Oracle TimesTen In-Memory Database SQL Reference Guide

Release 7.0

B31682-01

February 2007

Printed in the United States of America

Contents

5 SQL Statements

6 System and Replication Tables

About this Guide

Oracle TimesTen In-Memory Database is a high-performance, in-memory data manager that supports the ODBC (Open DataBase Connectivity) and JDBC (Java DataBase Connectivity) interfaces.

This guide is for application developers who use and administer TimesTen. It provides a reference for TimesTen SQL statements, expressions, and functions, including TimesTen SQL extensions.

To work with this guide, you should understand how database systems work. You should also have knowledge of SQL (Structured Query Language). See "Background reading" on page 2 if you are not familiar with these interfaces.

TimesTen documentation

TimesTen documentation is available on the product distribution media and on the Oracle Technology Network: http://www.oracle.com/technology/documentation/timesten_doc.html.

Including this guide, the TimesTen documentation set consists of these documents:

Book Titles	Description
Oracle TimesTen In-Memory Database Installation Guide	Contains information needed to install and configure TimesTen on all supported platforms.
Oracle TimesTen In-Memory Database Introduction	Describes all the available features in the Oracle TimesTen In-Memory Database.
Oracle TimesTen In-Memory Database Operations Guide	Provides information on configuring TimesTen and using the ttIsql utility to manage a data store. This guide also provides a basic tutorial for TimesTen.
Oracle TimesTen In-Memory Database C Developer's and Reference Guide and the *Oracle TimesTen In-Memory Database Java Developer's and Reference Guide*	Provide information on how to use the full set of available features in TimesTen to develop and implement applications that use TimesTen.
Oracle TimesTen In-Memory Database API Reference Guide	Describes all TimesTen utilities, procedures, APIs and provides a reference to other features of TimesTen.

Oracle TimesTen In-Memory Database SQL Reference Guide	Contains a complete reference to all TimesTen SQL statements, expressions and functions, including TimesTen SQL extensions.
Oracle TimesTen In-Memory Database Error Messages and SNMP Traps	Contains a complete reference to the TimesTen error messages and information on using SNMP Traps with TimesTen.
Oracle TimesTen In-Memory Database TTClasses Guide	Describes how to use the TTClasses C++ API to use the features available in TimesTen to develop and implement applications.
TimesTen to TimesTen Replication Guide	Provides information to help you understand how TimesTen Replication works and step-by-step instructions and examples that show how to perform the most commonly needed tasks. This guide is for application developers who use and administer TimesTen and for system administrators who configure and manage TimesTen Replication.
TimesTen Cache Connect to Oracle Guide	Describes how to use Cache Connect to cache Oracle data in TimesTen data stores. This guide is for developers who use and administer TimesTen for caching Oracle data.
Oracle TimesTen In-Memory Database Troubleshooting Procedures Guide	Provides information and solutions for handling problems that may arise while developing applications that work with TimesTen, or while configuring or managing TimesTen.

.

Background reading

For a Java reference, see:

- Horstmann, Cay and Gary Cornell. *Core Java(TM) 2, Volume I--Fundamentals (7th Edition) (Core Java 2)*. Prentice Hall PTR; 7 edition (August 17, 2004).

A list of books about ODBC and SQL is in the Microsoft ODBC manual included in your developer's kit. Your developer's kit includes the appropriate ODBC manual for your platform:

- *Microsoft ODBC 3.0 Programmer's Reference and SDK Guide* provides all relevant information on ODBC for Windows developers.

- *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, included online in PDF format, provides information on ODBC for UNIX developers.

For a conceptual overview and programming how-to of ODBC, see:

- Kyle Geiger. *Inside ODBC*. Redmond, WA: Microsoft Press. 1995.

For a review of SQL, see:

- Melton, Jim and Simon, Alan R. *Understanding the New SQL: A Complete Guide*. San Francisco, CA: Morgan Kaufmann Publishers. 1993.
- Groff, James R. / Weinberg, Paul N. *SQL: The Complete Reference, Second Edition.* McGraw-Hill Osborne Media. 2002.

For information about Unicode, see:

- The Unicode Consortium, *The Unicode Standard, Version 5.0*, Addison-Wesley Professional, 2006.
- The Unicode Consortium Home Page at http://www.unicode.org

Conventions used in this guide

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows refers to Windows 2000, Windows XP and Windows Server 2003. The term UNIX refers to Solaris, Linux, HP-UX, Tru64 and AIX.

TimesTen documentation uses these typographical conventions:

If you see...	It means...
`code font`	Code examples, filenames, and pathnames. For example, the `.odbc.ini.` or `ttconnect.ini` file.
`italic code font`	A variable in a code example that you must replace. For example: `Driver=`*`install_dir`*`/lib/libtten.sl` Replace *`install_dir`* with the path of your TimesTen installation directory.

TimesTen documentation uses these conventions in command line examples and descriptions:

If you see...	It means...
`fixed width italics`	Variable; must be replaced with an appropriate value.
`[ ]`	Square brackets indicate that an item in a command line is optional.

`{ }`	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
`|`	A vertical bar (or pipe) separates arguments that you may use more than one argument on a single command line.
`...`	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line.
`%`	The percent sign indicates the UNIX shell prompt.
`#`	The number (or pound) sign indicates the UNIX root prompt.

TimesTen documentation uses these variables to identify path, file and user names:

If you see...	It means...
`install_dir`	The path that represents the directory where the current release of TimesTen is installed.
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique alphanumeric instance name. This name appears in the install path. The instance name “giraffe” is used in examples in this guide.
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
`release` or *`rr`*	Two digits that represent the first two digits of the current TimesTen release number, with or without a dot. For example, 51 or 7.0 represents TimesTen Release 7.0.
`jdk_version`	Two digits that represent the version number of the major JDK release. Specifically, 14 represent JDK 1.4; 5 represents JDK 5.
`timesten`	A sample name for the TimesTen instance administrator. You can use any legal user name as the TimesTen administrator. On Windows, the TimesTen instance administrator must be a member of the Administrators group. Each TimesTen instance can have a unique instance administrator name.
`DSN`	The data source name.

Technical Support

For information about obtaining technical support for TimesTen products, go to the following Web address:

http://www.oracle.com/support/contact.html

1

Data Types

A data type defines a set of values. A reference to a data type specifies the set of values that can occur in a given context.

A data type is associated with each value retrieved from a table or computed in an expression and each constant.

TimesTen follows the ODBC standard for type conversion.

A discussion of this standard is not included in this guide. See Appendix D either in the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide* or the *Microsoft ODBC 3.0 Developer's Kit and Programmer's Reference* for more information.

If you are using the Cache Connect feature of TimesTen, see "Differences between Oracle and TimesTen tables" in Chapter 8, "Compatibility Between TimesTen and Oracle" of the *TimesTen Cache Connect to Oracle Guide*. This section compares valid data types for creating cache group columns, as well as type conversions for passthrough queries.

Type specifications

TimesTen supports the following data types in the default Oracle type mode. The type mode is a data store creation attribute; TypeMode = 0 indicates Oracle type mode; TypeMode = 1 indicates TimesTen mode.

For more information on types modes, see "TypeMode" in *Oracle TimesTen In-Memory Database API Reference Guide*.

Data type	Description
CHAR[ACTER] [(*n* [BYTE|CHAR])]	Fixed-length character string of length *n* bytes or characters. Default is 1 byte. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 8300 bytes. CHAR indicates that the column has character length semantics; The minimum CHAR length is 1 character. The maximum CHAR length depends on how many characters fit in 8300 bytes; this is determined by the DatabaseCharacterSet in use. For character set AL32UTF8, up to four bytes per character may be needed, so the CHAR length limit ranges from 2075 to 8300 depending on the character set. A zero-length string is interpreted as NULL. CHAR data is padded to the maximum column size with trailing blanks; Blank-padded comparison semantics are used. You can alternatively specify ORA_CHAR [(*n* [BYTE|CHAR])].

Data type (*continued*)	Description
NCHAR[(*n*)]	Fixed-length string of length *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; Nchar character limits are 1/2 the byte limits so the maximum size is 4150; Default and minimum bytes of storage is 2*n* (2). A zero-length string is interpreted as NULL. NCHAR data is padded to the maximum column size with U+0020 SPACE; Blank-padded comparison semantics are used. You can alternatively specify ORA_NCHAR[(*n*)].
VARCHAR[2] (*n* [BYTE|CHAR])	Variable-length character string having maximum length *n* bytes or characters. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 4194304 (2^{22}) bytes. You must specify *n.* CHAR indicates that the column has character length semantics. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. Do not use the VARCHAR type. Although it is currently synonymous with VARCHAR2, the VARCHAR type is scheduled to be redefined. You can alternatively specify ORA_VARCHAR2 (*n* [BYTE|CHAR]).

Data type (*continued*)	Description
NVARCHAR2(*n*)	Variable-length string of *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; NVARCHAR2 character limits are 1/2 the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify n. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. You can alternatively specify ORA_NVARCHAR2(*n*).
TT_TINYINT	Unsigned integer ranging from 0 to 255 (2^8-1). Use TT_TINYINT rather than the NUMBER data type. TT_TINYINT is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19 digit integers, use NUMBER (p) where p > 19. Since TT_TINYINT is unsigned, the negation of a TT_TINYINT is TT_SMALLINT.
TT_SMALLINT	A native signed 16 bit integer in the range –32,768 –(2^{15}) to 32,767 (2^{15}–1). Use TT_SMALLINT rather than SMALLINT; SMALLINT maps to the NUMBER data type. TT_SMALLINT is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19 digit integers, use NUMBER (p) where p > 19.
TT_INT[EGER]	A signed integer in the range –2,147,483,648 - -(2^{31}) to 2,147,483,647 (2^{31}–1). TT_INTEGER is a native signed integer data type. Use TT_INTEGER rather than INTEGER; INTEGER maps to the NUMBER data type. TT_INTEGER is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19 digit integers, use NUMBER (p) where p > 19.

Data type (*continued*)	Description
TT_BIGINT	A signed 8-byte integer in the range -9,223,372,036,854,775,808 ($-(2^{63})$) to 9,223,372,036,854,775,807 ($(2^{63} - 1)$). Use TT_BIGINT rather than the NUMBER data type. TT_BIGINT is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19 digit integers, use NUMBER (p) where p > 19.
NUMBER [(precision [,scale])]	Number having precision and scale. The precision ranges from 1 to 38 decimal. The scale ranges from -84 to 127. Both precision and scale are optional. If you do not specify a precision or a scale, then maximum precision of 38 and flexible scale are assumed. NUMBER supports scale > precision and negative scale. NUMBER stores zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} to (but not including) 1.0×10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0×10^{126}, then TimesTen returns an error.
BINARY_FLOAT	32-bit floating-point number; BINARY_FLOAT is a single-precision native floating-point type; Supports +Inf, -Inf and NaN values. BINARY_FLOAT is an approximate numeric value consisting of an exponent and mantissa; You can use Exponential or E-notation. BINARY_FLOAT has binary precision 24. Minimum positive finite value: 1.17549E-38F Maximum positive finite value: 3.40282E+38F

Data type (*continued*)	Description
BINARY_DOUBLE	64-bit floating -point number. BINARY_DOUBLE is a double-precision native floating point number; Supports +Inf, -Inf and Nan values. BINARY_DOUBLE is an approximate numeric value consisting of an exponent and mantissa; You can use Exponential or E-notation. BINARY_DOUBLE has binary precision 53. Minimum positive finite value: 2.22507485850720E-308 Maximum positive finite value: 1.79769313486231E+308
BINARY (*n*)	Fixed-length binary value of n bytes; *n* ranges from 1 to 8300. BINARY data is padded to the maximum column size with trailing zeroes. You can alternatively specify TT_BINARY (*n*).
VARBINARY (*n*)	Variable-length binary value having maximum length *n* bytes; *n* ranges from 1 to 4194304 (2^{22}). You can alternatively specify TT_VARBINARY(*n*).
TIME	A time of day between 00:00:00 (12 midnight) and 23:59:59 (11:59:59 pm), inclusive. The format is: HH:MI:SS; Storage size is 8 bytes. You can alternatively specify TT_TIME.
TT_DATE	Stores date information: century, year, month, date. The format is YYYY-MM-DD; MM is expressed as an integer; For example, 2006-10-28. Storage size is 4 bytes. Valid dates are between 1753-01-01 (January 1,1753) and 9999-12-31 (December 31, 9999).

Data type (*continued*)	Description
DATE	Stores date and time information: century, year, month, date, hour, minute and second: Format is: YYYY-MM-DD HHMMSS. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. The storage size is 7 bytes. There are no fractional seconds. You can alternatively specify ORA_DATE.
TT_TIMESTAMP	A data and time between 1753-01-01 00:00:00 (January 1, 1753 midnight) and 9999-12-31 23:59:59 pm (11:59:59 pm on December 31, 9999), inclusive. Any values for the fraction not specified in full microseconds result in a "Data Truncated" error. The format is YYYY-MM-DD HH:MI:SS [.FFFFFF]. Storage size is 8 bytes. TT_TIMESTAMP has a smaller storage size than TIMESTAMP and TT_TIMESTAMP is faster than TIMESTAMP because TT_TIMESTAMP is an 8 byte integer containing the number of microseconds since January 1, 1754; Comparisons are very fast. TIMESTAMP has a larger range than TT_TIMESTAMP in that TIMESTAMP can store datetime data as far back as 4712 BC; TIMESTAMP also supports up to 9 digits of fractional second precision whereas TT_TIMESTAMP supports 6 digits of fractional second precision. You can specify TT_TIMESTAMP (6).

Data type (*continued*)	Description
TIMESTAMP [(*fractional_seconds_precision*)]	Stores year, month, and day values of the date data type plus hour, minute, and second values of time. *Fractional_seconds_precision* is the number of digits in the fractional part of the seconds field. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. TT_TIMESTAMP has a smaller storage size than TIMESTAMP; TT_TIMESTAMP is faster than TIMESTAMP because TT_TIMESTAMP is an 8 byte integer containing the number of microseconds since January 1, 1754; Comparisons are very fast. TIMESTAMP has a larger range than TT_TIMESTAMP in that TIMESTAMP can store datetime data as far back as 4712 BC; TIMESTAMP also supports up to 9 digits of fractional second precision whereas TT_TIMESTAMP supports 6 digits of fractional second precision. The fractional seconds precision range is 0 to 9; The default is 6. Format is: YYYY-MM-DD HH:MI:SS [.FFFFFFFFF] Storage size 12 bytes. You can alternatively specify ORA_TIMESTAMP [(*fractional_seconds_precision*)]
INTERVAL [+/-] *IntervalQualifier*	TimesTen partially supports INTERVAL types, expressed with the type INTERVAL and an *IntervalQualifier.* An *IntervalQualifier* can only specify a single field type with no precision. The default leading precision is 8 digits for all INTERVAL types. The single field type can be one of: YEAR, MONTH, DAY, HOUR, MINUTE or SECOND. Currently, INTERVAL type can be specified only with a constant.

ANSI SQL data types

TimesTen supports ANSI SQL data types in Oracle type mode. These data types are converted to TimesTen data types and the data is stored as TimesTen data types:

ANSI SQL data type	TimesTen data type
CHARACTER VARYING (*n* [BYTE|CHAR]) or CHAR VARYING(*n*[BYTE|CHAR])	VARCHAR2 (*n* [BYTE|CHAR]) Character semantics is supported.
NATIONAL CHARACTER (*n*) or NATIONAL CHAR (*n*)	NCHAR (*n*)
NATIONAL CHARACTER VARYING (*n*) or NATIONAL CHAR VARYING (*n*) or NCHAR VARYING (*n*)	NVARCHAR2 (*n*)
INT[EGER]	NUMBER (38,0) TT_INTEGER is a native 32 bit integer type. Use TT_INTEGER as this data type is more compact and offers faster performance than the NUMBER type.
SMALLINT	NUMBER (38,0) TT_SMALLINT is a native signed integer data type. Use TT_SMALLINT as this data type is more compact and offers faster performance than the NUMBER type.
NUMERIC [(*p*[,*s*])] or DEC[IMAL] [(*p*[,*s*])]	NUMBER (*p*,*s*) Specifies a fixed-point number with precision *p* and scale *s;* Can only be used for fixed-point numbers. If no scale is specified, *s* defaults to 0.

ANSI SQL data type (*continued*)	TimesTen data type
FLOAT [(*b*)]	NUMBER Floating-point number with binary precision *b*. Acceptable values for *b* are between 1 and 126 binary digits. FLOAT is an exact numeric type; Use FLOAT to define a column with a floated scale and a specified precision. A floated scale is supported with the NUMBER type, but you cannot specify the precision. A lower precision requires less space, so because you can specify a precision with FLOAT, it may be more desirable than NUMBER. If you do not specify *b*, then the default precision is 126 binary (38 decimal). BINARY_FLOAT and BINARY_DOUBLE are inexact numeric types and are therefore different floating types than FLOAT. In addition, the semantics are different between FLOAT and BINARY_FLOAT/ BINARY_DOUBLE because BINARY_FLOAT and BINARY_DOUBLE conform to the IEEE standard. Internally, FLOAT is implements as type NUMBER. You can alternatively specify ORA_FLOAT. For example: FLOAT (24) = ORA_FLOAT (24) FLOAT (53) = ORA_FLOAT (53) FLOAT (n) = ORA_FLOAT (n)
REAL	NUMBER Floating -point number with a binary precision of 63. You can alternatively specify ORA_FLOAT (63) or FLOAT (63).

ANSI SQL data type (*continued*)	TimesTen data type
DOUBLE [PRECISION]	NUMBER Floating- point number with a binary precision of 126. You can alternatively specify FLOAT (126) or ORA_FLOAT (126).

Types supported for backward compatibility in Oracle type mode

TimesTen supports the following data types for backward compatibility in Oracle type mode.

For more information on types modes, see "TypeMode" in *Oracle TimesTen In-Memory Database API Reference Guide*.

Data type	Description
TT_CHAR [(*n* [BYTE|CHAR])]	Fixed-length character string of length *n* bytes or characters. Default is 1 byte. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 8300 bytes. CHAR indicates that the column has character length semantics; The minimum CHAR length is 1 character. The maximum CHAR length depends on how many characters fit in 8300 bytes; this is determined by the DatabaseCharacterSet in use. For character set AL32UTF8, up to four bytes per character may be needed, so the CHAR length limit ranges from 2075 to 8300 depending on the character set. A zero-length string is a valid non-NULL value. TT_CHAR data is padded to the maximum column size with trailing blanks; Blank-padded comparison semantics are used.

Data type (*continued*)	Description
TT_NCHAR[(*n*)]	Fixed-length string of *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; Nchar character limits are 1/2 the byte limits so the maximum size is 4150; Default and minimum bytes of storage is 2*n* (2). A zero-length string is a valid non-NULL value. TT_NCHAR data is padded to the maximum column size with U+0020 SPACE. Blank-padded comparison semantics are used.
TT_VARCHAR (*n* [BYTE|CHAR])	Variable-length character string having maximum length *n* bytes or characters. You must specify *n.* BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 4194304 (2^{22}) bytes. CHAR indicates that the column has character length semantics. A zero-length string is a valid non-NULL value. Blank-padded comparison semantics are used.
TT_NVARCHAR(*n*)	Variable-length string of *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; TT_NVARCHAR character limits are 1/2 the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify n. A zero-length string is a valid non-NULL value. Blank-padded comparison semantics are used.

Data type (*continued*)	Description
TT_DECIMAL[(*p*[,*s*])]	An exact numeric value with a fixed maximum precision (total number of digits) and scale (number of digits to the right of the decimal point). The precision *p* must be between 1 and 40. The scale must be between 0 and *p*. The default precision is 40 and the default scale is 0. Use the NUMBER data type rather than TT_DECIMAL; NUMBER offers better performance.

TimesTen type mapping

The names of the data types listed in the first column are the data types that existed in previous releases of TimesTen. If your TypeMode is set to 0 (the default), indicating Oracle type mode, then the name of the data type may be changed to a new name in Oracle type mode. (The name of the data type in Oracle type mode is listed in column 2.) The table illustrates the mapping of the data type in Column 1 to the corresponding data type in Column 2:

TimesTen data type	TimesTen data type in Oracle type mode
CHAR[ACTER][(*n*)]	TT_CHAR [(*n* [BYTE|CHAR])] In Oracle type mode, specify TT_CHAR. Character semantics is supported. For more information on type TT_CHAR, see "Types supported for backward compatibility in Oracle type mode" on page 18.
NCHAR [(*n)*]	TT_NCHAR[(*n*)] In Oracle type mode, specify TT_CHAR. For more information on TT_NCHAR, see "Types supported for backward compatibility in Oracle type mode" on page 18.
VARCHAR (*n*)	TT_VARCHAR (*n* [BYTE|CHAR]) In Oracle type mode, specify TT_VARCHAR. Character semantics is supported. For more information on TT_VARCHAR, see "Types supported for backward compatibility in Oracle type mode" on page 18.
NVARCHAR (*n)*	TT_NVARCHAR(*n*) In Oracle type mode, specify TT_NVARCHAR. For more information on TT_NVARCHAR, see "Types supported for backward compatibility in Oracle type mode" on page 18.
TINYINT	TT_TINYINT In Oracle type mode, specify TT_TINYINT. For more information on TT_TINYINT, see "Type specifications" on page 8.

TimesTen data type (*continued*)	TimesTen data type in Oracle type mode
SMALLINT	TT_SMALLINT In Oracle type mode, specify TT_SMALLINT. For more information on TT_SMALLINT, see "Type specifications" on page 8.
INT[EGER]	TT_INT[EGER] In Oracle type mode, specify TT_INTEGER. For more information on TT_INTEGER, see "Type specifications" on page 8.
BIGINT	TT_BIGINT In Oracle type mode, specify TT_BIGINT. For more information on TT_BIGINT, see "Type specifications" on page 8.
DEC[IMAL][(*p*[,*s*])] or NUMERIC[(*p*[,*s*])]	TT_DECIMAL[(*p*[,*s*])] In Oracle type mode, specify TT_DECIMAL. For more information on TT_DECIMAL, see "Types supported for backward compatibility in Oracle type mode" on page 18.
REAL or FLOAT (24)	BINARY_FLOAT In Oracle type mode, specify BINARY_FLOAT. For more information on BINARY_FLOAT, see "Type specifications" on page 8.
DOUBLE [PRECISION] or FLOAT [(53)]	BINARY_DOUBLE In Oracle type mode, specify BINARY_DOUBLE. For more information on BINARY_DOUBLE, see "Type specifications" on page 8.
BINARY (*n*)	BINARY (*n*) In Oracle type mode, the data type has the same name. For more information on BINARY (*n*), see "Type specifications" on page 8.

TimesTen data type (*continued*)	TimesTen data type in Oracle type mode
VARBINARY (*n*)	VARBINARY (*n*) In Oracle type mode, the data type has the same name. For more information on VARBINARY (*n*), see "Type specifications" on page 8.
TIME	TIME In Oracle type mode, the data type has the same name. For more information on TIME, see "Type specifications" on page 8.
DATE	TT_DATE In Oracle type mode, specify TT_DATE. For more information on TT_DATE, see "Type specifications" on page 8.
TIMESTAMP	TT_TIMESTAMP In Oracle type mode, specify TT_TIMESTAMP. For more information on TT_TIMESTAMP, see "Type specifications" on page 8.
INTERVAL *IntervalQualifier*	INTERVAL *IntervalQualifier* In Oracle type mode, the data type has the same name. For more information on INTERVAL, see "Type specifications" on page 8.

Character data types

Character data types store character (alphanumeric) data either in the database character set or the UTF-16 format.

Character data is stored in strings with byte values. The byte values correspond to one of the data store character sets defined when the data store is created. Both single byte and multibyte character sets are supported.

The character types are:

- CHAR
- NCHAR
- VARCHAR2
- NVARCHAR2

CHAR type

The CHAR type specifies a fixed length character string. If you insert a value into a CHAR column and the value is shorter than the defined column length, then TimesTen blank pads the value to the column length. If you insert a value into a CHAR column and the value is longer than the defined length, then TimesTen returns an error.

By default, the column length is defined in bytes. Use the CHAR qualifier to define the column length in characters. The size of a character ranges from 1 byte to 4 bytes depending on the database character set. The BYTE and CHAR qualifiers override the NLS_LENGTH_SEMANTICS parameter setting. For more information on NLS_LENGTH_SEMANTICS, see "ALTER SESSION" and "Setting globalization support attributes".

Note: With the CHAR type, a zero-length string is interpreted as NULL; With the TT_CHAR type, a zero-length string is a valid non-NULL value. Both CHAR and TT_CHAR use blank padded comparison semantics. The TT_CHAR type is supported for backward compatibility.

Example 1.1 The following example creates a table; columns are defined with type CHAR and TT_CHAR. Blank padded comparison semantics are used for these types.

```
Command> create table TypeDemo (Name CHAR (20), Name2 TT_CHAR (20));
Command> INSERT INTO TypeDemo VALUES ('SMITH     ',
'SMITH     ');
1 row inserted.
Command> DESCRIBE TypeDemo;

Table USER.TYPEDEMO:
  Columns:
    NAME                           CHAR (20)
```

```
    NAME2                          TT_CHAR (20)

1 table found.
(primary key columns are indicated with *)
Command> select * from TypeDemo;
< SMITH     , SMITH      >
1 row found.
Command> # Expect 1 row found; blank-padded comparison semantics
Command> select * from TypeDemo where Name = 'SMITH';
< SMITH     , SMITH      >
1 row found.
Command> select * from TypeDemo where Name2 = 'SMITH';
< SMITH     , SMITH      >
1 row found.
Command> # Expect 0 rows; blank padded comparison semantics.
Command> SELECT * FROM TypeDemo where Name > 'SMITH';
0 rows found.
Command> SELECT * FROM TypeDemo where Name2 > 'SMITH';
0 rows found.
```

Example 1.2 The following example ALTERs table TypeDemo adding column Name3; The column Name3 is defined with character semantics.

```
Command> ALTER TABLE TypeDemo
> ADD COLUMN Name3 CHAR (10 CHAR);
Command> DESCRIBE TypeDemo;

Table USER.TYPEDEMO:
  Columns:
    NAME                           CHAR (20)
    NAME2                          TT_CHAR (20)
    NAME3                          CHAR (10 CHAR)

1 table found.
```

NCHAR type

The NCHAR data type is a fixed length string of two-byte Unicode characters. NCHAR data types are padded to the specified length with the Unicode space character U+0020 SPACE; Blank-padded comparison semantics are used.

Note: With the NCHAR type, a zero-length string is interpreted as NULL; With the TT_NCHAR type, a zero-length string is a valid non-NULL value. Both NCHAR and TT_NCHAR use blank padded comparison semantics. The TT_NCHAR type is supported for backward compatibility.

Example 1.3 The following example ALTERs table TypeDemo adding column Name4; Data type is NCHAR.

```
Command> ALTER TABLE TypeDemo
> ADD COLUMN Name4 NCHAR (10);
Command> DESCRIBE TypeDemo;

Table USER.TYPEDEMO:
  Columns:
    NAME                            CHAR (20)
    NAME2                           TT_CHAR (20)
    NAME3                           CHAR (10 CHAR)
    NAME4                           NCHAR (10)

1 table found.
```

VARCHAR2 type

The VARCHAR2 data type specifies a variable length character string. When you define a VARCHAR2 column, you define the maximum number of bytes or characters. Each value is stored exactly as you specify it; the value cannot exceed the maximum length of the column.

You must specify the maximum length. The minimum must be at least 1 byte. Use the CHAR qualifier to specify the maximum length in characters. For example, VARCHAR2 (10 CHAR).

The size of a character ranges from 1 byte to 4 bytes depending on the database character set. The BYTE and CHAR qualifiers override the NLS_LENGTH_SEMANTICS parameter setting. For more information on NLS_LENGTH_SEMANTICS, see "ALTER SESSION" and "Setting globalization support attributes".

Note: Do not use the VARCHAR data type. Use VARCHAR2. Even though both data types are currently synonymous, the VARCHAR data type is scheduled to be redefined as a different data type with different semantics.

Note: With the VARCHAR2 type, a zero-length string is interpreted as NULL; With the TT_VARCHAR type, a zero-length string is a valid non-NULL value. VARCHAR2 uses nonpadded comparison semantics; TT_VARCHAR uses blank-padded comparison semantics. The TT_VARCHAR type is supported for backward compatibility.

Example 1.4 The following example ALTERs table TypeDemo adding columns Name5 and Name6; Name5 is defined with type VARCHAR2; Name6 is defined with TT_VARCHAR. The example illustrates the use of nonpadded comparison

semantics with column Name5 and blank-padded comparison semantics with column Table6:

```
Command> ALTER TABLE TypeDemo ADD COLUMN Name5 VARCHAR2 (20);
Command> ALTER TABLE TypeDemo ADD COLUMN Name6 TT_VARCHAR (20);
Command> DESCRIBE TypeDemo;

Table USER.TYPEDEMO:
  Columns:
    NAME                          CHAR (20)
    NAME2                         TT_CHAR (20)
    NAME3                         CHAR (10 CHAR)
    NAME4                         NCHAR (10)
    NAME5                         VARCHAR2 (20) INLINE
    NAME6                         TT_VARCHAR (20) INLINE

1 table found.
(primary key columns are indicated with *)

Command> #Insert SMITH followed by 5 spaces into all columns
Command> INSERT INTO TypeDemo VALUES
> ('SMITH     ', 'SMITH     ', 'SMITH     ', 'SMITH     ',
'SMITH     ', 'SMITH');
1 row inserted.
Command> # Expect 0; Nonpadded comparison semantics
Command> SELECT COUNT (*) FROM TypeDemo where Name5 = 'SMITH';
< 0 >
1 row found.

Command> # Expect 1; Blank-padded comparison semantics
Command> SELECT COUNT (*) FROM TypeDemo where Name6 = 'SMITH';
< 1 >
1 row found.
Command> # Expect 1; Nonpadded comparison semantics
Command> SELECT COUNT (*) FROM TypeDemo where Name5 > 'SMITH';
< 1 >
1 row found.
Command> # Expect 0; Blank-padded comparison semantics
Command> SELECT COUNT (*) FROM TypeDemo where Name6 > 'SMITH';
< 0 >
1 row found.
```

NVARCHAR2 type

The NVARCHAR2 data type is a variable length string of two-byte Unicode characters. When you define an NVARCHAR2 column, you define the maximum number of characters. Each value is stored exactly as you specify it; the value cannot exceed the maximum length of the column. You must specify a length.

Note: With the NVARCHAR2 type, a zero-length string is interpreted as NULL; With the TT_NVARCHAR type, a zero-length string is a valid non-NULL value. NVARCHAR2 uses nonpadded comparison semantics; TT_NVARCHAR uses blank-padded comparison semantics. The TT_NVARCHAR type is supported for backward compatibility.

Example 1.5 The following example ALTERs table TypeDemo adding column Name7; Data type is NVARCHAR2.

```
Command> ALTER TABLE TypeDemo ADD COLUMN Name7 NVARCHAR2 (20);
Command> DESCRIBE TypeDemo;

Table USER1.TYPEDEMO:
  Columns:
    NAME                           CHAR (20)
    NAME2                          TT_CHAR (20)
    NAME3                          CHAR (10 CHAR)
    NAME4                          NCHAR (10)
    NAME5                          VARCHAR2 (20) INLINE
    NAME6                          TT_VARCHAR (20) INLINE
    NAME7                          NVARCHAR2 (20) INLINE

1 table found.
```

Numeric data types

Numeric types store positive and negative fixed and floating-point numbers, zero, infinity, and values that are the undefined result of an operation (NaN or "not a number").

Exact and approximate types

TimesTen supports both exact and approximate numeric types. Arithmetic operations can be performed on numeric types only. Similarly, SUM and AVG aggregates require numeric types.

The exact numeric types are:

- TT_TINYINT
- TT_SMALLINT
- TT_INTEGER
- TT_BIGINT
- NUMBER

The approximate types are:

- BINARY_FLOAT
- BINARY_DOUBLE

TT_TINYINT type

The TT_TINYINT data type is an unsigned integer that ranges from 0 to 255 (2^8 -1). It requires 1 byte of storage and thus is more compact than the NUMBER data type. It also has better performance than the NUMBER data type. The data type of a negative TT_TINYINT is TT_SMALLINT. You cannot specify TINYINT.

Example 1.6 The example first attempts to create a table named Numerics that defines a column named Col1 with data type TINYINT. An error is generated. The column is redefined with data type TT_TINYINT.

```
Command> CREATE TABLE Numerics (Col1 TINYINT);
 3300: TINYINT is not a valid type name; use TT_TINYINT instead
The command failed.
```

```
Command> CREATE TABLE Numerics (Col1 TT_TINYINT);
Command> describe numerics;

Table USER1.NUMERICS:
  Columns:
    COL1                           TT_TINYINT

1 table found.
```

```
(primary key columns are indicated with *)
```

TT_SMALLINT type

The TT_SMALLINT data type is a signed integer that ranges from -32,768 (-2^{15}) to 32,767 ($2^{15} - 1$). It requires 2 bytes of storage and thus is more compact than the NUMBER data type. It also has better performance than the NUMBER data type. You can specify the data type SMALLINT, but it maps to NUMBER (38).

Example 1.7 The example ALTERs the table Numerics and adds Col2 with a data type of SMALLINT. A DESCRIBE of the table shows that the data type is NUMBER (38). Col2 is dropped. A second ALTER TABLE adds Col2 with a data type of TT_SMALLINT.

```
Command> ALTER TABLE Numerics ADD COLUMN Col2 SMALLINT;
Command> DESCRIBE Numerics;

Table USER1.NUMERICS:
  Columns:
    COL1                            TT_TINYINT
    COL2                            NUMBER (38)

1 table found.
(primary key columns are indicated with *)
```

```
Command> ALTER TABLE Numerics DROP COLUMN Col2;
Command> ALTER TABLE Numerics ADD COLUMN Col2 TT_SMALLINT;
Command> DESCRIBE NUMERICS;

Table USER1.NUMERICS:
  Columns:
    COL1                            TT_TINYINT
    COL2                            TT_SMALLINT

1 table found.
(primary key columns are indicated with *)
```

TT_INTEGER type

The TT_INTEGER data type is a signed integer that ranges from -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} -1$). It requires 4 bytes of storage and thus is more compact than the NUMBER data type. It also has better performance than the NUMBER data type. You can specify TT_INT for TT_INTEGER. If you specify either INTEGER or INT, these types are mapped to NUMBER (38).

Example 1.8 The example ALTERs the table Numerics and adds Col3 with a data type of INT. A DESCRIBE of the table shows that the data type is NUMBER (38). Col3 is dropped. A second ALTER TABLE adds Col2 with a data type of INTEGER. A

DESCRIBE of the table shows that the data type is NUMBER (38). Col3 is dropped. Col3 and Col4 are then added with a data type of TT_INTEGER and TT_INT. A DESCRIBE of the table shows the data types are TT_INTEGER.

```
Command> ALTER TABLE Numerics ADD Col3 INT;
Command> DESCRIBE Numerics;

Table USER1.NUMERICS:
  Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          NUMBER (38)

1 table found.
(primary key columns are indicated with *)
```

```
Command> ALTER TABLE Numerics DROP Col3;
Command> ALTER TABLE Numerics ADD Col3 INTEGER;
Command> DESCRIBE Numerics;

Table USER1.NUMERICS:
  Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          NUMBER (38)

1 table found.
(primary key columns are indicated with *)
```

```
Command> ALTER TABLE Numerics DROP Col3;
Command> ALTER TABLE Numerics ADD COLUMN Col3 TT_INTEGER;
Command> DESCRIBE Numerics;

Table USER1.NUMERICS:
  Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          TT_INTEGER

1 table found.
(primary key columns are indicated with *)
```

```
Command> ALTER TABLE Numerics ADD Col4 TT_INT;
Command> DESCRIBE Numerics;

Table USER1.NUMERICS:
  Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          TT_INTEGER
```

```
    COL4                            TT_INTEGER

1 table found.
(primary key columns are indicated with *)
```

TT_BIGINT type

The TT_BIGINT data type is a signed integer that ranges from -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63}-1$). It requires 8 bytes of storage and thus is more compact than the NUMBER data type. It also has better performance than the NUMBER data type. You cannot specify BIGINT.

Example 1.9 The example ALTERs table Numerics and attempts to add Col5 with a data type of BIGINT; An error is generated. A second ALTER TABLE successfully adds Col5 with a data type of TT_BIGINT.

```
Command> ALTER TABLE Numerics ADD COLUMN Col5 BIGINT;
 3300: BIGINT is not a valid type name; use TT_BIGINT instead
The command failed.
```

```
Command> ALTER TABLE Numerics ADD COLUMN Col5 TT_BIGINT;
Command> DESCRIBE Numerics;
Table USER1.NUMERICS:
  Columns:
    COL1                            TT_TINYINT
    COL2                            TT_SMALLINT
    COL3                            TT_INTEGER
    COL4                            TT_INTEGER
    COL5                            TT_BIGINT

1 table found.
(primary key columns are indicated with *)
```

NUMBER type

The NUMBER data type stores zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} to but not including 1.0×10^{126}. Each NUMBER value requires from 5 to 22 bytes.

Specify a fixed- point number as:

NUMBER (p)

where:

- *p* is the precision or the total number of significant decimal digits, where the most significant digit is the left-most non-zero digit and the least significant digit is the right-most known digit.

- *s* is the scale, or the number of digits from the decimal point to the least significant digit. The scale ranges from -84 to 127.
  - Positive scale is the number of significant digits to the right of the decimal point to and including the least significant digit.
  - Negative scale is the number of significant digits to the left of the decimal point to but not including the least significant digit. For negative scale, the least significant digit is on the left side of the decimal point, because the number is rounded to the specified number of places to the left of the decimal point.
- Scale can be greater than precision; e notation is a common example. When scale is greater than precision, the precision specifies the maximum number of significant digits to the right of the decimal point. For example, if you define your column as type NUMBER (4,5), and you insert.000127 into the column, the value is stored as.00013; A zero is required for the first digit after the decimal point and values after the fifth digit to the right of the decimal point are rounded.
- If a value exceeds the precision, then TimesTen returns an error; If a value exceeds the scale, then TimesTen rounds the value.

NUMBER (p)

This represents a fixed-point number with precision p and scale 0 and is equivalent to NUMBER (p,0).

Specify a floating-point number as:

NUMBER

If you do not specify precision and scale, the maximum precision and scale are used.

Example1.10 The example ALTERs table Numerics adding columns Col6, Col7, Col8, and Col9 defined with the NUMBER data type and specified with different precisians and scales.

```
Command> ALTER TABLE NUMERICS ADD COL6 NUMBER;
Command> ALTER TABLE Numerics ADD Col7 NUMBER (4,2);
Command> ALTER TABLE Numerics ADD Col8 NUMBER (4,-2);
Command> ALTER TABLE Numerics ADD Col8 NUMBER (2,4);
Command> ALTER TABLE Numerics ADD Col9 NUMBER (2,4);
Command> DESCRIBE NUMERICS;

Table USER1.NUMERICS:
  Columns:
    COL1                            TT_TINYINT
    COL2                            TT_SMALLINT
    COL3                            TT_INTEGER
    COL4                            TT_INTEGER
    COL5                            TT_BIGINT
```

```
    COL6                            NUMBER
    COL7                            NUMBER (4,2)
    COL8                            NUMBER (4,-2)
    COL9                            NUMBER (2,4)

1 table found.
(primary key columns are indicated with *)
```

Example1.11 The example CREATEs table NumberCombo and defines columns with the NUMBER data type using different precisions and scales. The value 123.89 is inserted into the columns.

```
Command> CREATE TABLE NumberCombo (Col1 NUMBER, Col2 NUMBER (3),
Col3 NUMBER (6,2), Col4 NUMBER (6,1), Col5 NUMBER (6,-2));
Command> DESCRIBE NumberCombo;

Table USER1.NUMBERCOMBO:
  Columns:
    COL1                            NUMBER
    COL2                            NUMBER (3)
    COL3                            NUMBER (6,2)
    COL4                            NUMBER (6,1)
    COL5                            NUMBER (6,-2)

1 table found.
(primary key columns are indicated with *)
```

```
Command> INSERT INTO NumberCombo VALUES
(123.89,123.89,123.89,123.89,123.89);
1 row inserted.
```

```
Command> VERTICAL ON;
Command> SELECT * FROM NumberCombo;

  COL1:   123.89
  COL2:   124
  COL3:   123.89
  COL4:   123.9
  COL5:   100

1 row found.
```

Example1.12 The example CREATEs a table and defines a column with data type NUMBER (4,2). An attempt to INSERT a value of 123.89 results in an overflow error.

```
Command> CREATE TABLE InvNumberValue (Col6 NUMBER (4,2));
Command> INSERT INTO InvNumberValue VALUES (123.89);
 2923: Number type value overflow
```

```
The command failed.
```

Example1.13 The example CREATEs a table and defines columns with the NUMBER data type using a scale that is greater than the precision. Values are inserted into the columns.

```
Command> CREATE TABLE NumberCombo2 (Col1 NUMBER (4,5),
Col2 NUMBER (4,5), Col3 NUMBER (4,5), Col4 NUMBER (2,7),
Col5 NUMBER (2,7), Col6 NUMBER (2,5), Col7 NUMBER (2,5));

Command> INSERT INTO NumberCombo2 VALUES (.01234, .00012, .000127,
.0000012, .00000123, 1.2e-4, 1.2e-5);
1 row inserted.
```

```
Command> DESCRIBE NumberCombo2;

Table USER1.NUMBERCOMBO2:
  Columns:
    COL1                            NUMBER (4,5)
    COL2                            NUMBER (4,5)
    COL3                            NUMBER (4,5)
    COL4                            NUMBER (2,7)
    COL5                            NUMBER (2,7)
    COL6                            NUMBER (2,5)
    COL7                            NUMBER (2,5)

1 table found.
(primary key columns are indicated with *)
```

```
Command> SELECT * FROM NumberCombo2;

  COL1:   .01234
  COL2:   .00012
  COL3:   .00013
  COL4:   .0000012
  COL5:   .0000012
  COL6:   .00012
  COL7:   .00001

1 row found.
```

Floating-Point numbers

Floating -point numbers can have a decimal point or can have no decimal point. An exponent may be used to increase the range (for example, 1.2 e^{-20}). Floating- point numbers do not have a scale because the number of digits that can appear after the decimal point is not restricted.

Binary floating -point numbers are stored using binary precision (the digits 0 and 1). For the NUMBER data type, values are stored using decimal precision (the digits 0 through 9).

Literal values that are within the range and precision supported by NUMBER are stored as NUMBER because literals are expressed using decimal precision.

BINARY_FLOAT

BINARY_FLOAT is a 32-bit single- precision floating -point number.

BINARY_DOUBLE

BINARY_DOUBLE is a 64-bit double- precision floating- point number.

Both BINARY_FLOAT and BINARY_DOUBLE support the special values Inf, -Inf and NaN (not a number) and conform to the IEEE standard.

Floating-point number limits:

- BINARY_FLOAT
  - Minimum positive finite value: 1.17549E-38F
  - Maximum positive finite value: 3.40282E+38F
- BINARY_DOUBLE
  - Minimum positive finite value: 2.22507485850720E-308
  - Maximum positive finite value: 1.79769313486231E+308

Example1.14 The example CREATEs a table and defines two columns with the BINARY_FLOAT and BINARY_DOUBLE data types.

```
Command> CREATE TABLE BfBd (Col1 BINARY_FLOAT, Col2 BINARY_DOUBLE);
Command> DESCRIBE BfBd;

Table UISER1.BFBD:
  Columns:
    COL1                            BINARY_FLOAT
    COL2                            BINARY_DOUBLE

1 table found.
(primary key columns are indicated with *)
```

FLOAT and FLOAT (n)

TimesTen also supports the ANSI type FLOAT. FLOAT is an exact numeric type and is implemented as the NUMBER type. The number *n* indicates the number of bits of precision the value can store. The value ranges from 1 to 126. To convert from binary precision to decimal precision, multiply *n* by 0.30103. To convert from decimal precision to binary precision, multiple the decimal

precision by 3.32193. The maximum 126 digits of binary precision is equivalent to approximately 38 digits of decimal precision.

Binary and Varbinary types

The BINARY data type is a fixed-length binary value with a length of *n* bytes. The value of *n* ranges from 1 to 8300 bytes. The BINARY data type requires *n* bytes of storage. Data is padded to the maximum column size with trailing zeros. Zero padded comparison semantics are used.

The VARBINARY data type is a variable-length binary value having a maximum length of n bytes. The value of n ranges from 1 to 4,194,304 (2^{22}) bytes.

Example1.15 The example CREATEs a table and defines 2 columns. Col1 is defined with data type BINARY and Col2 is defined with data type VARBINARY.

```
Command> CREATE TABLE BVar (Col1 BINARY (10), Col2 VARBINARY (10));
Command> DESCRIBE BVar;

Table USER1.BVAR:
  Columns:
    COL1                            BINARY (10)
    COL2                            VARBINARY (10) INLINE

1 table found.
(primary key columns are indicated with *)
```

Numeric precedence

The result type of an expression is determined by the operand with the highest type precedence. For example, the SUM of TT_INTEGER and BINARY_FLOAT types results in type BINARY_FLOAT because BINARY_FLOAT has higher numeric precedence than TT_INTEGER. Similarly, the product of NUMBER and BINARY_DOUBLE types result in type BINARY_DOUBLE because BINARY_DOUBLE has higher precedence than NUMBER.

The numeric precedence order is as follows (highest to lowest):

- BINARY_DOUBLE
- BINARY_FLOAT
- NUMBER
- TT_BIGINT
- TT_INTEGER
- TT_SMALLINT
- TT_TINYINT

Datetime data types

The datetime data types are:

- TIME (a time only data type)
- TT_DATE
- DATE
- TT_TIMESTAMP
- TIMESTAMP

TIME type

The format of a TIME value is HH:MI:SS and ranges from 00:00:00 (12:00:00 AM to 23:59:59 (11:59:59 PM). The TIME data type requires 8 bytes of storage.

TT_DATE type

The format of a TT_DATE value is YYYY-MM-DD and ranges from 1753-01-01 (January 1, 1753) to 9999-12-31 (December 31, 9999 AD). The TT_DATE data type requires 4 bytes of storage.

DATE type

The format of a DATE value is YYYY-MM-DD HH:MI:SS and ranges from -4712-01-01 (January 1, 4712 BC) to 9999-12-31 (December 31, 9999 AD). There are no fractional seconds. The DATE type requires 7 bytes of storage.

TT_TIMESTAMP type

The format of a TT_TIMESTAMP value is YYYY-MM-DD HH:MI:SS [.FFFFFF]. The fractional seconds precision is 6. The range is from 1753-01-01 00:00:00 (January 1, 1753 midnight) to 9999-12-31 23:59:59 (December 31, 9999 11:59:59 PM). The TT_TIMESTAMP type requires 8 bytes of storage. TT_TIMESTAMP is faster than the TIMESTAMP data type and has a smaller storage size than the TIMESTAMP type.

TIMESTAMP type

The format of a TIMESTAMP value is YYYY-MM-DD HH:MI:SS [.FFFFFFFFF]. The fractional seconds precision range is 0 to 9; the default is 6. The date range is from -4712-01-01 (January 1, 4712 BC) to 9999-12-31 (December 31, 9999 AD). The TIMESTAMP type requires 12 bytes of storage. The TIMESTAMP type has a larger date range than the TT_TIMESTAMP and supports more precision than the TT_TIMESTAMP.

TimesTen interval

Using INTERVAL types

If you are using TimesTen type mode, for information on INTERVAL, refer to documentation from previous releases of TimesTen.

TimesTen supports interval type only in a constant specification and intermediate expression result. Interval type can not be the final result. Columns cannot be defined with an INTERVAL type. See "Type specifications" on page 8.

You can specify a single-field interval literal in an expression, but you cannot specify a complete expression that returns an interval data type.

TimesTen supports interval literals of the form:

INTERVAL [+\-] *CharString IntervalQualifier*

Using DATE and TIME types

This section shows some DATE, TIME and TIMESTAMP data type examples:

Example1.16 To create a table named SAMPLE that contains both a column named DCOL with the type DATE and a column named TCOL with the type TIME, use:

```
CREATE TABLE SAMPLE (TCOL TIME, DCOL DATE);
```

Example1.17 To insert DATE and TIME values into the table SAMPLE, use:

```
INSERT INTO SAMPLE VALUES
(TIME '12:00:00', DATE '1998-10-28');
```

Example1.18 To select all rows in the table SAMPLE that are between noon and 4:00 p.m. on October 29, 1998, use:

```
SELECT * FROM SAMPLE WHERE DCOL = DATE '1998-10-29' AND
TCOL BETWEEN TIME '12:00:00' AND TIME '16:00:00';
```

Example1.19 To create a table named SAMPLE that contains a column named TSCOL with the type TIMESTAMP and then select all rows in the table that are between noon and 4:00 p.m. on October 29, 1998, use the statements:

```
CREATE TABLE SAMPLE2 (TSCOL TIMESTAMP);

INSERT INTO SAMPLE2 VALUES (TIMESTAMP '1998-10-28 12:00:00');

SELECT * FROM SAMPLE2
WHERE TSCOL
BETWEEN TIMESTAMP '1998-10-29 12:00:00'
AND '1998-10-29 16:00:00';
```

Note: TimesTen allows both literal and string formats of the TIME, DATE and TIMESTAMP types. For example, TimeString ('12:00:00') and TimeLiteral (Time '16:00:00') are both valid ways to specify a TIME value. TimesTen reads the first value as CHAR type and then later converts it to TIME type as needed. TimesTen reads the second value as TIME. The examples above use the literal format. Any values for the fraction not specified in full microseconds result in a "Data truncated" error.

Handling TIMEZONE conversions

TimesTen currently does not support TIMEZONE. TIME/TIMESTAMP data type values are stored without making any adjustment for time difference. Applications must assume one time zone and convert TIME/TIMESTAMP to that time zone before sending values to the database. For example, an application can assume its TIMEZONE to be Pacific Standard Time. If the application is using TIME/TIMESTAMP values from the Pacific Daylight Time or Eastern Daylight/Standard Time, the application must convert TIME/TIMESTAMP to Pacific Standard Time.

Date-time and interval types in arithmetic operations

If you are using TimesTen type mode, for information on Date-Time and Interval types in arithmetic operations, refer to documentation from previous releases of TimesTen.

Date-time refers to types DATE, TIME, and TIMESTAMP. Date and time arithmetic is supported with the following syntax:

TimeVal1 - *TimeVal2* or *TimestampVal1* - *TimestampVal2* or *DateVal1* - *DateVal2* returns the difference as an interval day to second.

TT_DateVal1 - *TT_DateVal2* returns the number of days difference as an integer.

DateTimeVal {+|-} *IntervalVal* or

IntervalVal + *DateTimeVal* or

IntervalVal1 {+|-} *IntervalVal2* or

IntervalVal {*|/} *NumericVal* or

NumericVal * *IntervalVal*

INTERVAL type cannot be the final result of a complete expression. Extract function must be used to extract the desired component of this interval result.

The following table lists the type that results from each operation:

Operand 1	Operator	Operand 2	Result type
TIME | DATE | TIMETAMP	-	TIME| DATE | TIMESTAMP	INTERVAL DAY TO SECOND
TT_DATE	-	TT_DATE	TT_BIGINT (Number of Days)
Date-time	+ or -	INTERVAL	Date-time
INTERVAL	+	Date-time	Date-time
INTERVAL	+ or -	INTERVAL	INTERVAL
INTERVAL	* or /	Numeric	INTERVAL
Numeric	*	INTERVAL	INTERVAL

Example1.20

```
SELECT TT_DATE1 - TT_DATE2 FROM t1;
SELECT EXTRACT(DAY FROM TIMESTAMP1-TIMESTAMP2) FROM t1;
SELECT * FROM t1 WHERE TIMESTAMP1 -TIMESTAMP2 =
NUMTODSINTERVAL(10, 'DAY');

SELECT SYSDATE + NUMTODSINTERVAL(20,'SECOND') FROM dual;

SELECT EXTRACT (SECOND FROM TIMESTAMP1-TIMESTAMP2) FROM dual;

/* select the microsecond difference between two timestamp values d1
and d2 */

SELECT 1000000*(EXTRACT(DAY FROM d1-d2)*24*3600+
EXTRACT(HOUR FROM d1-d2)*3600+
EXTRACT(MINUTE FROM d1-d2)*60+EXTRACT(SECOND FROM d1-d2)) FROM d1;
```

Example1.21

The example inserts timestamp values into 2 columns and then subtracts the two values using the EXTRACT function:

```
Command> CREATE TABLE ts (id TIMESTAMP, id2 TIMESTAMP);
Command> INSERT INTO ts VALUES (timestamp '2007-01-20 12:45:23',
timestamp '2006-12-25 17:34:22');
1 row inserted.
Command> SELECT EXTRACT (DAY FROM id - id2) from ts;
< 25 >
1 row found.
```

Example1.22 The following queries return errors:

```
You cannot select an INTERVAL result:

SELECT TIMESTAMP1 -TIMESTAMP2 FROM t1

SELECT DATE1 - DATE2 FROM t1;;
```

You cannot compare an interval year to month with an interval day to second:

```
SELECT * FROM t1 WHERE TIMESTAMP1 -TIMESTAMP2 =
NUMTOYMINTERVAL(10, 'YEAR');
```

You cannot compare and INTERVAL DAY TO SECOND with and INTERVAL DAY:

```
SELECT * FROM t1 WHERE TIMESTAMP1 -TIMESTAMP2 = INTERVAL '10'
DAY;
```

You cannot extract YEAR from an INTERVAL day to second:

```
SELECT EXTRACT (YEAR FROM TIMESTAMP1-TIMESTAMP2) FROM dual;
```

Restrictions on date-time and interval arithmetic operations

The following restrictions must be considered when performing date-time and interval arithmetic:

- The results for addition and subtraction with DATE and TIMESTAMP types for INTERVAL YEAR, INTERVAL MONTH are not closed. For example, adding 1 year to the DATE or TIMESTAMP of '2004-02-29' results in a Date arithmetic error (TimesTen error message 2787) because February 29, 2005 does not exist (2005 is not a leap year). Adding INTERVAL '1' month to DATE '2005-01-30' will also result in an the same error because February never has 30 days.
- The results are closed for INTERVAL DAY.

Storage requirements

Variable-length columns whose declared column length is > 128 bytes are stored out of line. Variable-length columns whose declared column length is <= 128 bytes are stored inline. For character semantics, the number of bytes stored out of line is dependent on the character set. For example, for a character set with 4 bytes per character, variable-length columns whose declared column length is > 32 (128/4) are stored out of line.

The storage requirements of the various data types are:

Type	Storage required
CHAR (*n* [BYTE|CHAR])	*n* bytes or if character semantics, n characters. If character semantics, the length of the column (*n*) is based on length semantics and character set.
VARCHAR2 (*n* [BYTE|CHAR])	For NOT INLINE columns: On 32-bit platforms, length of value + 20 bytes (minimum of 28 bytes). On 64-bit platforms, length of value + 24 bytes (minimum of 40 bytes). For INLINE columns: On 32-bit platforms, n + 4 bytes. On 64-bit platforms, n + 8 bytes. If character semantics, the length of the column (*n*) is based on length semantics and character set.
NCHAR*(n)*	Bytes required is 2* n where n is the number of characters.
NVARCHAR2 (*n)*	For NOT INLINE columns: On 32-bit platforms, 2*(length of value) + 20 bytes (minimum of 28 bytes). On 64-bit platforms, 2*(length of value) + 24 bytes (minimum of 40bytes). For INLINE columns: On 32-bit platforms, 2*(length of column) + 4 bytes. On 64-bit platforms, 2*(length of column) + 8 bytes.
TT_TINYINT	1 byte.
TT_SMALLINT	2 bytes.
TT_INT[EGER]	4 bytes.
TT_BIGINT	8 bytes.

Type	Storage required
NUMBER	5 to 22 bytes.
BINARY_FLOAT	4 bytes.
BINARY_DOUBLE	8 bytes.
TT_DECIMAL(*p*,*s*)	Approximately *p*/2 bytes.
TT_TIME	8 bytes.
TT_DATE	4 bytes.
DATE	7 bytes.
TT_TIMESTAMP	8 bytes.
TIMESTAMP	12 bytes.
BINARY (*n*)	*n* bytes.
VARBINARY (*n*)	For NOT INLINE columns: On 32-bit platforms, length of value + 20 bytes (minimum of 28 bytes). On 64-bit platforms, length of value + 24 bytes (minimum of 40 bytes). For INLINE columns: On 32-bit platforms, length of column + 4 bytes. On 64-bit platforms, length of column + 8 bytes.
INTERVAL	An INTERVAL cannot be stored in TimesTen.

Data type comparison rules

The following section describes how values of each data type are compared in TimesTen.

Numeric values

A larger value is greater than a smaller value. -1 is less than 10 and -10 is less than -1.

The floating-point value NaN is greater than any other numeric value and is equal to itself.

Date values

A later date is considered greater than an earlier one. For example, the date equivalent of '10-AUG-2005' is less than that of '30-AUG-2006' and '30-AUG-2006 1:15pm' is greater than '30-AUG-2006 10:10am'.

Character values

Character values are compared by:

- Binary or linguistic sorting
- Blank-padded or nonpadded comparison semantics

Binary and linguistic sorting

In binary sorting, TimesTen compares character strings according to the concatenated value of the numeric codes of the characters in the database character set. One character is greater than the other if it has a greater numeric values than the other in the character set. Blanks are less than any character.

Linguistic sorting is useful if the binary sequence of numeric codes does not match the linguistic sequence of the characters you are comparing. In linguistic sorting, SQL sorting and comparison are based on the linguistic rule set by NLS_SORT. For more information on linguistic sorts, see Linguistic sorts in the Operations Guide.

The default is binary sorting.

Blank-Padded and nonpadded comparison semantics

With blank-padded semantics, if two values have different lengths, TimesTen adds blanks to the shorter value until both lengths are equal. Values are then compared character by character up to the first character that differs. The value with the greater character in the first differing position is considered greater. If two values have no differing characters, then they are considered equal. Thus, two values are considered equal if they differ only in the number of trailing blanks.

Blank-padded semantics are used when both values in the comparison are expressions of type CHAR or NCHAR or text literals.

With nonpadded semantics, two values are compared character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values that have differing lengths are identical up to the end of the shorter one, then the longer one is considered greater. If two values of equal length have no differing characters, they are considered equal.

Nonpadded semantics are used when both values in the comparison have the type VARCHAR2 or NVARCHAR2.

As an example, with blank-padded semantics:

- ‘a ‘ = ‘a’

With nonpadded semantics:

- ‘a ‘ > ‘a’

Data conversion

Generally an expression cannot contain values of different data types. However, TimesTen supports both implicit and explicit conversion from one data type to another. Explicit conversion is recommended.

Implicit Data Conversion

The following rules apply:

- Conversions between exact numeric values (TT_TINYINT, TT_SMALLINT, TT_INTEGER, TT_BIGINT, NUMBER) and floating-point values (BINARY_FLOAT, BINARY_DOUBLE) can be inexact because the exact numeric values use decimal precision whereas the floating-point numbers use binary precision.
- When comparing a character value with any date, time, or datetime value, TimesTen converts the character data to the date, time, or datetime value.
- Implicit and explicit CHAR/VARCHAR2 <-> NCHAR/NVARCHAR2 conversions are supported except when the character set is TIMESTEN8. An example of explicit conversion:

```
Command> CREATE TABLE ConvDemo (c1 CHAR (10), x1 TT_INTEGER);
Command> CREATE TABLE ConvDemo2 (c1 NCHAR (10), x2 TT_INTEGER);
Command> INSERT INTO ConvDemo VALUES ('ABC', 10);
1 row inserted.
Command> INSERT INTO ConvDemo VALUES ('def', 100);
1 row inserted.
Command> INSERT INTO ConvDemo2 SELECT * FROM ConvDemo;
2 rows inserted.
Command> SELECT x1,x2,convdemo.c1, convdemo2.c1 FROM ConvDemo,
         ConvDemo2 where ConvDemo.c1 = ConvDemo2.c1;
X1, X2, C1, C1
< 10, 10, ABC       , ABC        >
< 100, 100, def       , def        >
2 rows found.
```

NULL values

A NULL value indicates the absence of a value; it is a placeholder for a value that is missing. Any column in a table or parameter in an expression, regardless of its data type, can contain NULL values unless you specify NOT NULL for the column when you create the table.

The following properties of NULL values affect operations on rows, parameters, or local variables:

- NULL values always sort highest in a sequence of values.

- Two NULL values are not equal to each other except in a GROUP BY or SELECT DISTINCT operation.
- An expression containing a NULL value evaluates to NULL; for example, (5 - Col), where Col is NULL, evaluates to NULL.

Because of these properties, TimesTen ignores columns, rows, or parameters containing NULL values:

- When joining tables if the join is on a column containing NULL values.
- When executing aggregate functions.

In several SQL predicates, described in Chapter 4, "Search Conditions, you can explicitly test for NULL values. In an application, you can use the ODBC functions **SQLBindCol**, **SQLBindParameter**, **SQLGetData**, and **SQLParamData,** or you can use the JDBC functions **PreparedStatement.setNull** and **ResultSet.getXXXX** with **ResultSet.wasNull** to handle input and output of NULL values.

INF and NAN

TimesTen supports the IEEE floating-point values Inf (positive infinity), -Inf (negative infinity) and NaN (not a number).

Constant Values:

The following constant values are supported. You can use these values in places where a floating-point constant is allowed:

- BINARY_FLOAT_INFINITY
- -BINARY_FLOAT_INFINITY
- BINARY_DOUBLE_INFINITY
- -BINARY_DOUBLE_INFINITY
- BINARY_FLOAT_NAN
- BINARY_DOUBLE_NAN

In the following example, a table is created with a column of type BINARY_FLOAT and a column of type TT_INTEGER. BINARY_FLOAT_INFINITY and BINARY_FLOAT_NAN are inserted into the column of type BINARY_FLOAT.

```
Command> CREATE TABLE BfDemo (Id BINARY_FLOAT, Id2 TT_INTEGER);
Command> INSERT INTO BfDemo VALUES (BINARY_FLOAT_INFINITY, 50);
1 row inserted.
Command> INSERT INTO BfDemo VALUES (BINARY_FLOAT_NAN, 100);
1 row inserted.
Command> SELECT * FROM BfDemo;
< INF, 50 >
< NAN, 100 >
2 rows found.
```

Primary Key Values

Inf, -Inf, and NaN are acceptable values in columns defined with a primary key. This is a deviation in behavior from NULL; NULL values are not allowed on columns defined with a primary key.

You can only insert Inf, -Inf, and NaN values into BINARY_FLOAT and BINARY_DOUBLE columns.

Selecting Inf and NaN (Floating-Point Conditions):

Floating-point conditions determine whether an expression is infinite or is the undefined result of an operation (NaN or “not a number”).

The syntax:

Expression IS [NOT] {NAN| INFINITE};

Expression must either resolve to a numeric data type or to a data type that can be implicitly converted to a numeric data type.

The following table describes the floating-point conditions:

Condition	Operation	Example
IS [NOT] NAN	Returns TRUE if *Expression* is the value NaN when NOT is not specified. Returns TRUE if *Expression* is not the value NaN when NOT is specified.	SELECT * FROM *BfDemo* WHERE *Id* IS NOT NAN; ID, ID2 < INF, 50 > 1 row found.
IS [NOT] INFINITE	Returns TRUE if *Expression* is the value +INF or -INF when NOT is not specified. Returns TRUE if *Expression* is neither +INF nor -INF when NOT is specified.	SELECT * FROM *BfDemo* WHERE *Id* IS NOT INFINITE; ID, ID2 < NAN, 100 > 1 row found.

Note: The constant keywords represent specific BINARY_FLOAT and BINARY_DOUBLE values; the comparison keywords correspond to properties of a value and are not specific to any type (although they can only evaluate to true for BINARY_FLOAT or BINARY_DOUBLE types or types that can be converted to BINARY_FLOAT or BINARY_DOUBLE).

Comparisons with Inf and NaN

The following rules apply:

- Comparison between Inf (or -Inf) and a finite value are as expected. For example, 5 > -Inf.
- (Inf = Inf) and (Inf > -Inf) both evaluate to True.
- (NaN = NaN) evaluates to True.

In reference to collating sequences:

- -Inf sorts lower than any other value.
- Inf sorts higher than any other value, but lower than Nan and NULL.
- NaN sorts higher than Inf.
- NULL sorts higher than NaN; NULL is always the largest value in any collating sequence.

Expressions involving Inf and NaN

- Expressions containing floating-point values may generate Inf, -Inf, or NaN. This can occur either because the expression generated overflow or exceptional conditions or because one or more of the values in the expression was Inf, -Inf, or NaN. Inf and NaN are generated in overflow or division by 0 conditions.
- Inf, -Inf, and NaN values are not ignored in aggregate functions; NULL values are. If you wish to exclude Inf and NaN from aggregates (or from any selection), use both the IS NOT NAN and IS NOT INFINITE predicates.

Overflow and truncation

Some operations can result in data overflow or truncation. Overflow results in an error and can generate Inf. Truncation results in loss of least significant data.

Exact values are truncated only when they are stored in the data store by an **INSERT** or **UPDATE** statement, and if the target column has smaller scale than the value. TimesTen returns a warning when such truncation occurs. If the value does not fit because of overflow, TimesTen returns the special value Inf and the value is not inserted.

TimesTen may truncate approximate values during computation and when the values are inserted into the data store or when data store values are updated. TimesTen returns an error only upon insertion or update. When overflow with approximate values occurs, TimesTen returns the special value Inf.

There are several circumstances that can cause overflow:

- ***During arithmetic operations.*** Overflow can occur when multiplication results in a number larger than the maximum value allowable in its type. Arithmetic operations are defined in Chapter 3.
- ***When using aggregate functions.*** Overflow can occur when the sum of several numbers exceeds the maximum allowable value of the result type. Aggregate functions are defined in Chapter 3.
- ***During type conversion.*** Overflow can also occur when, for example, a TT_INTEGER value is converted to a TT_SMALLINT value.

Truncation can cause an error or warning for alphanumeric or numeric data types:

- ***Character data.*** An error occurs if a string is truncated because it is too long for its target type. For NCHAR and NVARCHAR2 types, truncation always occurs on Unicode character boundaries. In the NCHAR data types, a single-byte value (half a Unicode character) has no meaning and is not possible.
- ***Numeric data.*** A warning occurs when any trailing non-zero digit is dropped from the fractional part of a numeric value.

Underflow

When an approximate numeric value is too close to zero to be represented by the hardware, TimesTen underflows to zero and returns a truncation warning.

Replication limits

TimesTen places the following limits on the size of data types in a data store that is being replicated:

- VARCHAR2 and VARBINARY columns cannot exceed 256,000 bytes. For character length semantics, the limit is 256,000 bytes. How many characters

that translates into depends on the database character set. Worst case is 256,000 bytes / 4 = 64,000 characters.

- NVARCHAR2 columns cannot exceed 128,000 characters (256,000 bytes).

TimesTen Type Mode (Backward Compatibility)

TimesTen supports a data type backward compatibility mode (called TimesTen type mode). The type mode is a data store creation attribute. TypeMode = 1 indicates TimesTen mode.

For more information on types modes, see "TypeMode" in *Oracle TimesTen In-Memory Database API Reference Guide*.

For information on data type usage in TimesTen type mode, refer to documentation from previous releases of TimesTen.

Data types supported in TimesTen type mode

Data type	Description
CHAR[ACTER] [(*n* [BYTE|CHAR])]	Fixed-length character string of length *n* bytes or characters. Default is 1 byte. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 8300 bytes. CHAR indicates that the column has character length semantics; The minimum CHAR length is 1 character. The maximum CHAR length depends on how many characters fit in 8300 bytes; this is determined by the DatabaseCharacterSet in use. For character set AL32UTF8, up to four bytes per character may be needed, so the CHAR length limit ranges from 2075 to 8300 depending on the character set. A zero-length string is a valid non-NULL value. CHAR data is padded to the maximum column size with trailing blanks; Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47. You can alternatively specify TT_CHAR [(*n* [BYTE|CHAR])].

Data type	Description
NCHAR[(*n*)]	Fixed-length string of *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; Nchar character limits are 1/2 the byte limits so the maximum size is 4150; Default and minimum bytes of storage is 2*n* (2). A zero-length string is a valid non-NULL value. NCHAR data is padded to the maximum column size with U+0020 SPACE. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47. You can alternatively specify TT_NCHAR[(*n*)]. NATIONAL CHARACTER and NATIONAL CHAR are synonyms for NCHAR.
VARCHAR (*n* [BYTE|CHAR])	Variable-length character string having maximum length *n* bytes or characters. You must specify *n*. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 4194304 (2^{22}) bytes. CHAR indicates that the column has character length semantics. A zero-length string is a valid non-NULL value. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47. You can alternatively specify TT_VARCHAR (*n* [BYTE|CHAR]).

Data type	Description
NVARCHAR(*n*)	Variable-length string of *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; NVARCHAR character limits are 1/2 the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify n. A zero-length string is a valid non-NULL value. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47. You can alternatively specify TT_NVARCHAR(*n*). NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, and NCHAR VARYING are synonyms for NVARCHAR.
TINYINT	Unsigned integer ranging from 0 to 255 (2^8-1). Since TINYINT is unsigned, the negation of a TINYINT is SMALLINT. You can alternatively specify TT_TINYINT.
SMALLINT	A native signed 16 bit integer in the range –32,768 –(2^{15}) to 32,767 (2^{15}–1). You can alternatively specify TT_SMALLINT.
INT[EGER]	A signed integer in the range –2,147,483,648 --(2^{31}) to 2,147,483,647 (2^{31}–1). You can alternatively specify TT_INTEGER.

Data type	Description
BIGINT	A signed 8-byte integer in the range -9,223,372,036,854,775,808 -(2^{63}) to 9,223,372,036,854,775,807 (2^{63} - 1). You can alternatively specify TT_BIGINT.
BINARY_FLOAT	32-bit floating-point number; BINARY_FLOAT is a single-precision native floating-point type; Supports +Inf, -Inf and NaN values. BINARY_FLOAT is an approximate numeric value consisting of an exponent and mantissa; You can use Exponential or E-notation. BINARY_FLOAT has binary precision 24. Minimum positive finite value: 1.17549E-38F Maximum positive finite value: 3.40282E+38F You can alternatively specify REAL or FLOAT (24).
BINARY_DOUBLE	64-bit floating -point number. BINARY_DOUBLE is a double-precision native floating point number; Supports +Inf, -Inf and Nan values. BINARY_DOUBLE is an approximate numeric value consisting of an exponent and mantissa; You can use Exponential or E-notation. BINARY_DOUBLE has binary precision 53. Minimum positive finite value: 2.22507485850720E-308 Maximum positive finite value: 1.79769313486231E+308 You can alternatively specify DOUBLE [PRECISION] or FLOAT [(53)].

Data type	Description
DEC[IMAL][(*p*[,*s*])] or NUMERIC[(*p*[,*s*])]	An exact numeric value with a fixed maximum precision (total number of digits) and scale (number of digits to the right of the decimal point). The precision *p* must be between 1 and 40. The scale must be between 0 and *p*. The default precision is 40 and the default scale is 0.
BINARY (*n*)	Fixed-length binary value of n bytes; *n* ranges from 1 to 8300. BINARY data is padded to the maximum column size with trailing zeroes.
VARBINARY (*n*)	Variable-length binary value having maximum length *n* bytes; *n* ranges from 1 to 4194304 (2^{22}).
TIME	A time of day between 00:00:00 (12 midnight) and 23:59:59 (11:59:59 pm), inclusive. The format is: HH:MI:SS; Storage size is 8 bytes.
DATE	Stores date information: century, year, month, date. The format is YYYY-MM-DD; MM is expressed as an integer; For example, 2006-10-28. Storage size is 4 bytes. Valid dates are between 1753-01-01 (January 1,1753) and 9999-12-31 (December 31, 9999). You can alternatively specify TT_DATE.
TIMESTAMP	A data and time between 1753-01-01 00:00:00 (January 1, 1753 midnight) and 9999-12-31 23:59:59 pm (11:59:59 pm on December 31, 9999), inclusive. Any values for the fraction not specified in full microseconds result in a “Data Truncated” error. The format is YYYY-MM-DD HH:MI:SS [.FFFFFF]. Storage size is 8 bytes. You can alternatively specify TT_TIMESTAMP or [TT_]TIMESTAMP (6).

Data type	Description
INTERVAL [+/-] *IntervalQualifier*	TimesTen partially supports INTERVAL types, expressed with the type INTERVAL and an *IntervalQualifier*. An *IntervalQualifier* can only specify a single field type with no precision. The default leading precision is 8 digits for all INTERVAL types. The single field type can be one of: YEAR, MONTH, DAY, HOUR, MINUTE or SECOND. Currently, INTERVAL type can be specified only with a constant.

Oracle data types supported in TimesTen type mode

Data type	Description
ORA_CHAR [(*n* [BYTE|CHAR])]	Fixed-length character string of length *n* bytes or characters. Default is 1 byte. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 8300 bytes. CHAR indicates that the column has character length semantics; The minimum CHAR length is 1 character. The maximum CHAR length depends on how many characters fit in 8300 bytes; this is determined by the DatabaseCharacterSet in use. For character set AL32UTF8, up to four bytes per character may be needed, so the CHAR length limit ranges from 2075 to 8300 depending on the character set. A zero-length string is interpreted as NULL. ORA_CHAR data is padded to the maximum column size with trailing blanks; Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47.
ORA_NCHAR[(*n*)]	Fixed-length string of length *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; Nchar character limits are 1/2 the byte limits so the maximum size is 4150; Default and minimum bytes of storage is 2*n* (2). A zero-length string is interpreted as NULL. ORA_NCHAR data is padded to the maximum column size with U+0020 SPACE; Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47.

Data type	Description
ORA_VARCHAR2 (*n* [BYTE|CHAR])	Variable-length character string having maximum length *n* bytes or characters. BYTE indicates that the column has byte length semantics; *n* ranges from a minimum of 1 byte to a maximum 4194304 (2^{22}) bytes. You must specify *n.* CHAR indicates that the column has character length semantics. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47.
ORA_NVARCHAR2(*n*)	Variable-length string of *n* two-byte Unicode characters. The number of bytes required is 2* *n* where *n* is the specified number of characters; ORA_NVARCHAR2 character limits are 1/2 the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify n. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-Padded and nonpadded comparison semantics" on page 47.

Data type	Description
NUMBER [(precision [,scale])]	Number having precision and scale. The precision ranges from 1 to 38 decimal. The scale ranges from -84 to 127. Both precision and scale are optional. If you do not specify a precision or a scale, then maximum precision of 38 and flexible scale are assumed. NUMBER supports scale > precision and negative scale. NUMBER stores zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} to (but not including) 1.0×10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0×10^{126}, then TimesTen returns an error. In TimesTen type mode, the NUMBER data type stores 10e-89 as its smallest (closest to zero) value.
ORA_DATE	Stores date and time information: century, year, month, date, hour, minute and second: Format is: YYYY-MM-DD HHMMSS. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. The storage size is 7 bytes. There are no fractional seconds.
ORA_TIMESTAMP [(*fractional_seconds_precision*)]	Stores year, month, and day values of the date data type plus hour, minute, and second values of time. *Fractional_seconds_precision* is the number of digits in the fractional part of the seconds field. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. The fractional seconds precision range is 0 to 9; The default is 6. Format is: YYYY-MM-DD HH:MI:SS [.FFFFFFFFF] Storage size 12 bytes.

2

Names

This chapter presents general rules for names used in TimesTen commands.

Basic names

Basic names identify columns, tables, views and indexes. Basic names must follow these rules:

- The maximum length of a basic name is 30 characters.
- A name can consist of any combination of letters (A to Z a to z), decimal digits (0 to 9), $, #, @, or underscore (_). For identifiers, the first character must be a letter (A-Z a-z) and not a digit or special character. However, for parameter names, the first character can be a letter (A-Z a-z), a decimal digit (0 to 9), or special characters $, #, @, or underscore (_).
- Lower case letters (a to z) are automatically changed to the corresponding upper case letters (A to Z).
- If you enclose a name in quotation marks, you can use any combination of characters (even if they are not in the set of legal characters). In that case, the first character can also be any character. If a column, table, or index is initially defined with a name enclosed in quotation marks and the name does not conform to the rule noted in the second bullet, then that name must always be enclosed in quotation marks whenever it is subsequently referenced.
- Unicode characters are not allowed in names.

Owner names

The *owner name* is the user name of the account that created the table. Tables and indexes defined by TimesTen itself have the owner *SYS or TTREP.* User objects cannot be created with owner names *SYS or TTREP.* TimesTen converts all owner and table names to upper case.

Owners of tables in TimesTen are determined by the user ID settings or login names. For cache groups, Oracle table owner names must always match TimesTen table owner names.

Owner names may be specified by the user during table creation, in addition to being automatically determined if they are left unspecified. See “CREATE

TABLE" on page 251. When creating owner names, follow the same rules as those for creating "Basic names".

Compound identifiers

Basic names and user names are simple names. In some cases, simple names are combined to form a *compound identifier*, which consists of an owner name combined with one or more basic names, with periods (.) between them.

In most cases you can abbreviate a compound identifier by omitting one of its parts. If you do not use a fully qualified name, a default value is automatically used in place of the missing part. For example, if you omit the owner name (and the period) when you refer to tables you own, TimesTen generates the owner name by using your login name.

A complete compound identifier, including all of its parts, is called a *fully qualified name*. Different owners can have tables and indexes with the same name; the fully qualified name of these objects must be unique.

The following are compound identifiers:

- ***Column identifier:*** [[*Owner.*]*TableName*.]*ColumnName*
- ***Index identifier:*** [*Owner.*]*IndexName*
- ***Table identifier****:* [*Owner.*]*TableName*
- ***Row identifier:*** [[*Owner.*]TableName.]ROWID

Dynamic parameters

Dynamic parameters are used to pass information between an application program and TimesTen. They are placeholders in SQL commands and are replaced at runtime with actual values.

A dynamic parameter name must be preceded by a colon (:) when used in a SQL command and must conform to the TimesTen rules for basic names. However, unlike identifiers, parameter names can start with any of the following characters:

- letters: A to Z
- letters: a to z.
- digits: 0 to 9
- special characters: # $ @ or underscore (_)

Note: Instead of using a:*DynamicParameter* sequence, the application can also use a ? for each dynamic parameter. See Example 3.1 on page 74.

3

Expressions

Expressions are used for the following purposes:

- The select list of the **SELECT** statement
- A condition of the WHERE clause and HAVING clause
- The GROUP BY and ORDER BY clauses
- The VALUES clause of the **INSERT** and **MERGE** statements
- The SET clause of the **UPDATE** and **MERGE** statements

ROWID specification

TimesTen assigns a unique ID called a ROWID to each row stored in a table. ROWID has type BINARY(16).

Because ROWID is not a real column, it does not require database space and cannot be updated, indexed or dropped.

The ROWID value persists throughout the life of the table row, but the system can reassign the ID to a different row after the original row is deleted. Zero is not a valid value for ROWID.

ROWID persists through recovery, backup and restore operations. It does not persist through replication, **ttMigrate** or **ttBulkCp** operations.

See "Expression specification" on page 72 for more information on ROWID.

ROWNUM specification

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which the row was selected. The first row selected has a ROWNUM of 1, the second a ROWNUM of 2, and so on.

Use ROWNUM to limit the number of rows returned by a query as in this example:

```
SELECT * FROM employees WHERE ROWNUM < 10;
```

The order in which rows are selected depends on the index used and/or the join order. If you specify an ORDER BY clause, ROWNUM is assigned before sorting. However, the presence of the ORDER BY clause may change the index used and/or the join order. If the order of selected rows changes, the ROWNUM value associated with each selected row could also change.

For example, the following query may return a different set of employees than the preceding query if a different index is used:

```
SELECT * FROM employees WHERE ROWNUM < 10 ORDER BY last_name;
```

Conditions testing for ROWNUM values greater than a positive integer are always false. For example, the following query returns no rows:

```
SELECT * FROM employees WHERE ROWNUM > 1;
```

Use ROWNUM to assign unique values to each row of a table. For example,

```
UPDATE my_table SET column1 = ROWNUM;
```

If your query contains either FIRST *NumRows* or ROWS M TO N, do not use ROWNUM to restrict the number of rows returned. For example, the following query results in an error message:

```
SELECT FIRST 2 * FROM employees WHERE ROWNUM <1 ORDER BY employee_id;

 2974: Using rownum to restrict number of rows returned cannot be
combined with first N or rows M to N
```

Expression specification

An *expression* specifies a *value* to be used in a SQL operation.

An expression can consist of a primary or several primaries connected by arithmetic operators, comparison operators, string or binary operators, bit operators or any of the functions described in this chapter. A primary is a signed or unsigned value derived from one of the items listed in the SQL syntax:

SQL syntax

{*ColumnName* | ROWID | {? | :*DynamicParameter*} |
AggregateFunction | *Constant* | (*Expression*)}

or

[[+ |-] {*ColumnName* | SYSDATE | TT_SYSDATE|GETDATE() |
{? | :*DynamicParameter*} | *AggregateFunction* |
Constant | {~ | + | -} *Expression*}]

[...]

or

Expression1 [& | | | ^ | + | / | * | -] *Expression2*

or

Expression1 | | *Expression2*

or

Expression

+, –	Unary plus and unary minus. Unary minus changes the sign of the primary. The default is to leave the sign unchanged.
ColumnName	Name of a column from which a value is to be taken. Column names are discussed in Chapter 2, “Names.
ROWID	TimesTen assigns a unique ID, called a ROWID to each row stored in a table. ROWID has type BINARY(16). The ROWID value can be retrieved through a pseudo column named ROWID.
? *:DynamicParameter*	A place holder for a dynamic parameter. The value of the dynamic parameter is supplied at runtime.
AggregateFunction	A computed value. See “Aggregate functions” on page 78.
Constant	A specific value. See “Constants” on page 82.
(*Expression*)	Any expression enclosed in parentheses.
Expression1 *Expression2*	*Expression1* and *Expression2*, when used with the bitwise operators, can be of integer or binary types. The types of both expressions must be compatible. See Chapter 1, “Data Types.

*	Multiplies two primaries.
/	Divides two primaries.
+	Adds two primaries.
–	Subtracts two primaries.
&	Bitwise AND of the two operands. Sets a bit to 1 if and only if both of the corresponding bits in *Expression1* and *Expression2* are 1, and to 0 if the bits differ or both are 0.
|	Bitwise OR of the two operands. Sets a bit to 1 if one or both of the corresponding bits in *Expression1* and *Expression2* are 1, and to 0 if both of the corresponding bits are 0.
~	Bitwise NOT of the operand. Takes only one *Expression* and inverts each bit in the operand, changing all the ones to zeros and zeros to ones.
^	Exclusive OR of the two operands. Sets the bit to 1 where the corresponding bits in its *Expression1* and *Expression2* are different, and to 0 if they are the same. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
||	Concatenates *Expression1* and *Expression2*, where both expressions are character strings. Forms a new string value that contains the values of both expressions. See also "CONCAT" on page 106.

Description

- Arithmetic operators can be used between numeric values. See "Numeric data types" on page 29
- Arithmetic operators can also be used between date-time values and interval types. The result of a date-time expression is either a date-time type or an interval type.
- Arithmetic operators cannot be applied to string values.
- Elements in an expression are evaluated in the following order:
  - Aggregate functions and expressions in parentheses.
  - Unary pluses and minuses.
  - The * and / operations.
  - The + and – operations.
  - Elements of equal precedence are evaluated in left-to-right order.
- You can enclose expressions in parentheses to control the order of their evaluation. For example:

  ```
  10 * 2 – 1 = 19 but 10 * (2 – 1) = 10
  ```

- Type conversion, truncation, underflow, or overflow can occur when some expressions are evaluated. See Chapter 1, "Data Types.
- If either operand in a numeric expression is NULL, the result is NULL.
- Since NVL takes two parameters, both designated as an "expression", TimesTen does not permit NULL in either position. If there is a NULL value in an expression, comparison operators and other predicates evaluate to NULL. See Chapter 4, "Search Conditions for more information on evaluation of comparison operators and predicates containing NULL values. We permit inserting of NULL, but in general INSERT takes only specific values, and not general expressions.
- The query optimizer and execution engine permit multiple ROWID lookups when a predicate specifies a disjunct of ROWID equalities or uses IN. For example, multiple fast ROWID lookups are executed for:

```
WHERE ROWID = :v1 OR ROWID = :v2
```

- or equivalently:

```
WHERE ROWID IN (:v1, :v2)
```

- The ? or :*DynamicParameter* can be used as a dynamic parameter in an expression, as shown in the following examples.

Example 3.1 In the WHERE clause of any SELECT statement:

```
SELECT *
FROM Purchasing.Orders
WHERE PartNumber = ?
AND OrderNumber > ?
ORDER BY OrderNumber
```

Example 3.2 In the WHERE and SET clauses of an UPDATE statement:

```
UPDATE Purchasing.Parts
SET SalesPrice = :DynamicParameter1
WHERE PartNumber = :DynamicParameter2
```

Example 3.3 In the WHERE clause of a DELETE statement:

```
DELETE FROM Purchasing.OrderItems
WHERE ItemNumber BETWEEN ? AND ?
```

Example 3.4 In the VALUES clause of an **INSERT** statement. In this example, both ? and :*DynamicParameter* are used where `:DynamicParameter1` corresponds to both the second and fourth columns of the `Purchasing.OrderItems` table. Therefore, only four distinct dynamic parameters need to be passed to this expression with the second parameter used for both the second and fourth columns.

```
INSERT INTO Purchasing.OrderItems VALUES
   (?,:DynamicParameter1,
```

```
        :DynamicParameter2,
        :DynamicParameter1,?)
```

This example demonstrates that both ? and :*DynamicParameter* can be used in the same SQL statement and shows the semantic difference between repeating both types of dynamic parameters.

Example 3.5 Examples of bitwise operators:

```
Command> SELECT 0x183D & 0x00FF from dual;
< 003D >
1 row found.
```

```
Command> SELECT ~255 FROM dual;
< -256 >
1 row found.
```

```
Command> SELECT 0x08 | 0x0F FROM dual;
< 0F >
1 row found.
```

Subqueries

TimesTen supports subqueries in **SELECT**, **CREATE VIEW**, **DELETE** or **UPDATE** statements or in an update SET clause, in a search condition and as a derived table. TimesTen supports table subqueries and scalar subqueries. It does not support row subqueries. A subquery can specify an aggregate with a HAVING clause or joined table. It can also be correlated.

Description

TimesTen supports queries that have the following characteristics.

Table subqueries:

- A subquery can appear in the WHERE clause or HAVING clause of any statement, except one that creates a MATERIALIZED VIEW. Only one table subquery can be specified in a predicate. These predicates can be specified in a WHERE or HAVING clause, an OR'd expression within a WHERE or HAVING clause, or an ON clause of a joined table. They cannot be specified in a CASE expression, a materialized view, or a HAVING clause that uses the + operator for outer joins.
- A subquery can be specified in an EXISTS or NOT EXISTS predicate, a quantified predicate with ANY or ALL, or a comparison predicate. The allowed operators for both comparison and quantified predicates are: =, <, >, <=, >=, <>. The subquery cannot be connected to the outer query through a UNIQUE or NOT UNIQUE operator.
- Only one subquery can be specified in a quantified or comparison predicate. Specify the subquery as either the right operand or the left operand of the predicate, but not both.
- The subquery should not have an ORDER BY clause.
- FIRST *NumRows* is not supported in subquery statements.
- A subquery cannot specify a UNION, MINUS or INTERSECT.
- In a query specified in a quantified or comparison predicate, the underlying SELECT must have a single expression in the select list. In a query specified in a comparison predicate, if the underlying select returns a single row, the return value is the select result. If the underlying select returns no row, the return value is NULL. It is an error if the subquery returns multiple rows.

Scalar subqueries (a scalar subquery returns a single value):

- A non-verifiable scalar subquery is one which has a predicate such that the optimizer cannot detect at compile time that the subquery returns at most one row for each row of the outer query. The subquery cannot be specified in an OR expression.
- Neither outer query nor any scalar subquery should have a DISTINCT modifier.

SQL Syntax

[NOT] EXISTS | [NOT] IN (*Subquery*)

Expression {= | <> | > | >= | < | <= } [ANY | ALL] *(Subquery)*

Expression [NOT] IN (*ValueList* | *Subquery*)

Examples

Examples of supported subqueries for a list of customers having at least one un-shipped order:

Example 3.6

```
SELECT customers.name FROM customers
WHERE EXISTS (SELECT 1 FROM orders
    WHERE customers.id = orders.custid
    AND orders.status = 'un-shipped');
```

Example 3.7

```
SELECT customers.name FROM customers
WHERE customers.id = ANY
(SELECT orders.custid FROM orders
WHERE orders.status = 'un-shipped');
```

Example 3.8

```
SELECT customers.name FROM customers
WHERE customers.id IN
(SELECT orders.custid FROM orders
WHERE orders.status = 'un-shipped');
```

Example 3.9

In this example, list items are shipped on the same date as when they are ordered:

```
SELECT line_items.id FROM line_items
WHERE line_items.ship_date =
(SELECT orders.order_date FROM orders
WHERE orders.id = line_items.order_id);
```

Aggregate functions

Aggregate functions specify a value computed with data from a set of rows described in an argument. The argument, enclosed in parentheses, is an expression.

Aggregate functions can be specified in the select list or the HAVING clause. See "INSERT SELECT" on page 292 for more information. The value of the expression is computed using each row that satisfies the WHERE clause.

SQL syntax

```
{AVG    ({Expression | [ALL | DISTINCT] ColumnName})
 MAX    ({Expression | [ALL | DISTINCT] ColumnName | ROWID})
 MIN    ({Expression | [ALL | DISTINCT] ColumnName | ROWID})
 SUM    ({Expression | [ALL | DISTINCT] ColumnName})
 COUNT ({ * | [ALL | DISTINCT] ColumnName | ROWID})
}
```

Expression	Specifies an argument for the aggregate function. The expression itself cannot be an aggregate function.
AVG	Computes the arithmetic mean of the values in the argument. NULL values are ignored. AVG can be applied only to numeric data types.
MAX	Finds the largest of the values in the argument (ASCII comparison for alphabetic types). NULL values are ignored. MAX can be applied to numeric, character, and BINARY data types.
MIN	Finds the smallest of the values in the argument (ASCII comparison for alphabetic types). NULL values are ignored. MIN can be applied to numeric, character, and BINARY data types.
SUM	Finds the total of all values in the argument. NULL values are ignored. SUM can be applied to numeric data types only.
COUNT *	Counts all rows that satisfy the WHERE clause, including rows containing NULL values. The data type of the result is TT_INTEGER. For more information on the number of rows in a table, see the description for the NUMTUPS field in **SYS.TABLES**.
COUNT *ColumnName*	Counts all rows in a specific column. Rows containing NULL values are not counted. The data type of the result is TT_INTEGER. For more information on the number of rows in a table, see the description for the NUMTUPS field in **SYS.TABLES**.

ALL	Includes any duplicate rows in the argument of an aggregate function. If neither ALL nor DISTINCT is specified, ALL is assumed.
DISTINCT	Eliminates duplicate column values from the argument of an aggregate function. Can be specified for more than one column.

Description

- If an aggregate function is computed over an empty table in which GROUP BY is not used, the results are as follows:
  - COUNT returns 0.
  - AVG, SUM, MAX, and MIN return NULL.
- If an aggregate function is computed over an empty group or an empty grouped table (GROUP BY is used):
  - COUNT returns nothing.
  - AVG, SUM, MAX, and MIN return nothing.
- See Chapter 1, "Data Types for information on:
  - Truncation and type conversion that may occur during the evaluation of aggregate functions.
  - Precision and scale of aggregate functions involving numeric arguments.
  - Control of the result type of an aggregate function.
- For SUM:
  - If the source is TT_TINYINT, TT_SMALLINT, or TT_INTEGER, the result data type is TT_INTEGER.
  - If the source is NUMBER, then the result data type is NUMBER with undefined scale and precision.
  - If the source is TT_DECIMAL, then the result data type is TT_DECIMAL with maximum precision.
  - For all other data types, the result data type is the same as the source.
  - See Example 3.12 on page 80.
- For MAX and MIN:
  - The result data type is the same as the source.
- For AVG:
  - AVG is evaluated as SUM/COUNT; the result data type is derived using the rule that is applied for the DIV operator.

Example 3.10 Calculate the average salary for employees in the HR schema; Use CAST to cast the average as the data type of the column:

```
Command> SELECT CAST(AVG (salary) AS NUMBER (8,2)) FROM employees;
< 6461.68 >
```

Example 3.11 Calculate the MAX salary for employees in the HR schema:

```
Command> SELECT MAX (salary) FROM employees;
< 24000 >
1 row found.
```

Example 3.12 The example uses DESCRIBE to show the data type that is returned when using the SUM aggregate. Table AGGREGATES is created and columns with different data types are defined:

```
Command> CREATE TABLE Aggregates (Col1 TT_TINYINT, Col2 TT_SMALLINT,
Col3 TT_INTEGER, Col4 TT_BIGINT, Col5 NUMBER (4,2),
Col6 TT_DECIMAL (6,2), Col7 BINARY_FLOAT, Col8 BINARY_DOUBLE);
Command> DESCRIBE SELECT SUM (Col1) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             TT_INTEGER
Command> DESCRIBE SELECT SUM (Col2) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             TT_INTEGER
Command> DESCRIBE SELECT SUM (Col3) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             TT_INTEGER
Command> DESCRIBE SELECT SUM (Col4) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             TT_BIGINT
Command> DESCRIBE SELECT SUM (Col5) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             NUMBER
Command> DESCRIBE SELECT SUM (Col6) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             TT_DECIMAL (40,2)
Command> DESCRIBE SELECT SUM (Col7) FROM Aggregates;

Prepared Statement:
  Columns:
    EXP                             BINARY_FLOAT
Command> DESCRIBE SELECT SUM (Col8) FROM Aggregates;
```

```
Prepared Statement:
  Columns:
    EXP                                   BINARY_DOUBLE
```

Constants

A constant is a literal value.

SQL syntax

{*IntegerValue* | *FloatValue* |*FloatingPointLiteral*| *FixedPointValue* |
‘*CharacterString*’| ‘National*CharacterString*’|*0xHexadecimalString* |
‘DateString’ | *DateLiteral* | *‘TimeString’* | *TimeLiteral* |
‘TimestampString’ | *TimestampLiteral* | *IntervalLiteral*
BINARY_FLOAT_INFINITY| BINARY_DOUBLE_INFINITY|
-BINARY_FLOAT_INFINITY| -BINARY_DOUBLE_INFINITY|
BINARY_FLOAT_NAN | BINARY_DOUBLE_NAN

}

IntegerValue	A whole number compatible with TT_INTEGER, TT_BIGINT or TT_SMALLINT data types or an unsigned whole number compatible with the TT_TINYINT data type. For example: `155, 5, -17`
FloatValue	A floating-point number compatible with the BINARY_FLOAT or BINARY_DOUBLE data types. Examples: `.2E-4, 1.23e -4, 27.03, -13.1`
FloatingPointLiteral	Floating point literals are compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. f or F indicates that the number is a 32-bit floating point number (of type BINARY_FLOAT). d or D indicates that the number is a 64-bit floating point number (of type BINARY_DOUBLE). For example: `123.23F, 0.5d`
FixedPointValue	A fixed-point number compatible with the BINARY_FLOAT, BINARY_DOUBLE or NUMBER data types. For example: `27.03`
CharacterString	A character string compatible with CHAR or VARCHAR2 data types. String constants are delimited by single quotation marks. For example: `'DON''T JUMP!'` Two single quotation marks in a row are interpreted as a single quotation mark, not as string delimiters or the empty string.

NationalCharacterString	A character string compatible with NCHAR or NVARCHAR2 data types. National string constants are preceded by an indicator consisting of either 'N' or 'n', and delimited by single quotation marks. For example:
	`N'Here''s how!'`
	Two single quotation marks in a row are interpreted as a single quotation mark.
	The contents of a national string constant may consist of any combination of:
	• ASCII characters
	• UTF-8 encoded Unicode characters
	• Escaped Unicode characters
	ASCII characters and UTF-8 encoded characters are converted internally to their corresponding UTF-16 format Unicode equivalents.
	Escaped Unicode characters are of the form `\uxxxx`, where `xxxx` is the four hex-digit representation of the Unicode character. For example:
	`N'This is an \u0061'`
	is equivalent to:
	`N'This is an a'`
	The `\u` itself can be escaped with another `\`. The sequence `\\u` is always converted to `\u`. No other escapes are recognized.

HexadecimalString	A string of hexadecimal digits 0 - 9 and A - F (or a - f) compatible with the BINARY, VARBINARY, CHAR and VARCHAR2 data types. A *HexadecimalString* constant must be prefixed with the characters “0x.” For example: `0xFFFAB0880088343330FFAA7` or `0x000A001231`
DateString	A string of the format `YYYY-MM-DD HH:MI:SS` enclosed in single quotation marks ('). For example: '2007-01-27 12:00:00'. The `YYYY` field must have a 4-digit value. The `MM` and `DD` fields must have 2-digit values. The only spaces allowed are trailing spaces (after the day field). The range is from '-4713-01-01' (January 1, 4712 BC) to '9999-12-31', (December 31, 9999). The time component is not required. For example: `‘2007-01-27’.` For TT_DATE data types, the string is of format `YYYY-MM-DD` and ranges from ‘1753-01-01’ to ‘9999-12-31’. If you are using TimesTen type mode, for information on *DateString*, refer to documentation from previous releases of TimesTen.

DateLiteral	Format: DATE *DateString*. For example:
	`DATE '2007-01-27' or DATE '2007-01-27 12:00:00'`
	For TT_DATE data types, use the literal TT_DATE. For example:
	`TT_DATE '2007-01-27'.` Do not specify a time portion with the TT_DATE literal.
	The DATE keyword is case-insensitive.
	TimesTen also supports ODBC-date-literal syntax.
	For example:
	`{d '2007-01-27'}.`
	Please refer to the *Microsoft ODBC Programmer's Reference and SDK Guide* included with your release of TimesTen.
	
	If you are using TimesTen type mode, for information on *DateLiteral*, refer to documentation from previous releases of TimesTen.

TimeString	A string of the format HH:MM:SS enclosed in single quotation marks ('). For example: `'20:25:30'` The range is '00:00:00' to '23:59:59', inclusive. Every component must be two digits. The only spaces allowed are trailing spaces (after the seconds field).

TimeLiteral	Format: TIME *TimeString*. For example:
	`TIME '20:25:30'`
	The TIME keyword is case-insensitive.
	Usage examples:
	`INSERT INTO timetable VALUES (TIME '10:00:00');`
	`SELECT * FROM timetable WHERE col1 < TIME '10:00:00';`
	TimesTen also supports ODBC-time-literal syntax.
	For example:
	`{t '12:00:00'}.`
	Please refer to the *Microsoft ODBC Programmer's Reference and SDK Guide* included with your release of TimesTen.

TimestampString	A string of the format `YYYY-MM-DD HH:MM:SS[.FFFFFFFFF]` enclosed in single quotation marks('). The range is from `'-4713-01-01' (January 1, 4712 BC) to '9999-12-31' (December 31, 9999).` The year field must be a 4-digit value. All other fields except for the fractional part must be 2-digit values. The fractional field can consist of 0 to 9 digits. For TT_TIMESTAMP data types, a string of format `YYYY-MM-DD HH:MM:SS[.FFFFFF]` enclosed in single quotation marks('). The range is from `'1753-01-01 00:00:00.000000'` to `'9999-12-31 23:59:59.999999'`. The fractional field can consist of 0 to 6 digits. If you have a CHAR column called `C1`, and want to enforce the TIME comparison, you can do the following: `SELECT * FROM testable WHERE C1 = TIME '12:00:00'` In this example, each CHAR value from `C1` is converted into a TIME value before comparison, provided that values in `C1` conform to the proper TIME syntax. If you are using TimesTen type mode, for information on *TimestampString*, refer to documentation from previous releases of TimesTen.
TimestampLiteral	Format: TIMESTAMP *TimestampString* For example: TIMESTAMP '2007-01-27 11:00:00.000000' For TIMESTAMP data types, the fraction field supports from 0 to 9 digits of fractional seconds; for TT_TIMESTAMP data types, the fraction field supports from 0 to 6 digits of fractional seconds. The TIMESTAMP keyword is case-insensitive. Literal syntax can be used if you want to enforce DATE/TIME/TIMESTAMP comparison for CHAR/VARCHAR data type. TimesTen also supports ODBC-timestamp-literal syntax. For example: `{ts '9999-12-31 12:00:00'}` Please refer to the *Microsoft ODBC Programmer's Reference and SDK Guide* included with your release of TimesTen. If you are using TimesTen type mode, for information on *TimestampLiteral*, refer to documentation from previous releases of TimesTen.
IntervalLiteral	Format: INTERVAL [+\-] *CharacterString IntervalQualifier.* For example INTERVAL '8' DAY

BINARY_ FLOAT_INFINITY| BINARY_ DOUBLE_INFINITY	INF (positive infinity) is an IEEE floating-point value that is compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. Use the constant values BINARY_FLOAT_INFINITY or BINARY_DOUBLE_INFINITY to represent positive infinity.
-BINARY_ FLOAT_INFINITY| -BINARY_DOUBLE_ INFINITY	-INF (negative infinity) is an IEEE floating-point value that is compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. Use the constant values -BINARY_FLOAT_INFINITY and -BINARY_DOUBLE_INFINITY to represent negative infinity.
BINARY_FLOAT_NAN| BINARY_DOUBLE_NAN	NaN ("not a number") is an IEEE floating-point value that is compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. Use the constant values BINARY_FLOAT_NAN or BINARY_DOUBLE_NAN to represent NaN ("not a number").

Format Models

A format model is a character literal that describes the format of datetime and numeric data stored in a character string. When you convert a character string into a date or number, a format model determines how TimesTen interprets the string.

Number format models

Use number format models in the following functions:

- In the TO_CHAR function to translate a value of NUMBER, BINARY_FLOAT, or BINARY_DOUBLE data type to VARCHAR2 data type.
- In the TO_NUMBER function to translate a value of CHAR or VARCHAR2 data type to NUMBER data type.

Number format elements

A number format model is composed of one or more number format elements. The table lists the elements of a number format model. Negative return values automatically contain a leading negative sign and positive values automatically contain a leading space unless the format model contains the *MI*, *S*, or *PR* format element.

The default American_america NLS setting is used.

Number format elements:

Element	Example	Description
, (comma)	9,999	Returns a comma in the specified position. You can specify multiple commas in a number format model. **Restrictions:** • A comma element cannot begin a number format model. • A comma cannot appear to the right of the decimal character or period in a number format model.
. (period)	99.99	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** • You can specify only one period in a format model.
$	$9999	Returns value with leading dollar sign.
0	0999 9990	Returns leading zeros. Returns trailing zeros.

Element	Example	Description
9	9999	Returns value with the specified number of digits with a leading space if positive or with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
B	B9999	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of zeros in the format model).
C	C999	Returns in the specified position the ISO currency symbol (the current value of the *NLS_ISO_CURRENCY* parameter).
D	99D99	Returns in the specified position the decimal character, which is the current value of the *NLS_NUMERIC_CHARACTER* parameter. The default is a period (.). **Restriction:** • You can specify only one decimal character in a number format model.
EEEE	9.9EEEE	Returns a value in scientific notation.
G	9G999	Returns in the specified position the group separator (the current value of the *NLS_NUMERIC_CHARACTER* parameter). You can specify multiple group separators in a number format model. **Restriction:** • A group separator cannot appear to the right of a decimal character or period in a number format model.
L	L999	Returns in the specified position the local currency symbol (the current value of the *NLS_CURRENCY* parameter).

Element	Example	Description
MI	999MI	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing blank. **Restriction:** • The MI format element can appear only in the last position of a number format model.
PR	999PR	Returns negative value in <angle brackets>. Returns positive value with a leading and trailing blank. **Restriction:** • The PR format element can appear only in the last position of a number format model.
RN	RN	Returns a value as Roman numerals in uppercase.
rn	rn	Returns a value as Roman numerals in lowercase. Value can be an integer between 1 and 3999.
S	S9999	Returns negative value with a leading minus sign (-). Returns positive value with a leading plus sign (+).
	9999S	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing plus sign (+). **Restriction:** • The S format element can appear only in the first or last position of a number format model.

Element	Example	Description
TM	TM	The text minimum number format model returns (in decimal output) the smallest number of characters possible. This element is case insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If the output exceeds 64 characters, then TimesTen automatically returns the number in scientific notation. **Restrictions:** • You cannot precede this element with any other element. • You can follow this element only with one 9 or one E or (e), but not with any combination of these. The following statement returns an error: • *SELECT TO_NUMBER (1234, 'TM9e') from DUAL;*
U	U9999	Returns in the specified position the euro (or other) dual currency symbol (the current value of the *NLS_DUAL_CURRENCY* parameter).

Element	Example	Description
V	999V99	Returns a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the V.
X	XXXX	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, then TimesTen rounds it to an integer. **Restrictions:** • This element accepts only positive values or 0. Negative values return an error. • You can precede this element only with 0 (which returns leading zeros) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, then the return always has a leading blank.

Datetime format models

Use datetime format models in the following functions:

- In the TO_CHAR or TO_DATE functions to translate a character value that is in a format other than the default format for a datetime value.
- In the TO_CHAR function to translate a datetime value that is in a format other than the default format into a string.

The total length of a datetime format model cannot exceed 22 characters.

The default American_america NLS setting is used.

Datetime format elements

A datetime format model is composed of one or more datetime format elements.

Datetime format elements

Element	Description
- / , . ; : “text”	Punctuation and quoted text is reproduced in the result.
AD A.D.	AD indicator with or without periods.
AM A.M.	Meridian indicator with or without periods.
BC B.C.	BC indicator with or without periods.
D	Day of week (1-7).
DAY	Name of day, padded with blanks to display width of widest name of day.
DD	Day of month (1-31).
DDD	Day of year.

Element	Description
DL	Returns a value in the long date format. In the default AMERICAN_AMERICA locale, this is equivalent to specifying the format ‘fmDay, Month dd, yyyy’. **Restriction:** Specify this format only with the *TS* element, separated by white space.
DS	Returns a value in the short date format. In the default AMERICAN_AMERICA locale, this is equivalent to specifying the format ‘MM/DD/RRRR’. **Restriction:** Specify this format only with the *TS* element, separated by white space.
DY	Abbreviated name of day.
FM	Returns a value with no leading or trailing blanks.
FX	Requires exact matching between the character data and the format model.
HH	Hour of day (1-12).
HH24	Hour of day (0-23).
J	Julian day; the number of days since January 1, 4712 BC. Number specified with J must be integers.
MI	Minute (0-59).
MM	Month (01-12; January = 01).
MON	Abbreviated name of month.
MONTH	Name of month padded with blanks to display width of the widest name of month.
RM	Roman numeral month (I-XII; January = I).
RR	Stores 20th century dates in the 21st century using only two digits.
RRRR	Rounds year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you do not want this functionality, then enter the 4-digit year.
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).

Element	Description
TS	Returns a value in the short time format. **Restriction:** Specify this format only with the *DL* or *DS* element, separated by white space.
X	Local radix character. Example: ‘HH:MI:SSXFF’.
Y,YYY	Year with comma in this position.
YYYY SYYYY	4-digit year; *S* prefixes BC dates with a minus sign.
YYY YY Y	Last 3, 2, or 1 digit (s) of year.

Format model for TO_CHAR of TimesTen types

Use this format model when invoking the TO_CHAR function to convert a datetime value of TT_TIMESTAMP or TT_DATE. In addition, use this format model when invoking the TO_CHAR function to convert any numeric value other than NUMBER or ORA_FLOAT.

- If a numeric value doesn't fit in the specified format, TimesTen truncates the value.
- The format string cannot exceed 50 characters.
- D always results in a decimal point. Its value cannot be changed with an NLS parameter.
- If a float with an absolute value less than `1e-126` or greater than `1e126` is specified as input to the TO_CHAR function, TimesTen returns an error.

Format	Description
DD	Day of month (1-31)
MM	Month (1-12)
MON	Month (three character prefix)
MONTH	Month (full name blank-padded to 9 characters)
YYYY	Year (four digits)
Y,YYY	Year (with comma as shown)
YYY	Year (last three digits)
YY	Year (last two digits)
Y	Year (last digit)
Q	Quarter
HH	Hour (1-12)
HH12	Hour (1-12)
HH24	Hour (0-23)
MI	Minute (0-59)
SS	Second (0-59)
FF	Fractions of a second to a precision of 6 digits
FF*n*	Fractions of a second to the precision specified by *n*

Format	Description
AM	Meridian indicator
A.M.	Meridian indicator
PM	Meridian indicator
P.M.	Meridian indicator
- / , . ; :	Punctuation to be output
"text"	Text to be output
9	Digit
0	Leading/trailing zero
.	Decimal point
,	Comma
EEEE	Scientific notation
S	Sign mode
B	Blank mode. If there are no digits, the string is filled with blanks.
FM	No-blank mode (Fill mode). If this element is used, trailing and/or leading spaces are suppressed.
$	Leading dollar sign

ASCIISTR

The ASCIISTR takes as its argument, either a string or an expression that resolves to a string, in any character set, and returns the ASCII version of the string in the database character set. Non-ASCII characters are converted to Unicode escapes.

SQL Syntax

ASCIISTR (*[N]'String'*)

Parameters

ASCIISTR has the parameter:

Parameter	Description
[N]'String'	The string passed to the ASCIISTR function. The string can be in any character set. The ASCII version of the string in the database character set is returned. Specify N if you wish to pass the string in UTF-16 format.

Description

- The ASCIISTR function allows you to see the representation of a string value that is not in your database character set.

Example 3.13

The following example invokes the ASCIISTR function passing as an argument the string `'Aäa'` in UTF-16 format; The ASCII version is returned in the WE8ISO8859P1 character set; The non-ASCII character ä is converted to Unicode encoding value:

```
Command> connect "dsn=test; ConnectionCharacterSet= WE8ISO8859P1";
Connection successful: DSN=test;UID=user1;DataStore=/datastore/
user1/test;
DatabaseCharacterSet=WE8ISO8859P1;
ConnectionCharacterSet=WE8ISO8859P1;PermSize=32;TypeMode=0;
(Default setting AutoCommit=1)
Command> SELECT ASCIISTR (n'Aäa') FROM DUAL;
< A\00E4a >
1 row found.
```

CASE

Specifies a conditional value. Both simple and searched case expressions are supported. Case expression can be specified anywhere an expression can be and can be used as often as needed.

Instead of using a series of if statements, case expression allows you to use a series of conditions that return the appropriate values when the conditions are met. With CASE expression, you can simplify queries and write more efficient code.

SQL Syntax

The syntax for a searched CASE expression is:

CASE

{WHEN *SearchCondition* THEN *Expression1*}[…]

[ELSE *Expression2*]

END

The syntax for a simple CASE expression is:

CASE *Expression*

{WHEN *CompExpression* THEN *Expression1*}[…]

[ELSE *Expression2*]

END

Parameters

CASE has the parameters:

Parameter	Description
WHEN *SearchCondition*	Specifies the search criteria. This clause cannot specify a subquery.
WHEN *CompExpression*	Specifies the operand to be compared.
Expression	Specifies the first operand to be compared with each *CompExpression*.
THEN *Expression1*	Specifies the resulting expression.
ELSE *Expression2*	If condition is not met, specifies the resulting expression. If no ELSE clause is specified, TimesTen adds and ELSE NULL clause to the expression.

Description

- CASE expression can not be specified in the value clause of an INSERT statement.

Examples

Example 3.14 To specify a searched CASE statement that specifies the value of a color, use:

```
SELECT CASE WHEN color=1 THEN 'red' WHEN color=2 THEN 'blue' ELSE
'yellow' END FROM cars.
```

Example 3.15 To specify a simple CASE statement that specifies the value of a color, use:

```
SELECT CASE color WHEN 1 THEN 'red' WHEN 2 THEN 'blue' ELSE
'yellow' END FROM cars.
```

CAST

Allows you to convert data of one type to another type. CAST can be used wherever a constant can be used. CAST is useful in specifying the exact data type for an argument. This is especially true for unary operators like '-' or functions with one operand like TO_CHAR or TO_DATE.

A value can only be CAST to a compatible data type, with the exception of NULL. NULL can be cast to any other data type. CAST is not needed to convert a NULL to the desired target type in an insert select.

The following conversions are supported:

- Numeric value to numeric or BCD (Binary Coded Decimal)
- NCHAR to NCHAR
- CHAR string to BINARY string or DATE, TIME or TIMESTAMP
- BINARY string to BINARY or CHAR string
- DATE, TIME or TIMESTAMP to CHAR

SQL Syntax

CAST
({*Expression* | NULL} AS *DataType*)

Parameters

CAST has the parameters:

Parameter	Description
Expression	Specifies the value to be converted.
AS *DataType*	Specifies the resulting data type.

Description

- CAST to a domain name is not supported.
- Casting a selected value may cause the SELECT statement to take more time and memory than a SELECT statement without a CAST expression.

Examples

```
INSERT INTO t1 values(TO_CHAR(CAST(? AS REAL)));
SELECT CONCAT(x1, CAST (? AS CHAR(10))) FROM t1;
SELECT * FROM t1 WHERE CAST (? AS INT)=CAST(? AS INT);
```

CHR

The CHR function returns the character having the specified binary value in the database character set.

SQL Syntax

CHR (*n)*

Parameters

CHR has the parameter:

Parameter	Description
n	The binary value in the database character set. The character having this binary value is returned. The result is of type VARCHAR2.

Description

- For single-byte character sets, if *n* >256, then TimesTen returns the binary value of *n* mod 256.
- For multibyte character sets, n must resolve to one code point. Invalid code points are not validated. If you specify an invalid code point, the result is indeterminate.

Note: When you use the CHR function, the code is not portable between ASCII- and EBCDIC- based machine architectures.

Example 3.16

The following example is run on an ASCII-based machine with the WE8ISO8859P1 character set.

```
Command> SELECT CHR(67)||CHR(65)||CHR(84) FROM DUAL;
< CAT >
1 row found.
```

On an EBCDIC-based machine with the character set WE8EBCDIC1047, the preceding example would have to be modified to the following:

```
Command> SELECT CHR(195)||CHR(193)||CHR(227) FROM DUAL;
< CAT >
1 row found.
```

USER functions

TimesTen supports the USER functions:

- USER
- CURRENT_USER
- SESSION_USER
- SYSTEM_USER

Each of these functions returns the name of the user that is currently connected to the TimesTen instance.

CURRENT_USER
USER

Returns the value of the user currently connected to the data store.

SQL Syntax

CURRENT_USER

or

USER

Parameters

USER and CURRENT_USER have no parameters.

Example 3.17

To return the name of the user who is currently connected to the data store:

```
SELECT CURRENT_USER FROM SYS.DUAL;
```

SESSION_USER

Returns the value of the user currently connected to the data store.

SQL Syntax

SESSION_USER

Parameters

SESSION_USER has no parameters.

Example 3.18

To return the name of the session user:

```
SELECT SESSION_USER FROM SYS.DUAL;
```

SYSTEM_USER

Returns the value of the current data store user name as identified by the operating system, which may or may not match the value of USER.

SQL Syntax

SYSTEM_USER

Parameters

SYSTEM_USER has no parameters.

Example 3.19

To return the name of the system user:

```
SELECT SYSTEM_USER FROM SYS.DUAL;
```

COALESCE

The COALESCE function returns the first non-null *expression* in the expression list. If all occurrences of *expression* evaluate to NULL, then the function returns NULL.

SQL Syntax

COALESCE(*Expression1*, *Expression2* [,...])

Parameters

COALESCE has the parameters:

Parameter	Description
Expression1, *Expression2* *[,...]*	The expressions in the expression list. The first non-null *expression* in the expression list is returned. Each *expression* is evaluated in order and there must be at least 2 expressions.

Description

- This function is a generalization of the **NVL** function.
- Use COALESCE as a variation of the **CASE** expression. For example,

```
COALESCE (Expression1, Expression2)
```

is equivalent to:

```
CASE WHEN Expression1 IS NOT NULL THEN Expression1
     ELSE Expression2
END
```

Example 3.20

The example illustrates the use of the COALESCE expression. The COALESCE expression is used to return the commission_pct for the first 10 employees with manager_id = 100. If the commission_pct is NOT NULL, then the original value for commission_pct is returned. If commission_pct is NULL, then 0 is returned.

```
Command> SELECT FIRST 10 employee_id, COALESCE (commission_pct, 0)
FROM employees WHERE manager_id = 100;
< 101, 0 >
< 102, 0 >
< 114, 0 >
< 120, 0 >
< 121, 0 >
< 122, 0 >
< 123, 0 >
< 124, 0 >
< 145, .4 >
< 146, .3 >
10 rows found.
```

CONCAT

The CONCAT function concatenates one character string with another to form a new character string.

SQL Syntax CONCAT(*Expression1*, *Expression2*)

Expression1	A CHAR, VARCHAR2, NCHAR or NVARCHAR2 expression.
Expression2	A CHAR, VARCHAR2, NCHAR or NVARCHAR2 expression.

Description

- CONCAT returns *Expression1* concatenated with *Expression2*.
- The type of *Expression1* and *Expression2* must be compatible.
- If *Expression2* is NULL, CONCAT returns *Expression1*. If *Expression1* is NULL, CONCAT returns *Expression2*.
- If both *Expression1* and *Expression2* are NULL, CONCAT returns NULL.
- The return type of CONCAT depends on the types of *Expression1* and *Expression2*. The following table summarizes how CONCAT's type is determined:

Expression1	Expression2	CONCAT
CHAR(*m*)	CHAR(*n*)	CHAR(*m*+*n*)
CHAR(*m*)	VARCHAR2(*n*)	VARCHAR2(*m*+*n*)
VARCHAR2(*m*)	CHAR(*n*)	VARCHAR2(*m*+*n*)
VARCHAR2(*m*)	VARCHAR2(*n*)	VARCHAR2(*m*+*n*)

- The treatment of NCHAR and NVARCHAR2 is similar. If one of the operands is of varying length, then the result is of varying length. Otherwise the result is of a fixed length.
- The concatenation of CHAR, NCHAR, VARCHAR2, and NVARCHAR2 types are supported. The result type of character types concatenated with ncharacter types is ncharacter types.

Example 3.21 The following example concatenates first names and last names.

```
CONCAT(CONCAT(FNAME, ' '), LNAME);

SELECT CONCAT(CONCAT(FNAME, ' '), LNAME), SAL
FROM EMPLOYEE;
```

Example 3.22 The following example concatenates column id with column id2. In this example, the result type is nchar (40).

```
Command> create table cat (id char (20), id2 nchar (20));
Command> insert into cat values ('abc', 'def');
1 row inserted.
Command> select concat (id,id2) from cat;
< abc                 def                  >
1 row found.
```

See Also The description of the | | operator in the section "Expression specification" on page 72.

DECODE

The DECODE function compares an expression to each search value one by one. If the expression is equal to the search value, then the result value is returned. If no match is found, then the default value (if specified) is returned; otherwise NULL is returned.

SQL Syntax

DECODE(*Expression, {SearchValue, Result* [,...])} [,*Default*])

Parameters

DECODE has the parameters:

Parameter	Description
Expression	The *expression* that is compared to the search value.
SearchValue	An *expression* is compared to one or more search values.
Result	If the *expression* is equal to a *SearchValue*, then the specified *Result* value is returned.
Default	If no match is found, the *Default* value is returned. *Default* is optional; If *Default* is not specified and no match is found, then NULL is returned.

Description

- If an expression is NULL, then the NULL expression equals a NULL search value.

Example 3.23

The following example invokes the DECODE function. In the LOCATIONS table, if the column, Country_id is equal to ‘IT’, then the function returns ‘Italy’; if the Country_id is equal to ‘JP’, then ‘Japan’ is returned; If the Country_id is equal to ‘US’, then ‘United States’ is returned. If the Country_id is not equal to ‘IT’ or ‘JP’ or ‘US’, then ‘Other’ is returned.

```
Command> SELECT Location_id,
>        DECODE (Country_id, 'IT', 'Italy',
>                            'JP', 'Japan',
>                            'US', 'United States',
>                                  'Other')
> FROM LOCATIONS WHERE Location_id < 2000;
LOCATION_ID, EXP
< 1000, Italy >
< 1100, Italy >
< 1200, Japan >
< 1300, Japan >
< 1400, United States >
< 1500, United States >
< 1600, United States >
```

```
< 1700, United States >
< 1800, Other >
< 1900, Other >
10 rows found.
```

EXTRACT

The EXTRACT function extracts and returns the value of a specified DateTime field from a DateTime or interval value expression as a NUMBER data type. This function can be useful for manipulating DateTime field values in very large tables.

If you are using TimesTen type mode, for information on the EXTRACT function, refer to documentation from previous releases of TimesTen.

SQL Syntax

EXTRACT (*DateTimeField* FROM *IntervalExpression* | *DateTimeExpression*)

DateTimeField	The field to be extracted from IntervalExpression or DateTimeExpression. Accepted fields are YEAR, MONTH, DAY, HOUR, MINUTE or SECOND.
IntervalExpression	An interval result.
DateTimeExpression	A datetime expression. For example, TIME, DATE, TIMESTAMP.

Description

- Some combinations of DateTime field and DateTime or interval value expression result in ambiguity. In these cases, TimesTen returns UNKNOWN.
- The field you are extracting must be a field of the *IntervalExpression* or *DateTimeExpression*. For example, you can extract only YEAR, MONTH, and DAY from a DATE value. Likewise, you can extract HOUR, MINUTE or SECOND only from the TIME, DATE, or TIMESTAMP data type.
- The fields are extracted into a NUMBER value.

Example 3.24 The following example extracts the second field out of the interval result `sysdate-t1.createTime`

```
SELECT EXTRACT(SECOND FROM sysdate-t1.createTime) FROM t1;
```

Example 3.25 The following example extracts the second field out of sysdate from the system table DUAL.

```
Command> select extract (second from sysdate) from dual;
< 20 >
1 row found.
```

LOWER and UPPER

The LOWER function converts expressions of type CHAR, NCHAR, VARCHAR2 or NVARCHAR2 to lowercase. The UPPER function converts expressions of type CHAR, NCHAR, VARCHAR2 or NVARCHAR2 to uppercase. Character semantics is supported for CHAR and VARCHAR2 types. The data type of the result is the same as the data type of the expression.

SQL Syntax

{UPPER | LOWER} (*Expression1*)

Expression1	An expression which is converted to lowercase (using LOWER) or uppercase (using UPPER).

Description

- LOWER(?) and UPPER(?) are not supported but you can combine it with the CAST operator, for example:
  `LOWER(CAST(? AS CHAR(30))) .`

Example 3.26

```
Command> SELECT LOWER (last_name) FROM EMPLOYEES
WHERE employee_id = 100;
< king >
1 row found.
```

MOD

Returns the remainder of an INTEGER expression divided by a second INTEGER expression.

SQL Syntax

MOD(*Expression1*, *Expression2*)

Expression1	An INTEGER expression.
Expression2	An INTEGER expression.

Description

- MOD returns the remainder of *Expression1* divided by *Expression2*.
- If *Expression2* is 0, then MOD returns *Expression1*.
- If either *Expression1* or *Expression2* is NULL, MOD returns NULL.
- MOD is treated as a binary arithmetic operation, so the return type is determined according to the rules specified in Chapter 1, "Data Types.
- The MOD function behaves differently from the classical mathematical modulus function, when one of the operands is negative. The following table illustrates this difference:

M	N	Classical Modulus	MOD(M,N)
11	3	2	2
11	-3	-1	2
-11	3	1	-2
-11	-3	-2	-2

Example 3.27 The following example tests if the value of the expression M is divisible by the value of expression N.

```
SELECT M, N
FROM TEST
WHERE MOD(M, N) = 0;
```

NCHR

The NCHR function returns the character having the specified Unicode value.

SQL Syntax

NCHR (*n)*

Parameters

NCHR has the parameter:

Parameter	Description
n	The specified Unicode value. The character having this Unicode value is returned. The result is of type NVARCHAR2.

Example 3.28

The following example returns the nchar character 187:

```
Command> SELECT NCHR(187) FROM DUAL;
< > >
1 row found.
```

NLSSORT

Returns the sort key value for the given string.

SQL Syntax

NLSSORT (*String* [,'NLS_SORT = *SortName*'])

Parameters

NLSSORT has the following parameters:

Parameter	Description
String	Supported data types for *String* are CHAR, VARCHAR2, NCHAR and NVARCHAR2. Given the *String*, NLSSORT returns the sort key value used to sort the *String*.
['NLS_SORT = *SortName*']	*SortName* is either the linguistic sort sequence or BINARY. If you omit this parameter, then the default sort sequence for your session is used. Append to the *SortName* the suffix -ai for accent-insensitive sorting or -ci for case-insensitive sorting. For more information on acceptable linguistic *SortName* values, see "Supported Linguistic Sorts" in the Operations Guide.

Description

- The returned sort key value is of type VARBINARY.
- You can create a linguistic index for linguistic comparisons. See Example 3.30 on page 115.

Example 3.29

The following example illustrates sorting and comparison operations based on a linguistic sort sequence rather than on the binary value of the string. In addition, the example shows the same results can be obtained by using the ALTER SESSION... SET NLS_SORT statement.

```
Command> CREATE TABLE NsortDemo (Name VARCHAR2 (15));
Command> INSERT INTO NsortDemo VALUES ('Gaardiner');
1 row inserted.
Command> INSERT INTO NsortDemo VALUES ('Gaberd');
1 row inserted.
Command> INSERT INTO NsortDemo VALUES ('Gaasten');
1 row inserted.
Command> # Perform Sort
Command> SELECT * FROM NsortDemo ORDER BY Name;
< Gaardiner >
< Gaasten >
< Gaberd >
3 rows found.
```

```
Command> #Use function to perform Sort
Command> SELECT * FROM NsortDemo ORDER BY NLSSORT (Name,
'NLS_SORT = XDanish');
< Gaberd >
< Gaardiner >
< Gaasten >
3 rows found.

Command>#  comparison operation
Command> SELECT * FROM NsortDemo where Name > 'Gaberd';
0 rows found.

Command> #Use function in comparison operation
Command> SELECT * FROM NsortDemo WHERE NLSSORT (Name, 'NLS_SORT =
XDanish') >
> NLSSORT ('Gaberd', 'NLS_SORT = XDanish');
< Gaardiner >
< Gaasten >
2 rows found.

Command> #Use ALTER SESSION to obtain the same results
Command> ALTER SESSION SET NLS_SORT = 'XDanish';
Command> SELECT * FROM NsortDemo ORDER BY Name;
< Gaberd >
< Gaardiner >
< Gaasten >
3 rows found.
Command> SELECT * FROM NsortDemo where Name > 'Gaberd';
< Gaardiner >
< Gaasten >
2 rows found.
```

Example 3.30 The following example creates a linguistic index:

```
Command> CREATE INDEX DanishIndex ON NsortDemo (NLSSORT (Name,
'NLS_SORT = XDanish'));
Command> indexes N%;

Indexes on table USER1.NSORTDEMO:
  DANISHINDEX: non-unique T-tree index on columns:
    NLSSORT(NAME,'NLS_SORT = XDanish')
  1 index found.

1 table found.
```

NUMTODSINTERVAL

Converts a number or expression to an INTERVAL DAY TO SECOND type.

SQL Syntax

NUMTODSINTERVAL (*Expression1*, *IntervalUnit*)

Parameters

NUMTODSINTERVAL has the parameters:

Parameter	Description
Expression1	The argument can be any NUMBER value or an expression that can be implicitly converted to a NUMBER value.
IntervalUnit	One of the string constants: 'DAY', 'HOUR', 'MINUTE', or 'SECOND'.

Example 3.31 Example using NUMTODSINTERVAL with SYSDATE:

```
Command> SELECT SYSDATE + NUMTODSINTERVAL(20,'SECOND') FROM dual;
< 2007-01-28 09:11:06 >
```

NUMTOYMINTERVAL

Converts a number or expression to an INTERVAL YEAR TO MONTH type.

SQL Syntax

NUMTOYMINTERVAL (*Expression1*, '*IntervalUnit*')

Parameters

NUMTOYMINTERVAL has the parameters:

Parameter	Description
Expression1	The argument can be any NUMBER value or an expression that can be implicitly converted to a NUMBER value.
IntervalUnit	One of the string constants 'YEAR' or 'MONTH'.

Example 3.32

An example using NUMTOYMINTERVAL:

```
Command>  SELECT SYSDATE + NUMTOYMINTERVAL(1,'MONTH') FROM dual;
< 2007-02-28 09:23:28 >
1 row found.
```

NVL

The NVL function replaces a null value with a second value.

SQL Syntax

NVL(*Expression1*, *Expression2*)

Parameters

NVL has the parameters:

Parameter	Description
Expression1	The expression whose values are to be tested for NULL.
Expression2	The alternate value to use if the value of `Expression1` is NULL.

Description

- The type of *Expression1* and *Expression2* must be compatible.
- If Expression1 is NULL, the NVL function returns *Expression2*. If Expression1 is NOT NULL, the NVL function returns *Expression1*.
- The NVL function can be used in the WHERE or HAVING clause of SELECT, UPDATE, or DELETE statements and in the SELECT list of a SELECT statement.

Example 3.33

The following example tests whether values of the expression MIDDLENAME is NULL. For each expression, the string `'No Middle Name'` is returned if the expression value is NULL. Otherwise, the original expression value is returned.

```
NVL(MIDDLENAME, 'No Middle Name')
```

Example 3.34

```
SELECT FIRSTNAME,
       NVL(MIDDLENAME, 'No Middle Name'),
       LASTNAME
FROM EMPLOYEES;
```

RTRIM

The RTRIM function removes trailing spaces from columns or operands of expressions; Data type of expression is of type CHAR, VARCHAR2, NCHAR or NVARCHAR2.

SQL Syntax

RTRIM (*Expression*)

Parameters

RTRIM has the parameters:

Parameter	Description
Expression	The CHAR, VARCHAR2, NCHAR or NVARCHAR2 operand or column to be trimmed.

Description

- For expressions of type CHAR or VARCHAR2, the data type returned is VARCHAR2; For expressions of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2.

Example 3.35 The following example trims the trailing spaces from col1 in table RtrimTest.

```
Command> CREATE TABLE RtrimTest (col1 VARCHAR2 (25));
Command> INSERT INTO RtrimTest VALUES ('abc     ');
1 row inserted.
Command> SELECT * FROM rtrimtest;
< abc      >
1 row found.
Command> SELECT RTRIM (col1) FROM RtrimTest;
< abc >
1 row found.
```

SYSDATE and GETDATE

Returns the date in the format YYYY-MM-DD HH:MM:SS. The date represents the local current date and time, which is determined by the system on which the statement is executed.

If you are using TimesTen type mode, for information on SYSDATE, refer to documentation from previous releases of TimesTen.

SQL Syntax

SYSDATE | GETDATE( )

Parameters

The SYSDATE and GETDATE( ) functions have no parameters.

Description

- SYSDATE and GETDATE( ) perform identically; SYSDATE is compatible with Oracle syntax, and GETDATE( ) is compatible with Microsoft SQL Server syntax.
- SYSDATE and GETDATE( ) have no arguments, and return a DATE value.
- The SYSDATE or GETDATE( ) value is only retrieved during execution.
- Any required changes to the date (to incorporate a different time zone or Daylight Savings Time, for example) must occur at the system level. The date cannot be altered using SYSDATE or GETDATE( ).
- The SYSDATE and GETDATE( ) functions return the DATE data type. The DATE format is 'YYYY-MM-DD HH:MM:SS'.
- SYSDATE and GETDATE( ) are built-in functions and can be used anywhere a date expression may be used. They can be used in a **SELECT** projection list, a WHERE clause or to insert values. They cannot be used with a SUM or AVG aggregate (operands must be numeric) or with a COUNT aggregate (column names are expected).
- SYSDATE and GETDATE( ) return the same DATE value in a single SQL statement context.
- The literals TT_SYSDATE and ORA_SYSDATE are supported. TT_SYSDATE returns the TT_TIMESTAMP data type; ORA_SYSDATE returns the DATE data type. See Example 3.40 on page 121.

Example 3.36

In this example, invoking SYSDATE returns the same date and time for all rows in the table:

```
Command> SELECT SYSDATE FROM dual;
< 2006-09-03 10:33:43 >
1 row found.
```

Example 3.37 This example invokes SYSDATE to insert the current data and time into column DateCol:

```
Command> CREATE TABLE t (DateCol DATE);
Command> INSERT INTO t VALUES (SYSDATE);
1 row inserted.
Command> SELECT * FROM t;
< 2006-09-03 10:35:50 >
1 row found.
```

Example 3.38 In this example, GETDATE( ) inserts the same date value for each new row in the table, even if the query takes several seconds.

```
NSERT INTO t1 SELECT GETDATE(), col1
   FROM t2 WHERE ...;
```

Example 3.39 TO_CHAR is used with SYSDATE to return the DATE from table DUAL:

```
Command> SELECT TO_CHAR (SYSDATE) FROM dual;
< 2006-09-03 10:56:35 >
1 row found.
```

Example 3.40 The example invokes TT_SYSDATE to return the TT_TIMESTAMP data type and then invokes ORA_SYSDATE to return the DATE data type:

```
Command> select tt_sysdate from dual;
< 2006-10-31 20:02:19.440611 >
1 row found.
Command> select ora_sysdate from dual;
< 2006-10-31 20:02:30 >
1 row found.
```

TO_CHAR

TimesTen's TO_CHAR function converts a DATE, TIMESTAMP or numeric input value to a VARCHAR2.

If you are using TimesTen type mode, for information on the TO_CHAR function, refer to documentation from previous releases of TimesTen.

SQL Syntax

TO_CHAR (*Expression1*[, *Expression2* [, *Expression3*]])

Parameters

TO_CHAR has the parameters:

Parameter	Description
Expression1	A DATE, TIMESTAMP or numeric expression.
Expression2	The format string. If omitted, TimesTen uses the default date format (YYYY-MM-DD).
Expression3	A CHAR or VARCHAR2 expression to specify the NLS parameter which is currently ignored.

Description

- TO_CHAR supports different datetime format models depending on the data type specified for the expression. For information on the datetime format model used for TO_CHAR of data type DATE or TIMESTAMP, see "Datetime format models" on page 93. For information on the datetime format model used for TO_CHAR of data type TT_DATE or TT_TIMESTAMP, see "Format model for TO_CHAR of TimesTen types" on page 96.
- TO_CHAR supports different number format models depending on the numeric data type specified for the expression. For information on the number format model used for TO_CHAR of data type NUMBER or ORA_FLOAT, see "Number format models" on page 88. For information on the number format model used for TO_CHAR of all other numeric data types, see "Format model for TO_CHAR of TimesTen types" on page 96.

Example 3.41

```
SELECT FIRSTNAME,
       TO_CHAR (HireDate, 'MONTH DD, YYY')
       TO_CHAR (Salary, '$999.99')
FROM EMPLOYEES;
```

Example 3.42

```
SELECT TO_CHAR(-0.12,'$B99.9999') FROM t1;
```

Example 3.43

```
SELECT TO_CHAR(-12, 'B99999PR') FROM t1;
```

Example 3.44

```
SELECT TO_CHAR(-12,'FM99999') FROM t1;
```

Example 3.45

```
SELECT TO_CHAR(1234.1,'9,999.999') FROM t1;
```

TO_DATE

TimesTen's TO_DATE function converts a CHAR or VARCHAR2 argument to a value of DATE data type

If you are using TimesTen type mode, for information on the TO_DATE function, refer to documentation from previous releases of TimesTen.

SQL Syntax

TO_DATE (*Expression1*[, *Expression2* [, *Expression3*]])

Parameters

TO_DATE has the parameters:

Parameter	Description
Expression1	A CHAR or VARCHAR2 expression.
Expression2	The format string. This expression is usually required; it is optional only when *Expression1* is in the default date format YYYY-MM-DD HHMMSS.
Expression3	A CHAR or VARCHAR2 expression to specify the NLS parameter which is currently ignored.

Description

- You can use a datetime format model with the TO_DATE function. For more information on datetime format models, see "Datetime format models" on page 93.

Example 3.46

```
Command> SELECT TO_DATE ('1999, JAN 14', 'YYYY, MON DD') FROM dual;
< 1999-01-14 00:00:00 >
1 row found.
```

Example 3.47

```
Command> SELECT TO_CHAR(TO_DATE('1999-12:23','YYYY-MM:DD')) FROM
dual;
< 1999-12-23 00:00:00 >
1 row found.
```

Example 3.48

```
Command> SELECT TO_CHAR(TO_DATE('12-23-1997 10 AM:56:20',
'MM-DD-YYYY HH AM:MI:SS'),'MONTH,DD YYYY HH:MI-SS') FROM dual;
< DECEMBER ,23 1997 10:56-20 >
1 row found.
```

TO_NUMBER

Converts an expression whose value is of type CHAR, VARCHAR2, NCHAR, NVARCHAR2, BINARY_FLOAT or BINARY_DOUBLE to a value of NUMBER type.

SQL Syntax

TO_NUMBER (*Expression*[, *format[,nlsparam]])*

Parameters

TO_NUMBER has the parameters:

Parameter	Description
Expression	The expression to be converted.
format	If specified, the format is used to convert Expression to a value of NUMBER type. The format consists of a format string that identifies the number format model. The format string can be either a constant or a parameter.
nlsparam	Optional NLS parameter setting. This parameter is not currently supported and will result in an error if used.

Description

- You can use a number format model with the TO_NUMBER function. For more information on number format models, see “Number format models” on page 88.

Example 3.49

```
Command> SELECT TO_NUMBER ('100.00', '999D99') FROM DUAL
< 100 >
1 row found.
```

Example 3.50

```
Command> SELECT TO_NUMBER ('1210.73', '9999.99') FROM DUAL;
< 1210.73 >
1 row found.
```

TRUNC (expression)

Returns a number truncated to a certain number of decimal places.

SQL Syntax

TRUNC (*Expression [,m]*)

Parameters

TRUNC has the parameters:

Parameter	Description
Expression	The Expression to truncate. Operands must be of type NUMBER. An error is returned if operands are not of type NUMBER. The value returned is of type NUMBER.
[,m]	The number of decimal places to truncate to. If *m* is omitted, then the number is truncated to 0 places; *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

Example 3.51

```
SELECT TRUNC (15.79,1) FROM DUAL;
< 15.7 >
1 row found.
```

Example 3.52

```
SELECT TRUNC (15.79,-1) FROM DUAL;
< 10 >
1 row found.
```

TRUNC (date)

Returns date with the time portion of the day truncated to the unit specified by the format model *fmt*. The value returned is of type DATE. If you do not specify *fmt*, then date is truncated to the nearest day.

SQL Syntax

TRUNC (*date [,fmt]*)

Parameters

TRUNC (date) has the parameters:

Parameter	Description
date	The date that is truncated. Specify the DATE data type for date. The function returns data type DATE with the time portion of the day truncated to the unit specified by the format model. If you do not specify *fmt*, the date is truncated to the nearest day. An error is returned if you do not specify the DATE data type.
[,fmt]	The format model truncating unit. Specify either a constant or a parameter for *fmt*.

Description

The following table lists the format models you can use with the TRUNC (date) function and the truncating unit associated with each format model. The default model, 'DD' returns the date truncated to the day with a time of midnight:

Format Model	Truncating Unit
CC SCC	One greater than the first two digits of a four-digit year
SYYYY YYYY YEAR SYEAR YYY YY Y	Year
IYYY IY IY I	ISO Year
Q	Quarter

Format Model	Truncating Unit
MONTH MON MM RM	Month
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

Example 3.53

```
Command> SELECT TRUNC (TO_DATE ('27-OCT-92','DD-MON-YY'),'YEAR')
FROM DUAL;
< 2092-01-01 00:00:00 >
1 row found.
```

TT_HASH

The TT_HASH function returns the hash value of an expression or list of expressions. This value is the value that would be used by a hash index.

SQL Syntax

TT_HASH(*Expression* [,...])

Parameters

TT_HASH has the parameters:

Parameter	Description
Expression [,...]	One or more expressions to be used to determine the hash value of the expression or list of expressions.

Description

- Each expression must have a known data type and must be non-nullable. The hash value of the expression depends on both the value of the expression and its type. For example, TT_HASH of an TT_INTEGER with value 25 may be different from TT_HASH of a NUMBER or BINARY_DOUBLE with value 25. If you specify a list of expressions, the TT_HASH result will depend on the order of the expressions in the list.
- Since constants and expressions that are not simple column references are subject to internal typing rules, over which applications have no control, the best way to ensure that TT_HASH computes the desired value for expressions that are not simple column references is to **CAST** the expression to the desired type.
- The result type of TT_HASH is TT_INTEGER in 32-bit mode and TT_BIGINT in 64 bit mode.
- TT_HASH can be used in a SQL statement anywhere an expression can be used. For example, TT_HASH can be used in a SELECT list, a WHERE or HAVING clause, an ORDER BY clause, or a GROUP BY clause.
- The output of error messages, trace messages, and ttAXactAdmin display the hash value as a signed decimal so that the value matches TT_HASH output.

Example 3.54 The following query finds the set of rows whose primary key columns hash to a given hash value:

```
SELECT * FROM t1
       WHERE TT_HASH(pkey_col1, pkey_col2, pkey_col3) = 12345678;
```

UNISTR

The UNISTR takes as its argument, a string that resolves to data of type NVARCHAR2 and returns the value in UTF-16 format. Unicode escapes are supported; that is you can specify the Unicode encoding value of the characters in the string.

SQL Syntax

UNISTR (*'String'*)

Parameters

UNISTR has the parameter:

Parameter	Description
'String'	The string passed to the UNISTR function. The string resolves to type NVARCHAR2; the value returned is in UTF-16 format. You can specify Unicode escapes as part of the string.

Example 3.55

The following example invokes the UNISTR function passing as an argument the string 'A\00E4a'; The value returned is the value of the string in UTF-16 format:

```
Command> SELECT UNISTR ('A\00E4a') FROM DUAL;
<Aäa>
 1 row found.
```

String functions

TimesTen supports the string functions in SELECT statements:

- SUBSTR
- INSTR
- LENGTH

A selected value that specifies a string function causes the SELECT result to be materialized. This causes overhead in both time and space.

SUBSTR

Returns a CHAR, VARCHAR2 or NVARCHAR2 that represents a substring of a CHAR or NCHAR string. The returned substring is of a specified number of characters, beginning from a designated starting point, relative to either the beginning or end of the string.

SQL Syntax

{SUBSTR | SUBSTRB | SUBSTR4}=(*char*, *m*, *n*)

Parameters

SUBSTR has the parameters:

Parameter	Description
char	The string for which this function returns a substring If *char* is a CHAR string, the result is a CHAR or VARCHAR2 string. If *char* is a NCHAR string, the result is a NVARCHAR2 string.
m	The position at which to begin the substring. If *m* is positive, the first character of the returned string is *m* characters from the beginning of the string specified in *char.* Otherwise it is *m* characters from the end of the string. If ABS(*m)* is bigger than the length of the character string, a NULL value is returned.
n	The number of characters to be included in the substring. If *n* is omitted, all characters to the end of the string specified in *char* are returned. If *n* is less than 1 or if *char*, *m or n* is NULL, NULL is returned.

Description

- SUBSTR calculates lengths using characters as defined by character set. SUBSTRB uses bytes instead of characters. SUBSTR4 uses UCS4 code points.

Example 3.56 Select the first three characters of name:

```
SELECT SUBSTR(name,1,3) FROM employees;
```

Example 3.57 Select the last five characters of name:

```
SELECT SUBSTR(name,-1,5) FROM employees;
```

INSTR

Determines the first position, if any, at which one string occurs within another. If the substring does not occur in the string, then 0 is returned. The position returned is always relative to the beginning of *CharExpr2*. INSTR returns type NUMBER.

If you are using TimesTen type mode, for information on the INSTR function, refer to documentation from previous releases of TimesTen.

SQL Syntax

{INSTR | INSTRB | INSTR4} (*CharExpr2*, *CharExp1* [,*m*[,*n*]])

Parameters

INSTR has the parameters:

Parameter	Description
CharExpr1	The substring to be found in string *CharExpr2*. If *CharExpr1* does not occur in *CharExpr2*, then zero is returned. If either string is of length zero, NULL is returned.
CharExpr2	The string to be searched to find the position of *CharExpr1*.
m	The optional position at which to begin the search. If *m* is specified as zero, the result is zero. If *m* is positive, the search begins at the *CharExpr2*+*m*. If *m* is negative, the search begins *m* characters from the end of *CharExpr2*.
n	If *n* is specified it must be a positive value and the search returns the position of the *n*th occurrence of *CharExpr1*

Description

- INSTR calculates strings using characters as defined by character set. INSTRB uses bytes instead of characters. INSTR4 uses UCS4 code points.

Example 3.58

The following example uses instr to determine the position at which the substring ‘ing’ occurs in the string ‘Washington’:

```
Command>  select instr ('Washington', 'ing') from dual;
< 5 >
1 row found.
```

LENGTH

Returns the length of a given character string in an expression. LENGTH returns type NUMBER.

If you are using TimesTen type mode, for information on the LENGTH function, refer to documentation from previous releases of TimesTen.

SQL Syntax

{LENGTH|LENGTHB|LENGTH4} (*CharExpr*)

Parameters

LENGTH has the parameter:

Parameter	Description
CharExpr	The string for which to return the length.

Description

- The LENGTH functions return the length of *CharExpr*. LENGTH calculates the length using characters as defined by the character set. LENGTHB uses bytes rather than characters. LENGTH4 uses UCS4 code points.

Example 3.59 To determine the length of the string ‘William’:

```
Command> SELECT LENGTH('William') from dual;
< 7 >
1 row found.
```

4

Search Conditions

A search condition specifies criteria for choosing rows to select, update, or delete. Search conditions are parameters that can exist in clauses and expressions of any DML statements, such as **SELECT**, **UPDATE** and **DELETE** and some DDL statements, such as **CREATE VIEW**.

Search condition general syntax

A search condition is a single predicate or several predicates connected by the logical operators AND or OR. A predicate is an operation on expressions that evaluates to TRUE, FALSE, or UNKNOWN. If a predicate evaluates to TRUE for a row, the row qualifies for further processing. If the predicate evaluates to FALSE or NULL for a row, the row is not available for operations.

SQL syntax

[NOT]
{*BetweenPredicate* | *ComparisonPredicate* | *InPredicate* | *LikePredicate*|
NullPredicate | *InfinitePredicate* | *NanPredicate* |
QuantifiedPredicate |(*SearchCondition*)}
[{AND | OR} [NOT]
{*BetweenPredicate* | *ComparisonPredicate* |*InPredicate* | *LikePredicate*|
NullPredicate | *QuantifiedPredicate* |(*SearchCondition*)}
] [...]

Parameters

NOT, AND, OR	Logical operators with the following functions: • NOT negates the value of the predicate that follows it. • AND evaluates to TRUE if both the predicates it joins evaluate to TRUE. • OR evaluates to TRUE if either predicate it joins evaluates to TRUE, and to FALSE if both predicates evaluates to FALSE. • See "Description" on page 137 for a description of how these operators work when predicates evaluate to NULL.
BetweenPredicate	Determines whether an expression is within a certain range of values. For example: A BETWEEN B AND C is equivalent to A >= B AND A<= C.
ComparisonPredicate	Compares two expressions or list of two expressions using one of the operators <, <=, >, >=, =, <>.
InPredicate	Determines whether an expression or list of expressions matches an element within a specified set.
ExistsPredicate	Determines whether a subquery returns any row.
LikePredicate	Determines whether an expression contains a particular character string pattern.
NullPredicate	Determines whether a value is NULL.
InfinitePredicate	Determines whether an expression is infinite (positive or negative infinity).
NanPredicate	Determines whether an expression is the undefined result of an operation ("not a number.")
QuantifiedPredicate	Determines whether an expression or list of expressions bears a particular relationship to a specified set.
(*SearchCondition*)	One of the above predicates, enclosed in parentheses.

Description

- Predicates in a search condition are evaluated as follows:
  - Predicates in parentheses are evaluated first.
  - NOT is applied to each predicate.
  - AND is applied next, left to right.
  - OR is applied last, left to right.

  The following figure shows the values that result from logical operations. A question mark (?) represents the NULL value.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

- When the search condition for a row evaluates to NULL, the row doesn't satisfy the search condition and the row is not operated on.
- You can compare only compatible data types.
  - TT_TINYINT, TT_SMALLINT, TT_INTEGER, TT_BIGINT, NUMBER, BINARY_FLOAT and BINARY_DOUBLE are compatible.
  - CHAR, VARCHAR2, BINARY, and VARBINARY are compatible, regardless of length.
  - CHAR, VARCHAR2, NCHAR, NVARCHAR2, TT_TIME, DATE and TIMESTAMP are compatible.
- See Chapter 3, "Expressions," for information on value extensions during comparison operations.
- See "Numeric data types," for information about how TimesTen compares values of different but compatible types.

ALL/ NOT IN predicate (subquery)

The ALL or NOT IN predicate indicates that the operands on the left side of the comparison must compare in the same way with all of the values that the subquery returns. The ALL predicate evaluates to TRUE if the expression or list of expressions relates to all rows returned by the subquery as specified by the comparison operator. Similarly, the NOT IN predicate evaluates to TRUE if the expression or list of expressions does not equal the value returned by the subquery.

SQL syntax

RowValueConstructor {CompOp ALL| NOT IN} *(Subquery)*

The syntax for *RowValueConstructor*:

RowValueConstructorElement |
(*RowValueConstuctorList*) |
Subquery

The syntax for *RowValueConstructorList:*

RowValueConstructorElement
[{, *RowValueConstructorElement*} ...]

The syntax for *RowValueConstructorElement:*

Expression | NULL

The syntax for *CompOp:*

{= | <> | > | >= | < | <= }

Parameters

Expression	The syntax of expressions is defined under "Expression specification" on page 72. Both numeric and non-numeric expressions are allowed for ALL predicates, but both expression types must be compatible with each other.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.
Subquery	The syntax of subqueries is defined under "Subqueries" on page 76

Description

- The ALL predicate, which returns zero or more rows, uses a *comparison operator* modified with the keyword ALL. See "Numeric data types," for information about how TimesTen compares values of different but compatible types.
- If <RowValueConstructorList> is specified only the operators = and <> are allowed.

Example 4.1 Examples of NOT IN with subqueries:

```
SELECT * FROM customers
WHERE cid NOT IN
(SELECT cust_id FROM returns)
AND cid > 5000;
```

```
SELECT * FROM customers
WHERE cid NOT IN
(SELECT cust_id FROM returns)
AND cid NOT IN
(SELECT cust_id FROM complaints);
```

```
SELECT COUNT(*) From customers
WHERE cid NOT IN
(SELECT cust_id FROM returns)
AND cid NOT IN
(SELECT cust_id FROM complaints);
```

Example 4.2 Select all books that are not from `exclBookList` or if the price of the book is higher than $20.

```
SELECT * FROM books WHERE id NOT IN (SELECT id FROM exclBookList)
OR books.price>20;
```

Example 4.3 The following query returns the employee_id and job_id from the job_history table. It illustrates use of expression list and subquery with the NOT IN predicate.

```
Command> SELECT employee_id, job_id FROM job_history WHERE
(employee_id, job_id) NOT IN (SELECT employee_id, job_id FROM
employees);
< 101, AC_ACCOUNT >
< 101, AC_MGR >
< 102, IT_PROG >
< 114, ST_CLERK >
< 122, ST_CLERK >
< 176, SA_MAN >
< 200, AC_ACCOUNT >
< 201, MK_REP >
8 rows found.
```

ALL/NOT IN predicate (value list)

The ALL/NOT IN *quantified predicate* compares an expression or list of expressions with a list of specified values. The ALL predicate evaluates to TRUE if all the values in the *ValueList* relate to the expression or list of expressions as indicated by the comparison operator. Similarly, the NOT IN predicate evaluates to TRUE if the expression or list of expressions does not equal one of the values in the list.

SQL syntax

RowValueConstructor {CompOp ALL | NOT IN} *ValueList*

The syntax for *RowValueConstructor*:

RowValueConstructorElement |
(*RowValueConstuctorList*) |

The syntax for *RowValueConstructorList*:

RowValueConstructorElement
[{, *RowValueConstructorElement*} ...]

The syntax for *RowValueConstructorElement:*

Expression | NULL

The syntax for *CompOp:*

{= | <> | > | >= | < | <= }

The syntax for more than one element in the *ValueList:*

({*Constant* | *?* | *:DynamicParameter*} [,...])

The syntax for one element in the *ValueList* not enclosed in parentheses:

Constant | *?* | *:DynamicParameter*

The syntax for an empty *ValueList:*

( )

The syntax for the *ValueList* for a list of expressions*:*

(({*Constant* | *?* | *:DynamicParameter*} [,...]))

Parameters

Expression	Specifies a value to be obtained. The values in *ValueList* must be compatible with the expression. For information on the syntax of expressions, see "Expression specification" on page 72.
=	Is equal to.
<>	Is not equal to.

>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.
ALL	The predicate is TRUE if *all* the values in the *ValueList* relate to the expression or list of expressions as indicated by the comparison operator.
ValueList	A list of values that are compared against the expression's or list of expression's value. The *ValueList* cannot contain a column reference or a subquery. The *ValueList* can be nested if the left operand of the *ValueList* is a list. Elements of the *ValueList:* • *Constant*—Indicates a specific value. See "Constants" on page 82. • *?,:DynamicParameter*—Placeholder for a dynamic parameter in a prepared SQL statement. The value of the dynamic parameter is supplied when the statement is executed. • Empty list, which are sometimes generated by SQL generation tools.

Description

- If *X* is the value of *Expression*, and (*a,b, ..., z*) represents the elements in *ValueList*, and *OP* is a comparison operator, then the following is true:
  - `X OP ALL (a,b,...,z)` is equivalent to `X OP a AND X OP b AND...AND X OP z`.
- If *X* is the value of *Expression* and (*a,b,..., z*) are the elements in a *ValueList*, then the following is true:
  - `X NOT IN (a,b,...,z)` is equivalent to `NOT (X IN (a,b,...,z))`.
- Character strings are compared according to the ASCII collating sequence for ASCII data.
- NULL cannot be specified in *ValueList*.
- See "Numeric data types" on page 29 for information about how TimesTen compares values of different but compatible types.
- NOT IN or NOT EXISTS with ALL can be specified in an OR expression.
- IN and EXISTS with ALL can be specified in an OR expression.
- When evaluating an empty *ValueList*, the result of *Expression* NOT IN is true.

- If <RowValueConstructorList> is specified only the operators = and <> are allowed.

Example 4.4 To query an empty select list for a NOT IN condition:

```
SELECT * FROM t1 WHERE x1 NOT IN ();
```

ANY/ IN predicate (subquery)

An ANY predicate compares two expressions using a *comparison operator*. The predicate evaluates to TRUE if the first expression relates to *any row* returned by the subquery as specified by the comparison operator. Similarly, the IN predicate compares an expression or list of expressions with a table subquery. The IN predicate evaluates to TRUE if the expression or list of expressions is equal to a value returned by a subquery.

SQL syntax

RowValueConstructor {CompOp ANY| IN} *(Subquery)*

The syntax for *RowValueConstructor*:

RowValueConstructorElement |
(*RowValueConstuctorList*) |
Subquery

The syntax for *RowValueConstructorList:*

RowValueConstructorElement
[{, *RowValueConstructorElement*} ...]

The syntax for *RowValueConstructorElement:*

Expression | NULL

The syntax for *CompOp:*

{= | <> | > | >= | < | <= }

Parameters

Expression	The syntax of expressions is defined under "Expression specification" on page 72. Both numeric and non-numeric expressions are allowed for ANY predicates, but both expression types must be compatible with each other.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.
Subquery	The syntax of subqueries is defined under "Subqueries" on page 76

Description

- The ANY predicate, which returns zero or more rows, uses a *comparison operator* modified with the keyword ANY. See "Numeric data types," for

information about how TimesTen compares values of different but compatible types.

Example 4.5 Retrieve a list of customers having at least one un-shipped order:

```
SELECT customers.name FROM customers
WHERE customers.id = ANY
(SELECT orders.custid FROM orders
WHERE orders.status = 'un-shipped');
```

Example 4.6 Example of IN predicate with subquery. SELECTs customers having at least one un-shipped order:

```
SELECT customers.name FROM customers
WHERE customers.id IN
(SELECT orders.custid FROM orders
WHERE orders.status = 'un-shipped');
```

Example 4.7 Uses an aggregate query that specifies a subquery with IN, to find the maximum price of a book in the `exclBookList`:

```
SELECT MAX(price) FROM books WHERE id IN (SELECT id FROM
exclBookList);
```

Example 4.8 Illustrates the use of a list of expressions with the IN predicate and a subquery.

```
SELECT * from t1 where (x1,y1) in (SELECT x2,y2 from t2);
```

Example 4.9 Illustrates the use of a list of expressions with the ANY predicate and a subquery.

```
SELECT * from t1 where (x1,y1) < ANY (SELECT x2,y2 from t2);
```

Example4.10 The following example illustrates the use of a list of expressions with the ANY predicate.

```
Command> columnlabels on;
Command> SELECT * FROM t1;
X1, Y1
< 1, 2 >
< 3, 4 >
2 rows found.
Command> SELECT * FROM t2;
X2, Y2
< 3, 4 >
< 1, 2 >
2 rows found.
Command> SELECT * FROM t1 WHERE (x1,y1) < ANY (SELECT x2,y2 from
t2);
X1, Y1
< 1, 2 >
```

```
1 row found.
```

ANY/ IN predicate (value list)

The ANY/IN *quantified predicate* compares an expression or list of expressions with a list of specified values. The ANY predicate evaluates to TRUE if one or more of the values in the *ValueList* relate to the expression or list of expressions as indicated by the comparison operator. Similarly, the IN predicate evaluates to TRUE if the expression or list of expressions is equal to one of the values in the list.

SQL syntax

RowValueConstructor {CompOp {ANY| SOME} | IN} *ValueList*

The syntax for *RowValueConstructor*:

RowValueConstructorElement |
(*RowValueConstuctorList*) |

The syntax for *RowValueConstructorList*:

RowValueConstructorElement
[{, *RowValueConstructorElement*} ...]

The syntax for *RowValueConstructorElement:*

Expression | NULL

The syntax for *CompOp:*

{= | <> | > | >= | < | <= }

The syntax for more than one element in the *ValueList:*

({*Constant* | *?* | *:DynamicParameter*} [,...])

The syntax for one element in the *ValueList* not enclosed in parentheses:

Constant | *?* | *:DynamicParameter*

The syntax for an empty *ValueList:*

( )

The syntax for the *ValueList* for a list of expressions*:*

(({*Constant* | *?* | *:DynamicParameter*} [,...]))

Parameters

Expression	Specifies a value to be obtained. The values in *ValueList* must be compatible with the expression. For information on the syntax of expressions, see "Expression specification" on page 72.
=	Is equal to.
<>	Is not equal to.

>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.
{ANY| SOME}	The predicate is TRUE if *one or more* of the values in the *ValueList* relate to the expression or list of expressions as indicated by the comparison operator. SOME is a synonym for ANY.
ValueList	A list of values that are compared against the expression's or list of expression's value. The *ValueList* cannot contain a column reference or a subquery. The *ValueList* can be nested if the left operand of the *ValueList* is a list. Elements of the *ValueList:* • *Constant*—Indicates a specific value. See "Constants" on page 82. • *?,:DynamicParameter*—Placeholder for a dynamic parameter in a prepared SQL statement. The value of the dynamic parameter is supplied when the statement is executed. • Empty list, which are sometimes generated by SQL generation tools.

Description

- If *X* is the value of *Expression*, and (*a,b, ..., z*) represents the elements in *ValueList*, and *OP* is a comparison operator, then the following is true:
  - `X OP ANY (a,b,...,z)` is equivalent to `X OP a OR X OP b OR...OR X OP z`.
- If *X* is the value of *Expression* and (*a,b,..., z*) are the elements in a *ValueList*, then the following is true:
  - `X IN (a,b,...,z)` is equivalent to `X = a OR X = b OR...OR X = z`.
- Character strings are compared according to the ASCII collating sequence for ASCII data.
- NULL cannot be specified in *ValueList*.
- See "Numeric data types" on page 29 for information about how TimesTen compares values of different but compatible types.
- When evaluating an empty *ValueList*, the result of *Expression* IN is false.

Example4.11 Select all item numbers containing orders of 100, 200, or 300 items.

```
SELECT DISTINCT OrderItems.ItemNumber
FROM OrderItems
WHERE OrderItems.Quantity = ANY (100, 200, 300)
```

Example4.12 Get part numbers of parts whose weight is 12, 16, or 17.

```
SELECT Parts.PartNumber FROM Parts
WHERE Parts.Weight IN (12, 16, 17);
```

Example4.13 Get part number of parts whose serial number is '1123-P-01', '1733-AD-01', :*SerialNumber* or :*SerialInd*, where :*SerialNumber* and :*SerialInd* are dynamic parameters whose values are supplied at runtime.

```
SELECT PartNumber FROM Purchasing.Parts
WHERE SerialNumber
IN ('1123-P-01', '1733-AD-01',:SerialNumber, :SerialInd);
```

Example4.14 To query an empty select list for IN condition:

```
SELECT * FROM t1 WHERE x1 IN ();
```

Example4.15 Illustrates the use of a list of expressions with in:

```
SELECT * FROM t1 WHERE (x1,y1) IN ((1,2), (3,4));
```

Example4.16 The following example illustrates the use of a list of expressions for the IN predicate. The query returns the department_name for departments with department_id = 240 and location_id = 1700. Note: The expression on the right side of the IN predicate must be enclosed in double parentheses (( )).

```
Command> select department_name from departments where
(department_id, location_id) in ((240,1700));
< Government Sales >
1 row found.
```

BETWEEN predicate

A BETWEEN predicate determines whether a value is:

- Greater than or equal to a second value, and
- Less than or equal to a third value.

The predicate evaluates to TRUE if a value falls within the specified range.

SQL syntax

Expression1 [NOT] BETWEEN *Expression2* AND *Expression3*

Parameters

Expression1, *Expression2*, *Expression3*	The syntax for expressions is defined in "Expression specification" on page 72. Both numeric and non-numeric expressions are allowed in BETWEEN predicates, but all expressions must be compatible with each other.

Description

- BETWEEN evaluates to FALSE and NOT BETWEEN evaluates to TRUE if the second value is greater than the third value.
- Consult the following table if either *Expression2* or *Expression3* is NULL for BETWEEN or NOT BETWEEN:

Expression2	Expression3	BETWEEN	NOT BETWEEN
<= *Expression1*	NULL	NULL	NULL
> *Expression1*	NULL	FALSE	TRUE
NULL	>= *Expression1*	NULL	NULL
NULL	< *Expression1*	NULL	NULL

- *Expression2* and *Expression3* constitute a range of possible values for which *Expression2* is the lowest possible value and *Expression3* is the highest possible value within the specified range. In the BETWEEN predicate, the low value must be specified first.

  Comparisons are conducted as described in "Comparison predicate" on page 151.
- The BETWEEN predicate is not supported for NCHAR types.

Example4.17 Parts sold for under $250.00 and over $1500.00 are discounted 25%.

```
UPDATE Purchasing.Parts
SET SalesPrice = SalesPrice * 0.75
WHERE SalesPrice NOT BETWEEN 250.00 AND 1500.00;
```

Comparison predicate

A *comparison* predicate compares two expressions using a *comparison operator*. The predicate evaluates to TRUE if the first expression relates to the second expression as specified by the comparison operator.

SQL syntax

RowValueConstructor CompOp RowValueConstructor2

The syntax for *RowValueConstructor*:

RowValueConstructorElement |
(*RowValueConstuctorList*) |
Scalar *Subquery*

The syntax for *RowValueConstructorList:*

RowValueConstructorElement
[{, *RowValueConstructorElement*} ...]

The syntax for *RowValueConstructor2 (*one expression)

Expression

The syntax for *RowValueConstructor2 (*list of expressions)

((*Expression*[,...]))

The syntax for *CompOp:*

{= | <> | > | >= | < | <= }

Parameters

Expression	The syntax for expressions is defined under "Expression specification" on page 72. Both numeric and non-numeric expressions are allowed in comparison predicates, but both expressions must be compatible with each other.
ScalarSubquery	A subquery that returns a single value. Scalar subqueries and their restrictions are defined under "Subqueries" on page 76.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.

Description

- Character strings are compared according to the ASCII collating sequence for ASCII data.
- If there is a NULL value on either or both sides of a comparison predicate, the predicate evaluates to NULL and the row is not operated on.
- If <RowValueConstructorList> is specified only the operators = and <> are allowed.
- See "Numeric data types" on page 29 for information about how TimesTen compares values of different but compatible types.

Example4.18 Retrieve part numbers of parts requiring fewer than 20 delivery days:

```
SELECT PartNumber FROM Purchasing.SupplyPrice
WHERE DeliveryDays < 20;
```

Example4.19 The query returns the last_name of employees where salary = 9500 and commission_pct = .25. Note: The expression on the right side of the equal sign must be enclosed in double parentheses (( )).

```
Command> select last_name from employees
where(salary,commission_pct) = ((9500,.25));
< Bernstein >
1 row found.
```

Example4.20 The query returns the last_name of the employee whose manager_id = 205. The employee's department_id and manager_id is stored in both the employees and departments tables. A subquery is used to extract the information from the departments table.

```
Command> select last_name from employees where (department_id,
manager_id) = (select department_id, manager_id from departments
where manager_id = 205);
< Gietz >
1 row found.
```

EXISTS predicate

An EXISTS predicate checks for the existence or nonexistence of a table subquery. The predicate evaluates to TRUE if the subquery returns at least one row for EXISTS and no rows for NOT EXISTS

SQL syntax

[NOT] EXISTS (*Subquery*)

Parameters

The EXISTS predicate has the following parameter:

Subquery	The syntax of subqueries is defined under "Subqueries" on page 76

Description

- When a subquery is introduced with EXISTS, the subquery functions as an *existence* test. EXISTS tests for the presence or absence of an empty set of rows. If the subquery returns at least one row, the subquery evaluates to true.
- When a subquery is introduced with NOT EXISTS, the subquery functions as an *absence* test. NOT EXISTS tests for the presence or absence of an empty set of rows. If the subquery returns no rows, the subquery evaluates to true.
- If join order is issued using the **ttOptSetOrder** built-in procedure that conflicts with the join ordering requirements of the NOT EXISTS subquery, the specified join order is ignored, TimesTen issues a warning and the query is executed.
- The following table describes supported and unsupported usages of EXISTS and NOT EXISTS in TimesTen;

Query/subquery description	Not Exists	Exists
Aggregates in subquery	Supported	Supported
Aggregates in main query	Supported	Supported
Subquery in OR clause	Supported	Supported
Join ordering using the **ttOptSetOrder** built-in procedure	Limited support	Supported

Example4.21 Get a list of customers having at least one unshipped order.

```
SELECT customers.name FROM customers
WHERE EXISTS (SELECT 1 FROM orders
WHERE customers.id = orders.custid
AND orders.status = 'un-shipped');
```

Example4.22 Get a list of customers having at no unshipped orders.

```
SELECT customers.name FROM customers
WHERE  NOT EXISTS (SELECT 1 FROM orders
WHERE customers.id = orders.custid
AND orders.status = 'un-shipped');
```

IS INFINITE predicate

An IS INFINITE predicate determines whether an expression is infinite (positive infinity (INF) or negative infinity (-INF).

SQL syntax

Expression IS [NOT] INFINITE

Parameters

Expression	Expression to test.

Description

- An IS INFINITE predicate evaluates to TRUE if the expression is positive or negative infinity.
- An IS NOT INFINITE predicate evaluates to TRUE if expression is neither positive nor negative infinity.
- The expression must either resolve to a numeric data type or to a data type that can be implicitly converted to a numeric data type.
- Two positive infinity values are equal to each other. Two negative infinity values are equal to each other.
- Expressions containing floating-point values may generate Inf, -Inf, or NaN. This can occur either because the expression generated overflow or exceptional conditions or because one or more of the values in the expression was Inf, -Inf, or NaN. Inf and NaN are generated in overflow or division by 0 conditions.
- Inf, -Inf, and NaN values are not ignored in aggregate functions; NULL values are. If you wish to exclude Inf and NaN from aggregates (or from any selection), use both the IS NOT NAN and IS NOT INFINITE predicates.
- Negative infinity (-INF) sorts lower than all other values. Positive infinity (INF) sorts higher than all other values, but lower than NaN (“not a number) and the NULL value.
- For more information on Inf and Nan, see “INF and NAN” on page 51.

IS NAN predicate

An IS NAN predicate determines whether an expression is the undefined result of an operation (that is, is “not a number” or NaN).

SQL syntax

Expression IS [NOT] NAN

Parameters

Expression	Expression to test.

Description

- An IS NAN predicate evaluates to TRUE if the expression is “not a number.”
- An IS NOT NAN predicate evaluates to TRUE if expression is not “not a number.”
- The expression must either resolve to a numeric data type or to a data type that can be implicitly converted to a numeric data type.
- Two NaN (“not a number”) values are equal to each other.
- Expressions containing floating-point values may generate Inf, -Inf, or NaN. This can occur either because the expression generated overflow or exceptional conditions or because one or more of the values in the expression was Inf, -Inf, or NaN. Inf and NaN are generated in overflow or division by 0 conditions.
- Inf, -Inf, and NaN values are not ignored in aggregate functions; NULL values are. If you wish to exclude Inf and NaN from aggregates (or from any selection), use both the IS NOT NAN and IS NOT INFINITE predicates.
- NaN (“not a number”) sorts higher than all other values including positive infinity, but lower than the NULL value.
- For more information on Inf and Nan, see “INF and NAN” on page 51.

IS NULL predicate

An *IS NULL predicate* determines whether an expression has the value NULL. The predicate evaluates to TRUE if the expression is NULL. If the NOT option is used, the predicate evaluates to TRUE if the expression is NOT NULL.

SQL syntax

{*ColumnName* | *Constant* | (*Expression*)} IS [NOT] NULL

Parameters

ColumnName	The name of a column from which a value is to be taken; column names are discussed in Chapter 2, "Names."
Constant	A specific value. See "Constants" on page 82.
(*Expression*)	Expression to test.

Example4.23

Vendors with no personal contact names are identified.

```
SELECT *
FROM Purchasing.Vendors
WHERE ContactName IS NULL;
```

LIKE predicate

A *LIKE predicate* determines whether a CHAR, VARCHAR2, NCHAR, or NVARCHAR2 expression contains a given pattern. The predicate evaluates to TRUE if an expression contains the pattern.

SQL syntax

Expression [NOT] LIKE
{'*PatternString*'| {? | *:DynamicParameter*}}
[ESCAPE {'*EscapeChar*' | {? | *:DynamicParameter}*}]

Parameters

Expression	The syntax of expressions is presented in Chapter 3, "Expressions."
PatternString	Describes what you are searching for in the expression. The pattern may consist of characters only (including digits and special characters). For example, `NAME LIKE 'Annie'` evaluates to `TRUE` only for a name of Annie with no spaces. Upper case and lower case are significant. You can also use the predicate to test for a partial match by using the following symbols in the pattern: _ Represents any single character. For example, `BOB` and `TOM` both satisfy the predicate `NAME LIKE '_O_'`. % Represents any string of zero or more characters. For example, `MARIE` and `RENATE` both satisfy the predicate `NAME LIKE '%A%'`. You can use the _ and % symbols multiple times and in any combination in a pattern. You cannot use these symbols literally within a pattern unless you use the ESCAPE clause and precede the symbols with the escape character, described by the *EscapeChar* parameter.

EscapeChar	Describes an optional escape character which can be used to interpret the symbols _ and % literally in the pattern. The escape character must be a single character. When it appears in the pattern, it must be followed by the escape character itself, the _ symbol or the % symbol. Each such pair represents a single literal occurrence of the second character in the pattern. The escape character is always case sensitive. The escape character cannot be _ or %.
? :*DynamicParameter*	Indicates a dynamic parameter in a prepared SQL statement. The parameter value is supplied when the statement is executed.

Description

- As long as no escape character is specified, the _ or % in the pattern acts as a wild card character. If an escape character is specified, then the wild card or escape character that follows is treated literally. If the character following an escape character is not a wild card or the escape character, an error results.
- If the value of the expression, the pattern, or the escape character is NULL, then the LIKE predicate evaluates to NULL and the row is not operated on.

Example4.24 Vendors located in states beginning with an “A” are identified.

```
SELECT VendorName FROM Purchasing.Vendors
WHERE VendorState LIKE 'A%';
```

Vendors whose names begin with `ACME_` are identified (note use of the `ESCAPE` clause).

```
SELECT VendorName FROM Purchasing.Vendors
WHERE VendorName LIKE 'ACME!_%' ESCAPE '!';
```

NCHAR and NVARCHAR2

The LIKE predicate can be used for pattern matching of NCHAR and NVARCHAR2 strings. The pattern matching characters are:

U+005F SPACING UNDERSCORE	Represents any single Unicode character.
U+0025 PERCENT SIGN	Represents any string of zero or more Unicode characters.

Description

- The escape character is similarly supported as a single Unicode character or parameter.
- The types of the LIKE operands can be any combination of character types.
- Case and accent insensitive NLS_SORT is supported with the LIKE predicate.

Examples

In these examples, the Unicode character U+0021 EXCLAMATION MARK is being used to escape the Unicode character U+005F SPACING UNDERSCORE. Unicode character U+0025 PERCENT SIGN is not escaped, and assumes its pattern matching meaning.

VendorName is an NCHAR or NVARCHAR2 column.

Example4.25

```
SELECT VendorName FROM Purchasing.Vendors
WHERE VendorName LIKE N'ACME!_%' ESCAPE N'!';
```

is equivalent to:

```
SELECT VendorName FROM Purchasing.Vendors
WHERE VendorName LIKE N'ACME!\u005F\u0025' ESCAPE N'!';
```

5

SQL Statements

This chapter discusses the SQL statements available in TimesTen SQL. For each statement, this chapter specifies the supported syntax, explains the parameters, and provides any other relevant information.

Data Manipulation Language (DML) statements, such as INSERT, UPDATE and DELETE, manipulate database objects. Data Definition Language (DDL) statements, such as CREATE TABLE and DROP TABLE, change the database schema.

Access Control and SQL statements

For each statement in this chapter, this guide gives a brief description of the privileges required if your TimesTen instance has Access Control enabled. For a description of Access Control Privileges, see Chapter 7, "Access Control Privileges." Some general guidelines to remember are:

- To CREATE any object within a data store, a user must have **DDL** privileges.
- If a user has no privileges, he or she can **SELECT** or **WRITE** on only those tables that he or she owns. The user can query any data that he or she owns or perform an INSERT into, DELETE from or UPDATE on any tables that he or she owns.To limit access to particular objects, an **Instance Administrator** can create those objects with a common user name, without privileges. Those objects are then available to that user name without **WRITE** or **SELECT** privileges.
- If a user has DDL privileges, he or she can CREATE, DROP or ALTER any object for any user.
- If a user has **WRITE** privileges, he or she can INSERT into, DELETE from or UPDATE any user's tables.
- If a user has **SELECT** privileges, he or she can query any user data.

List of SQL Statements

ALTER ACTIVE STANDBY PAIR

You can change an active standby pair by:

- Adding or dropping a subscriber data store
- Altering store attributes—only the PORT and TIMEOUT attributes can be set for subscribers
- Including tables, sequences or cache groups in replication
- Excluding tables, sequences or cache groups from replication

See "Changing the configuration of an active standby pair" on page 166 of *TimesTen to TimesTen Replication Guide*.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges.

SQL Syntax

```
ALTER ACTIVE STANDBY PAIR
    [ADD SUBSCRIBER FullStoreName] |
    [DROP SUBSCRIBER FullStoreName] |
    [ALTER STORE FullStoreName SET StoreAttribute]
  { INCLUDE | EXCLUDE }{TABLE | SEQUENCE | CACHE GROUP }
    [Owner.]{TableName | SequenceName | CacheGroupName} [,...]
```

Parameters

ALTER ACTIVE STANDBY has the parameters:

ADD SUBSCRIBER *FullStoreName*	Indicates an additional subscriber data store. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
DROP SUBSCRIBER *FullStoreName*	Indicates that updates should no longer be sent to the specified subscriber data store. This operation fails if the replication scheme has only one subscriber. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
ALTER STORE *FullStoreName* SET *StoreAttribute*	Indicates changes to the attributes of a data store. Only the PORT and TIMEOUT attributes can be set for subscribers. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.

FullStoreName	The data store, specified as one of the following: • SELF • The prefix of the data store file name For example, if the data store path is `directory/subdirectory/data.ds0`, then `data` is the data store name that should be used. This is the data store file name specified in the **DataStore** attribute of the DSN description with optional host ID in the form: *DataStoreName* [ON *Host*] *Host* can be either an IP address or a literal host name assigned to one or more IP addresses, as described in "Configuring host IP addresses" in the *TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: "MyHost-500".
{ INCLUDE | EXCLUDE }{TABLE | SEQUENCE | CACHE GROUP } [*Owner.*]{*TableName* | *SequenceName* | *CacheGroupName*} [,...]	Includes in or excludes from replication the tables, sequences or cache groups listed. INCLUDE adds the tables, sequences or cache groups to replication. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE removes the tables, sequences or cache groups from replication. Use one EXCLUDE clause for each object type (table, sequence or cache group). [*Owner.*]{*TableName* | *SequenceName* | *CacheGroupName*} [,...] is the list of table, sequence or cache group names, including optional owners, to be added to or removed from the data stores.

Description

- Your must stop the replication agent before altering the active standby pair.
- You may only alter the active standby pair replication scheme on the active data store. See "Changing the configuration of an active standby pair" on page 166 of *TimesTen to TimesTen Replication Guide* for more information.
- Use ADD SUBSCRIBER *FullStoreName* to add a subscriber to the replication scheme.
- Use DROP SUBSCRIBER *FullStoreName* to drop a subscriber from the replication scheme.

- Use ALTER STORE *FullStoreName* SET *StoreAttribute* to change the attributes for the specified data store. Only the PORT and TIMEOUT attributes can be set for subscribers.
- Use { INCLUDE | EXCLUDE }{TABLE | SEQUENCE | CACHE GROUP } [*Owner.*]{*TableName* | *SequenceName* | *CacheGroupName*} [,...] to include the listed tables, sequences or cache groups in the replication scheme, or to exclude them from the replication scheme. Use one INCLUDE clause for each object type (table, sequence or cache group). Use one EXCLUDE clause for each object type (table, sequence or cache group).

Example 5.1 Add a subscriber to the replication scheme.

```
ALTER ACTIVE STANDBY PAIR
    ADD SUBSCRIBER rep4;
```

Example 5.2 Drop two subscribers from the replication scheme.

```
ALTER ACTIVE STANDBY PAIR
    DROP SUBCRIBER rep3
    DROP SUBSCRIBER rep4;
```

Example 5.3 Alter the store attributes of the `rep3` and `rep4` data stores.

```
ALTER ACTIVE STANDBY PAIR
    ALTER STORE rep3 SET PORT 23000 TIMEOUT 180
    ALTER STORE rep4 SET PORT 23500 TIMEOUT 180;
```

Example 5.4 Add a table, a sequence and two cache groups to the replication scheme.

```
ALTER ACTIVE STANDBY PAIR
    INCLUDE TABLE my.newtab
    INCLUDE SEQUENCE my.newseq
    INCLUDE CACHE GROUP my.newcg1, my.newcg2;
```

See Also "CREATE ACTIVE STANDBY PAIR" on page 204
"DROP ACTIVE STANDBY PAIR" on page 276

ALTER CACHE GROUP

The ALTER CACHE GROUP statement allows changes to the state, interval and mode of AUTOREFRESH.

Updates on Oracle can be propagated back to the TimesTen cache group with the use of AUTOREFRESH. AUTOREFRESH can be enabled when the cache group is a user managed cache group or is declared both as READONLY and with an AUTOREFRESH clause.

Any values or states set by ALTER CACHE GROUP are persistent; they are stored in the data store and survive daemon and Cache Agent restarts.

For a description of cache group types, see "User and system managed cache groups" on page 211.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

This statement changes the mode of the cache group, which determines which rows are updated during an AUTOREFRESH:

```
ALTER CACHE GROUP [Owner.]GroupName
        SET AUTOREFRESH MODE
        {INCREMENTAL | FULL}
```

This statement changes the AUTOREFRESH interval on the cache group:

```
ALTER CACHE GROUP [Owner.]GroupName
        SET AUTOREFRESH INTERVAL IntervalValue
        {MINUTE[S] | SECOND[S] | MILLISECOND[S] }
```

This statement alters the state of AUTOREFRESH:

```
ALTER CACHE GROUP [Owner.]GroupName
        SET AUTOREFRESH STATE
        {ON | OFF | PAUSED}
```

.

[Owner.]GroupName	Name assigned to the new cache group.
AUTOREFRESH	Indicates that changes on Oracle should be automatically propagated to TimesTen. For details, see "AUTOREFRESH in Cache Groups" on page 219.

MODE	Determines which rows in the cache are updated during an autorefresh. If the INCREMENTAL clause is specified, TimesTen refreshes only rows that have been changed on Oracle since the last propagation. If the FULL clause is specified or if there is neither FULL nor INCREMENTAL clause specified, TimesTen updates all rows in the cache with each autorefresh. The default mode is INCREMENTAL.
INTERVAL *IntervalValue*	Indicates the interval at which autorefresh should occur in units of minutes, seconds or milliseconds. An integer value that specifies how often AUTOREFRESH should be scheduled, in minutes, seconds or milliseconds. The default value is 10 minutes. If the specified interval is not long enough for an AUTOREFRESH to complete, a runtime warning is generated and the next AUTOREFRESH waits until the current one finishes. An informational message is generated in the daemon log if the wait queue reaches 10.
STATE	Specifies whether AUTOREFRESH should be changed to on, off or paused. By default, the AUTOREFRESH STATE is on.
ON	AUTOREFRESH is scheduled to occur at the specified interval.
OFF	A scheduled AUTOREFRESH is cancelled, and TimesTen does not try to maintain the information necessary for an INCREMENTAL refresh. Therefore if AUTOREFRESH is turned on again at a later time, the first refresh is FULL.
PAUSED	A scheduled AUTOREFRESH is cancelled, but TimesTen tries to maintain the information necessary for an INCREMENTAL refresh. Therefore if AUTOREFRESH is turned on again at a later time, a full refresh may not be necessary.

Description

- A refresh does not occur immediately after issuing an ALTER CACHE GROUP...SET AUTOREFRESH STATE command. This statement only changes the state of AUTOREFRESH. When the transaction that contains the ALTER CACHE GROUP statement is committed, the Cache Agent is notified to schedule an AUTOREFRESH immediately, but the commit goes through without waiting for the completion of the refresh. The scheduling of the AUTOREFRESH is part of the transaction, but the refresh itself is not.
- If you issue an ALTER CACHE GROUP... SET AUTOREFRESH STATE OFF, and there is an autorefresh operation currently running, then:
  - If LockWait interval is 0, the ALTER statement will fail with a lock timeout error.

  - If LockWait interval is non-zero, then the current autorefresh transaction is preempted (rolled back), and the ALTER statement continues. This affects all cache groups with the same autorefresh interval.
- Replication cannot occur between cache groups with AUTOREFRESH and cache groups without AUTOREFRESH.

Example 5.5 This example demonstrates how to set up AUTOREFRESH on the AutorefreshCustomers cache group, which implements incremental refreshes, without supplying an Oracle Admin password to TimesTen.

Before creating a cache group that utilizes AUTOREFRESH, you must:

- Supply the cache administrator user ID and password for the data store, either:
  - through the built-in procedure **ttCacheUidPwdSet**, or
  - with the utility **ttAdmin** -cacheUidPwdSet command
- Start the Cache agent for the data store, either:
  - through the built-in procedure **ttCacheStart**, or
  - with the utility **ttAdmin** -cacheStart command

For example:

```
ttAdmin -cacheUidPwdSet -cacheUid scott -cachePwd tiger DSN;
ttAdmin -cacheStart DSN;
```

ALTER REPLICATION

The ALTER REPLICATION statement adds, alters, or drops replication elements and changes the replication attributes of participating data stores.

Most ALTER REPLICATION operations are supported only when the replication agent is stopped (**ttAdmin** `-repStop`). However, it is possible to dynamically add a subscriber data store to a replication scheme while the replication agent is running. See Chapter 6, "Altering Replication" in the *TimesTen Replication Guide* for more information.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges.

SQL syntax

The ALTER REPLICATION statement has the syntax:

ALTER REPLICATION [*Owner.*]*ReplicationSchemeName*
ElementOperation [...] | *StoreOperation*

You can specify one or more of the *ElementOperations*:

ADD ELEMENT *ElementName*
{DATASTORE | {TABLE [*Owner.*]*TableName* [*CheckConflicts*]} |
SEQUENCE [Owner.]*SequenceName*}
{ MASTER | PROPAGATOR } *FullStoreName*
{ SUBSCRIBER *FullStoreName* [, …]
[*ReturnServiceAttribute*] } […] }
{ INCLUDE | EXCLUDE }{TABLE | CACHE GROUP | SEQUENCE }
[*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...]

ALTER ELEMENT { *ElementName* | * IN *FullStoreName*]
ADD SUBSCRIBER *FullStoreName* [, …] [*ReturnServiceAttribute*] |
ALTER SUBSCRIBER *FullStoreName* [, …] |
SET [*ReturnServiceAttribute*] |
DROP SUBSCRIBER *FullStoreName* [, …]

ALTER ELEMENT * IN *FullStoreName*
SET { MASTER | PROPAGATOR } *FullStoreName*

ALTER ELEMENT *ElementName*
{SET NAME *NewElementName* | SET *CheckConflicts*}

ALTER ELEMENT *ElementName*
{ INCLUDE | EXCLUDE }{TABLE | CACHE GROUP | SEQUENCE}
[*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...]

DROP ELEMENT { *ElementName* | * IN *FullStoreName* }

CheckConflicts can only be set when replicating TABLE elements. The syntax is described in "CHECK CONFLICTS" on page 240.

Syntax for *ReturnServiceAttribute* is:
{ RETURN RECEIPT [BY REQUEST] | NO RETURN }

The possible *StoreOperations* are:

ADD STORE *FullStoreName* [*StoreAttribute* [, …]]

ALTER STORE *FullStoreName* SET *StoreAttribute* [, …]

Syntax for the *StoreAttribute* is:
[DISABLE RETURN {SUBSCRIBER | ALL} *NumFailures* |
RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED |
DURABLE COMMIT {ON | OFF} |
RESUME RETURN *MilliSeconds* |
LOCAL COMMIT ACTION {NO ACTION| COMMIT} |
RETURN WAIT TIME *Seconds* |
COMPRESS TRAFFIC {ON | OFF} |
PORT *PortNumber* |
TIMEOUT *Seconds* |
FAILTHRESHOLD *Value*]

Parameters The ALTER REPLICATION statement has the parameters:

[*Owner.*]*ReplicationSchemeName*	Name assigned to the replication scheme.
ADD ELEMENT *ElementName*	Adds a new ELEMENT to the existing replication scheme. *ElementName* is an identifier of up to 30 characters. With DATASTORE elements, the *ElementName* must be unique with respect to other DATASTORE element names within the first 20 chars. If the ELEMENT is a DATASTORE, all tables and cache groups are included in the data store. SEQUENCE elements that are part of the data store do not have their return services modified by this statement.
ADD ELEMENT *ElementName* DATASTORE { INCLUDE | EXCLUDE } {TABLE | CACHE GROUP | SEQUENCE} [*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...]	Adds a new DATASTORE ELEMENT to the existing replication scheme. *ElementName* is an identifier of up to 30 characters. With DATASTORE elements, the *ElementName* must be unique with respect to other DATASTORE element names within the first 20 chars. INCLUDE includes in the data store only the tables and cache groups listed. Use one INCLUDE clause for each object type (table, cache group or sequence). EXCLUDE includes in the data store all tables and cache groups *except* the tables, cache groups and sequences listed. Use one EXCLUDE clause for each object type (table, cache group or sequence). [*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...] is the list of table names, cache group names or sequence names, including optional owners, to be included or excluded from the DATASTORE ELEMENT. If the element is a sequence, RETURN attributes are not applied, no conflict checking is supported and sequences that cycle return an error.
ADD SUBSCRIBER *FullStoreName*	Indicates an additional subscriber data store. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.

ALTER ELEMENT * IN *FullStoreName*	Makes a change to all elements for which *FullStoreName* is the MASTER or PROPAGATOR. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description. This syntax can be used on a set of element names to: • Add, alter, or drop subscribers. • Set the MASTER or PROPAGATOR status of the element set. SEQUENCE elements that are part of the data store being altered do not have their return services modified by this statement.
ALTER ELEMENT *ElementName*	Name of the element to which a subscriber is to be added or dropped.
ALTER ELEMENT *ElementName1* SET NAME *ElementName2*	Renames *ElementName1* with the name *ElementName2*. You can only rename elements of type TABLE.
ALTER ELEMENT *ElementName* DATASTORE { INCLUDE | EXCLUDE } {TABLE | CACHE GROUP | SEQUENCE } [*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...]	Adds or removes tables or cache groups in an existing data store element. INCLUDE adds to the data store the tables and cache groups listed. Use one INCLUDE clause for each object type (table or cache group). EXCLUDE removes from the data store the tables and cache groups listed. Use one EXCLUDE clause for each object type (table, cache group or sequence). [*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...] is the list of table names, cache group names or sequence names including optional owners, to be added to or removed from the DATASTORE ELEMENT. If the element is a sequence, RETURN attributes are not applied, no conflict checking is supported and sequences that cycle return an error.

ALTER SUBSCRIBER *FullStoreName* SET RETURN RECEIPT [BY REQUEST] | NO RETURN	Indicates an alteration to a subscriber data store to enable, disable, or change the return receipt service.*FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
CheckConflicts	Check for replication conflicts when simultaneously writing to bi-directionally replicating TABLE elements between data stores. You cannot check for conflicts when replicating elements of type DATASTORE. See "CHECK CONFLICTS" on page 240.
COMPRESS TRAFFIC {ON | OFF}	Compress replicated traffic to reduce the amount of network bandwidth. ON specifies that all replicated traffic for the data store defined by STORE be compressed. OFF (the default) specifies no compression. See "Compressing replicated traffic" in the *TimesTen to TimesTen Replication Guide* for details.
DISABLE RETURN {SUBSCRIBER | ALL} *NumFailures*	Set the return service failure policy so that return service blocking is disabled after the number of timeouts specified by *NumFailures*. Selecting SUBSCRIBER applies this policy only to the subscriber that fails to acknowledge replicated updates within the set timeout period; ALL applies this policy to all subscribers should any of the subscribers fail to respond. This failure policy can be specified for either the RETURN RECEIPT or RETURN TWOSAFE service. See "Managing return service timeout errors and replication state changes" in the *TimesTen to TimesTen Replication Guide* for details.
DURABLE COMMIT {ON | OFF}	Set to override the **DurableCommits** setting on a data store and enable durable commit when return service blocking has been disabled by DISABLE RETURN.

DROP ELEMENT * IN *FullStoreName*	Deletes the replication description of all elements for which *FullStoreName* is the MASTER. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
DROP ELEMENT *ElementName*	Deletes the replication description of *ElementName*.
DROP SUBSCRIBER *FullStoreName*	Indicates that updates should no longer be sent to the specified subscriber data store. This operation fails if your replication scheme has only one subscriber. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
FAILTHRESHOLD *Value*	For disk-based logging, *Value* is the number of log files that can accumulate for a subscriber data store. If this value is exceeded, the subscriber is set to the Failed state. The value 0 means “No Limit.” This is the default. See "Managing the log on a replicated data store" in the *TimesTen Replication Guide* for more information.

FullStoreName	The data store, specified as one of the following: • SELF • The prefix of the data store file name For example, if the data store path is `directory/subdirectory/data.ds0`, then `data` is the data store name that should be used. This is the data store file name specified in the **DataStore** attribute of the DSN description with optional host ID in the form: *DataStoreName* [ON *Host*] *Host* can be either an IP address or a literal host name assigned to one or more IP addresses, as described in "Configuring host IP addresses" in the *TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: “MyHost-500”.
LOCAL COMMIT ACTION {NO ACTION | COMMIT}	Specifies the default action to be taken for a RETURN TWOSAFE transaction in the event of a timeout. **NO ACTION**. On timeout, the commit function returns to the application, leaving the transaction in the same state it was in when it entered the commit call, with the exception that the application is not able to update any replicated tables. The application can only reissue the commit. The transaction may not be rolled back. This is the default. **COMMIT**. On timeout, the commit function writes a COMMIT log record and ends the transaction locally. No more operations are possible on the same transaction. This setting can be overridden for specific transactions by calling the **ttRepSyncSet** procedure with the *localAction* parameter.

MASTER *FullStoreName*	The data store on which applications update the specified ELEMENT. The MASTER data store sends updates to its SUBSCRIBER data stores. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
NO RETURN	Specifies that no return service is to be used. This is the default. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
PORT *PortNumber*	The TCP/IP port number on which the replication agent on this data store listens for connections. If not specified, the replication agent allocates a port number automatically. All TimesTen data stores that replicate to each other must use the same port number.
PROPAGATOR *FullStoreName*	The data store that receives replicated updates and passes them on to other data stores.
RESUME RETURN *MilliSeconds*	If return service blocking has been disabled by DISABLE RETURN, this attribute sets the policy on when to re-enable return service blocking. Return service blocking is re-enabled as soon as the failed subscriber acknowledges the replicated update in a period of time that is less than the specified *MilliSeconds*.
RETURN RECEIPT [BY REQUEST]	Enables the return receipt service, so that applications that commit a transaction to a master data store are blocked until the transaction is received by all subscribers. RETURN RECEIPT applies the service to all transactions. If you specify RETURN RECEIPT BY REQUEST, you can use the **ttRepSyncSet()** procedure to enable the return receipt service for selected transactions. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.

RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED	Set the return service failure policy so that return service blocking is either enabled or disabled when the replication agent is in the “stop” or “pause” state. OFF is the default when using the RETURN RECEIPT service. ON is the default when using the RETURN TWOSAFE service. See "Managing return service timeout errors and replication state changes" in the *TimesTen to TimesTen Replication Guide* for details.
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master data store are blocked until the transaction is committed on all subscribers. RETURN TWOSAFE applies the service to all transactions. If you specify RETURN TWOSAFE BY REQUEST, you can use the **ttRepSyncSet** procedure to enable the return receipt service for selected transactions. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgement. The default value is 10 seconds. A value of ‘0’ means there is no wait time. Your application can override this timeout setting by calling the **ttRepSyncSet** procedure with the *returnWait* parameter
SET { MASTER | PROPAGATOR } *FullStoreName*	Sets the given data store to be the MASTER or PROPAGATOR of the given element(s). The *FullStoreName* must the be data store’s file base name.

SUBSCRIBER *FullStoreName*	A data store that receives updates from the MASTER data store(s). *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
TIMEOUT *Seconds*	The amount of time a data store waits for a response from another data store before resending the message. Default: 120 seconds.

Description

- ALTER ELEMENT DROP SUBSCRIBER deletes a subscriber for a particular replication element.
- ALTER ELEMENT SET NAME may be used to change the name of a replication element when it conflicts with one already defined at another data store. SET NAME does not admit the use of * IN *FullStoreName*. The *FullStoreName* must the be data store's file base name. For example, if the data store file name is `data.ds0`, then `data` is the file base name.
- ALTER ELEMENT SET MASTER may be used to change the master data store for replication elements. The * IN *FullStoreName* option must be used for the MASTER operation. That is, a master data store must transfer ownership of all of its replication elements, thereby giving up its master role entirely. Typically, this option is used in ALTER REPLICATION statements requested at SUBSCRIBER data stores after the failure of a (common) MASTER.

To transfer ownership of the master elements to the subscriber:

- Manually drop the replicated elements by executing an ALTER REPLICATION DROP ELEMENT statement for each replicated table.
- Use ALTER REPLICATION ADD ELEMENT to add each table back to the replication scheme, with the newly designated MASTER / SUBSCRIBER roles.

ALTER REPLICATION ALTER ELEMENT SET MASTER does not automatically retain the old master as a subscriber in the scheme. If this is desired, execute an ALTER REPLICATION ALTER ELEMENT ADD SUBSCRIBER statement.

Note: There is no ALTER ELEMENT DROP MASTER. Each replication element must have exactly one MASTER data store, and the currently designated MASTER cannot be deleted from the replication scheme.

Example 5.6 This example sets up replication for an additional table WestLeads that is updated on data store West and replicated to data store East.

```
ALTER REPLICATION R1
   ADD ELEMENT E3 TABLE WestLeads
```

```
    MASTER West ON "WestCoast"
    SUBSCRIBER East ON "EastCoast";
```

Example 5.7 This example adds an additional subscriber (Backup) to table WestLeads.

```
ALTER REPLICATION R1
   ALTER ELEMENT E3
     ADD SUBSCRIBER Backup ON "BackupServer";
```

Example 5.8 This example changes the element name of table WestLeads from E3 to NewElementName.

```
ALTER REPLICATION R1
   ALTER ELEMENT E3
     SET NAME NewElementName;
```

Example 5.9 This example makes NewWest the master for all elements for which West currently is the master.

```
ALTER REPLICATION R1
   ALTER ELEMENT * IN West
     SET MASTER NewWest;
```

Example5.10 This element changes East's port number.

```
ALTER REPLICATION R1
   ALTER STORE East ON "EastCoast" SET PORT 22251;
```

Example5.11 This example adds `my.tab1` table to the ds1 data store element in `my.rep1` replication scheme.

```
ALTER REPLICATION my.rep1
  ALTER ELEMENT ds1 DATASTORE
     INCLUDE TABLE my.tab1;
```

Example5.12 This example add `my.cg1` cache group to `ds1` data store in `my.rep1` replication scheme.

```
ALTER REPLICATION my.rep1
  ALTER ELEMENT ds1 DATASTORE
     INCLUDE CACHE GROUP my.cg1;
```

Example5.13 This example adds `ds1` data store to `my.rep1` replication scheme. Include `my.tab2` table, `my.cg2` cache group, and `my.cg3` cache group in the data store.

```
ALTER REPLICATION my.rep1
  ADD ELEMENT ds1 DATASTORE
     MASTER rep2
```

```
    SUBSCRIBER rep1, rep3
    INCLUDE TABLE my.tab2
    INCLUDE CACHE GROUP my.cg2, my.cg3;
```

Example5.14 This example adds `ds2` data store to a replication scheme but exclude `my.tab1` table, `my.cg0` cache group and `my.cg1` cache group.

```
ALTER REPLICATION my.rep1
  ADD ELEMENT ds2 DATASTORE
    MASTER rep2
    SUBSCRIBER rep1
    EXCLUDE TABLE my.tab1
    EXCLUDE CACHE GROUP my.cg0, my.cg1;
```

To drop a table from a data store, see "Altering a replicated table" in *TimesTen to TimesTen Replication Guide*.

See Also

“ALTER ACTIVE STANDBY PAIR” on page 164
“CREATE ACTIVE STANDBY PAIR” on page 204
“CREATE REPLICATION” on page 233
“DROP ACTIVE STANDBY PAIR” on page 276
“DROP REPLICATION” on page 281

ALTER SESSION

The ALTER SESSION statement changes the NLS_SORT, NLS_LENGTH_SEMANTICS, and NLS_NCHAR_CONV_EXCP session parameters dynamically.

Access Control

There are no special privileges required for this operation.

SQL syntax

```
ALTER SESSION SET
      {NLS_SORT = {BINARY| SortName} |
       NLS_LENGTH_SEMANTICS = {BYTE| CHAR} |
       NLS_NCHAR_CONV_EXCP = {TRUE|FALSE}
      }
```

Parameters

The ALTER SESSION statement has the parameters:

NLS_SORT = {BINARY| *SortName}*	Indicates which collation sequence to use for linguistic comparisons. Append _CI or _AI to either BINARY or the *SortName* value if you wish to do case-insensitive or accent-insensitive sorting. If you do not specify NLS_SORT, the default is BINARY. See Example 5.15. For a complete list of supported values for *SortName*, see "Supported Linguistic Sorts" in the Operations Guide. For more information on case-insensitive or accent-insensitive sorting, see "Case-insensitive and accent-insensitive linguistic sorts".
NLS_LENGTH_SEMANTICS = {BYTE | CHAR}	Sets the default length semantics configuration. BYTE indicates byte length semantics; CHAR indicates character length semantics. The default is BYTE. For more information on length semantics, see "Length semantics and data storage".
NLS_NCHAR_CONV_EXCP = {TRUE | FALSE}	Determines whether an error should be reported when there is data loss during an implicit or explicit character type conversion between NCHAR/NVARCHAR2 data and CHAR/VARCHAR2 data. Specify TRUE to enable error reporting; Specify FALSE to not report errors. The default is FALSE.

Description

- The ALTER SESSION statement affects commands that are subsequently executed by the session. The new session parameters take effect immediately.

- The NLS_SORT setting affects materialized views and cache group maintenance. Use the NLSSORT() function rather than relying on the NLS_SORT setting.
- Character length and byte length semantics are supported to resolve potential ambiguity regarding column length and storage size. Multibyte encoding character sets are supported (For example, UTF-8 or AL32UTF8). Multibyte encodings require varying amounts of storage per character depending on the character. For example, an UTF-8 character may require from 1 to 4 bytes.

  If, for example, a column is defined as CHAR (10), you may assume that the 10 characters will fit in this column regardless of character set encoding. However, for UTF-8 character set encoding, up to 40 bytes would be required. TimesTen supports character length and byte length semantics to avoid such ambiguity.
- Operations involving character comparisons support linguistic sensitive collating sequences. Case-insensitive sorts may affect DISTINCT value interpretation. Supported collating sequence sensitive operations:
  - MIN,MAX
  - BETWEEN
  - =,!=, >, >=,<,<=
  - DISTINCT
  - CASE
  - GROUP BY
  - HAVING
  - ORDER BY
  - IN
  - LIKE
- Primary key indexes are based on the BINARY collating sequence. You cannot use non-BINARY NLS_SORT with equality searches on the primary key index.
- Implicit and explicit CHAR <-> NCHAR conversions are supported. CHAR <-> NCHAR conversions are not allowed when using the TIMESTEN8 character set.
- You can use the SQL string functions with the supported character sets. For example, UPPER and LOWER functions support non-ASCII CHAR/VARCHAR2 characters as well as NCHAR/NVARCHAR2 characters.
- TIMESTEN8 character set restrictions:
  - Character set conversions are not allowed.
  - BINARY is the only acceptable collating sequence.
  - CHAR semantics are ignored. Characters are assumed to be single-byte.

  - UPPER and LOWER functions support ASCII characters only. Results for non-ASCII characters are undefined; no error is returned.
- NLS_SORT settings other than BINARY could have a performance impact on character operations.
- Choice of character set could have an impact on memory consumption for CHAR/VARCHAR2 column data.
- The character sets of all data stores involved in a replication scheme must match.

Example 5.15 The following example uses the ALTER SESSION statement to change the NLS_SORT setting from BINARY to BINARY_CI to BINARY_AI. The database and connection character sets are `WE8ISO8859P1`.

```
Command> connect "dsn=cs;ConnectionCharacterSet=WE8ISO8859P1";
Connection successful: DSN=cs;UID=user;DataStore=/datastore/user/
cs;
DatabaseCharacterSet=WE8ISO8859P1;
ConnectionCharacterSet=WE8ISO8859P1;PermSize=32;TypeMode=0;
(Default setting AutoCommit=1)
Command>#Create the Table
Command> CREATE TABLE CollatingDemo (Letter VARCHAR2 (10));
Command>#Insert values
Command> INSERT INTO CollatingDemo VALUES ('a');
1 row inserted.
Command> INSERT INTO CollatingDemo VALUES ('A');
1 row inserted.
Command> INSERT INTO CollatingDemo VALUES ('Y');
1 row inserted.
Command> INSERT INTO CollatingDemo VALUES ('ä');
1 row inserted.
Command>#SELECT
Command> SELECT * FROM CollatingDemo;
< a >
< A >
< Y >
< ä >
4 rows found.
Command>#SELECT with ORDER BY
Command> SELECT * from CollatingDemo ORDER BY Letter;
< A >
< Y >
< a >
< ä >
4 rows found.
Command>#set NLS_SORT to BINARY_CI and SELECT
Command> ALTER SESSION SET NLS_SORT = BINARY_CI;
Command> SELECT * from CollatingDemo ORDER BY Letter;
< a >
```

```
< A >
< Y >
< ä >
4 rows found.
Command>#Set NLS_SORT to BINARY_AI and SELECT
Command> ALTER SESSION SET NLS_SORT = BINARY_AI;
Command> SELECT * from CollatingDemo ORDER BY Letter;
< ä >
< a >
< A >
< Y >
4 rows found.
```

ALTER TABLE

The ALTER TABLE statement changes an existing table definition.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

To add columns:

```
ALTER TABLE [Owner.]TableName
ADD [COLUMN] ColumnName ColumnDataType
    [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL]
```

or

```
ALTER TABLE [Owner.]TableName
ADD (ColumnName ColumnDataType
    [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL] [, ... ] )
```

To remove columns:

```
ALTER TABLE [Owner.]TableName
DROP [COLUMN] ColumnName
```

or

```
ALTER TABLE [Owner.]TableName DROP
(ColumnName [, ... ] )
```

To add a foreign key and optionally add ON DELETE CASCADE:

```
ALTER TABLE [Owner.]TableName
ADD [CONSTRAINT ForeignKeyName] FOREIGN KEY
  (ColumnName [,...]) REFERENCES RefTableName
    [(ColumnName [,...])] [ON DELETE CASCADE]
```

To remove a foreign key:

```
ALTER TABLE [Owner.]TableName
DROP CONSTRAINT ForeignKeyName
```

To resize a hash index:

```
ALTER TABLE [Owner.]TableName
SET PAGES = {RowPages | CURRENT}
```

To change the primary key to use a hash index:

```
ALTER TABLE [Owner.]TableName
USE HASH INDEX PAGES = {RowPages | CURRENT}
```

To change the primary key to use a T-tree index:

```
ALTER TABLE [Owner.]TableName
USE TREE INDEX
```

To change the default value of a column:

```
ALTER TABLE [Owner.]TableName
MODIFY (ColumnName DEFAULT DefaultVal)
```

To add or drop a UNIQUE constraint on a column:

```
ALTER TABLE Owner.]TableName
{ADD | DROP} UNIQUE (ColumnName)
```

To remove the default value of a column that is nullable, by changing it to NULL:

```
ALTER TABLE [Owner.]TableName
MODIFY (ColumnName DEFAULT NULL)
```

To add LRU aging:

```
ALTER TABLE [Owner.]TableName
ADD AGING LRU [ON | OFF]
```

To add time-based aging:

```
ALTER TABLE [Owner.]TableName
ADD AGING USE ColumnName LIFETIME num1
        {MINUTE[S] | HOUR[S] | DAY[S]}
        [CYCLE num2 {MINUTE[S] | HOUR[S] | DAY[S] }]
        [ON | OFF]
```

To change the aging state:

```
ALTER TABLE [Owner.]TableName
SET AGING  [ON | OFF]
```

To drop aging:

```
ALTER TABLE [Owner.]TableName
DROP AGING
```

For time-based aging, to change the LIFETIME:

```
ALTER TABLE [Owner.]TableName
SET AGING LIFETIME num1 {MINUTE[S] | HOUR[S] | DAY[S]}
```

For time-based aging, to change the CYCLE:

```
ALTER TABLE [Owner.]TableName
SET AGING CYCLE num2 {MINUTE[S] | HOUR[S] | DAY[S]}
```

Parameters The ALTER TABLE statement has the parameters:

[*Owner.*] *TableName*	Identifies the table to be altered.
UNIQUE	Specifies that in the column *ColumnName* each row must contain a unique value.

MODIFY	Specifies that an attribute of a given column is to be changed to a new value.
DEFAULT [*DefaultVal* | NULL]	Specifies that the column has a default value, *DefaultVal*. If NULL, specifies that the default value of the columns is to be dropped. If a column with a default value of SYSDATE is added, the value of the column of the existing rows only is the system date at the time the column was added. If the default value is one of the USER functions the column value is the user value of the session that executed the ALTER TABLE statement. Altering the default value of a column has no impact on existing rows.
ColumnName	Name of the column to for which the UNIQUE CONSTRAINT or default value is to be changed. A new column cannot have the same name as an existing column or another new column.
ColumnDataType	Type of the column to be added. Some types require additional parameters. See Chapter 1, "Data Types" for the data types that can be specified.
INLINE | NOT INLINE	By default, variable-length columns whose declared column length is > 128 bytes are stored out of line. Variable-length columns whose declared column length is <= 128 bytes are stored inline. The default behavior can be overridden during table creation through the use of the INLINE and NOT INLINE keywords.
CONSTRAINT	Specifies that a foreign key is to be dropped. Optionally specifies that an added foreign key is named by the user.
ForeignKeyName	Name of the foreign key to be added or dropped. All foreign keys are assigned a default name by the system if the name was not specified by the user. Either the user-provided name or system name can be specified in the DROP FOREIGN KEY command.
FOREIGN KEY	Specifies that a foreign key is to be added or dropped. See "FOREIGN KEY" on page 254.
REFERENCES	Specifies that the foreign key references another table.
RefTableName	The name of the table that the foreign key references.
[ON DELETE CASCADE]	Enables the ON DELETE CASCADE referential action. If specified, when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign key values are also deleted.

USE HASH INDEX PAGES = {*RowPages* | CURRENT}	Specifies that a hash index is to be used for the primary key. If the primary key already uses a hash index, then this clause is equivalent to the SET PAGES clause.
USE TREE INDEX	Specifies that a T-tree index is to be used for the primary key. If the primary key already uses a T-tree index, then this clause is ignored.
SET PAGES	Resizes the hash index based on the expected number of row pages in the table. Each row page can contain up to 256 rows of data. This number determines the number of hash buckets created for the hash index. The minimum is 1. If your estimate is too small, performance can be degraded. You can specify a constant (RowPages) or the CURRENT number of row pages. See “Column Definition” on page 257 for a description of hash indexes and pages.
RowPages	The number of row pages expected.
CURRENT	Use the number of row pages currently in use.
ADD AGING LRU [ON | OFF]	Adds LRU aging to an existing table that has no aging policy defined. With LRU aging, rows of the table are aged using the LRU policy. By removing the approximate least recently used data, LRU aging keeps only the recently used data in cache. Data is removed if the data store space in-use exceeds a specified threshold value. LRU aging frees up space occupied by infrequently used data. ON or OFF settings refer to the aging state. The OFF setting indicates the LRU aging policy is defined, but the aging state is disabled. Because the aging state is OFF, aging is not done automatically. The ON setting indicates the LRU aging policy is defined and the aging state is enabled. Because the aging state is ON, aging is done automatically. The default is ON. For more information on LRU aging, see "Aging for tables:". For detailed information on LRU aging, see "Implementing aging in your tables".

ADD AGING USE *ColumnName*... [ON | OFF]	Adds time-based aging to a table that has no aging policy defined. With time-based aging, rows of the table are aged using the time-based aging policy. Time-based aging allows you to specify the lifetime data is kept in cache. The time unit is expressed in terms of minutes, hours, or days. In order to use time-based aging, you must define a column in the table you are aging. The value of this column is subtracted from SYSDATE, truncated using the specified unit (minute, hour, day) and then compared to the LIFETIME value you specify. If the result is *greater than* the LIFETIME value, then the row is a candidate for aging. Define the column as NON-NULL and of type TIMESTAMP. The values of the timestamp columns are updated by your applications. If the value of this column is unknown for some rows, and you do not want the rows to be aged, define the column with a large default value. *ColumnName* refers to the timestamp column you defined on the table you are aging. The time-based aging policy is defined when you specify the AGING USE clause. ON or OFF settings refer to the aging state. The OFF setting indicates the time-based aging policy is defined, but the aging state is disabled. Because the aging state is OFF, aging is not done automatically. The ON setting indicates the time-based aging policy is defined and the aging state is enabled. Because the aging state is ON, aging is done automatically. The default is ON. Time-based aging allows you to implement sliding windows for your application. For more information on time-based aging, see "Aging for tables:". For detailed information on time-based aging, see "Implementing aging in your tables".

LIFETIME *Num1* {MINUTE[S] |HOUR[S]| DAY[S]}	This clause is specific to time-based aging. LIFETIME is the minimum amount of time data is kept in cache. For example, keep data in cache for *Num1* unit of time where unit can be minutes or hours or days. This means rows that exceed the specified LIFETIME value are aged out (deleted from the table). *Num1* must be a positive integer constant; Its value indicates the amount of time in minutes, hours, or days that the rows should stay in cache. The concept of time resolution is supported. The time-unit specified for the lifetime of data is the resolution granularity for which data should be kept in cache. If, for example, DAYS is specified as the time resolution, then all rows whose timestamp belongs to the same day are aged out at the same time. If HOURS is specified as the time resolution, then all rows with timestamp values within that hour are aged at the same time. Thus, a LIFETIME of *3* DAYS is different than a LIFETIME of *72* HOURS (3*24) or a LIFETIME of *432* MINUTES (3*24*60). Days represent all days of the week including weekends and holidays. See Example 5.24 Specify the LIFETIME after the ADD AGING USE clause if you are adding time-based aging to your table. Specify the LIFETIME clause after the SET AGING clause to change the LIFETIME setting.
CYCLE *Num2* {MINUTE[S] | HOUR[S] |DAY[S] }	This clause is specific to time-based aging. It indicates how often the system should examine rows to see if data exceeds the specified LIFETIME values and should be aged out (deleted). *Num2* must be a positive integer constant. If you specify 0 for *Num2*, then aging is continuous and the aging thread never sleeps. If you are adding time-based aging to your table, then this clause is optional. If you do not specify the CYCLE clause, then the default value is 5 minutes. This means that if aging is enabled, rows are examined every 5 minutes starting as soon as possible. To change the CYCLE, use ALTER TABLE... SET AGING CYCLE...
SET AGING [ON | OFF]	Changes the aging state. The aging policy must be previously defined. If you change the aging state to ON, then aging is done automatically. If you change the aging state to OFF, then aging is not done automatically. If you wish to control aging via an external scheduler, then disable aging and invoke ttAgingScheduleNow.

DROP AGING	Once you define an aging policy, you cannot alter it. Drop aging, then redefine.
SET AGING LIFETIME *Num1* {MINUTE[S] | HOUR[S] | DAY[S] }	Use this clause to change the LIFETIME for time-based aging.
SET AGING CYCLE *Num2* {MINUTE[S] | HOUR[S] |DAY[S]}	Use this clause to change the CYCLE for time-based aging.

Description

- The ALTER TABLE statement cannot be used to alter a temporary table.
- The ALTER TABLE ADD statement adds one or more new columns to an existing table. The new columns are added to the end of all existing rows of the table in one new partition.
- Columns referenced by materialized views cannot be dropped.
- Only one partition is added to the table per statement regardless of the number of columns added.
- The new columns cannot be declared NOT NULL.
- NULL is the initial value for all added columns, unless a default value is specified for the new column.
- The total number of columns in the table cannot exceed 255. In addition, the total number of partitions in a table cannot exceed 255, one of which is used by TimesTen.
- The PRIMARY KEY clause cannot be used when adding new columns; you cannot use ALTER TABLE to change the primary key of a table.
- As the result of an ALTER TABLE ADD statement, an additional read occurs for each new partition during queries. Therefore, altered tables may have slightly degraded performance. The performance can only by restored by dropping and recreating the table, or by using the `ttMigrate create -c -noRepUpgrade` command, and restoring the table using the `ttRestore -r -noRepUpgrade` command. Dropping the added column does not recover the lost performance or decrease the number of partitions.
- The ALTER TABLE DROP statement removes one or more columns from an existing table. The dropped columns are removed from all current rows of the table. Subsequent SQL statements must not attempt to make any use of the dropped columns; they are treated as if they never existed. Columns that are in the table's primary key cannot be dropped. Columns that are in any of the table's foreign keys cannot be dropped, until all foreign keys have been

dropped. Columns that are indexed cannot be dropped until all indexes on the column have been dropped. ALTER TABLE cannot be used to drop all of the columns of a table; use DROP TABLE instead.

- When a column is dropped from a table, all commands referencing that table need to be recompiled. An error may result at recompilation time if a dropped column was referenced. The application must re-prepare those commands, and rebuild any parameters and result columns. When a column is added to a table, the commands that contain a SELECT * statement are invalidated. Only these commands must be re-prepared. All other commands continue to work as expected.
- When you drop a column, the column space is not freed.
- When you add a UNIQUE constraint, there is overhead incurred (in terms of additional space and additional time). This is because an index is created to maintain the UNIQUE constraint. You cannot use the DROP INDEX statement to drop an index used to maintain the UNIQUE constraint.
- A UNIQUE constraint and its associated index cannot be dropped if it is being used as a unique index on a replicated table.
- Use ALTER TABLE...USE TREE INDEX if your application performs range queries over a table's primary key.
- Use ALTER TABLE...USE HASH INDEX if your application performs exact match lookups on a table's primary key.
- An error is generated if a table has no primary key and either the USE HASH INDEX clause or the USE TREE INDEX clause is specified.
- To change the ON DELETE CASCADE triggered action, drop then redefine the foreign key constraint.
- If access control is disabled, no privilege is necessary to execute a DELETE statement which triggers the ON DELETE CASCADE action. Otherwise, WRITE privileges are required on all tables affected by the ON DELETE CASCADE action.
- ON DELETE CASCADE is supported on "detail tables" of a materialized view. If you have a materialized view defined over a child table, a deletion from the parent table will cause cascaded deletes in the child table. This, in turn, will trigger changes in the materialized view.
- The total number of rows reported by the DELETE statement does not include rows deleted from child tables as a result of the ON DELETE CASCADE action.
- For ON DELETE CASCADE: Since different paths may lead from a parent table to a child table, the following rule is enforced:
  - Either all paths from a parent table to a child table are 'delete' paths or all paths from a parent table to a child table are 'do not delete' paths.
  - Specify ON DELETE CASCADE on all child tables on the 'delete' path.

  - This rule does not apply to paths from one parent to different children or from different parents to the same child.
- For ON DELETE CASCADE: A second rule is also enforced:
  - If a table is reached by a 'delete' path, then all its children are also reached by a 'delete' path.
- For ON DELETE CASCADE with replication, the following restrictions apply:
  - The foreign keys specified with ON DELETE CASCADE must match between the Master and subscriber for replicated tables. Checking will be done at runtime. If there is an error, the receiver thread will stop working.
  - All tables in the delete cascade tree have to be replicated if any table in the tree is replicated. This restriction will be checked when the replication scheme is created or when a foreign key with ON DELETE CASCADE is added to one of the replication tables. If an error is found, the operation is aborted. You may be required to drop the replication scheme first before trying to change the foreign key constraint.
  - You must stop the replication agent before adding or dropping a foreign key on a replicated table.
- The ALTER TABLE ADD/DROP CONSTRAINT statement has the following restrictions:
  - When a foreign key is dropped, TimesTen also drops the index associated with the foreign key. Attempting to drop an index associated with a foreign key using the regular DROP INDEX statement results in an error.
  - Foreign keys cannot be added or dropped on tables in a cache group.
  - Foreign keys cannot be added or dropped on tables that participate in TimesTen replication. If the operation is attempted on a table that is either being replicated or is a replicated table, TimesTen returns an error.
  - Foreign keys cannot be added or dropped on views or temporary tables.

Aging for tables:

- Aging: Usage
  - When you define an aging policy for a table, you define:
    1. The type of aging: LRU aging or time-based.
    2. The state of aging either ON (enabled) or OFF (disabled).
    3. For time-based aging, the timestamp column, LIFETIME, and CYCLE.

Note: LRU attributes are defined by calling procedure ttAgingLruConfig. LRU attributes are not defined at the SQL level. If you do not call this procedure, then the default values are used.

– Define an aging policy on either an existing table or a new table. For an existing table, use the ALTER TABLE... ADD AGING clause. For a new table, use the CREATE TABLE... AGING clause.

To add LRU aging on an existing table:

ALTER TABLE... ADD AGING LRU [ON|OFF]

See Example 5.25.

To add time-based aging on an exising table:

ALTER TABLE... ADD AGING USE ...

See Example 5.26.

To add LRU aging on a new table:

CREATE TABLE... AGING LRU [ON|OFF]

To add time-based aging on a new table:

CREATE TABLE... AGING USE....

For more information on defining aging on a new table,

See **CREATE TABLE**.

– After you have defined the aging policy for the table, you cannot alter the aging policy. This means you cannot change the policy from LRU to time-based or from time-based to LRU. For time-based aging, you cannot change the timestamp column. Essentially, you cannot specify the ALTER TABLE...ADD AGING clause on a table with an existing aging policy.

– To change the aging policy, drop aging then redefine the aging policy:

ALTER TABLE... DROP AGING

See Example 5.27.

– After you have defined the aging policy for the table, you can change the aging state; that is you can change the state defined in the aging policy from ON (automatic aging enabled) to OFF (automatic aging disabled) or from OFF (automatic aging disabled) to ON (automatic aging enabled). To do this:

ALTER TABLE... SET AGING [ON | OFF]

See Example 5.28.

– For time-based aging, you can change the LIFETIME and CYCLE:

ALTER TABLE...SET AGING LIFETIME *Num1*

{MINUTE[S] |HOUR[S] |DAY[S] }

ALTER TABLE... SET AGING CYCLE *Num2*

{MINUTE[S] |HOUR[S] |DAY[S] }

• Aging: Built-in Procedures

– Use ttAgingLRUConfig to specify the following properties for LRU aging:

1. LowUsageThreshold: (percentage of Perm-size allocated)

2. HighUsageThreshold: (percentage of Perm-size allocated)

3. Aging Cycle: The number of minutes the aging thread sleeps before waking up.

- The default property settings for LRU aging are:

  1. LowUsageThreshold: 80%
  2. HighUsageThreshold: 90%
  3. Aging Cycle: 1 minute

- For LRU aging, the aging thread wakes up at the specified aging cycle and checks if the space usage is over the HighUsageThreshold; If it is not, it goes back to sleep; Otherwise, it begins removing the least used data until either the space usage is at or below the LowUsageThreshold percentage or there is no data left to remove.
- ttAgingLRUConfig applies to *all* tables with LRU aging. Arguments to this procedure can be omitted; If omitted, the previous setting is retained. Arguments are positive integers.
- For more information on the ttAgingLRUConfig procedure, see "ttAgingLRUConfig".
- Use ttAgingScheduleNow to schedule the aging process as soon as possible (the process starts when the subdaemon awakes). When you call this procedure, the aging process is started for a table regardless if its aging setting is OFF (disabled) or ON (enabled). The aging process is started only once; the previous aging state is unchanged. For example, if aging state is OFF when you call ttAgingScheduleNow, the aging process starts. When aging is complete, if your aging state is OFF, aging does not continue. To continue aging, you must call ttAgingScheduleNow again or change the aging state to ON.
- If you call ttAgingScheduleNow and the aging setting is ON, the aging cycle is reset based on the time ttAgingScheduleNow was called.
- ttAgingScheduleNow is useful if you wish to schedule the aging process as soon as possible or if you wish to start the aging process at a specified time (via an external scheduler e.g. cron job). In the latter case, set the aging state to OFF, and then call ttAgingScheduleNow to invoke aging.
- To schedule aging on one table, call ttAgingScheduleNow specifying the name of the table and enclosing the table name in quotes. The specified table can be defined with either LRU aging or time-based aging:

```
CALL ttAgingScheduleNow  ('TableName')
```

- To schedule aging on all tables, including tables defined with both *LRU* aging and *time-based* aging, call ttAgingScheduleNow without any arguments.

```
CALL ttAgingScheduleNow ( );
```

  - For more information on the ttAgingScheduleNow procedure, see "ttAgingScheduleNow".
- Aging: Specifics
  - LRU and time-based aging can be combined in one system. If you use only LRU aging, the aging thread wakes up based on the cycle specified for the whole data store; If you use only time-based aging, the aging thread wakes up based on an optimal frequency. This frequency is determined by the values specified in the CYCLE clause for all tables. If you use both LRU and time-based aging, then the thread wakes up based on a combined consideration of both types.
  - You can implement sliding windows with time-based aging. The table contains the last *Num1* time unit of data. Sliding OUT is handled automatically by aging. Data is brought into cache, and past data that falls out of the window is aged out. As the window changes, data is not guaranteed to be deleted; data is deleted when the aging thread wakes up.
- Aging: For LRU aging, the following rules determine if a row is accessed or referenced:
  - Any rows used to build the result set of a SELECT statement.
  - Any rows used to build the result set of an INSERT SELECT statement.
  - Any rows that are about to be updated or deleted.
- Aging: LRU and compiled commands:
  - Compiled commands when you either drop LRU aging from or add LRU aging to tables that are referenced in the commands.
- Aging: Foreign Key Issues
  - Tables that are related via foreign keys must have the same aging policy.
  - For LRU aging, if a child row is not a candidate for aging, then this child row is not deleted; Nor will the parent row be deleted; ON DELETE CASCADE settings are ignored.
  - For time-based aging, if a parent row is a candidate for aging, then all child rows are deleted. ON DELETE CASCADE (whether specified or not) is ignored.
- Aging: Restrictions
  - LRU aging and time-based aging are not supported on detail tables of materialized views.
  - LRU aging and time-based aging are not supported on global temporary tables.
  - You cannot drop the timestamp column that is used for time-based aging.
  - You cannot add or modify a column and then use that column as a timestamp column for time-based aging. This is because you cannot add or

modify a column and define it to be NOT NULL; The timestamp column used for time-based aging must be NOT NULL. See Example 5.29.

Example5.16 Column `ReturnRate` is added to table `Parts`.

```
ALTER TABLE Parts ADD COLUMN ReturnRate DOUBLE;
```

Example5.17 Columns `NumAssign` and `PrevDept` are added to table `Contractor`.

```
ALTER TABLE Contractor
ADD ( NumAssign INTEGER, PrevDept CHAR(30) );
```

Example5.18 Columns `Addr1` and `Addr2` are removed from table `Employee`.

```
ALTER TABLE Employee DROP ( Addr1, Addr2 );
```

Example5.19 The UNIQUE title column of the Books table is dropped in the following statement:

```
ALTER TABLE Books DROP UNIQUE (Title);
```

Example5.20 The following statement adds the column x1 to table T1 with a the default value of 5:

```
ALTER TABLE T1 ADD (x1 INT DEFAULT 5);
```

Example5.21 The following statement changes the default value of column x1 to 2:

```
ALTER TABLE T1 MODIFY (x1 DEFAULT 2);
```

Example5.22 The example illustrates the use of T-tree and Hash indices. Table Pkey is created; Col1 is defined as the primary key. By default, a t-tree index is created. The table is then ALTERed changing the index on Col1 to a hash index. The table is ALTERed again changing the index back to a t-tree index.

```
Command> CREATE TABLE Pkey (Col1 TT_INTEGER PRIMARY KEY,
Col2 VARCHAR2 (20));
Command> INDEXES Pkey;

Indexes on table SAMPLEUSER.PKEY:
  PKEY: unique T-tree index on columns:
    COL1
  1 index found.

1 table found.
```

Now ALTER table Pkey to use a HASH INDEX:

```
Command> ALTER TABLE Pkey USE HASH INDEX PAGES = CURRENT;
Command> INDEXES PKEY;
```

```
Indexes on table SAMPLEUSER.PKEY:
  PKEY: unique hash index on columns:
    COL1
  1 index found.

1 table found.
```

Now ALTER table Pkey to use a T-tree index:

```
Command> ALTER TABLE PKEY USE TREE INDEX;
Command> INDEXES PKEY;

Indexes on table SAMPLEUSER.PKEY:
  PKEY: unique T-tree index on columns:
    COL1
  1 index found.

1 table found.
```

Example 5.23 The example generates an error when attempting to ALTER a table to define either a T-tree or Hash index on a column without a primary key.

```
Command> CREATE TABLE IllegalIndex (Col1 CHAR (20));
Command> ALTER TABLE IllegalIndex USE TREE INDEX;
 2810: The table has no primary key so cannot change its index
type
The command failed.
```

```
Command> ALTER TABLE IllegalIndex USE HASH INDEX PAGES = CURRENT;
 2810: The table has no primary key so cannot change its index
type
The command failed.
```

Example 5.24 The following is an example of how time resolution works with aging.

If Lifetime is 3 days (resolution is in days):

```
If (SYSDATE - ColumnValue) <= 3, don't age
If (SYSDATE - ColumnValue) > 3, then candidate to age
```

```
If (SYSDATE - ColumnValue) = 3 days, 22 hours:

Row will not be aged out if you specified a lifetime of
3 days.  Row will be aged out if you specified a lifetime
of 72 hours.
```

Example 5.25 The following example alters a table adding LRU aging. The table has no previous aging policy. The aging state is ON by default.

```
ALTER TABLE AgingDemo3
      ADD AGING LRU;

Command> DESCRIBE AgingDemo3;
Table USER.AGINGDEMO3:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
  Aging lru on

1 table found.
(primary key columns are indicated with *)
```

Example 5.26 The following example alters a table adding time-based aging. The table has no previous aging policy. AgingColumn is the timestamp column; LIFETIME is 2 days; CYCLE is 30 minutes.

```
ALTER TABLE AgingDemo4
      ADD AGING USE AgingColumn LIFETIME 2 DAYS CYCLE 30
      MINUTES;

Command> DESCRIBE AgingDemo4;
Table USER.AGINGDEMO4:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
    AGINGCOLUMN                      TIMESTAMP (6) NOT NULL
  Aging use AGINGCOLUMN lifetime 2 days cycle 30 minutes on
```

Example 5.27 The following example generates an error message. It illustrates that once you create an aging policy, you cannot change it. You must drop aging and redefine.

```
CREATE TABLE AgingDemo5
      (AgingId NUMBER NOT NULL PRIMARY KEY
       ,Name   VARCHAR2 (20)
       ,AgingColumn TIMESTAMP NOT NULL
      )
      AGING USE AgingColumn LIFETIME 3 DAYS OFF;

ALTER TABLE AgingDemo5
      ADD AGING LRU;

 2980: Cannot add aging policy to a table with an existing aging policy.
Have to drop the old aging first
The command failed.

DROP aging on the table and redefine with LRU aging.

ALTER TABLE AgingDemo5
      DROP AGING;

ALTER TABLE AgingDemo5
      ADD AGING LRU;
```

```
Command> DESCRIBE AgingDemo5;

Table USER.AGINGDEMO5:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
    AGINGCOLUMN                      TIMESTAMP (6) NOT NULL
  Aging lru on

1 table found.
(primary key columns are indicated with *)
```

Example 5.28 The following example ALTERS a table by setting the aging state to OFF. The table has been defined with a time-based aging policy. If you set the aging state to OFF, aging is not done automatically. This is useful if you wish to use an external scheduler to control the aging process. Simply set aging state to OFF and then invoke ttAgingScheduleNow to start the aging process.

```
Command> DESCRIBE AgingDemo4;
Table USER.AGINGDEMO4:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
    AGINGCOLUMN                      TIMESTAMP (6) NOT NULL
  Aging use AGINGCOLUMN lifetime 2 days cycle 30 minutes on
```

```
ALTER TABLE AgingDemo4
      SET AGING OFF;
```

Note that when you DESCRIBE AgingDemo4, the aging policy is defined and the aging state is set to OFF.

```
Command> DESCRIBE AGINGDEMO4;

Table USER.AGINGDEMO4:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
    AGINGCOLUMN                      TIMESTAMP (6) NOT NULL
  Aging use AGINGCOLUMN lifetime 2 days cycle 30 minutes off

1 table found.
(primary key columns are indicated with *)
```

Now call ttAgingScheduleNow to invoke aging via an external scheduler:

```
Command> call ttAgingScheduleNow ('AgingDemo4');
```

Example 5.29 Attempt to alter a table adding a timestamp column and then use that column for time-based aging. Error is generated:

```
Command> describe x;

Table USER1.X:
  Columns:
   *ID                                TT_INTEGER NOT NULL

1 table found.
(primary key columns are indicated with *)
Command> ALTER TABLE x ADD COLUMN t TIMESTAMP;
Command> ALTER TABLE x ADD AGING USE t LIFETIME 2 DAYS;
 2993: Aging column cannot be nullable
The command failed.
```

See Also “CREATE TABLE” on page 251
“DROP TABLE” on page 282

ALTER USER

The ALTER USER statement allows an internal user or TimesTen administrator to change the user password.

Access Control

If Access Control is enabled for your TimesTen instance, this statement allows a user or the instance administrator to change the user's password. If your TimesTen instance does not use Access Control, this operation is not available. If Access Control is not enabled, TimesTen returns an error when this statement is called.

SQL syntax

ALTER USER *User* IDENTIFIED BY '*Password*'

Parameters

The ALTER USER statement has the parameters:

User	Name of the user whose password is being changed.
IDENTIFIED BY	Specifies how the TimesTen instance uniquely identifies the user.
Password	Internal users must be identified to TimesTen by a password. To perform data store operations using an internal user name, the user must supply this password.

Description

- Instance users may have internal user names or external user names.
  - Internal user names are defined strictly within a TimesTen instance.
  - External user names are defined by some external authority, such as the operating system. External user names cannot be assigned a TimesTen password.
- An internal user connected as *User* may execute this command to change their own TimesTen password. Passwords are case-sensitive.
- TimesTen instance users are user names that have been identified to the instance. User names are case-insensitive and apply to all data stores in the instance.
- An internal user name takes precedence over an external user of the same name.
- Changes to user identification and privileges take place at the next connection time.

Example5.30 To change the password for internal user TERRY to "12345" from its current setting, use:

```
ALTER USER terry IDENTIFIED BY '12345';
```

See Also

"CREATE USER" on page 270
"GRANT" on page 287
"REVOKE" on page 300

CREATE ACTIVE STANDBY PAIR

This statement creates an active standby pair. It includes an active master data store, a standby master data store, and may also include one or more read-only subscribers. The active master data store replicates updates to the standby master data store, which propagates the updates to the subscribers.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges

SQL Syntax

CREATE ACTIVE STANDBY PAIR
FullStoreName, *FullStoreName* [*ReturnServiceAttribute*]
[SUBSCRIBER *FullStoreName* [,...]]
[STORE *FullStoreName* [*StoreAttributes* [,...]]]
[{ INCLUDE | EXCLUDE }{TABLE | SEQUENCE | CACHE GROUP }
[*Owner.*]{*TableName* | *SequenceName* | *CacheGroupName*} [,...]]

The syntax for *ReturnServiceAttribute* is
{ RETURN RECEIPT [BY REQUEST] |
RETURN TWOSAFE [BY REQUEST] |
NO RETURN }

Syntax for *StoreAttributes* is:
[DISABLE RETURN {SUBSCRIBER | ALL} *NumFailures*]
[RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED]
[DURABLE COMMIT {ON | OFF}]
[RESUME RETURN *MilliSeconds*]
[LOCAL COMMIT ACTION {NO ACTION | COMMIT}]
[RETURN WAIT TIME *Seconds*]
[COMPRESS TRAFFIC {ON | OFF}
[PORT *PortNumber*]
[TIMEOUT *Seconds*]
[FAILTHRESHOLD *Value*]

Parameters CREATE ACTIVE STANDBY has the parameters:

FullStoreName	The data store, specified as one of the following: • SELF • The prefix of the data store file name For example, if the data store path is `directory/subdirectory/data.ds0`, then `data` is the data store name that should be used. This is the data store file name specified in the **DataStore** attribute of the DSN description with optional host ID in the form: *DataStoreName* [ON *Host*] *Host* can be either an IP address or a literal host name assigned to one or more IP addresses, as described in "Configuring host IP addresses" in the *TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: “MyHost-500”.
RETURN RECEIPT [BY REQUEST]	Enables the return receipt service, so that applications that commit a transaction to a master data store are blocked until the transaction is received by all subscribers. RETURN RECEIPT applies the service to all transactions. If you specify RETURN REQUEST BY REQUEST, you can use the **ttRepSyncSet()** procedure to enable the return receipt service for selected transactions. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.

RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master data store are blocked until the transaction is committed on all subscribers. RETURN TWOSAFE applies the service to all transactions. If you specify RETURN TWOSAFE BY REQUEST, you can use the **ttRepSyncSet** procedure to enable the return receipt service for selected transactions. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
NO RETURN	Specifies that no return service is to be used. This is the default. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
SUBSCRIBER *FullStoreName* [,...]]	A data store that receives updates from a master data store. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.
STORE *FullStoreName* [*StoreAttributes* [,...]]	Defines the attributes for the specified data store. Data store attributes include PORT, TIMEOUT and FAILTHRESHOLD. *FullStoreName* is the data store file name specified in the **DataStore** attribute of the DSN description.

{ INCLUDE | EXCLUDE } {TABLE | CACHE GROUP } [*Owner.*]{*TableName* | *SequenceName* | *CacheGroupName*} [,...]	Includes in or excludes from replication the tables, sequences or cache groups listed. INCLUDE adds the tables, sequences or cache groups to replication. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE removes the tables, sequences or cache groups from replication. Use one EXCLUDE clause for each object type (table, sequence or cache group). [*Owner.*]{*TableName* | *SequenceName* | *CacheGroupName*} [,...] is the list of table, sequence or cache group names, including optional owners, to be added to or removed from the data stores.
DISABLE RETURN {SUBSCRIBER | ALL} *NumFailures*	Set the return service failure policy so that return service blocking is disabled after the number of timeouts specified by *NumFailures*. Selecting SUBSCRIBER applies this policy only to the subscriber that fails to acknowledge replicated updates within the set timeout period; ALL applies this policy to all subscribers should any of the subscribers fail to respond. This failure policy can be specified for either the RETURN RECEIPT or RETURN TWOSAFE service. See "Managing return service timeout errors and replication state changes" in the *TimesTen to TimesTen Replication Guide* for details.
DURABLE COMMIT {ON | OFF}	Set to override the **DurableCommits** setting on a data store and enable durable commit when return service blocking has been disabled by DISABLE RETURN.

FAILTHRESHOLD *Value*	For disk-based logging, *Value* is the number of log files that can accumulate for a subscriber data store. If this value is exceeded, the subscriber is set to the Failed state. The value 0 means “No Limit.” This is the default. See "Managing the log on a replicated data store" in the *TimesTen to TimesTen Replication Guide* for more information.
LOCAL COMMIT ACTION {NO ACTION | COMMIT}	Specifies the default action to be taken for a return twosafe transaction in the event of a timeout. **Note:** This attribute is only valid when the RETURN TWOSAFE or RETURN TWOSAFE BY REQUEST attribute is set in the SUBSCRIBER clause. **NO ACTION** -- On timeout, the commit function returns to the application, leaving the transaction in the same state it was in when it entered the commit call, with the exception that the application is not able to update any replicated tables. The application can reissue the commit or rollback the call. **COMMIT** -- On timeout, the commit function writes a COMMIT log record and effectively ends the transaction locally. No more operations are possible on the same transaction. This default setting can be overridden for specific transactions by calling the *localAction* parameter in the **ttRepSyncSet()** procedure.

MASTER *FullStoreName*	The data store on which applications update the specified ELEMENT. The MASTER data store sends updates to its SUBSCRIBER data stores. The *FullStoreName* must be the data store specified in the **DataStore** attribute of the DSN description.
NO RETURN	Specifies that no return service is to be used. This is the default. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
PORT *PortNumber*	The TCP/IP port number on which the replication agent for the data store listens for connections. If not specified, the replication agent automatically allocates a port number.
PROPAGATOR *FullStoreName*	The data store that receives replicated updates and passes them on to other data stores. The *FullStoreName* must be the data store specified in the **DataStore** attribute of the DSN description.

Description

- CREATE ACTIVE STANDBY PAIR is immediately followed by the names of the two master data stores. The master data stores are later designated as ACTIVE and STANDBY using the **ttRepStateSet** procedure. See "Setting up an active standby pair" in Chapter 7 of *TimesTen to TimesTen Replication Guide*.
- The SUBSCRIBER clause lists one or more read-only subscriber data stores. You can designate up to 62 subscriber data stores.
- Replication between the active master data store and the standby master data store can be RETURN TWOSAFE, RETURN RECEIPT, or asynchronous. RETURN TWOSAFE ensures no transaction loss.
- Use the INCLUDE and EXCLUDE clauses to exclude the listed tables, sequences and cache groups from replication, or to include only the listed tables, sequences and cache groups, excluding all others.

Example 5.31 This example creates an active standby pair whose master data stores are `rep1` and `rep2`. There is one subscriber, `rep3`. The type of replication is RETURN RECEIPT. The statement also sets PORT and TIMEOUT attributes for the master data stores.

```
CREATE ACTIVE STANDBY PAIR rep1, rep2 RETURN RECEIPT
  SUBSCRIBER rep3
  STORE rep1 PORT 21000 TIMEOUT 30
  STORE rep2 PORT 22000 TIMEOUT 30;
```

See Also “ALTER ACTIVE STANDBY PAIR” on page 164
“DROP ACTIVE STANDBY PAIR” on page 276

CREATE CACHE GROUP

The CREATE CACHE GROUP statement:

- Creates the table defined by the cache group.
- Inserts all new information associated with the cache group in the appropriate system tables.
- Prevents illegal definitions (e.g., overlapping cache groups).

A *cache group* is a set of cached tables related through foreign keys. There is one root table, which does not reference any of the other tables. All other *cache tables* in the cache group reference exactly one other table in the cache group. In other words, the foreign key relationships form a tree.

A *cache table* is a set of rows satisfying the conditions:

- The rows constitute a subset of the rows of a vertical partition of an Oracle table.
- The rows are stored in a TimesTen table with the same name as the Oracle table.

In other words, if you select some of the columns of an Oracle table, and then take a subset of those rows, the resulting set of rows constitutes a table.

If a data store has more than one cache group, the cache groups must correspond to different Oracle (and TimesTen) tables.

cache group instance refers to a row in the root table and all the child table rows related directly or indirectly to the root table rows.

User and system managed cache groups

A cache group can be either system managed or user managed.

- A *system managed cache group* is fully managed by TimesTen and has fixed properties.

  The READONLY, ASYNCHRONOUS WRITETHROUGH, and SYNCHRONOUS WRITETHROUGH cache groups are system managed cache groups.

  – READONLY cache groups are updated in Oracle, and the updates are propagated from Oracle to the cache.

  – ASYNCHRONOUS cache groups are updated in the cache and the updates are propagated to Oracle. Transactions continue executing on the cache without waiting for a commit on Oracle.

  – WRITETHROUGH cache groups are updated in the cache and the updates are propagated to Oracle. Transactions are committed on the cache after notification that a commit has occurred on Oracle.

Because TimesTen manages system managed cache groups, including loading and unloading the cache group, certain statements and clauses cannot be used with these cache groups, including:

  – WHERE clauses (The clause can be used in a READONLY CACHE GROUP definition.)
  – READONLY, PROPAGATE and NOT PROPAGATE in the table definition

  In addition, AUTOREFRESH, REFRESH and FLUSH are not allowed on WRITETHROUGH cache groups. For READONLY cache groups, AUTOREFRESH is on by default.

- A *user managed cache group* must be managed by the application or user. Certain statements and clauses cannot be used with these cache groups, including:

  The table-level READONLY keyword can only be used for user managed cache groups.

In addition, both TimesTen and Oracle must be able to parse all WHERE clauses.

ASYNCHRONOUS WRITETHROUGH cache groups cannot be created while the replication agent is running.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

There are CREATE CACHE GROUP statements to create each type of cache group:

- CREATE READONLY CACHE GROUP
- CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP
- CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP
- CREATE USERMANAGED CACHE GROUP

The statements used to create each type of cache group are described in separate sections.

CREATE READONLY CACHE GROUP

For READONLY cache groups, the syntax is:

```
CREATE READONLY CACHE GROUP [Owner.]GroupName
  [AUTOREFRESH
    [MODE {INCREMENTAL | FULL}]
    [INTERVAL IntervalValue {MINUTE[S] | SECOND[S] |
          MILLESECOND[S] }]
    [STATE {ON|OFF|PAUSED}]
```

```
 ]
 FROM
  {[Owner.]TableName (
     {ColumnDefinition[,…]}
     [,PRIMARY KEY(ColumnName[,…])]
     [,FOREIGN KEY(ColumnName [,…])
                REFERENCES RefTableName (ColumnName [,…])
                              [ON  DELETE CASCADE]
 [UNIQUE HASH ON (HashColumnName[,…]) PAGES=PrimaryPages]
 [WHERE ExternalSearchCondition]
 [AGING USE ColumnName
                LIFETIME Num1 {MINUTE[S] |HOUR[S] | DAY[S]}
                [CYCLE Num2 {MINUTE[S] |HOUR[S] |DAY[S]}]
 [ON|OFF]
 ]
} [,...];
```

CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP

For ASYNCHRONOUS WRITETHROUGH cache groups, the syntax is:

```
CREATE [ASYNCHRONOUS] WRITETHROUGH CACHE GROUP
 [Owner.]GroupName
 FROM
   {[Owner.]TableName (
     {ColumnDefinition[,…]}
    [,PRIMARY KEY(ColumnName[,…])]
     [FOREIGN KEY(ColumnName [,…])
         REFERENCES RefTableName (ColumnName [,…])]
                 [ ON DELETE CASCADE ]
 UNIQUE HASH ON (HashColumnName[,…]) PAGES=PrimaryPages]
 [AGING {LRU|
           USE ColumnName
                LIFETIME Num1 {MINUTE[S] |HOUR[S] |DAY[S]}
               [CYCLE Num2 {MINUTE[S] |HOUR[S] |DAY[S]}]
           }[ON|OFF]
 ]
} [,...];
```

CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP

For SYNCHRONOUS WRITETHROUGH cache groups, the syntax is:

```
CREATE SYNCHRONOUS WRITETHROUGH
CACHE GROUP [Owner.]GroupName
 FROM
    {[Owner.]TableName (
      {ColumnDefinition[,…]}
     [,PRIMARY KEY(ColumnName[,…])]
      [FOREIGN KEY(ColumnName [,…])
          REFERENCES RefTableName (ColumnName [,…])}]
                    [ ON DELETE CASCADE ]
 [UNIQUE HASH ON (HashColumnName[,…]) PAGES=PrimaryPages]
 [AGING {LRU|
           USE ColumnName
                LIFETIME Num1 {MINUTE[S] |HOUR[S] |DAY[S]}
                [CYCLE Num2 {MINUTE[S] |HOUR[S] |DAY[S]}]
           }[ON|OFF]
 ]
} [,...];
```

CREATE USERMANAGED CACHE GROUP

For user managed cache groups, the syntax is:

```
CREATE [USERMANAGED] CACHE GROUP [Owner.]GroupName
 [AUTOREFRESH
    [MODE {INCREMENTAL | FULL}]
    [INTERVAL IntervalValue {MINUTE[S] | SECOND[S] |
          MILLESECOND[S] }]
    [STATE {ON|OFF|PAUSED}]
 ]
 FROM
  {[Owner.]TableName (
    {ColumnDefinition[,…]}
    [,PRIMARY KEY(ColumnName[,…])]
    [FOREIGN KEY(ColumnName[,…])
              REFERENCES RefTableName (ColumnName [,…])]
                        [ON DELETE CASCADE]
    [, {READONLY | PROPAGATE | NOT PROPAGATE}]
```

```
[UNIQUE HASH ON (HashColumnName[,…]) PAGES=PrimaryPages]
[WHERE ExternalSearchCondition]
[AGING {LRU|
        USE ColumnName
            LIFETIME Num1 {MINUTE[S] |HOUR[S] |DAY[S]}
            [CYCLE Num2 {MINUTE[S] |HOUR[S] |DAY[S]}]
        }[ON|OFF]
 ]
} [,...];
```

Parameters The parameters for the general cache group description (everything before the FROM keyword) are:.

[*Owner.*]*GroupName*	Owner and name assigned to the new cache group.
AUTOREFRESH	The AUTOREFRESH parameter automatically propagates changes from the Oracle database to the TimesTen cache group. For details, see "AUTOREFRESH in Cache Groups" on page 219.
MODE [INCREMENTAL | FULL]	Determines which rows in the cache are updated during an autorefresh. If the INCREMENTAL clause is specified, TimesTen refreshes only rows that have been changed on Oracle since the last propagation. If the FULL clause is specified, TimesTen updates all rows in the cache with each autorefresh. The default AUTOREFRESH mode is INCREMENTAL.
INTERVAL *IntervalValue*	Indicates the interval at which autorefresh should occur in units of minutes, seconds or milliseconds. *IntervalValue* is an integer value that specifies how often autorefresh should be scheduled, in MINUTES, SECONDS or MILLISECONDS. The default *IntervalValue* value is 5 minutes. If the specified interval is not long enough for an autorefresh to complete, a runtime warning is generated and the next autorefresh waits until the current one finishes. An informational message is generated in the daemon log if the wait queue reaches 10.
STATE [ON | OFF | PAUSED]	Specifies whether autorefresh should be ON or OFF when the cache group is created. You can alter this setting later through the ALTER CACHE GROUP command. By default, the AUTOREFRESH STATE is PAUSED.
FROM	Designates one or more table definitions for the cache group.

Everything after the FROM keyword comprises the definitions of the Oracle tables cached in the cache group. The syntax for each table definition is similar to that of a CREATE TABLE statement. However, primary key constraints are *required* for the cache group table.

Table definitions have the parameters:

[*Owner*.]*TableName*	Owner and name to be assigned to the new table. If you do not specify the owner name, your login becomes the owner name for the new table.
ColumnDefinition	Name of an individual column in a table, its data type and whether or not it is nullable. Each table must have at least one column. See "Column Definition" on page 257.
PRIMARY KEY (*ColumnName*[,…])	Specifies that the table has a primary key. Primary key constraints are required for a cache group. *ColumnName* is the name of the column that forms the primary key for the table to be created. Up to 16 columns can be specified for the primary key. Cannot be specified with UNIQUE in one specification.
FOREIGN KEY (*ColumnName*[,…])	Specifies that the table has a foreign key. *ColumnName* is the name of the column that forms the foreign key for the table to be created. See "FOREIGN KEY" on page 254.
REFERENCES *RefTableName* (*ColumnName*[,…])	Specifies the table which the foreign key is associated with. *RefTableName* is the name of the referenced table and *ColumnName* is the name of the column referenced in the table.
[ON DELETE CASCADE]	Enables the ON DELETE CASCADE referential action. If specified, when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign key values are also deleted.
READONLY	Specifies that changes cannot be made on the cached table.
PROPAGATE | NOT PROPAGATE	Specifies whether changes to the cached table are automatically propagate changes to the cached table to the corresponding Oracle table at commit time.
UNIQUE HASH ON *HashColumnName*	Specifies that a hash index is created on this table. *HashColumnName* identifies the column that is to participate in the hash key of this table. The columns specified in the hash index must be identical to the columns in the primary key.

PAGES=*PrimaryPages*	Specifies the expected number of pages in the table. The *PrimaryPages* number determines the number of hash buckets created for the hash index. The minimum is 1. If your estimate is too small, performance is degraded. See the CREATE TABLE section for more information.
WHERE *ExternalSearchCondition*	The WHERE clause evaluated by Oracle for the cache group table. This WHERE clause is added to every LOAD and REFRESH operation on the cache group. It may not directly reference other tables. It is parsed by both TimesTen and Oracle. See the section "Using WHERE clauses" in Chapter 3 of the *TimesTen Cache Connect to Oracle Guide*.
AGING LRU [ON | OFF]	If specified, rows of the table are aged using the LRU policy. By removing the approximate least recently used data, LRU aging keeps only the recently used data in cache. Data is removed if the data store space in-use exceeds a specified threshold value. LRU aging frees up space occupied by infrequently used data. The LRU aging policy is defined when you specify the AGING LRU clause. ON or OFF settings refer to the aging state. The OFF setting indicates the LRU aging policy is defined, but the aging state is disabled. Because the aging state is OFF, aging is not done automatically. The ON setting indicates the LRU aging policy is defined and the aging state is enabled. Because the aging state is ON, aging is done automatically. The default is ON. LRU aging is not supported for cache groups with autorefresh. For more information on LRU aging, see the section “Aging in Cache Groups” on page 221. For detailed information on LRU aging, see "Implementing aging in a cache group" of the TimesTen Cache Connect to Oracle Guide.

AGING USE *ColumnName*... [ON | OFF]	If specified, rows of the table are aged using the time-based aging policy. Time-based aging allows you to specify the lifetime data is kept in cache. The time unit is expressed in terms of days, hours, or minutes. In order to use time-based aging, you must define a column on the root table. The value of this column is subtracted from SYSDATE, truncated using the specified unit (minute, hour, day) and then compared to the LIFETIME value you specify. If the result is *greater than* the LIFETIME value, then the row is a candidate for aging. Define the column as NOT NULL and of type TIMESTAMP. The values of the timestamp columns are updated by your applications. If the value of this column is unknown for some rows, and you do not want the rows to be aged, define the column with a large default value. *ColumnName* refers to the timestamp column you defined on the root table. The time-based aging policy is defined when you specify the AGING USE clause. ON or OFF settings refer to the aging state. The OFF setting indicates the time-based aging policy is defined, but the aging state is disabled. Because the aging state is OFF, aging is not done automatically. The ON setting indicates the time-based aging policy is defined and the aging state is enabled. Because the aging state is ON, aging is done automatically. The default is ON. For more information on time-based aging, see the section "Aging in Cache Groups" on page 221. For detailed information on time-based aging, see "Implementing aging in a cache group" of the TimesTen Cache Connect to Oracle Guide..

LIFETIME *Num1* {MINUTE[S] |HOUR[S] | DAY[S] }	This required clause is specific to time-based aging; It follows the AGING USE *ColumnName* clause. LIFETIME is the minimum amount of time data is kept in cache. For example, keep data in cache for *Num1* unit of time where unit can be minutes, hours or days. This means rows that exceed the specified LIFETIME value are aged out (deleted from the table). *Num1* must be a positive integer constant; Its value indicates the amount of time in minutes, hours or days that the rows should stay in cache. The concept of time resolution is supported. The time-unit specified for the lifetime of data is the resolution granularity for which data should be kept in cache. If, for example, DAYS is specified as the time resolution, then all rows whose timestamp belongs to the same day are aged out at the same time. If HOURS is specified as the time resolution, then all rows with timestamp values within that hour are aged out at the same time. Thus, a LIFETIME of *3* DAYS is different than a LIFETIME of *72* HOURS (3*24) or a LIFETIME of *432* MINUTES (3*24*60). Days represent all days of the week including weekends and holidays.
[CYCLE *Num2* {MINUTE[S] |HOUR[S] | DAY[S] }]	This optional clause is specific to time-based aging and follows the LIFETIME clause. It indicates how often the system should examine rows to see if data exceeds the specified LIFETIME values and thus should be aged out (deleted). *Num2* must be a positive integer constant. If you do not specify the CYCLE clause, then the default value is 5 minutes. This means that if aging is enabled, rows are examined every 5 minutes starting as soon as possible. If you specify 0 for *Num2*, then aging is continuous and the aging thread never sleeps.

AUTOREFRESH in Cache Groups

The AUTOREFRESH parameter automatically propagates changes from the Oracle database to the TimesTen cache. By default, system managed cache groups automatically propagate changes from the Oracle database to the TimesTen cache. For a description of cache group types, see "User and system managed cache groups" on page 211.

TimesTen supports FULL or INCREMENTAL AUTOREFRESH. In FULL mode, the entire cache is periodically unloaded and then reloaded. In INCREMENTAL mode, TimesTen installs triggers in the Oracle database to track any changes to it, and periodically updates only the rows that have changed

in Oracle. The first incremental refresh is always a full refresh, unless the PAUSED state is being used. The default mode is INCREMENTAL.

FULL AUTOREFRESH is more efficient when most of the Oracle table rows are changed. INCREMENTAL AUTOREFRESH is more efficient when there are fewer changes.

AUTOREFRESH is scheduled by TimesTen when the transaction that contains the specified AUTOREFRESH statement is committed. The statement types that cause AUTOREFRESH to be scheduled are:

- A CREATE CACHE GROUP statement in which AUTOREFRESH is specified, and the AUTOREFRESH state is specified as ON
- An ALTER CACHE GROUP statement in which the AUTOREFRESH state has been changed to ON
- A LOAD CACHE GROUP statement on an empty cache group whose autorefresh state is PAUSED. (The state is ON after the LOAD.)

The specified interval determines how often AUTOREFRESH occurs.

The current STATE of AUTOREFRESH can be ON, OFF or PAUSED. By default, the AUTOREFRESH STATE is PAUSED.

To use AUTOREFRESH in a cache group, you must first run the TimesTen ttAdmin utility and supply an Oracle ID and PWD.: a cacheUid and a cachePwd or run the ttCacheUidPwdSet built-in procedure.

Before you can activate an AUTOREFRESH cache group using incremental refreshes, certain triggers must be installed in Oracle. For details, see the *TimesTen Cache Connect Guide*. TimesTen can automatically install all necessary triggers and procedures for you in Oracle.

In order to use AUTOREFRESH with a cache group, you must specify AUTOREFRESH when you create the cache group. You can change the MODE, STATE and INTERVAL AUTOREFRESH settings after a cache group has been created by using the ALTER CACHE GROUP command.

Once a cache group has been specified as either AUTOREFRESH or PROPAGATE, you cannot change these attributes.

The NOT PROPAGATE clause cannot be used with an AUTOREFRESH statement.

There are specific requirements and restrictions when replication cache groups. See "Replicating cache groups," in the *TimesTen to TimesTen Replication Guide*.

Description

- Two cache groups cannot have the same owner name and group name. If you do not specify the owner name, your login becomes the owner name for the new cache group.
- Dynamic parameters are not allowed in the WHERE clause.
- Cache group tables cannot be created as temporary tables.

- Each table must correspond to a table in Oracle.
- Each column in the table must match each column in the Oracle table, both in name and in type. See "Compatibility Between TimesTen and Oracle," in the *TimesTen Cache Connect to Oracle Guide*. In addition, each column name must be fully qualified with an owner and table name when referenced in a WHERE clause.
- The WHERE clause can only directly refer to the cache group table. Outside tables can only be referenced with a sub-select.
- Generally, you do not have to fully qualify the column names in the WHERE clause of the CREATE CACHE GROUP, LOAD, UNLOAD, REFRESH or FLUSH statements. However, since TimesTen automatically generates queries that join multiple tables in the same cache group, a column needs to be fully qualified if there is more than one table in the cache group that contains columns with the same name.
- By default, a T-tree index is created to enforce the primary key for a cache group table. Use the UNIQUE HASH clause to specify a hash index for the primary key.
  - If your application performs range queries over a cache group table's primary key, then choose a T-tree index for that cache group table by omitting the UNIQUE HASH clause.
  - If, however, your application performs only exact match lookups on the primary key, then a hash index may offer better response time and throughput. In such a case, specify the UNIQUE HASH clause. See "CREATE TABLE" on page 251 for more information on the UNIQUE HASH clause.
- Use ALTER TABLE to change the representation of the primary key index for a table.
- If access control is disabled, no privilege is necessary to execute a DELETE statement which triggers the ON DELETE CASCADE action. Otherwise, WRITE privileges are required on all tables affected by the ON DELETE CASCADE action.
- The total number of rows reported by the DELETE statement does not include rows deleted from child tables as a result of the ON DELETE CASCADE action.
- ON DELETE CASCADE restrictions:
  - For cache group tables with the propagate attribute and for tables of SWT and AWT cache groups, foreign keys specifed with ON DELETE CASCADE must be a proper subset of foreign keys with ON DELETE CASCADE in Oracle.

Aging in Cache Groups

- Aging: Usage

- Aging on a cache group is based on the instance. Define aging on the root table only.
- When you define an aging policy on the root table, you define:
  1. The type of aging: LRU or time-based.
  2. The state of aging either ON (enabled) or OFF (disabled).
  3. For time-based aging: the timestamp column, LIFETIME, and CYCLE.
- LRU attributes are defined by calling procedure ttAgingLruConfig. LRU attributes are not defined at the table level. If you do not call this procedure, then the default values are used.
- Define an aging policy for either an existing cache group or a new cache group. For an existing cache group, use the ALTER TABLE... ADD AGING clause. For a new cache group, specify the AGING clause on the root table. See Example 5.36.

  To add LRU aging on an existing table:

  ALTER TABLE... ADD AGING LRU [ON|OFF]

  To add time-based aging on an exising table:

  ALTER TABLE... ADD AGING USE ...
- After you have defined the aging policy for the root table, you cannot alter the aging policy. This means you cannot change the policy from LRU to time-based or from time-based to LRU. For time-based aging, you cannot change the timestamp column. Essentially, you cannot specify the ALTER TABLE...ADD AGING clause on a table with an existing aging policy.
- To change the aging policy, drop aging then redefine the aging policy:

  ALTER TABLE... DROP AGING
- After you have defined the aging policy for the table, you can change the aging state; that is you can change the state defined in the aging policy from ON (automatic aging enabled) to OFF (automatic aging disabled) or from OFF (automatic aging disabled) to ON (automatic aging enabled). To do this:

  ALTER TABLE... SET AGING [ON | OFF]
- For time-based aging, you can change the LIFETIME and CYCLE:

  ALTER TABLE...SET AGING LIFETIME *Num1*
  {MINUTE[S] |HOUR[S] |DAY[S]}

  ALTER TABLE... SET AGING CYCLE *Num2*
  {MINUTE[S] |HOUR[S] |DAY[S]}

  For more information, see **ALTER TABLE**.

• Aging: Built-in Procedures

– Use ttAgingLRUConfig to specify the following properties for LRU aging:

    1. LowUsageThreshold: (percentage of Perm-size allocated)
    2. HighUsageThreshold: (percentage of Perm-size allocated)
    3. AgingCycle: How often rows are examined for aging. (In minutes).

– The default attribute settings for LRU aging are:

    1. LowUsageThreshold: 80%
    2. HighUsageThreshold: 90%
    3. Aging Cycle: 1 minute

– For LRU aging, the aging thread wakes up at the specified aging cycle and checks if the space usage is over the HighUsageThreshold; If it is not, it goes back to sleep; Otherwise, it begins removing the least used data until either the space usage is at or below the LowUsageThreshold percentage or there is no data left to remove.

– ttAgingLRUConfig applies to *all* tables with LRU aging. Arguments to this procedure can be omitted; If omitted, the previous setting is retained. Arguments are positive integers.

– For more information on the ttAgingLRUConfig procedure, see "ttAgingLRUConfig".

– Use ttAgingScheduleNow to schedule the aging process as soon as possible. (the process starts when the subdaemon awakes). When you call this procedure, the aging process is started for a table regardless if its aging state is OFF (disabled) or ON (enabled). The aging process is started only once; the previous aging state is unchanged. For example, if aging state is OFF when you call ttAgingScheduleNow, the aging process starts. When aging is complete, if your aging state is OFF, aging does not continue. To continue aging, you must call ttAgingScheduleNow again or change the aging state to ON.

– If you call ttAgingScheduleNow, and the aging state is ON, the aging cycle is reset based on the time ttAgingScheduleNow was called.

– ttAgingScheduleNow is useful if you wish to schedule the aging process as soon as possible or if you wish to start the aging process at a specified time (via an external scheduler e.g. cron job). In the latter case, set the aging state to OFF and then call ttAgingScheduleNow to invoke aging.

– To schedule aging on one table, call ttAgingScheduleNow specifying the name of the table and enclosing the table name in quotes. The specified table can be defined with either LRU aging or time-based aging:

```
CALL ttAgingScheduleNow ('TableName')
```

– To schedule aging on all tables, including tables defined with both *LRU* aging and *time-based* aging, call ttAgingScheduleNow without any arguments.

```
CALL ttAgingScheduleNow ( );
```

  - For more information on the ttAgingScheduleNow procedure, see "ttAgingScheduleNow".

- Aging: Specifics
  - LRU and time-based aging can be combined in one system. If you use only LRU aging , the aging thread wakes up based on the cycle specified for the whole data store; If you use only time-based aging, the aging thread wakes up based on an optimal frequency. This frequency is determined by the values specified in the CYCLE clause for all tables. If you use both LRU and time-based aging, then the thread wakes up based on a combined consideration of both types.
  - You can implement sliding windows with time-based aging; The table contains the last *Num1* time unit of data. For autorefresh cache groups, both sliding IN and sliding OUT are handled automatically by aging. New data is loaded at every autorefresh interval and expired data is deleted according to the aging schedule.
  - For non-autorefresh cache groups, sliding OUT is handled automatically by aging. Use manual load, load on select, refresh, or inserts into TimesTen to bring data into cache. Past data that falls out of the window is aged out. As the window changes, data is not guaranteed to be deleted; data is deleted when the aging thread wakes up.
- Aging: For LRU aging, the following rules determine if a row is accessed or referenced:
  - Any rows used to build the result set of a SELECT statement.
  - Any rows used to build the result set of an INSERT SELECT statement.
  - Any rows that are about to be updated or deleted.
- Aging: LRU and compiled commands:
  - Compiled commands are marked invalid and need recompilation when you either drop LRU aging from or add LRU aging to tables that are referenced in the commands.
- Aging: Foreign Key Issues
  - For LRU aging, if a child row is not a candidate for aging, then this child row is not deleted; Nor will the parent row be deleted; ON DELETE CASCADE settings are ignored.
  - For time-based aging, if a parent row is a candidate for aging, then all child rows are deleted. ON DELETE CASCADE (whether specified or not) is ignored.
- Aging: Restrictions
  - LRU aging and time-based aging can only be specified on the root table.
  - LRU aging is not supported on a cache group defined with the autorefresh attribute.

- Aging cannot be added, altered, or dropped for an autorefresh cache group while the cache agent is active. Stop the cache agent first.
- LRU aging and time-based aging are not supported on detail tables of materialized views.
- LRU aging and time-based aging are not supported on global temporary tables.
- You cannot drop the timestamp column that is used for time-based aging.

Example5.32 Create a READONLY cache group:

```
CREATE READONLY CACHE GROUP CustomerOrders
AUTOREFRESH INTERVAL 10 MINUTES
FROM
CUSTOMER (CUSTID INT NOT NULL,
       NAME CHAR(100) NOT NULL,
       ADDR CHAR(100),
       ZIP INT,
       REGION CHAR(10),
       PRIMARY KEY(CUSTID)),
ORDERTAB (ORDERID INT NOT NULL,
       CUSTID INT NOT NULL,
       PRIMARY KEY (ORDERID),
       FOREIGN KEY (CUSTID) REFERENCES CUSTOMER(CUSTID));
```

Example5.33 Create an ASYNCHROUS WRITETHROUGH cache group:

```
CREATE ASYNCHRONOUS WRITETHROUGH cache group Customers
FROM
CUSTOMER (CUSTID INT NOT NULL,
       NAME CHAR(100) NOT NULL,
       ADDR CHAR(100),
       ZIP INT,
       PRIMARY KEY(CUSTID));
```

Example5.34 Create a SYNCHRONOUS WRITETHROUGH cache group:

```
CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP Customers
FROM
CUSTOMER (CUSTID INT NOT NULL,
       NAME CHAR(100) NOT NULL,
       ADDR CHAR(100),
       ZIP INT,
       PRIMARY KEY(CUSTID));
```

Example5.35 Create a USERMANAGED cache group:

```
CREATE USERMANAGED CACHE GROUP UpdateAnywhereCustomers
AUTOREFRESH
```

```
       MODE INCREMENTAL
       INTERVAL 30 SECONDS
       STATE ON
FROM
CUSTOMER (CUSTID INT NOT NULL,
       NAME CHAR(100) NOT NULL,
       ADDR CHAR(100),
       ZIP INT,
       PRIMARY KEY(CUSTID),
       PROPAGATE);
```

Example 5.36 Create a cache group with time-based aging; Specify AgeTimestamp as the column for aging; LIFETIME 2 hours, CYCLE 30 minutes; Aging state is not specified, so the default setting (ON) is used.

```
CREATE READONLY CACHE GROUP AgingCacheGroup
AUTOREFRESH
       MODE INCREMENTAL
       INTERVAL 5 MINUTES
       STATE PAUSED
FROM
CUSTOMER (CustomerId NUMBER NOT NULL,
       AgeTimestamp TIMESTAMP NOT NULL,
       PRIMARY KEY (CustomerId))
       AGING USE AgeTimeStamp LIFETIME 2 HOURS CYCLE 30 MINUTES;
```

```
Command> describe customer;
Table USER.CUSTOMER:
  Columns:
   *CUSTOMERID                    NUMBER NOT NULL
    AGETIMESTAMP                  TIMESTAMP (6) NOT NULL
  Aging use AGETIMESTAMP lifetime 2 hours cycle 30 minutes on

1 table found.
(primary key columns are indicated with *)
```

(See Also “ALTER CACHE GROUP” on page 167
“ALTER TABLE” on page 186
“DROP CACHE GROUP” on page 277
“FLUSH CACHE GROUP” on page 285
“UNLOAD CACHE GROUP” on page 321

CREATE INDEX

The CREATE INDEX statement creates a *T-tree index* on one or more columns of a table and assigns a name to the new index. A T-tree index is an index structure designed for in-memory applications. A T-tree:

- Speeds up range searches (but can also be used for efficient equality searches).
- Is optimized for in-memory data management.
- Provides efficient sorting by column value.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

CREATE [UNIQUE] INDEX [*Owner.*]*IndexName* ON
[*Owner.*]*TableName* ({ *ColumnName* [ASC | DESC] }
[, ...])

Parameters

The CREATE INDEX statement has the parameters:

UNIQUE	Prohibits duplicates in the index. If UNIQUE is specified, each possible combination of index key column values can occur in only one row of the table. If UNIQUE is omitted, duplicate values are allowed. When you create a unique index, all existing rows must have unique values in the indexed column(s).
[*Owner.*] *IndexName*	Name to be assigned to the new index. A table cannot have two indexes with the same name. If the owner is specified, it must be the same as the owner of the table.
[*Owner.*] *TableName*	Designates the table for which an index is to be created.
ColumnName	Name of a column to be used as an index key. You can specify up to 16 columns in order from major index key to minor index key.
ASC | DESC	Specifies the order of the index to be either ascending (the default) or descending. In TimesTen, this parameter is currently ignored.

Description

- The CREATE INDEX statement enters the definition of the index in the system catalog and initializes the necessary data structures. Any rows in the table are then added to the index. In TimesTen, performance is the same regardless of whether the table is created, indexed and populated or created, then populated and indexed.

- If UNIQUE is specified, all existing rows must have unique values in the indexed column(s).
- The new index is maintained automatically until the index is deleted by a **DROP INDEX** statement or until the table associated with it is dropped.
- Any prepared statements that reference the table with the new index are automatically prepared again the next time they are executed. Then the statements can take advantage, if possible, of the new index.
- NULL compares higher than all other values for sorting.
- An index on a temporary table cannot be created by a connection if any other connection has a non-empty instance of the table.
- If you are using linguistic comparisons, you can create a linguistic index. A linguistic index uses sort key values and storage is required for these values. Only one unique value for NLS_SORT is allowed for an index. For more information on linguistic indexes and linguistic comparisons, see **Using linguistic indexes** in the Globalization Support chapter.

Example 5.37 This unique index ensures that all part numbers are unique.

```
CREATE UNIQUE INDEX Purchasing.PartNumIndex
ON Purchasing.Parts (PartNumber);
```

Example 5.38 Create a linguistic index named german_index on table employees1. If you wish to have more than one linguistic sort, create a second linguistic index.

```
Command> create table employees1 (id character (21),
id2 character (21));
Command>  create index german_index on employees1
(NLSSORT(id, 'NLS_SORT=GERMAN'));
Command>  create index german_index2 on employees1
NLSSORT(id2, 'nls_sort=german_ci'));
Command> indexes employees1;

Indexes on table SAMPLEUSER.EMPLOYEES1:
  GERMAN_INDEX: non-unique T-tree index on columns:
    NLSSORT(ID,'NLS_SORT=GERMAN')
  GERMAN_INDEX2: non-unique T-tree index on columns:
    NLSSORT(ID2,'nls_sort=german_ci')
  2 indexes found.


1 table found.
```

See Also "DROP INDEX" on page 278

CREATE MATERIALIZED VIEW

The CREATE MATERIALIZED VIEW statement creates a view of the table specified in the *SelectQuery* clause. The original tables used to create a view are referred to as “detail” tables.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

CREATE MATERIALIZED VIEW *ViewName* AS *SelectQuery*
[PRIMARY KEY (*ColumnName* [,…])]
[UNIQUE HASH ON (*HashColumnName* [,…])
PAGES = *PrimaryPages*]

Parameters

The CREATE MATERIALIZED VIEW statement has the parameters:

ViewName	Name assigned to the new view.
SelectQuery	Selects column from the detail table(s) to be used in the view. Can also create indexes on the view.
ColumnName	Name of the column(s) that forms the primary key for the view to be created. Up to 16 columns can be specified for the primary key. Each result column name of a viewed table must be unique. The column name definition cannot contain the table or owner component.
HashColumnName	Column defined in the view that is to participate in the hash key of this table. The columns specified in the hash index must be identical to the columns in the primary key.
PrimaryPages	Specifies the expected number of pages in the table. This number determines the number of hash buckets created for the hash index. The minimum is 1. If your estimate is too small, performance is degraded. See the CREATE TABLE section for more information.

Description

Restrictions on the materialized view and detail tables

- A materialized view is read-only and cannot be updated directly. A materialized view is updated only when changes are made to the associated detail tables. Therefore a materialized view cannot be the target of a DELETE, UPDATE or INSERT statement.

- Materialized views defined on replicated tables may result in replication failures or inconsistencies if the materialized view is specified so that overflow or underflow conditions occur when the materialized view is updated.
- Detail tables can be replicated, but views themselves cannot be replicated. If detail tables are replicated, TimesTen automatically updates the corresponding view(s).
- Neither a view nor its detail tables can be part of a cache group.
- TimesTen does not automatically create indexes on materialized views or detail tables. Referential constraints cannot be defined on materialized views.
- By default, a T-tree index is created to enforce the primary key for a materialized view. Use the UNIQUE HASH clause to specify a hash index for the primary key.
  - If your application performs range queries over a materialized view's primary key, then choose a T-tree index for that view by omitting the UNIQUE HASH clause.
  - If, however, your application performs only exact match lookups on the primary key, then a hash index may offer better response time and throughput. In such a case, specify the UNIQUE HASH clause. See "CREATE TABLE" on page 251 for more information on the UNIQUE HASH clause.
- Use ALTER TABLE to change the representation of the primary key index or resize a hash index.
- You cannot add or drop columns with the ALTER TABLE statement. To change the structure of the materialized view, drop then recreate the view.
- A view cannot be dropped with a DROP TABLE statement. Use the DROP VIEW statement.

Restrictions on the MATERIALIZED VIEW query

There are several restrictions on the query that is used to define the materialized view.

- A SELECT * query in a materialized view definition is expanded at view creation time. Any columns added after a materialized view is created do not affect the materialized view.
- Temporary tables cannot be used in a materialized view definition. Non-materialized views and derived tables cannot be used to define a materialized view.
- All columns in the GROUP BY list must be included in the select list.
- Aggregate view must include a COUNT(*) in the SELECT list.
- SUM and COUNT are allowed, but not expressions involving them, including AVG.

- The following cannot be used in a SELECT statement that is creating a materialized view:
  - DISTINCT
  - FIRST
  - HAVING
  - ORDER BY
  - UNION
  - UNION ALL
  - MINUS
  - INTERSECT
  - JOIN
  - User functions: USER, CURRENT_USER, SESSION_USER
  - Subqueries
  - NEXTVAL and CURRVAL
  - Derived tables and joined tables
- Each expression in the select list must have a unique name. A name of a simple column expression would be that column's name unless a column alias is defined. RowId is considered an expression and needs an alias.
- No SELECT FOR UPDATE or SELECT FOR INSERT statements can be used on a view.
- OUTER JOINs are allowed, but the **SELECT** list must project at least one non-nullable column from each of the inner tables specified in the OUTER JOIN. Outer join syntax for a **SELECT** in a materialized view definition is identical to that in a top-level **SELECT**. The restrictions noted in the description of **SELECT** statements apply. The (+) symbol must be used to specify OUTER JOINs of a materialized view.
- Each inner table can only be outer joined with at most one table.
- Self joins are allowed. A self join is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition.

Creates a materialized view of columns from the *customer* and *bookOrder* tables.

```
CREATE MATERIALIZED VIEW CustOrder AS
SELECT custNo, custName, ordNo, book
FROM customer, bookOrder
WHERE customer.custNo=bookOrder.custNo;
```

Example5.39 Creates a materialized view of columns *x1* and *y1* from the *t1* table.

```
CREATE MATERIALIZED VIEW v1 AS SELECT x1, y1 FROM t1
PRIMARY KEY (x1) UNIQUE HASH (x1) PAGES=100;
```

Example5.40 Creates a materialized view from an outer join of columns *x1* and *y1* from the *t1 and t2* tables.

```
CREATE MATERIALIZED VIEW v2 AS SELECT x1, y1 FROM t1, t2
WHERE x1=x2(+);
```

See Also
“CREATE TABLE” on page 251
“CREATE VIEW” on page 272
“DROP VIEW” on page 284

CREATE REPLICATION

TimesTen SQL configuration for replication provides a programmable way to configure replication. The configuration can be embedded in C, C++ or Java code. Replication can be configured locally or from remote systems using client/server.

In addition, you need to use the **ttRepAdmin** utility to maintain operations not covered by the supported SQL statements. Use **ttRepAdmin** to change replication state, duplicate data stores, list the replication configuration and view replication status.

The CREATE REPLICATION statement:

- Defines a replication scheme at a participating data store.
- Installs the specified configuration in the executing data store's replication system tables.
- Typically consists of one or more replication ELEMENT specifications and zero or more STORE specifications.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges.

Definitions

A *replication element* is an entity that TimesTen synchronizes between data stores. A replication element can be a whole table or a data store. A data store can include most types of tables and cache groups. It can include *only specified* tables and cache groups, or include all tables *except specified* tables and cache groups. It cannot include temporary tables or views, whether materialized and non-materialized.

A *replication scheme* is a set of replication elements, as well as the data stores that maintain copies of these elements.

When replicating cache groups:

- When replicating cache groups between data stores, both cache groups must be identical, with the exception of the settings for AUTOREFRESH and PROPAGATE.
- When replicating a cache group with AUTOREFRESH, the cache group on the subscriber must set the autorefresh STATE to OFF. In a bi-directional replication scheme, one of the cache groups must set the autorefresh STATE to OFF.
- If a master cache group specifies PROPAGATE, the subscriber cache group must set the autorefresh STATE to OFF.

For more detailed information on SQL configuration for replication, see "Replicating cache groups" in the *TimesTen Replication Guide*.

SQL syntax

CREATE REPLICATION [*Owner.*]*ReplicationSchemeName*

{ ELEMENT *ElementName*
{ DATASTORE | { TABLE [*Owner.*]*TableName* [*CheckConflicts*]} |
SEQUENCE [Owner.]*SequenceName*}
{ MASTER | PROPAGATOR } *FullStoreName*
[TRANSMIT { NONDURABLE | DURABLE }]
{ SUBSCRIBER *FullStoreName* [, ...] [*ReturnServiceAttribute*] } [, ...] }
[...]
[{INCLUDE | EXCLUDE}{TABLE | CACHE GROUP | SEQUENCE}
[*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...]]

[STORE *FullStoreName* [*StoreAttributes* [, ...]]] [...]

Syntax for *CheckConflicts* is described in "CHECK CONFLICTS" on page 240.

Syntax for *ReturnServiceAttribute* is:

{ RETURN RECEIPT [BY REQUEST] |
RETURN TWOSAFE [BY REQUEST] |
NO RETURN }

Syntax for *StoreAttributes* is:

[DISABLE RETURN {SUBSCRIBER | ALL} *NumFailures*]
[RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED]
[DURABLE COMMIT {ON | OFF}]
[RESUME RETURN *MilliSeconds*]
[LOCAL COMMIT ACTION {NO ACTION | COMMIT}]
[RETURN WAIT TIME *Seconds*]
[COMPRESS TRAFFIC {ON | OFF}
[PORT *PortNumber*]
[TIMEOUT *Seconds*]
[FAILTHRESHOLD *Value*]

Parameters The CREATE REPLICATION statement has the parameters:

[*Owner.*]*ReplicationSchemeName*	Name assigned to the new replication scheme. Replication schemes should have names that are unique from all other data store objects.
CheckConflicts	Check for replication conflicts when simultaneously writing to bi-directionally replicated data stores. See "CHECK CONFLICTS" on page 240.

COMPRESS TRAFFIC {ON | OFF}	Compress replicated traffic to reduce the amount of network bandwidth. ON specifies that all replicated traffic for the data store defined by STORE be compressed. OFF (the default) specifies no compression. See "Compressing replicated traffic" in the *TimesTen to TimesTen Replication Guide* for details.
DATASTORE	Define entire data store as ELEMENT. This type of ELEMENT can only be defined for a master data store that is *not* configured with an ELEMENT of type TABLE in the same or a different replication scheme. See "Defining replication elements" in the *TimesTen to TimesTen Replication Guide* for details.
{ INCLUDE | EXCLUDE }{TABLE | CACHE GROUP | SEQUENCE} [*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...]	INCLUDE includes in the DATASTORE element only the tables, sequences or cache groups listed. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE includes in the DATASTORE element all tables, sequences or cache groups *except* for those listed. Use one EXCLUDE clause for each object type (table, sequence or cache group). [*Owner.*]{*TableName* | *CacheGroupName* | *SequenceName*} [,...] is the list of table, sequence or cache group names, including optional owner names, to be included or excluded from the DATASTORE element.
DISABLE RETURN {SUBSCRIBER | ALL} *NumFailures*	Set the return service failure policy so that return service blocking is disabled after the number of timeouts specified by *NumFailures*. Selecting SUBSCRIBER applies this policy only to the subscriber that fails to acknowledge replicated updates within the set timeout period; ALL applies this policy to all subscribers should any of the subscribers fail to respond. This failure policy can be specified for either the RETURN RECEIPT or RETURN TWOSAFE service. See "Managing return service timeout errors and replication state changes" in the *TimesTen to TimesTen Replication Guide* for details.
DURABLE COMMIT {ON | OFF}	Set to override the **DurableCommits** setting on a data store and enable durable commit when return service blocking has been disabled by DISABLE RETURN.

ELEMENT *ElementName*	The entity that TimesTen synchronizes between data stores. TimesTen supports the entire data store (DATASTORE) and whole tables (TABLE) as replication elements. *ElementName* is the name given to the replication element. The *ElementName* for a TABLE element can be up to 30 characters in length; the *ElementName* for a DATASTORE element must be unique with respect to other DATASTORE element names within the first 20 chars. Each *ElementName* must be unique within a replication scheme. Also, you cannot define two ELEMENT descriptions for the same element. See "Defining replication elements" in the *TimesTen to TimesTen Replication Guide* for details.
FAILTHRESHOLD *Value*	For disk-based logging, *Value* is the number of log files that can accumulate for a subscriber data store. If this value is exceeded, the subscriber is set to the Failed state. The value 0 means “No Limit.” This is the default. See "Managing the log on a replicated data store" in the *TimesTen to TimesTen Replication Guide* for more information.

FullStoreName	The data store, specified as one of the following: • SELF • The prefix of the data store file name For example, if the data store path is `directory/subdirectory/data.ds0`, then `data` is the data store name that should be used. This is the data store file name specified in the **DataStore** attribute of the DSN description with optional host ID in the form: *DataStoreName* [ON *Host*] *Host* can be either an IP address or a literal host name assigned to one or more IP addresses, as described in "Configuring host IP addresses" in the *TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: “MyHost-500”. Host names can be up to 30 characters long.
LOCAL COMMIT ACTION {NO ACTION | COMMIT}	Specifies the default action to be taken for a return twosafe transaction in the event of a timeout. **Note:** This attribute is only valid when the RETURN TWOSAFE or RETURN TWOSAFE BY REQUEST attribute is set in the SUBSCRIBER clause. **NO ACTION** -- On timeout, the commit function returns to the application, leaving the transaction in the same state it was in when it entered the commit call, with the exception that the application is not able to update any replicated tables. The application can reissue the commit or rollback the call. **COMMIT** -- On timeout, the commit function writes a COMMIT log record and effectively ends the transaction locally. No more operations are possible on the same transaction. This default setting can be overridden for specific transactions by calling the *localAction* parameter in the **ttRepSyncSet()** procedure.

MASTER *FullStoreName*	The data store on which applications update the specified ELEMENT. The MASTER data store sends updates to its SUBSCRIBER data stores. The *FullStoreName* must be the data store specified in the **DataStore** attribute of the DSN description.
NO RETURN	Specifies that no return service is to be used. This is the default. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
PORT *PortNumber*	The TCP/IP port number on which the replication agent for the data store listens for connections. If not specified, the replication agent automatically allocates a port number.
PROPAGATOR *FullStoreName*	The data store that receives replicated updates and passes them on to other data stores. The *FullStoreName* must be the data store specified in the **DataStore** attribute of the DSN description.
RESUME RETURN *MilliSeconds*	If return service blocking has been disabled by DISABLE RETURN, this attribute sets the policy on when to re-enable return service blocking. Return service blocking is re-enabled as soon as the failed subscriber acknowledges the replicated update in a period of time that is less than the specified *MilliSeconds*.
RETURN RECEIPT [BY REQUEST]	Enables the return receipt service, so that applications that commit a transaction to a master data store are blocked until the transaction is received by all subscribers. RETURN RECEIPT applies the service to all transactions. If you specify RETURN REQUEST BY REQUEST, you can use the **ttRepSyncSet()** procedure to enable the return receipt service for selected transactions. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.

RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED	Set the return service failure policy so that return service blocking is either unchanged or disabled when the replication agent is in the Stop or Pause state. OFF is the default when using the return receipt service. ON is the default when using the return twosafe service See "Managing return service timeout errors and replication state changes" in the *TimesTen to TimesTen Replication Guide* for details.
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master data store are blocked until the transaction is committed on all subscribers. **Note:** This service can only be used in a bi-directional replication scheme where the elements are defined as DATASTORE. RETURN TWOSAFE applies the service to all transactions. If you specify RETURN TWOSAFE BY REQUEST, you can use the **ttRepSyncSet** procedure to enable the return receipt service for selected transactions. For details on the use of the return services, see "Using a return service" in the *TimesTen to TimesTen Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgement. The default value is 10 seconds. A value of '0' means that there is no wait time. Your application can override this timeout setting by calling the *returnWait* parameter in the **ttRepSyncSet** procedure.
SEQUENCE [*Owner.*]*SequenceName*	Define the sequence specified by [*Owner.*]*SequenceName* as ELEMENT. See "Defining replication elements" in the *TimesTen to TimesTen Replication Guide* for details.
STORE *FullStoreName*	Defines the attributes for a given data store. Data store attributes include PORT, TIMEOUT, and FAILTHRESHOLD. The *FullStoreName* must be the data store specified in the **DataStore** attribute of the DSN description.

SUBSCRIBER *FullStoreName*	A data store that receives updates from the MASTER data store(s). The *FullStoreName* must be the data store specified in the **DataStore** attribute of the DSN description.
TABLE [*Owner.*]*TableName*	Define the table specified by [*Owner.*]*TableName* as ELEMENT. See "Defining replication elements" in the *TimesTen to TimesTen Replication Guide* for details.
TIMEOUT *Seconds*	The amount of time a data store waits for a response from another data store before resending the message. Default: 120 seconds.
TRANSMIT {DURABLE | NONDURABLE}	Specifies whether to flush the master log to disk before sending a batch of committed transactions to the subscribers. TRANSMIT NONDURABLE specifies that records in the master log are not to be flushed to disk before they are sent to subscribers. This setting can only be used if the specified ELEMENT is a DATSTORE. This is the default for RETURN TWOSAFE transactions. TRANSMIT DURABLE specifies that records are to be flushed to disk before they are sent to subscribers. This is the default for asynchronous and RETURN RECEIPT transactions. **Note:** TRANSMIT DURABLE has no effect on RETURN TWOSAFE transactions. **Note:** TRANSMIT DURABLE cannot be set for active standby pairs. See "Setting transmit durability on data store elements" and "Impact of TRANSMIT DURABLE/ NONDURABLE on master data store recovery" in the *TimesTen to TimesTen Replication Guide* for more information.

CHECK CONFLICTS

Syntax

The syntax for CHECK CONFLICTS is:

{NO CHECK |

```
CHECK CONFLICTS BY ROW TIMESTAMP
        COLUMN ColumnName
        [ UPDATE BY { SYSTEM | USER } ]
        [ ON EXCEPTION { ROLLBACK [ WORK ] | NO ACTION } ]
        [ {REPORT TO 'FileName'
                [ FORMAT { XML | STANDARD } ] | NO REPORT
          } ]
}
```

Note: A CHECK CONFLICT clause can only be used for ELEMENTS of type TABLE.

Parameters

The CHECK CONFLICTS portion of the CREATE REPLICATION or ALTER REPLICATION statement has the parameters:

CHECK CONFLICTS BY ROW TIMESTAMP	Indicates that all update and uniqueness conflicts are to be detected. Conflicts are resolved in the manner specified by the ON EXCEPTION parameter. It also detects delete conflicts with UPDATE operations.
COLUMN *ColumnName*	Indicates the column in the replicated table to be used for timestamp comparison. The table is specified in the ELEMENT description by *TableName*. *ColumnName* is a nullable column of type BINARY(8) used to store a timestamp that indicates when the row was last updated. TimesTen rejects attempts to update a row with a lower timestamp value than the stored value. The specified *ColumnName* must exist in the replicated table on both the master and subscriber data stores.
NO CHECK	Specify to suppress conflict resolution for a given element.
UPDATE BY {SYSTEM | USER}	Specifies whether the timestamp values are maintained by TimesTen (SYSTEM) or the application (USER). The replicated table in the master and subscriber data stores must use the same UPDATE BY specification. See "System timestamp column maintenance" and "User timestamp column maintenance"in Chapter 8 of the *TimesTen to TimesTen Replication Guide* for more information. Default is UPDATE BY SYSTEM.

ON EXCEPTION {ROLLBACK [WORK] | NO ACTION}	Specifies how to resolve a detected conflict. ROW TIMESTAMP conflict detection has the resolution options: • ROLLBACK [WORK] — Abort the transaction that contains the conflicting action. • NO ACTION — Complete the transaction without performing the conflicting action (**UPDATE**, **INSERT**, or **DELETE**). Default is ON EXCEPTION ROLLBACK [WORK]
REPORT TO '*FileName*'	Specifies the file to log updates that fail the timestamp comparison. *FileName* is a SQL character string that cannot exceed 1,000 characters. (SQL character string literals are single-quoted strings that may contain any sequence of characters, including spaces.) The same file can be used to log failed updates for multiple tables.
[FORMAT { XML | STANDARD }]	Optionally specifies the conflict report format for an element. The default format is STANDARD.
NO REPORT	Specify to suppress logging of failed timestamp comparisons.

Description

- The names of all data stores on the same host must be unique for each replication scheme for each TimesTen instance.
- Replication elements can only be updated (by normal application transactions) through the MASTER data store. PROPAGATOR and SUBSCRIBER data stores are read only.
- If you define a replication scheme that permits multiple data stores to update the same table, see Chapter 8, "Conflict Resolution and Failure Recovery" in the *TimesTen Replication Guide* for recommendations on how to avoid conflicts when updating rows.
- SELF is intended for replication schemes where all participating data stores are local. Do not use SELF for a distributed replication scheme in a production environment, where spelling out the hostname for each data store in a script allows it to be used at each participating data store.
- Each attribute for a given STORE may be specified only once, or not at all.
- Specifying the PORT of a data store for one replication scheme specifies it for all replication schemes. All other data store attributes are specific to the replication scheme specified in the command.
- For replication schemes, *DataStoreName* is always the prefix of the TimesTen data store checkpoint file names. These are the files with the `.ds0` and `.ds1` suffixes that are saved on disk by checkpoint operations.

- If a row with a default NOT INLINE VARCHAR value is replicated, the receiver creates a copy of this value for each row instead of pointing to the default value if and only if the default value of the receiving node is different from the sending node.
- To use timestamp comparison on replicated tables, you must specify a nullable column of type BINARY(8) to hold the timestamp value. Define the timestamp column when you create the table. You cannot add the timestamp column with the ALTER TABLE statement. In addition, the timestamp column cannot be part of a primary key or index.
- If you specify the XML report format, two XML documents are generated:
  - FileName.xml: This file contains the DTD for the report and the root node for the report. It includes the document definition and the include directive.
  - FileName.include: This file is included in FileName.xml and contains all the actual conflicts.
  - The FileName.include file can be truncated; do not truncate the FileName.xml file.
  - For a complete description of the XML format, including examples of each conflict, see **Reporting conflicts to an XML file** in the *TimesTen Replication Guide*.
- If you specify a report format for an element and then drop the element, the corresponding report files are not deleted.

Example 5.41 Replicate the contents of REPL.TAB from MASTERDS to two subscribers, SUBSCRIBER1DS and SUBSCRIBER2DS.

```
CREATE REPLICATION REPL.TWOSUBSCRIBERS
       ELEMENT E TABLE REPL.TAB
         MASTER MASTERDS ON "SERVER1"
         SUBSCRIBER SUBSCRIBER1DS ON "SERVER2",
                    SUBSCRIBER2DS ON "SERVER3";
```

Example 5.42 Replicate the entire MASTERDS data store to the subscriber, SUBSCRIBER1DS. The FAILTHRESHOLD specifies that a maximum of 10 of log files can accumulate on MASTERDS before it assumes SUBSCRIBERIDS has failed.

```
CREATE REPLICATION REPL.WHOLESTORE
  ELEMENT E DATASTORE
     MASTER MASTERDS ON "SERVER1"
     SUBSCRIBER SUBSCRIBER1DS ON "SERVER2"
  STORE MASTERDS FAILTHRESHOLD 10;
```

Example5.43 Bi-directionally replicate the entire WESTDS and EASTDS data stores and enable the RETURN TWOSAFE service.

```
CREATE REPLICATION REPL.BIWHOLESTORE
  ELEMENT E1 DATASTORE
     MASTER WESTDS ON "WESTCOAST"
     SUBSCRIBER EASTDS ON "EASTCOAST"
        RETURN TWOSAFE
  ELEMENT E2 DATASTORE
     MASTER EASTDS ON "EASTCOAST"
     SUBSCRIBER WESTDS ON "WESTCOAST"
        RETURN TWOSAFE;
```

Example5.44 Same as Example 5.41, only enable the return receipt service for select transaction updates to the SUBSCRIBER1DS subscriber.

```
CREATE REPLICATION REPL.TWOSUBSCRIBERS
  ELEMENT E TABLE REPL.TAB
     MASTER MASTERDS ON "SERVER1"
     SUBSCRIBER SUBSCRIBER1DS ON "SERVER2"
        RETURN RECEIPT BY REQUEST
     SUBSCRIBER SUBSCRIBER2DS ON "SERVER3";
```

Example5.45 Replicate the contents of the CUSTOMERSWEST table from the WEST data store to the ROUNDUP data store and the CUSTOMERSEAST table from the EAST data store. Enable the return receipt service for all transactions.

```
CREATE REPLICATION R
       ELEMENT WEST TABLE CUSTOMERSWEST
         MASTER WEST ON "SERVERWEST"
         SUBSCRIBER ROUNDUP ON "SERVERROUNDUP"
            RETURN RECEIPT
       ELEMENT EAST TABLE CUSTOMERSEAST
         MASTER EAST ON "SERVEREAST"
         SUBSCRIBER ROUNDUP ON "SERVERROUNDUP"
            RETURN RECEIPT;
```

Example5.46 Replicate the contents of the REPL.TAB table from the CENTRALDS data store to the PROPDS data store, which propagates the changes to the BACKUP1DS and BACKUP2DS data stores.

```
CREATE REPLICATION REPL.PROPAGATOR
       ELEMENT A TABLE REPL.TAB
         MASTER CENTRALDS ON "FINANCE"
         SUBSCRIBER PROPRDS ON "NETHANDLER"
       ELEMENT B TABLE REPL.TAB
         PROPAGATOR PROPRDS ON "NETHANDLER"
         SUBSCRIBER BACKUP1DS ON "BACKUPSYSTEM1"
```

```
BACKUP2DS ON "BACKUPSYSTEM2";
```

Example 5.47 Bi-directionally replicate the contents of the REPL.ACCOUNTS table between the EASTDS and WESTDS data stores. Each data store is both a master and a subscriber for the REPL.ACCOUNTS table.

Because the REPL.ACCOUNTS table can be updated on either the EASTDS or WESTDS data store, it includes a timestamp column (TSTAMP). The CHECK CONFLICTS clause establishes automatic timestamp comparison to detect any update conflicts between the two data stores. In the event of a comparison failure, the entire transaction that includes an update with the older timestamp is rolled back (discarded).

```
CREATE REPLICATION REPL.R1
ELEMENT ELEM_ACCOUNTS_1 TABLE REPL.ACCOUNTS
   CHECK CONFLICTS BY ROW TIMESTAMP
      COLUMN TSTAMP
      UPDATE BY SYSTEM
      ON EXCEPTION ROLLBACK
   MASTER WESTDS ON "WESTCOAST"
   SUBSCRIBER EASTDS ON "EASTCOAST"
ELEMENT ELEM_ACCOUNTS_2 TABLE REPL.ACCOUNTS
   CHECK CONFLICTS BY ROW TIMESTAMP
      COLUMN TSTAMP
      UPDATE BY SYSTEM
      ON EXCEPTION ROLLBACK
   MASTER EASTDS ON "EASTCOAST"
   SUBSCRIBER WESTDS ON "WESTCOAST";
```

Example 5.48 Replicate the contents of the REPL.ACCOUNTS table from the ACTIVEDS data store to the BACKUPDS data store, using the return twosafe service, and using TCP/IP port 40000 on ACTIVEDS and TCP/IP port 40001 on BACKUPDS. The transactions on ACTIVEDS need to be committed whenever possible, so configure replication so that the transaction is committed even after a replication timeout using LOCAL COMMIT ACTION, and so that the return twosafe service is disabled when replication is stopped. To avoid significant delays in the application if the connection to the BACKUPDS data store is interrupted, configure the return service to be disabled after five transactions have timed out, but also configure the return service to be re-enabled when the BACKUPDS data store's replication agent responds in under 100 milliseconds. Finally, the bandwidth between data stores is limited, so configure replication to compress the data when it is replicated from the ACTIVEDS data store.

```
CREATE REPLICATION REPL.R
ELEMENT ELEM_ACCOUNTS_1 TABLE REPL.ACCOUNTS
   MASTER ACTIVEDS ON "ACTIVE"
   SUBSCRIBER BACKUPDS ON "BACKUP"
```

```
      RETURN TWOSAFE
ELEMENT ELEM_ACCOUNTS_2 TABLE REPL.ACCOUNTS
   MASTER ACTIVEDS ON "ACTIVE"
   SUBSCRIBER BACKUPDS ON "BACKUP"
      RETURN TWOSAFE
STORE ACTIVEDS ON "ACTIVE"
   PORT 40000
   LOCAL COMMIT ACTION COMMIT
   RETURN SERVICES OFF WHEN REPLICATION STOPPED
   DISABLE RETURN SUBSCRIBER 5
   RESUME RETURN 100
   COMPRESS TRAFFIC ON
STORE BACKUPDS ON "BACKUP"
   PORT 40001;
```

See Also

“ALTER ACTIVE STANDBY PAIR” on page 164
“ALTER REPLICATION” on page 170
“CREATE ACTIVE STANDBY PAIR” on page 204
“DROP ACTIVE STANDBY PAIR” on page 276
“DROP REPLICATION” on page 281

CREATE SEQUENCE

The CREATE SEQUENCE statement creates a new sequence number generator that can subsequently be used by multiple users to generate unique integers. You use the CREATE SEQUENCE statement to define the initial value of the sequence, define the increment value, the maximum or minimum value and determine if the sequence "cycles."

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL Syntax

CREATE SEQUENCE [*Owner.*]*SequenceName*
[INCREMENT BY *IncrementValue*]
[MINVALUE *MinimumValue*]
[MAXVALUE *MaximumValue*]
[CYCLE]
[CACHE *CacheValue*]
[START WITH *StartValue*]

Parameters

The CREATE SEQUENCE statement has the parameters:

SEQUENCE [*Owner.*]*SequenceName*	Name of the sequence number generator.
INCREMENT BY *IncrementValue*	The incremental value between consecutive numbers. This value can be either a positive or negative integer. It cannot be 0. If the value is positive, it is an ascending sequence. If the value is negative, it is descending. The default value is 1. In a descending sequence, the range starts from MAXVALUE to MINVALUE, and vice versa for ascending sequence.
MINVALUE *MinimumValue*	Specifies the minumum value for the sequence. The default minimum value is 1.
MAXVALUE *MaximumValue*	The largest possible value for an ascending sequence, or the starting value for a descending sequence. The default maximum value is (2^{63}) -1, which is the maximum of BIGINT.

CYCLE	Indicates that the sequence number generator continues to generate numbers after it reaches the maximum or minimum value. By default, sequences do not cycle. Once the number reaches the maximum value in the ascending sequence, the sequence wraps around and generates numbers from its minimum value. For a descending sequence, when the minimum value is reached, the sequence number wraps around, beginning from the maximum value. If CYCLE is not specified, the sequence number generator stops generating numbers when the maximum/minimum is reached and TimesTen returns an error.
CACHE *CacheValue*	CACHE indicates the range of numbers that are cached each time. When a restart occurs, unused cached numbers will be lost. If you specify a *CacheValue* of 1, then each use of the sequence results in an update to the database; Larger cache values result in fewer changes to the database and less overhead. The default is 20.
START WITH *StartValue*	Specifies the first sequence number to be generated. Use this clause to start an ascending sequence at a value that is greater than the minumum value or to start a descending sequence at a value less than the maximum. The *StartValue* must be greater or equal *MinimumValue* and *StartValue* must be less than or equal to *MaximumValue*.

Description

- All parameters in the CREATE SEQUENCE statement must be integer values.
- If you do not specify a value in the parameters, TimesTen defaults to an ascending sequence that starts with 1, increments by 1, has the default maximum value and does not cycle.
- There is no “ALTER SEQUENCE” statement in TimesTen. To alter a sequence, use the DROP SEQUENCE statement and then create a new sequence with the same name. For example, to change the MINVALUE, drop the sequence and recreate it with the same name and with the desired MINVALUE.

Incrementing SEQUENCE values with CURRVAL and NEXTVAL

To refer to the SEQUENCE values in a SQL statement, use CURRVAL and NEXTVAL.

- CURRVAL returns the value of the last call to NEXTVAL if there is one in the current session, otherwise it returns an error.
- NEXTVAL increments the current sequence value by the specified increment and returns the value for each row accessed.
- NEXTVAL and CURRVAL can be used in the:

  - *SelectList* of a SELECT statement, but not the *SelectList* of a subquery
  - *SelectList* of an INSERT SELECT
  - SET clause of an UPDATE statement
- In a single SQL statement with multiple NEXTVAL references, Timesten only increments the sequence once, returning the same value for all occurrences of NEXTVAL.
- If a SQL statement contains both NEXTVAL and CURRVAL, NEXTVAL is executed first. CURRVAL and NEXTVAL have the same value in that SQL statement.
- The current value of a sequence is a connection-specific value. If there are two concurrent connections to the same data store, each connection has its own CURRVAL of the same sequence set to its last NEXTVAL reference.
- In the case of recovery, sequences are not rolled back. It is possible that the range of values of a sequence can have gaps. Each sequence value is still unique.
- When the maximum value is reached, SEQUENCE either wraps or issues an error statement, depending on the value of the CYCLE option of the CREATE SEQUENCE.

Note: Sequences with the CYCLE attribute cannot be replicated.

Example5.49

```
CREATE SEQUENCE mysequence INCREMENT BY 1 MINVALUE 2
       MAXVALUE 1000;
```

Example5.50 This example assumes that tab1 has 1 row in the table and that CYCLE is used:

```
CREATE SEQUENCE s1 MINVALUE 2 MAXVALUE 4 CYCLE;

SELECT s1.NEXTVAL FROM tab1;
/* Returns the value of 2; */

SELECT s1.NEXTVAL FROM tab1;
/* Returns the value of 3; */

SELECT s1.NEXTVAL FROM tab1;
/* Returns the value of 4; */
```

After the maximum value is reached, the cycle starts from the minimum value for an ascending sequence.

```
SELECT s1.NEXTVAL FROM tab1;
/* Returns the value of 2; */
```

Example5.51 To create a sequence and generate a sequence number:

```
CREATE SEQUENCE seq INCREMENT BY 1;
INSERT INTO student VALUES (seq.NEXTVAL, 'Sally');
```

Example5.52 To use a sequence in an UPDATE SET clause:

```
UPDATE student SET studentno = seq.NEXTVAL WHERE name = 'Sally';
```

Example5.53 To use a sequence in a query:

```
SELECT seq.CURRVAL FROM student;
```

See Also "DROP SEQUENCE" on page 280

CREATE TABLE

The CREATE TABLE statement defines a table.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

The syntax for a persistent table is:

```
CREATE TABLE [Owner.]TableName
(
    {{ColumnDefinition} [,...]
     [PRIMARY KEY (ColumnName [,...]) |
     [[CONSTRAINT ForeignKeyName]
        FOREIGN KEY ([ColumnName] [,...])
        REFERENCES RefTableName
            [(ColumnName [,...])] [ON DELETE CASCADE]] [...]
     }
)
[UNIQUE HASH ON (HashColumnName [,...])
    PAGES = PrimaryPages]
[AGING  {LRU|
          USE ColumnName
              LIFETIME Num1 {MINUTE[S] | HOUR[S] |DAY[S]}
                  [CYCLE Num2 {MINUTE[S] |HOUR[S] |DAY[S]}]
        }[ON|OFF]
]
```

The syntax for a temporary table is:

```
CREATE GLOBAL TEMPORARY TABLE [Owner.]TableName
(
    {{ColumnDefinition} [,...]
     [PRIMARY KEY (ColumnName [,...]) |
     [[CONSTRAINT ForeignKeyName]
        FOREIGN KEY ([ColumnName] [,...])
        REFERENCES RefTableName
            [(ColumnName [,...])] [ON DELETE CASCADE]] [...]
      }
)
[UNIQUE HASH ON (HashColumnName [,...])
```

PAGES = *PrimaryPages*]
[ON COMMIT { DELETE | PRESERVE } ROWS]

Parameters The CREATE TABLE statement has the parameters:

[*Owner.*]*TableName*	Name to be assigned to the new table. Two tables cannot have the same owner name and table name. If you do not specify the owner name, your login name becomes the owner name for the new table. Owners of tables in TimesTen are determined by the user ID settings or login names. Oracle table owner names must always match TimesTen table owner names. For rules on creating names, see "Basic names" on page 67.
GLOBAL TEMPORARY	Specifies that the table being created is a temporary table. A temporary table is similar to a persistent table but it is effectively materialized only when referenced in a connection. A global temporary table definition is persistent and is visible to all connections, but the table instance is local to each connection. It is created when a command referencing the table is compiled for a connection and dropped when the connection is disconnected. All instances of the same temporary table have the same name but they are identified by an additional connection ID together with the table name. Global temporary tables are allocated in temp space. The contents of a temporary table cannot be shared between connections. Each connection sees only its own content of the table and compiled commands that reference temporary tables are not shared among connections. Temporary tables cannot be used as part of a cache group or a replication scheme. Temporary tables are automatically excluded when DATASTORE level replication is defined. A cache group table cannot be defined as a temporary table. Changes to temporary tables cannot be tracked with XLA. Operations on temporary tables do generate log records; the amount of log they generate is less than for permanent tables. Local temporary tables are not supported.
ColumnDefinition	An individual column in a table. Each table must have at least one column. See "Column Definition" on page 257.

ColumnName	Name of the column(s) that forms the primary key for the table to be created. Up to 16 columns can be specified for the primary key. For a foreign key, the *ColumnName* is optional. If not specified for a foreign key, the reference is to the parent table's primary key.
PRIMARY KEY	PRIMARY KEY may only be specified once in a table definition. It provides a way of identifying one or more columns that, together, form the primary key of the table. The contents of the primary key have to be unique and NOT NULL. Cannot specify a column as both UNIQUE and a single column PRIMARY KEY.
CONSTRAINT *ForeignKeyName*	Specifies an optional user-defined name for a foreign key. If not provided by the user, the system provides a default name.

FOREIGN KEY	This specifies a foreign key constraint between the new table and the referenced table identified by *RefTableName*. There are two lists of columns specified in the foreign key constraint. Columns in the first list are columns of the new table and are called the referencing columns. Columns in the second list are columns of the referenced table and are called referenced columns. These two lists must match in data type, including length, precision and scale. The referenced table must already have a primary key or unique index on the referenced column. The column name list of referenced columns is optional. If omitted, the primary index of *RefTableName* is used. The declaration of a foreign key creates a T-tree index on the referencing columns. The user cannot drop the referenced table or its referenced index until the referencing table is dropped. The foreign key constraint asserts that each row in the new table must match a row in the referenced table such that the contents of the referencing columns are equal to the contents of the referenced columns. Any INSERT, DELETE or UPDATE statements that violate the constraint return TimesTen error 3001. TimesTen supports SQL-92 “NO ACTION” update and delete rules and ON DELETE CASCADE. Foreign key constraints are not deferrable. A foreign key can be defined on a global temporary table, but it can only reference a global temporary table. If a parent table is defined with COMMIT DELETE, the child table must also have the COMMIT DELETE attribute. A foreign key cannot reference an active parent table. An active parent table is one that has some instance materialized for a connection.
[ON DELETE CASCADE]	Enables the ON DELETE CASCADE referential action. If specified, when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign key values are also deleted.
UNIQUE	UNIQUE provides a way of identifying a column where each row must contain a unique value.
UNIQUE HASH ON	Hash index for the table. Only unique hash indexes are created. This parameter is used for equality predicates. See “Description” for more detail. UNIQUE HASH ON requires that a primary key be defined.

HashColumnName	Column defined in the table that is to participate in the hash key of this table. The columns specified in the hash index must be identical to the columns in the primary key.
PrimaryPages	Specifies the expected number of pages in the table. This number affects the number of buckets that are allocated for the table's hash index. The minimum is 1. If your estimate is too small, performance is degraded.
[ON COMMIT { DELETE | PRESERVE } ROWS]	The optional statement specifies whether to delete or preserve rows when a transaction that touches a global temporary table is committed. If not specified, the rows of the temporary table are deleted.
AGING LRU [ON | OFF]	If specifed, rows of the table are aged using the LRU policy. By removing the approximate least recently used data, LRU aging keeps only the recently used data in cache; Data is removed if the data store space in-use exceeds a specified threshold value. LRU aging frees up space occupied by infrequently used data. The LRU aging policy is defined when you specify the AGING LRU clause. ON or OFF settings refer to the aging state. The OFF setting indicates the LRU aging policy is defined, but the aging state is disabled. Because the aging state is OFF, aging is not done automatically. The ON setting indicates the LRU aging policy is defined and the aging state is enabled. Because the aging state is ON, aging is done automatically. The default is ON. For more information on LRU aging, see "Aging:" on page 260. For detailed information on LRU aging, see "Implementing aging in your tables".

AGING USE *ColumnName*... [ON | OFF]	If specified, rows of the table are aged using the time-based aging policy. Time-based aging allows you to specify the lifetime data is kept in cache. The time unit is expressed in terms of minutes, hours, or days. In order to use time-based aging, you must define a column in the table you are aging. The value of this column is subtracted from SYSDATE, truncated using the specified unit (minute, hour, day) and then compared to the LIFETIME value you specify. If the result is *greater than* the LIFETIME value, then the row is a candidate for aging. Define the column as NOT NULL and of type TIMESTAMP. The values of the timestamp columns are updated by your applications. If the value of this column is unknown for some rows, and you do not want the rows to be aged, define the column with a large default value. *ColumnName* refers to the timestamp column you defined on the table you are aging. The time-based aging policy is defined when you specify the AGING USE clause. ON or OFF settings refer to the aging state. The OFF setting indicates the time-based aging policy is defined, but the aging state is disabled. Because the aging state is OFF, aging is not done automatically. The ON setting indicates the time-based aging policy is defined and the aging state is enabled. Because the aging state is ON, aging is done automatically. The default is ON. Time-based aging allows you to implement sliding windows for your application. For more information on time-based aging, see "Aging:". For detailed information on time-based aging, see "Implementing aging in your tables".

LIFETIME *Num1* {MINUTE[S]|HOUR[S] | DAY[S]}	This required clause is specific to time-based aging; It follows the AGING USE *ColumnName* clause. LIFETIME is the minimum amount of time data is kept in cache. For example, keep data in cache for *Num1* unit of time where unit can be minutes or hours or days. This means rows that exceed the specified LIFETIME value are aged out (deleted from the table). *Num1* must be a positive integer constant; Its value indicates the amount of time in minutes, hours, or days that the rows should stay in cache. The concept of time resolution is supported. The time-unit specified for the lifetime of data is the resolution granularity for which data should be kept in cache. If, for example, DAYS is specified as the time resolution, then all rows whose timestamp belongs to the same day are aged out at the same time. If HOURS is specified as the time resolution, then all rows with timestamp values within that hour are aged out at the same time. Thus, a LIFETIME of *3* DAYS is different than a LIFETIME of *72* HOURS (3*24) or a LIFETIME of *432* MINUTES (3*24*60). Days represent all days of the week including weekends and holidays. For more information, see Example 5.66.
[CYCLE *Num2* {MINUTE[S] |HOUR[S] | DAY[S]}]	This optional clause is specific to time-based aging and follows the LIFETIME clause. It indicates how often the system should examine rows to see if data exceeds the specified LIFETIME values and thus should be aged out (deleted). *Num2* must be a positive integer constant. If you do not specify the CYCLE clause, then the default value is 5 minutes. This means that if aging is enabled, rows are examined every 5 minutes starting as soon as possible. If you specify 0 for *Num2*, then aging is continuous and the aging thread never sleeps.

Column Definition

SQL syntax

ColumnName ColumnDataType
[DEFAULT DefaultVal]
[[NOT] INLINE]
[PRIMARY KEY | UNIQUE |
NULL [UNIQUE] |
NOT NULL [PRIMARY KEY | UNIQUE]]

[Parameters

The column definition has the parameters:

ColumnName	Name to be assigned to one of the columns in the new table. No two columns in the table can be given the same name. You can define a maximum of 255 columns in a table.
DEFAULT *DefaultVal*	Indicates that if a value is not specified for the column in an INSERT, the default value *DefaultVal* is inserted into the column. The default value specified must have a compatible type with the column's data type. A default value can be as long as the data type of the associated column allows. Legal data types for *DefaultVal* can be one of: NULL *ConstantValue* -- See "Constants" on page 82 SYSDATE and GETDATE USER CURRENT_USER SYSTEM_USER If default value is one of the users, the column's data type must be either CHAR or VARCHAR and the column's width must be at least 30 characters.
ColumnDataType	Type of data the column can contain. Some data types require that you indicate a length. See Chapter 1, "Data Types," for the data types that can be specified.
INLINE | NOT INLINE	By default, variable-length columns whose declared column length is > 128 bytes are stored out of line. Variable-length columns whose declared column length is <= 128 bytes are stored inlined. The default behavior can be overridden during table creation through the use of the INLINE and NOT INLINE keywords.
NULL	Indicates that the column can contain NULL values.
NOT NULL	Indicates that the column cannot contain NULL values. If NOT NULL is specified, any statement that attempts to place a NULL value in the column is rejected.
UNIQUE	A unique constraint placed on the column. No two rows in the table may have the same value for this column. TimesTen creates a unique T-tree index to enforce uniqueness. This means that a column with a unique constraint can use more memory and time during execution than a column without the constraint. Cannot be used with PRIMARY KEY.
PRIMARY KEY	A unique NOT NULL constraint placed on the column. No two rows in the table may have the same value for this column.; Cannot be used with UNIQUE.

Description

- The TimesTen system currently supports one hash index per table. Furthermore, hash indexes are only supported over the primary key of tables.
- By default, a T-tree index is created to enforce the primary key. Use the UNIQUE HASH clause to specify a hash index for the primary key.
  - If your application performs range queries over a table's primary key, then choose a T-tree index for that table by omitting the UNIQUE HASH clause.
  - If, however, your application performs only exact match lookups on the primary key, then a hash index may offer better response time and throughput. In such a case, specify the UNIQUE HASH clause.
- Use ALTER TABLE to change the representation of the primary key index for a table.
- A hash index is created with a fixed number of buckets that remains constant for the life of the table or until the hash index is resized using an ALTER TABLE statement to change hash index size. Fewer buckets in the hash index results in more hash collisions. More buckets reduce collisions but can waste memory. Hash key comparison is a fast operation, so a small number of hash collisions does not cause a performance problem for TimesTen.

  The bucket count is derived as the ratio of the maximum table cardinality, derived from the value of PAGES, to the value 20.

  To ensure that the hash index is sized correctly, an application must indicate the expected size of the table. This is done with the PAGES parameter. The PAGES parameter should be the expected number of rows in the table, divided by 256. (Since 256 is the number of rows TimesTen stores on each page, the value provided is the expected number of pages in the table.) The application may specify a larger value for PAGES, and therefore fewer rows per bucket on average, if memory use is not an overriding concern.
- At most 16 columns are allowed in a hash key.
- All columns participating in the primary key are non-NULL.
- A unique hash index can be specified only for the primary key.
- A PRIMARY KEY that is specified in the *ColumnDefinition* can only be specified for one column.
- PRIMARY KEY cannot be specified in both the *ColumnDefinition* parameters and CREATE TABLE parameters.
- For both primary key and foreign key constraints, duplicate column names are not allowed in the constraint column list.
- You cannot create a table that has a foreign key referencing a cached table.
- UNIQUE column constraint and default column values are not supported with materialized views.

- To change the ON DELETE CASCADE triggered action, drop then redefine the foreign key constraint.
- If access control is disabled, no privilege is necessary to execute a DELETE statement which triggers the ON DELETE CASCADE action. Otherwise, WRITE privileges are required on all tables affected by the ON DELETE CASCADE action.
- ON DELETE CASCADE is supported on “detail tables” of a materialized view. If you have a materialized view defined over a child table, a deletion from the parent table will cause cascaded deletes in the child table. This, in turn, will trigger changes in the materialized view.
- The total number of rows reported by the DELETE statement does not include rows deleted from child tables as a result of the ON DELETE CASCADE action.
- For ON DELETE CASCADE: Since different paths may lead from a parent table to a child table, the following rule is enforced:
  - Either all paths from a parent table to a child table are ‘delete’ paths or all paths from a parent table to a child table are ‘do not delete’ paths. Specify ON DELETE CASCADE on all child tables on the ‘delete’ path.
  - This rule does not apply to paths from one parent to different children or from different parents to the same child.
- For ON DELETE CASCADE: A second rule is also enforced:
  - If a table is reached by a ‘delete’ path, then all its children are also reached by a ‘delete’ path.
- For ON DELETE CASCADE with replication, the following restrictions apply:
  - The foreign keys specified with ON DELETE CASCADE must match between the Master and subscriber for replicated tables. Checking will be done at runtime. If there is an error, the receiver thread will stop working.
  - All tables in the delete cascade tree have to be replicated if any table in the tree is replicated. This restriction will be checked when the replication scheme is created or when a foreign key with ON DELETE CASCADE is added to one of the replication tables. If an error is found, the operation is aborted. You may be required to drop the replication scheme first before trying to change the foreign key constraint.
  - You must stop the replication agent before adding or dropping a foreign key on a replicated table.

Aging:

- Aging: Usage
  - When you define an aging policy for a table, you define:
    1. The type of aging: LRU aging or time-based.

2. The state of aging either ON (enabled) or OFF (disabled).
3. For time-based aging, the timestamp column, LIFETIME, and CYCLE.

Note: LRU attributes are defined by calling procedure ttAgingLruConfig. LRU attributes are not defined at the SQL level. If you do not call this procedure, then the default values are used.

– Define an aging policy on either an existing table or a new table. For an existing table, use the ALTER TABLE... ADD AGING clause. For a new table, use the CREATE TABLE... AGING clause.

  To add LRU aging on an existing table:

  ALTER TABLE... ADD AGING LRU [ON|OFF]

  To add time-based aging on an exising table:

  ALTER TABLE... ADD AGING USE ...

  To add LRU aging on a new table:

  CREATE TABLE... AGING LRU [ON|OFF]

  See Example 5.67.

  To add time-based aging on a new table:

  CREATE TABLE... AGING USE....

  See Example 5.68.

– After you have defined the aging policy for the table, you cannot alter the aging policy. This means you cannot change the policy from LRU to time-based or from time-based to LRU. For time-based aging, you cannot change the timestamp column. Essentially, you cannot specify the ALTER TABLE...ADD AGING clause on a table with an existing aging policy.

– To change the aging policy, drop aging then redefine the aging policy:

  ALTER TABLE... DROP AGING

  See Example 5.69.

– After you have defined the aging policy for the table, you can change the aging state; that is you can change the state defined in the aging policy from ON (automatic aging enabled) to OFF (automatic aging disabled) or from OFF (automatic aging disabled) to ON (automatic aging enabled). To do this:

  ALTER TABLE... SET AGING [ON | OFF]

– For time-based aging, you can change the LIFETIME and CYCLE:

  ALTER TABLE...SET AGING LIFETIME *Num1*
  {MINUTES |HOURS|DAYS}
  ALTER TABLE... SET AGING CYCLE *Num2*
  {MINUTES |HOURS|DAYS}

  For more information, see **ALTER TABLE**.

- Aging: Built-in Procedures
  - Use ttAgingLRUConfig to specify the following properties for LRU aging:

    1. LowUsageThreshold: (percentage of Perm-size allocated)

    2. HighUsageThreshold: (percentage of Perm-size allocated)

    3. Aging Cycle: The number of minutes the aging thread sleeps before waking up.

  - The default property settings for LRU aging are:

    1. LowUsageThreshold: 80%

    2. HighUsageThreshold: 90%

    3. Aging Cycle: 1 minute

  - For LRU aging, the aging thread wakes up at the specified aging cycle and checks if the space usage is over the HighUsageThreshold; If it is not, it goes back to sleep; Otherwise, it begins removing the least used data until either the space usage is at or below the LowUsageThreshold percentage or there is no data left to remove.
  - ttAgingLRUConfig applies to *all* tables with LRU aging. Arguments to this procedure can be omitted; If omitted, the previous setting is retained. Arguments are positive integers.
  - For more information on the ttAgingLRUConfig procedure, see "ttAgingLRUConfig".
  - Use ttAgingScheduleNow to schedule the aging process as soon as possible (the process starts when the subdaemon awakes). When you call this procedure, the aging process is started for a table regardless if its aging setting is OFF (disabled) or ON (enabled). The aging process is started only once; the previous aging state is unchanged. For example, if aging state is OFF when you call ttAgingScheduleNow, the aging process starts. When aging is complete, if your aging state is OFF, aging does not continue. To continue aging, you must call ttAgingScheduleNow again or change the aging state to ON.
  - If you call ttAgingScheduleNow and the aging setting is ON, the aging cycle is reset based on the time ttAgingScheduleNow was called.
  - ttAgingScheduleNow is useful if you wish to schedule the aging process as soon as possible or if you wish to start the aging process at a specified time (via an external scheduler e.g. cron job). In the latter case, set the aging state to OFF, and then call ttAgingScheduleNow to invoke aging.
  - To schedule aging on one table, call ttAgingScheduleNow specifying the name of the table and enclosing the table name in quotes. The specified table can be defined with either LRU aging or time-based aging:

    CALL ttAgingScheduleNow (‘TableName’)

  - To start immediate aging on all tables, including tables defined with both *LRU* aging and *time-based* aging, call ttAgingScheduleNow without any arguments.

    ```
    CALL ttAgingScheduleNow ( );
    ```

  - For more information on the ttAgingScheduleNow procedure, see "ttAgingScheduleNow".

- Aging: Specifics
  - LRU and time-based aging can be combined in one system. If you use only LRU aging , the aging thread wakes up based on the cycle specified for the whole data store; If you use only time-based aging, the aging thread wakes up based on an optimal frequency. This frequency is determined by the values specified in the CYCLE clause for all tables. If you use both LRU and time-based aging, then the thread wakes up based on a combined consideration of both types.
  - You can implement sliding windows with time-based aging; The table contains the last *Num1* time unit of data. Sliding OUT is handled automatically by aging. Data is brought into cache, and past data that falls out of the window is aged out. As the window changes, data is not guaranteed to be deleted; data is deleted when the aging thread wakes up.
- Aging: For LRU aging, the following rules determine if a row is accessed or referenced:
  - Any rows used to build the result set of a SELECT statement.
  - Any rows used to build the result set of an INSERT SELECT statement.
  - Any rows that are about to be updated or deleted.
- Aging: LRU and compiled commands:
  - Compiled commands are marked invalid and need recompilation when you either drop LRU aging from or add LRU aging to tables that are referenced in the commands.
- Aging: Foreign Key Issues
  - Tables that are related via foreign keys must have the same aging policy.
  - For LRU aging, if a child row is not a candidate for aging, then this child row is not deleted; Nor will the parent row be deleted; ON DELETE CASCADE settings are ignored.
  - For time-based aging, if a parent row is a candidate for aging, then all child rows are deleted. ON DELETE CASCADE (whether specified or not) is ignored.
- Aging: Restrictions
  - LRU aging and time-based aging are not supported on detail tables of materialized views.

- LRU aging and time-based aging are not supported on global temporary tables.
- You cannot drop the timestamp column that is used for time-based aging.
- The aging policy and aging state must be the same in all sites of replication.

Example5.54 A T-tree index is created on `PartNumber` because it is the primary key.

```
Command> CREATE TABLE Price
> (PartNumber INTEGER NOT NULL PRIMARY KEY,
>  VendorNumber INTEGER NOT NULL,
>  VendPartNum CHAR(20) NOT NULL,
>  UnitPrice DECIMAL(10,2),
>  DeliveryDays SMALLINT,
>  DiscountQty SMALLINT);
Command> indexes price;

Indexes on table SAMPLEUSER.PRICE:
  PRICE: unique T-tree index on columns:
    PARTNUMBER
  1 index found.

1 table found.
```

Example5.55 A hash index is created on column `ClubName`, the primary key.

```
CREATE TABLE Recreation.Clubs
(ClubName CHAR(15) NOT NULL PRIMARY KEY,
 ClubPhone SMALLINT,
 Activity CHAR(18))
UNIQUE HASH ON (ClubName) PAGES = 30;
```

Example5.56 A T-tree index is created on the two columns `MemberName` and `Club` because together they form the primary key.

```
Command> CREATE TABLE Recreation.Members
> (MemberName CHAR(20) NOT NULL,
>  Club CHAR(15) NOT NULL,
>  MemberPhone SMALLINT,
>  PRIMARY KEY (MemberName, Club));
Command> indexes recreation.members;

Indexes on table RECREATION.MEMBERS:
  MEMBERS: unique T-tree index on columns:
    MEMBERNAME
    CLUB
  1 index found.
```

```
1 table found.
```

Example 5.57 No hash index is created on the table `Recreation.Events`.

```
CREATE TABLE Recreation.Events
(SponsorClub CHAR(15),
 Event CHAR(30),
 Coordinator CHAR(20),
 Results VARBINARY(10000));
```

Example 5.58 A hash index is created on the column `VendorNumber`.

```
CREATE TABLE Purchasing.Vendors
(VendorNumber INTEGER NOT NULL PRIMARY KEY,
 VendorName CHAR(30) NOT NULL,
 ContactName CHAR(30),
 PhoneNumber CHAR(15),
 VendorStreet CHAR(30) NOT NULL,
 VendorCity CHAR(20) NOT NULL,
 VendorState CHAR(2) NOT NULL,
 VendorZipCode CHAR(10) NOT NULL,
 VendorRemarks VARCHAR(60))
UNIQUE HASH ON (VendorNumber) PAGES = 101;
```

Example 5.59 A hash index is created on the columns `MemberName` and `Club` because together they form the primary key.

```
CREATE TABLE Recreation.Members
(MemberName CHAR(20) NOT NULL,
 Club CHAR(15) NOT NULL,
 MemberPhone SMALLINT,
 PRIMARY KEY (MemberName, Club))
UNIQUE HASH ON (MemberName, Club) PAGES = 100;
```

Example 5.60 A hash index is created on the columns FirstName and LastName because together they form the primary key in the table Authors.

A foreign key is created on the columns AuthorFirstName and AuthorLastName in the table Books that references the primary key in the table Authors.

```
CREATE TABLE Authors
(FirstName VARCHAR(255) NOT NULL,
 LastName VARCHAR(255) NOT NULL,
 Description VARCHAR(2000),
 PRIMARY KEY (FirstName, LastName))
UNIQUE HASH ON (FirstName, LastName) PAGES=20;

CREATE TABLE Books
(Title VARCHAR(100),
```

```
 AuthorFirstName VARCHAR(255),
 AuthorLastName VARCHAR(255),
 Price DECIMAL(5,2),
 FOREIGN KEY (AuthorFirstName, AuthorLastName)
 REFERENCES Authors(FirstName, LastName));
```

Example5.61 The following statement overrides the default character of VARCHAR columns and creates a table where one VARCHAR (10) column is NOT INLINE and one VARCHAR (144) is INLINE:

```
CREATE TABLE t1 (c1 VARCHAR(10) NOT INLINE NOT NULL,
c2 VARCHAR(144) INLINE NOT NULL);
```

Example5.62 The following statement creates a table with a UNIQUE column for book titles:

```
CREATE TABLE Books
(Title VARCHAR(100) UNIQUE,
 AuthorFirstName VARCHAR(255),
 AuthorLastName VARCHAR(255),
 Price DECIMAL(5,2),
 FOREIGN KEY (AuthorFirstName, AuthorLastName)
 REFERENCES Authors(FirstName, LastName));
```

Example5.63 The following statement creates a table with a default value of 1 on column `x1` and a default value of `SYSDATE` on column `d`:

```
CREATE TABLE t1 (x1 INT DEFAULT 1, d TIMESTAMP DEFAULT SYSDATE);
```

Example5.64 The example CREATEs table TtreeEx; Col1 is defined as the primary key. A T-tree index is created by default.

```
Command> create table TtreeEx (Col1 TT_INTEGER PRIMARY KEY);
Command> INDEXES TtreeEx;

Indexes on table SAMPLEUSER.TTREEEX:
  TTREEEX: unique T-tree index on columns:
    COL1
  1 index found.

1 table found.
```

Example5.65 The following statement illustrates the use of the ON DELETE CASCADE clause for parent/child tables of the HR schema. Tables with foreign keys have been altered to enable ON DELETE CASCADE.

```
ALTER TABLE countries
ADD CONSTRAINT countr_reg_fk
         FOREIGN KEY (region_id)
```

```
            REFERENCES regions(region_id) ON DELETE CASCADE;
ALTER TABLE locations
     ADD CONSTRAINT loc_c_id_fk
          FOREIGN KEY (country_id)
         REFERENCES countries(country_id) ON DELETE CASCADE;
ALTER TABLE departments
     ADD CONSTRAINT dept_loc_fk
          FOREIGN KEY (location_id)
         REFERENCES locations (location_id) ON DELETE CASCADE;
ALTER TABLE employees
    ADD CONSTRAINT     emp_dept_fk
                      FOREIGN KEY (department_id)
                       REFERENCES departments ON DELETE CASCADE;
ALTER TABLE employees
   ADD CONSTRAINT     emp_job_fk
                      FOREIGN KEY (job_id)
                       REFERENCES jobs (job_id);
ALTER TABLE job_history
     ADD CONSTRAINT     jhist_job_fk
                      FOREIGN KEY (job_id)
                      REFERENCES jobs;
ALTER TABLE job_history
    ADD CONSTRAINT     jhist_emp_fk
                      FOREIGN KEY (employee_id)
                      REFERENCES employees ON DELETE CASCADE;
ALTER TABLE job_history
    ADD CONSTRAINT     jhist_dept_fk
                      FOREIGN KEY (department_id)
                      REFERENCES departments ON DELETE CASCADE;
    ;
```

Example5.66 The following is an example of how time resolution works with aging.

If Lifetime is 3 days (resolution is in days):

```
If (SYSDATE - ColumnValue) <= 3, don't age
If (SYSDATE - ColumnValue) > 3, then candidate to age
```

```
If (SYSDATE - ColumnValue) = 3 days, 22 hours:

Row will not be aged out if you specified a lifetime of
3 days.  Row will be aged out if you specified a lifetime
of 72 hours.
```

Example5.67 The following example creates a table with LRU aging. Aging state is ON by default.

```
CREATE TABLE AgingDemo
       (AgingId NUMBER NOT NULL PRIMARY KEY
```

```
       ,Name   VARCHAR2 (20)
      )
      AGING LRU;
Command> DESCRIBE AgingDemo;
Table USER.AGINGDEMO:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
  Aging lru on

1 table found.
(primary key columns are indicated with *)
```

Example5.68 The following example creates a table with time-based aging. Lifetime is 3 days; Cycle is not specified, so the default is 5 minutes; Aging state is OFF.

```
CREATE TABLE AgingDemo2
      (AgingId NUMBER NOT NULL PRIMARY KEY
       ,Name   VARCHAR2 (20)
       ,AgingColumn TIMESTAMP NOT NULL
      )
      AGING USE AgingColumn LIFETIME 3 DAYS OFF;
Command> DESCRIBE AgingDemo2;
Table USER.AGINGDEMO2:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
    AGINGCOLUMN                      TIMESTAMP (6) NOT NULL
  Aging use AGINGCOLUMN lifetime 3 days cycle 5 minutes off

1 table found.
(primary key columns are indicated with *)
```

Example5.69 The following example generates an error message. It illustrates that once you create an aging policy, you cannot change it. You must drop aging and redefine..

```
CREATE TABLE AgingDemo2
      (AgingId NUMBER NOT NULL PRIMARY KEY
       ,Name   VARCHAR2 (20)
       ,AgingColumn TIMESTAMP NOT NULL
      )
      AGING USE AgingColumn LIFETIME 3 DAYS OFF;
ALTER TABLE AgingDemo2
      ADD AGING LRU;
```

```
 2980: Cannot add aging policy to a table with an existing aging policy.
Have to drop the old aging first
The command failed.
```

DROP aging on the table and redefine with LRU aging.

```
ALTER TABLE AgingDemo2
       DROP AGING;
```

```
ALTER TABLE AgingDemo2
       ADD AGING LRU;
```

```
Command> DESCRIBE AgingDemo2;

Table USER.AGINGDEMO2:
  Columns:
   *AGINGID                          NUMBER NOT NULL
    NAME                             VARCHAR2 (20) INLINE
    AGINGCOLUMN                      TIMESTAMP (6) NOT NULL
  Aging lru on

1 table found.
(primary key columns are indicated with *)
```

See Also

“ALTER TABLE” on page 186
“DROP TABLE” on page 282
“TRUNCATE TABLE” on page 320
“UPDATE” on page 323

CREATE USER

The CREATE USER statement identifies a user to the TimesTen instance.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires instance administrator privileges. If your TimesTen instance does not use Access Control, this operation is not available. If Access Control is not enabled, TimesTen returns an error when this statement is called.

SQL syntax

CREATE USER *User* IDENTIFIED { BY '*Password*' | EXTERNALLY }

Parameters

The CREATE USER statement has the parameters:

User	Name of the user that is being added to the instance.
IDENTIFIED	Specifies how the TimesTen instance uniquely identifies the user.
BY *Password*	Internal users must be given a TimesTen password. To perform data store operations using an internal user name, the user must supply this password.
EXTERNALLY	Identifies the operating system user name *User* to the TimesTen instance. To perform data store operations as an external user, the accessing process must have had a TimesTen external user name created that matches the user name authenticated by the operating system or network. A password is not required by TimesTen since the user would have been authenticated by the operating system at login time.

Description

- Instance users may have internal user names or external user names.
  - Internal user names are defined strictly within a TimesTen instance.
  - External user names are defined by some external authority, such as the operating system. External user names cannot be assigned a TimesTen password.
- An internal user connected as *User* may execute this command to change their own TimesTen password. Passwords are case-sensitive.
- TimesTen instance users are user names that have been identified to the instance. User names are case-insensitive. They apply to all data stores in the instance.
- An internal user name takes precedence over an external user of the same name.
- Changes to user identification and privileges take place at the next connection time.

Example 5.70 To create the internal user TERRY with the password 'secret', use:

```
CREATE USER terry IDENTIFIED BY 'secret';
```

Example 5.71 To identify the external user PAT to the TimesTen instance, use:

```
CREATE USER pat IDENTIFIED EXTERNALLY;
```

See Also
"ALTER USER" on page 203
"DROP USER" on page 283
"GRANT" on page 287
"REVOKE" on page 300

CREATE VIEW

The CREATE VIEW statement creates a view of the tables specified in the *SelectQuery* clause. The original tables used to create a view are referred to as "detail" tables. The resulting table is referred to as a VIEW.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

CREATE VIEW *ViewName* AS *SelectQuery*

Parameters

The CREATE VIEW statement has the parameters:

ViewName	Name assigned to the new view.
SelectQuery	Selects column from the detail table(s) to be used in the view. Can also create indexes on the view.

Restrictions on the VIEW query

There are several restrictions on the query that is used to define the view.

- A SELECT * query in a view definition is expanded at view creation time. Any columns added after a view is created do not affect the view.
- The following cannot be used in a SELECT statement that is creating a view:
  - DISTINCT
  - FIRST
  - ORDER BY
  - Arguments
  - Temporary tables
- Each expression in the select list must have a unique name. A name of a simple column expression would be that column's name unless a column alias is defined. *RowId* is considered an expression and needs an alias.
- No SELECT FOR UPDATE or SELECT FOR INSERT statements can be used on a view.
- Certain TimesTen query restrictions are not checked when a non-materialized view is created. Views that violate those restrictions may be allowed to be created, but an error is returned when the view is referenced later in an executed statement.

Restrictions on the VIEW

- When a view is referenced in the FROM clause of a SELECT statement, its name is replaced by its definition as a derived table at parsing time. If it is not possible to merge all clauses of a view to the same clause in the original select

to form a legal query without the derived table, the content of this derived table is materialized. For example, if both the view and the referencing select specify aggregates, the view is materialized before its result can be joined with other tables of the select.

- A view cannot be dropped with a DROP TABLE statement. You must use the DROP VIEW statement.
- A view cannot be altered with an ALTER TABLE statement.
- Referencing a view can fail due to dropped or altered detail tables.

Example 5.72 Creates a (non-materialized) view from the table *t1*.

```
CREATE VIEW v1 AS SELECT * FROM t1;
```

Example 5.73 Creates a (non-materialized) view from an aggregate query on the table *t1*.

```
CREATE VIEW v1 (max1) AS SELECT max(x1) FROM t1;
```

See Also
"CREATE MATERIALIZED VIEW" on page 229
"CREATE TABLE" on page 251
"DROP VIEW" on page 284

DELETE

The DELETE statement deletes rows from a table.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges or data store object ownership.

SQL syntax

DELETE [FIRST *NumRows*] FROM [*Owner.*]*TableName*[*CorrelationName*] [WHERE *SearchCondition*]

Parameters

The DELETE statement has the parameters:

FIRST *NumRows*	Specifies the number of rows to delete. FIRST *NumRows* is not supported in subquery statements. *NumRows* must be either a positive INTEGER or a dynamic parameter placeholder. The syntax for a dynamic parameter placeholder is either ? or :*DynamicParameter*. The value of the dynamic parameter is supplied when the statement is executed.
[*Owner.*]*TableName* [*CorrelationName*]	Designates a table from which any rows satisfying the search condition are to be deleted. *Owner.*]*TableName* [*CorrelationName*] where, [*Owner.*]*TableName* identifies a table to be deleted. *CorrelationName* specifies a synonym for the immediately preceding table. When accessing columns of that table, use the correlation name instead of the actual table name within the DELETE statement. The correlation name must conform to the syntax rules for a basic name (see "Basic names" on page 67).
SearchCondition	Specifies which rows are to be deleted. If no rows satisfy the search condition, the table is not changed. If the WHERE clause is omitted, all rows are deleted. The search condition can contain a subquery.

Description

- If all the rows of a table are deleted, the table is empty but continues to exist until you issue a DROP TABLE statement.
- If a foreign key references the target table, then logging to disk must be enabled for constraint enforcement.
- The DELETE operation fails if it violates any foreign key constraint. See "CREATE TABLE" on page 251 for a description of the foreign key constraint.

Example 5.74 Rows for orders whose quantity is less than 50 are deleted.

```
DELETE FROM Purchasing.OrderItems
WHERE Quantity < 50;
```

Example 5.75 The following query deletes all the duplicate orders assuming that *id* is not a primary key:

```
DELETE FROM orders A
WHERE EXISTS (SELECT 1 FROM orders B
WHERE A.id = B.id and A.rowid < B.rowid);
```

Example 5.76 The following sequence of statements causes a foreign key violation.

```
CREATE TABLE Master
(Name CHAR(30), Id CHAR(4) NOT NULL PRIMARY KEY);

CREATE TABLE Details(MasterId CHAR(4),
Description VARCHAR(200), FOREIGN KEY (MasterId) REFERENCES
Master(Id));
INSERT INTO Master('Elephant', '0001');
INSERT INTO Details('0001', 'A VERY BIG ANIMAL');
DELETE FROM Master WHERE Id = '0001';
```

Example 5.77 If you attempt to delete a "busy" table, an error results. In this example, *t1* is a "busy" table that is a parent table with foreign key constraints based on it.

```
CREATE TABLE t1 (a INT NOT NULL, b INT NOT NULL,
       PRIMARY KEY (a));
CREATE TABLE t2 (c INT NOT NULL,
       FOREIGN KEY (c) REFERENCES t1(a));
INSERT INTO t1 VALUES (1,1);
INSERT INTO t2 VALUES (1);
DELETE FROM t1;
```

An error is returned:

```
SQL ERROR (3001):  Foreign key violation [TTFOREIGN_0] a row in
child table T2 has a parent in the delete range.
```

DROP ACTIVE STANDBY PAIR

This statement drops an active standby pair replication scheme.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges and WRITE privileges or data store object ownership.

SQL syntax

DROP ACTIVE STANDBY PAIR

Parameters

DROP ACTIVE STANDBY has no parameters.

Description

The active standby pair is dropped, but all objects such as tables, cache groups, and materialized views still exist on the node on which the statement was issued.

See Also

"ALTER ACTIVE STANDBY PAIR" on page 164
"CREATE ACTIVE STANDBY PAIR" on page 204

DROP CACHE GROUP

The DROP CACHE GROUP statement drops the table associated with the cache group, and removes the cache group definition from the CACHE_GROUP system table.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges or data store object ownership.

SQL syntax

DROP CACHE GROUP [*Owner.*]*GroupName*

Parameters

The DROP CACHE GROUP statement has the parameter:

[*Owner.*]*GroupName*	Name of the cache group to be deleted.

Description

- If you attempt to delete a cache group table that is in use, TimesTen returns an error.
- ASYNCHRONOUS WRITETHROUGH cache groups cannot be dropped while the replication agent is running.
- Automatically installed Oracle objects for AUTOREFRESH cache groups are uninstalled by the Cache agent. If the Cache agent is not running during the DROP CACHE GROUP operation, the Oracle objects are uninstalled on the next startup of the Cache agent.
- If you issue a DROP CACHE GROUP statement, and there is an autorefresh operation currently running, then:
  - If LockWait interval is 0, the DROP CACHE GROUP statement will fail with a lock timeout error.
  - If LockWait interval is non-zero, then the current autorefresh transaction is pre-empted (rolled back), and the DROP statement continues. This affects all cache groups with the same autorefresh interval.
-

Example 5.78

```
DROP CACHE GROUP WesternCustomers
```

See Also

"ALTER CACHE GROUP" on page 167
"CREATE CACHE GROUP" on page 211

DROP INDEX

The DROP INDEX statement deletes the specified index.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges or data store object ownership.

SQL syntax

DROP INDEX [*Owner.*]*IndexName* [FROM [*Owner.*]*TableName*]

Parameters

The DROP INDEX statement has the parameters:

[*Owner.*]*IndexName*	Name of the index to be dropped. It may include the name of the owner of the table that has the index.
[*Owner.*]*TableName*	Name of the table upon which the index was created.

Description

- If you attempt to drop a "busy" index—an index that is in use or that enforces a foreign key—an error results. To drop a foreign key, and consequently the index associated with it, use **ALTER TABLE**.
- If an index is created through a UNIQUE column constraint (See **CREATE TABLE**.), it can only be dropped by dropping the constraint with an **ALTER TABLE** DROP UNIQUE statement.
- If a DROP INDEX operation is or was active in an uncommitted transaction, other transactions doing DML operations that do not access that index are blocked.
- If an index is dropped, any prepared statement that uses the index is prepared again automatically the next time the statement is executed.
- If no table name is specified, the index name must be unique for the specified or implicit owner. The implicit owner, in the absence of a specified table or owner, is the current user running the program.
- If no index owner is specified and a table is specified, the default owner is the table owner.
- If a table is specified and no owner is specified for it, the default table owner is the current user running the program.
- The table and index owners must be the same.
- An index on a temporary table cannot be dropped by a connection if some other connection has some non-empty instance of the table.

Example 5.79

Drop index PartsOrderedIndex which is defined on table OrderItems using one of the following:

```
DROP INDEX PartsOrderedIndex
   FROM Purchasing.OrderItems
```

or

```
DROP INDEX Purchasing.PartsOrderedIndex
```

See Also “CREATE INDEX” on page 227

DROP SEQUENCE

The DROP SEQUENCE statement removes an existing sequence number generator.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL Syntax

DROP SEQUENCE [*Owner.*]*SequenceName*

Parameters

The DROP SEQUENCE statement has the parameter:

[*Owner.*]*SequenceName*	Name of the sequence number generator

Description

- Sequences can be dropped while they are in use.
- There is no ALTER SEQUENCE statement in TimesTen. To alter a sequence, use the DROP SEQUENCE statement and then create a new sequence with the same name. For example, to change the MINVALUE, one has to drop the sequence and recreate it with the same name and with the desired MINVALUE.

Example 5.80

The following command drops mysequence:

```
DROP SEQUENCE mysequence;
```

See Also

"CREATE SEQUENCE" on page 247

DROP REPLICATION

The DROP REPLICATION statement destroys a replication scheme and deletes it from the executing data store.

Access Control If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges.

SQL syntax DROP REPLICATION [*Owner.*]*ReplicationSchemeName*

Parameters The DROP REPLICATION statement has the parameter:

[Owner.]ReplicationSchemeName	Name assigned to the replication scheme.

Description Dropping the last replication scheme at a data store does not delete the replicated tables. These tables exist and persist at a data store whether or not any replication schemes are defined.

Example 5.81 The following command erases the executing data store's knowledge of replication scheme, *R*:

```
DROP REPLICATION R;
```

See Also
“ALTER REPLICATION” on page 170
“CREATE REPLICATION” on page 233

DROP TABLE

The TABLE statement deletes the specified table, including any hash indexes and any T-tree indexes associated with it.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

DROP TABLE [*Owner.*]*TableName*

Parameters

The DROP TABLE statement has the parameter:

[*Owner.*]*TableName*	Identifies the table to be dropped.

Description

- If you attempt to drop a "busy" table—a table that is in use—an error results. See Example 5.41 for more information.
- If a DROP TABLE operation is or was active in an uncommitted transaction, other transactions doing DML operations that do not access that table are allowed to proceed.
- If the table is a replicated table, you must use DROP REPLICATION to drop the replication scheme before the DROP TABLE statement. Otherwise, you receive a "Cannot drop replicated table or index" error.
- A temporary table cannot be dropped by a connection if some other connection has some non-empty instance of the table.

Example 5.82

```
CREATE TABLE VendorPerf
   (OrderNumber INTEGER,
    DelivDay SMALLINT,
    DelivMonth SMALLINT,
    DelivYear SMALLINT,
    DelivQty SMALLINT,
    Remarks VARCHAR(60))
CREATE UNIQUE INDEX VendorPerfIndex
ON VendorPerf (OrderNumber);
```

The following command drops the table and index.

```
DROP TABLE VendorPerf ;
```

DROP USER

The DROP USER statement removes a user from the instance.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires instance administration privileges. If your TimesTen instance does not use Access Control, this operation is not available. If Access Control is not enabled, TimesTen returns an error when this statement is called.

SQL syntax

DROP USER *User*

Parameters

The DROP USER statement has the parameter:

User	Name of the user that is being removed from the instance, whether a TimesTen internal user or an external user that has been previously granted privileges to the TimesTen instance. If the user is identified by the operating system (an external user), the user name must first have been introduced to the TimesTen instance through a CREATE USER statement.

Description

- TimesTen instance users are user names that have been identified to the instance. They apply to all data stores in the instance.
- Instance users may have internal user names or external user names.
  - Internal user names are defined strictly within a TimesTen instance.
  - External user names are defined by some external authority, such as the operating system.
- Changes to user identification and privileges take place at the next connection time.

Example 5.83

To remove user TERRY from the instance, use:

```
DROP USER terry;
```

DROP VIEW

The DROP VIEW statement deletes the specified view, including any hash indexes and any T-tree indexes associated with it.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires DDL privileges.

SQL syntax

DROP [MATERIALIZED] VIEW *ViewName*

Parameters

The DROP VIEW statement has the parameters:

MATERIALIZED	Specifies that the view is materialized.
ViewName	Identifies the view to be dropped.

Description

When you perform a DROP VIEW operation on a materialized view, the detail tables are updated and locked. An error may result if the detail table was already locked by another transaction.

Example 5.84

The following command drops the *CustOrder* view.

```
DROP VIEW CustOrder
```

See Also

"CREATE MATERIALIZED VIEW" on page 229
"CREATE VIEW" on page 272

FLUSH CACHE GROUP

The FLUSH CACHE GROUP statement flushes data from TimesTen to Oracle. This statement is only available for user managed cache groups. For a description of cache group types, see "User and system managed cache groups" on page 211.

There are two variants to this operation: one that accepts a WHERE clause, and one that accepts a WITH ID clause.

FLUSH CACHE GROUP is meant to be used when commit propagation (from TimesTen to Oracle) is turned off. So rather than propagating every transaction upon commit, many transactions can be committed before changes are propagated to Oracle. For each cache instance ID, if the cache instance exists in Oracle, the operation in Oracle consists of an update. If the cache instance does not exist in Oracle, TimesTen inserts it.

This is useful, for example, in a shopping cart application in which many changes may be made to the cart, which utilizes TimesTen as a high-speed cache, before the order is committed to the master Oracle table.

Note: Using a WITH ID clause usually results in better system performance than using a WHERE clause.

Only inserts and updates are flushed. Inserts are propagated as inserts if the record doesn't exist in Oracle, or as updates (if the record already exists). It is not possible to flush a delete. That is, if a record is deleted on TimesTen, there is no way to "flush" that delete to Oracle so the delete is also performed on the Oracle table. Deletes must be propagated either manually or by turning commit propagation on. Attempts to flush deleted records are silently ignored. No error or warning is issued. Records from tables that are specified as READ ONLY or PROPAGATE cannot be flushed to Oracle.

SQL syntax

FLUSH CACHE GROUP [*Owner.*]*GroupName*
[WHERE *ConditionalExpression*];

FLUSH CACHE GROUP [*Owner.*]*GroupName*
WITH ID (*ColumnValueList*)

Parameters

The FLUSH CACHE GROUP statement has the parameters:

[*Owner.*]*GroupName*	Name of the cache group to be flushed.
ConditionalExpression	A search condition to qualify the target rows of the operation.

Description

- WHERE clauses are generally used to apply the operation to a set of instances, rather than to a single instance or to all instances. The flush

operation uses the WHERE clause to determine which instances to send to Oracle.

- All table names used in cache group WHERE clauses should be fully qualified with an owner name to allow other users to execute the same WHERE clauses against the same cache group. Without an owner name, all tables referenced by cache group WHERE clauses are assumed to be owned by the current login name executing the cache group operation.
- When the WHERE clause is omitted, the entire content of the cache group is flushed to Oracle. When the WHERE clause is included, it is allowed to include only the root table.
- If propagates to Oracle are turned off (such as when the **ttCachePropagateFlagSet** built-in procedure has been called with an argument of zero in the current transaction) then all tables, with the exception of read-only tables, can be flushed to Oracle. Otherwise, only tables which are not marked as READ ONLY or PROPAGATE can be flushed to Oracle.

Example 5.85

```
FLUSH CACHE GROUP MarketBasket;
```

Example 5.86

```
FLUSH CACHE GROUP MarketBasket;
WITH ID(10);
```

GRANT

The GRANT statement assigns one or more privileges to a user.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires instance administrator privileges. If your TimesTen instance does not use Access Control, this operation is not available. If Access Control is not enabled, TimesTen returns an error when this statement is called.

SQL syntax

GRANT {*Privilege* [,...] | ALL [PRIVILEGES]} TO {*User* |PUBLIC} [,...]

Parameters

The GRANT statement has the parameters:

Privilege	Acceptable values for *Privilege* include: **ADMIN** CONNECT CREATE DATASTORE **DDL** WRITE SELECT For a description of the TimesTen Access Control privileges, see Chapter 7, "Access Control Privileges."
ALL [PRIVILEGES]	Assigns all TimesTen privileges to the user.
User	Name of the user to whom privileges are to be granted. The user name must first have been introduced to the TimesTen instance through a CREATE USER statement.
PUBLIC	Specifies that the privilege is granted to all user names defined in the TimesTen instance now and in the future.

Description

- Privileges for a user are granted at the instance level only, except that a user always has WRITE and SELECT privileges to any table they own, even if the privileges have not been granted explicitly to the user. They apply to all data stores and their objects in the instance. This release of TimesTen does not support a finer granularity of authentication, such as data store or table level privileges.
- Privileges are cumulative. Granting a lower level privilege to a user does not degrade higher privileges which have already been granted to the user. Granting a high level privilege to a user who does not have lower level privileges does not give the user the lower level privileges. To remove a privilege from a user, you must use the **REVOKE** statement.
- Privileges are determined at connect time and remain in effect until disconnect. Changes to or revocations of privileges for a user or user identification do not take effect until the user makes a new connection.

- A user always has WRITE and SELECT privileges to any table they own, even if the privileges have not been granted specifically to the user.
- The WITH GRANT OPTION clause is not supported. The user cannot pass privileges on to other users.

Example5.87 Assuming that the user Terry has no privileges, to grant administrative privileges to the user Terry:

```
GRANT ADMIN TO terry;
```

Example5.88 Assuming that the user Terry has no privileges, to restrict user Terry to read operations only, use:

```
GRANT SELECT TO terry;
```

INSERT

The INSERT statement adds rows to a table.

The following expressions can used in the values clause of an **INSERT** statement:

- TO_CHAR
- TO_DATE
- *Sequence* NEXTVAL and *Sequence* CURRVAL
- CAST
- DEFAULT
- SYSDATE and GETDATE
- USER
- CURRENT_USER
- SYSTEM_USER

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges or data store object ownership.

SQL syntax

INSERT INTO [*Owner.*]*TableName* (*ColumnName*)
VALUES (*SingleRowValues*)

Parameters

The INSERT statement has the parameters:

[*Owner.*]*TableName*	Table to which data is to be added.
ColumnName	Column for which values are supplied. If you omit any of the table's columns from the column name list, the INSERT command places the default value in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns and no default value has been defined for the column, the INSERT command fails. You can omit the column name list if you provide values for all columns of the table in the same order the columns were specified in the CREATE TABLE statement. If too few values are provided, the remaining columns are assigned default values.
VALUES	Values corresponding to the columns in the column name list or the columns specified in the CREATE TABLE statement if no column name list exists. You can also insert the sequence CURRVAL column into a table.

SingleValue	A specific single value constant.
SingleRowValues	Defines column values when you insert a single row. The syntax for *SingleRowValues* follows.

SingleRowValues

SQL syntax

The *SingleRowValues* parameter has the syntax:

{NULL | { ? | :*DynamicParameter*} | {*Constant* } | DEFAULT} [,...]

Parameters

The *SingleRowValues* parameter has the parameters:

NULL	Null value.
? *:DynamicParameter*	Place holder for a dynamic parameter in a prepared SQL statement. The value of the dynamic parameter is supplied when the statement is executed.
Constant	A specific value. See "Constants" on page 82.
DEFAULT	Specifies that the column should be updated with the default value.

Description

- If you omit any of the table's columns from the column name list, the INSERT command places the default value in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns and does no default value, the INSERT command fails.
- BINARY and VARBINARY data can be inserted in character or hexadecimal format:
  - Character format requires single quotes.
  - Hexadecimal format requires the prefix '`0x` before the value.
- If the target table has a foreign key constraint, then logging to disk must be enabled for the purpose of constraint enforcement.
- The INSERT operation fails if it violates any foreign key constraint. See "CREATE TABLE" on page 251 for a description of the foreign key constraint.

Example5.89 A new single row is added to the `Purchasing.Vendors` table.

```
INSERT INTO Purchasing.Vendors
VALUES (9016,
       'Secure Systems, Inc.',
       'Jane Secret',
       '454-255-2087',
       '1111 Encryption Way',
       'Hush',
```

```
       'MD',
       '00007',
       'discount rates are secret');
```

Example 5.90 :pNo and :pName are dynamic parameters whose values are supplied at runtime.

```
INSERT INTO Purchasing.Parts
            (PartNumber, PartName)
VALUES (:pNo, :pName);
```

INSERT SELECT

The INSERT SELECT command inserts the results of a query into a table.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges or data store object ownership.

SQL syntax

INSERT INTO [*Owner.*]*TableName* [(*ColumnName* [,...])] *InsertQuery*

Parameters

The INSERT SELECT statement has the parameters:

[*Owner.*]*TableName*	Table to which data is to be added.
ColumnName	Column for which values are supplied. If you omit any of the table's columns from the column name list, the INSERT command places the default value in the omitted columns. If the table definition specifies NOT NULL, without a default value, for any of the omitted columns, the INSERT command fails. You can omit the column name list if you provide values for all columns of the table in the same order the columns were specified in the CREATE TABLE statement. If too few values are provided, the remaining columns are assigned default values.
InsertQuery	Any supported SELECT query. See "SELECT" on page 302.

Description

- The column types of the result set must be compatible with the column types of the target table.
- You can specify a sequence CURRVAL or NEXTVAL when inserting values.
- The target table cannot be referenced in the FROM clause of the *InsertQuery*.
- In the *InsertQuery*, the ORDER BY clause is allowed. The sort order may be modified using the ORDER BY clause when the result set is inserted into the target table, but the order is not guaranteed.
- The INSERT operation fails if there is an error in the *InsertQuery*.
- If the target table has a foreign key constraint, then logging to disk must be enabled for constraint enforcement.

Example 5.91

New rows are added to the `Purchasing.Parts` table that describe which parts are delivered in 20 days or less.

```
INSERT INTO Purchasing.Parts
SELECT PartNumber, DeliveryDays
FROM Purchasing.SupplyPrice
WHERE DeliveryDays < 20;
```

LOAD CACHE GROUP

The LOAD CACHE GROUP statement loads data from an Oracle table into a TimesTen cache group.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges.

SQL syntax

LOAD CACHE GROUP [*Owner.*]*GroupName*
[WHERE *ConditionalExpression*]
COMMIT EVERY *TransactionSize* ROWS

or

LOAD CACHE GROUP [*Owner.*]*GroupName*
WITH ID (*ColumnValueList*)

Parameters

The LOAD CACHE GROUP has the parameters:

[*Owner.*]*GroupName*	Name assigned to the cache group.
ConditionalExpression	A search condition to qualify the target rows of the operation.
TransactionSize	The number of rows to insert into the cache group before committing the work. It must be a positive integer.
ColumnValueList	A list of literals or binding parameters to specify a list of column values.

Description

- Loads all new instances from Oracle that satisfy the cache group definition and are not yet present in the cache group.
- LOAD CACHE GROUP is in its own transaction, and must be the first operation in a transaction.
- When the COMMIT clause is omitted, the load occurs in a single transaction, with all work committed after every row has loaded successfully. Errors cause a rollback. When rows are committed periodically, errors abort the remainder of the load and only rollback to the last commit.
- If you use subqueries in the WHERE clause of the **LOAD CACHE GROUP** statement, the table names in the subqueries must be fully qualified.
- When loading an AUTOREFRESH or READONLY cache group:
  - The AUTOREFRESH state must be paused, and
  - The LOAD CACHE GROUP statement cannot have a WHERE clause, and
  - The cache group must be empty.

- If either a DDL operation or a write operation (**INSERT**, **CREATE VIEW**, **UPDATE**) was performed on tables that belong to a given cache group, after the LOAD statement, TimesTen returns an error.

Example 5.92

```
CREATE CACHE GROUP Recreation.Cache
    FROM Recreation.Clubs (
        ClubName CHAR(15) NOT NULL,
        ClubPhone SMALLINT,
        Activity CHAR(18),
        PRIMARY KEY(ClubName))
    WHERE (Recreation.Clubs.Activity IS NOT NULL);

LOAD CACHE GROUP Recreation.Cache
    COMMIT EVERY 30 ROWS;
```

MERGE

The MERGE statement allows you to select rows from one or more sources for update or insertion into a target table. You can specify conditions that are used to evaluate what rows are updated or inserted into the target table.

Use this statement to combine multiple INSERT and UPDATE statements.

MERGE is a deterministic statement: You cannot update the same row of the target table multiple times in the same MERGE statement.

Access Control

If Access Control is enabled for your TimesTen instance, WRITE privileges are required on the target table; SELECT privileges are required on the source table.

SQL Syntax

MERGE INTO [*Owner.*]*TargetTableName* [*Alias*] USING
{[*Owner.*]*SourceTableName* | (*Subquery*) } [*Alias*] ON (*Condtion*)
{*MergeUpdateClause MergeInsertClause* |
MergeInsertClause MergeUpdateClause |
MergeUpdateClause | *MergeInsertClause*
}

The syntax for MergeUpdateClause:
WHEN MATCHED THEN UPDATE SET *SetClause* [WHERE *Condition1*]

The syntax for MergeInsertClause:
WHEN NOT MATCHED THEN INSERT [*Columns* [,...]] VALUES
({{ *Expression* | DEFAULT|NULL} [,...] }) [WHERE *Condition2*]

Parameters

The MERGE statement has the parameters:

[*Owner.*] *TargetTableName*	Name of the target table. This is the table into which rows are either updated or inserted.
[Alias]	You can optionally specify an alias name for the target or source table.
USING {[*Owner.*] *SourceTableName* | (*Subquery*) } [*Alias*]	The USING clause indicates the table name or the subquery that is used for the source of the data. Use a subquery if you wish to use joins or aggregates. You can optionally specify an alias for the table name or the subquery.

ON (*Condition*)	You specify the condition that is used to evaluate each row of the target table to determine if the row should be considered for either a Merge Insert or a Merge Update. If the condition is true when evaluated, then the *MergeUpdateClause* is considered for the target row using the matching row from the *SourceTableName*. An error is generated if more than one row in the source table matches the same row in the target table. If the condition is not true when evaluated, then the *MergeInsertClause* is considered for that row.
SET *SetClause*	Clause used with the UPDATE statement. For information on the UPDATE statement, see "UPDATE" on page 323.
[WHERE *Condition1*]	For each row that matches the ON (*Condition*), *Condition1* is evaluated. If the condition is true when evaluated, then the row is updated. You can refer to either the target table or the source table in this clause; You cannot use a subquery. The clause is optional.
INSERT [*Columns* [,...]] VALUES ({{ *Expression* | DEFAULT|NULL} [,...] })	Columns to insert into the target table. For more information on the INSERT statement, see "INSERT" on page 289.
[WHERE *Condition2*]	If specified, *Condition2* is evaluated. If the condition is true when evaluated, then the row is inserted into the target table. The condition can refer to the source table only. You cannot use a subquery.

Description

- You can specify the *MergeUpdateClause* by itself or with the *MergeInsertClause*. Alternatively, you can specify the *MergeInsertClause* by itself or with the *MergeUpdateClause*. If you specify both, you can specify them in either order.
- Restrictions on the *MergeUpdateClause*:
  - You cannot update a column that is referenced in the ON *condition* clause.
  - You cannot update source table columns.
- Restrictions on the *MergeInsertClause*:
  - You cannot insert values of target table columns.
- Other restrictions:
  - Do not use the set operators in the subquery of the source table.
  - Do not use a subquery in the WHERE condition of either the *MergeUpdateClause* or the *MergeInsertClause*.
  - The target table cannot be a detail table of a materialized view.

Example5.93 In this example, a table called Contacts is created with columns Employee_id and Manager_id. One row is inserted into the Contacts table with values 101 and NULL for Employee_id and Manager_id respectively. The MERGE statement is used to insert rows into the contacts table using the data in the employees table. A SELECT FIRST 3 rows is used to illustrate that in the case where employee_id is equal to 101, manager_id is updated to 100; The remaining 106 rows from the employees table are inserted into the contacts table:

```
Command>  create table contacts (employee_id number (6) not null
primary key, manager_id number (6));
Command> select employee_id,manager_id from employees where
employee_id =101;
< 101, 100 >
1 row found.
Command> insert into contacts values (101,null);
1 row inserted.
Command> select count (*) from employees;
< 107 >
1 row found.
Command> MERGE INTO contacts c
>     USING employees e
>     ON (c.employee_id = e.employee_id)
> WHEN MATCHED THEN
>      UPDATE SET c.manager_id = e.manager_id
> WHEN NOT MATCHED THEN
>     INSERT (employee_id, manager_id)
>             VALUES (e.employee_id, e.manager_id);
107 rows merged.
Command> select count (*) from contacts;
< 107 >
1 row found.
Command> select first 3 employee_id,manager_id from employees;
< 100, <NULL> >
< 101, 100 >
< 102, 100 >
3 rows found.
Command> select first 3 employee_id, manager_id from contacts;
< 100, <NULL> >
< 101, 100 >
< 102, 100 >
3 rows found.
```

REFRESH CACHE GROUP

The REFRESH CACHE GROUP statement is equivalent to an UNLOAD followed by a LOAD.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges.

SQL syntax

REFRESH CACHE GROUP [*Owner.*]*GroupName*
[WHERE *ConditionalExpression*]
COMMIT EVERY *TransactionSize* ROWS

or

REFRESH CACHE GROUP [*Owner.*]*GroupName*
WITH ID (*ColumnValueList*)

Parameters

The REFRESH CACHE GROUP has the parameters:

[*Owner.*]*GroupName*	Name assigned to the cache group.
ConditionalExpression	A search condition to qualify the target rows of the operation.
TransactionSize	The positive integer number of rows to insert into the cache group before committing the work.
ColumnValueList	A list of literals or binding parameters to specify a list of column values.

Description

- A REFRESH CACHE GROUP statement must be in its own transaction.
- Refreshing the cache group is similar to loading the cache group, except all instances to be refreshed (that are already in the cache) are replaced with the most current Oracle records.
- When refreshing an AUTOREFRESH or READONLY cache group:
  - The AUTOREFRESH statement must be paused, and
  - The REFRESH statement cannot have a WHERE clause.
- If you use subqueries in the WHERE clause of the REFRESH CACHE GROUP statement, the table names in the subqueries must be fully qualified.
- When the COMMIT clause is executed, TimesTen returns an error if any DDL operation was performed or any write operation (**INSERT**, **CREATE VIEW**, **UPDATE**) was performed on tables that belong to the given cache group.

Example5.94

```
REFRESH CACHE GROUP Recreation.Cache COMMIT EVERY 30 ROWS;
```

Is equivalent to:

```
UNLOAD CACHE GROUP Recreation.Cache;
LOAD CACHE GROUP Recreation.Cache COMMIT EVERY 30 ROWS;
```

See Also

“ALTER CACHE GROUP” on page 167
“CREATE CACHE GROUP” on page 211
“DROP CACHE GROUP” on page 277
“FLUSH CACHE GROUP” on page 285
“LOAD CACHE GROUP” on page 293
“UNLOAD CACHE GROUP” on page 321

REVOKE

The REVOKE statement removes one or more privileges from a user.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires instance administrator privileges. If your TimesTen instance does not use Access Control, this operation is not available. If Access Control is not enabled, TimesTen returns an error when this statement is called.

SQL syntax

REVOKE {*Privilege* [, …] | ALL [PRIVILEGES]}
FROM {*User* |PUBLIC} [,...]

Parameters

The REVOKE statement has the parameters:

Privilege	Acceptable values for *Privilege* include: ADMIN CONNECT CREATE DATASTORE DDL WRITE SELECT For a description of the TimesTen Access Control privileges, see Chapter 7, "Access Control Privileges."
ALL [PRIVILEGES]	Revokes all TimesTen privileges to the user.
User	Name of the user to whom privileges are to be revoked. The user name must first have been introduced to the TimesTen instance through a CREATE USER statement.
PUBLIC	Specifies that the privilege is revoked to all user names defined in the TimesTen instance now and in the future.

Description

- User may be PUBLIC.
- Privileges are cumulative. Revoking a single privilege from a user does not revoke any lower or higher level privilege for that user.
- Revoking SELECT and WRITE privileges for a user does not remove those privileges for objects that the user owns.

Example 5.95

To revoke WRITE and DDL privileges for the user TERRY, use:

```
REVOKE WRITE, DDL FROM terry;
```

Example 5.96

To revoke WRITE privileges for all users, use:

```
REVOKE WRITE FROM PUBLIC;
```

See Also
“ALTER USER” on page 203
“CREATE USER” on page 270
“DROP USER” on page 283
“GRANT” on page 287

SELECT

The SELECT statement retrieves data from one or more tables. The retrieved data is presented in the form of a table that is called the “result table” or “query result.”

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires SELECT privileges or data store object ownership.

SQL syntax

The general syntax for a SELECT statement is:

```
SELECT [FIRST NumRows | ROWS M TO N] [ALL | DISTINCT] SelectList
    FROM TableSpec [,...]
    [WHERE SearchCondition]
    [GROUP BY Expression [,...]]
    [HAVING SearchCondition]
    [ORDER BY {ColumnID| ColumnAlias | Expression {ASC | DESC}]
    [,...]

    [FOR UPDATE [OF [[Owner.]TableName.]ColumnName [,...] ]
        [NOWAIT | WAIT Seconds] ]
```

The syntax for a SELECT statement that contains the set operators UNION, UNION ALL, MINUS, or INTERSECT is:

```
SELECT [ROWS M TO N] [ALL] SelectList
    FROM TableSpec [,...]
    [WHERE SearchCondition]
    [GROUP BY Expression [,...]]
    [HAVING SearchCondition] [,...]

{UNION [ALL] | MINUS | INTERSECT}

SELECT [ROWS M TO N] [ALL] SelectList
    FROM TableSpec [,...]
    [WHERE SearchCondition]
    [GROUP BY Expression [,...]]
    [HAVING SearchCondition] [,...]
    [ORDER BY {ColumnID| ColumnAlias | Expression {ASC | DESC}]
```

Parameters The SELECT statement has the parameters:

FIRST *NumRows*	Specifies the number of rows to retrieve. *NumRows* must be either a positive INTEGER or a dynamic parameter placeholder. The syntax for a dynamic parameter placeholder is either ? or :*DynamicParameter*. The value of the dynamic parameter is supplied when the statement is executed.
ROWS *M* TO *N*	Specifies the range of rows to retrieve where *M* is the first row to be selected and *N* is the last row to be selected. Row counting starts at row 1. The query SELECT ROWS 1 to *N* ... returns the same rows as SELECT FIRST *NumRows* assuming the queries are ordered and *N* and *NumRows* have the same value. Use either a positive INTEGER or a dynamic parameter placeholder for M and N values. The syntax for a dynamic parameter placeholder is either ? or :*DynamicParameter*. The value of the dynamic parameter is supplied when the statement is executed.
ALL	Prevents elimination of duplicate rows from the query result. If neither ALL nor DISTINCT is specified, ALL is assumed.
DISTINCT	Ensures that each row in the query result is unique. All NULL values are considered equal for this comparison. Duplicate rows are not evaluated.
SelectList	Specifies how the columns of the query result are to be derived. The syntax of *SelectList* is presented under "SelectList" on page 313.
FROM *TableSpec*	Identifies the tables referenced in the SELECT statement. The maximum number of tables per query is 24. *TableSpec* identifies a table from which rows are selected. The table can be a derived table, which is the result of a SELECT statement in the FROM clause. The syntax of *TableSpec* is presented under "TableSpec" on page 316.

WHERE *SearchCondition*	The WHERE clause determines the set of rows to be retrieved. Normally, rows for which *SearchCondition* is FALSE or NULL are excluded from processing, but *SearchCondition* can be used to specify an outer join in which rows from an outer table that do not have *SearchCondition* evaluated to TRUE with respect to any rows from the associated inner table are also returned, with projected expressions referencing the inner table set to NULL. The unary (+) operator may follow some column and ROWID expressions to indicate an outer join. The (+) operator must follow all column and ROWID expressions in the join condition(s) that refer to the inner table. There are several conditions on the placement of the (+) operator. These generally restrict the type of outer join queries that can be expressed. The (+) operator may appear in WHERE clauses, but not in HAVING clauses. Two tables cannot be outer joined together. An outer join condition cannot be connected by OR. See Chapter 4, "Search Conditions," for additional information on search conditions.
GROUP BY *Expression* [,...]	The GROUP BY clause identifies one or more expressions to be used for grouping when aggregate functions are specified in the select list and when you want to apply the function to groups of rows. The expression can be of various complexities. For example, it can designate single or multiple columns; it can include aggregate functions, arithmetic operations, the ROWID pseudo-column, or NULL. It can also be a date or user function, a constant, or a dynamic parameter. When you use the GROUP BY clause, the select list can contain *only* aggregate functions and columns referenced in the GROUP BY clause. If the select list contains an *, a *TableName*.*, or an *Owner.TableName*.* construct, then the GROUP BY clause must contain all columns that the * includes. NULL values are considered equivalent in grouping rows. If all other columns are equal, all NULLs in a column are placed in a single group. If the GROUP BY clause is omitted, the entire query result is treated as one group.
HAVING	The HAVING clause can be used in a SELECT query to filter groups of an aggregate result. The existence of a HAVING clause in a SELECT query turns the query into an aggregate query. All columns referenced outside the sources of aggregate functions in every clause except the WHERE clause must be included in the GROUP BY clause. Subqueries can be specified in the HAVING clause.

(+)	A simple join (also called an inner join) returns a row for each pair of rows from the joined tables that satisfy the join condition specified in *SearchCondition*. Outer joins an extension of this operator in which all rows of the “outer” table are returned, whether or not matching rows from the joined inner table are found. In the case no matching rows are found, any projected expressions referencing the inner table are given value NULL.
ORDER BY	Sorts the query result rows in order by specified columns or expressions. Specify the sort key columns in order from major sort key to minor sort key. You can specify as many as 255 columns. For each column, you can specify whether the sort order is to be ascending or descending. If neither ASC nor DESC is specified, ascending order is used. Character strings are compared according to the ASCII collating sequence for ASCII data. The ORDER BY clause supports column aliases. Column aliases can be referenced only in an ORDER BY clause. A single query may declare several column aliases with the same name, but any reference to that alias results in an error. NCHAR types are not supported with ORDER BY.
ColumnID	Must correspond to a column in the select list. You can identify a column to be sorted by giving its name or by giving its ordinal number. The first column in the select list is column number 1. It is better to use a column number when referring to columns in the select list if they are not simple columns. Some examples are aggregate functions, arithmetic expressions, and constants. A *ColumnID* in the ORDER BY clause has the syntax: {*ColumnNumber* | [[*Owner.*]*TableName.*] *ColumnName*}
ColumnAlias	Used in an ORDER BY clause, the column alias must correspond to a column in the select list. The same alias can identify multiple columns. {* | [*Owner.*]*TableName*.* | { Expression | [[*Owner.*]*TableName.*]*ColumnName* | [[*Owner.*]*TableName*.]ROWID } [[AS] ColumnAlias] } [,...]

FOR UPDATE [OF [[*Owner*.] *TableName*.] *ColumnName* [,...]] [NOWAIT | WAIT *Seconds*]	FOR UPDATE • Maintains a lock on an element (usually a row) until the end of the current transaction, regardless of isolation. All other transactions are excluded from performing any operation on that element until the transaction is committed or rolled back. • FOR UPDATE may be used with joins and the ORDER BY, GROUP BY, and DISTINCT clauses. Update locks are obtained on either tables or rows depending on the table/row locking method chosen by the optimizer. • Rows from all tables that satisfy the WHERE clause are locked in UPDATE mode unless the FOR UPDATE OF clause is specified; this clause specifies which tables to lock. • If using row locks, all qualifying rows in all tables from the table list in the FROM clause are locked in update mode. Qualifying rows are those rows that satisfy the WHERE clause. If using table locks, the table is locked in update mode whether or not there are any qualifying rows. • If the serializable isolation level and row locking are turned on, non-qualifying rows are downgraded to Shared mode. If a read-committed isolation level and row locking are turned on, non-qualifying rows are unlocked. • SELECT...FOR UPDATE locks are not blocked by SELECT locks. • SELECT...FOR UPDATE does not support subqueries. FOR UPDATE [OF [[*Owner.*]*TableName*.]*ColumnName* [,...]] • Optionally includes the name of the column or columns in the table to be locked for update. [NOWAIT | WAIT *Seconds*] • Specifies how to proceed if the selected rows are locked. • NOWAIT specifies that there is no waiting period for locks and an error is returned if the lock is not available. • WAIT *Seconds* specifies the lock timeout setting. – An error is returned if the lock is not obtained in the specified amount of time. – The lock timeout setting is expressed in seconds or a fraction of a second; The data type for *Seconds* is NUMBER; Values between 0.0 and 1000000.0 are valid. • If neither NOWAIT nor WAIT is specified, the lock timeout interval for the transaction is used.

SelectQuery1 {UNION [ALL] | MINUS | INTERSECT} *SelectQuery2*	Specifies that the results of *SelectQuery1* and *SelectQuery2* are to be combined, where *SelectQuery1* and *SelectQuery2* are general SELECT statements with some restrictions. The UNION operator combines the results of two queries where the SelectList are compatible. If UNION ALL is specified, duplicate rows from both SELECT statements are retained; duplicates are removed. The MINUS operator combines rows returned by the first query but not by the second into a single result. The INTERSECT operator combines only those rows returned by both queries into a single result. The data type of corresponding selected entries in both SELECT statements must be compatible. One type can be converted to the other type using the **CAST** operator. Nullability does not need to match. The length of a column in the result is the longer length of correspondent selected values for the column. The column names of the final result are the column names of the leftmost select. You can combine multiple queries using the set operators UNION, UNION ALL, MINUS, and INTERSECT. One or both operands of a set operator can be a set operator. Multiple or nested set operators are evaluated from left to right. The set operators can be mixed in the same query. Restrictions on the SELECT statement that specify the set operators are: • Neither SELECT statement can specify FIRST *NumRows*. • The set operators can only be used in the outermost level of a SELECT statement, an INSERT SELECT statement, a derived table or a view. • ORDER BY can be specified to sort the final result but cannot be used with any individual operand of a set operator. Only column names of tables or column alias from the leftmost SELECT can be specified in the ORDER BY clause. • GROUP BY can be used to group an individual SELECT operand of a set operator but cannot be used to group a set operator result. • The set operators cannot be used in materialized view or a joined table.

Description

- When using a correlation name, the correlation name must conform to the syntax rules for a basic name (see "Basic names" on page 67). All correlation names within one SELECT query must be unique. Correlation names are useful when you join a table to itself. Define multiple correlation names for the table in the FROM clause and use the correlation names in the *SelectList* and the WHERE clause to qualify columns from that table.
- If using a subquery, SELECT ... FOR UPDATE may not be used in the outer nor inner queries, as subqueries are not fully supported.
- If your query specifies either FIRST *NumRows* or ROWS M TO N, ROWNUM may not be used to restrict the number of rows returned.
- FIRST*NumRows* and ROWS M TO N cannot be used together in the same SELECT statement.

Example5.97 This example shows the use of a column alias in the SELECT statement:

```
SELECT max(salary) AS max_salary
FROM employee WHERE employee.age < 30;
```

Example5.98 This example assumes there are two tables, *Orders* and *LineItems*.

The Orders table is created as shown below:

```
CREATE TABLE Orders(OrderNo INTEGER, OrderDate DATE, Customer
CHAR(20));
CREATE TABLE LineItems(OrderNo INTEGER, LineNo INTEGER,
    Qty INTEGER, UnitPrice DECIMAL(10,2));
```

Thus for each order, there is one record in the *Orders* table and a record for each "line" of the order in *LineItems*.

To find the total value of all Orders entered since the beginning of the year, use the HAVING clause to select only those orders that were entered on or after January 1, 2000:

```
SELECT O.OrderNo, CUSTOMER, ORDERDATE, SUM(Qty * UnitPrice)
FROM Orders O, LineItems L
WHERE O.OrderNo=L.OrderNo
GROUP BY O.OrderNo, CUSTOMER, ORDERDATE
HAVING ORDERDATE >= DATE '2000-01-01';
```

Example5.99 This query locks all rows in *TableA* where:

- *TableA.Column1* equals at least one *TableB.Column1* value where *TableB.Column2* is greater than 5.

In addition, this query locks all rows in *TableB* where:

- *TableB.Column2* is greater than 5
- *TableB.Column1* equals at least one *TableA.Column1* value.

If no WHERE clause is specified, all rows in both tables would be locked.

```
SELECT * FROM TableA, TableB
WHERE TableA.Column1 = TableB.Column1 AND TableB.Column2 > 5
FOR UPDATE
```

Example 5.100

This query returns an error, since the inner table *t2* corresponds to two outer tables (*t1* and *t3*):

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) and t3.y = t2.y(+);
```

This example demonstrates valid syntax:

```
SELECT * FROM t1, t2
WHERE t1.x = t2.x(+);
```

Example 5.101

This query returns an error, because an outer join condition cannot be connected by OR:

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) OR t3.y = 5;
```

But the following query is valid:

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) AND (t3.y = 4 OR t3.y = 5);
```

Example 5.102

A condition cannot use the IN operator to compare a column marked with (+). For example, the following query returns an error:

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) AND t2.y(+) IN (4,5);
```

But the following query is valid:

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) AND t1.y IN (4,5);
```

Example 5.103

The following query results in an inner join instead of an outer join, because the (+) operator was not specified in each of the join conditions, and the condition without the (+) is treated as an inner join condition:

```
SELECT * FROM t1, t2
WHERE t1.x = t2.x(+) AND t1.y = t2.y;
```

Example 5.104

In the following query, the WHERE clause contains a condition that compares an inner table column of an outer join with a constant. The (+) operator was not specified and hence the condition is treated as an inner join condition.

```
SELECT * FROM t1, t2
WHERE t1.x = t2.x(+) AND t2.y = 3;
```

Example 5.105 The following query returns an error because two tables cannot be outer joined together:

```
SELECT * FROM t1, t2
WHERE x1 = x2(+) AND y2 = y1(+);
```

Example 5.106 Finds the current sequence value in the *student* table.

```
SELECT seq.CURRVAL FROM student;
```

Example 5.107 In the following query, the condition '`x2 + y2(+) = 1`' is treated as an inner join condition because the (+) operator was not specified for the column '`x2`' of inner table '`t2`.' The statement returns an error because two tables cannot be outer joined together:

```
SELECT * FROM t1, t2
WHERE x1 = x2(+) AND x2 + y2(+) = 1;
```

Example 5.108 The following query does not specify an outer join because the (+) operator is not specified in a join condition:

```
SELECT * FROM t1, t2
WHERE x2(+) = 1;
```

Example 5.109 The following query produces a derived table, as it contains a SELECT statement in the FROM clause:

```
SELECT * FROM t1, (SELECT MAX(x2) MAXX2 FROM t2) tab2 WHERE t1.x1 =
tab2.MAXX2;
```

Example 5.110 The following query joins the results of two SELECT statements.

```
SELECT * FROM t1 WHERE x1 IN (SELECT x2 FROM t2) UNION SELECT *
FROM t1 WHERE x1 IN (SELECT x3 FROM t3);
```

Example 5.111 Select all orders that have the same price as the highest price in its category:

```
SELECT * FROM orders WHERE price = (SELECT MAX(price) FROM stock
WHERE stock.cat=orders.cat);
```

Example 5.112 The example illustrates the use of the INTERSECT set operator. There is a department_id in the employees table that is NULL. In the departments table, the department_id is defined as a NOT NULL primary key. The rows returned from using the INTERSECT set operator does not include the row in the departments table whose department_id is NULL.

```
Command> SELECT department_id FROM employees INTERSECT SELECT
department_id FROM departments;
< 10 >
```

```
< 20 >
< 30 >
< 40 >
< 50 >
< 60 >
< 70 >
< 80 >
< 90 >
< 100 >
< 110 >
11 rows found.
```

```
Command> SELECT DISTINCT department_id FROM employees;
< 10 >
< 20 >
< 30 >
< 40 >
< 50 >
< 60 >
< 70 >
< 80 >
< 90 >
< 100 >
< 110 >
< <NULL> >
12 rows found.
```

Example 5.113 The example illustrates the use of the MINUS set operator by combining rows returned by first query but not the second. The row containing the NULL department_id in the employees table is the only row returned.

```
Command> SELECT department_id FROM employees MINUS SELECT
department_id FROM departments;
< <NULL> >
1 row found.
```

Example 5.114 The following example sums the salaries for employees in the employees table and uses the SUBSTR expression to group the data by job function.

```
Command> SELECT SUBSTR (JOB_ID, 4,10), SUM (SALARY) FROM EMPLOYEES
GROUP BY SUBSTR (JOB_ID,4,10);

< PRES, 24000 >
< VP, 34000 >
< PROG, 28800 >
< MGR, 24000 >
< ACCOUNT, 47900 >
< MAN, 121400 >
```

```
< CLERK, 133900 >
< REP, 273000 >
< ASST, 4400 >
9 rows found.
```

Example 5.115 The example illustrates the use of the SUBSTR expression in a GROUP BY clause and the use of a subquery in a HAVING clause. The first 10 rows are returned.

```
Command> SELECT ROWS 1 TO 10 SUBSTR (JOB_ID, 4,10),department_id,
manager_id, SUM (SALARY) FROM employees
>GROUP BY SUBSTR (JOB_ID,4,10),department_id, manager_id
> HAVING (department_id, manager_id) IN
> (SELECT department_id, manager_id FROM employees x
> WHERE x.department_id = employees.department_id)
> ORDER BY substr (job_id, 4,10),department_id,manager_id;
< ACCOUNT, 100, 108, 39600 >
< ACCOUNT, 110, 205, 8300 >
< ASST, 10, 101, 4400 >
< CLERK, 30, 114, 13900 >
< CLERK, 50, 120, 22100 >
< CLERK, 50, 121, 25400 >
< CLERK, 50, 122, 23600 >
< CLERK, 50, 123, 25900 >
< CLERK, 50, 124, 23000 >
< MAN, 20, 100, 13000 >
10 rows found.
```

Example 5.116 The example locks the employees table for update and waits 10 seconds for the lock to be available. An error will be returned if the lock is not acquired in 10 seconds. The first 5 rows are selected.

```
Command> SELECT FIRST 5 last_name FROM employees FOR UPDATE WAIT 10;
< King >
< Kochhar >
< De Haan >
< Hunold >
< Ernst >
5 rows found.
```

Example 5.117 The example locks the departments table for update. If the selected rows are locked by another process, an error is returned if the lock is not available. This is because NOWAIT is specified.

```
Command>  SELECT FIRST 5 last_name e FROM employees e, departments
d WHERE e.department_id = d.department_id FOR UPDATE OF
d.department_id NOWAIT;
< Whalen >
```

```
< Hartstein >
< Fay >
< Raphaely >
< Khoo >
5 rows found.
```

SelectList

SQL syntax

The *SelectList* parameter of the SELECT statement has the syntax:

{* | [*Owner.*]*TableName*.* |
{ *Expression* | [[*Owner.*]*TableName*.]*ColumnName* |
[[*Owner.*]*TableName*.]ROWID | NULL
}
[[AS] *ColumnAlias*] } [,...]

Parameters

The *SelectList* parameter of the SELECT statement has the parameters:

*	Includes, as columns of the query result, all columns of all tables specified in the FROM clause.
[*Owner.*]*TableName*.*	Includes all columns of the specified table in the result.
Expression	The expression can be of any complexity. For example, it can designate a single column of one of the tables specified in the FROM clause, or it can involve aggregate functions, arithmetic operations, multiple columns or NULL. When the Select is an aggregate query or expression, then you cannot specify the name(s) of the column(s) that are not in your GROUP BY clause.
[[*Owner.*]*Table*.] *ColumnName*	Includes a particular column from the named owner's indicated table. You can also specify the CURRVAL or NEXTVAL column of a sequence.
[[*Owner.*]*Table*.] ROWID	Includes the ROWID pseudo-column from the named owner's indicated table.

NULL	When NULL is specified, the default resulting data type is VARCHAR(0). You can use the CAST function to convert the result to a different data type. NULL can be specified in the ORDER BY clause.
ColumnAlias	Used in an ORDER BY clause, the column alias must correspond to a column in the select list. The same alias can identify multiple columns. {* | [*Owner.*]*TableName*.* | { *Expression* | [[*Owner.*]*TableName*.]*ColumnName* | [[*Owner.*]*TableName*.]ROWID } [[AS] *ColumnAlias*] } [,...]

Description

- The clauses must be specified in the order given in the syntax diagram.
- A result column in the select list can be derived in any of the following ways:
  - A result column can be taken directly from one of the tables listed in the FROM clause.
  - Values in a result column can be computed, using an arithmetic expression, from values in a specified column of a table listed in the FROM clause.
  - Values in several columns of a single table can be combined in an arithmetic expression to produce the result column values.
  - Values in columns of different tables can be combined in an arithmetic expression to produce the result column values.
  - The aggregate functions (AVG, MAX, MIN, SUM, and COUNT) can be used to compute result column values over groups of rows. Aggregate functions can be used alone or in an expression. You can specify aggregate functions containing the DISTINCT option that operate on different columns in the same table. If the GROUP BY clause is not specified, the function is applied over all rows that satisfy the query. If the GROUP BY clause is specified, the function is applied once for each group defined by the GROUP BY clause. When you use aggregate functions with the GROUP BY clause, the select list can contain aggregate functions, arithmetic expressions, and columns in the GROUP BY clause.
  - A result column containing a fixed value can be created by specifying a constant or an expression involving only constants.
- In addition to specifying how the result columns are derived, the select list also controls their relative position from left to right in the query result. The first result column specified by the select list becomes the leftmost column in the query result.
- Result columns in the select list are numbered from left to right. The leftmost column is number 1. Result columns can be referred to by column number in

the ORDER BY clause; this is especially useful if you want to refer to a column defined by an arithmetic expression or an aggregate.

- To join a table with itself, define multiple correlation names for the table in the FROM clause and use the correlation names in the select list and the WHERE clause to qualify columns from that table.
- When you use the GROUP BY clause, one answer is returned per group in accordance with the select list:
  - The WHERE clause eliminates rows before groups are formed.
  - The GROUP BY clause groups the resulting rows.
  - The select list aggregate functions are computed for each group.

Example 5.118

One value, the average number of days you wait for a part, is returned by the statement:

```
SELECT AVG(DeliveryDays)
FROM Purchasing.SupplyPrice;
```

The part number and delivery time for all parts that take fewer than 20 days to deliver are returned by the statement:

```
SELECT PartNumber, DeliveryDays
FROM Purchasing.SupplyPrice
WHERE DeliveryDays < 20;
```

Multiple rows may be returned for a single part.

Example 5.119

The part number and average price of each part are returned by the statement:

```
SELECT PartNumber, AVG(UnitPrice)
FROM Purchasing.SupplyPrice
GROUP BY PartNumber;
```

Example 5.120

In this example, the join returns names and locations of California suppliers. Rows are returned in ascending PartNumber order; rows containing duplicate `PartNumbers` are returned in ascending `VendorName` order.

The FROM clause defines two correlation names (*v* and *s*), which are used in both the select list and the WHERE clause.

`VendorNumber` is the only common column between `Vendors` and `SupplyPrice`.

```
SELECT PartNumber, VendorName, s.VendorNumber,
       VendorCity
FROM Purchasing.SupplyPrice s,
     Purchasing.Vendors v
WHERE s.VendorNumber = v.VendorNumber
AND VendorState = 'CA'
ORDER BY PartNumber, VendorName;
```

Example 5.121 This query joins table Purchasing.Parts to itself to determine which parts have the same sales price as the part whose serial number is '1133-P-01'.

```
SELECT q.PartNumber, q.SalesPrice
FROM Purchasing.Parts p, Purchasing.Parts q
WHERE p.SalesPrice = q.SalesPrice AND
      p.SerialNumber = '1133-P-01';
```

Example 5.122 This example shows how to retrieve the ROWID of a specific row. The retrieved ROWID value can be used later for another SELECT, CREATE VIEW, or UPDATE statement.

```
SELECT ROWID
FROM Purchasing.Vendors
WHERE VendorNumber = 123;
```

Example 5.123 This example shows how to use a column alias to retrieve data from the table employees.

```
SELECT max(salary) AS max_salary FROM employees;
```

TableSpec

SQL syntax The *TableSpec* parameter of the SELECT statement has the syntax:

{[*Owner.*]*TableName* [*CorrelationName*] | *JoinedTable* | *DerivedTable}*

A simple table specification has the syntax:

[*Owner.*]*TableName*

Parameters The *TableSpec* parameter of the SELECT statement has the parameters:

[*Owner.*]*TableName*	Identifies a table to be referenced.
CorrelationName	*CorrelationName* specifies a synonym for the immediately preceding table. When accessing columns of that table, use the correlation name instead of the actual table name within the statement. The correlation name must conform to the syntax rules for a basic name (see "Basic names" on page 67). All correlation names within one statement must be unique.
DerivedTable	Specifies a table derived from the evaluation of a SELECT query. No FIRST *NumRows* or Rows *M* to *N* clauses are allowed in this SELECT query.
JoinedTable	Specifies the query that defines the table join. The syntax of *JoinedTable* is presented under "JoinedTable" on page 317.

DerivedTable

SQL syntax

A derived table is the result of select statement in the FROM clause, with an alias.

The syntax for *DerivedTable* is:

(*Subquery*) [*CorrelationName*]

Parameters

The *DerivedTable* parameter of the *TableSpec* clause of a SELECT statement has the parameters:

Subquery	For information on subqueries, see "Subqueries" on page 76 .
CorrelationName	*CorrelationName* must be different from any table name referenced in the query; *CorrelationName* is optional.

Description

When using a derived table, these restrictions apply:

- The DUAL table can be used in a SELECT statement that references no other tables, but needs to return at least one row. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once.
- The SelectQuery cannot refer to a column from another derived table.
- The derived table cannot be used as an operand of a joined table.
- The derived table cannot be used as a target of a DELETE or UPDATE statement.
- A derived table name cannot be the same as any table name referenced in the statement.
- A derived table cannot be used as a table of a joined table.
- A derived table cannot be used as a target of a **DELETE** or an **UPDATE** statement.

JoinedTable

SQL syntax

The *JoinedTable* specifies a table derived from a CROSS JOIN, INNER, LEFT or RIGHT OUTER JOIN.

The syntax for *JoinedTable* is:

{*CrossJoin* | *QualifiedJoin*}

Where *CrossJoin* is:

TableSpec1 CROSS JOIN *TableSpec2*

and QualifiedJoin is:

TableSpec1 [*JoinType*] JOIN *TableSpec2* ON *SearchCondition*

In the *QualifiedJoin* parameter, *JoinType* syntax is:
{INNER | LEFT [OUTER] | |RIGHT [OUTER]}

Parameters

The *JoinedTable* parameter of the *TableSpec* clause of a SELECT statement has the parameters:

CrossJoin	Performs a CROSS JOIN on two tables. A CROSS JOIN returns a result table that is the cartesian product of the input tables. The result is the same as that of a query with the syntax: SELECT *Selectlist* FROM *Table1, Table2*
QualifiedJoin	Specifies that the Join is the result of a join of type *JoinType*.
TableSpec1	Specifies the first table of the JOIN clause.
TableSpec2	Specifies the second table of the JOIN clause.
JoinType JOIN	Specifies the type of join to perform. Supported join types are: INNER LEFT [OUTER] RIGH [OUTER] An INNER JOIN returns a result table that combines the rows from two tables that meet the *SearchCondition*. A LEFT OUTER JOIN returnS join rows that match the *SearchCondition* and rows from the first table that do not have the *SearchCondition* evaluated to true with any row from the second table. A RIGHT OUTER JOIN returns join rows that match the *SearchCondition* and rows from the second table that do not have the *SearchCondition* evaluated to true with any row from the first table.
ON SearchCondition	Specifies the search criteria to be used in a JOIN parameter. This *SearchCondition* can only refer to tables referenced in the current qualified JOIN.

Description

- FULL OUTER JOIN is not supported
- A joined table can be used to replace a table in a FROM clause, anywhere except in a statement to define a materialized view. Therefore, a joined table can be used in a UNION, MINUS or INTERSECT, a subquery, a non-materialized view or a derived table.
- A temporary table cannot be specified as an operand of a joined table, but a view can.

- OUTER JOIN can be specified in two ways, using the (+) operator in the SearchCondition of the WHERE clause, or to use a JOIN table operation. The two cannot co-exist in the same statement.
- Join order and grouping can be specified with a JoinedTable operation, but not with (+). For example, the following operation is not supported:

  t LEFT JOIN (t2 INNER JOIN t3 ON x2=x3) ON (x1 = x2 + x3)

Example 5.124 The following statement joins tables t1 and t2, returning all the rows from t1 where x1 is less than 10:

```
SELECT * FROM t1 LEFT JOIN t2 ON x1=x2 WHERE x1<10;
```

See Also
"CREATE TABLE" on page 251
"INSERT" on page 289
"INSERT SELECT" on page 292
"UPDATE" on page 323

TRUNCATE TABLE

The TRUNCATE TABLE statement is similar to a DELETE statement that deletes all rows. However, it is faster than DELETE in most circumstances, as DELETE removes each row individually.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires ADMIN privileges.

SQL syntax

TRUNCATE TABLE [*Owner.*]*TableName*

Parameters

The TRUNCATE TABLE has the parameter:

Owner.]*TableName*	Identifies a table to be truncated.

Description

- TRUNCATE operations can be rolled back.
- Subsequent INSERT statements are not allowed in the same transaction as a TRUNCATE statement.
- Concurrent read committed read operations are allowed; and semantics of the reads are the same as for read committed reads in presence of DELETE statements
- TRUNCATE is allowed even when there are child tables. However, child tables need to be empty for TRUNCATE to proceed. If any of the child tables have any rows in them, TimesTen returns an error indicating that a child table is not empty.
- TRUNCATE is not supported with detail tables of a Materialized View and a table that is a part of a cache group or a temporary table.
- When a table contains out-of-line varying-length data, the performance of TRUNCATE TABLE is similar to that of DELETE statement that deletes all rows in a table. For more details on out-of line data, see "Numeric data types" on page 29.
- Where tables are being replicated, the TRUNCATE statement replicates to the subscriber, even when no rows are operated upon.
- When tables are being replicated with timestamp conflict checking enabled, conflicts are not reported.
- DROP TABLE and ALTER TABLE operations cannot be used to change hash pages on uncommitted truncated tables.

Example 5.125

To delete all the rows from the Recreation.Clubs table, use:

```
TRUNCATE TABLE Recreation.Clubs;
```

See Also

"ALTER TABLE" on page 186
"DROP TABLE" on page 282

UNLOAD CACHE GROUP

The UNLOAD CACHE GROUP statement deletes all rows from the cache group.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges.

SQL syntax

UNLOAD CACHE GROUP [*Owner.*]*GroupName*
[WHERE *ConditionalExpression*]

UNLOAD CACHE GROUP [*Owner.*]*GroupName*
WITH ID (*ColumnValueList*);

Parameters

The UNLOAD CACHE GROUP has the parameter:

[Owner.]GroupName	Name assigned to the cache group.
ColumnValueList	A list of literals or binding parameters to specify a list of column values.
ConditionalExpression	A search condition to qualify the target rows of the operation.

Description

- This syntax causes the entire content of the cache group to be deleted from the data store.
- If the cache group is replicated, an UNLOAD CACHE GROUP command deletes the entire content of the cache group at replicas as well.
- The UNLOAD CACHE GROUP statement can be used for any type of cache group. For a description of cache group types, see "User and system managed cache groups" on page 211.
- Use the UNLOAD CACHE GROUP statement carefully with cache groups that have the AUTOFRESH attribute. A row that is unloaded can reappear in the cache group as the result of an autorefresh operation if the row or its child rows are updated in Oracle.

Example 5.126

```
CREATE CACHE GROUP Recreation.Cache
    FROM Recreation.Clubs (
        ClubName CHAR(15) NOT NULL,
        ClubPhone SMALLINT,
        Activity CHAR(18),
        PRIMARY KEY(ClubName))
    WHERE (Recreation.Clubs.Activity IS NOT NULL);
```

```
UNLOAD CACHE GROUP Recreation.Cache;
does a special ON DELETE CASCADE.
```

See Also

“ALTER CACHE GROUP” on page 167
“CREATE CACHE GROUP” on page 211
“DROP CACHE GROUP” on page 277
“FLUSH CACHE GROUP” on page 285
“LOAD CACHE GROUP” on page 293
“MERGE” on page 295

UPDATE

The UPDATE statement updates the values of one or more columns in all rows of a table or in rows that satisfy a search condition.

Access Control

If Access Control is enabled for your TimesTen instance, this statement requires WRITE privileges or data store object ownership.

SQL syntax

The UPDATE statement has the syntax:

UPDATE [FIRST *NumRows*]
{[*Owner.*]*TableName* [*CorrelationName*]}
SET {*ColumnName* =
{*Expression* | NULL | DEFAULT}} [,...]
[WHERE *SearchCondition*]

Parameters

The UPDATE statement has the parameters:

FIRST *NumRows*	Specifies the number of rows to update. FIRST *NumRows* is not supported in subquery statements. *NumRows* must be either a positive INTEGER or a dynamic parameter placeholder. The syntax for a dynamic parameter placeholder is either ? or :*DynamicParameter*. The value of the dynamic parameter is supplied when the statement is executed.
[*Owner.*]*TableName* [*CorrelationName*]	[*Owner.*]*TableName* identifies a table to be updated. *CorrelationName* specifies a synonym for the immediately preceding table. When accessing columns of that table, use the correlation name instead of the actual table name within the statement. The correlation name must conform to the syntax rules for a basic name (see "Basic names" on page 67). All correlation names within one statement must be unique.
SET *ColumnName*	Column to be updated. You can update several columns of the same table with a single UPDATE statement. Primary key columns can be included in the list of updated columns as long as the values of the primary key columns are not changed.
Expression	Any expression that does not contain an aggregate function. The expression is evaluated for each row qualifying for the update operation. The data type of the expression must be compatible with the updated column's data type. *Expression* can specify a column or sequence CURRVAL or NEXTVAL reference when updating values.
NULL	Puts a NULL value in the specified column of each row satisfying the WHERE clause. The column must allow NULL values.

DEFAULT	Specifies that the column should be updated with the default value.
WHERE *SearchCondition*	The search condition can contain a subquery. All rows for which the search condition is TRUE are updated as specified in the SET clause. Rows that do not satisfy the search condition are not affected. If no rows satisfy the search condition, the table is not changed.

Description

- If the WHERE clause is omitted, all rows of the table are updated as specified by the SET clause.
- The TimesTen Data Manager generates a warning when a character or binary string is truncated during an UPDATE operation.
- The target table of the UPDATE statement is designated by *TableName*.
- A table on which a unique constraint is defined cannot be updated to contain duplicate rows.
- If the target table has a foreign key constraint, then logging to disk must be enabled for constraint enforcement.
- The UPDATE operation fails if it violates any foreign key constraint. See "CREATE TABLE" on page 251 for a description of the foreign key constraint.

Example 5.127 This example increases the price of parts costing more than $500 by 25%.

```
UPDATE Purchasing.Parts
SET SalesPrice = SalesPrice * 1.25
WHERE SalesPrice > 500.00;
```

Example 5.128 This example updates the column with the NEXTVAL value from sequence seq.

```
UPDATE student SET studentno = seq.NEXTVAL WHERE name = 'Sally';
```

Example 5.129 The following query updates the status of all the customers who have at least one un-shipped order:

```
UPDATE customers SET customers.status = 'un-shipped'
WHERE customers.id = ANY
(SELECT orders.custid FROM orders
WHERE orders.status = 'un-shipped');
```

Example 5.130 The following query updates all the duplicate orders assuming that *id* is not a primary key:

```
UPDATE orders A
WHERE EXISTS (SELECT 1 FROM orders B
WHERE A.id = B.id and A.rowid < B.rowid);
```

Join Update

TimesTen supports “join update” statements. A join update can be used to update one or more columns of a table using the result of a subquery.

Syntax

UPDATE [*Owner.*]*TableName*
SET *ColumnName*=*Subquery*
[WHERE *SearchCondition*]

or

UPDATE [*Owner.*]*TableName*
SET (*ColumnName*[,…])=*Subquery*
[WHERE *SearchCondition*]

Parameters

The UPDATE statement has the parameters:

[*Owner.*]*TableName*	[*Owner.*]*TableName* identifies a table to be updated.
SET (*ColumnName*[,...])= *Subquery*	Column to be updated. You can update several columns of the same table with a single UPDATE statement. The SET clause can contain only one subquery, although this subquery can be nested. The number of values in the *SelectList* of the subquery must be the same as the number of columns specified in the SET clause. An error is returned if the subquery returns more than one row for any updated row.
WHERE *SearchCondition*	The search condition can contain a subquery. All rows for which the search condition is TRUE are updated as specified in the SET clause. Rows that do not satisfy the search condition are not affected. If no rows satisfy the search condition, the table is not changed.

Description

The subquery in the SET clause of a join update does not reduce the number of rows from the target table that are to be updated. The reduction must be done by specifying the WHERE clause. Thus, if a row from the target table qualifies the WHERE clause but the subquery returns no rows for this row, this row is updated with NULL value in the updated column.

Example 5.131

```
UPDATE t1 SET x1=(SELECT x2 FROM t2 WHERE id1=id2);
UPDATE t1 SET (x1,y1)=(SELECT x2,y2 FROM t2 WHERE id1=id2);
```

If a row from t1 has no match in t2, its x1 value in the first select and its x1,y1 values in the second select is set to NULL.

Example 5.132 In order to restrict the update to update only rows from t1 that have a match in t2, a where clause with subquery has to be provided as follows:

```
UPDATE t1 SET x1=(SELECT x2 FROM t2 WHERE id1=id2) WHERE id1 IN (SELECT
id2 FROM t2);
UPDATE t1 SET (x1,y1)=(SELECT x2,y2 FROM t2 WHERE id1=id2) WHERE
id1 IN (SELECT id2 FROM t2);
```

See Also "SELECT" on page 302

6

System and Replication Tables

TimesTen stores metadata (information about the contents of your data store) in *system tables* in your data store.

Your applications can read but cannot update the system tables. If your application defines a table with the same name as a system table name, then your application can read a system table by prefixing the name with `SYS.` For example, `SELECT * FROM SYS.TABLES` selects rows from the TABLES system table. Use the TTREP prefix when using the replication tables.

Information specific to system tables:

- Locks acquired by users on system tables may prevent others from defining data or executing the ODBC function **SQLPrepare or the JDBC method Connection.prepareStatement**.
- The last character in name columns is always a space. Therefore, while the column length of name columns is 31, the maximum object name length is 30.
- 

On 64-bit systems, TimesTen system tables declare certain fields as data type TT_BIGINT. When retrieving these columns with an ODBC program, the application must bind them using SQL_C_BINARY.
For information on SQL_C_BINARY, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide.*

Note: Some tables contain columns, named SYS*number.* Because these columns contain values used internally by TimesTen, they are not documented in this chapter.

System table list

Table	See
SYS.CACHE_GROUP	page 331
SYS.COLUMNS	page 333
SYS.COLUMN_HISTORY	page 330
SYS.COL_STATS	page 336
SYS.DUAL	page 337
SYS.INDEXES	page 338
SYS.MONITOR	page 340
SYS.OBJ_ACC_RIGHT	page 330
SYS.PLAN	page 347
SYS.SEQUENCES	page 350
SYS.SYNONYMS	page 352
SYS.SYS_ACC_RIGHT	page 330
SYS.TABLES	page 353
SYS.TABLE_HISTORY	page 330
SYS.TBL_STATS	page 357
SYS.TCOL_STATS	page 358
SYS.TINDEXES	page 359
SYS.TRANSACTION_LOG_API	page 361
SYS.TTABLES	page 362
SYS.TTBL_STATS	page 366
SYS.USERS	page 330
SYS.VIEWS	page 367
SYS.XLASUBSCRIPTIONS	page 368

Replication table list

Table	See
TTREP.REPELEMENTS	page 369
TTREP.REPLICATIONS	page 373
TTREP.REPPEERS	page 374
TTREP.REPSTORES	page 378
TTREP.REPSUBSCRIPTIONS	page 379
TTREP.REPTABLES	page 381
TTREP.TTSTORES	page 385

Tables reserved for internal or future use

Tables reserved for internal use are system tables that are defined in your data store and are used internally by TimesTen:

- SYS.COLUMN_HISTORY
- SYS.TABLE_HISTORY

Tables reserved for future use are system tables that are defined in your data store and are reserved for future use by TimesTen:

- SYS.OBJ_ACC_RIGHT
- SYS.SYS_ACC_RIGHT
- SYS.USERS

SYS.CACHE_GROUP

The CACHE_GROUP table describes the definition of a TimesTen cache.

Table name CACHE_GROUP

Columns

Column name	Type	Description
CGNAME	TT_CHAR (31) NOT NULL	Group name.
CGOWNER	TT_CHAR (31) NOT NULL	Group owner.
CGID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Id of this cache group.
ROOT	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Unique identifier for cache group's root table
SOURCE	TT_CHAR (8) NOT NULL	Data source for caching. In this release the only legal value is ‘ORACLE’.
CGDURATION	TT_INTEGER NOT NULL	Duration
TBLCNT	TT_SMALLINT NOT NULL	Number of tables in cache group.
REFRESH_MODE	TT_CHAR(1) NOT NULL	The current auto refresh mode. ‘N’: No auto refresh. ‘I’: Incremental auto refresh. ‘F’: Full auto refresh.
REFRESH_STATE	TT_CHAR(1) NOT NULL	The current auto refresh mode. ‘N’: Off. ‘Y’: On. ‘P’: Paused.

REFRESH_INTERVAL	TT_BIGINT NOT NULL	Auto refresh interval in milliseconds.
CGATTRIBUTES	BINARY (4) NOT NULL	Bits 0-7 are for cache group types. Bits 8-15 are for autoload options. Bit 0: 1 - READONLY Bit 1: 1 - SYNCHRONOUS WRITETHROUGH Bit 2: 1 - AUTOREFRESH Bit 3: 1 - PROPAGATE Bit 8: 1 - Autoload on Create (Always 1 for AUTOREFRESH) Bit 9: 1 - Autoload on Demand (Only be available with SYNCHRONOUS WRITETHROUGH)
REFRESH_WITH_LIMIT	TT_INTEGER NOT NULL	The maximum number of autorefresh change log records kept in the trigger log table at Oracle. A larger value causes the autorefresh to use more space at Oracle, while it prevents the truncation of logs that are not autorefreshed to TimesTen yet, and therefore reduces the possible fallback to full refresh. The field is used only by incremental autorefresh

SYS.COLUMNS

The COLUMNS table describes every column in every table in the data store, including the name of the column, the type of the column and whether the column is nullable.

Table name COLUMNS

Columns

Column name	Type	Description
ID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of column's table.
COLNUM	TT_SMALLINT NOT NULL	Ordinal number of column in table (starting at 1).
COLNAME	TT_CHAR (31) NOT NULL	Column name.
COLOPTIONS	BINARY(1) NOT NULL	Column specification flags: 0x01 - column is in a primary key. 0x02 - column value is varying-length (VARCHAR[2],NVARCHAR[2], VARBINARY). 0x04 - column value can be NULL. 0x08 - column values are unique.

COLTYPE	TT_INTEGER NOT NULL	Data type of column 1 TT_SMALLINT 2 TT_INTEGER 3 BINARY_FLOAT 4 BINARY_DOUBLE 5 TT_CHAR 6 TT_VARCHAR 7 BINARY 8 VARBINARY 11 TT_DECIMAL 12 TT_NCHAR 13 TT_NVARCHAR 14 TT_DATE 15 TIME 16 TT_TIMESTAMP 20 TT_TINYINT 21 TT_BIGINT 22 TT_VARCHAR (inline) 23 VARBINARY (inline) 24 TT_NVARCHAR (inline) 25 NUMBER 26 CHAR 27 VARCHAR2 28 NCHAR 29 NVARCHAR2 30 DATE 31 TIMESTAMP 32 VARCHAR2 (inline) 33 NVARCHAR2 (inline)
TYPE_ATTR	TT_INTEGER NOT NULL	Reserved for internal use.
COLLEN	INTEGER NOT NULL for 32-bit systems; BIGINT NOT NULL for 64-bit systems	Length of the column (maximum length for varying-length columns).
INLINELEN	TT_INTEGER NOT NULL	Identifies how many bytes a given column contributes to the inline width of a row.

REPUSERID	TT_INTEGER NOT NULL	User-defined identifier for column (set with ttSetUserColumnID built-in function).
DEFAULTVALSTR	TT_VARCHAR (409600) NOT INLINE	The default column value.
CHAR_USED	TT_CHAR (1)	Indicates the semantics for the column: ‘B’ for BYYE ‘C’ for CHAR NULL for non-character columns

SYS.COL_STATS

The COL_STATS table stores the statistics for table columns in the data store. Statistics include the number of unique values, number of nulls, number of rows and other information regarding the distribution of column values. No values are present if statistics have not been computed.

Table name COL_STATS

Columns

Column name	Type	Description
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen table identifier.
COLNUM	TT_SMALLINT NOT NULL	Ordinal number of column in table (starting at 1).
INFO	VARBINARY (4000000) NOT INLINE NOT NULL	Contains a binary representative of the column value distribution information. See **ttOptUpdateStats** for an explanation of the distribution information stored in this column. A text representation of this information can be retrieved using **ttOptGetColStats**.

SYS.DUAL

The DUAL table can be used in a SELECT statement that references no other tables, but needs to return at least one row. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once.

Table name DUAL

Columns

Column name	Type	Description
DUMMY	TT_VARCHAR (1) NOT INLINE NOT NULL	'X'

SYS.INDEXES

The INDEXES table stores information about the indexes in the data store, including the name, the type (T-tree or hash), the index key and whether the index is unique.

Table name INDEXES

Columns

Column name	Type	Description
IXNAME	TT_CHAR (31) NOT NULL	Index name.
IXOWNER	TT_CHAR (31) NOT NULL	Name of index's owner.
IXID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of index.
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of index's table.
IXTYPE	TT_INTEGER NOT NULL	Index type: 0 - hash index. 1 - T-tree index.
ISUNIQUE	BINARY(1) NOT NULL	Uniqueness: 0 - nonunique index. 1 - unique index.
ISPRIMARY	BINARY(1) NOT NULL	Primary key: 0 - not a primary key for table. 1 - primary key for table.
USETMPHEAP	TT_SMALLINT NOT NULL	Reserved for internal use.

Column name	Type	Description
KEYCNT	TT_SMALLINT NOT NULL	Number of columns in the index key.
KEYCOLS	BINARY(32) NOT NULL	Array of 2-byte INTEGER column numbers of index key, mapped to BINARY.
PAGESPARAM	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Number of pages specified for hash index.
NLSSORTID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	For internal use only.
NLSSORTPARM	VARBINARY (1000) NOT INLINE	For internal use only.
NLSSORTSTR	TT_VARCHAR(200) NOT INLINE	For internal use only.
NLSSORTBUFSIZE	TT_SMALLINT	For internal use only.
NLSSORTMAXSIZE	TT_SMALLINT NOT NULL	For internal use only.

SYS.MONITOR

The MONITOR table stores information about system performance. It contains a single row with statistics about certain events. For many columns, statistics are gathered starting from the first connection to the data store. For some columns, statistics are gathered as needed. TimesTen does not gather statistics from the time of the first connection for these columns:

- PERM_ALLOCATED_SIZE
- PERM_IN_USE_SIZE
- TEMP_ALLOCATED_SIZE
- LAST_LOG_FILE
- REPHOLD_LOG_FILE
- REPHOLD_LOG_OFF
- FIRST_LOG_FILE
- CHECKPOINT_BYTES_WRITTEN

For most columns, the MONITOR table is reset whenever there are no connections to the data store. TimesTen does not reset the values of the following columns, even when there are no connections to the data store:

- PERM_ALLOCATED_SIZE
- PERM_IN_USE_SIZE
- TEMP_ALLOCATED_SIZE
- LAST_LOG_FILE
- REPHOLD_LOG_FILE
- REPHOLD_LOG_OFF
- FIRST_LOG_FILE

Information in the MONITOR table is frequently updated by TimesTen. To prevent these updates from slowing down the system, they are not protected by latches. Hence values in the MONITOR table are not absolutely accurate. They can be used as a reliable indication of various activities in the system.

Table name MONITOR

Columns

Column name	Type	Description
TIME_OF_1ST_CONNECT	TT_CHAR(32) NOT NULL	Time at which the first connection was made.
DS_CONNECTS	TT_INTEGER NOT NULL	Number of connects to the data store.
DS_DISCONNECTS	TT_INTEGER NOT NULL	Number of disconnects from the data store.
DS_CHECKPOINTS	TT_INTEGER NOT NULL	Number of checkpoints taken.
DS_CHECKPOINTS_ FUZZY	TT_INTEGER NOT NULL	Number of fuzzy checkpoints taken.
DS_COMPACTS	TT_INTEGER NOT NULL	Number of data store compactions.
PERM_ALLOCATED_SIZE	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Allocated size in kilobytes of the permanent data partition
PERM_IN_USE_SIZE	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Size in kilobytes of the portion of the permanent data partition that is currently in use.
PERM_IN_USE_HIGH_ WATER	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	The highest amount (in kilobytes) of permanent data partition memory in use since the first connection to the data store.

Column name	Type	Description
TEMP_ALLOCATED_SIZE	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Allocated size in kilobytes of the temporary data partition
TEMP_IN_USE_SIZE	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Size in kilobytes of the portion of the temporary data partition that's currently in use.
TEMP_IN_USE_HIGH_WATER	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	The highest amount (in kilobytes) of temporary data partition memory in use since the first connection to the data store.
XACT_BEGINS	TT_BIGINT NOT NULL	Number of transactions started.
XACT_COMMITS	TT_BIGINT NOT NULL	Number of durable and non-durable transactions committed.
XACT_D_COMMITS	TT_BIGINT NOT NULL	Number of transactions committed durably.
XACT_ROLLBACKS	TT_BIGINT NOT NULL	Number of transactions rolled back.
LOG_FORCES	TT_BIGINT NOT NULL	Number of log flushes to disk.
DEADLOCKS	TT_BIGINT NOT NULL	Number of deadlocks.

Column name	Type	Description
LOCK_TIMEOUTS	TT_BIGINT NOT NULL	Number of lock requests denied due to timeouts.
LOCK_GRANTS_ IMMED	TT_BIGINT NOT NULL	Number of lock requests granted without a wait.
LOCK_GRANTS_WAIT	TT_BIGINT NOT NULL	Number of lock requests granted after a wait.
CMD_PREPARES	TT_BIGINT NOT NULL	Number of commands prepared (compiled).
CMD_REPREPARES	TT_BIGINT NOT NULL	Number of commands re-prepared.
CMD_TEMP_INDEXES	TT_BIGINT NOT NULL	Number of temporary indexes created during query execution.
LAST_LOG_FILE	TT_INTEGER NOT NULL	Number of last log file.
REPHOLD_LOG_FILE	TT_INTEGER NOT NULL	Number of last log file held by replication.
REPHOLD_LOG_OFF	TT_INTEGER NOT NULL	Offset in last log file held by replication.
REP_XACT_COUNT	TT_INTEGER NOT NULL	The number of replicated transactions generated on the local store that are being replicated to at least one peer data store.
REP_CONFLICT_COUNT	TT_INTEGER NOT NULL	The number of replicated transactions that ran into a conflict when being applied on the local store.

Column name	Type	Description
REP_PEER_ CONNECTIONS	TT_INTEGER NOT NULL	The sum of all peer connections initiated by the local replication agent. There will be one connection for every peer relationship where the local store is the master. If a transport level failure results in the establishment of a new connection this count will be incremented.
REP_PEER_RETRIES	TT_INTEGER NOT NULL	The number of retry attempts while trying to establish a new peer connection.
FIRST_LOG_FILE	TT_INTEGER NOT NULL	The number of the oldest existing (not yet purged) log file.
LOGBYTES_TO_LOG_ BUFFER	TT_BIGINT NOT NULL	The number of bytes written to the log since first connect. This value includes the sizes of actual log records plus any log overhead.
LOG_FS_READS	TT_BIGINT NOT NULL	The number of times that a log read could not be satisfied from the in-memory log buffer.

Column name	Type	Description
LOG_FS_WRITES	TT_BIGINT NOT NULL	The number of times TimesTen wrote the contents of the in-memory log buffer to the operating system. This column does not count the number of times data was flushed to disk. It only counts writes to the operating system's file buffers.
LOG_BUFFER_WAITS	TT_BIGINT NOT NULL	The number of times a thread was delayed while trying to insert a log record into the log buffer because the log buffer was full. Generally speaking, if this value is increasing, it indicates that the log buffer is too small.
CHECKPOINT_BYTES_WRITTEN	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	The number of bytes written to disk by the most recent checkpoint operation.

Column name	Type	Description
REQUIRED_RECOVERY	TT_INTEGER NOT NULL	If 1, when the data store was initially loaded into RAM at TIME_OF_1ST_CONNECT, recovery ran. This means that the previous time the data store was in memory, the data store did not shut down cleanly. When it was loaded into memory this time, the log was replayed and other operations were performed in an attempt to recover data; If DurableCommit had been set to 0, transactions could have been lost. If 0, the data store was previously shut down cleanly. As a result, this time the data store was restarted cleanly.
TYPE_MODE	TT_INTEGER NOT NULL	If 0, Oracle mode; If 1, TimesTen mode.

SYS.PLAN

The PLAN table contains the execution plan that the TimesTen query optimizer prepares after an application calls **ttOptSetFlag** (see "Generating the plan"and "Modifying plan generation"in the *Oracle TimesTen In-Memory Database Operations Guide*).

The execution plan includes the operation performed at each step and the table or index that it references.

Table name PLAN

Columns

Column name	Type	Description
STEP	TT_INTEGER NOT NULL	Ordinal number of the operation, starting at 1.
LEVEL	TT_INTEGER NOT NULL	Level of this operation in the plan tree.

Column name	Type	Description
OPERATION	TT_CHAR (31) NOT NULL	Type of operation, one of: `TblLkSerialScan` -- full table scan `RowLkSerialScan` -- full table scan `TblLkTtreeScan` -- ttree scan `RowLkTtreeScan` -- ttree scan `TblLkHashScan` -- hash lookup `RowLkHashScan` -- hash lookup `TblLkRowidScan` -- rowid lookup `RowLkRowidScan` -- rowid lookup `TblLkUpdate` -- updates one or more rows `RowLkUpdate` -- updates one or more rows `TblLkDelete` -- deletes one or more rows `RowLkDelete` -- deletes one or more rows `TblLkInsert` -- inserts one or more rows `RowLkInsert` -- inserts one or more rows `TmpTtreeScanTmpHashScan` -- create a temporary index `NestedLoop [OuterJoin | SemiJoin]` -- nested loop join (with optional outer join or semi-join) `MergeJoin` -- merge join `OrderBy` -- sorts rows (requires extra temp space) `SortedDistinct` -- identifies distinct rows from a sorted list (requires minimal extra space) `Distinct` -- identifies distinct rows from an unsorted list (requires extra temporary space) `SortedGroupBy` -- identifies distinct groups from a sorted list (requires minimal extra space) `GroupBy` -- identifies distinct groups from an unsorted list (requires extra temp space) `TmpTable` -- materializes intermediate results (requires extra temporary space) `TblLkUpdView` -- updates a view based on changes to detail table(s) `RowLkUpdView` -- updates a view based on changes to detail table(s) `OracleInsert` -- flushes changes to Oracle `ZeroTblScan` -- evaluates a predicate on a single set of values (no scan required) `ViewUniqueMatchScan` -- uniquely identifies those view rows that need to be updated (requires extra temp space

Column name	Type	Description
TBLNAME	TT_CHAR (31)	Name of table scanned at this step. Column is NULL if no table is scanned.
IXNAME	TT_CHAR (31)	Name of index used at this step. T-tree index names may have a “(D)” after the name, which indicates a descending scan. Column is NULL if no index is scanned.
PRED	TT_VARCHAR (1024)	Predicate applied during table or index scan or join. Column is NULL if no predicate applies.
OTHERPRED	TT_VARCHAR (1024)	Predicate applied after table or index scan or join. Column is NULL if no predicate applies.

SYS.SEQUENCES

The SEQUENCES table contains all the information about sequences. Data from the system table is restored to the new data store during a CREATE SEQUENCE statement.

Table name SEQUENCES

Columns

Column name	Type	Descriptions
NAME	TT_CHAR(31) NOT NULL	Sequence Name
OWNER	TT_CHAR(31) NOT NULL	Sequence Owner
MINVAL	TT_BIGINT NOT NULL	Minimum Value
MAXVAL	TT_BIGINT NOT NULL	Maximum Value
INCREMENT	TT_BIGINT NOT NULL	Increment value
CACHESIZE	TT_BIGINT NOT NULL	Number of sequence number to be cached. For internal TimesTen use.
LASTNUMBER	TT_BIGINT NOT NULL	Last number incremented.
SEQID	TT_INTEGER NOT NULL on 32-bit systems; TT_BIGINT NOT NULL on 64-bit systems	ID of the sequence row
CYCLE	BINARY(1) NOT NULL	Flag to indicate to wrap around value.

IS_REPLICATED	BINARY(1) NOT NULL	0 – Sequences are not being replicated 1 – Sequences are being replicated
REPACCESS	TT_CHAR(1) NOT NULL	Flag to indicate that sequences cannot be incremented on subscriber only data stores.

SYS.SYNONYMS

The SYNONYMS table contains information about synonyms.

Table name SYNONYMS

Columns

Column name	Type	Descriptions
NAME	TT_CHAR(31) NOT NULL	Reserved for future use.
OWNER	TT_CHAR(31) NOT NULL	Reserved for future use.
OBJNAME	TT_CHAR(31) NOT NULL	Reserved for future use.
OBJOWNER	TT_CHAR(31)	Reserved for future use.

SYS.TABLES

The TABLES table stores information about the tables in the data store, including the name, the owner, the number of columns, the size of a row and the primary key (if any). The TABLES table also stores information on system tables.

Specific column information is stored in the COLUMNS table.

Table name TABLES

Columns

Column name	Type	Descriptions
TBLNAME	TT_CHAR(31) NOT NULL	Table name.
TBLOWNER	TT_CHAR(31) NOT NULL	Name of user who owns the table.
OWNER	TT_INTEGER NOT NULL	Owner of table: 0 - TimesTen system table. 1 - User table.
NUMVARY	TT_SMALLINT NOT NULL	Number of varying-length columns in table.
NUMNULL	TT_SMALLINT NOT NULL	Number of nullable columns in table.
NUMCOLS	TT_SMALLINT NOT NULL	Number of columns in table.
LENGTH	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Length of in-line portion of each row.
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier for table.

NUMTUPS	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Table cardinality. This value is precise only when no INSERT or DELETE transactions are active. The value includes uncommitted inserts, but not uncommitted deletes. Consequently, the value of this field may be larger than the actual table cardinality.
MAXTUPS	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Maximum table cardinality.
PRIMCNT	TT_SMALLINT NOT NULL	Number of columns in primary key (0 if none).
PRIMCOLS	BINARY(32) NOT NULL	Array of 2-byte INTEGER column numbers of primary key, mapped to BINARY.
CACHEFLAG	BINARY(1) NOT NULL	1 - if the table is in a cache group, 0 otherwise.
XLAFLAG	BINARY(1) NOT NULL	If set, updates to this table should be transmitted to the transaction log API.
PXLAFLAG	BINARY(1) NOT NULL	If set, indicates that persistent XLA has been enabled for this particular user table.
CACHEGROUP	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Id of cache group that this table belongs to.

MVID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	If the table is a VIEW, indicates the ID of the associated row in the VIEWS system table
MVIDS	TT_VARCHAR(1024) NOT INLINE	If the table is a VIEW detail table, indicates the ID of the array of the Ids of the rows in the VIEWS system table of the materialized views that reference this detail table.
PERMLTBLID	TT_INTEGER NOT NULL	The associated permanent table's ID.
REPNUMKEY COLS	TT_SMALLINT NOT NULL	Number of columns in the replication key described by REPKEYCOLS
REPTSCOLNUM	TT_SMALLINT NOT NULL	Column number of the column used for replication's timestamp-based conflict checking.
REPRETURN SERVICE	TT_CHAR(1) NOT NULL	Return service for this subscriber with respect to this replication element: ‘C’ - RETURN COMMIT ‘R’ - RETURN RECEIPT ‘2’ - RETURN TWOSAFE ‘\0’ - NO RETURN services
REPRETURNBY REQUEST	BINARY(1) NOT NULL	0 - RETURN services are provided unconditionally 1 - RETURN services are provided only BY REQUEST. This field is ignored if REPRETURNSERVICE = '\0'
REPUSERID	TT_BIGINT NOT NULL	User-defined identifier for table (set with ttSetUserTableID built-in function).

REPKEYCOLS	BINARY(32) NOT NULL	Column numbers used by replication for unique identification of a row. (an array of 2-byte INTEGERs, mapped to BINARY)
REPACCESS	TT_CHAR(1) NOT NULL	The access restrictions imposed by replication: ‘-’ - no access permitted ‘s’- may be read by read-only (SELECT) transactions ‘r’ - may be read by updating transactions ‘w’ - may be updated w => r and r => s.
REPTSUPDATE RULE	TT_CHAR(1) NOT NULL	The rule for maintaining the TS_COLUMN for a timestamp-based conflict detector: ‘\0‘ - rule not defined ‘U‘ - BY USER ‘S‘ - BY SYSTEM (default)

SYS.TBL_STATS

The TBL_STATS table stores the statistics for tables in the data store, namely the number of rows in the table. No values are present if the statistics have not been computed.

Column-specific statistics are stored in the COL_STATS table. See "SYS.COL_STATS" on page 336.

Table name TBL_STATS

Columns

Column name	Type	Description
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of table.
NUMTUPS	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Number of rows in the table.
LASTSTATSUPDATE	TT_CHAR(25)	Time of most recent update of this table, in the following format: Day Mon DD HH:MM:SS YYYY (e.g., Sun Jan 01 18:24:12 1995). The string is NULL-terminated. This column is NULL if no statistics update has been performed on the table.

SYS.TCOL_STATS

The TCOL_STATS table stores the statistics for table columns in temporary table instances associated with active sessions. Statistics include the number of unique values, number of nulls, number of rows and other information regarding the distribution of column values. No values are present if statistics have not been computed.

Table name TCOL_STATS

Columns

Column name	Type	Description
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen table identifier.
COLNUM	TT_SMALLINT NOT NULL	Ordinal number of column in table (starting at 1).
INFO	VARBINARY (4000000) NOT NULL NOT INLINE	Contains a binary representative of the column value distribution information. See **ttOptUpdateStats** for an explanation of the distribution information stored in this column. A text representation of this information can be retrieved using **ttOptGetColStats**.

SYS.TINDEXES

The INDEXES table stores information about the indexes in the temporary table instances associated with active sessions, including the name, the type (T-tree or hash), the index key and whether the index is unique.

Table name TINDEXES

Columns

Column name	Type	Description
IXNAME	TT_CHAR (31) NOT NULL	Index name.
IXOWNER	TT_CHAR (31) NOT NULL	Name of index's owner.
IXID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of index.
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of index's table.
IXTYPE	TT_INTEGER NOT NULL	Index type: 0 - hash index. 1 - T-tree index.
ISUNIQUE	BINARY(1) NOT NULL	Uniqueness: 0 - nonunique index. 1 - unique index.
ISPRIMARY	BINARY(1) NOT NULL	Primary key: 0 - not a primary key for table. 1 - primary key for table.
USETMPHEAP	TT_SMALLINT NOT NULL	

Column name	Type	Description
KEYCNT	TT_SMALLINT NOT NULL	Number of columns in the index key.
KEYCOLS	BINARY(32) NOT NULL	Array of 2-byte INTEGER column numbers of index key, mapped to BINARY.
PAGESPARAM	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Number of pages specified for hash index.
NLSSORTID	TT_INTEGER NOT NULL	For internal use only.
NLSSORTPARM	VARBINARY(1000) NOT INLINE	For internal use only.
NLSSORTSTR	TT_VARCHAR(200) NOT INLINE	For internal use only.
NLSSORTBUF SIZE	TT_SMALLINT	For internal use only.
NLSSORTMAX SIZE	TT_SMALLINT	For internal use only.

SYS.TRANSACTION_LOG_API

The TRANSACTION_LOG_API table keeps track of the persistent Transaction Log API bookmarks. Each row in the system table corresponds to a persistent bookmark. Each persistent bookmark has a text identifier associated with it, which is used to keep track of the bookmark.

Table name TRANSACTION_LOG_API

Columns

Column name	Type	Description
ID	TT_CHAR(31) NOT NULL	A text tag identifier used to keep track of the bookmark.
READLSNHIGH	TT_INTEGER NOT NULL	The high value of the read log record to which this bookmark points.
READLSNLOW	TT_INTEGER NOT NULL	The low value of the read log record to which this bookmark points.
PURGELSNHIGH	TT_INTEGER NOT NULL	The high value of the lowest LSN required by this bookmark.
PURGELSNLOW	TT_INTEGER NOT NULL	The low value of the lowest LSN required by this bookmark.
PID	TT_INTEGER NOT NULL	The process ID of the process to last open the XLA bookmark.
INUSE	BINARY (1) NOT NULL	Bookmark being used by any persistent Transaction Log API connection.

SYS.TTABLES

The TABLES table stores information about temporary table instances associated with active sessions, including the name, the owner, the number of columns, the size of a row and the primary key (if any). The TABLES table also stores information on system tables.

Specific column information is stored in the COLUMNS table.

Table name TTABLES

Columns

Column name	Type	Descriptions
TBLNAME	TT_CHAR(31) NOT NULL	Table name.
TBLOWNER	TT_CHAR(31) NOT NULL	Name of user who owns the table.
OWNER	TT_INTEGER NOT NULL	Owner of table: 0 - TimesTen system table. 1 - User table.
NUMVARY	TT_SMALLINT NOT NULL	Number of varying-length columns in table.
NUMNULL	TT_SMALLINT NOT NULL	Number of nullable columns in table.
NUMCOLS	TT_SMALLINT NOT NULL	Number of columns in table.
LENGTH	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Length of in-line portion of each row.
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier for table.

NUMTUPS	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Table cardinality. This value is precise only when no INSERT or DELETE transactions are active. The value includes uncommitted inserts, but not uncommitted deletes. Consequently, the value of this field may be larger than the actual table cardinality.
MAXTUPS	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Maximum table cardinality.
PRIMCNT	TT_SMALLINT NOT NULL	Number of columns in primary key (0 if none).
PRIMCOLS	BINARY(32) NOT NULL	Array of 2-byte INTEGER column numbers of primary key, mapped to BINARY.
CACHEFLAG	BINARY(1) NOT NULL	1 - if the table is in a cache group, 0 otherwise.
XLAFLAG	BINARY(1) NOT NULL	If set, updates to this table should be transmitted to the transaction log API.
PXLAFLAG	BINARY(1) NOT NULL	If set, indicates that persistent XLA has been enabled for this particular user table.
CACHEGROUP	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Id of cache group that this table belongs to.

MVID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	If the table is a VIEW, indicates the ID of the associated row in the VIEWS system table
MVIDS	TT_VARCHAR(1024) NOT INLINE	If the table is a VIEW detail table, indicates the ID of the array of the Ids of the rows in the VIEWS system table of the materialized views that reference this detail table.
PERMLTBLID	TT_INTEGER NOT NULL	The associated permanent table's ID.
REPNUMKEYCOLS	TT_SMALLINT NOT NULL	Number of columns in the replication key described by REPKEYCOLS
REPTSCOLNUM	TT_SMALLINT NOT NULL	Column number of the column used for replication's timestamp-based conflict checking.
REPRETURNSERVICE	TT_CHAR(1) NOT NULL	Return service for this subscriber with respect to this replication element: ‘C’ - RETURN COMMIT ‘R’ - RETURN RECEIPT ‘2’ - RETURN TWOSAFE ‘\0’ - NO RETURN services
REPRETURNBY REQUEST	BINARY(1) NOT NULL	0 - RETURN services are provided unconditionally 1 - RETURN services are provided only BY REQUEST. This field is ignored if REPRETURNSERVICE = '\0'
REPUSERID	TT_BIGINT NOT NULL	User-defined identifier for table (set with ttSetUserTableID built-in function).

REPKEYCOLS	BINARY(32) NOT NULL	Column numbers used by replication for unique identification of a row. (an array of 2-byte INTEGERs, mapped to BINARY)
REPACCESS	TT_CHAR(1) NOT NULL	The access restrictions imposed by replication: ‘-’ - no access permitted ‘s’- may be read by read-only (SELECT) transactions ‘r’ - may be read by updating transactions ‘w’ - may be updated w => r and r => s.
REPTSUPDATERULE	TT_CHAR(1) NOT NULL	The rule for maintaining the TS_COLUMN for a timestamp-based conflict detector: ‘\0‘ - rule not defined ‘U‘ - BY USER ‘S‘ - BY SYSTEM (default)

SYS.TTBL_STATS

The TTBL_STATS table stores the statistics for temporary table instances associated with active sessions, namely the number of rows in the table. No values are present if the statistics have not been computed.

Column-specific statistics are stored in the COL_STATS table.
See "SYS.COL_STATS" on page 336.

Table name TTBL_STATS

Columns

Column name	Type	Description
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	TimesTen identifier of table.
NUMTUPS	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	Number of rows in the table.
LASTSTATSUPDATE	TT_CHAR(25)	Time of most recent update of this table, in the following format: Day Mon DD HH:MM:SS YYYY (e.g., Sun Jan 01 18:24:12 1995). The string is NULL-terminated. This column is NULL if no statistics update has been performed on the table.

SYS.VIEWS

The VIEWS table stores the statistics for views in the data store.

Table name VIEWS

Columns

Column name	Type	Description
NAME	TT_CHAR(31) NOT NULL	View name.
OWNER	TT_CHAR(31) NOT NULL	View owner.
ID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	ID of the view row.
TBLID	TT_INTEGER NOT NULL for 32-bit systems; TT_BIGINT NOT NULL for 64-bit systems	ID of the view.
SQL	TT_VARCHAR(409600) NOT NULL NOT INLINE	View select statement.

SYS.XLASUBSCRIPTIONS

The XLASUBSCRIPTIONS table stores information needed for table subscriptions at the bookmark level.

Table name XLASUBSCRIPTIONS

Columns

Column name	Type	Description
BOOKMARK	TT_CHAR(31) NOT NULL	Bookmark Name.
TBLNAME	TT_CHAR(31) NOT NULL	The name of the subscribed table.
TBLOWNER	TT_CHAR(31) NOT NULL	Owner of the subscribed table.

TTREP.REPELEMENTS

The TTREP.REPELEMENTS table describes elements in a replication scheme. In this release, the only elements recorded are tables.

Table name REPELEMENTS

Columns

Column name	Type	Description
REPLICATION_ NAME	TT_CHAR(31) NOT NULL	Name for a replication scheme.
REPLICATION_ OWNER	TT_CHAR(31) NOT NULL	The replication scheme's owner.
ELEMENT_NAME	TT_CHAR(31) NOT NULL	The replication name for this element, logically different from the DS_OBJ_NAME of the underlying data base object. For example, the ELEMENT_NAME for a replicated table may differ from the table name. This name must be unique in a replication scheme.
ELEMENT_TYPE	TT_CHAR(1) NOT NULL	The type of this replication element: 'T' – Table 'D' – Data store 'S' – Sequence
OWNED_BY_ SYSTEM	BINARY(1) NOT NULL	0x01 - Element is maintained by the system and cannot be directly referenced by SQL statements. 0x00 - Element is defined and maintained by a user.
MASTER_ID	TT_BIGINT NOT NULL	The TT_STORE_ID for the master or propagator of this element.

OLD_MASTER_ID	TT_BIGINT NOT NULL	The TT_STORE_ID for the immediately preceding MASTER for this element; -1 if none.
IS_PROPAGATOR	BINARY(1) NOT NULL	0 if the MASTER_ID identifies a true MASTER store; 1 if it, in fact, identifies a PROPAGATOR.
DS_OBJ_NAME	TT_CHAR(31) NOT NULL	If this replication refers to a single, underlying data base object, then this is its name. Specifically, it is the name of the replicated table if ELEMENT_TYPE = 'T'; it is NULL if ELEMENT _TYPE = 'D'. DS_OBJ_OWNER._DS_OBJ_ NAME need not be unique in a replication scheme, but each occurrence must be associated with a distinct ELEMENT_NAME.
DS_OBJ_OWNER	TT_CHAR(31) NOT NULL	The owner of the replication element – if defined; NULL otherwise. This is always the owner of the table. DS_OBJ_OWNER._DS_ OBJ_NAME need not be unique in a replication scheme, but each occurrence must be associated with a distinct ELEMENT_NAME.

DS_OBJ_ID	TT_INTEGER	If the ELEMENT_TYPE = 'T': Table ID - Table is in the owning (master or propagator) data store. 1- Table is in the subscriber data store. If the ELEMENT_TYPE = 'D': 0 - Data store is a master or propagator. 1- Data store is a subscriber. NULL - If the store has been migrated, restored or upgraded from an earlier version.
DURABLE_ TRANSMIT	BINARY(1) NOT NULL	0 - Transactions are made durable before they are transmitted (default). 1 - Transactions are not made durable before they are transmitted.
CONFLICT_CHECKS	BINARY(8) NOT NULL	A bit map indicating which conflict detectors are enabled. This field is either: 0x0000000000000000 (no configured conflict detector, the default) or: 0x0000000000000001 (ROW TIMESTAMP conflict detector).
TS_COLUMN_NAME	TT_CHAR(31)	The name of the timestamp column specified in the *CheckConflicts* portion of a CREATE REPLICATION statement. This column must be of type BINARY(8) and permit NULL values.

TS_EXCEPTION_ ACTION	TT_CHAR(1) NOT NULL	The action to take upon detecting a conflict by a timestamp-based detector. The action is specified by the ON EXCEPTION clause in the *CheckConflicts* portion of a CREATE REPLICATION statement. They appear in this column as: '\0' - action not defined 'N' - NO ACTION 'R' - rollback transaction (default)
TS_UPDATE_RULE	TT_CHAR (1) NOT NULL	The rule for maintaining the timestamp for a timestamp-based conflict detector: '\0'- rule not defined 'U' - by user 'S' - by system (default)
TS_REPORT_FILE	TT_VARCHAR(1000) NOT INLINE	The name of the file to which the replication agent reports timestamp conflicts. This file is specified by the REPORT TO clause in the *CheckConflicts* portion of a CREATE REPLICATION statement.
IS_MASTER_ PROPAGATOR	BINARY(1) NOT NULL	Indicates if the store is both a master and a propagator.
EXTERNAL_DB	TT_CHAR (1)	
REPORT_FORMAT	TT_CHAR(1)	The report format for the replication conflict file: NULL - No report file specified therefore no format 'S' - Standard format 'X' - XML format

TTREP.REPLICATIONS

The REPLICATIONS table collects together general information about all replication schemes in which the local store participates. The table indicates whether a replication scheme was created by **ttRepAdmin** -upgrade or by a CREATE REPLICATION statement.

Table name REPLICATIONS

Columns

Column name	Type	Description
REPLICATION_NAME	TT_CHAR (31) NOT NULL	Name for a replication scheme.
REPLICATION_OWNER	TT_CHAR (31) NOT NULL	The replication scheme's owner.
REPLICATION_ORIGIN	TT_CHAR (1) NOT NULL	'U' - created by ttRepAdmin -upgrade 'C' - created by CREATE REPLICATION (or a ttRepAdmin command that was translated into CREATE REPLICATION).
REPLICATION_VERSION	TT_INTEGER NOT NULL	The number of ALTER REPLICATION commands applied to this replication scheme after its initial creation.
SOURCE_STORE_ID_ALIGN	TT_INTEGER NOT NULL	Used internally to properly align the SOURCE_STORE_ID column.
SOURCE_STORE_ID	TT_BIGINT NOT NULL	If this replication scheme was created by restoring it from a backup, the store ID of the store from which this replication scheme was backed up and restored; otherwise -1 (the invalid store ID).
CHECKSUM	TT_BIGINT	Indicates that the replication scheme has been updated.

TTREP.REPPEERS

The REPPEERS table displays status information about the stores in a replication scheme. After the initial upgrade, the REPPEERS table contains peer information only about the local store and other stores that it transmits updates to.

Table name REPPEERS

Columns

Column name	Type	Description
REPLICATION_NAME	TT_CHAR(31) NOT NULL	Name for a replication scheme.
REPLICATION_OWNER	TT_CHAR(31) NOT NULL	The replication scheme's owner
TT_STORE_ID	TT_BIGINT NOT NULL	The system-generated identifier for a transmitting store in a replication scheme. This store maintains all fields below SUBSCRIBER_ID except for COMMIT_TIMESTAMP and COMMIT0_SEQNUM.
SUBSCRIBER_ID	TT_BIGINT NOT NULL	The identifier for a store that subscribes to at least one replication element owned by TT_STORE_ID. If a valid ID then this record describes the status of TT_STORE_ID/ SUBSCRIBER_ID as a sender/ subscriber pair.
COMMIT_TIMESTAMP	TT_INTEGER	This field and COMMIT_SEQNUM together store the value of the Commit Ticket Number of the refreshed transaction that the subscriber has just committed.

Column name	Type	Description
COMMIT_SEQNUM	TT_INTEGER	This field and COMMIT_TIMESTAMP together store the value of the Commit Ticket Number of the refreshed transaction that the subscriber has just committed.
SENDLSNHIGH	TT_INTEGER	The log file number of the highest TT_STORE_ID log sequence number sent to and acknowledged by SUBSCRIBER_ID.
SENDLSNLOW	TT_INTEGER	The log file offset of the highest TT_STORE_ID log sequence number sent to and acknowledged by SUBSCRIBER_ID.
REPTABLESLSNHIGH	TT_INTEGER	For TimesTen internal use.
REPTABLESLSNLOW	TT_INTEGER	For TimesTen internal use.

Column name	Type	Description
STATE	TT_INTEGER	The state of replication kept by TT_STORE_ID with respect to this SUBSCRIBER_ID: 0 - START: Replication is in the active state and all log updates are retained until they have been applied at SUBSCRIBER_ID. 1 - PAUSE: Replication is not in the active state but all log updates are retained until they have been applied at SUBSCRIBER_ID. 2 - STOP: Replication is not in the active state and log updates are not retained. 4 - FAILED: Replication is not in the active state, log updates are not retained, and the log updates that need to be retained exceed the user defined threshold - TTREP.REPSTORES.FAIL_THRESHOLD. When this state has been communicated to SUBSCRIBER_ID it is changed to STOP.
TIMESEND	TT_INTEGER	The timestamp (in seconds) for the time of the last known successful transmission from TT_STORE_ID to SUBSCRIBER_ID.
TIMERECV	TT_INTEGER	The timestamp (in seconds) for the time TT_STORE_ID last received a transmission from SUBSCRIBER_ID.
PROTOCOL	TT_INTEGER	A number in the range 0 to 5 indicating the protocol level that replication uses for communication between TT_STORE_ID and SUBSCRIBER_ID. A higher number indicates a newer protocol.

Column name	Type	Description
LATENCY	BINARY_ DOUBLE	An estimate of the time interval (in seconds) from the commit of a transaction on TT_STORE_ID to its receipt of acknowledgement that it has been applied at the subscriber identified by SUBSCRIBER_ID.
TPS	TT_INTEGER	An estimate of the number of transactions per second that are committed on TT_STORE_ID and successfully received by the subscriber identified by SUBSCRIBER_ID.
RECSPERSEC	TT_INTEGER	An estimate of the number of records per second retrieved by the subscriber identified by SUBSCRIBER_ID from the store TT_STORE_ID.
DISKLESS_UNINITIALIZED	BINARY(1)	0 if TT_STORE_ID is either disk-based or is diskless and has not initialized SUBSCRIBER_ID for diskless replication; 1 otherwise.
CTNLISTINDEX	TT_INTEGER	For internal use by the replication agent.

TTREP.REPSTORES

The REPSTORES table lists the replication attributes of store's that participate in every TimesTen replication scheme in which the local store participates. Each store is identified by a unique TT_STORE_ID that TimesTen replication assigns to it. A TT_STORE_ID may appear at most once for a given replication scheme, but may appear multiple times in the REPSTORES table. Various replication schemes may define different replication store attributes for the same store

Table name REPSTORES

Columns

Column name	Type	Description
REPLICATION_NAME	TT_CHAR(31) NOT NULL	Name for a replication scheme.
REPLICATION_OWNER	TT_CHAR(31) NOT NULL	The replication scheme's owner
TT_STORE_ID	TT_BIGINT NOT NULL	System-generated identifier for a *HOST_NAME*, *TT_STORE_NAME* pair
PEER_TIMEOUT	TT_INTEGER NOT NULL	The number of seconds for this store to wait for a subscriber response before trying to reconnect.
FAIL_THRESHOLD	TT_INTEGER NOT NULL	The number of log files whose accumulation makes this store, in this replication scheme, mark subscribers "failed." (See the STATE field.)
HEARTBEAT_FACTOR	BINARY_ DOUBLE	A multiplier of the current heartbeat frequency.

TTREP.REPSUBSCRIPTIONS

The REPSBUBSCRIPTIONS registers each subscribing store that maintains a secondary copy of a replication element.

Table name REPSUBSCRIPTIONS

Columns

Column name	Type	Description
REPLICATION_NAME	TT_CHAR (31) NOT NULL	Name for a replication scheme.
REPLICATION_OWNER	TT_CHAR (31) NOT NULL	The replication scheme's owner.
ELEMENT_NAME	TT_CHAR(31) NOT NULL	The replication name for this element, logically distinct from the name of an underlying data store object.
SUBSCRIBER_ID	TT_BIGINT NOT NULL	The TT_STORE_ID for a subscriber to this element. A subscriber may not subscribe more than once to a replication element in a replication scheme.
RETURN_SERVICE	TT_CHAR(1) NOT NULL	Return service for this subscriber with respect to this replication element: ’C’ - RETURN COMMIT ’R’ - RETURN RECEIPT ‘\0’ - No RETURN services ‘2‘ - RETURN TWOSAFE

Column name	Type	Description
RETURN_BY_REQUEST	BINARY(1) NOT NULL	The type of return services for this element. 0 - RETURN services are provided unconditionally 1 - RETURN services are provided only BY REQUEST This field is ignored if RETURN_SERVICES = '\0'.
PRIVILEGES	TT_CHAR(1) NOT NULL	Privileges for this subscriber with respect to this replication element: '\0' - no special subscriber privileges

TTREP.REPTABLES

The REPTABLES table contains subscriber-relative information about each of the columns in each table transmitted to a subscriber. This information appears in REPTABLES in the owner (transmitter) store but not in REPTABLES in the subscriber store.

Table name REPTABLES

Columns

Column name	Type	Description
REPLICATION_NAME	TT_CHAR(31) NOT NULL	Name for a replication scheme.
REPLICATION_OWNER	TT_CHAR(31) NOT NULL	The replication scheme's owner.
ELEMENT_NAME	TT_CHAR(31) NOT NULL	The replication name for this element, logically different from the REF_NAME of the underlying data base object. For example, the ELEMENT_NAME for a replicated table may differ from the table name. This name must be unique in a replication scheme.
SUBSCRIBER_ID	TT_BIGINT NOT NULL	The TT_STORE_ID for a subscriber to this element. A subscriber may not subscribe more than once to a replication element in a replication scheme.
COLNUM	TT_SMALLINT NOT NULL	Ordinal number of column in table (starting at 1).

Column name	Type	Description
COLOPTIONS	BINARY(1) NOT NULL	Column specification flags: 0x01 - column is in a primary key. 0x02 - column value is varying-length (VARCHAR[2], NVARCHAR[2], VARBINARY) 0x04 - column value can be NULL. 0x08 - column values are unique.
COLTYPE	TT_INTEGER NOT NULL	Data type of column 1 TT_CHAR 2 TT_DECIMAL 3 TT_DECIMAL 4 TT_INTEGER 5 TT_SMALLINT 6 BINARY_FLOAT 7 BINARY_FLOAT 8 BINARY_DOUBLE 9 TT_DATE 10 TIME 11 TT_TIMESTAMP 12 TT_VARCHAR 13 DATE 14 TIMESTAMP 15 NUMBER 16 CHAR 17 VARCHAR2 18 NCHAR 19 NVARCHAR2 - 1 LONGVARCHAR - 2 BINARY - 3 VARBINARY - 4 LONGVARBINARY - 5 TT_BIGINT - 6 TT_TINYINT - 7 BIT - 8 WCHAR - 9 WVARCHAR - 10 WLONGVARCHAR

Column name	Type	Description
COLLEN	TT_INTEGER NOT NULL	Length of the column (maximum length for varying-length columns).
COLPRECISION	TT_INTEGER NOT NULL	The number of digits in a fixed-point number, or the number of digits in the mantissa of a floating point number
COLSCALE	TT_INTEGER NOT NULL	A non-negative number. A scale of 0 indicates an integer with no digits to the right of a decimal point. For a scale of *S*, the exact numeric value is the integer value of the significant digits multiplied by: 10 (exp -*S*).
PTNNUM	TT_SMALLINT NOT NULL	The table partition that contains the column.
PTNCOLOFF	TT_INTEGER NOT NULL	The offset of the column within the partition.
PTNNULLOFF	TT_INTEGER NOT NULL	The offset to the null byte within the partition.

Column name	Type	Description
REPKEYPOSITION	TT_SMALLINT NOT NULL	The ordinal position of this column in the replication key described by the REPKEYCOLS.
TS_EXCEPTION_ACTION	TT_CHAR(1) NOT NULL	The action to take upon detecting a conflict by a timestamp-based detector. The action is specified by the ON EXCEPTION clause in the *CheckConflicts* portion of a CREATE REPLICATION statement. They appear in this column as: \0' - action not defined 'N' - NO ACTION 'R' - ROLLBACK WORK (default)

TTREP.TTSTORES

The TTSTORES table maps a store's pair to a unique TT_STORE_ID. The TT_STORE_ID is a foreign key for all other replication schema tables that refer to a store in a replication scheme.

Table name TTSTORES

Columns

Column name	Type	Description
TT_STORE_ID	TT_BIGINT NOT NULL	Unique, system-generated identifier for a HOST_NAME/ TT_STORE_NAME pair.
HOST_NAME	TT_VARCHAR (200) NOT NULL NOT INLINE	Name of the participating host node.
TT_STORE_NAME	TT_VARCHAR (200) NOT NULL NOT INLINE	The name for this data store.
IS_LOCAL_STORE	BINARY(1) NOT NULL	1 if this TT_STORE_ID represents the local data store; 0 otherwise.
MAJOR_RELEASE	TT_INTEGER NOT NULL	The major release part of this data store's TimesTen release number. 0 indicates the current release.
MINOR_RELEASE	TT_INTEGER NOT NULL	The minor release part of this store's TimesTen release number.
REP_SCHEMA_VERSION	TT_INTEGER NOT NULL	The version of the replication schema in this data store.
REP_PORT_NUMBER	TT_INTEGER NOT NULL	The port number that replication uses to communicate with this data store. 0 if automatically assigned.

Column name	Type	Description
RRPOLICY	TT_CHAR (1)	Subscribers affected by return service failure policy. Legal values are: ‘S’ - Single subscriber ‘A’ - All subscribers ‘N’ No policy
RRTRIGGER	TT_INTEGER	Number of timeouts before the return service failure policy is triggered
RRRESUME_LATENCY	TT_INTEGER	Resume latency in milliseconds.
RRDURABLE	BINARY (1)	Durable commits on RETURN RECEIPT failure. Legal values are: 1 - True 0 - False
RET_LOCAL_ACTION	TT_CHAR (1)	Default commit behavior for RETURN TWOSAFE transactions: ‘C’ - COMMIT ‘N’ - NO ACTION
RET_WAIT_TIME	TT_INTEGER	The defaulted timeout value for RETURN TWOSAFE transactions.
RET_WHEN_STOPPED	BINARY (1)	If either the replication agent for the data store is stopped or if the data store is used as master and the replication agent for the data store is set to STOP, then if the value of the column is a non-zero value, return services for the data store are suspended.
COMPRESSION	TT_CHAR (1)	If Y, indicates compression of all data from the data store.

Column name	Type	Description
MASTER	TT_CHAR (1)	Active/Standby or Subscriber store: Values are: 'Y' - active or standby store 'N' - subscriber store NULL - all other cases.
ROLE	TT_CHAR (1)	Role is one of: 'A' - active 'S' - standby NULL - all other cases.
TS	TT_BIGINT	The timestamp at which the specified role change was made.
CONFLICT_REPORT_STOP	TT_INTEGER	Reserved for future use.
CONFLICT_REPORT_RESTART	TT_INTEGER	Reserved for future use.
CONFLICT_REPORT_FLUSH_METHOD	TT_INTEGER	Reserved for future use.

7

Access Control Privileges

This chapter describes which privileges are required to perform TimesTen operations when Access Control is enabled on your TimesTen instance.

For details about any operation, see the reference for each particular operation.

Privilege descriptions

In addition to the instance administrator, access rights for data store operations can be assigned to any instance user based on a set of specific privileges. Some characteristics of Access Control privileges are:

- To give a user access to controlled operations, you must use the **GRANT** statement. To remove a privilege from a user, you must use the **REVOKE** statement.
- Assigned privileges provide access to all data stores and all objects in those data stores in a particular TimesTen instance. TimesTen does not support a finer granularity of authentication, such as table level privileges.
- At install time, the CONNECT and CREATE DATASTORE privileges are granted to PUBLIC. All users have these privileges unless they are explicitly revoked.
- A user always has WRITE and SELECT privileges to any table they own, even if the privileges have not been granted explicitly to the user.
- Revoking SELECT and WRITE privileges for a user does not remove those privileges for objects that the user owns.
- Privileges are determined at connect time and remain in effect until disconnect.
- Privileges are cumulative. Granting a lower level privilege to a user does not degrade higher privileges which have already been granted to the user. Granting a high level privilege to a user who does not have lower level privileges does not give the user the lower level privileges.

The privileges that TimesTen supports are:

Privilege	Description
Instance Administrator	The system user who installed TimesTen is the TimesTen administrator. Some operations can only be performed by the instance administrator. This privilege cannot be granted or revoked using the TimesTen SQL statements **GRANT** and **REVOKE**.
ADMIN	Privilege to perform administrative operations on a data store. Some data store operations, such as the administration of the replication feature, require administrative privilege.
CONNECT	Privilege to connect to any data store in the TimesTen instance. At the time that Access Control is enabled on a TimesTen instance, all users are granted the CONNECT privilege.
CREATE DATASTORE	Privilege to create a data store. The user who creates a data store must have CONNECT privilege before creating a data store and DDL privilege to create data store objects. At the time that Access Control is enabled on a TimesTen instance, all users are granted the CREATE DATASTORE privilege.
DDL	Privilege to make changes to the data store schema, such as CREATE and ALTER operations.
WRITE	Privilege to perform UPDATE, DELETE and INSERT operations on a data store. Owners of tables always have the WRITE privilege on the tables they own.
SELECT	Privilege to perform only SELECT operations on a table. Owners of tables always have the SELECT privilege on the tables they own.

Operations requiring instance Administrator privilege

The following operations can only be performed by the Instance Administrator.

SQL operations

ALTER USER

CREATE USER

GRANT

REVOKE

Utilities

ttDaemonAdmin

ttmodinstall

Operations requiring ADMIN privilege

The following operations require ADMIN privilege.

Attributes

All first connection attributes require ADMIN privilege. These are:

AutoCreate
CkptFrequency
CkptLogVolume
CkptRate
Connections
ForceConnect
LogAutoTruncate
LogBuffSize
LogFileSize
LogFlushMethod
Logging
LogPurge
MemoryLock
Overwrite
PermSize
Preallocate
RecoveryThreads
TempSize

The following data store attributes require the ADMIN privilege:

Authenticate
GroupRestrict

Built-in Procedures

ttBackupStatus
ttBookmark
ttCachePolicyGet
ttCachePolicySet
ttCacheSqlGet
ttCacheStart
ttCacheStop
ttCacheUidPwdSet
ttCkpt
ttCkptBlocking
ttCkptConfig
ttCompact
ttCompactTS
ttDurableCommit
ttLockLevel
ttLockWait
ttOptSetMaxCmdFreeListCnt
ttOptSetMaxPriCmdFreeListCnt
ttRamPolicyGet
ttRamPolicySet
ttRepDeactivate
ttRepPolicyGet
ttRepPolicySet
ttRepStart
ttRepStateGet
ttRepStateSave
ttRepStateSet
ttRepStop
ttRepSubscriberStateSet
ttRepSubscriberWait

ttRepSyncGet
ttRepSyncSet
ttRepTransmitGet
ttRepTransmitSet
ttUserPrivileges
ttUsers
ttWarnOnLowMemory
ttXlaBookmarkCreate
ttXlaBookmarkDelete
ttXlaSubscribe
ttXlaUnsubscribe

SQL operations

ALTER ACTIVE STANDBY PAIR
ALTER REPLICATION
CREATE ACTIVE STANDBY PAIR
CREATE REPLICATION
DROP REPLICATION
LOAD CACHE GROUP
REFRESH CACHE GROUP
TRUNCATE TABLE
UNLOAD CACHE GROUP

Utilities

ttAdmin
ttBackup
ttCheck
ttDestroy
ttMigrate
ttRepAdmin
ttRestore

Use of some options of the following utilities also require this privilege:

ttIsql
ttTraceMon
ttXactAdmin
ttXactLog

Utility C API

ttXactIdRollback
ttBackup
ttDestroyDataStore
ttDestroyDataStoreForce
ttRamGrace
ttRamLoad
ttRamPolicy
ttRamUnload
ttRepDuplicateEx
ttRestore

XLA Functions

ttXlaDeleteBookmark

Operations requiring CONNECT privilege

Any user or application that wishes to connect to a TimesTen data store must have CONNECT privilege. By default, all users have CONNECT privilege, unless they have been explicitly revoked.

Operations requiring CREATE DATASTORE privilege

Any user or application that wishes to create a TimesTen data store must have CREATE DATASTORE privilege. By default, all users have CREATE DATASTORE privilege, unless they have been explicitly revoked.

Operations requiring DDL privilege

The following operations require DDL privilege.

Built-in Procedures

ttAgingLRUConfig
ttCacheAWTThresholdSet
ttCacheSqlGet
ttOptClearStats
ttOptEstimateStats
ttOptGetMaxCmdFreeListCnt
ttOptSetColIntvlStats
ttOptSetColStats
ttOptSetTblStats
ttOptUpdateStats
ttSetUserColumnID
ttSetUserTableID

SQL operations

ALTER CACHE GROUP
ALTER SESSION
ALTER TABLE
CREATE CACHE GROUP
CREATE INDEX
CREATE MATERIALIZED VIEW
CREATE SEQUENCE
CREATE TABLE
CREATE VIEW
DROP CACHE GROUP
DROP INDEX
DROP SEQUENCE
DROP TABLE
DROP VIEW

Operations requiring WRITE privilege

The following operations require WRITE privilege.

Built-in Procedures

ttCachePropagateFlagSet

ttApplicationContext

SQL operations

ALTER TABLE...ON DELETE CASCADE clause

DELETE

DROP ACTIVE STANDBY PAIR

INSERT

INSERT SELECT

on target table of MERGE

UPDATE

XLA functions

ttXlaOpenTimesTen

ttXlaPersistOpen

Operations requiring SELECT privilege

The following operations require SELECT privilege.

Built-in Procedures

ttBlockInfo

ttLogBufPrint

ttLogHolds

ttOptGetColStats

ttOptGetFlag

ttOptGetOrder

ttOptSetFlag

ttOptShowJoinOrder

ttOptUseIndex

ttSize

SQL operations

SELECT

on source table of MERGE

Utilities

ttMigrate

ttTail

ttSize

Use of some options of the following utilities also require this privilege:

ttIsql

ttTraceMon

ttXactAdmin

ttXactLog

8

Reserved Words

The following table includes words reserved by TimesTen for use in SQL statements.

Syntax errors may occur if you use these words as identifiers in SQL statements. In order to use one of these words as an identifier (such as a table or column name), enclose the reserved word in quotes:

Reserved Words		
AGING	CROSS	GROUP
ALL	CURRENT_SCHEMA	HAVING
ANY	CURRENT_USER	INNER
AS	CURSOR	INT
BETWEEN	DATASTORE_OWNER	INTEGER
BIGINT	DATE	INTERSECT
BINARY	DEC	INTERVAL
BINARY_DOUBLE_INFINITY	DECIMAL	INTO
BINARY_DOUBLE_NAN	DEFAULT	IS
BINARY_FLOAT_INFINITY	DESTROY	JOIN
BINARY_FLOAT_NAN	DISTINCT	LEFT
CASE	DOUBLE	LIKE
CHAR	FIRST	LONG
CHARACTER	FLOAT	MINUS
COLUMN	FOR	NATIONAL
CONNECTION	FOREIGN	NCHAR
CONSTRAINT	FROM	NO

Reserved Words		
NULL	ROWS	UPDATE
NUMERIC	SELECT	USER
NVARCHAR	SELF	USING
ON	SESSION_USER	VARBINARY
ORA_SYSDATE	SET	VARCHAR
ORDER	SMALLINT	VARYING
PRIMARY	SOME	WHEN
PROPAGATE	SYSDATE	WHERE
PUBLIC	SYSTEM_USER	
READONLY	TIME	
REAL	TINYINT	
RETURN	TT_SYSDATE	
RIGHT	UNION	
ROWNUM	UNIQUE	

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W