp function example

The **Interp** function is an example of a server function that uses the time series API. This function interpolates between values of a regular time series to fill in null elements.

This function does not handle individual null columns. It assumes that all columns are of type FLOAT.

Interp might be used as follows:

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

This example, along with many others, is supplied in the \$INFORMIXDIR/extend/TimeSeries.*version* directory.

To use the **Interp** function, create a server function:

```
create function Interp(TimeSeries) returns TimeSeries
external name '/tmp/Interpolate.bld(ts_interp)'
language c not variant;
```

You can now use the **Interp** function in a DB-Access statement. For example, consider the difference in output between the following two queries (the output has been reformatted; the actual output you would see would not be in tabular format):

```
select stock_data from daily_stocks where stock_name = 'IBM';
```

2011-01-03 00:00:00	1	1	1	1
2011-01-04 00:00:00	2	2	2	2
NULL				
2011-01-06 00:00:00	3	3	3	3

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

2011-01-03 00:00:00	1	1	1	1
2011-01-04 00:00:00	2	2	2	2
2011-01-05 00:00:00	2.5	2.5	2.5	2.5
2011-01-06 00:00:00	3	3	3	3

```
/*
 * SETUP:
 * create function Interp(TimeSeries) returns TimeSeries
 * external name 'Interpolate.so(ts_interp)'
 * language c not variant;
 *
 *
 * USAGE:
 * select Interp(stock_data) from daily_stocks where stock_id = 901;
 */
```

```
#include <stdio.h>
#include <mi.h>
#include <tseries.h>
```

```
# define TS_MAX_COLS    100
# define DATATYPE       "smallfloat"
```

```
/*
```

```

* This example interpolates between values to fill in null elements.
* It assumes that all columns are of type smallfloat and that there
are
* less than 100 columns in each element.
*/

ts_timeseries *
ts_interp(tsPtr, fParamPtr)
    ts_timeseries    *tsPtr;
    MI_FPARAM        *fParamPtr;
{
    ts_tsdesc        *descPtr;
    ts_tselem        tselem;
    ts_tscan         *scan;
    MI_CONNECTION     *conn;
    ts_typeinfo       *typeinfo;
    int               scancode;
    mi_real           *values[TS_MAX_COLS];
    mi_real           lastValues[TS_MAX_COLS], newValues[TS_MAX_COLS];
    mi_boolean        nulls[TS_MAX_COLS];
    mi_integer        minElem, curElem, elem;
    mi_integer        i;
    mi_boolean        noneYet;
    mi_integer        ncols;
    char              strbuf[100];

    /* get a connection for libmi */
    conn = mi_open(NULL, NULL, NULL);

    /* open a descriptor for the timeseries */
    descPtr = ts_open(conn, tsPtr, mi_fp_retype(fParamPtr, 0), 0);

    if ((ncols = (mi_integer) mi_fp_funcstate(fParamPtr)) == 0) {
        ncols = ts_col_cnt(descPtr);

        if (ncols > TS_MAX_COLS) {
            sprintf(strbuf, "Timeseries elements have too many columns,
100 is
the max, got %d instead.", ncols);
            mi_db_error_raise(NULL, MI_FATAL, strbuf, 0);
        }

        for (i = 1; i < ncols; i++) {
            typeinfo = ts_colinfo_number(descPtr, i);

            if (strlen(typeinfo->ti_typename) != strlen(DATATYPE) &&
memcmp(typeinfo->ti_typename, DATATYPE, strlen(DATATYPE)) !=
0){
                sprintf(strbuf, "column was not a %s, got %s instead.", DATATYPE,
typeinfo->ti_typename);
                mi_db_error_raise(NULL, MI_FATAL, strbuf, 0);
            }
        }

        mi_fp_setfuncstate(fParamPtr, (void *) ncols);

        noneYet = MI_TRUE;
        minElem = -1;
        curElem = 0;
        /* begin a scan of the whole timeseries */
        scan = ts_begin_scan(descPtr, 0, NULL, NULL);
        while ((scancode = ts_next(scan, &tselem)) != TS_SCAN_EOS)
        {
            switch(scancode) {
                case TS_SCAN_ELEM:

```

```

/* if this element is not null expand its values */
noneYet = MI_FALSE;
ts_get_all_cols(descPtr, tselem, (void **) values, nulls, curElem);
if (minElem == -1) {
    /* save each element */
    for (i = 1; i < ncols; i++)
        lastValues[i] = *values[i];
}
else {
    /* calculate the average */
    for (i = 1; i < ncols; i++) {
        newValues[i] = (*values[i] + lastValues[i])/2.0;
        lastValues[i] = *values[i];
        values[i] = &newValues[i];
    }

    /* update the missing elements */

    tselem = ts_make_elem(descPtr, (void **) values, nulls, &elem);
    for (elem = minElem; elem < curElem; elem++)
        ts_put_nth_elem(descPtr, tselem, elem);

    minElem = -1;
}

break;
case TS_SCAN_NULL:
if (noneYet)
    break;
/* remember the first null element */
if (minElem == -1)
    minElem = curElem;
break;
}

curElem++;
}
ts_end_scan(scan);
ts_close(descPtr);
return(tsPtr);
}

```

Appendix B. The TSIncLoad procedure example

The **TSIncLoad** procedure loads data into a database that contains a time series of corporate bond prices.

The **TSIncLoad** procedure loads time-variant data from a file into a table that contains time series. It assumes that the table is already populated with the time-invariant data. If the table already has time series data, the new data overwrites the old data or is appended to the existing time series, depending on the time stamps.

To set up the **TSIncLoad** example, create the procedure, the row subtype, and the database table as shown in the following example. The code for this example is in the `$INFORMIXDIR/extend/TimeSeries.version/examples` directory.

```
create procedure if not exists TSIncLoad( table_name lvvarchar,
                                         file_name lvvarchar,
                                         calendar_name lvvarchar,
                                         origin datetime year to fraction(5),
                                         threshold integer,
                                         regular boolean,
                                         container_name lvvarchar,
                                         nelems integer)
external name '/tmp/Loader.bld(TSIncLoad)'
language C;

create row type day_info (
    ValueDate      datetime year to fraction(5),
    carryover      char(1),
    spread          integer,
    pricing_bmk_id integer,
    price           float,
    yield           float,
    priority        char(1) );

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

create index if not exists corporatesIdx on corporates( Secid);

execute procedure TSContainerCreate('ctnr_daily', 'rootdbs',
    'day_info', 0, 0);

insert into corporates values ( 25000006, 'container(ctnr_daily)',
    origin(2011-01-03 00:00:00.00000),
    calendar(daycal), threshold(0));

execute procedure TSIncLoad('corporates',
    '/tmp/daily.dat',
    'daycal',
    '2011-01-03 00:00:00.00000',
    0,
    't',
    'ctnr_daily',
    1);
```

Any name can be used for the **corporates** table. The **corporates** table can have any number of columns in addition to the **Secid** and **series** columns.

Each line of the data file has the following format:

Secid year-mon-day carryover spread pricing_bmk_id price yield priority

For example:

25000006 2010-1-7 m 2 12 2.2000000000 22.2 6

You can run the **TSIncLoad** procedure with an SQL statement like:

```
execute procedure TSIncLoad( 'corporates',
                             'data_file_name',
                             'cal_name',
                             '2010-1-1',
                             20,
                             't',
                             'container-name',
                             1);

#include <ctype.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "datetime.h"
#include "mi.h"
#include "tseries.h"

#define DAY_INFO_TYPE_NAME "day_info"
#define DAILY_COL_COUNT 7

typedef struct
{
    mi_integer fd;
    mi_unsigned_integer flags;
#define LDBUF_LAST_CHAR_EOL 0x1

    mi_integer buf_index;
    mi_integer buf_len;
    mi_integer line_no;
    mi_lvarchar *file_name;
    mi_string data[2048];
}
FILE_BUF;

#define STREAM_EOF (-1)

typedef struct sec_entry_s
{
    mi_integer sec_id;
    ts_tsdesc *tsdesc;
    int in_row; /* Indicates whether the time series is stored in
row. */
    struct sec_entry_s *next;
}
sec_entry_t;

typedef struct
{
    mi_lvarchar *table_name;
    MI_TYPEID ts_typeid; /* The type id of timeseries(day_info) */
    mi_string *calendar_name;
    mi_datetime *origin;
    mi_integer threshold;
    mi_boolean regular;
    mi_string *container_name;
    mi_integer nelems; /* For created time series. */
```

```

mi_integer hash_size;
MI_CONNECTION *conn;
sec_entry_t **hash;
/* Value buffers -- only allocated once. */
MI_DATUM col_data[ DAILY_COL_COUNT];
mi_boolean col_is_null[ DAILY_COL_COUNT];
char *carryover;
char *priority;
mi_double_precision price, yield;

mi_integer instances_created;
/* A count of the number of tsinstancetable entries added. Used
to
    * decide when to update statistics on this table.
    */
MI_SAVE_SET *save_set;
}
loader_context_t;

/*
*****
* name:      init_context
*
* purpose:   Initialize the loader context structure.
*
* notes:
*****
*/
static void
init_context( mi_lvarchar *table_name,
              mi_lvarchar *calendar_name,
              mi_datetime *origin,
              mi_integer threshold,
              mi_boolean regular,
              mi_lvarchar *container_name,
              mi_integer nelems,
              loader_context_t *context_ptr)
{
    mi_string buf[256];
    mi_integer table_name_len = mi_get_varlen( table_name);
    MI_ROW *row = NULL;
    MI_DATUM retbuf = 0;
    mi_integer retlen = 0;
    mi_lvarchar *typename = NULL;
    MI_TYPEID *typeid = NULL;
    mi_integer err = 0;

    if( table_name_len > IDENTSIZE)
mi_db_error_raise( NULL, MI_EXCEPTION, "The table name is too long");

    memset( context_ptr, 0, sizeof( *context_ptr));
    context_ptr->conn = mi_open( NULL, NULL, NULL);

    typename = mi_string_to_lvarchar
        ( "timeseries(" DAY_INFO_TYPE_NAME ")");
    typeid = mi_typename_to_id( context_ptr->conn, typename);
    mi_var_free( typename);
    if( NULL == typeid)
mi_db_error_raise( NULL, MI_EXCEPTION,
    "Type timeseries(" DAY_INFO_TYPE_NAME ") not defined.");
    context_ptr->ts_typeid = *typeid;

    context_ptr->table_name = table_name;

    context_ptr->calendar_name = mi_lvarchar_to_string( calendar_name);

```

```

context_ptr->origin = origin;
context_ptr->threshold = threshold;
context_ptr->regular = regular;
context_ptr->container_name = mi_lvarchar_to_string( container_name);
context_ptr->nelems = nelems;

/* Use the size (count) of the table as the hash table size. */
sprintf( buf, "select count(*) from %.*s;",
        table_name_len,
        mi_get_vardata( table_name));
if( MI_OK != mi_exec( context_ptr->conn, buf, MI_QUERY_BINARY))
mi_db_error_raise( NULL, MI_EXCEPTION, "mi_exec failed");
if( MI_ROWS != mi_get_result( context_ptr->conn))
{
sprintf( buf, "Could not get size of %.*s table.",
        table_name_len,
        mi_get_vardata( table_name));
mi_db_error_raise( NULL, MI_EXCEPTION, buf);
}
if( NULL == (row = mi_next_row( context_ptr->conn, &err)))
mi_db_error_raise( NULL, MI_EXCEPTION, "mi_next_row failed");
if( MI_NORMAL_VALUE != mi_value( row, 0, &retbuf, &retlen)
|| 0 != dectoint( (mi_decimal *) retbuf, &context_ptr->hash_size))
context_ptr->hash_size = 256;
(void) mi_query_finish( context_ptr->conn);
context_ptr->hash
= mi_zalloc( context_ptr->hash_size*sizeof( *context_ptr->hash));

context_ptr->col_data[1] = (MI_DATUM) mi_new_var(1); /* carryover
*/
context_ptr->col_data[6] = (MI_DATUM) mi_new_var(1); /* priority
*/

if( NULL == context_ptr->hash
|| NULL == context_ptr->col_data[1]
|| NULL == context_ptr->col_data[6])
mi_db_error_raise( NULL, MI_EXCEPTION, "Not enough memory.");

context_ptr->carryover
= mi_get_vardata( (mi_lvarchar *) context_ptr->col_data[1]);
context_ptr->col_data[4] = (MI_DATUM) &context_ptr->price;
context_ptr->col_data[5] = (MI_DATUM) &context_ptr->yield;
context_ptr->priority
= mi_get_vardata( (mi_lvarchar *) context_ptr->col_data[6]);

context_ptr->save_set = mi_save_set_create( context_ptr->conn);
} /* End of init_context. */

/*
*****
* name:      close_context
*
* purpose:   Close the context structure. Free up all allocated memory.
*
*****
*/
static void
close_context( loader_context_t *context_ptr)
{
    mi_free( context_ptr->hash);
    context_ptr->hash = NULL;
    context_ptr->hash_size = 0;

    mi_var_free( (mi_lvarchar *) context_ptr->col_data[1]);
    mi_var_free( (mi_lvarchar *) context_ptr->col_data[6]);
    context_ptr->col_data[1] = context_ptr->col_data[6] = 0;

```

```

context_ptr->carryover = context_ptr->priority = NULL;

(void) mi_save_set_destroy( context_ptr->save_set);
context_ptr->save_set = NULL;

(void) mi_close( context_ptr->conn);

mi_free( context_ptr->calendar_name);
context_ptr->calendar_name = NULL;
mi_free( context_ptr->container_name);
context_ptr->container_name = NULL;

context_ptr->conn = NULL;
} /* End of close_context. */

/*
*****
* name:      update_series
*
* purpose:   Update all the time series back into the table.
*
* returns:
*
* notes:
*****
*/
static void
update_series( loader_context_t *context_ptr)
{
    mi_integer i = 0;
    register struct sec_entry_s *entry_ptr = NULL;
    struct sec_entry_s *next_entry_ptr = NULL;
    MI_STATEMENT *statement = NULL;
    char buf[256];
    mi_integer rc = 0;
    MI_DATUM values[2] = {0, 0};
    mi_integer lengths[2] = {-1, sizeof( mi_integer)};
    static const mi_integer nulls[2] = {0, 0};
    static const mi_string const *types[2]
= {"timeseries(day_info)", "integer"};
    mi_unsigned_integer yield_count = 0;

    sprintf( buf, "update %.*s set series = ? where Secid = ?;",
        mi_get_varlen( context_ptr->table_name),
        mi_get_vardata( context_ptr->table_name));
    statement = mi_prepare( context_ptr->conn, buf, NULL);
    if( NULL == statement)
mi_db_error_raise( NULL, MI_EXCEPTION, "mi_prepare failed");

    /* Look at all the entries in the hash table. */
    for( i = context_ptr->hash_size - 1; 0 <= i; i--)
    {
        for( entry_ptr = context_ptr->hash[i];
            NULL != entry_ptr;
            entry_ptr = next_entry_ptr)
        {
            if( NULL != entry_ptr->tsdesc)
            {
                yield_count++;
                if( 0 == (yield_count & 0x3f))
                {
                    if( mi_interrupt_check())
mi_db_error_raise( NULL, MI_EXCEPTION, "Load aborted.");
                    mi_yield();
                }
            }
        }
    }
}

```

```

        values[0] = ts_get_ts( entry_ptr->tsdesc);
        values[1] = (MI_DATUM) entry_ptr->sec_id;
        lengths[0] = mi_get_varlen( ts_get_ts( entry_ptr->tsdesc));

        if( mi_exec_prepared_statement( statement,
            MI_BINARY,
            1,
            2,
            values,
            lengths,
            (int *) nulls,
            (char **) types,
            0,
            NULL)
            != MI_OK)
            mi_db_error_raise( NULL, MI_EXCEPTION,
                "mi_exec_prepared_statement(update) failed");
        ts_close( entry_ptr->tsdesc);
    }
    next_entry_ptr = entry_ptr->next;
    mi_free( entry_ptr);
}
context_ptr->hash[i] = NULL;
}
} /* End of update_series. */

/*
*****
* name:      open_buf
*
* purpose:   Open a file for reading and attach it to a buffer.
*
*****
*/
static void
open_buf( mi_lvarchar *file_name,
          FILE_BUF *buf_ptr)
{
    mi_string *file_name_str = mi_lvarchar_to_string( file_name);

    memset( buf_ptr, 0, sizeof( *buf_ptr));
    buf_ptr->fd = mi_file_open( file_name_str, O_RDONLY, 0);
    mi_free( file_name_str);
    buf_ptr->file_name = file_name;

    if( MI_ERROR == buf_ptr->fd)
    {
        char buf[356];
        mi_integer name_len = (256 < mi_get_varlen( file_name))
            ? 256 : mi_get_varlen( file_name);

        sprintf( buf, "mi_file_open(%.s) failed",
            name_len, mi_get_vardata( file_name));

        mi_db_error_raise( NULL, MI_EXCEPTION, buf);
    }
    buf_ptr->buf_index = 0;
    buf_ptr->buf_len = 0;
    buf_ptr->line_no = 1;
} /* End of open_buf. */

/*
*****
* name:      get_char

```

```

*
* purpose:  Get the next character from a buffered file.
*
* returns:  The character or STREAM_EOF
*
*****
*/
static mi_integer
get_char( FILE_BUF *buf_ptr)
{
    register mi_integer c = STREAM_EOF;

    if( buf_ptr->buf_index >= buf_ptr->buf_len)
    {
        buf_ptr->buf_index = 0;
        buf_ptr->buf_len = mi_file_read( buf_ptr->fd,
            buf_ptr->data,
            sizeof( buf_ptr->data));
        if( MI_ERROR == buf_ptr->buf_len)
        {
            char buf[356];
            mi_integer name_len = (256 < mi_get_varlen( buf_ptr->file_name))
            ? 256 : mi_get_varlen( buf_ptr->file_name);

            sprintf( buf, "mi_file_read(%.s) failed",
                name_len, mi_get_vardata(buf_ptr->file_name));

            mi_db_error_raise( NULL, MI_EXCEPTION, buf);
        }
        if( 0 == buf_ptr->buf_len)
            return( STREAM_EOF);
    }

    /* Increment buf_ptr->line_no until we have started on the next
line,
    * not when the newline character is seen.
    */
    if( buf_ptr->flags & LDBUF_LAST_CHAR_EOL)
    {
        buf_ptr->line_no++;
        buf_ptr->flags &= ~LDBUF_LAST_CHAR_EOL;
    }

    c = buf_ptr->data[ buf_ptr->buf_index++];
    if( '\n' == c)
        buf_ptr->flags |= LDBUF_LAST_CHAR_EOL;
    return( c);
} /* End of get_char. */

/*
*****
* name:      close_buf
*
* purpose:   Close a file attached to a buffer.
*
* notes:
*****
*/
static void
close_buf( FILE_BUF *buf_ptr)
{
    mi_file_close( buf_ptr->fd);
    buf_ptr->fd = MI_ERROR;
    buf_ptr->buf_index = 0;
    buf_ptr->buf_len = 0;
    buf_ptr->file_name = NULL;

```

```

} /* End of close_buf. */

/*
*****
* name:      get_token
*
* purpose:   Get the next token from an input stream.
*
* returns:   The token in a buffer and the next character after the
buffer.
*
* notes:     Assumes that the tokens are separated by white space.
*****
*/
static mi_integer
get_token( FILE_BUF *buf_ptr,
           mi_string *token,
           size_t token_buf_len)
{
    register mi_integer c = get_char( buf_ptr);
    register mi_integer i = 0;

    while( STREAM_EOF != c && isspace( c))
        c = get_char( buf_ptr);

    for( ;STREAM_EOF != c && ! isspace( c); c = get_char(
buf_ptr))
    {
        if( i >= token_buf_len - 1)
        {
            char err_buf[128];

            sprintf( err_buf, "Word is too long on line %d.", buf_ptr->line_no);
            mi_db_error_raise( NULL, MI_EXCEPTION, err_buf);
        }
        token[i++] = c;
    }
    token[i] = 0;

    return( c);
} /* End of get_token. */

/*
*****
* name:      increment_instances_created
*
* purpose:   Increment the instances_created field and update statistics
when it crosses a threshold. If the statistics for the
time series instance table were never updated then the
server
series
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_tsdsc **tsdsc_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);

    *tsdsc_ptr = sec_entry_ptr->tsdsc;
    if( NULL == sec_entry_ptr->tsdsc)
        /* An invalid security ID. */
        return( 1);

    if( context_ptr->col_is_null[0]
&& TS_IS_IRREGULAR( ts_get_ts( sec_entry_ptr->tsdsc)))
    {
        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:      TSInclLoad
*
* purpose:   UDR for incremental loading of timeseries from a file.
*
*****
*/
void
TSInclLoad( mi_lvarchar *table_name, /* the table that holds the time
series. */
            mi_lvarchar *file_name,
            /* The name of the file containing the data. It must be accessible
            * on the server machine.
            */
            /*
            * The following parameters are only used to create new time
            * series.
            */
            mi_lvarchar *calendar_name,
            mi_datetime *origin,
            mi_integer threshold,
            mi_boolean regular,
            mi_lvarchar *container_name,
            mi_integer nelems,
            MI_FPARAM *fParamPtr)
{
    FILE_BUF buf = {0};
    ts_tselem day_elem = NULL;
    ts_tsdesc *tsdesc = NULL;
    ts_timeseries *ts = NULL;
    mi_datetime tstamp = {0};
    loader_context_t context = {0};
    mi_unsigned_integer yield_count = 0;
    sec_entry_t *sec_entry_ptr = NULL;
    int null_line = 0;

    init_context( table_name,
                  calendar_name,
                  origin,
                  threshold,
                  regular,
                  container_name,
                  nelems,
                  &context);

    open_buf( file_name, &buf);

    while( read_line( &context,
                      &buf,
                      &tsdesc,
                      &day_elem,
                      &null_line,
                      &tstamp,
                      &sec_entry_ptr))
    {
        yield_count++;
    }
}

```

```

/* Periodically (once every 64 input lines) check for interrupts
and
* yield the processor to other threads.
*/
if( 0 == (yield_count & 0x3f))
{
    if( mi_interrupt_check())
        mi_db_error_raise( NULL, MI_EXCEPTION, "Load aborted.");
    mi_yield();
}

if( null_line)
    continue;

ts = ts_put_elem_no_dups( tsdesc, day_elem, &tstamp);
if( sec_entry_ptr->in_row && TS_IS_INCONTAINER( ts))
{
    sec_entry_ptr->in_row = 0;
    increment_instances_created( &context);
}
}

if( NULL != day_elem)
    ts_free_elem( tsdesc, day_elem);

close_buf( &buf);
update_series( &context);
close_context( &context);
} /* End of TSIncLoad. */

```

Appendix C. Accessibility

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

Accessibility features for IBM Informix products

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

Accessibility features

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

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

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

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

IBM and accessibility

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

Dotted decimal syntax diagrams

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

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

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

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

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

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

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

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

- ? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.
- * Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be

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

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
 2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
 3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

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

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

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

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

Privacy policy considerations

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

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

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

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

Trademarks

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

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

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

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

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

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

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

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

Index

A

- Abs function 7-11
- Absolute value, determining 7-11
- Accessibility C-1
 - dotted decimal format of syntax diagrams C-1
 - keyboard C-1
 - shortcut keys C-1
 - syntax diagrams, reading in a screen reader C-1
- Acos function 7-11
- Adding previous values to current 7-90
- Adding two time series 7-77
- AggregateBy function 7-11, 7-15
- Aggregating time series values 7-11
- ALTER TYPE statement 1-26
- AndOp function 5-1, 6-1
- Applets 8-1
- Apply function 7-18
 - virtual tables 4-8
- ApplyBinaryTsOp function 7-23
- ApplyCalendar function 7-24
- Applying a calendar to a time series 7-24
- Applying an expression to a time series 7-18
- ApplyOpToTsSet function 7-25
- ApplyUnaryTsOp function 7-26
- Arc cosine, determining 7-11
- Arc sine, determining 7-27
- Arc tangent, determining 7-27
- Arithmetic functions
 - binary 7-27
 - unary 7-156
- Asin function 7-27
- Atan function 7-27
- Atan2 function 7-27
- autopool container pool 3-19
- Average, computing running 7-148

B

- BaseTableName parameter 4-5
- beforeFirst method 8-4
- Binary arithmetic functions
 - Atan2 7-27
 - description of 7-27
 - Divide 7-46
 - Minus 7-74
 - Mod 7-75
 - Plus 7-77
 - Pow 7-77
 - Times 7-86
- BulkLoad function 3-35, 7-30

C

- Calendar 1-4
- Calendar data type 2-4
- Calendar pattern 1-4
- Calendar pattern routines 5-1
- Calendar patterns 1-13
 - collapsing 5-3
 - data type for 2-1

- Calendar patterns (*continued*)
 - expanding 5-4
 - getting 7-48
 - intersection of two 5-1
 - interval options 2-1
 - reversing intervals for 5-4
 - specification for 2-1
 - start date for 2-4
 - system table for 2-8
 - union of two 5-5
- Calendar routines 6-1
- CalendarPattern data type 2-1
- CalendarPatterns system table
 - defined 2-8
- Calendars 1-13
 - applying new to time series 7-24
 - calibrated search using 7-161
 - data type for 2-4
 - getting 7-49
 - intersection of time series, from 7-71
 - intervals, determining number of between time stamps 6-2
 - intervals, determining number of between timestamps 9-9
 - lagging 7-74
 - names of, getting 9-27
 - predefined 3-13
 - relative search using 7-161
 - returned time series and 2-8
 - specifying 2-4, 3-24
 - start date for 2-4
 - system table for 2-8
 - timestamp, getting after intervals 6-4, 9-11
 - timestamps, getting in a range 6-3, 9-10, 9-11
 - union of two 6-5
- CalendarTable system table
 - defined 2-8
- Calibrated search type 7-161
- CalIndex function 6-2
- CalPattStartDate function 5-2
- CalRange function 6-3
- CalStamp function 6-4
- CalStartDate function 6-5
- Change Data Capture and time series 1-24
- CLASSPATH variable 8-3
- Clip function 7-31
- ClipCount function 7-35
- ClipGetCount function 7-37
- Clipping a time series 7-18, 7-31, 7-35
- Closing a time series 9-12
- Collapse function 5-3
- Collapsing a calendar pattern 5-3
- Columns
 - data, getting 9-28
 - ID number, getting 9-14, 9-15
 - number of in a time series, getting 9-14
 - numbering with Java 8-4
 - TimeSeries type 3-13
 - type information, getting for 9-15
- Command-line loader application 3-29, 3-30
- Comparing two time stamps 9-21
- Comparing two values 7-91

- compliance with standards xviii
- Compressed data 1-10
 - creating time series 7-118
 - deleting 3-37
 - inserting 3-37
- Compression
 - specifying 3-24
- Container pool
 - default 3-19
 - round-robin order 3-20
 - user-defined policy 3-20
- Container pools 1-15
 - creating 3-19
 - user-defined policy 3-21
- Containers 1-15
 - creating 3-15, 7-93
 - dbspaces 1-15
 - destroying 7-98
 - determining implicitly 3-27
 - instance ID of a time series in a, getting 7-71
 - managing 7-99
 - monitor 3-18
 - moving 3-19
 - multiple writers 7-99
 - name of, getting 7-51, 9-29
 - name, setting 7-84
 - rolling window 1-15, 3-16
 - specifying 3-24
 - system table for 2-9, 2-10, 2-11, 2-12
 - time series, determining if it is in a 9-36
 - TSContainerNElems 7-104
 - TSContainerPctUsed 7-106
 - TSContainerTotalPages 7-112
 - TSContainerTotalUsed 7-113
 - TSContainerUsage 7-114
- Converting
 - element to a row 9-24
 - row to element 9-50
 - time series data to tabular form 7-87
- Copying
 - one time series into another 9-49
 - time series 9-16
- Cos function 7-38
- Cosine, determining 7-38
- CountIf function 7-38
- Counting elements returned by an expression 7-38
- CREATE ROW TYPE statement 3-13
- CREATE TABLE statement 3-14
- Creating
 - irregular time series 7-118
 - regular time series 7-116
 - table for time series 3-14
 - time series 3-22, 3-27, 9-17, 9-18
 - time series from function output 3-27
 - time series subtype 3-13
 - time series with input function 3-24
 - time series with metadata 3-23
 - virtual tables 4-5

D

- Data
 - file formats 3-35
 - loading from a file 7-30
 - loading into a time series with BulkLoad 3-35
- Data structures
 - ts_timeseries 9-2

- Data structures (*continued*)
 - ts_tscan 9-2
 - ts_tsdesc 9-2
 - ts_tselem 9-3
- Data Studio 1-3
 - TimeSeries plug-in 3-27, 3-28, 3-29
- Data type mapping 8-4
- Data types
 - Calendar 2-4
 - CalendarPattern 2-1
 - DATETIME 3-13
 - restrictions for time series 3-13
 - TimeSeriesMeta 3-23
- Database
 - requirements 1-26
- DATETIME data type 3-13
- dbload utility 3-34
- dbspace, time series containers 1-15
- Decay, computing 7-122
- DelClip function 7-42
- DelElem function 7-43
- Deleting
 - element 7-43, 9-22
 - elements in a clip 7-42, 7-45
 - elements in a range 7-44
 - null elements 7-75
- Deleting time series data 3-37, 7-108
- DelRange function 7-44
- DelTrim function 7-45
- Directory 1-27, 1-28
- Disabilities, visual
 - reading syntax diagrams C-1
- Disability C-1
- Divide function 7-46
- Dividing one time series by another 7-46
- Documentation files, Java 8-3
- Dotted decimal format of syntax diagrams C-1
- DROP statement, virtual tables 4-26

E

- Eclipse command-line loader application 3-29, 3-30
- Element 1-4
- Elements
 - columns in, getting number of 9-14
 - converting to a row 9-24
 - data from one column in, getting 9-28
 - deleting 7-43, 9-22
 - deleting from a clip 7-42, 7-45
 - deleting from a range 7-44
 - deleting null 7-75
 - first in a time series, getting 7-53, 9-25
 - freeing memory for 9-26
 - getting 7-52, 9-22, 9-27
 - hidden, determining if 9-23
 - hidden, revealing 7-83, 9-50
 - hiding 7-67, 9-34
 - inserting 7-69, 7-77, 7-79, 9-36, 9-38, 9-46, 9-47
 - inserting a set of 7-70, 7-81
 - inserting at an offset 7-80, 9-48
 - inserting at end of a time series 9-48
 - last valid, getting 7-58
 - last, getting 7-56, 9-37
 - next valid, getting 7-61
 - next, getting 9-41
 - null, determining if 9-24
 - number in time series clip, getting 7-37

- Elements (*continued*)
 - number of, getting 7-60, 9-41
 - offset, getting for an 7-62, 9-20, 9-43
 - timestamp, getting for an 9-38
 - timestamp, getting last before 7-65, 9-45
 - timestamp, getting nearest to an 9-42
 - updating 7-159, 9-46, 9-47, 9-54
 - updating a set of 7-160
- Enterprise Replication and time series 1-24
- Examples
 - directory 1-27, 1-28
 - stock data 1-28
 - virtual tables 4-8
- Exp function 7-48
- Expand function 5-4
- Expanding a calendar pattern 5-4
- Exponentiating a time series 7-48

F

- Flags
 - argument 7-9
 - getting for a time series 9-30
 - TS_CREATE_IRR 9-17, 9-18
 - TS_PUTELEM_NO_DUPS 7-9
 - TS_SCAN_EXACT_START 7-87, 9-7
 - TS_SCAN_HIDDEN 7-37, 7-87, 9-7
 - TS_SCAN_SKIP_BEGIN 7-87, 9-7
 - TS_SCAN_SKIP_END 7-87, 9-7
 - TSOPEN_NO_NULLS 7-9
 - TSOPEN_RDWRITE 7-9
 - TSOPEN_READ_HIDDEN 7-9
 - TSOPEN_REDUCED_LOG 7-9
 - TSOPEN_WRITE_AND_HIDE 7-9
 - TSOPEN_WRITE_HIDDEN 7-9
 - TSWRITE_AND_REVEAL 7-9
- Freeing memory for a time series 9-26
- Freeing memory for a time series element 9-26
- Function output, creating time series with 3-27

G

- GetCalendar function 7-48
- GetCalendarName function 7-49
- GetCompression function 7-51
- GetContainerName function 7-51
- GetElem function 7-52
- GetFirstElem function 7-53
- GetFirstElementStamp function 7-54
- GetHertz function 7-54
- GetIndex function 7-55
- getInt method 8-4
- GetInterval function 7-55
- GetLastElem function 7-56, 7-57, 7-61
- GetLastElementStamp function 7-57
- GetLastValid function 7-58
- GetMetaData function 7-59
- GetMetaTypeName function 7-59
- GetNelems function 7-60
- GetNextValid function 7-61
- GetNthElem function 7-62
- GetOrigin function 7-64
- GetPacked function 7-64
- GetPreviousValid function 7-65
- GetStamp function 7-66
- GetThreshold function 7-67

- getTimestamp method 8-4
- getVersion method 8-5
- GMT, converting to 9-40

H

- Hardware requirements 1-25
- HDR and time series 1-24
- Hertz data 1-8
 - creating time series 7-118
 - deleting 3-37
 - inserting 3-37
 - specifying 3-24
- Hidden elements 4-3
- HideElem function 7-67
- Hiding an element 7-67, 9-34

I

- IfmxTimeSeries object 8-4
- IfmxTimeSeries.jar 8-3
- Indexes
 - base tables 4-2
- industry standards xviii
- Informix JDBC Driver 8-1
- Informix Warehouse Accelerator
 - virtual tables 4-2
- Input function, creating time series with 3-24
- InsElem function 3-36, 7-69
- INSERT statement 3-24
- Inserting
 - element 7-69, 7-77, 7-79, 9-36, 9-38, 9-46, 9-47
 - element at an offset 7-80, 9-48
 - element at end of a time series 9-48
 - elements, set of 7-70, 7-81
 - time series into another time series 7-82
- InsSet function 3-36, 7-70
- Instance ID, getting for a time series 7-71
- InstanceId function 7-71
- Intersect function 7-71
- Intersection
 - calendar patterns, of 5-1
 - calendars, of 6-1
 - time series, of 7-71
- Interval
 - calendar pattern, for 1-13, 2-1
 - getting for a time series 7-55
 - number of between time stamps, determining 9-9
- Irregular time series 1-7
 - creating with metadata 7-118
 - creating with TSCreateIrr 7-118
 - determining if 9-37
 - specifying 3-24
- IsRegular function 7-73

J

- jar file 8-3
- Java class library 8-1
 - directory 8-3
 - sample programs 8-3
- Java Developers' Kit 8-3
- javadoc 8-3
- Javadoc 8-3
- JDBC 8-1
- JDBC 2.0 specification 8-1

- JSON
 - time series 1-12
- JSON data
 - loading 3-33

L

- Lag function 7-74
- Lagging, creating new time series 7-74
- LessThan operator 1-26
- load command 3-34
- Loader program 3-31
- Loading data 3-27
 - from a file 3-35, 7-30
 - JSON 3-33
 - time series 1-23
 - using virtual tables 3-34
- Loading data from a database 3-29
- Loading data from a file 3-28
- Loading time series data 3-1
 - command-line loader application 3-29, 3-30
- Local time, converting to 9-32
- Logn function 7-74

M

- Mapping API functions to SQL functions 9-3
- Mapping data types
 - Java programs 8-4
- Metadata
 - adding to a time series 7-160, 9-54
 - creating a time series with 7-116, 7-118, 9-18
 - creating for a time series 3-23
 - getting from a time series 7-59, 9-31
 - getting the type name of 7-59
 - getting type ID from a time series 9-31
 - using distinct type TimeSeriesMeta 3-23
- mi_set_trace_file() API routine, virtual tables 4-26
- mi_set_trace_level() API routine, virtual tables 4-27
- Minus function 7-74
- Mod function 7-75
- Modulus, computing of division of two time series 7-75
- Multiplying one time series by another 7-86

N

- Natural logarithm, determining 7-74
- Negate function 7-75
- Negating a time series 7-75
- next method 8-4
- NotOp function 5-4
- Null elements 4-3
- NullCleanup function 7-75

O

- Offsets 1-7
 - converting to time stamp 9-51
 - determining 9-20
 - element, getting for 9-43
 - inserting an element at 7-80, 9-48
 - timestamp, getting for 7-55, 7-66, 9-35
- onpload utility 3-34
- OpenAdmin Tool for Informix 1-3
- Opening a time series 9-44

- Operators
 - LessThan 1-26
- Optim Developer Studio
 - TimeSeries plug-in 3-27
- ORDER BY clause, virtual tables 4-8
- Origin 1-4
- Origin of a time series
 - changing 7-85
 - getting 7-64, 9-31
 - specifying 3-24
- OrOp function 5-5, 6-5
- Output of a function, creating time series with 3-27

P

- Packed elements
 - compressed data 1-10
- Packed time series 1-8
- Patterns 1-13
- Performance, virtual tables 4-2
- planning 1-19
- pload utility 3-34
- Plus function 7-77
- Positive function 7-77
- Pow function 7-77
- Properties of time series 1-19
- PutElem function 3-36, 7-77
- PutElemNoDups function 7-79
- PutNthElem function 7-80
- PutSet function 3-36, 7-81
- PutTimeSeries function 7-82

R

- Raising one time series to the power of another 7-77
- readSQL method 8-4
- Regular time series 1-7
 - creating with metadata 7-116
 - creating with TSCreate 7-116
 - determining if 7-73
 - specifying 3-24
- Regularity 1-4
- Relative search type 7-161
- Replicating time series data 1-24
- ResultSet interface 8-4
- Retrieving time series data (Java) 8-4
- RevealElem function 7-83
- Revealing a hidden element 7-83, 9-50
- Rolling window containers 3-16
- Round function 7-84
- Rounding a time series to a whole number 7-84
- Routines
 - API
 - ts_begin_scan 9-7
 - ts_cal_index 9-9
 - ts_cal_pattstartdate 9-9
 - ts_cal_range 9-10
 - ts_cal_range_index 9-11
 - ts_cal_stamp 9-11, 9-12
 - ts_close 9-12
 - ts_col_cnt 9-14
 - ts_col_id 9-14
 - ts_colinfo_name 9-15
 - ts_colinfo_number 9-15
 - ts_copy 9-16
 - ts_create 9-17

Routines (continued)

API (continued)

ts_create_with_metadata 9-18
 ts_current_offset 9-20
 ts_current_timestamp 9-21
 ts_datetime_cmp 9-21
 ts_del_elem 9-22
 ts_elem 9-22
 TS_ELEM_HIDDEN 9-23
 TS_ELEM_NULL 9-24
 ts_elem_to_row 9-24
 ts_end_scan 9-25
 ts_first_elem 9-25
 ts_free 9-26
 ts_free_elem 9-26
 ts_get_all_cols 9-27
 ts_get_calname 9-27
 ts_get_col_by_name 9-28
 ts_get_col_by_number 9-28
 ts_get_compressed() 9-29
 ts_get_containername 9-29
 ts_get_flags 9-30
 ts_get_hertz() 9-30
 ts_get_metadata 9-31
 ts_get_origin 9-31
 ts_get_packed() 9-32
 ts_get_stamp_fields 9-32
 ts_get_threshold 9-33
 ts_get_ts 9-33
 ts_get_typeid 9-34
 ts_hide_elem 9-34
 ts_index 9-35
 ts_ins_elem 9-36
 TS_IS_INCONTAINER 9-36
 TS_IS_IRREGULAR 9-37
 ts_last_elem 9-37
 ts_last_valid 9-38
 ts_make_elem 9-38
 ts_make_elem_with_buf 9-39
 ts_make_stamp 9-40
 ts_nelems 9-41
 ts_next 9-41
 ts_next_valid 9-42
 ts_nth_elem 9-43
 ts_open 9-44
 ts_previous_valid 9-45
 ts_put_elem 9-46
 ts_put_elem_no_dups 9-47
 ts_put_last_elem 9-48
 ts_put_nth_elem 9-48
 ts_put_ts 9-49
 ts_reveal_elem 9-50
 ts_row_to_elem 9-50
 ts_time 9-13, 9-51, 9-52, 9-53
 ts_upd_elem 9-54
 ts_update_metadata 9-54

SQL, calendar

AndOp 6-1
 CalIndex 6-2
 CalRange 6-3
 CalStamp 6-4
 CalStartDate 6-5
 OrOp 6-5

SQL, calendar pattern

AndOp 5-1
 CalPattStartDate 5-2
 Collapse 5-3

Routines (continued)

SQL, calendar pattern (continued)

Expand 5-4
 NotOp 5-4
 OrOp 5-5

SQL, time series

Abs 7-11
 Acos 7-11
 AggregateBy 7-11, 7-15
 Apply 7-18
 ApplyBinaryTsOp 7-23
 ApplyCalendar 7-24
 ApplyOpToTsSet 7-25
 ApplyUnaryTsOp 7-26
 Asin 7-27
 Atan 7-27
 Atan2 7-27
 BulkLoad 3-35, 7-30
 Clip 7-31
 ClipCount 7-35
 ClipGetCount 7-37
 Cos 7-38
 CountIf 7-38
 DelClip 7-42
 DelElem 7-43
 DelRange 7-44
 DelTrim 7-45
 Divide 7-46
 Exp 7-48
 GetCalendar 7-48
 GetCalendarName 7-49
 GetCompression 7-51
 GetContainerName 7-51
 GetElem 7-52
 GetFirstElem 7-53
 GetFirstElementStamp 7-54
 GetHertz 7-54
 GetIndex 7-55
 GetInterval 7-55
 GetLastElem 7-56, 7-57, 7-61
 GetLastElementStamp 7-57
 GetLastValid 7-58
 GetMetaData 7-59
 GetMetaTypeName 7-59
 GetNelems 7-60
 GetNextValid 7-61
 GetNthElem 7-62
 GetOrigin 7-64
 GetPacked 7-64
 GetPreviousValid 7-65
 GetStamp 7-66
 GetThreshold 7-67
 HideElem 7-67
 InsElem 3-36, 7-69
 InsSet 3-36, 7-70
 InstanceId 7-71
 Intersect 7-71
 IsRegular 7-73
 Lag 7-74
 Logn 7-74
 Minus 7-74
 Mod 7-75
 Negate 7-75
 NullCleanup 7-75
 Plus 7-77
 Positive 7-77
 Pow 7-77

Routines (*continued*)

SQL, time series (*continued*)

- PutElem 3-36, 7-77
- PutElemNoDups 7-79
- PutNthElem 7-80
- PutSet 3-36, 7-81
- PutTimeSeries 7-82
- RevealElem 7-83
- Round 7-84
- SetContainerName 7-84
- SetOrigin 7-85
- Sin 7-85
- Sqrt 7-86
- Tan 7-86
- Times 7-86
- TimeSeriesRelease 7-86
- Transpose 7-87
- TsAddPrevious 7-90
- TSCmp 7-91
- TSContainerCreate 7-93
- TSContainerDestroy 7-98
- TSContainerLock 7-99
- TSContainerManage 7-99
- TSContainerPurge 7-108
- TSCreate 7-116
- TSCreateIrr 7-118
- TSDecay 7-122
- TSL_Attach 7-123
- TSL_Commit 7-124
- TSL_Flush 7-126
- TSL_FlushAll 7-128
- TSL_FlushInfo 7-129
- TSL_FlushStatus 7-131
- TSL_GetKeyContainer 7-131
- TSL_GetLogMessage 7-132
- TSL_Init 7-133
- TSL_Put 7-135
- TSL_PutRow 7-137
- TSL_PutSQL 7-138
- TSL_SessionClose 7-139
- TSL_SetLogMode 7-140
- TSL_Shutdown 7-141
- TSPrevious 7-142
- TSRollup 7-142
- TSRunningAvg 7-92, 7-148
- TSRunningCor 7-149
- TSRunningMed 7-150
- TSRunningSum 7-151
- TSRunningVar 7-152
- TSSetToList 7-153
- TSToXML 7-154
- Union 7-157
- UpdElem 7-159
- UpdMetaData 7-160
- UpdSet 7-160
- WithinC 7-161
- WithinR 7-161

Row converting to an element 9-50

RSS and time series 1-24

Running average, computing 7-148

Running sum, computing 7-151

S

Scanning

beginning for a time series 9-7

ending for a time series 9-25

Screen reader

reading syntax diagrams C-1

SDS and time series 1-24

SELECT DISTINCT statement 1-26

Servlets 8-1

session_number.trc file 4-26

SetContainerName function 7-84

SetOrigin function 7-85

Shortcut keys

keyboard C-1

Sin function 7-85

Sine, determining 7-85

Software requirements 1-25

SQL statements

ALTER TYPE 1-26

CREATE ROW TYPE 3-13

CREATE TABLE 3-14

INSERT 3-24

restrictions for time series 1-26

SELECT DISTINCT 1-26

UPDATE 3-35

virtual tables 4-1

Sqrt function 7-86

Square root, determining 7-86

standards xviii

Start date

calendar of 2-4

calendar pattern of 2-4

Storage

time series data 1-20

Storage, for time series 1-15

Subtracting, one time series from another 7-74

Sum, running 7-151

Syntax diagrams

reading in a screen reader C-1

System tables

CalendarPatterns 2-8

CalendarTable 2-8

TSContainerTable 2-9

TSContainerUsageActiveWindowVTI 2-11

TSContainerUsageDormantWindowVTI 2-12

TSContainerWindowTable 2-10

TSInstanceTable 2-12

T

Table 2-10, 2-11, 2-12

Table. 2-9

Tables, virtual 4-1, 4-8

Tabular form, converting time series data to 7-87

Tan function 7-86

Tangent, determining 7-86

Threshold for containers

specifying 3-24

time series

examples directory 1-27, 1-28

Time series 1-19

accessing 1-24

calendar pattern routines 5-1

calendar routines 6-1

compressed 1-10

concepts 1-4

creating 3-1

data types 2-8

decisions 1-19

deleting data 3-37

deleting elements 7-108

- Time series (*continued*)
 - example of creating and loading 3-1
 - example of creating for compressed data 3-7
 - example of creating for hertz data 3-5
 - example of creating for JSON data 3-10
 - hardware and software requirements 1-25
 - Informix Warehouse Accelerator 1-24
 - insert through virtual tables 4-4
 - loading data 1-23
 - loading from a database 3-29
 - loading from a file 3-28
 - loading methods 3-27
 - loading with the plug-in 3-27
 - overview 1-1
 - planning 1-19
 - properties 1-19
 - solution architecture 1-3
 - SQL restrictions for 1-26
 - SQL routines 3-31
 - storage planning 1-20
- Time series data
 - command-line loader application 3-29, 3-30
- Time series functions
 - TSContainerNElems 7-104
 - TSContainerPctUsed 7-106
 - TSContainerTotalPages 7-112
 - TSContainerTotalUsed 7-113
 - TSContainerUsage 7-114
 - TSCreateExpressionVirtualTab 4-13
- Time Series Java class version 8-5
- Timepoint 1-4
- Timepoints
 - arbitrary 1-7
- Times function 7-86
- TimeSeries
 - database requirements 1-26
 - loading data 3-29, 3-30, 3-31
 - replicating 1-24
- TimeSeries data type 1-5, 2-6, 3-1
- TimeSeries plug-in 1-3, 3-1, 3-27, 3-28, 3-29
 - command-line loader application 3-29, 3-30
- TimeSeries routines
 - parallelizable 7-7
- TimeSeriesMeta distinct type 3-23
- TimeSeriesRelease function 7-86
- Timestamps
 - calendar, getting from a 9-11
 - comparing 9-21
 - current, getting 9-21
 - getting after intervals 6-4
 - GMT, converting to 9-40
 - local time, converting to 9-32
 - offset associated with 1-7
 - offset, converting from 9-51
 - offset, getting for 7-66
 - offset, getting from 9-35
 - range, getting from a calendar 9-10, 9-11
 - returning set of valid in range 6-3
- traceFileName parameter 4-26
- traceLevelSpec parameter 4-27
- Tracing, virtual tables 4-26
- Transpose function 7-87
- ts_begin_scan function 9-7
- ts_cal_index function 9-9
- ts_cal_pattstartdate function 9-9
- ts_cal_range function 9-10
- ts_cal_range_index function 9-11
- ts_cal_stamp function 9-11, 9-12
- ts_close procedure 9-12
- ts_col_cnt function 9-14
- ts_col_id function 9-14
- ts_colinfo_name function 9-15
- ts_colinfo_number function 9-15
- ts_copy function 9-16
- ts_create function 9-17
- TS_CREATE_IRR flag 9-17, 9-18
- ts_create_with_metadata function 9-18
- ts_current_offset function 9-20
- ts_current_timestamp function 9-21
- ts_datetime_cmp function 9-21
- ts_del_elem function 9-22
- ts_elem function 9-22
- TS_ELEM_HIDDEN macro 9-23
- TS_ELEM_NULL macro 9-24
- ts_elem_to_row 9-24
- ts_end_scan procedure 9-25
- ts_first_elem function 9-25
- ts_free procedure 9-26
- ts_free_elem procedure 9-26
- ts_get_all_cols procedure 9-27
- ts_get_calname function 9-27
- ts_get_col_by_name function 9-28
- ts_get_col_by_number function 9-28
- ts_get_compressed() function 9-29
- ts_get_containername function 9-29
- ts_get_flags function 9-30
- ts_get_hertz() function 9-30
- ts_get_metadata function 9-31
- ts_get_origin function 9-31
- ts_get_packed() function 9-32
- ts_get_stamp_fields procedure 9-32
- ts_get_threshold function 9-33
- ts_get_ts function 9-33
- ts_get_typeid function 9-34
- ts_hide_elem function 9-34
- ts_index function 9-35
- ts_ins_elem function 9-36
- TS_IS_INCONTAINER macro 9-36
- TS_IS_IRREGULAR macro 9-37
- ts_last_elem function 9-37
- ts_last_valid function 9-38
- ts_make_elem function 9-38
- ts_make_elem_with_buf function 9-39
- ts_make_stamp function 9-40
- ts_nelems function 9-41
- ts_next function 9-41
- ts_next_valid function 9-42
- ts_nth_elem function 9-43
- ts_open function 9-44
- ts_previous_valid function 9-45
- ts_put_elem function 9-46
- ts_put_elem_no_dups function 9-47
- ts_put_last_elem function 9-48
- ts_put_nth_elem function 9-48
- ts_put_ts function 9-49
- TS_PUTelem_NO_DUPS flag 7-9
- ts_reveal_elem function 9-50
- ts_row-to_elem function 9-50
- TS_SCAN_EXACT_END flag 9-7
- TS_SCAN_EXACT_START flag 7-87, 9-7
- TS_SCAN_HIDDEN flag 7-37, 7-87, 9-7
- TS_SCAN_SKIP_BEGIN flag 7-87, 9-7
- TS_SCAN_SKIP_END flag 7-87, 9-7
- ts_time function 9-13, 9-51, 9-52, 9-53

- ts_timeseries data structure 9-2
- ts_tscan data structure 9-2
- ts_tsdesc data structure 9-2
- ts_tselem data structure 9-3
- ts_upd_elem function 9-54
- ts_update_metadata function 9-54
- TS_VTI_DEBUG trace class 4-27
- TSAddPrevious function 7-90
- TSCmp function 7-91
- TSColName parameter 4-5
- TSContainerCreate procedure 7-93
- TSContainerDestroy procedure 7-98
- TSContainerLock procedure 7-99
- TSContainerManage procedure 7-99
- TSContainerNElems 7-104
- TSContainerPctUsed 7-106
- TSContainerPurge function 7-108
- TSContainerTable system table 2-9
- TSContainerTotalPages 7-112
- TSContainerTotalUsed 7-113
- TSContainerUsage 7-114
- TSContainerUsageActiveWindowVTI virtual table 2-11
- TSContainerUsageDormantWindowVTI virtual table 2-12
- TSContainerWindowTable system table 2-10
- TSCreate function 7-116
- TSCreateExpressionVirtualTab 4-5, 4-13
- TSCreateIrr function 7-118
- TSCreateVirtualTab procedure 4-5
- TSDecay function 7-122
- TSInstanceTable system table 2-12
- TSL_Attach function 7-123
- TSL_Commit function 7-124
- TSL_Flush function 7-126
- TSL_FlushAll function 7-128
- TSL_FlushInfo function 7-129
- TSL_FlushStatus function 7-131
- TSL_GetKeyContainer function 7-131
- TSL_GetLogMessage function 7-132
- TSL_Init function 7-133
- TSL_Put function 7-135
- TSL_PutRow function 7-137
- TSL_PutSQL function 7-138
- TSL_SessionClose function 7-139
- TSL_SetLogMode function 7-140
- TSL_Shutdown procedure 7-141
- TSOPEN_NO_NULLS flag 7-9
- TSOPEN_RDWRITE flag 7-9
- TSOPEN_READ_HIDDEN flag 7-9
- TSOPEN_WRITE_AND_HIDE flag 7-9
- TSOPEN_WRITE_HIDDEN flag 7-9
- TSPrevious function 7-142
- TSRollup function 7-142
- TSRowNameToList function 7-145
- TSRowNumToList function 7-146
- TSRowToList function 7-147
- TSRunningAvg function 7-92, 7-148
- TSRunningCor function 7-149
- TSRunningMed function 7-150
- TSRunningSum function 7-151
- TSRunningVar function 7-152
- TSSetToList function 7-153
- TSSetTraceFile function 4-26
- TSSetTraceLevel function 4-26, 4-27
- TSToXML function 7-154
- TSVTMode parameter 4-16
- TSWRITE_AND_REVEAL flag 7-9
- Type map 8-4

U

Unary arithmetic functions

- Abs 7-11
- Acos 7-11
- Asin 7-27
- Atan 7-27
- Cos 7-38
- description 7-156
- Exp 7-48
- Logn 7-74
- Negate 7-75
- Positive 7-77
- Round 7-84
- Sin 7-85
- Sqrt 7-86
- Tan 7-86

Union function 7-157

Union of time series 7-157

UPDATE statement 3-35

UPDATE STATISTICS statement 4-2

Updating

- element 9-46, 9-47
- element in a time series 9-54
- metadata in a time series 9-54

Updating a set of elements 7-160

Updating an element 7-159

UpdElem function 7-159

UpdMetaData function 7-160

UpdSet function 7-160

V

Version, TimeSeries Java class 8-5

Virtual table 4-1

Virtual table interface 4-8

Virtual tables

- creating 4-16
- creating fragmented tables 4-10
- creating with expressions 4-13
- display of data 4-3
- insert time series data 4-4
- performance 4-2
- structure 4-2

VirtualTableName parameter 4-5

Visual disabilities

- reading syntax diagrams C-1

W

WithinC function 7-161

WithinR function 7-161



Printed in USA

SC27-4535-03



Spine information:

Informix Product Family Informix

Version 12.10

IBM Informix TimeSeries Data User's Guide

