Oracle® TimesTen In-Memory Database

SQL Reference

Release 22.1
F35402-03
July 2025

Oracle TimesTen In-Memory Database SQL Reference, Release 22.1

F35402-03

Contents

4 Functions

5 Search Conditions

6 SQL Statements

7 Privileges

8 Reserved Words

About This Content

This document covers TimesTen support for SQL.

Audience

This document is intended for all users of SQL in a TimesTen database. The document contains a complete description of the Structured Query Language (SQL) used to manage information in a TimesTen database.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New

This section summarizes the new features of Oracle TimesTen In-Memory Database Release 22.1 that are documented in this guide. It provides links to more information.

New features in Release 22.1.1.1.0

- There is support for global indexes. See:
  - CREATE INDEX
  - CREATE TABLE
  - ALTER TABLE
- The CREATE PROFILE and ALTER PROFILE statements support password complexity checker functions. These functions enable passwords to be verified. See CREATE PROFILE and ALTER PROFILE for details.
- There is support for static read-only cache groups in TimesTen Scaleout. See CREATE CACHE GROUP for more information.
- You can create a hybrid cache group. This is a dynamic read-only cache group where the root table does not exist in the Oracle database. See CREATE CACHE GROUP for details.
- The following optimizer hints are supported in TimesTen Classic and are specific to TimesTen Cache.
  - TT_DynamicPassThrough
  - TT_DynamicLoadMultiplePKs
  - TT_DynamicLoadRootTbl

  See Statement Level Optimizer Hints for details.
- There is support for the TT_CountAsInt optimizer hint. This hint controls the return data type for the COUNT function. See Statement Level Optimizer Hints for details.

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. For more information, refer to ODBC API reference documentation, which is available from Microsoft or a variety of third parties. The following site contains Microsoft's ODBC API reference documentation:

https://msdn.microsoft.com/en-us/library/ms714562(VS.85).aspx

If you are using TimesTen Cache, see Mappings Between Oracle Database and TimesTen Data Types in *Oracle TimesTen In-Memory Database Cache Guide*. This section compares valid data types for creating cache group columns and type conversions for passthrough queries.

The following subjects describe data types in TimesTen:

- Type Specifications
- ANSI SQL Data Types
- Types Supported for Backward Compatibility
- Character Data Types
- Numeric Data Types
- BINARY and VARBINARY Data Types
- Numeric Precedence
- LOB Data Types
- ROWID Data Type
- Datetime Data Types
- TimesTen Intervals
- Storage Requirements
- Data Type Comparison Rules
- Data Type Conversion
- Null Values
- INF and NAN
- Overflow and Truncation
- Underflow

Type Specifications

Table 1-1 shows the supported data types in TimesTen.

Table 1-1 Supported data types

Data type	Description
BINARY(*n*)	Fixed-length binary value of *n* bytes Supported values for *n* range from 1 to 8300. BINARY data is padded to the maximum column size with trailing zeroes. Alternatively, specify TT_BINARY(*n*). See BINARY and VARBINARY Data Types for details.
BINARY_DOUBLE	A 64-bit floating-point number BINARY_DOUBLE is a double-precision native floating point number that 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 See BINARY_DOUBLE for details.
BINARY_FLOAT	A 32-bit floating-point number BINARY_FLOAT is a single-precision native floating-point type that 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 See "BINARY_FLOAT" for details.
BLOB	A variable-length binary large object. The valid range is from 1 to 16,777,216 bytes. See BLOB for details.
CHAR[ACTER][(*n*[BYTE|CHAR])]	Fixed-length character string of length *n* bytes or *n* characters. BYTE indicates that the column has byte-length semantics. The valid values for *n* range from a minimum of one byte to a maximum of 8300 bytes. CHAR indicates that the column has character-length semantics. The minimum CHAR length is one character. The maximum CHAR length depends on how many characters fit in 8300 bytes. This is determined by the database character set 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. If you do not specify BYTE or CHAR, the default is BYTE. If you do not specify *n*, the default is 1. 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. See CHAR for details.

Table 1-1 (Cont.) Supported data types

Data type	Description
CLOB	A variable-length character large object containing single-byte or multibyte characters. The valid range is from 1 to 4,194,304 bytes of data depending on the database character set. Specifically, the maximum size of a CLOB is stated in bytes (4,194,304 bytes), but the CLOB data type stores characters. Depending on the database character set and the actual characters being stored, the CLOB data type can store between 1,048,576 characters and 4,194,304 characters. See CLOB for details.
DATE	Date and time information: century, year, month, day, hour, minute, and second Format is: YYYY-MM-DD HHMISS. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. There are no fractional seconds. See DATE for details.
INTERVAL [+/-] *IntervalQualifier*	Interval type 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 eight digits for all interval types. The single field type can be: year, month, day, hour, minute, or second. Currently, interval types can be specified only with a constant. **Note**: You cannot specify a column of an interval type. These are non-persistent types used in SQL expressions at runtime. In addition, for those comparisons where an interval data type is returned, the interval data type cannot be the final result of a complete expression. The EXTRACT function must be used to extract the desired component of this interval result. See TimesTen Intervals for details.
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 half the byte limits so the maximum size is 4150. 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. See NCHAR for details.
NCLOB	A national variable-length character large object containing Unicode characters. The valid range is from 1 to 2,097,152 characters. See NCLOB for details.

Table 1-1 (Cont.) Supported data types

Data type	Description
NUMBER[(*p* [,*s*])]	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, TimesTen assumes the maximum precision of 38 and flexible scale. NUMBER supports negative scale and scale greater than precision. NUMBER stores both zero and positive and negative fixed numbers with absolute values from 1.0 x 10^{-130} to (but not including) 1.0 x 10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0 x 10^{126}, then TimesTen returns an error. See NUMBER for details.
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 half the byte limits. The valid range is from 1 to 2,097,152 characters. You must specify *n*. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. See NVARCHAR2 for details.
ROWID	An 18-byte character string that represents the address of a table row or materialized view row in TimesTen Classic. Specify a literal ROWID value as a CHAR constant enclosed in single quotes. See ROWID Data Type for details.
TIME	A time of day between 00:00:00 (midnight) and 23:59:59 (11:59:59 pm), inclusive The format is: *HH:MI:SS*. Alternatively, specify TT_TIME. See TIME for details.
TIMESTAMP [(*fractional_seconds_precision*)]	Year, month, and day values of the date plus hour, minute, and second values of the time The *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 eight-byte integer containing the number of microseconds since January 1, 1753. Comparisons are very fast. TIMESTAMP has a larger range than TT_TIMESTAMP in that TIMESTAMP can store date and time data as far back as 4712 BC. TIMESTAMP also supports up to nine digits of fractional second precision whereas TT_TIMESTAMP supports six 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] See TIMESTAMP for details.

Table 1-1 (Cont.) Supported data types

Data type	Description
TT_BIGINT	A signed eight-byte integer in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. 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. See TT_BIGINT for details.
TT_DATE	Date information: century, year, month, and day The format is YYYY-MM-DD, where MM is expressed as an integer such as 2006-10-28. Valid dates are between 1753-01-01 (January 1, 1753) and 9999-12-31 (December 31, 9999). See TT_DATE for details.
TT_INTEGER	A signed integer in the range -2,147,483,648 to 2,147,483,647. 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. See TT_INTEGER for details.
TT_SMALLINT	A native signed 16-bit integer in the range -32,768 to 32,767. 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. See TT_SMALLINT for details.
TT_TIMESTAMP	A date and time between 1753-01-01 00:00:00 (midnight on January 1, 1753) 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 [.FFFFFFFFF]. TT_TIMESTAMP has a smaller storage size than TIMESTAMP and is faster than TIMESTAMP because TT_TIMESTAMP is an eight-byte integer containing the number of microseconds since January 1, 1753. Comparisons are very fast. TIMESTAMP has a larger range than TT_TIMESTAMP in that TIMESTAMP can store date and time data as far back as 4712 BC. TIMESTAMP also supports up to nine digits of fractional second precision whereas TT_TIMESTAMP supports six digits of fractional second precision. You can specify TT_TIMESTAMP(6). See TT_TIMESTAMP for details.

Table 1-1 (Cont.) Supported data types

Data type	Description
TT_TINYINT	Unsigned integer ranging from 0 to 255. 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 a TT_SMALLINT. See TT_TINYINT for details.
VARBINARY(*n*)	Variable-length binary value with length *n* bytes. The valid range is from 1 to 4,194,304 bytes. Alternatively, specify TT_VARBINARY(*n*). See BINARY and VARBINARY Data Types for details.
VARCHAR[2](*n*[BYTE|CHAR])	Variable-length character string with a length of *n* bytes or *n* characters. BYTE indicates that the column has byte-length semantics. CHAR indicates that the column has character-length semantics. For byte-length semantics, the valid range is from 1 to 4,194,304 bytes. For character-length semantics, the valid range is from 1 to 1,048,576 characters. If you do not specify BYTE or CHAR, the default is BYTE. You must specify a value for *n*. 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. See VARCHAR2 for details.

ANSI SQL Data Types

TimesTen supports ANSI SQL data types. These data types are converted to TimesTen data types with data stored as TimesTen data types. Table 1-2 shows how the ANSI SQL data types are mapped to TimesTen data types.

Table 1-2 Data type mapping: ANSI SQL to TImesTen

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.
DOUBLE [PRECISION]	NUMBER Floating-point number with a binary precision of 126. Alternatively, specify FLOAT(126).

Table 1-2 (Cont.) Data type mapping: ANSI SQL to TImesTen

ANSI SQL data type	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 implemented as type NUMBER.
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.
NATIONAL CHARACTER(*n*) or NATIONAL CHAR(*n*)	NCHAR(*n*)
NATIONAL CHARACTER VARYING(*n*) or NATIONAL CHAR VARYING(*n*) or NCHAR VARYING(*n*)	NVARCHAR2(*n*)
NUMERIC[(*p*[,*s*])] or DEC[IMAL][(*p*[,*s*])]	NUMBER(*p*,*s*) Specifies a fixed-point number with precision *p* and scale *s*. This can only be used for fixed-point numbers. If no scale is specified, *s* defaults to 0.
REAL	NUMBER Floating-point number with a binary precision of 63. Alternatively, specify FLOAT(63).
SMALLINT	NUMBER(38,0) TT_SMALLINT is a native signed integer data type. Using TT_SMALLINT is more compact and offers faster performance than the NUMBER type.

Types Supported for Backward Compatibility

TimesTen supports the data types shown in Table 1-3.

Table 1-3 Data types supported for backward compatibility

Data type	Description
TT_CHAR[(*n*[BYTE|CHAR])]	Fixed-length character string of length *n* bytes or characters Default is one byte. BYTE indicates that the column has byte-length semantics. Supported values for *n* range from a minimum of one byte to a maximum 8300 bytes. CHAR indicates that the column has character-length semantics. The minimum CHAR length is one character. The maximum CHAR length depends on how many characters fit in 8300 bytes. This is determined by the database character set 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. If you insert a zero-length (empty) string into a column, the SQL NULL value is inserted. TT_CHAR data is padded to the maximum column size with trailing blanks. Blank-padded comparison semantics are used.
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 half the byte limits so the maximum size is 4150. If you insert a zero-length (empty) string into a column, the SQL NULL value is inserted. TT_NCHAR data is padded to the maximum column size with U+0020 SPACE. 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 half the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify *n*. If you insert a zero-length (empty) string into a column, the SQL NULL value is inserted. 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. Supported values for *n* range from a minimum of 1 byte to a maximum 4194304 (2^{22}) bytes. CHAR indicates that the column has character-length semantics. If you insert a zero-length (empty) string into a column, the SQL NULL value is inserted. Blank-padded comparison semantics are used.

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 database character sets defined when the database is created. TimesTen supports both single and multibyte character sets.

The character types are as follows:

- CHAR
- NCHAR
- VARCHAR2
- NVARCHAR2

CHAR

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, 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 one byte to four bytes depending on the database character set. The BYTE and CHAR qualifiers override the NLS_LENGTH_SEMANTICS parameter setting. See "ALTER SESSION" for more information about NLS_LENGTH_SEMANTICS. Also see Setting Globalization Support Attributes in *Oracle TimesTen In-Memory Database Operations Guide*.

> **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.

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), nnme2 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.
```

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

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.
>
> The NCHAR data type is encoded as UTF-16.

The following example alters table typedemo to add 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

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 one byte. Use the CHAR qualifier to specify the maximum length in characters. For example, VARCHAR2(10 CHAR).

The size of a character ranges from one byte to four bytes depending on the database character set. The BYTE and CHAR qualifiers override the NLS_LENGTH_SEMANTICS parameter setting. See "ALTER SESSION" for more information on NLS_LENGTH_SEMANTICS. Also see Setting Globalization Support Attributes in *Oracle TimesTen In-Memory Database Operations Guide*.

The NULL value is stored as a single bit for each nullable field within the row. An INLINE VARCHAR2(*n*) whose value is NULL takes (null bit) + four bytes + *n* bytes of storage, or *n* more bytes of storage than a NOT INLINE VARCHAR2(*n*) whose value is NULL. This storage principal holds for all variable length data types: TT_VARCHAR, TT_NVARCHAR, VARCHAR2, NVARCHAR2, VARBINARY.

> **Note**
>
> - Do not use the VARCHAR data type. Use VARCHAR2. Even though both data types are currently synonymous, the VARCHAR data type may be redefined as a different data type with different semantics.
> - 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.

The following example alters table typedemo, adding columns name5 and name6. The name5 column is defined with type VARCHAR2. The name6 column is defined with TT_VARCHAR. The example illustrates the use of nonpadded comparison semantics with column name5 and blank-padded comparison semantics with column name6:

```
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

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.

> **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.
>
> The NVARCHAR2 data type is encoded as UTF-16.

The following example alters table typedemo to add column name7. Data type is NVARCHAR2.

```
Command> ALTER TABLE typedemo ADD COLUMN Nnme7 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, meaning not a number).

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

Following are the exact numeric types.

- NUMBER
- TT_BIGINT
- TT_INTEGER
- TT_SMALLINT
- TT_TINYINT

The approximate types are:

- BINARY_DOUBLE
- BINARY_FLOAT
- FLOAT and FLOAT(*n*)

NUMBER

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

Specify a fixed-point number as NUMBER(*p*,*s*), where the following holds:

- The argument *p* is the precision or the total number of significant decimal digits, where the most significant digit is the left-most nonzero digit and the least significant digit is the right-most known digit.
- The argument *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 up to and including the least significant digit.
  - Negative scale is the number of significant digits to the left of the decimal point up 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. For example, in the case of E-notation. 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 the 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. TimesTen rounds values after the fifth digit to the right of the decimal point.

If a value exceeds the precision, then TimesTen returns an error. If a value exceeds the scale, then TimesTen rounds the value.

NUMBER(*p*) 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, TimesTen uses the maximum precision and scale.

The following example alters table numerics by adding columns col6, col7, col8, and col9 defined with the NUMBER data type and specified with different precisions 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 *)
```

The next 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.
```

The next 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.
```

The next 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.
```

TT_BIGINT

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 eight 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.

This example alters table numerics and attempts to add col5 with a data type of BIGINT. TimesTen generates an error. A second ALTER TABLE successfully adds col5 with the data type 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 *)
```

TT_INTEGER

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 four 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).

The following example alters the table numerics and adds col3 with the data type INT. Describing the table shows that the data type is NUMBER(38). The column col3 is dropped. A second ALTER TABLE adds col2 with the data type INTEGER. Describing the table shows that the data type is NUMBER(38). The column col3 is dropped. Columns col3 and col4 are then added with the data types TT_INTEGER and TT_INT. Describing the table shows both data types as 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  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  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_SMALLINT

The TT_SMALLINT data type is a signed integer that ranges from -32,768 (-2^{15}) to 32,767 (2^{15}-1). It requires two 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).

The following example alters the table numerics and adds col2 with the data type SMALLINT. Describing the table shows that the data type is NUMBER(38). The column col2 is dropped. A second ALTER TABLE adds col2 with the data type 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  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_TINYINT

The TT_TINYINT data type is an unsigned integer that ranges from 0 to 255 (2^8 -1). It requires one 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.

The following example first attempts to create a table named numerics that defines a column named col1 with data type TINYINT. TimesTen returns an error. The example then redefines the column 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 *)
```

Floating-Point Numbers

Floating-point numbers can be with or without a decimal point. An exponent may be used to increase the range (for example, 1.2E-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.

Use one of the following data types for floating-point numbers:

- BINARY_DOUBLE
- BINARY_FLOAT
- FLOAT and FLOAT(*n*)

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

The following 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 *)
```

BINARY_FLOAT

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

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 value of *n* indicates the number of bits of precision that can be stored, 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, multiply 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 Data Types

The BINARY data type is a fixed-length binary value with a length of *n* bytes, where 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, where the value of *n* ranges from 1 to 4,194,304 (2^{22}) bytes.

The following example creates a table and defines two columns: col1 is defined with data type BINARY and col2 with data type VARBINARY. Then, binary data is inserted into each column. Note that the BINARY value is padded to the right with zeros.

> **Note**
>
> See the description for the *HexadecimalLiteral* in "Constants" for details on assigning hexadecimal literals as binary data in TimesTen.

```
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 *)

Command> INSERT INTO bvar (col1, col2) VALUES (0x4D7953514C, 0x39274D);
1 row inserted.

Command> SELECT * FROM bvar;
 < 4D7953514C0000000000, 39274D >
1 row found.
```

Numeric Precedence

The result type of an expression is determined by the operand with the highest type precedence. The numeric precedence order is as follows (highest to lowest):

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

For example, the sum of TT_INTEGER and BINARY_FLOAT values is type BINARY_FLOAT because BINARY_FLOAT has higher numeric precedence. Similarly, the product of NUMBER and BINARY_DOUBLE values is type BINARY_DOUBLE.

LOB Data Types

LOB data types are not supported in TimesTen Scaleout.

The large object (LOB) data types can store large and unstructured data such as text, image, video, and spatial data. LOBs include the BLOB, CLOB and NCLOB data types.

You can insert or update data in a column that is of a LOB data type. For update operations, you can set the LOB value to NULL, an empty value through EMPTY_CLOB or EMPTY_BLOB, or replace the entire LOB with new data. You can update a LOB value with another LOB value. If you delete a row containing a LOB column, you also delete the LOB value.

LOB data type semantics are similar to the following SQL semantics:

- BLOB data types use SQL VARBINARY semantics.
- CLOB data types use SQL VARCHAR2 semantics.
- NCLOB data types use SQL NVARCHAR2 semantics.

The following SQL statements, operators, and functions accept one or more of the LOB data types as arguments.

- SQL statements: CREATE TABLE, SELECT, INSERT, and UPDATE
- Operators: LIKE and IS [NOT] NULL
- Functions: ASCIISTR, CONCAT, INSTR, INSTRB, INSTR4, LENGTH, LENGTHB, LOWER, LPAD, NLSSORT, NVL, TRIM, LTRIM, RTRIM, SUBSTR, SUBSTRB, SUBSTR4, REPLACE, RPAD, SOUNDEX, TO_DATE, TO_NUMBER, TO_CHAR, and UPPER

> **Note**
>
> Support for LOB data types is detailed in documentation for the above statements, operators, and functions. Refer to SQL Statements, Search Conditions, and Functions, respectively.

Description

- LOB conversion SQL functions (TO_BLOB, TO_CLOB, and TO_LOB) convert to the desired LOB data type.
- LOB columns are always stored out of line, so you cannot use the INLINE attribute when declaring LOB columns.
- You can define multiple columns of the LOB data type within a single table.
- You cannot create a primary key on a LOB column. You cannot define an index on a LOB column.
- You cannot create a materialized view if the detail table contains a LOB column.
- In addition to SQL, you can use LOB specific APIs in PL/SQL, ODBC, JDBC, OCI, and PRO*C/C++ for creating and updating LOBs. See the appropriate TimesTen developer's guide for more information on these APIs.

The following sections describe each LOB data type in more detail:

- BLOB
- CLOB
- NCLOB

In addition, the following sections provide more details on LOBs in general:

- Difference Between NULL and Empty LOBs
- Initializing LOBs

BLOB

The Binary LOB (BLOB) data type stores unstructured binary large objects. The maximum size for BLOB data is 16 MB.

> **Note**
>
> For details on assigning hexadecimal literals as binary data in TimesTen, see the description for the *HexadecimalLiteral* in "Constants".

When you define a BLOB in a column, you do not define the maximum number of characters as you would with VARBINARY and other variable length data types. Instead, the definition for the column would be as follows:

```
Command> CREATE TABLE blob_content (id NUMBER PRIMARY KEY,
                       blob_column BLOB );
```

To manipulate a BLOB, the following functions are provided:

- There are two methods to initialize a BLOB, including the EMPTY_BLOB function to initialize an empty BLOB. For details on initializing a BLOB, see "Initializing LOBs". For details on how an empty LOB is different from a NULL LOB, see "Difference Between NULL and Empty LOBs".
- To convert a binary value to a BLOB, use the TO_LOB or TO_BLOB functions. See "TO_BLOB" and "TO_LOB" for more details.

CLOB

The Character LOB (CLOB) data type stores single-byte and multibyte character data. The maximum size for CLOB data is 4 MB. The maximum number of characters that can be stored in the CLOB depends on whether you are using a single or multibyte character set.

When you define a CLOB in a column, you do not define the maximum number of characters as you would with VARCHAR and other variable length data types. Instead, the definition for the column would be as follows:

```
Command> CREATE TABLE clob_content (id NUMBER PRIMARY KEY,
                       clob_column CLOB );
```

To manipulate a CLOB, the following functions are provided:

- There are two methods to initialize a CLOB, including the EMPTY_CLOB function to initialize an empty CLOB. For details on initializing a CLOB, see "Initializing LOBs". For details on how an empty LOB is different from a NULL LOB, see "Difference Between NULL and Empty LOBs" below.
- To convert a character string to a CLOB, use the TO_LOB or TO_CLOB functions. See "TO_CLOB" and "TO_LOB" for more details.

NCLOB

The National Character LOB (NCLOB) data type stores Unicode data. The maximum size for an NCLOB data is 4 MB.

When you define a NCLOB in a column, you do not define the maximum number of characters as you would with VARCHAR and other variable length data types. Instead, the definition for the column would be as follows:

```
Command> CREATE TABLE nclob_content (id NUMBER PRIMARY KEY,
                        nclob_column NCLOB );
```

The following functions support the NCLOB data type:

- There are two methods to initialize an NCLOB, including the EMPTY_CLOB function to initialize an empty NCLOB. For details on initializing a NCLOB, see "Initializing LOBs". For details on how an empty LOB is different from a NULL LOB, see "Difference Between NULL and Empty LOBs", immediately below.
- To convert a character string to an NCLOB, use the TO_LOB or TO_CLOB functions. See "TO_CLOB" and "TO_LOB" for more details.

Difference Between NULL and Empty LOBs

A NULL LOB has a different meaning than an empty LOB.

- A NULL LOB has the value of NULL, so NULL is returned if you request a NULL LOB.
- An empty LOB is initialized with either the EMPTY_CLOB or EMPTY_BLOB functions. These functions initialize the LOB to be a zero-length, non-NULL value. You can also use the EMPTY_CLOB or EMPTY_BLOB functions to initialize a LOB in a non-nullable column.

Initializing LOBs

You can initialize a LOB in one of two ways:

- You can insert an empty LOB into a BLOB, CLOB or NCLOB column by using the EMPTY_BLOB or EMPTY_CLOB functions. This is useful when you do not have any data, but want to create the LOB in preparation for data. It is also useful for initializing non-nullable LOB columns.
- Initialize the LOB by inserting data directly. There is no need to initialize a LOB using the EMPTY_BLOB or EMPTY_CLOB functions, you can simply insert the data directly.

The following demonstrates examples of each type of initialization:

You can initialize a LOB with the EMPTY_CLOB function, as shown with the following example:

```
Command> INSERT INTO clob_content (id, clob_column)
       VALUES (1, EMPTY_CLOB( ) );
1 row inserted.
```

You can initialize a LOB by inserting data directly, as shown with the following example:

```
Command> INSERT INTO clob_content(id, clob_column)
       VALUES (4, 'Demonstration of the LOB initialization.');
1 row inserted.
```

You can initialize or update an existing LOB value with the UPDATE statement, as shown with the following examples:

```
Command> UPDATE blob_content
       SET blob_column = 0x000AF4511
       WHERE id = 1;
1 row updated.

Command> SELECT * FROM blob_content;
 < 1, 0000AF4511 >
1 rows found.

Command> UPDATE clob_content
       SET clob_column = 'Demonstration of the CLOB data type '
       WHERE id = 1;
```

```
1 row updated.

Command> SELECT * FROM clob_content;
 < 1, Demonstration of the CLOB data type >
```

ROWID Data Type

The ROWID data type is not supported in TimesTen Scaleout. The address of a row in a table or materialized view is called a *rowid*. The rowid data type is ROWID. You can examine a rowid by querying the ROWID pseudocolumn. See ROWID Pseudocolumn for details on the ROWID pseudocolumn.

Specify literal ROWID values in SQL statements as constants enclosed in single quotes, as follows:

```
Command> SELECT ROWID, last_name FROM employees
       WHERE department_id = 20;

< BMUFVUAAACOAAAALhM, Hartstein >
< BMUFVUAAACOAAAAMhM, Fay >
2 rows found.

Command> SELECT ROWID, last_name FROM employees
       WHERE ROWID='BMUFVUAAACOAAAALhM';
< BMUFVUAAACOAAAALhM, Hartstein >
1 row found.
```

Use the ROWID data type as follows:

- As the data type for a table column or materialized view column
- In these types of expressions:
  - Literals
  - Comparisons: <, <=, >, >=, BETWEEN
  - CASE Expressions
  - CAST
  - COALESCE
  - COUNT
  - DECODE
  - GREATEST
  - IN
  - IS NULL
  - LEAST
  - MAX
  - MIN
  - NVL
  - TO_CHAR
  - TT_HASH
- In ORDER BY and GROUP BY clauses

- In INSERT...SELECT statements. Column col1 has been defined with the ROWID data type for these examples:

```
Command> DESCRIBE master;

Table MYUSER.MASTER:
  Columns:
   *ID                     ROWID NOT NULL
    NAME                   CHAR (30)

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

Command> INSERT INTO master(id, name) SELECT ROWID, last_name
        FROM employees;
107 rows inserted.
Command> SELECT * FROM master;
< BMUFVUAAACOAAAAGhG, King                       >
< BMUFVUAAACOAAAAHhG, Kochhar                    >
< BMUFVUAAACOAAAAIhG, De Haan                    >
...
107 rows found.
```

You can use the TO_CHAR function with the ROWID pseudocolumn as shown below:

```
Command> INSERT INTO master(id, name)
        SELECT TO_CHAR(ROWID), last_name
        FROM employees;
107 rows inserted.
Command> SELECT * FROM master;
< BMUFVUAAACOAAAAGhG, King                       >
< BMUFVUAAACOAAAAHhG, Kochhar                    >
...
107 rows found.
```

You can use the CAST function with the ROWID pseudocolumn as shown below:

```
Command> CREATE TABLE master (id CHAR(20) NOT NULL PRIMARY KEY,
                     name CHAR(30));
Command> INSERT INTO master(id, name)
        SELECT CAST(ROWID AS CHAR(20)), last_name
        FROM employees;
107 rows inserted.
```

Implicit type conversions are supported for assigning values and comparison operations between ROWID and CHAR or between ROWID and VARCHAR2 data.

When CHAR, VARCHAR2, and ROWID operands are combined in COALESCE, DECODE, NVL, or CASE expressions (see CASE Expressions), the result data type is ROWID. Expressions with CHAR and VARCHAR2 values are converted to ROWID values to evaluate the expression.

To use ROWID values with string functions such as CONCAT, the application must convert ROWID values explicitly to CHAR values using the SQL TO_CHAR function.

Datetime Data Types

The datetime data types are as follows:

- DATE
- TIME

- TIMESTAMP
- TT_DATE
- TT_TIMESTAMP

DATE

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 seven bytes of storage.

TimesTen does not support user-specified NLS_DATE_FORMAT settings. You can use the SQL TO_CHAR and TO_DATE functions to specify other formats.

TIME

The format of a TIME value is *HH:MI:SS* and ranges from 00:00:00 (midnight) to 23:59:59 (11:59:59 pm). The TIME data type requires eight bytes of storage.

TIMESTAMP

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 and supports more precision than TT_TIMESTAMP.

TimesTen does not support user-specified NLS_TIMESTAMP_FORMAT settings. The SQL TO_CHAR and TO_DATE functions can be used to specify other formats.

TT_DATE

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

TT_TIMESTAMP

The format of a TT_TIMESTAMP value is YYYY-MM-DD HH:MI:SS [.FFFFFFFFF]. 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 eight bytes of storage. TT_TIMESTAMP is faster than the TIMESTAMP data type and has a smaller storage size.

TimesTen Intervals

This section includes the following topics:

- Using Interval Data Types
- Using DATE and TIME Data Types
- Handling Time Zone Conversions
- Datetime and Interval Data Types in Arithmetic Operations

Using Interval Data Types

TimesTen supports interval types only in a constant specification or intermediate expression result. Interval types cannot be the final result. Columns cannot be defined with an interval type. See "Type Specifications".

You can specify a single-field literal that is an interval in an expression, but you cannot specify a complete expression that returns an interval data type. Instead, the EXTRACT function must be used to extract the desired component of the interval result.

TimesTen supports interval literals of the following form:

INTERVAL [+/-] *CharString IntervalQualifier*

Using DATE and TIME Data Types

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

To create a table named sample that contains a column dcol of type DATE and a column tcol of type TIME, use the following:

```
CREATE TABLE sample (tcol TIME, dcol DATE);
```

To insert DATE and TIME values into the sample table, use this:

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

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

```
SELECT * FROM sample WHERE dcol = DATE '1998-10-29'
AND tcol BETWEEN TIME '12:00:00' AND TIME '16:00:00';
```

To create a table named sample2 that contains a column tscol of type TIMESTAMP and then select all rows in the table that are between noon and 4:00 p.m. on October 29, 1998, use these 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 enables 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 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 Time Zone Conversions

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

Datetime and Interval Data Types in Arithmetic Operations

You can perform numeric operations on date, timestamp and interval data. TimesTen calculates the results based on the rules:

- You can add or subtract a numeric value to or from a DATE or TIMESTAMP value. TimesTen internally converts TIMESTAMP values to DATE values.
- You can add or subtract a numeric value to or from a TT_DATE or TT_TIMESTAMP value and the resulting value is TT_DATE or TT_TIMESTAMP respectively.
- Numeric values are treated as number of days. For example, SYSDATE + 1 is tomorrow. SYSDATE - 7 is one week ago.
- Subtracting two date columns results in the number of days between the two dates. The return type is numeric.
- You cannot add date values. You cannot multiple or divide date or timestamp values.

Table 1-4 is a matrix of datetime arithmetic operations.

Table 1-4 DateTime arithmetic operations

Blank	DATE	TT_DATE	TIMESTAMP	TT_TIMESTAMP	NUMERIC	INTERVAL
DATE	not applicable	not applicable	not applicable	not applicable	not applicable	not applicable
+ (plus)	unsupported	unsupported	unsupported	unsupported	DATE	DATE
- (minus)	NUMBER	NUMBER	INTERVAL	INTERVAL	DATE	DATE
*** (multiply)**	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
/ (divide)	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
TT_DATE	not applicable	not applicable	not applicable	not applicable	not applicable	not applicable
+ (plus)	unsupported	unsupported	unsupported	unsupported	TT_DATE	TT_DATE
- (minus)	NUMBER	TT_BIGINT	INTERVAL	INTERVAL	TT_DATE	TT_DATE
*** (multiply)**	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
/ (divide)	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
TIMESTAMP	not applicable	not applicable	not applicable	not applicable	not applicable	not applicable
+ (plus)	unsupported	unsupported	unsupported	unsupported	DATE	TIMESTAMP
- (minus)	INTERVAL	INTERVAL	INTERVAL	INTERVAL	DATE	TIMESTAMP
*** (multiply)**	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
/ (divide)	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
TT_TIMESTAMP	not applicable	not applicable	not applicable	not applicable	not applicable	not applicable
+ (plus)	unsupported	unsupported	unsupported	unsupported	TT_TIMESTAMP	TT_TIMESTAMP
- (minus)	INTERVAL	INTERVAL	INTERVAL	INTERVAL	TT_TIMESTAMP	TT_TIMESTAMP
*** (multiply)**	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported

Table 1-4 (Cont.) DateTime arithmetic operations

Blank	DATE	TT_DATE	TIMESTAMP	TT_TIMESTAMP	NUMERIC	INTERVAL
/ (divide)	unsupported	unsupported	unsupported	unsupported	unsupported	unsupported
NUMERIC	not applicable	not applicable	not applicable	not applicable	not applicable	not applicable
+ (plus)	DATE	TT_DATE	DATE	TT_TIMESTAMP	Not applicable	unsupported
- (minus)	unsupported	unsupported	unsupported	unsupported	Not applicable	unsupported
*** (multiply)**	unsupported	unsupported	unsupported	unsupported	Not applicable	INTERVAL
/ (divide)	unsupported	unsupported	unsupported	unsupported	Not applicable	unsupported
INTERVAL	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
+ (plus)	DATE	TT_DATE	TIMESTAMP	TT_TIMESTAMP	unsupported	INTERVAL
- (minus)	unsupported	unsupported	unsupported	unsupported	unsupported	INTERVAL
*** (multiply)**	unsupported	unsupported	unsupported	unsupported	INTERVAL	unsupported
/ (divide)	unsupported	unsupported	unsupported	unsupported	INTERVAL	unsupported

> **Note**
>
> An interval data type cannot be the final result of a complete expression. The EXTRACT function must be used to extract the desired component of this interval result.

```
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;
```

This example inserts TIMESTAMP values into two 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.
```

The following queries return errors. You cannot select an interval result:

```
SELECT timestamp1 - timestamp2 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 an INTERVAL DAY TO SECOND with an 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 Datetime and Interval Arithmetic Operations

Consider these restrictions when performing datetime and interval arithmetic:

- The results for addition and subtraction with DATE and TIMESTAMP types for INTERVAL YEAR and INTERVAL MONTH are not closed. For example, adding one year to the DATE or TIMESTAMP of '2004-02-29' results in a date arithmetic error (TimesTen error 2787) because February 29, 2005 does not exist (2005 is not a leap year). Adding INTERVAL '1' month to DATE '2005-01-30' also results in the same error because February never has 30 days.
- The results are closed for INTERVAL DAY.
- An interval data type cannot be the final result of a complete expression. The EXTRACT function must be used to extract the desired component of the interval result.

Storage Requirements

Variable-length columns whose declared column length is greater than 128 bytes are stored out of line. Variable-length columns whose declared column length is less than or equal to 128 bytes are stored inline. All LOB data types are stored out of line.

For character semantics, the number of bytes stored out of line depends on the character set. For example, for a character set with four bytes per character, variable-length columns whose declared column length is greater than 32 (128/4) are stored out of line.

Table 1-5 shows the storage requirements of the various data types.

Table 1-5 Data type storage requirements

Type	Storage required
BINARY(*n*)	*n* bytes
BINARY_DOUBLE	Eight bytes
BINARY_FLOAT	Four bytes
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.
DATE	Seven bytes
Interval	An interval type cannot be stored in TimesTen
NCHAR(*n*)	Bytes required is 2**n* where *n* is the number of characters
NUMBER	Five to 22 bytes
NVARCHAR2(*n*)	For NOT INLINE columns: 2*(length of value) + 24 bytes (minimum of 40 bytes). For INLINE columns: 2*(length of column) + 8 bytes.

Table 1-5 (Cont.) Data type storage requirements

Type	Storage required
ROWID	Twelve bytes
TIMESTAMP	Twelve bytes
TT_BIGINT	Eight bytes
TT_DATE	Four bytes
TT_INT[EGER]	Four bytes
TT_SMALLINT	Two bytes
TT_TIME	Eight bytes
TT_TIMESTAMP	Eight bytes
TT_TINYINT	One byte
VARBINARY(*n*)	For NOT INLINE columns: Length of value + 24 bytes (minimum of 40 bytes). For INLINE columns: Length of column + 8 bytes.
VARCHAR2(*n*[BYTE|CHAR])	For NOT INLINE columns: Length of value + 24 bytes (minimum of 40 bytes). NULL value is stored as (null bit) + 8 bytes, or 8.125 bytes. This storage principal holds for all variable length NOT INLINE data types: TT_VARCHAR, TT_NVARCHAR, VARCHAR2, NVARCHAR2, and VARBINARY. For INLINE columns: n + 8 bytes. NULL value is stored as (null bit) + *n* + 8 bytes. If character semantics, the length of the column (*n*) is based on length semantics and character set.
BLOB and CLOB	Length of value + 48 bytes (minimum of 56 bytes)
NCLOB	2 * (length of value) + 48 bytes (minimum of 56 bytes)

Data Type Comparison Rules

This 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 1:15 pm' is greater than '30-AUG-2006 10:10 am'.

Character Values

Character values are compared in the following ways:

- Binary and Linguistic Sorting
- Blank-padded and Non-Padded 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 Sort Rules Support Linguistic Conventions in *Oracle TimesTen In-Memory Database Operations Guide*.

The default is binary sorting.

Blank-padded and Non-Padded 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.

An example with blank-padded semantics:

```
'a   ' = 'a'
```

An example with nonpadded semantics:

```
'a   ' > 'a'
```

Data Type 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. Because algorithms for implicit conversion are subject to change across software releases and the behavior of explicit conversions is more predictable, TimesTen recommends explicit conversion.

Implicit Data Type Conversion

TimesTen converts a value from one data type to another when such a conversion is sensible.

Table 1-6 and Table 1-7 use a matrix to illustrate TimesTen implicit data type conversions. YES in the cell indicates the conversion is supported. NO in the cell indicates the conversion is not supported. The rules for implicit conversion follow the table.

Table 1-6 Implicit data type conversion

Blank	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	TT_DATE	TIMESTAMP	TT_TIMESTAMP
CHAR	n/a	YES	YES	YES	YES	YES	YES	YES
VARCHAR2	YES	n/a	YES	YES	YES	YES	YES	YES
NCHAR	YES	YES	n/a	YES	YES	YES	YES	YES
NVARCHAR2	YES	YES	YES	n/a	YES	YES	YES	YES
DATE	YES	YES	YES	YES	n/a	YES	YES	YES
TT_DATE	YES	YES	YES	YES	YES	n/a	YES	YES
TIMESTAMP	YES	YES	YES	YES	YES	YES	n/a	YES
TT_TIMESTAMP	YES	YES	YES	YES	YES	YES	YES	n/a
NUMERIC	YES	YES	YES	YES	NO	NO	NO	NO
BLOB	NO	NO	NO	NO	NO	NO	NO	NO
CLOB	YES	YES	YES	YES	NO	NO	NO	NO
NCLOB	YES	YES	YES	YES	NO	NO	NO	NO
BINARY/ VARBINARY	YES	YES	YES	YES	NO	NO	NO	NO
ROWID	YES	YES	YES	YES	NO	NO	NO	NO

Table 1-7 Implicit data type conversion (continuation of preceding table)

Blank	NUMERIC	BLOB	CLOB	NCLOB	BINARY/ VARBINARY	ROWID
CHAR	YES	YES	YES	YES	YES	YES
VARCHAR2	YES	YES	YES	YES	YES	YES
NCHAR	YES	YES	YES	YES	YES	YES
NVARCHAR2	YES	YES	YES	YES	YES	YES
DATE	NO	NO	NO	NO	NO	NO
TT_DATE	NO	NO	NO	NO	NO	NO
TIMESTAMP	NO	NO	NO	NO	NO	NO
TT_TIMESTAMP	NO	NO	NO	NO	NO	NO
NUMERIC	n/a	NO	NO	NO	NO	NO
BLOB	NO	n/a	NO	NO	YES	NO

Table 1-7 (Cont.) Implicit data type conversion (continuation of preceding table)

Blank	NUMERIC	BLOB	CLOB	NCLOB	BINARY/ VARBINARY	ROWID
CLOB	NO	NO	n/a	YES	NO	NO
NCLOB	NO	NO	YES	n/a	NO	NO
BINARY/ VARBINARY	NO	YES	YES	YES	n/a	NO
ROWID	NO	NO	NO	NO	NO	n/a

The following rules apply:

- During arithmetic operations on and comparisons between character and non-character data types, TimesTen converts from any character data type to a numeric or datetime data type as appropriate. In arithmetic operations between CHAR/VARCHAR2 and NCHAR/NVARCHAR2, TimesTen converts to a NUMBER.
- During arithmetic operations, floating point values INF and NAN are not supported when converting character values to numeric values.
- During concatenation operations, TimesTen converts non-character data types to CHAR, NCHAR, VARCHAR2, or NVARCHAR2 depending on the other operand.
- When comparing a character value with a numeric value, TimesTen converts the character data to a numeric value.
- When comparing a character value with a datetime value, TimesTen converts the character data to a datetime value.
- During conversion from a timestamp value to a DATE value, the fractional seconds portion of the timestamp value is truncated.
- Conversions from BINARY_FLOAT to BINARY_DOUBLE are exact.
- Conversions from BINARY_DOUBLE to BINARY_FLOAT are inexact if the BINARY_DOUBLE value uses more bits of precision that supported by the BINARY_FLOAT.
- Conversions between either character values or 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 character values and the exact numeric values use decimal precision whereas the floating-point numbers use binary precision.
- When manipulating numeric values, TimesTen usually adjusts precision and scale to allow for maximum capacity. In such cases, the numeric data type resulting from such operations can differ from the numeric data type found in the underlying tables.
- When making assignments, TimesTen converts the value on the right side of the equal sign (=) to the data type of the target of the assignment on the left side.
- When you use a SQL function or operator with an argument of a data type other than the one it accepts, TimesTen converts the argument to the accepted data type as long as TimesTen supports the implicit conversion.
- During INSERT, INSERT... SELECT, and UPDATE operations, TimesTen converts the value to the data type of the affected column.
- Implicit and explicit CHAR/VARCHAR2 <-> NCHAR/NVARCHAR2 conversions are supported. An example of implicit 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

The value NULL indicates the absence of a value. It is a placeholder for a value that is missing. Use a NULL when the actual value is not known or when a value would not be meaningful. Do not use NULL to represent a numeric value of zero, because they are not equivalent. Any parameter in an expression can contain NULL regardless of its data type. In addition, any column in a table can contain NULL, regardless of its data type, unless you specify NOT NULL or PRIMARY KEY integrity constraints for the column when you create the table.

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

- By default, NULL is sorted as the highest value in a sequence of values. However, you can modify the sort order value for NULL with NULLS FIRST or NULLS LAST in the ORDER BY clause.
- Two NULL values are not equal to each other except in a GROUP BY or SELECT DISTINCT operation.
- An arithmetic expression containing a NULL evaluates to NULL. In fact, all operators (except concatenation) return NULL when given a NULL operand. For example, (5-col), where col is NULL, evaluates to NULL.
- To test for NULL, use the comparison conditions IS NULL or IS NOT NULL. Because NULL represents a lack of data, a NULL cannot be equal or unequal to any value or to another NULL. Thus, the statement select * from employees where mgr_id = NULL evaluates to 0, since you cannot use this comparison to NULL. However, the statement select * from employees where mgr_id is NULL provides the CEO of the company, since that is the only employee without a manager. For details, see "IS NULL Predicate".
- You can use the NULL value itself directly as an operand of an operator or predicate. For example, the (1 = NULL) comparison is supported. This is the same as if you cast NULL to the appropriate data type, as follows: (1 = CAST(NULL AS INT)). Both methods are supported and return the same results.

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

- Joining tables if the join is on a column containing NULL.
- Executing aggregate functions.

In several SQL predicates, you can explicitly test for NULL. APIs supported by TimesTen offer ways to handle null values. For example, in an ODBC application, use the functions SQLBindCol, SQLBindParameter, SQLGetData, and SQLParamData 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

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

- 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, Ii2 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 different from NULL, which is not allowed in 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, meaning not a number).

Consider the following 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.

The following rules apply to comparisons with Inf and NaN:

- 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 lower than NaN and NULL and higher than any other value.
- 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-zero conditions.
- Inf, -Inf, and NaN values are not ignored in aggregate functions. NULL values are. If you want to exclude Inf and NaN from aggregates, or from any SELECT result, 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 database 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 does not insert the specified value.

TimesTen may truncate approximate values during computations, when values are inserted into the database, or when database values are updated. TimesTen returns an error only upon INSERT 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 allowed in its type. See "Expressions" for more information.
- When aggregate functions are used, overflow can occur when the sum of several numbers exceeds the maximum allowable value of the result type.
- 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, as follows.

- For 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.
- For numeric data, a warning occurs when any trailing nonzero 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.

2

Names, Namespace and Parameters

This chapter presents general rules for names and parameters used in TimesTen SQL statements. It includes the following topics:

Basic Names

Basic names, or simple names, identify such objects as 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 (_).

  If a column name contains the # special character, enclose the column name in quotation marks. If an object name contains the $ special character, enclose the object name in quotation marks.
- TimesTen changes lowercase letters (a to z) to the corresponding uppercase letters (A to Z). Thus names are not case-sensitive.
- If you enclose a name in quotation marks, you can use any combination of characters even if they are not in the set of supported characters. When the name is enclosed in quotes, the first character in the name can be any character, including one or more spaces.

  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 database 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". When creating owner names, follow the same rules as those for creating basic names. See "Basic Names" for details.

Compound Identifiers

Basic names and user names are simple names. In some cases, simple names are combined and 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 for 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*
- [*Owner*.]*IndexName*
- *Table identifier*: [*Owner*.]*TableName*
- *Row identifier*: [[*Owner*.]*TableName*.]*rowid*

Namespace

In SQL syntax, object names that share the same namespace must each be unique. This is so that when a name is referenced in any SQL syntax, the exact object can be found.

If the object name provided is not qualified with the name (namespace) of the user that owns it, then the search order for the object is as follows:

1. Search for any match from all object names within the current user namespace. If there is a match, the object name is resolved.
2. If no match is found in the user namespace, search for any match from the PUBLIC namespace, which contains objects such as public synonyms. Public synonyms are pre-defined for SYS and TTREP objects. If there is a match, the object name is resolved. Otherwise, the object does not exist.

Any tables, views, materialized views, sequences, private synonyms, PL/SQL packages, functions, procedures, and cache groups owned by the same user share one namespace and so the names for each of these objects must be unique within that namespace. Indexes are created in their own namespace.

For example, because tables and views are in the same namespace, a table and a view owned by the same user cannot have the same name. However, tables and indexes are in different namespaces, so a table and an index owned by the same user can have the same name.

Tables that are owned by separate users can have the same name, since they exist in separate user namespaces.

Dynamic Parameters

Dynamic parameters pass information between an application program and TimesTen. TimesTen uses dynamic parameters as placeholders in SQL commands and at runtime replaces the parameters 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:

- Uppercase letters: A to Z
- Lowercase letters: a to z
- Digits: 0 to 9
- Special characters: # $ @ _

> **Note**
>
> Instead of using a *:DynamicParameter* sequence, the application can use a ? for each dynamic parameter.

Enhanced ":" style parameter markers have this form:

```
:parameter [INDICATOR] :indicator
```

The :*indicator* is considered to be a component of the :*parameter*. It is not counted as a distinct parameter. Do not specify '?' for this style of parameter marker.

Duplicate Parameter Names

Consider this SQL statement:

```
SELECT * FROM t1 WHERE c1=:a AND c2=:a AND c3=:b AND c4=:a;
```

Traditionally in TimesTen, multiple instances of the same parameter name in a SQL statement are considered to be multiple occurrences of the *same* parameter. When assigning parameter numbers to parameters, TimesTen assigns parameter numbers only to the first occurrence of each parameter name. The second and subsequent occurrences of a given name do not get their own parameter numbers. In this case, a TimesTen application binds a value for every unique parameter in a SQL statement. It cannot bind different values for different occurrences of the same parameter name nor can it leave any parameters or parameter occurrences unbound.

In Oracle Database, multiple instances of the same parameter name in a SQL statement are considered to be different parameters. When assigning parameter numbers, Oracle Database assigns a number to each parameter occurrence without regard to name duplication. An Oracle database application, at a minimum, binds a value for the first occurrence of each parameter name. For the subsequent occurrences of a given parameter, the application can either leave the parameter occurrence unbound or it can bind a different value for the occurrence.

The following table shows a query with the parameter numbers that TimesTen and Oracle Database assign to each parameter.

Query	TimesTen parameter number	Oracle Database parameter number
SELECT *	n/a	n/a
FROM t1	n/a	n/a
WHERE c1=:a	1	1
AND c2=:a	1	2
AND c3=:b	2	3
AND c4=:a;	1	4

The total number of parameter numbers for TimesTen in this example is 2. The total number of parameters for Oracle Database in this example is 4. The parameter bindings provided by an application produce different results for the traditional TimesTen behavior and the Oracle Database behavior.

You can use the DuplicateBindMode general connection attribute to determine whether applications use traditional TimesTen parameter binding for duplicate occurrences of a parameter in a SQL statement or Oracle-style parameter binding. Oracle-style parameter binding is the default.

Inferring Data Type from Parameters

Consider this statement:

```
SELECT :a FROM dual;
```

TimesTen cannot infer the data type of parameter a from the query. TimesTen returns this error:

```
2778: Cannot infer type of parameter from its use
The command failed.
```

Use the CAST function to declare the data type for parameters:

```
SELECT CAST (:a AS NUMBER) FROM dual;
```

3

Expressions

Expressions are used for the following purposes:

- The select list of the INSERT...SELECT statement
- A condition of the WHERE clause and the 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

The following sections describe expressions in TimesTen:

- Expression Specification
- Subqueries
- Constants
- Format Models
- CASE Expressions
- ROWID Pseudocolumn
- ROWNUM Pseudocolumn
- Pseudocolumns in TimesTen Scaleout

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 Functions. A *primary* is a signed or unsigned value derived from one of the items listed in the SQL syntax.

SQL syntax

```
{ColumnName | ROWID | {? | :DynamicParameter} |
 Function | Constant | (Expression)}
```

Or:

```
[[+ |-] {ColumnName | SYSDATE | TT_SYSDATE|GETDATE() |
{? | :DynamicParameter} | Function |
Constant | {~ | + | -} Expression}]
 [...]
```

Or:

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

Or:

Expression1 | | *Expression2*

Or:

Expression

Component	Description
+, –	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 See "Names, Namespace and Parameters" for more information.
ROWID	Unique ID for each row stored in a table The rowid value can be retrieved through the ROWID pseudocolumn.
? *:DynamicParameter*	A placeholder for a dynamic parameter The value of the dynamic parameter is supplied at runtime.
Function	A computed value See " Functions" for more information.
Constant	A specific value See "Constants" for details.
(*Expression*)	Any expression enclosed in parentheses
Expression1 *Expression2*	The specified expressions *Expression1* and *Expression2,* when used with the bitwise operators, can be of integer or binary types. The data types of the expressions must be compatible. See "Data Types" for more information.
*	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. Sets a bit 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. Sets a bit 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.
||	Concatenate operator Concatenates *Expression1* and *Expression2*, where both expressions are character strings. Forms a new string value that contains the values of both expressions. See "CONCAT" for more information.

Description

- Arithmetic operators can be used between numeric values. See "Numeric Data Types" for more information.
- Arithmetic operators can also be used between datetime values and interval types. The result of a datetime expression is either a datetime data type or an interval data type.
- Arithmetic operators cannot be applied to string values.
- Elements in an expression are evaluated in the following order:
  - 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. An example follows.

  ```
  10 * 2 – 1 = 19 but 10 * (2 – 1) = 10
  ```
- Type conversion, truncation, underflow, or overflow can occur when some expressions are evaluated. See "Data Types" for more information.
- 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 Search Conditions for more information on evaluation of comparison operators and predicates containing NULL values. TimesTen permits inserting 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.

Examples

This example shows a dynamic parameter in the WHERE clause of any SELECT statement:

```
SELECT * FROM purchasing.orders
  WHERE partnumber = ? AND ordernumber > ?
  ORDER BY ordernumber;
```

This example shows a dynamic parameter in the WHERE and SET clauses of an UPDATE statement:

```
UPDATE purchasing.parts
  SET salesprice = :dynamicparameter1
  WHERE partnumber = :dynamicparameter2;
```

This example shows a dynamic parameter in the WHERE clause of a DELETE statement:

```
DELETE FROM purchasing.orderitems
  WHERE itemnumber BETWEEN ? AND ?;
```

This example shows a dynamic parameter 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.

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

SQL syntax

[NOT] EXISTS | [NOT] IN (*Subquery*)
Expression {= | <> | > | >= | < | <= } [ANY | ALL] (*Subquery*)
Expression [NOT] IN (*ValueList* | *Subquery*)

Where *ValueList* is a list of constant expressions. Each constant expression specifies a constant value or an expression that evaluates to a constant value (such as a number, character string, or date). This includes support for bound values (? or :*DynamicParameter*), NULL, and calls to functions that return constant values.

Description

Table subquery:

- 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 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.
- 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.

A scalar subquery returns a single value. A nonverifiable scalar subquery 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.

Examples

Examples of supported subqueries for 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 = 'unshipped');

SELECT customers.name FROM customers
  WHERE customers.id = ANY
    (SELECT orders.custid FROM orders
  WHERE orders.status = 'unshipped');

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

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);
```

Constants

A constant is a literal value.

SQL syntax

```
{IntegerValue | FloatValue |FloatingPointLiteral|
    FixedPointValue | 'CharacterString'|
    'NationalCharacterString' | HexadecimalLiteral |
    '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
}
```

Constant	Description
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 These 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.

Constant	Description

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 \u*xxxx*, where *xxxx* is the four hexadecimal-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.

Constant	Description

HexadecimalLiteral	Hexadecimal literals
	Hexadecimal literals containing digits 0 - 9 and A - F (or a - f) are compatible with the BINARY, VARBINARY, CHAR, VARCHAR2 and BLOB data types. A *HexadecimalLiteral* constant should be prefixed with the characters "0x." For example:
	`0xFFFAB0880088343330FFAA7`
	Or:
	`0x000A001231`
	Hexadecimal digits provided with an odd length are pre-fixed with a zero to make it even. For example, the value 0x123 is converted to 0x0123.
	If you provide a character literal, the binary values of the characters are used. For example, the following demonstrates what is stored when inserting a hexadecimal literal and a character literal in a VARBINARY column colbin in table tabvb:
	`Command> INSERT INTO tabvb VALUES (0x1234);`
	`1 row inserted.`
	`Command> INSERT INTO tabvb VALUES ('1234');`
	`1 row inserted.`
	`Command> SELECT colbin FROM tabvb;`
	`< 1234 >`
	`< 31323334 >`
	`2 rows found.`
	However, Oracle Database differs in that it only accepts character literals, such as '1234', and translates the character literal as a binary literal of 0x1234. As a result, insert into tabvb values ('1234'); behaves differently between Oracle Database and TimesTen. Oracle Database does not accept 0x1234 as a hexadecimal literal.

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 four-digit value. The MM and DD fields must have two-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'.

Constant	Description
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'}. See ODBC documentation for details.
TimeString	A string of the format HH:MI: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'}

Constant	Description
TimestampString	A string of the format YYYY-MM-DD HH:MI: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 four-digit value. All other fields except for the fractional part must be two-digit values. The fractional field can consist of zero to nine digits. For TT_TIMESTAMP data types, a string of format YYYY-MM-DD HH:MI: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 zero to six 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.

TimestampLiteral	Format: TIMESTAMP *TimestampString*
	For example:
	`TIMESTAMP '2007-01-27 11:00:00.000000'`
	For TIMESTAMP data types, the fraction field supports from zero to nine digits of fractional seconds. For TT_TIMESTAMP data types, the fraction field supports from zero to six digits of fractional seconds.
	The TIMESTAMP keyword is case-insensitive.
	Use literal syntax to enforce DATE/TIME/TIMESTAMP comparisons for CHAR and VARCHAR2 data types.
	TimesTen also supports ODBC timestamp literal syntax. For example:
	`{ts '9999-12-31 12:00:00'}`

IntervalLiteral	Format: INTERVAL [+\-] *CharacterString IntervalQualifier* For example: `INTERVAL '8' DAY`
BINARY_FLOAT_INFINITY | BINARY_DOUBLE_INFINITY	Positive 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	Negative 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.

Constant	Description
BINARY_FLOAT_NAN | BINARY_DOUBLE_NAN	Non-numbers 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.

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.

This section covers the following format models:

- Number Format Models
- Datetime Format Models
- Format Model for ROUND and TRUNC Date Functions
- Format Model for TO_CHAR of TimesTen Datetime Data Types

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.

The default american_america NLS language and territory setting is used.

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.

Table 3-1 ***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.

Table 3-1 (Cont.) *Number format elements*

Element	Example	Description
0	0999 9990	Returns leading zeros. Returns trailing zeros.
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.
D	99D99	Returns the decimal character in the specified position. 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 the group separator in the specified position. 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 the local currency symbol in the specified position.
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 (+).
S	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.

Table 3-1 (Cont.) *Number format elements*

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 the euro or other dual currency symbol in the specified position.
V	999V99	Returns a value multiplied by 10^n (and if necessary, rounds it up), where *n* is the number of 9s 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, TO_DATE, and TO_TIMESTAMP 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 language and territory setting are used.

Datetime Format Elements

A datetime format model is composed of one or more datetime format elements. Table 3-2 shows the datatime format elements. In addition:

- For input format models, the format elements cannot appear more than once, and format elements that represent similar information cannot be combined. For example, you cannot use the SYYYY and the BC format elements in the same format string.

- For DATE format elements, capitalization in a spelled-out word, abbreviation or Roman numeral, follows the capitalization in the corresponding format element. For example, the date format model DAY produces capitalized words, like MONDAY, whereas Day produces Monday, and day produces monday.

Table 3-2 Datetime format elements

Element	Description
-/,.;:"text"	Punctuation and quoted text are 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.
CC SCC	Valid in TO_CHAR. Century: • If the last two digits of a 4-digit year are between 01 and 99 (inclusive), the century is one greater than the first two digits of the year. • If the last two digits of a 4-digit year are 00, the century is the same as the first two digits of that year. For example, 2002 returns 21 and 2000 returns 20.
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.
DL	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	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.
FF[1-9]	Valid in TO_TIMESTAMP. Fractional seconds. No radix character is printed. Use the X format element to add the radix character. Use the 1 to 9 numbers after FF to specify the number of digits in the fractional second part of the datatime value returned. If you do not specify a digit, the value is determined by the precision specified for the datetime data type, or by the data type's default precision.
FM	Returns a value with no leading or no trailing blanks.
FX	Returns exact matching between the character data and the format model.
HH HH12	Hour of day (1-12).

Table 3-2 (Cont.) Datetime format elements

Element	Description
HH24	Hour of day (0-23).
IW	Valid in TO_CHAR. Calendar week of year (1-52 or 1-53), as defined by the ISO 8601 standard. • A calendar week starts on Monday. • The first calendar week of the year includes January 4. • The first calendar week of the year may include December 29, 30, and 31. • The last calendar week of the year may include January 1, 2, and 3.
IYYY	Valid in TO_CHAR. 4-digit year of the year containing the calendar week, as defined by the ISO 8601 standard.
IYY IY I	Valid in TO_CHAR. Last 3, 2, or 1 digit(s) of the year containing the calendar week, as defined by the ISO 8601 standard.
J	Julian day. This is the number of days since January 1, 4712 BC. Numbers specified with J must be integers.
MI	Minute (0-59).
MM	Month (01-12, where January = 01).
MON	Abbreviated name of month.
MONTH	Name of month padded with blanks to display width of the widest name of month.
PM P.M.	Meridian indicator with or without periods.
Q	Valid in TO_CHAR. Quarter of the year (1, 2, 3, 4). January = 1.
RM	Roman numeral month (I-XII. January = I).
RR	For 20th century dates in the 21st century using only two digits.
RRRR	Rounded 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).
TS	Returns a value in the short time format. **Restriction**: Specify this format only with the DL or DS element, separated by white space.
WW	Valid in TO_CHAR. Week of year (1-53) where week one starts on the first day of the year and continues to the seventh day of the year.
W	Valid in TO_CHAR. Week of month (1-5) where week one starts on the first day of the month and ends on the seventh.

Table 3-2 (Cont.) Datetime format elements

Element	Description
X	Local radix character For example: 'HH:MI:SSXFF'
Y,YYY	Year with comma in this position
YEAR SYEAR	Valid in TO_CHAR. Year, spelled out. S prefixes BC dates with a minus sign (-).
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 ROUND and TRUNC Date Functions

The table lists the format models you can use with the ROUND and TRUNC date functions and the units to which they round and truncate dates. The default model DD returns the date rounded or truncated to the day with a time of midnight.

Format model	Rounding or truncating unit
CC SCC	Century If the last two digits of a four-digit year are between 01 and 99 (inclusive), then the century is one greater than the first two digits of that year. If the last two digits of a four-digit year are 00, then the century is the same as the first two digits of that year. For example, 2002 returns 21; 2000 returns 20.
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IYY IY I	ISO year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Name of month (rounds up on the sixteenth day)

Format model	Rounding or truncating unit
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 week, which is Monday
W	Same day of the week as the first day of the month
DDD DD J	Day of year
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

Format Model for TO_CHAR of TimesTen Datetime Data 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 does not 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)

Format	Description
SS	Second (0-59)
FF	Fractions of a second to a precision of six digits
FFn	Fractions of a second to the precision specified by n
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 or 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 leading spaces are suppressed.
$	Leading dollar sign

CASE Expressions

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

Instead of using a series of IF statements, the CASE expression enables you to use a series of conditions that return the appropriate values when the conditions are met. With CASE, 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 an ELSE NULL clause to the expression.

Description

You cannot specify the CASE expression in the value clause of an INSERT statement.

Examples

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;
```

To specify a simple CASE statement that specifies the value of a color, use the following.

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

ROWID Pseudocolumn

TimesTen assigns a unique id called a rowid to each row stored in a table. This *rowid* (called a ROWID pseudocolumn) has data type ROWID.

In TimesTen Scaleout, the ROWID pseudocolumn has a different meaning than in TimesTen Classic.

In TimesTen Scaleout:

- TimesTen Scaleout uses ROWID to ensure uniqueness across all elements.
- ROWID is the identifier of a specific copy of a row. If that copy is not available because the element that has the copy is not available, then you cannot access the row by ROWID. In this case, you should access the row by primary key.
- Each copy of a row has different ROWID values. This is true for a duplicate distribution scheme where K-safety is set to 1 and for all tables (no matter what the distribution

scheme is) where K-safety is set to 2. In these cases, when using ROWID based access, TimesTen Scaleout returns the value of the ROWID in the first data space.

- Applications should not store ROWID values and try to use these values later.
- ROWID values may change if the location of the data changes (through data redistribution).

  For information on the ttGridAdmin dbDistribute command, see Set or Modify the Distribution Scheme of a Database (dbDistribute) in the *Oracle TimesTen In-Memory Database Reference*.
- TimesTen Scaleout does not support the ROWID data type.

In TimesTen Classic:

- You can examine a rowid by querying the ROWID pseudocolumn.
- The ROWID is a pseudocolumn (not an actual column) and thus does not require database space. You cannot update, index, or drop ROWID.
- The ROWID value persists throughout the life of the table row.
- ROWID values persist through recovery, backup and restore operations. However copies of rows that are created as a result of replication or as a result of running ttMigrate to migrate rows out of the database and then back into the database or running ttBulkCp to copy rows out of the database and then back into the database have different ROWID values than the original rows.

For TimesTen Scaleout, see Understanding ROWID in Data Distribution in *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

For TimesTen Classic, see Expression Specification for more information on rowids. See ROWID Data Type for more information about the ROWID data type.

ROWNUM Pseudocolumn

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 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 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
```

Pseudocolumns in TimesTen Scaleout

Pseudocolumns are not actual columns in a table but behave like columns. A pseudocolumn is an assigned value used in the same context as a column, but is not stored.

You can perform select operations, but you cannot perform insert or update operations on a pseudocolumn.

Pseudocolumns in TimesTen Scaleout:

- elementId#: An element stores a portion of the database. Use the elementId# pseudocolumn to determine the element from which you accessed the row. This pseudocolumn returns a NOT NULL TT_INTEGER data type.
- replicaSetId#: Use this pseudocolumn to determine the replica set in which the row is stored. This pseudocolumn returns a NOT NULL TT_INTEGER data type.
- dataspaceId#: Use this pseudocolumn to determine the data space in which the copy of the row resides. This pseudocolumn returns a NOT NULL TT_INTEGER data type.

> **Note**
>
> - For DML operations, use replicaSetId# instead of elementId#. This is also true for SELECT...FOR UPDATE.
> - For SELECT operations, use replicaSetId# unless you want to select rows from a specific element. In this case, use the TT_GridQueryExec (GLOBAL) optimizer hint with your SELECT statement. See "TT_GridQueryExec Optimizer Hint" for more information.

These sections illustrate how to use pseudocolumns:

- Using Pseudocolumns to Locate Data
- Working With Pseudocolumns and Duplicate Tables
- Using Pseudocolumns to Locate a Local Element
- Displaying the Element Id Associated With an Instance

Using Pseudocolumns to Locate Data

This example illustrates how to use pseudocolumns to locate data. It determines the element to which the application is connected. It then issues a query on the customers table and returns the elementId#, replicaSetId#, and dataspaceId# where the data is located.

```
Command> SELECT elementid# FROM dual;
< 1 >
1 row found.
```

```
Command> SELECT elementId#,replicasetid#,dataspaceId#,cust_id,last_name,first_name
       FROM customers WHERE cust_id BETWEEN 910 AND 920
       ORDER BY cust_id, last_name, first_name;
< 3, 2, 1, 910, Riley, Tessa >
< 1, 1, 1, 911, Riley, Rashad >
< 1, 1, 1, 912, Riley, Emma >
< 1, 1, 1, 913, Rivera, Erin >
< 1, 1, 1, 914, Roberts, Ava >
< 1, 1, 1, 915, Roberts, Lee >
< 3, 2, 1, 916, Roberts, Clint >
< 5, 3, 1, 917, Robertson, Faith >
< 3, 2, 1, 918, Robinson, Miguel >
< 3, 2, 1, 919, Robinson, Mozell >
< 5, 3, 1, 920, Rodgers, Darryl >
11 rows found.
```

Working With Pseudocolumns and Duplicate Tables

This example illustrates how to use pseudocolumns with duplicate tables. It uses the ttIsql describe command on the account_status table to validate the table has a duplicate distribution scheme. The example then issues a query from a connection that has been connected to element 1. The example returns the elementId#, replicasetId#, and dataspaceId# so the location of the data can be determined. The example repeats the same query from a connection that has been connected to element 2. The example illustrates the data is located on the element to which the application is connected and thus is present in every element of the database (duplicate distribution scheme).

```
Command> describe account_status;

Table SAMPLEUSER.ACCOUNT_STATUS:
  Columns:
   *STATUS                     NUMBER (2) NOT NULL
    DESCRIPTION                VARCHAR2 (100) INLINE NOT NULL
  DUPLICATE

1 table found.
(primary key columns are indicated with *)
Command> SELECT elementId# FROM dual;
< 1 >
1 row found.

Command> SELECT elementId#,replicaSetId#,dataspaceId#, *
       FROM account_status;
< 1, 1, 1, 10, Active - Account is in good standing >
< 1, 1, 1, 20, Pending - Payment is being processed >
< 1, 1, 1, 30, Grace - Automatic payment did not process successfully >
< 1, 1, 1, 40, Suspend - Account is in process of being disconnected >
< 1, 1, 1, 50, Disconnected - You can no longer make calls or receive calls >
5 rows found.
```

Issue the same query from a connection to element 2.

```
Command> SELECT elementid# from dual;
< 2 >
1 row found.

Command> SELECT elementId#,replicaSetId#,dataspaceId#, *
       FROM account_status;
< 2, 1, 2, 10, Active - Account is in good standing >
< 2, 1, 2, 20, Pending - Payment is being processed >
< 2, 1, 2, 30, Grace - Automatic payment did not process successfully >
```

```
< 2, 1, 2, 40, Suspend - Account is in process of being disconnected >
< 2, 1, 2, 50, Disconnected - You can no longer make calls or receive calls >
5 rows found.
```

Using Pseudocolumns to Locate a Local Element

This example illustrates how to use pseudocolumns to return information for the element to which the application is connected. It assumes you have created a grid with six data instances and K-safety set to 2. The purpose of this example is to show you how to identify the element id, replica set, and data space group for the element to which the application is connected.

In this example, your connection is connected to element 1. Selecting from the `dual` table returns the element id, replica set id, and data space id of the current local connection. In this example, element 1 is in replica set 1 and data space 1.

```
Command> SELECT elementId#,replicaSetId#,dataspaceId# FROM dual;
< 1, 1, 1 >
1 row found.
```

In this example, your connection is connected to element 3. Element 3 is in replica set 2 and in data space 1.

```
Command> SELECT elementId#,replicaSetId#,dataspaceId# FROM dual;
< 3, 2, 1 >
1 row found.
```

Displaying the Element Id Associated With an Instance

This example illustrates how to use the `ttGridAdmin dbStatus -element` command to display the element id associated with each instance. This command also gives the status of each element. (You must issue this command from the active management instance and you must issue it as the instance administrator.)

See Monitor the Status of a Database (dbStatus) in the *Oracle TimesTen In-Memory Database Reference* for more information.

```
$ ttGridAdmin dbStatus -element
Database database1 element level status as of Thu Apr  5 12:57:44 PDT 2018

Host            Instance Elem Status Date/Time of Event  Message
--------------- -------- ---- ------ ------------------- -------
host1           instance1    1 opened 2018-04-05 11:15:33
host2           instance2    2 opened 2018-04-05 11:15:33
host3           instance3    3 opened 2018-04-05 11:15:33
host4           instance4    4 opened 2018-04-05 11:15:33
host5           instance5    5 opened 2018-04-05 11:15:33
host6           instance6    6 opened 2018-04-05 11:15:33
```

4

Functions

Functions manipulate data and return a result. In addition to an alphabetical listing of all functions, this chapter contains an overview of functions including:

- Numeric Functions
- Bitwise Functions
- Character Functions Returning Character Values
- Character Functions Returning Number Values
- String Functions
- LOB Functions
- NLS Character Set Functions
- General Comparison Functions
- Null-Related Comparison Functions
- Conversion Functions
- Datetime Functions
- Aggregate Functions
- Analytic Functions
- Encoding and Decoding Functions
- User and Session Functions
- Function in TimesTen Scaleout

Numeric Functions

Numeric functions accept numeric input and return numeric values. The numeric functions are:

- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- COS
- COSH
- CEIL
- EXP
- FLOOR
- LN

- LOG
- MOD
- POWER
- ROUND (Expression)
- SIGN
- SIN
- SINH
- SQRT
- TAN
- TANH
- TRUNC (Expression)

Bitwise Functions

Bitwise functions include:

- BITAND
- BITNOT
- BITOR
- BITXOR

Character Functions Returning Character Values

The character functions that return character values are:

- CHR
- CONCAT
- LOWER and UPPER
- LPAD
- LTRIM
- NCHR
- NLSSORT
- REPLACE
- RPAD
- RTRIM
- SOUNDEX
- SUBSTR, SUBSTRB, SUBSTR4
- TRIM

Character Functions Returning Number Values

Character functions that return number values are:

- ASCIISTR
- INSTR, INSTRB, INSTR4
- LENGTH, LENGTHB, LENGTH4

String Functions

TimesTen supports these string functions in SELECT statements:

- INSTR, INSTRB, INSTR4
- LENGTH, LENGTHB, LENGTH4
- SUBSTR, SUBSTRB, SUBSTR4

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

LOB Functions

LOB functions are not supported in TimesTen Scaleout.

The following EMPTY_* functions initialize LOBs to a non-null value:

- EMPTY_BLOB
- EMPTY_CLOB

The following TO_* functions convert specific data types into the desired LOB data type.

- TO_BLOB
- TO_CLOB
- TO_LOB
- TO_NCLOB

NLS Character Set Functions

The NLS character set functions return information about the specified character set.

- NLS_CHARSET_ID
- NLS_CHARSET_NAME

General Comparison Functions

The general comparison functions perform comparisons between input expressions. The general comparison functions are:

- GREATEST
- LEAST

Null-Related Comparison Functions

The null-related comparison functions compare expressions against NULL or return NULL based on comparison of expressions. The null-related comparison functions are:

- COALESCE
- NULLIF
- NVL

Conversion Functions

Conversion functions convert a value from one data type to another. Some of the conversion function names follow the convention of TO_*datatype*.

The SQL conversion functions are:

- ASCIISTR
- CAST
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- TO_CHAR
- TO_DATE
- TO_NUMBER
- UNISTR

Datetime Functions

For a full description of the datetime data types, see "Datetime Data Types".

The datetime functions are:

- ADD_MONTHS
- EXTRACT
- MONTHS_BETWEEN
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- ROUND (Date)
- SYSDATE and GETDATE
- TIMESTAMPADD
- TIMESTAMPDIFF
- TO_DATE
- TO_TIMESTAMP
- TRUNC (Date)

Aggregate Functions

Aggregate functions perform a specific operation over all rows in a group. Aggregate functions return a single result row based on groups of rows, rather than on single rows. They are commonly used with the GROUP BY clause in a SELECT statement, where the returned rows are divided into groups. If you omit the GROUP BY clause, the aggregate functions in the select list are applied to all the rows in the queried table or view.

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

Many aggregate functions that take a single argument can use the ALL or DISTINCT keywords. The default is ALL. See each aggregate function syntax to see if ALL or DISTINCT can be used.

- Specify DISTINCT in an aggregate function to consider only distinct values of the argument expression.
- Specify ALL in an aggregate function to consider all values, including duplicates.

For example, the DISTINCT average of 1, 1, 1, and 3 is 2. The ALL average for these results is 1.5.

The ROLLUP and CUBE clauses within a GROUP BY clause produce superaggregate rows where the column values are represented by null values. Because the superaggregate rows are denoted by NULL, it can be a challenge to differentiate between query results that include a null value and the superaggregate result. In addition, within the returned subtotals, how do you find the exact level of aggregation for a given subtotal? Use the GROUP_ID, GROUPING and GROUPING_ID functions to resolve these issues.

See "Data Types" for information about the following.

- 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.

The following is a list of aggregate functions:

- AVG
- COUNT
- GROUP_ID
- GROUPING
- GROUPING_ID
- MAX
- MIN
- SUM

Analytic Functions

Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. The group of rows is called a **window** and is defined by the *analytic_clause*.

Analytic functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins, WHERE, GROUP BY, and HAVING clauses are completed before the analytic functions are processed. The final ORDER BY clause is used to order the result of analytic functions. Analytic functions can appear in the select list of a query or subquery and in the ORDER BY clause.

Analytic functions allow you to divide query result sets into groups of rows called partitions. You can define partitions on columns or expressions. You can partition a query result set into just one partition holding all rows, a few large partitions or many small partitions holding just a few rows each.

You can define a sliding window for each row in the partition. This window determines the range of rows used to perform the calculations for the current row. Window sizes are based on a physical number of rows. The window has a starting row and an ending row and the window may move at one or both ends. For example, a window defined for a cumulative sum function would have its starting row fixed at the first row of the partition and the ending rows would slide from the start point to the last row of the partition. In contrast, a window defined for a moving average would have both the start point and end point slide.

You can set the window as large as all the rows in the partition or as small as one row within a partition.

You can specify multiple ordering expressions within each function. This is useful when using functions that rank values because the second expression can resolve ties between identical values for the first expression.

Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

Restrictions:

- Analytic functions are not supported in materialized views.

The list of analytic functions follows. Functions followed by an asterisk (*) support the *WindowingClause*.

- AVG *
- COUNT *
- DENSE_RANK
- FIRST_VALUE *
- LAST_VALUE *
- MAX *
- MIN *
- RANK
- ROW_NUMBER
- SUM *

SQL Syntax

Analytic function syntax:

```
AnalyticFunctionName ([arguments]) OVER ([AnalyticClause])

AnalyticClause::= QueryPartitionClause [ORDER BY OrderByClause [,...]
                      [WindowingClause]] |
          ORDER BY OrderByClause [,...] [WindowingClause]

QueryPartitionClause::= PARTITION BY { Expression[,Expression]... |
                     (Expression [,Expression]...)
                    }
```

OrderByClause::= *Expression* [ASC|DESC] [NULLS {FIRST|LAST}]

WindowingClause::= ROWS { BETWEEN *StartPoint* AND *EndPoint* |
StartPoint
}

StartPoint::= UNBOUNDED PRECEDING | CURRENT ROW | *PosNumConstantExpr*
{ PRECEDING | FOLLOWING }

EndPoint::= UNBOUNDED FOLLOWING | CURRENT ROW | *PosNumConstantExpr*
{ PRECEDING | FOLLOWING }

Parameters

Parameter	Description
AnalyticFunctionName	Name of analytic function.
arguments	Arguments for the analytic function. Number of arguments depends on the analytic function. Refer to the particular function for specific information on the arguments to the function.
OVER ([*AnalyticClause*])	Indicates that the function is an analytic function. This clause is computed after the FROM, WHERE, GROUP BY, and HAVING clauses. If you do not specify the *AnalyticClause*, then the analytic function is evaluated over the entire result set without partitioning, ordering, or using a window.
QueryPartitionClause	Optional clause used in *AnalyticClause*. Denoted by the PARTITION BY clause. If specified, the query result set is partitioned into groups based on the *Expression* list. If you omit this clause, then the function treats all rows of the query result set as a single group. You can specify multiple analytic functions in the same query using either the same or different PARTITION keys. Valid values for *Expression* are constants, columns, non-analytic functions or function expressions.
ORDER BY *OrderByClause*	Optional clause used in *AnalyticClause*. Use this clause to specify how data is ordered within the partition. *Expression* cannot be a column alias or position. You can order the values in a partition on multiple keys each defined by *Expression* and each qualified by an ordering sequence. Analytic functions operate in the order specified in this clause. However this clause does not guarantee the order of the result. Use the ORDER BY clause of the query to guarantee the final result ordering. If you specify the ORDER BY *OrderByClause* and you do not specify either a *QueryPartitionClause* or a *WindowingClause*, then the default window is ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. If you do not specify the ORDER BY *OrderByClause*, then the order is indeterminate.

Parameter	Description
ASC | DESC	Specifies the ordering sequence (ascending or descending). ASC is the default. Clause is optional.
NULLS FIRST | NULLS LAST	Specifies whether rows that contain NULL values are specified first or last in the ordering sequence. NULLS LAST is the default for ascending order. NULLS FIRST is the default for descending order. Clause is optional.
WindowingClause	Clause is denoted by the ROWS keyword. Specifies for each row a window expressed in physical units (rows). The window is used for calculating the function result. The function is applied to all the rows in the window. The window moves through the query result set or partition from top to bottom. You cannot specify the *WindowingClause* if you have not specified the ORDER BY *OrderByClause*. The value returned by the analytic function may produce nondeterministic results unless the ordering sequence results in unique ordering. In this case, specify multiple columns in the *OrderByClause* to achieve unique ordering. For the list of functions that allow the *WindowingClause*, see "Analytic Functions".
BETWEEN...AND	Use the BETWEEN...AND clause to specify a start point (*StartPoint*) and end point (*EndPoint*) for the window. If you omit the BETWEEN...AND clause and attempt to specify one end point, then TimesTen considers this end point the start point and the end point defaults to the current row.
StartPoint	Valid values are UNBOUNDED PRECEDING, CURRENT ROW, *PosNumConstantExpr* PRECEDING or *PosNumConstantExpr* FOLLOWING. *PosNumConstantExpr* must be either a constant positive numeric value or an expression that evaluates to a constant positive numeric value.
EndPoint	Valid values are UNBOUNDED FOLLOWING, CURRENT ROW, *PosNumConstantExpr* PRECEDING or *PosNumConstantExpr* FOLLOWING. *PosNumConstantExpr* must be either a constant positive numeric value or an expression that evaluates to a constant positive numeric value.
UNBOUNDED PRECEDING	Use UNBOUNDED PRECEDING to indicate that the window starts at the first row of the partition. Cannot be used as the end point.
UNBOUNDED FOLLOWING	Use UNBOUNDED FOLLOWING to indicate that the window ends at the last row of the partition. Cannot be used as the start point.

Parameter	Description
CURRENT ROW	As a start point, CURRENT ROW specifies that the window begins at the current row. In this case, the end point cannot be *PosNumConstantExpr* PRECEDING. As an end point, CURRENT ROW specifies that the window ends at the current row. In this case, the start point cannot be *PosNumConstantExpr* FOLLOWING.
PosNumConstantExpr {PRECEDING | FOLLOWING }	If *PosNumConstantExpr* FOLLOWING is the start point, then the end point must be *PosNumConstantExpr* FOLLOWING or UNBOUNDED FOLLOWING. If *PosNumConstantExpr* PRECEDING is the end point, then the start point must be *PosNumConstantExpr* PRECEDING or UNBOUNDED PRECEDING. The end point *PosNumConstantExpr* must be greater or equal to the start point *PosNumConstantExpr*. *PosNumConstantExpr* must be either a constant positive numeric value or an expression that evaluates to a constant positive numeric value.

Encoding and Decoding Functions

The encoding and decoding functions let you inspect and decode data.

- DECODE
- TT_HASH
- VSIZE

User and Session Functions

TimesTen supports these user and session functions:

- CURRENT_USER
- SESSION_USER
- SYS_CONTEXT
- SYSTEM_USER
- UID
- USER

Function in TimesTen Scaleout

The ELEMENTIDCOMPUTE SQL function is only supported in TimesTen Scaleout.

ABS

The ABS function returns the absolute value of *Expression*.

SQL syntax

ABS(*Expression*)

Parameters

ABS has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type. Absolute value of *Expression* is returned.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER with maximum precision and scale. Otherwise, ABS returns the same data type as the numeric data type of *Expression*.
- If the value of *Expression* is NULL, NULL is returned. If the value of the *Expression* is -INF, INF is returned.

Examples

Create table abstest and define columns with type BINARY_FLOAT and TT_INTEGER. Insert values -BINARY_FLOAT_INFINITY and -10. Call ABS to return the absolute value. You see INF and 10 are the returned values:

```
Command> CREATE TABLE abstest (col1 BINARY_FLOAT, col2 TT_INTEGER);
Command> INSERT INTO abstest
       VALUES (-BINARY_FLOAT_INFINITY, -10);
1 row inserted.
Command> SELECT ABS (col1) FROM abstest;
< INF >
1 row found.
Command> SELECT ABS (col2) FROM abstest;
< 10 >
1 row found.
```

ACOS

The ACOS function returns the arc cosine of *Expression*.

SQL syntax

ACOS(*Expression*)

Parameters

ACOS has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- *Expression* must be in the range of -1 to 1. ACOS returns a value in the range of 0 to pi, expressed in radians.
- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the ACOS function to return the arc cosine of .3.

```
Command> SELECT ACOS(.3) "Arc cosine of .3" FROM dual;
< 1.2661036727794991112593187304122222822 >
1 row found.
```

ADD_MONTHS

The ADD_MONTHS function returns the date resulting from *date* plus *integer* months.

SQL syntax

ADD_MONTHS(*Date*,*Integer*)

Parameters

ADD_MONTHS has the parameters:

Parameter	Description
Date	A datetime value or any value that can be converted to a datetime value.
Integer	An integer or any value that can be converted to an integer.

Description

- The return type is always DATE regardless of the data type of *date*. Supported data types are DATE, TIMESTAMP, ORA_TIMESTAMP and ORA_DATE.
- Data types TIME, TT_TIME, TT_DATE and TT_TIMESTAMP are not supported.
- If *date* is the last day of the month or if the resulting month has fewer days than the day component of *date*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *date*.

Examples

Call the ADD_MONTHS function to add 1 month to date January 31, 2007. The last day of February is returned.

```
Command> SELECT ADD_MONTHS (DATE '2007-01-31', 1) FROM dual;
< 2007-02-28 00:00:00 >
1 row found.
```

ADD_MONTHS returns data type DATE if *date* is of type TIMESTAMP:

```
Command> DESCRIBE SELECT ADD_MONTHS (TIMESTAMP '2007-01-31 10:00:00', 1)
    FROM dual;
```

```
Prepared Statement:
  Columns:
    EXP                     DATE NOT NULL
```

Use the HR schema to select the first 5 rows of the employees table, showing employee_id, last_name and hire_date. Create new table temp_hire_date using the CREATE TABLE ... AS SELECT statement. Call ADD_MONTHS to add 23 months to the original hire_date.

```
Command> SELECT FIRST 5 employee_id, last_name, hire_date FROM employees;
< 100, King, 1987-06-17 00:00:00 >
< 101, Kochhar, 1989-09-21 00:00:00 >
< 102, De Haan, 1993-01-13 00:00:00 >
< 103, Hunold, 1990-01-03 00:00:00 >
< 104, Ernst, 1991-05-21 00:00:00 >
5 rows found.
Command> CREATE TABLE temp_hire_date (employee_id, last_name,
          hire_date) AS SELECT FIRST 5 employee_id, last_name,
        ADD_MONTHS (hire_date, 23) FROM employees;
5 rows inserted.
Command> SELECT * FROM temp_hire_date;
< 100, King, 1989-05-17 00:00:00 >
< 101, Kochhar, 1991-08-21 00:00:00 >
< 102, De Haan, 1994-12-13 00:00:00 >
< 103, Hunold, 1991-12-03 00:00:00 >
< 104, Ernst, 1993-04-21 00:00:00 >
5 rows found.
```

ASCIISTR

The ASCIISTR function takes as its argument, either a string or any 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 or expression that evaluates to a string that is passed to the ASCIISTR function. The string can be in any character set. Value can be of any supported character data types including CHAR, VARCHAR, VARCHAR2, NCHAR, NVARCHAR, NVARCHAR2, CLOB, or NCLOB data types. Both TimesTen and Oracle Database data types are supported. The ASCII version of the string in the database character set is returned. Specify N if you want to pass the string in UTF-16 format.

Description

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

Examples

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;
(Default setting AutoCommit=1)
Command> SELECT ASCIISTR (n'Aäa') FROM dual;
< A\00E4a >
1 row found.
```

ASIN

The ASIN function returns the arc sine of *Expression*.

SQL syntax

```
ASIN(Expression)
```

Parameters

ASIN has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- *Expression* must be in the range of -1 to 1. ASIN returns a value in the range of -pi/2 to pi/2, expressed in radians.
- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the ASIN function to return the arc sine of .3.

```
Command> SELECT ASIN(.3) "Arc sine of .3" FROM dual;
< .3046926540153975079720029612275291599 >
1 row found.
```

ATAN

The ATAN function returns the arc tangent of *Expression*.

SQL syntax

```
ATAN(Expression)
```

Parameters

ATAN has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- *Expression* can be in an unbounded range. ATAN returns a value in the range of -pi/2 to pi/2, expressed in radians.
- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the ATAN function to return the arc tangent of .3.

```
Command> SELECT ATAN(.3) "Arc tangent of .3" FROM dual;
< .2914567944778670919956046214328911935013 >
1 row found.
```

ATAN2

The ATAN2 function returns the arc tangent of *Expression1* and *Expression2*.

SQL syntax

ATAN2(*Expression1*,*Expression2*)

Parameters

ATAN2 has the parameters:

Parameter	Description
Expression1	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.
Expression2	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- *Expression1* can be in an unbounded range. ATAN2 returns a value in the range of -pi to pi depending on the signs of *Expression1* and *Expression2*, expressed in radians.
- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression1* or *Expression2* is NULL, or the value of both *Expression1* and *Expression2* are NULL, NULL is returned.

Example

Use the ATAN2 function to return the arc tangent of .3 and .2.

```
Command> SELECT ATAN2(.3,.2) "Arc tangent of .3 and .2" FROM dual;
< .982793723247329067985710611014666014499 7 >
1 row found.
```

AVG

Computes the arithmetic mean of the values in the argument. Null values are ignored.

SQL syntax

AVG ([ALL | DISTINCT] *Expression*) [OVER ([*AnalyticClause*])]

Parameters

AVG has the following parameters:

Parameter	Description
Expression	Can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type.
ALL	Includes 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.
OVER ([*AnalyticClause*])	If specified, indicates aggregate analytic function. For more information on analytic functions, see "Analytic Functions".

Description

- If AVG is computed over an empty table in which GROUP BY is not used, then AVG returns NULL.
- If AVG is computed over an empty group or an empty grouped table (GROUP BY is used), AVG returns nothing.
- AVG is evaluated as SUM/COUNT. The result data type is derived using the rule that is applied for the DIV operator.
- If you do not specify the *AnalyticClause* in your query, then AVG acts as an aggregate function.
- If you specify DISTINCT and the *AnalyticClause*, then you can only specify the *QueryPartitionClause*. The *OrderByClause* and *WindowingClause* are not allowed.

Examples

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 >
```

BITAND

Computes an AND operation on the bits of *expression1* and *expression2*.

SQL syntax

BITAND (*Expression1*, *Expression2*)

Parameters

BITAND has the following parameters:

Parameter	Description
Expression1 *Expression2*	AND operation computed on the bits of *Expression1* and *Expression2*.

Description

- This function returns TT_INTEGER NOT NULL.
- The BITAND function is rewritten into (expression1 & expression2). See the bitwise AND operator for more information.
- The AND operation compares two bit values. If the values are the same, the operator returns 1. If the values are different, the operator returns 0.

Examples

Use the BITAND function to return the result of the bitwise AND (&) operation on two expressions.

```
Command> SELECT BITAND (1,2) FROM dual;
< 0 >
1 row found.
```

BITNOT

Computes a NOT operation on the bits of *expression*.

SQL syntax

BITNOT (*Expression*)

Parameters

BITNOT has the following parameters:

Parameter	Description
Expression	NOT operation computed on the bits of *Expression*.

Description

- This function returns TT_INTEGER NOT NULL.

- The BITNOT function is rewritten into (~expression). See the bitwise NOT operator for more information.

BITOR

Computes an OR operation on the bits of *expression1* and *expression2*.

SQL syntax

BITOR (*Expression1*, *Expression2*)

Parameters

BITOR has the following parameters:

Parameter	Description
Expression1 *Expression2*	OR operation computed on the bits of *Expression1*, *Expression2*.

Description

- This function returns TT_INTEGER NOT NULL.
- The BITOR function is rewritten into (expression1 | expression2). See the bitwise OR operator for more information.

BITXOR

Computes an exclusive OR operation on the bits of *expression1* and *expression2*.

SQL syntax

BITXOR (*Expression1*, *Expression2*)

Parameters

BITXOR has the following parameters:

Parameter	Description
Expression1 *Expression2*	Exclusive OR operation computed on the bits of *Expression1* and *Expression2*.

Description

- This function returns TT_INTEGER NOT NULL.
- The BITXOR function is rewritten into (expression1 ^expression2). See the bitwise exclusive OR operator for more information.

CAST

Enables 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 data type. CAST is not needed to convert a NULL literal to the desired target type.

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.

Examples

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

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

CEIL

The CEIL function returns the smallest integer greater than or equal to *Expression*.

SQL syntax

```
CEIL(Expression)
```

Parameters

CEIL has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER with maximum precision and scale. Otherwise, CEIL returns the same data type as the numeric data type of *Expression*.
- If the value of *Expression* is NULL, NULL is returned. If the value of *Expression* is -INF, INF, or NaN, the value returned is -INF, INF, or NaN respectively.

Examples

Sum the commission_pct for employees in the employees table, and then call CEIL to return the smallest integer greater than or equal to the value returned by SUM. You see the value returned by the SUM function is 7.8 and the value returned by the CEIL function is 8.

```
Command> SELECT SUM (commission_pct) FROM employees;
< 7.8 >
1 row found.
Command> SELECT CEIL (SUM (commission_pct)) FROM employees;
< 8 >
1 row found.
```

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 two expressions.

Description

- This function is a generalization of the NVL function.
- Use COALESCE as a variation of the CASE Expressions. For example:

  COALESCE (*Expression1*, *Expression2*)

  is equivalent to:

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

Examples

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*)

Parameters

CONCAT has the parameters:

Parameter	Description
Expression1	A CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB expression.
Expression2	A CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB 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 treatment of NCHAR and NVARCHAR2 is similar. If one of the operands is of varying length, the result is of varying length. Otherwise the result is of a fixed length.
- The return data type of CONCAT depends on the types of *Expression1* and *Expression2*. In concatenations of two different data types, the database returns the data type that can contain the result. Therefore, if one of the arguments is a national character data type, the returned value is a national character data type. If one of the arguments is a LOB, the returned value is a LOB.

  The following table provides examples of how the return 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*)
CLOB	NCLOB	NCLOB
NCLOB	NCHAR	NCLOB
NCLOB	CHAR(*n*)	NCLOB
NCHAR(*n*)	CLOB	NCLOB

Examples

The following example concatenates first names and last names.

```
Command> SELECT CONCAT(CONCAT(first_name, ' '), last_name), salary
       FROM employees;
< Steven King, 24000 >
< Neena Kochhar, 17000 >
< Lex De Haan, 17000 >
< Alexander Hunold, 9000 >
...
107 rows found.
```

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 "Expression Specification" for a description of the || operator.

COS

The COS function returns the cosine of *Expression* (an angle expressed in radians).

SQL syntax

COS(*Expression*)

Parameters

COS has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the COS function to return the cosine of 180 degrees.

```
Command> SELECT COS(180 * 3.14159265359/180) "Cosine of 180 degrees" FROM dual;
< -.999999999999999999999999999997862483333497 >
1 row found.
```

COSH

The COSH function returns the hyperbolic cosine of *Expression*.

SQL syntax

```
COSH(Expression)
```

Parameters

COSH has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the COSH function to return the hyperbolic cosine of 0.

```
Command> SELECT COSH(0) "Hyperbolic cosine of 0" FROM dual;
< 1 >
1 row found.
```

COUNT

The COUNT function returns the number of rows returned by the query. You can use it as an aggregate or analytic function. See Aggregate Functions for more information on aggregate functions. See Analytic Functions for information on analytic functions.

SQL Syntax

```
COUNT ({* | [ALL | DISTINCT]{Expression|ROWID}})
[OVER ([AnalyticClause])]
```

Parameters

COUNT has the parameters:

Parameter	Description
Expression	Can be any numeric data type or any non-numeric type that can be implicitly converted to a numeric type. Returns the number of rows, where *Expression* is not null. For more information on the number of rows in a table, see the description for the NUMTUPS field in SYS.TABLES in *Oracle TimesTen In-Memory Database System Tables and Views Reference*.
*	If you specify *, the COUNT function returns all rows, including duplicate and null values. COUNT does not ever return null.
ALL	Includes duplicate rows in the argument of an aggregate function. If neither ALL nor DISTINCT is specified, ALL is assumed.

Parameter	Description
DISTINCT	Eliminates duplicate column values from the argument of an aggregate function.
ROWID	TimesTen assigns a unique ID called a rowid to each row stored in a table. The rowid value can be retrieved through the ROWID pseudocolumn. See ROWID Pseudocolumn for more details.
OVER ([*AnalyticClause*])	If specified, indicates aggregate analytic function. See Analytic Functions for more information on analytic functions.

Description

- The default return data type is TT_BIGINT. You can specify the TT_CountAsInt optimizer hint to control whether the COUNT function returns a data type of TT_INTEGER or a data type of TT_BIGINT. If you specify a value of 1 for the hint, the return data type is TT_INTEGER. If you specify a value of 0 (or if you do not specify this hint), the return data type is TT_BIGINT. The TT_CountAsInt optimizer hint is supported at the statement and at the connection levels. See Statement Level Optimizer Hints for more information on statement level optimizer hints. See OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for information on connection level optimizer hints.
- If an aggregate function is computed over an empty table in which GROUP BY is not used, COUNT returns 0.
- If an aggregate function is computed over an empty group or an empty grouped table (GROUP BY is used), COUNT returns nothing.
- If you do not use the *AnalyticClause* in your query, then COUNT acts as an aggregate function.
- If you specify DISTINCT and the *AnalyticClause*, then you can only specify the *QueryPartitionClause*. The *OrderByClause* and *WindowingClause* are not allowed.

Examples

Count the number of employees.

```
Command> SELECT COUNT(*) "TOTAL EMP" FROM employees;

TOTAL EMP
< 107 >
1 row found.
```

Count the number of managers by selecting out each individual manager ID without duplication.

```
Command> SELECT COUNT(DISTINCT manager_id) "Managers" FROM employees;

MANAGERS
< 18 >
1 row found.
```

CURRENT_USER

Returns the name of the TimesTen user currently connected to the database.

SQL syntax

```
CURRENT_USER
```

Parameters

CURRENT_USER has no parameters.

Examples

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

```
SELECT CURRENT_USER FROM dual;
```

DECODE

The DECODE function compares an expression to each search value one by one. If the expression is equal to the search value, 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. *Expression* can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data types. Both TimesTen and Oracle Database data types are supported.
SearchValue	An expression is compared to one or more search values.
Result	If the expression is equal to a *SearchValue*, 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.

Examples

The following example invokes the DECODE function. In the locations table, if the column country_id is equal to 'IT', the function returns 'Italy'. If the country_id is equal to 'JP', the function returns 'Japan'. If the country_id is equal to 'US', 'United States' is returned. If the country_id is not equal to 'IT' or 'JP' or 'US', the function returns 'Other'.

```
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.
```

DENSE_RANK

The DENSE_RANK function is an analytic function that computes the rank of rows in an ordered group of rows and returns the ranks as type NUMBER.

SQL syntax

DENSE_RANK () OVER ([*QueryPartitionClause*] *OrderByClause*)

Parameters

DENSE_RANK has the parameters:

Parameter	Description
QueryPartitionClause	See "Analytic Functions" for information on syntax, semantics, and restrictions.
OrderByClause	See "Analytic Functions" for information on syntax, semantics, and restrictions.

Description

- The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query. Rank values are not skipped in the event of ties. Rows with equal values for the ranking criteria receive the same rank.
- DENSE_RANK computes the rank of each row returned from a query with respect to the other rows, based on the values of the *Expressions* in the *OrderByClause*.

Example

Select the department name, employee name, and salary of all employees who work in the human resources or purchasing department. Compute a rank for each unique salary in each of the two departments. The salaries that are equal receive the same rank.

```
Command> SELECT d.department_name, e.last_name, e.salary, DENSE_RANK()
          OVER (PARTITION BY e.department_id ORDER BY e.salary) AS dense
         FROM employees e, departments d
         WHERE e.department_id = d.department_id
         AND d.department_id IN ('30', '40')
         ORDER BY e.last_name, e.salary, d.department_name, dense;
< Purchasing, Baida, 2900, 4 >
< Purchasing, Colmenares, 2500, 1 >
< Purchasing, Himuro, 2600, 2 >
< Purchasing, Khoo, 3100, 5 >
< Human Resources, Mavris, 6500, 1 >
< Purchasing, Raphaely, 11000, 6 >
< Purchasing, Tobias, 2800, 3 >
7 rows found.
```

ELEMENTIDCOMPUTE

The ELEMENTIDCOMPUTE function is only supported in TimesTen Scaleout. This function returns the id of the element to which the distribution key belongs.

SQL syntax

```
ELEMENTIDCOMPUTE (Expression [,...])
```

Parameters

ELEMENTIDCOMPUTE has the parameters:

Parameter	Description
Expression	One or more expressions.

Description

- The ELEMENTIDCOMPUTE SQL function accepts one or more expressions. This list of expressions represents a hash distribution key. The element id returned by this function is stable within the transaction. If the TimesTen Scaleout topology changes, the element id of the particular row may change since the row may be mapped to a different element.
- You can use this function to predict into which element a particular distribution key is inserted.
- ELEMENTIDCOMPUTE returns a TT_INTEGER data type.

> **Note**
>
> If you have set K-safety to 2, ELEMENTIDCOMPUTE returns the id of one of the elements of the replica set. The value returned may not be the element id to which you are connected even though the data row maps to the local element id.

Examples

These examples illustrate the use of the ELEMENTIDCOMPUTE function:

This example, (the first example), invokes the ELEMENTIDCOMPUTE function to return the element id of one element in the replica set to which the cust_id distribution key belongs. The query also returns the cust_id, last_name, and first_name columns from the customers table where the cust_id is between 910 and 920.

```
Command> SELECT ELEMENTIDCOMPUTE (cust_id), cust_id,last_name,first_name
       FROM customers WHERE cust_id BETWEEN 910 AND 920
       ORDER BY cust_id, last_name, first_name;
< 3, 910, Riley, Tessa >
< 1, 911, Riley, Rashad >
< 1, 912, Riley, Emma >
< 1, 913, Rivera, Erin >
< 1, 914, Roberts, Ava >
< 1, 915, Roberts, Lee >
< 3, 916, Roberts, Clint >
< 5, 917, Robertson, Faith >
< 3, 918, Robinson, Miguel >
```

```
< 3, 919, Robinson, Mozell >
< 5, 920, Rodgers, Darryl >
11 rows found.
```

In this example, (the second example), elements 1 and 2 are in the same replica set. This example shows a connection to element 1 and a second connection to the replica (element 2). When connected to element 2, the value returned from ELEMENTIDCOMPUTE is not element 2, even though the data row maps to element 2. This example illustrates that the value returned may not be the element id to which the application is connected.

```
Command> SELECT elementId# FROM dual;
< 1 >
1 row found.

Command> SELECT FIRST 5 ELEMENTIDCOMPUTE (cust_id), elementid# FROM customers;
< 1, 1 >
< 1, 1 >
< 1, 1 >
< 1, 1 >
< 1, 1 >
5 rows found.

Command> SELECT elementId# FROM dual;
< 2 >
1 row found.

Command> SELECT FIRST 5 ELEMENTIDCOMPUTE (cust_id), elementid# FROM customers;
< 1, 2 >
< 1, 2 >
< 1, 2 >
< 1, 2 >
< 1, 2 >
5 rows found.
```

This example, (the third example), illustrates how to use the ELEMENTIDCOMPUTE function to predict into which element a particular row of data gets inserted. In the customers table, there are 1 to 1000 cust_id values. Predict into which element cust_id 1003, 2000 or 2400 is inserted. For example, a cust_id value of 1003, if inserted into the customers table, is predicted to reside in the replica set containing element 5.

```
Command> SELECT FIRST 1 ELEMENTIDCOMPUTE (CAST (? AS NUMBER)) FROM customers;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 '_QMARK_1' (NUMBER) > 1003
< 5 >
1 row found.
```

A cust_id value of 2000 if inserted into the customers table is predicted to reside in the replica set containing element 3.

```
Command> SELECT FIRST 1 ELEMENTIDCOMPUTE (CAST (? AS NUMBER)) FROM customers;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.
```

```
Enter Parameter 1 '_QMARK_1' (NUMBER) > 2000
< 3 >
1 row found.
```

A cust_id value of 2400 if inserted into the customers table is predicted to reside in the replica set containing element 1.

```
Command> SELECT FIRST 1 ELEMENTIDCOMPUTE (CAST (? AS NUMBER)) FROM customers;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 '_QMARK_1' (NUMBER) > 2400
< 1 >
1 row found.
```

EMPTY_BLOB

A BLOB column can be initialized to a zero-length, empty BLOB using the EMPTY_BLOB function. This function initializes the LOB to a non-null value, so can be used for initializing any BLOB that has been declared as NOT NULL.

This function is not supported in TimesTen Scaleout.

SQL syntax

```
EMPTY_BLOB ()
```

Parameters

EMPTY_BLOB has no parameters.

Description

You can only use EMPTY_BLOB in the VALUES clause of an INSERT statement or the SET clause of an UPDATE statement.

Examples

The following example uses the EMPTY_BLOB function to initialize a non-null BLOB column to a zero-length value.

```
Command> CREATE TABLE blob_content (id NUMBER PRIMARY KEY,
          blob_column BLOB NOT NULL); -- Does not allow a NULL value

Command> INSERT INTO blob_content (id, blob_column)
        VALUES (1, EMPTY_BLOB( ) );
1 row inserted.
```

EMPTY_CLOB

A CLOB or NCLOB column can be initialized to a zero-length, empty CLOB or NCLOB using the EMPTY_CLOB function. Both CLOB and NCLOB data types are initialized with the EMPTY_CLOB function. This function initializes the LOB to a non-null value, so can be used for initializing any CLOB or NCLOB that has been declared as NOT NULL.

This function is not supported in TimesTen Scaleout.

SQL syntax

```
EMPTY_CLOB ()
```

Parameters

EMPTY_CLOB has no parameters.

Description

You can only use EMPTY_CLOB in the VALUES clause of an INSERT statement or the SET clause of an UPDATE statement.

Examples

The following example uses the EMPTY_CLOB function to initialize a non-null CLOB column to a zero-length value.

```
Command> CREATE TABLE clob_content (id NUMBER PRIMARY KEY,
          clob_column CLOB NOT NULL ); -- Does not allow a NULL value

Command> INSERT INTO clob_content (id, clob_column)
        VALUES (1, EMPTY_CLOB( ) );
1 row inserted.
```

EXP

The EXP function returns e raised to the nth power (where e = 2.71828183...).

SQL syntax

EXP(*Expression*)

Parameter

EXP has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- EXP returns a value of the same type as *Expression*.
- If Expression is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the EXP function to return e to the fourth power.

```
Command> SELECT EXP(4) "e to the 4th power" FROM dual;
< 54.5981500331442390781102612028608784030 8 >
1 row 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.

SQL syntax

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

Parameters

EXTRACT has the following parameters:

Parameter	Description
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 *DateTimeField* and *DateTimExpression* or *IntervalExpression* 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.

Examples

The following example extracts the second field out of the interval result sysdate-t1.createtime.

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

The following example extracts the second field out of sysdate from the dual system table.

```
Command> SELECT EXTRACT (SECOND FROM sysdate) FROM dual;
< 20 >
1 row found.
```

FIRST_VALUE

The FIRST_VALUE function is an analytic function that returns the first value in an ordered set of values.

SQL syntax

FIRST_VALUE (*Expression* [IGNORE NULLS]) OVER (*AnalyticClause*)

Parameters

FIRST_VALUE has the parameters:

Parameter	Description
Expression	For information on supported *Expressions*, see "Analytic Functions".
IGNORE NULLS	Specify IGNORE NULLS if you want FIRST_VALUE to return the first non-null value in the set or NULL if all values in the set are NULL. Clause is optional.
OVER (*AnalyticClause*)	For information on syntax, semantics, and restrictions, see "Analytic Functions".

Description

- If the first value in the set is NULL, then FIRST_VALUE returns NULL unless you specify IGNORE NULLS. Specify IGNORE NULLS if you want the function to return the first non-null value in the set or NULL if all values in the set are NULL.

Example

Use the FIRST_VALUE function to select for each employee in department 90, the last name of the employee with the lowest salary.

```
Command> SELECT department_id, last_name, salary, FIRST_VALUE (last_name) OVER
         (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS lowest_sal
        FROM
         (SELECT * FROM employees WHERE department_id = 90 ORDER BY employee_id)
          ORDER BY department_id, last_name, salary, lowest_sal;
< 90, De Haan, 17000, Kochhar >
< 90, King, 24000, Kochhar >
< 90, Kochhar, 17000, Kochhar >
3 rows found.
```

FLOOR

The FLOOR function returns the largest integer equal to or less than *Expression*.

SQL syntax

```
FLOOR (Expression)
```

Parameters

FLOOR has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER with maximum precision and scale. Otherwise, FLOOR returns the same data type as the numeric data type of *Expression*.
- If the value of *Expression* is NULL, NULL is returned. If the value of *Expression* is -INF, INF, or NaN, the value returned is -INF, INF, or NaN respectively.

Examples

Sum the commission_pct for employees in the employees table. Then call FLOOR to return the largest integer equal to or less than the value returned by SUM. You see the value returned by the SUM function is 7.8 and the value returned by the FLOOR function is 7:

```
Command> SELECT SUM (commission_pct) FROM employees;
< 7.8 >
1 row found.
Command> SELECT FLOOR (SUM (commission_pct)) FROM employees;
< 7 >
1 row found.
```

GREATEST

The GREATEST function returns the greatest of the list of one or more expressions.

SQL syntax

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

Parameters

GREATEST has the parameter:

Parameter	Description
Expression [,...]	List of one or more expressions that is evaluated to determine the greatest expression value. Operand or column can be numeric, character or date. Each expression in the list must be from the same data type family.

Description

- Each expression in the list must be from the same data type family or date subfamily. Data type families include numeric, character and date. The date family includes four subfamilies: date family, TIME family, TT_DATE family, and TT_TIMESTAMP family. As an example, do not specify a numeric expression and a character expression in the list of expressions. Similarly, do not specify a date expression and a TT_TIMESTAMP expression in the list of expressions.
- If the first *Expression* is numeric, then TimesTen determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type before the comparison, and returns that data type.
- If the first *Expression* is in the character family, and the operand or column is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If the operand or column is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. The returned data type length is equal to the length of the largest expression. If one operand or column is of type CHAR or VARCHAR2 and the second operand or column is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2.
- TimesTen uses nonpadded comparison semantics for data types from the character family.
- If the first expression is in the date family, the data type returned is the same data type as the first expression.
- If any of the expressions is NULL, the result is NULL.

- If the first *Expression* is in the character family, and the operand or column is of type TT_CHAR or TT_VARCHAR, the data type returned is TT_VARCHAR. If the operand or column is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR. The returned data type length is equal to the largest of the expressions.
- You can specify a maximum of 256 expressions.

Use the GREATEST function to return the string with the greatest value:

```
Command> SELECT GREATEST ('GREAT', 'GREATER', 'GREATEST') FROM dual;
< GREATEST >
1 row found.
```

Use the GREATEST function to return the numeric expression with the greatest value. In this example, BINARY_DOUBLE is the data type with the highest numeric precedence, so arguments are implicitly converted to BINARY_DOUBLE before the comparison and the data type BINARY_DOUBLE is returned:

```
Command> SELECT GREATEST (10, 10.55, 10.1D) FROM dual;
< 10.5500000000000 >
1 row found.
```

Use the DESCRIBE command to confirm the data type returned is BINARY_DOUBLE:

```
Command> DESCRIBE SELECT GREATEST (10, 10.55, 10.1D) FROM dual;

Prepared Statement:
  Columns:
    EXP                     BINARY_DOUBLE NOT NULL
```

Use the GREATEST function to return the DATE expression with the greatest value. DATE and TIMESTAMP are in the same date family.

```
Command> SELECT GREATEST (DATE '2007-09-30',TIMESTAMP '2007-09-30:10:00:00')
       FROM dual;
< 2007-09-30 10:00:00 >
1 row found.
```

Attempt to use the GREATEST function to return the greatest value in the list of TT_DATE and TT_TIMESTAMP expressions. You see an error because TT_DATE and TT_TIMESTAMP are in different date subfamilies and cannot be used in the same list of expressions.

```
Command> SELECT GREATEST (TT_DATE '2007-09-30',
                 TT_TIMESTAMP '2007-09-30:10:00:00')
       FROM dual;
2817: Invalid data type TT_TIMESTAMP for argument 2 for function GREATEST
The command failed.
```

Use the GREATEST function to return the TT_DATE expression with the greatest value.

```
Command> SELECT GREATEST (TT_DATE '2007-09-30',
                 TT_DATE '2007-09-29',
                 TT_DATE '2007-09-28')
       FROM dual;
< 2007-09-30 >
1 row found.
```

GROUP_ID

The GROUP_ID function identifies duplicate groups in a SELECT query resulting from a GROUP BY clause. This function returns the number 0 for a unique group; any subsequent duplicate grouping row receives a higher number, starting with 1. The GROUP_ID function filters out duplicate groupings from the query result. If you have complicated queries that may generate duplicate values, you can eliminate those rows by including the HAVING GROUP_ID() = 0 condition.

> **Note**
>
> See "GROUP BY Clause" for details on the GROUP BY clause. See "SELECT" for details on the HAVING clause.

Syntax

The GROUP_ID function is applicable only in a SELECT statement that contains a GROUP BY clause; it can be used in the select list and HAVING clause of the SELECT query.

```
GROUP_ID()
```

Parameters

GROUP_ID has no parameters.

Example

The following example shows how GROUP_ID returns 0 for a unique group and a number > 0 to identify duplicate groups. The following example prints out the department number, manager id and the sum of the salary within the manager. The resulting output is grouped using the ROLLUP clause on the manager and department providing superaggregate results.

```
Command> SELECT department_id as DEPT, manager_id AS MGR,
          GROUP_ID(), SUM(salary) as SALARY
        FROM employees
        WHERE manager_id > 146
        GROUP BY manager_id, ROLLUP(manager_id, department_id)
        ORDER BY manager_id, department_id;

DEPT, MGR, EXP, SALARY
< 80, 147, 0, 46600 >
< <NULL>, 147, 1, 46600 >
< <NULL>, 147, 0, 46600 >
< 80, 148, 0, 51900 >
< <NULL>, 148, 0, 51900 >
< <NULL>, 148, 1, 51900 >
< 80, 149, 0, 43000 >
< <NULL>, 149, 0, 7000 >
< <NULL>, 149, 0, 50000 >
< <NULL>, 149, 1, 50000 >
< 20, 201, 0, 6000 >
< <NULL>, 201, 0, 6000 >
< <NULL>, 201, 1, 6000 >
< 110, 205, 0, 8300 >
< <NULL>, 205, 0, 8300 >
< <NULL>, 205, 1, 8300 >
16 rows found.
```

GROUPING

The GROUPING function enables you to determine whether a NULL is a stored NULL or an indication of a subtotal or grand total. Using a single column as its argument, GROUPING returns a 1 when it encounters a null value created by a ROLLUP or CUBE operation, indicating a subtotal or grand total. Any other type of value, including a stored NULL, returns a 0.

> **Note**
>
> See "GROUP BY Clause" for details on ROLLUP and CUBE clauses.

Syntax

The GROUPING function is applicable only in a SELECT statement that contains a GROUP BY clause. It can be used in the select list and HAVING clause of the SELECT query that includes the GROUP BY clause. The expression indicated in the GROUPING function syntax must match one of the expressions contained in the GROUP BY clause.

The following syntax uses GROUPING to identify the results from the expression listed as an aggregate or not:

```
SELECT ...  [GROUPING(Expression) ... ]  ...
  GROUP BY ... { RollupCubeClause | GroupingSetsClause } ...
```

The following syntax uses GROUPING within a HAVING clause to identify the results from the expression listed as an aggregate or not:

```
SELECT ...
  GROUP BY ... { RollupCubeClause | GroupingSetsClause } ...
 HAVING GROUPING(Expression) = 1
```

Parameters

Parameter	Description
Expression	Valid expression syntax. See Expressions.
RollupCubeClause	The GROUP BY clause may include one or more ROLLUP or CUBE clauses. See "GROUP BY Clause" for full details.
GroupingSetsClause	The GROUP BY clause may include one or more GROUPING SETS clauses. The GROUPING SETS clause enables you to explicitly specify which groupings of data that the database returns.

Examples

The following example shows how the grouping function returns a '1' when it encounters the grand total for the department.

```
Command> columnlabels on;

Command> SELECT department_id AS DEPT,
       GROUPING(department_id) AS DEPT_GRP, SUM(salary) AS SALARY
     FROM emp_details_view
     GROUP BY ROLLUP(department_id)
     ORDER BY department_id;
```

```
DEPT, DEPT_GRP, SALARY
< 10, 0, 4400 >
< 20, 0, 19000 >
< 30, 0, 24900 >
< 40, 0, 6500 >
< 50, 0, 156400 >
< 60, 0, 28800 >
< 70, 0, 10000 >
< 80, 0, 304500 >
< 90, 0, 58000 >
< 100, 0, 51600 >
< 110, 0, 20300 >
< <NULL>, 1, 684400 >
12 rows found.
```

The following example shows that you can use the GROUPING function for each column to determine which null values are for the totals.

```
Command> SELECT department_id AS DEPT, job_id AS JOB,
     GROUPING(department_id) AS DEPT_GRP, GROUPING(job_id) AS JOB_GRP,
     GROUPING_ID(department_id, job_id) AS GRP_ID, SUM(salary) AS SALARY
    FROM emp_details_view
    GROUP BY CUBE(department_id, job_id)
    ORDER BY department_id, job_id, grp_id ASC;

DEPT, JOB, DEPT_GRP, JOB_GRP, GRP_ID, SALARY
< 10, AD_ASST, 0, 0, 0, 4400 >
< 10, <NULL>, 0, 1, 1, 4400 >
< 20, MK_MAN, 0, 0, 0, 13000 >
< 20, MK_REP, 0, 0, 0, 6000 >
< 20, <NULL>, 0, 1, 1, 19000 >
< 30, PU_CLERK, 0, 0, 0, 13900 >
< 30, PU_MAN, 0, 0, 0, 11000 >
< 30, <NULL>, 0, 1, 1, 24900 >
...
< 110, AC_ACCOUNT, 0, 0, 0, 8300 >
< 110, AC_MGR, 0, 0, 0, 12000 >
< 110, <NULL>, 0, 1, 1, 20300 >
< <NULL>, AC_ACCOUNT, 1, 0, 2, 8300 >
< <NULL>, AC_MGR, 1, 0, 2, 12000 >
< <NULL>, AD_ASST, 1, 0, 2, 4400 >
< <NULL>, AD_PRES, 1, 0, 2, 24000 >
< <NULL>, AD_VP, 1, 0, 2, 34000 >
< <NULL>, FI_ACCOUNT, 1, 0, 2, 39600 >
< <NULL>, FI_MGR, 1, 0, 2, 12000 >
< <NULL>, HR_REP, 1, 0, 2, 6500 >
< <NULL>, IT_PROG, 1, 0, 2, 28800 >
< <NULL>, MK_MAN, 1, 0, 2, 13000 >
< <NULL>, MK_REP, 1, 0, 2, 6000 >
< <NULL>, PR_REP, 1, 0, 2, 10000 >
< <NULL>, PU_CLERK, 1, 0, 2, 13900 >
< <NULL>, PU_MAN, 1, 0, 2, 11000 >
< <NULL>, SA_MAN, 1, 0, 2, 61000 >
< <NULL>, SA_REP, 1, 0, 2, 243500 >
< <NULL>, SH_CLERK, 1, 0, 2, 64300 >
< <NULL>, ST_CLERK, 1, 0, 2, 55700 >
< <NULL>, ST_MAN, 1, 0, 2, 36400 >
< <NULL>, <NULL>, 1, 1, 3, 684400 >
50 rows found.
```

GROUPING_ID

The GROUPING_ID function returns a number that shows the exact GROUP BY level of aggregation resulting from a ROLLUP or CUBE clause.

> **Note**
>
> See "GROUP BY Clause" for details on ROLLUP and CUBE clauses.

The GROUPING_ID function takes the ordered list of grouping columns from the ROLLUP or CUBE as input and computes the grouping ID as follows:

1. Applies the GROUPING function to each of the individual columns in the list. The result is a set of ones and zeros, where 1 represents a superaggregate generated by the ROLLUP or CUBE.
2. Puts these ones and zeros in the same order as the order of the columns in its argument list to produce a bit vector.
3. Converts this bit vector from a binary number into a decimal (base 10) number, which is returned as the grouping ID.

For instance, if you group with CUBE(department_id, job_id), the returned values are as shown in Table 4-1.

Table 4-1 GROUPING_ID Example for CUBE(department_id, job_id)

Aggregation Level	Bit Vector	GROUPING_ID
Normal grouping rows for department and job	0 0	0
Subtotal for department_id, aggregated at job_id	0 1	1
Subtotal for job_id, aggregated at department_id	1 0	2
Grand total	1 1	3

The GROUPING_ID function can be used in a query to filter rows so that only the summary rows are displayed. You can use the GROUPING_ID function in the HAVING clause to restrict output to only those rows that contain totals and subtotals. This can be accomplished when adding a comparison of the GROUPING_ID function results as greater than zero in the HAVING clause.

Syntax

The GROUPING_ID function is applicable only in a SELECT statement that contains the GROUP BY clause, a GROUPING function, and one of the following clauses: ROLLUP, CUBE or GROUPING SETS clauses. It can be used in the select list and HAVING clause of the SELECT query.

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

Parameters

Parameter	Description
Expression	Valid expression syntax. See Expressions.

Examples

The following example has the HAVING clause filter on the GROUPING_ID function, where the returned value is greater than zero. This excludes rows that do not contain either a subtotal or grand total. The following example shows the subtotals for the departments are identified with a group ID of 1, subtotals for the job ID with a group ID of 2 and the grand total with a group ID of 3:

```
Command> SELECT department_id AS DEPT, job_id AS JOB,
          GROUPING_ID(department_id, job_id) AS GRP_ID,
          SUM(salary) AS SALARY
         FROM emp_details_view
         GROUP BY CUBE(department_id, job_id)
         HAVING GROUPING_ID(department_id, job_id) > 0
         ORDER BY department_id, job_id, grp_id ASC;

DEPT, JOB, GRP_ID, SALARY
< 10, <NULL>, 1, 4400 >
< 20, <NULL>, 1, 19000 >
< 30, <NULL>, 1, 24900 >
< 40, <NULL>, 1, 6500 >
< 50, <NULL>, 1, 156400 >
< 60, <NULL>, 1, 28800 >
< 70, <NULL>, 1, 10000 >
< 80, <NULL>, 1, 304500 >
< 90, <NULL>, 1, 58000 >
< 100, <NULL>, 1, 51600 >
< 110, <NULL>, 1, 20300 >
< <NULL>, AC_ACCOUNT, 2, 8300 >
< <NULL>, AC_MGR, 2, 12000 >
< <NULL>, AD_ASST, 2, 4400 >
< <NULL>, AD_PRES, 2, 24000 >
< <NULL>, AD_VP, 2, 34000 >
< <NULL>, FI_ACCOUNT, 2, 39600 >
< <NULL>, FI_MGR, 2, 12000 >
< <NULL>, HR_REP, 2, 6500 >
< <NULL>, IT_PROG, 2, 28800 >
< <NULL>, MK_MAN, 2, 13000 >
< <NULL>, MK_REP, 2, 6000 >
< <NULL>, PR_REP, 2, 10000 >
< <NULL>, PU_CLERK, 2, 13900 >
< <NULL>, PU_MAN, 2, 11000 >
< <NULL>, SA_MAN, 2, 61000 >
< <NULL>, SA_REP, 2, 243500 >
< <NULL>, SH_CLERK, 2, 64300 >
< <NULL>, ST_CLERK, 2, 55700 >
< <NULL>, ST_MAN, 2, 36400 >
< <NULL>, <NULL>, 3, 684400 >
31 rows found.
```

INSTR, INSTRB, INSTR4

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

SQL syntax

```
{INSTR | INSTRB | INSTR4} ( SourceExpr, SearchExpr [,m[,n]])
```

Parameters

INSTR has the parameters:

Parameter	Description
SourceExpr	The string to be searched to find the position of *SearchExpr*. Value can be any supported character data types including CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB data types. Both TimesTen and Oracle Database data types are supported.
SearchExpr	The substring to be found in string *SourceExpr*. If *SearchExpr* does not occur in *SourceExpr*, zero is returned. If either string is of length zero, NULL is returned.
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.

Examples

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.
```

The following example uses INSTR to provide the number of employees with a '650' area code as input to the COUNT function:

```
Command> SELECT COUNT(INSTR(phone_number, '650')) FROM employees;
< 107 >
1 row found.
```

LAST_VALUE

The LAST_VALUE function is an analytic function that returns the last value in an ordered set of values.

SQL syntax

LAST_VALUE (*Expression* [IGNORE NULLS]) OVER (*AnalyticClause*)

Parameters

LAST_VALUE has the parameters:

Parameter	Description
Expression	See "Analytic Functions" for information on supported *Expressions*.
IGNORE NULLS	Specify IGNORE NULLS if you want LAST_VALUE to return the last non-NULL value in the set or NULL if all values in the set are NULL. Clause is optional.
OVER (*AnalyticClause*)	See "Analytic Functions" for information on syntax, semantics, and restrictions.

Description

- If the last value in the set is NULL, then LAST_VALUE returns NULL unless you specify IGNORE NULLS. Specify IGNORE NULLS if you want the function to return the last non-null value in the set or NULL if all values in the set are NULL.

Example

Use the LAST_VALUE function to return for each row the hire date of the employee with the highest salary.

```
Command> SELECT last_name, salary, hire_date,
          LAST_VALUE (hire_date) OVER (ORDER BY salary
          ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
         FROM
          (SELECT * FROM employees WHERE department_id = 90 ORDER BY hire_date)
         ORDER BY last_name, salary, hire_date, lv;
< De Haan, 17000, 1993-01-13 00:00:00, 1987-06-17 00:00:00 >
< King, 24000, 1987-06-17 00:00:00, 1987-06-17 00:00:00 >
< Kochhar, 17000, 1989-09-21 00:00:00, 1987-06-17 00:00:00 >
3 rows found.
```

LEAST

The LEAST function returns the smallest of the list of one or more expressions.

SQL syntax

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

Parameters

LEAST has the parameter:

Parameter	Description
Expression [,...]	List of one or more expressions that is evaluated to determine the smallest expression value. Operand or column can be numeric, character, or date. Each expression in the list must be from the same data type family.

Description

- Each expression in the list must be from the same data type family or date subfamily. Data type families include numeric, character and date. The date family includes four subfamilies: date family, TIME family, TT_DATE family, and TT_TIMESTAMP family. As an example, do not specify a numeric expression and a character expression in the list of expressions. Similarly, do not specify a date expression and a TT_TIMESTAMP expression in the list of expressions.
- If the first *Expression* is numeric, then TimesTen determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type before the comparison, and returns that data type.
- If the first *Expression* is in the character family, and the operand or column is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If the operand or column is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. The returned data type length is equal to the length of the largest expression. If one operand or column is of type CHAR or VARCHAR2 and the second operand or column is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2.
- TimesTen uses nonpadded comparison semantics for data types from the character family.
- If the first expression is in the date family, the data type returned is the same data type as the first expression.
- If any of the expressions is NULL, the result is NULL.
- If the first *Expression* is in the character family, and the operand or column is of type TT_CHAR or TT_VARCHAR, the data type returned is TT_VARCHAR. If the operand or column is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR. The returned data type length is equal to the largest of the expressions.
- You can specify a maximum of 256 expressions.

Use the LEAST function to return the string with the smallest value:

```
Command> SELECT LEAST ('SMALL','SMALLER','SMALLEST') FROM dual;
< SMALL >
1 row found.
```

Use the LEAST function to return the numeric expression with the smallest value. In this example, NUMBER is the data type with the highest numeric precedence, so arguments are implicitly converted to NUMBER before the comparison and the data type NUMBER is returned. First describe the table leastex to see the data types defined for columns col1 and col2. Then SELECT * from leastex to see the data. Then invoke the LEAST function.

```
Command> DESCRIBE leastex;

Table SAMPLEUSER.LEASTEX:
  Columns:
    COL1                    NUMBER (2,1)
    COL2                    TT_BIGINT

1 table found.
(primary key columns are indicated with *)
Command> SELECT * FROM leastex;
< 1.1, 1 >
1 row found.
Command> SELECT LEAST (Col2,Col1) from leastex;
< 1 >
1 row found.
```

Use the DESCRIBE command to confirm that the data type returned is NUMBER:

```
Command> DESCRIBE SELECT LEAST (Col2,Col1) FROM leastex;

Prepared Statement:
  Columns:
    EXP                     NUMBER
```

Use the LEAST function to return the DATE expression with the smallest value. DATE and TIMESTAMP are in the same date family.

```
Command> SELECT LEAST (DATE '2007-09-17',
         TIMESTAMP '2007-09-17:10:00:00') FROM dual;
< 2007-09-17 00:00:00 >
1 row found.
```

Attempt to use the LEAST function to return the smallest value in the list of TT_DATE and TT_TIMESTAMP expressions. You see an error because TT_DATE and TT_TIMESTAMP are in different date subfamilies and cannot be used in the same list of expressions.

```
Command> SELECT LEAST (TT_DATE '2007-09-17',
         TT_TIMESTAMP '2007-09-17:01:00:00') FROM dual;
2817: Invalid data type TT_TIMESTAMP for argument 2 for function LEAST
The command failed.
```

Use the LEAST function to return the TIME expression with the smallest value.

```
Command> SELECT LEAST (TIME '13:59:59', TIME '13:59:58',
         TIME '14:00:00') FROM dual;
< 13:59:58 >
1 row found.
```

LENGTH, LENGTHB, LENGTH4

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

SQL syntax

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

Parameters

LENGTH has the parameter:

Parameter	Description
CharExpr	The string for which to return the length. Supported data types for *CharExpr* for are CHAR, VARCHAR2, NCHAR, or NVARCHAR2. LENGTH and LENGTHB also support CLOB, NCLOB, and BLOB data types.

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.

Examples

Determine the length of the string 'William':

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

The following determines the length of the NCLOB data:

```
Command> SELECT nclob_column FROM nclob_content;
< Demonstration of the NCLOB data type >
1 row found.

Command> SELECT LENGTH(nclob_column) FROM nclob_content;
< 36 >
1 row found.
```

LN

The LN function returns the natural logarithm of *Expression*, where the value of *Expression* is greater than 0.

SQL syntax

LN(*Expression*)

Parameters

LN has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If Expression is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the LN function to return the natural logarithm of 95.

```
Command> SELECT LN(95) "Natural logarithm of 95" FROM dual;
< 4.55387689160054083460978676511404117675 >
1 row found.
```

LOG

The LOG function returns the logarithm, base *Expression2*, of *Expression1*. The base *Expression2* can be any positive value other than 0 or 1 and *Expression1* can be any positive value.

SQL syntax

LOG(*Expression1*,*Expression2*)

Parameters

LN has the parameter:

Parameter	Description
Expression1	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.
Expression2	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression1* or *Expression2* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression1* or *Expression2* is NULL, or if both *Expression1* and *Expression2* are NULL, NULL is returned.

Example

Use the LOG function to return the natural logarithm of 100.

```
Command> SELECT LOG(10,100) "Log base 10 of 100" FROM dual;
< 2 >
1 row found.
```

LOWER and UPPER

The LOWER function converts expressions of type CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, or NCLOB to lowercase. The UPPER function converts expressions of type CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, or NCLOB to uppercase. Character semantics are 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)
```

Parameters

LOWER and UPPER have the following parameter:

Parameter	Description
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)))
```

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

LPAD

The LPAD function returns *Expression1*, left-padded to length *n* characters with the sequence of characters in *Expression2*. This function is useful for formatting the output of a query.

SQL syntax

LPAD (*Expression1*, *n* [,*Expression2*])

Parameters

LPAD has the parameters:

Parameter	Description
Expression1	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column to be left-padded. If *Expression1* is longer than *n*, then LPAD returns the portion of *Expression1* that fits in *n*.
n	Length of characters returned by the LPAD function. Must be a NUMBER integer or a value that can be implicitly converted to a NUMBER integer.
Expression2	Sequence of characters to be left-padded in *Expression1*. If you do not specify *Expression2*, the default is a single blank. Operand or column can be of type CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB.

Description

- If *Expression1* is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If *Expression1* is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. If *Expression1* is a LOB, the data type returned is the same as the LOB data type provided.
- The returned data type length is equal to *n* if *n* is a constant. Otherwise, the maximum result length of 8300 is returned.
- You can specify TT_CHAR, TT_VARCHAR, TT_NCHAR, and TT_NVARCHAR for *Expression1* and *Expression2*. If *Expression1* is of type TT_CHAR or TT_VARCHAR, the data type returned is TT_VARCHAR. If *Expression1* is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR.
- For CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB data types:
  - If either *Expression1* or *Expression2* is NULL, the result is NULL. If *n* is less than or equal to 0, the result is NULL.
- For TT_CHAR, TT_VARCHAR, TT_NCHAR and TT_NVARCHAR types:
  - If either *Expression1* or *Expression2* is not NULL and if *n* is less than or equal to 0, the result is the empty string.

Examples

The following prints out the last names of the first 5 employees, left-padded with periods out to 20 characters.

```
Command> SELECT FIRST 5 LPAD (last_name, 20, '.')
       FROM employees
       ORDER BY last_name;
< ................Abel >
< ................Ande >
```

```
< ............Atkinson >
< ..............Austin >
< ................Baer >
5 rows found.
```

Use LPAD function to left-pad the string 'LPAD Function' with string 'DEMO-ONLY' plus 2 spaces. The DEMO-ONLY string is replicated as much as it can as defined by the total characters output by the function, which is replicated three times.

```
Command> SELECT LPAD ('LPAD Function', 46, 'DEMO-ONLY  ') FROM dual;
< DEMO-ONLY  DEMO-ONLY  DEMO-ONLY  LPAD Function >
1 row found.
```

Call LPAD function with length of -1. NULL is returned.

```
Command> SELECT LPAD ('abc', -1, 'a')  FROM dual;
< <NULL> >
1 row found.
```

LTRIM

The LTRIM function removes from the left end of *Expression1* all of the characters contained in *Expression2*. TimesTen begins scanning *Expression1* from its first character and removes all characters that appear in *Expression2* until reaching a character not in *Expression2* and returns the result.

SQL syntax

```
LTRIM (Expression1 [,Expression2])
```

Parameters

LTRIM has the parameters:

Parameter	Description
Expression1	The CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column to be trimmed. If *Expression1* is a character literal, then enclose it in single quotes.
Expression2	Optional expression used for trimming *Expression1*. If *Expression2* is a character literal, enclose it in single quotes. If you do not specify *Expression2*, it defaults to a single blank. Operand or column can be of type CHAR,VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB.

Description

- If *Expression1* is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If *Expression1* is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. If *Expression1* is a CLOB or NCLOB, the data type returned is the same as the LOB data type provided. The returned data type length is equal to the data type length of *Expression1*.
- If *Expression1* is a data type defined with CHAR length semantics, the returned length is expressed in CHAR length semantics.
- If either *Expression1* or *Expression2* is NULL, the result is NULL.
- You can specify TT_CHAR, TT_VARCHAR, TT_NCHAR, and TT_NVARCHAR for *Expression1* and *Expression2*. If *Expression1* is of type TT_CHAR or TT_VARCHAR, the data type returned is

TT_VARCHAR. If *Expression1* is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR.

- If *Expression1* is of type CHAR or VARCHAR2 and *Expression2* is of type NCHAR or NVARCHAR2, then *Expression2* is demoted to CHAR or VARCHAR2 before LTRIM is invoked. The conversion of *Expression2* could be lost. If the trim character of *Expression2* is not in the database character set, then the query may produce unexpected results.
- For CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB types:
  - If all the characters in *Expression1* are removed by the LTRIM function, the result is NULL.
- For TT_CHAR, TT_VARCHAR, TT_NCHAR and TT_NVARCHAR types:
  - If all the characters in *Expression1* are removed by the LTRIM function, the result is the empty string.

Examples

Call the LTRIM function to remove left-most 'x' and 'y' from string. LTRIM removes individual occurrences of 'x' and 'y', not pattern 'xy'.

```
Command> SELECT LTRIM ('xxxyyyxyxyLTRIM Example', 'xy') FROM dual;
< LTRIM Example >
1 row found.
```

Call the LTRIM function to remove YYYY-MM-DD from SYSDATE. Call TO_CHAR to convert SYSDATE to VARCHAR2.

```
Command> SELECT LTRIM (TO_CHAR(SYSDATE), '2007-08-21') FROM dual;
<  22:54:39 >
1 row found.
```

Call LTRIM to remove all characters from *Expression1*. In the first example, the data type is CHAR, so NULL is returned. In the second example, the data type is TT_CHAR, so the empty string is returned.

```
Command> CREATE TABLE ltrimtest (col1 CHAR (4), col2 TT_CHAR (4));
Command> INSERT INTO ltrimtest VALUES ('ABBB','ABBB');
1 row inserted.
Command> SELECT LTRIM (col1, 'AB') FROM ltrimtest;
< <NULL> >
1 row found.
Command> SELECT LTRIM (col2, 'AB') FROM ltrimtest;
<  >
1 row found.
```

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. MAX is an aggregate function and can also be an aggregate analytic function. See "Aggregate Functions" for details on aggregate functions. See "Analytic Functions" for more information on analytic functions.

SQL syntax

MAX ([ALL | DISTINCT]{*Expression* | ROWID}) [OVER ([*AnalyticClause*])]

Parameters

MAX has the parameters:

Parameter	Description
Expression	Can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type.
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.
ROWID	TimesTen assigns a unique ID called a rowid to each row stored in a table. The rowid value can be retrieved through the ROWID pseudocolumn. See "ROWID Pseudocolumn" for more details.
OVER ([*AnalyticClause*])	If specified, indicates aggregate analytic function. See "Analytic Functions" for more information on analytic functions.

Description

- If MAX is computed over an empty table in which GROUP BY is not used, MAX returns NULL.
- If MAX is computed over an empty group or an empty grouped table (GROUP BY is used), MAX returns nothing.
- The result data type is the same as the source.
- If you do not use the *AnalyticClause* in your query, then MAX acts as an aggregate function.

Examples

Find the largest salary:

```
Command> SELECT MAX(salary) "Max Salary" FROM employees;

MAX SALARY
< 24000 >
1 row found.
```

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. See "Aggregate Functions" for more details on aggregate functions. MIN can also be an aggregate analytic function. See "Analytic Functions" for information.

SQL syntax

```
MIN ([ALL | DISTINCT]{Expression|ROWID}) [OVER (AnalyticClause)]
```

Parameters

MIN has the parameters:

Parameter	Description
Expression	Can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type.
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.
ROWID	TimesTen assigns a unique ID called a rowid to each row stored in a table. The rowid value can be retrieved through the ROWID pseudocolumn. See "ROWID Pseudocolumn" for more details.
OVER (*AnalyticClause*)	If specified, indicates aggregate analytic function. See "Analytic Functions" for more information on analytic functions.

Description

- If the MIN function is computed over an empty table in which GROUP BY is not used, MIN returns NULL.
- If the MIN function is computed over an empty group or an empty grouped table (GROUP BY is used), MIN returns nothing.
- The result data type is the same as the source.
- If you do not use the *AnalyticClause* in your query, then MIN acts as an aggregate function.

Examples

Show the smallest salary:

```
Command> SELECT MIN(salary) "Min Salary" FROM employees;

MIN SALARY
< 2100 >
```

Show the earliest hire date:

```
Command> SELECT MIN(hire_date) "Earliest Hire Date" FROM employees;

EARLIEST HIRE DATE
< 1987-06-17 00:00:00 >
1 row found.
```

MOD

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

SQL syntax

```
MOD(Expression1, Expression2)
```

Parameters

MOD has the following parameters:

Parameter	Description
Expression1	An INTEGER expression.

Parameter	Description
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 the "Data Types" chapter.
- The MOD function behaves differently from the classic mathematical modulus function when one of the operands is negative. The following table illustrates this difference:

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

The following example tests whether the value of the expression *m* is divisible by the value of expression *n*.

```
SELECT m, n FROM test WHERE MOD(m, n) = 0;
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns number of months between dates *date1* and *date2*.

SQL syntax

```
MONTHS_BETWEEN(date1, date2)
```

Parameters

MONTHS_BETWEEN has the parameters:

Parameter	Description
date1	A datetime value or any value that can be converted to a datetime value.
date2	A datetime value or any value that can be converted to a datetime value.

Description

Input parameters can be any combination of all supported datetime data types, excluding the TIME or TT_TIME data types. The supported datetime data types include DATE, TIMESTAMP, TT_DATE, TT_TIMESTAMP, ORA_DATE, and ORA_TIMESTAMP. See "Data Types" for details on datetime data types.

The return data type is a NUMBER.

MONTHS_BETWEEN returns number of months between dates *date1* and *date2*.

- If *date1* is later than *date2*, the returned result is positive.
- If *date1* is earlier than *date2*, the returned result is negative.
- If *date1* and *date2* are both either the same day of the month or the last day of the month, the returned result is an integer. For all other cases, the returned result is a fraction based on a 31-day month that considers the difference in time components for *date1* and *date2* parameters.

Examples

The following examples calculate months between two given dates.

```
Command> SELECT MONTHS_BETWEEN(DATE '1995-02-02', DATE '1995-01-01')
     AS Months FROM dual;

MONTHS
< 1.03225806451613 >
1 row found.

Command> SELECT MONTHS_BETWEEN(DATE '2010-02-02', DATE '2010-10-01') "Months"
    FROM dual;

MONTHS
< -7.96774193548387 >
1 row found.
```

The following command uses CAST to explicitly convert CHAR strings into timestamps. The first result is rounded to an integer.

```
Command> SELECT ROUND ( MONTHS_BETWEEN (CAST ('2010-04-15 14:13:52'
     AS TIMESTAMP), CAST ('2000-12-31 00:00:00' AS TIMESTAMP))),
     MONTHS_BETWEEN (CAST ('2010-04-15 14:13:52' AS TIMESTAMP),
     CAST ('2000-12-31 00:00:00' AS TIMESTAMP))
    FROM dual;

< 112, 111.502998805257 >
1 row found.
```

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

The following example returns the NCHAR character 187:

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

NLS_CHARSET_ID

NLS_CHARSET_ID returns the character set ID number corresponding to the character set name.

SQL syntax

```
NLS_CHARSET_ID(String)
```

Parameters

NLS_CHARSET_ID has the parameter:

Parameter	Description
String	The input string argument is a run-time VARCHAR2 value that represents the character set. This string is case-insensitive. If the input string corresponds to a supported TimesTen character set, the associated character set ID number is returned; otherwise, NULL is returned. Providing CHAR_CS returns the database character set ID number. Providing NCHAR_CS returns the national character set ID number. Other input string values are interpreted as Oracle Database NLS character set names, such as AL32UTF8.

Examples

The following example returns the character set ID number of character set US7ASCII:

```
Command> SELECT nls_charset_id('US7ASCII') FROM dual;
< 1 >
1 row found.
```

Also see the example in the next section, "NLS_CHARSET_NAME", that uses the NLS_CHARSET_ID result as input to NLS_CHARSET_NAME.

NLS_CHARSET_NAME

NLS_CHARSET_NAME returns the name of the character set corresponding to the character set ID number.

SQL syntax

```
NLS_CHARSET_NAME(Number)
```

Parameters

NLS_CHARSET_NAME has the parameter:

Parameter	Description
Number	The number represents a character set ID. If the number does not correspond to a supported TimesTen character set ID, NULL is returned.

Description

The character set name is returned as a VARCHAR2 value in the database character set.

Examples

The following example returns the database character set corresponding to character set ID number 1:

```
Command> SELECT nls_charset_name(1) FROM dual;
< US7ASCII >
1 row found.
```

The following example gets the same result, determining the name of the database character set by providing CHAR_CS as the character set name within the NLS_CHARSET_ID function, whose results are provided to the NLS_CHARSET_NAME function:

```
SELECT NLS_CHARSET_NAME(NLS_CHARSET_ID('CHAR_CS')) FROM dual;
< US7ASCII >
1 row found.
```

See the previous section, "NLS_CHARSET_ID" for related information.

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	Given the *String*, NLSSORT returns the sort key value used to sort the *String*. Supported data types for *String* are CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, and NCLOB.
['NLS_SORT = *SortName*']	*SortName* is either the linguistic sort sequence or BINARY. If you omit this parameter, then the default sort sequence for the session is used. Append to the *SortName* the suffix -ai for accent-insensitive sorting or -ci for case-insensitive sorting.

Description

- The returned sort key value is of type VARBINARY.
- You can create a linguistic index for linguistic comparisons.

Examples

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';
< Gardiner >
1 row 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';
Session altered.
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.
```

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 range index on columns:
    NLSSORT(NAME,'NLS_SORT = XDanish')
  1 index found.
1 index found on 1 table.
```

NULLIF

NULLIF compares two expressions. If the values are equal, NULLIF returns a NULL; otherwise, the function returns the first expression.

SQL syntax

NULLIF(*Expression1*, *Expression2*)

Parameters

NULLIF has the following parameters:

Parameter	Description
Expression1	The expression that is tested to see whether it is equal to *Expression2*. You cannot specify the literal NULL for *Expression1*.
Expression2	The expression that is tested to see whether it is equal to *Expression1*.

Description

- If both parameters are numeric data types, Timesten determines the argument with the higher numeric precedence, implicitly converts the other argument to this data type, and returns this data type. If the parameters are not numeric data types, they must be in the same data type family.
- LOB data types are not supported in NULLIF. The TIME data type is only supported if both columns are of the TIME data type.
- The NULLIF function is logically equivalent to the following CASE expression:

  CASE WHEN *Expression1* = *Expression2* THEN NULL ELSE *Expression1* END

> **Note**
>
> See "CASE Expressions" for more details.

Examples

The following example selects employees who have changed jobs since they were hired, which is indicated by a different job_id in the job_history table from the current job_id in the employees table. Thus, when you apply NULLIF to the old and new job_id entries, those that are the same returns a NULL; those that are different indicate those employees who have changed jobs.

```
Command> SELECT e.last_name, NULLIF(e.job_id, j.job_id) "Old Job ID"
       FROM employees e, job_history j
       WHERE e.employee_id = j.employee_id
       ORDER BY last_name, "Old Job ID";

< De Haan, AD_VP >
< Hartstein, MK_MAN >
< Kaufling, ST_MAN >
< Kochhar, AD_VP >
< Kochhar, AD_VP >
```

```
< Raphaely, PU_MAN >
< Taylor, SA_REP >
< Taylor, <NULL> >
< Whalen, AD_ASST >
< Whalen, <NULL> >
10 rows 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'.

Examples

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'.

Examples

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, which can be a CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, NCLOB, or BLOB expression.
Expression2	The alternate value to use if the value of *Expression1* is NULL, which can be a CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, NCLOB, or BLOB expression.

Description

- The data types of *Expression1* and *Expression2* must be compatible. If the data types are different, the data types are implicitly converted, if possible. If they cannot be implicitly converted, an error is returned.

  The following describes how the implicit conversion of data types is performed:

  - If *Expression1* is character data, then *Expression2* is converted to the same data type of *Expression1* and returns the result in a VARCHAR2 in the character set of *Expression1*.
  - If *Expression1* is numeric data, then TimesTen determines which expression has the highest numeric precedence and implicitly converts the other argument to that data type, which is also the data type that is returned.
- 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.

Examples

This example checks for null values of commission_pct and replaces them with 'Not Applicable' for employees whose last names start with "B".

```
Command> SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable')
       FROM employees
       WHERE last_name LIKE 'B%'
       ORDER BY last_name;

< Baer, Not Applicable >
< Baida, Not Applicable >
< Banda, .1 >
< Bates, .15 >
< Bell, Not Applicable >
< Bernstein, .25 >
< Bissot, Not Applicable >
< Bloom, .2 >
```

```
< Bull, Not Applicable >
9 rows found.
```

POWER

The POWER function returns *Base* raised to the *Exponent* power. The *Base* and *Exponent* can be any numbers, but if the *Base* is negative, the *Exponent* must be an integer.

SQL syntax

POWER (*Base*, *Exponent*)

Parameters

POWER has the parameters:

Parameter	Description
Base	Operand or column can be any numeric type. POWER returns this value raised to *Exponent* power.
Exponent	Operand or column can be any numeric type. If *Base* is negative, *Exponent* must be an integer.

Description

If either *Base* or *Exponent* is of type BINARY_FLOAT or BINARY_DOUBLE, the data type returned is BINARY_DOUBLE. If the *Base* is of type NUMBER and the *Exponent* is not of type BINARY_FLOAT or BINARY_DOUBLE, the date type returned is NUMBER with maximum precision and scale. If *Base* is one of the TT* numeric types (TT_BIGINT, TT_INTEGER, TT_SMALLINT, or TT_TINYINT), the data type returned is BINARY_DOUBLE.

Example

Use the POWER function to return the commission_pct squared for the employee with employee_id equal to 145.

```
Command> SELECT employee_id, commission_pct FROM employees
       WHERE employee_id = 145;

< 145, .4 >
1 row found.

Command> SELECT POWER (commission_pct,2) FROM employees
       WHERE employee_id = 145;
< .16 >
1 row found.
```

RANK

The RANK function is an analytic function that calculates the rank of a value in a group of values.

SQL syntax

RANK () OVER ([*QueryPartitionClause*] *OrderByClause*)

Parameters

RANK has the parameters:

Parameter	Description
QueryPartitionClause	See "Analytic Functions" for information on syntax, semantics, and restrictions.
OrderByClause	See "Analytic Functions" for information on syntax, semantics, and restrictions.

Description

- The return type is NUMBER.
- Rows with equal values for the ranking criteria receive the same rank. TimesTen then adds the number of tied rows to the ties rank to calculate the next rank. Therefore, the ranks may not be consecutive numbers.
- RANK computes the rank of each row returned from a query with respect to the other rows returned by the query, based on the values of the expressions in the *OrderByClause*.

Example

Use the RANK function to rank the first 10 employees in department 80 based on their salary and commission. Identical salary values receive the same rank and cause nonconsecutive ranks.

```
Command> SELECT first 10 department_id, last_name, salary, commission_pct,
          RANK() OVER (PARTITION BY department_id
          ORDER BY salary DESC, commission_pct) "Rank"
         FROM employees WHERE department_id = 80
         ORDER BY department_id, last_name, salary, commission_pct, "Rank";
< 80, Abel, 11000, .3, 5 >
< 80, Ande, 6400, .1, 31 >
< 80, Banda, 6200, .1, 32 >
< 80, Bates, 7300, .15, 26 >
< 80, Bernstein, 9500, .25, 14 >
< 80, Bloom, 10000, .2, 9 >
< 80, Cambrault, 7500, .2, 23 >
< 80, Cambrault, 11000, .3, 5
< 80, Doran, 7500, .3, 24 >
< 80, Errazuriz, 12000, .3, 3 >
10 rows found.
```

REPLACE

REPLACE substitutes a sequence of characters in a given string with another set of characters or removes the string entirely.

SQL syntax

REPLACE (*String*, *SearchString* [,*ReplacementString*])

Parameters

REPLACE has the parameters:

Parameter	Description
String	Source string containing the substring to replace.
SearchString	String of characters to be replaced in the original string. If *SearchString* is NULL, the original *String* is returned without any modification.
ReplacementString	String of characters that are used to replace all occurrences of the search string in the original string. If *ReplacementString* is omitted or NULL, all occurrences of *SearchString* are removed from the source *String*.

Description

- REPLACE returns a string where every occurrence of the *SearchString* is replaced with *ReplacementString* in the source *String*.
- *String*, *SearchString* and *ReplacementString* can be any of the following data types: CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. Both TimesTen and Oracle Database data types are supported. All non-character data types, except for BLOB, are implicitly converted to a string data type.
- If *String* is a CHAR or VARCHAR2, the returned string is of data type VARCHAR2. If *String* is an NCHAR or NVARCHAR2, the returned string is of data type NVARCHAR2. For CLOB or NCLOB data types, the data type returned is the same as the data type provided in *String*. The character set is the same as the source *String*.
- If the returned string length is zero, NULL is returned for Oracle Database data types and a zero length string is returned for TimesTen data types. See Data Types for details on all data types.

Examples

The following prints out all locations in Canada, replacing the country code of CA with Canada for easier readability.

```
Command> SELECT location_id, street_address,
         city, state_province, postal_code,
        REPLACE(country_id, 'CA', 'Canada')
        FROM LOCATIONS
        WHERE country_id LIKE 'CA';

< 1800, 147 Spadina Ave, Toronto, Ontario, M5V 2L7, Canada >
< 1900, 6092 Boxwood St, Whitehorse, Yukon, YSW 9T2, Canada >
2 rows found.
```

ROUND (Date)

Returns date rounded 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 rounded to the nearest day.

SQL syntax

```
ROUND (Date [,Fmt])
```

Parameters

ROUND (*Date*) has the parameters:

Parameter	Description
Date	The date that is rounded. Must resolve to a date value. If you do not specify *fmt*, then *date* is rounded to the nearest day.
[,*Fmt*]	The format model rounding unit. Specify either a constant or a parameter for *fmt*.

Description

- Date can be of type DATE or TIMESTAMP. The data type returned is DATE.
- Data types TT_DATE and TT_TIMESTAMP are not supported.
- For the supported format models to use in *fmt*, see "Format Model for ROUND and TRUNC Date Functions".

Examples

Round *Date* to the first day of the following year by specifying 'YEAR' as the format model:

```
Command> SELECT ROUND (DATE '2007-08-25','YEAR') FROM dual;
< 2008-01-01 00:00:00 >
1 row found.
```

Omit *Fmt*. Specify *Date* as type TIMESTAMP with a time of 13:00:00. *Date* is rounded to nearest day:

```
Command> SELECT ROUND (TIMESTAMP '2007-08-16 13:00:00') FROM dual;
< 2007-08-17 00:00:00 >
1 row found.
```

ROUND (Expression)

The ROUND function returns *Expression1* rounded to *Expression2* places to the right of the decimal point.

SQL syntax

ROUND (*Expression1* [,*Expression2*])

Parameters

ROUND has the parameters:

Parameter	Description
Expression1	Operand or column can be any numeric type.
Expression2	Operand or column that indicates how many places to round. Can be negative to round off digits left of the decimal point. If you omit *Expression2*, then *Expression1* is rounded to 0 places. Must be an integer.

Description

- If you omit *Expression2*, the data type returned is the same as the numeric data type of *Expression1*.
- If you specify *Expression2*, the data type returned is NUMBER with maximum precision and scale.

- If *Expression1* is of type BINARY_FLOAT or BINARY_DOUBLE, the value of *Expression1* is rounded to the nearest even value. Otherwise, the value of *Expression1* is rounded away from 0 (for example, to *x*+1 when *x*.5 is positive and to *x*-1 when *x*.5 is negative).

Examples

Round a number two places to the right of the decimal point.

```
Command> SELECT ROUND (15.5555,2) FROM dual;
< 15.56 >
1 row found.
```

Round a number to the left of the decimal point by specifying a negative number for *Expression2*.

```
Command> SELECT ROUND (15.5555,-1) FROM dual;
< 20 >
1 row found.
```

Round a floating point number. Floating point numbers are rounded to nearest even value. Contrast this to rounding an expression of type NUMBER where the value is rounded up (for positive values).

```
Command> SELECT ROUND (1.5f), ROUND (2.5f) FROM dual;
< 2.00000000000000, 2.00000000000000 >
1 row found.
Command> SELECT ROUND (1.5), ROUND (2.5) FROM dual;
< 2, 3 >
1 row found.
```

ROW_NUMBER

The ROW_NUMBER function is an analytic function that assigns a unique number to each row to which it is applied (either each row in a partition or each row returned by the query), in the ordered sequence of rows specified in *OrderByClause*, beginning with 1.

SQL syntax

```
ROW_NUMBER () OVER ( [QueryPartitionClause] OrderByClause )
```

Parameters

ROW_NUMBER has the parameters:

Parameter	Description
QueryPartitionClause	See "Analytic Functions" for information on syntax, semantics, and restrictions.
OrderByClause	See "Analytic Functions" for information on syntax, semantics, and restrictions.

Description

- ROWNUM pseudo column returns a number indicating the order in which TimesTen selects a row from a table or a set of joined rows. In contrast, the analytic function, ROW_NUMBER, gives superior support in ordering the results of a query before assigning the number.
- By nesting a subquery, using ROW_NUMBER, inside a query that retrieves the ROW_NUMBER values for a specified range, you can find a precise subset or rows from the results of the inner query. For consistent results, the query must ensure a deterministic sort order.

- The return data type is NUMBER.

Example

Use ROW_NUMBER to return the three highest paid employees in each department. Fewer then three rows are returned for departments with fewer than three employees.

```
Command> SELECT FIRST 10 department_id, first_name, last_name, salary
       FROM
        (SELECT department_id, first_name, last_name, salary, ROW_NUMBER()
          OVER (PARTITION BY department_id ORDER BY salary desc) rn
         FROM employees )
       WHERE rn <= 3
       ORDER BY department_id, salary DESC, last_name;
< 10, Jennifer, Whalen, 4400 >
< 20, Michael, Hartstein, 13000 >
< 20, Pat, Fay, 6000 >
< 30, Den, Raphaely, 11000 >
< 30, Alexander, Khoo, 3100 >
< 30, Shelli, Baida, 2900 >
< 40, Susan, Mavris, 6500 >
< 50, Adam, Fripp, 8200 >
< 50, Matthew, Weiss, 8000 >
< 50, Payam, Kaufling, 7900 >
10 rows found.
```

RPAD

The RPAD function returns *Expression1*, right-padded to length *n* characters with *Expression2*, replicated as many times as necessary. This function is useful for formatting the output of a query.

SQL syntax

RPAD (*Expression1*, *n* [,*Expression2*])

Parameters

RPAD has the parameters:

Parameter	Description
Expression1	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column to be right-padded. If *Expression1* is longer than *n*, then RPAD returns the portion of *Expression1* that fits in *n*.
n	Length of characters returned by RPAD function. Must be a NUMBER integer or a value that can be implicitly converted to a NUMBER integer.
Expression2	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column to be right-padded to *Expression1*. If you do not specify *Expression2*, the default is a single blank.

Description

- If *Expression1* is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If *Expression1* is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. If *Expression1* is a LOB, the data type returned is the same as the LOB data type provided.

- The returned data type length is equal to *n* if *n* is a constant. Otherwise, the maximum result length of 8300 is returned.
- You can specify TT_CHAR, TT_VARCHAR, TT_NCHAR, and TT_NVARCHAR for *Expression1* and *Expression2*. If *Expression1* is of type TT_CHAR or TT_VARCHAR, the data type returned is TT_VARCHAR. If *Expression1* is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR.
- For CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB data types:
  - If either *Expression1* or *Expression2* is NULL, the result is NULL. If *n* is less than or equal to 0, the result is NULL.
- For TT_CHAR, TT_VARCHAR, TT_NCHAR and TT_NVARCHAR types:
  - If either *Expression1* or *Expression2* is not NULL and if *n* is less than or equal to 0, the result is the empty string.

Examples

Concatenate first_name and last_name from the employees table. Call the RPAD function to return first_name right-padded to length 12 with spaces and call RPAD a second time to return last_name right-padded to length 12 with spaces. Select first five rows.

```
Command> SELECT FIRST 5 CONCAT (RPAD (first_name,12),
         RPAD (last_name,12)) FROM employees
           ORDER BY first_name, last_name;
< Adam        Fripp       >
< Alana       Walsh       >
< Alberto     Errazuriz   >
< Alexander   Hunold      >
< Alexander   Khoo        >
5 rows found.
```

Call the RPAD function to return last_name right-padded to length 20 characters with the dot ('.') character. Use the employees table and select first five rows.

```
Command> SELECT FIRST 5 RPAD (last_name,20,'.') FROM employees
         ORDER BY last_name;
< Abel................ >
< Ande................ >
< Atkinson............ >
< Austin.............. >
< Baer................ >
5 rows found.
```

RTRIM

The RTRIM function removes from the right end of *Expression1* all of the characters contained in *Expression2*. TimesTen scans *Expression1* backward from its last character and removes all characters that appear in *Expression2* until reaching a character not in *Expression2* and then returns the result.

SQL syntax

RTRIM (*Expression1* [,*Expression2*])

Parameters

RTRIM has the parameters:

Parameter	Description
Expression1	The CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column to be trimmed. If *Expression1* is a character literal, then enclose it in quotes.
Expression2	Optional expression used for trimming *Expression1*. If *Expression2* is a character literal, enclose it in single quotes. If you do not specify *Expression2*, it defaults to a single blank. Operand or column can be of type CHAR,VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB.

Description

- If *Expression1* is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If *Expression1* is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. If *Expression1* is a CLOB or NCLOB, the data type returned is the same as the LOB data type provided. The returned data type length is equal to the data type length of *Expression1*.
- If *Expression1* is a data type defined with CHAR length semantics, the returned length is expressed in CHAR length semantics.
- If either *Expression1* or *Expression2* is NULL, the result is NULL.
- You can specify TT_CHAR, TT_VARCHAR, TT_NCHAR, and TT_NVARCHAR for *Expression1* and *Expression2*. If *Expression1* is of type TT_CHAR or TT_VARCHAR, the data type returned is TT_VARCHAR. If *Expression1* is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR.
- If *Expression1* is of type CHAR or VARCHAR2 and *Expression2* is of type NCHAR or NVARCHAR2, then *Expression2* is demoted to CHAR or VARCHAR2 before RTRIM is invoked. The conversion of *Expression2* could be lost. If the trim character of *Expression2* is not in the database character set, then the query may produce unexpected results.
- For CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB and NCLOB types:
  - If all the characters in *Expression1* are removed by the RTRIM function, the result is NULL.
- For TT_CHAR, TT_VARCHAR, TT_NCHAR and TT_NVARCHAR types:
  - If all the characters in *Expression1* are removed by the RTRIM function, the result is the empty string.

Examples

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.
```

Call the RTRIM function to remove right-most 'x' and 'y' from string. RTRIM removes individual occurrences of 'x' and 'y', not pattern 'xy'.

```
Command> SELECT RTRIM ('RTRIM Examplexxxyyyxyxy', 'xy') FROM dual;
< RTRIM Example >
1 row found.
```

Call RTRIM to remove all characters from *Expression1*. In the first example, the data type is CHAR, so NULL is returned. In the second example, the data type is TT_CHAR, so the empty string is returned.

```
Command> CREATE TABLE rtrimtest (col1 CHAR (4), col2 TT_CHAR (4));
Command> INSERT INTO rtrimtest VALUES ('BBBA', 'BBBA');
1 row inserted.
Command> SELECT RTRIM (col1, 'AB') FROM rtrimtest;
< <NULL> >
1 row found.
Command> SELECT RTRIM (col2, 'AB') FROM rtrimtest;
<  >
1 row found.
```

SESSION_USER

Returns the name of the TimesTen user currently connected to the database.

SQL syntax

```
SESSION_USER
```

Parameters

SESSION_USER has no parameters.

Examples

To return the name of the session user:

```
SELECT SESSION_USER FROM dual;
```

SIGN

The SIGN function returns the sign of *Expression*.

SQL syntax

SIGN (*Expression*)

Parameters

SIGN has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER with maximum precision and scale. Otherwise, the data type returned is TT_INTEGER.

  For numeric types that are not binary floating-point numbers, the sign is:

  – -1 if the value of *Expression* is <0

  – 0 if the value of *Expression* is = 0

  – 1 if the value of *Expression* is > 0

- For binary floating-point numbers (BINARY_FLOAT and BINARY_DOUBLE), this function returns the sign bit of the number. The sign bit is:
  - -1 if the value of *Expression* is <0
  - +1 if the value of *Expression* is >= 0 or the value of *Expression* is equal to NaN

Examples

These examples illustrate use of the SIGN function with different data types. Table signex has been created and the columns have been defined with different data types. First, describe the table signex to see the data types of the columns. Then select each column to retrieve values for that column. Use the SIGN function to return the sign for the column.

```
Command> DESCRIBE signex;

Table SAMPLEUSER.SIGNEX:
  Columns:
    COL1                    TT_INTEGER
    COL2                    TT_BIGINT
    COL3                    BINARY_FLOAT
    COL4                    NUMBER (3,2)

1 table found.
(primary key columns are indicated with *)
Command> SELECT col1 FROM signex;
< 10 >
< -10 >
< 0 >
3 rows found.
Command> SELECT SIGN (col1) FROM signex;
< 1 >
< -1 >
< 0 >
3 rows found.
Command> SELECT col2 FROM signex;
< 0 >
< -3 >
< 0 >
3 rows found.
Command> SELECT SIGN (col2) FROM signex;
< 0 >
< -1 >
< 0 >
3 rows found.
Command> SELECT col3 FROM signex;
< 3.500000 >
< -3.560000 >
< NAN >
3 rows found.
Command> SELECT SIGN (col3) FROM signex;
< 1 >
< -1 >
< 1 >
3 rows found.
Command> SELECT col4 FROM signex;
< 2.2 >
< -2.2 >
< 0 >
3 rows found.
Command> SELECT SIGN (col4) FROM signex;
< 1 >
```

```
< -1 >
< 0 >
3 rows found.
```

SIN

The SIN function returns the sine of *Expression* (an angle expressed in radians).

SQL syntax

```
SIN(Expression)
```

Parameters

SIN has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the SIN function to return the sine of 30 degrees.

```
Command> SELECT SIN(30 * 3.14159265359/180) FROM dual;
< .50000000000000298434573127255848979959561 >
1 row found.
```

SINH

The SINH function returns the hyperbolic sine of *Expression*.

SQL syntax

```
SINH(Expression)
```

Parameters

SINH has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.

- If the value of *Expression* is NULL, NULL is returned.

Example

Use the SINH function to return the hyperbolic sine of 1.

```
Command> SELECT SINH(1) "Hyperbolic sine of 1" FROM dual;
< 1.17520119364380145688238185059560081516 >
1 row found.
```

SOUNDEX

The SOUNDEX function determines a phonetic signature for a string and allows comparisons of strings based on phonetic similarity. SOUNDEX lets you compare words that are spelled differently, but sound alike in English.

SQL syntax

SOUNDEX (*InputString*)

Parameters

SOUNDEX has the parameters:

Parameter	Description
InputString	Valid types are CHAR, VARCHAR2, NCHAR and NVARCHAR2 with both ORA and TT variants and CLOB and NCLOB. If the data type is CLOB or NCLOB, TimesTen performs implicit conversion before returning the result.

Description

- Converts an alpha-numeric string into a 4 character code, beginning with the first letter encountered in the string, followed by three numbers.
- The phonetic representation is defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:
  1. Retain the first letter of the string and drop all other occurrences of the following letters: A, E, I, O, U. The treatment of the letters is case insensitive.
  2. Drop all occurrences of H, W, and Y.
  3. Assign numbers to the remaining letters (after the first) as follows:

     ```
     B, F, P, V = 1
     C, G, J, K, Q, S, X, Z = 2
     D, T = 3
     L = 4
     M, N = 5
     R = 6
     ```
  4. If two or more letters with the same number were adjacent in the original name (before step 1), omit all but the first.
  5. Return the first four characters of the result (padded with '0' if the result has less than four characters).
- The function returns NULL if no soundex code could be generated for the *InputString*. For example, NULL is returned when the *InputString* contains no English letters.
- The input to output type mapping is:

Input Type	Output Type
VARCHAR2(*x*), CHAR, CLOB	VARCHAR2(4)
TT_CHAR(*x*), TT_VARCHAR(*x*)	TT_VARCHAR(4)
NVARCHAR2(*x*), NCHAR(*x*), NCLOB	NVARCHAR2(4)
TT_NCHAR(*x*), TT_NVARCHAR(*x*)	TT_NVARCHAR(4)

Examples

Use SOUNDEX function to return the phonetic signature for employees with last name equal to 'Taylor'.

```
Command> SELECT last_name, first_name, SOUNDEX (last_name)
       FROM employees where last_name = 'Taylor';
< Taylor, Jonathon, T460 >
< Taylor, Winston, T460 >
2 rows found.
```

Invoke the function again to return the phonetic signature for the string 'Tailor'. Invoke the function a third time to return the last name and first name of each employee whose last name is phonetically similar to the string 'Tailor'.

```
Command> SELECT SOUNDEX ('Tailor') FROM dual;
< T460 >
1 row found.

Command> SELECT last_name, first_name FROM employees WHERE SOUNDEX (last_name) =
         SOUNDEX ('Tailor');
< Taylor, Jonathon >
< Taylor, Winston >
2 rows found.
```

SQRT

The SQRT function returns the square root of *Expression*.

SQL syntax

SQRT(*Expression*)

Parameters

SQRT has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER with maximum precision and scale. If *Expression* is of type BINARY_FLOAT, the data type returned is BINARY_FLOAT. Otherwise, the data type returned is BINARY_DOUBLE.
- If *Expression* is of type NUMBER, the value of *Expression* cannot be negative.
- If *Expression* resolves to a binary floating-point number (BINARY_FLOAT or BINARY_DOUBLE):

- If the value of the *Expression* is > = 0, the result is positive.
- If the value of the *Expression* is = -0, the result is -0.
- If the value of the *Expression* is < 0, the result is NaN.

Examples

Use SQRT function to return the square root of the absolute value of -10. Then cast the value as BINARY_FLOAT.

```
Command> SELECT CAST (SQRT (ABS (-10)) AS BINARY_FLOAT ) FROM dual;
< 3.162278 >
1 row found.
```

SUBSTR, SUBSTRB, SUBSTR4

Returns a string that represents a substring of a source 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}=(*Source*, *m*, *n*)

Parameters

SUBSTR has the parameters:

Parameter	Description
Source	The string for which this function returns a substring. Value can be any supported character data types including CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB data types. Both TimesTen and Oracle Database data types are supported. If *Source* is a CHAR string, the result is a CHAR or VARCHAR2 string. If *Source* is a NCHAR string, the result is a NVARCHAR2 string. If *Source* is a LOB, the result is the same LOB data type.
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.

Examples

In the first five rows of employees, select the first three characters of last_name:

```
SELECT FIRST 5 SUBSTR(last_name,1,3) FROM employees;
< Kin >
< Koc >
< De  >
```

```
< Hun >
< Ern >
5 rows found.
```

In the first five rows of employees, select the last five characters of last_name:

```
SELECT FIRST 5 SUBSTR(last_name,-5,5) FROM employees;
< <NULL> >
< chhar >
<  Haan >
< unold >
< Ernst >
5 rows found.
```

SUM

Finds the total of all values in the argument. Null values are ignored. SUM is an aggregate function. SUM can also be an aggregate analytic function. See "Aggregate Functions" for more details on aggregate functions. See "Analytic Functions" for more information on analytic functions.

SQL syntax

SUM ([ALL | DISTINCT] *Expression*) [OVER ([*AnalyticClause*])]

Parameters

SUM has the parameters:

Parameter	Description
Expression	Can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type.
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.
OVER ([*AnalyticClause*])	If specified, indicates aggregate analytic function. See "Analytic Functions" for more information on analytic functions.

Description

- If the SUM function is computed over an empty table in which GROUP BY is not used, SUM returns NULL.
- If the SUM function is computed over an empty group or an empty grouped table (GROUP BY is used), SUM returns nothing.
- If the source is TT_TINYINT, TT_SMALLINT, or TT_INTEGER, the result data type is TT_BIGINT.
- If the source is NUMBER, then the result data type is NUMBER with undefined scale and precision.
- For all other data types, the result data type is the same as the source.
- If you do not use the *AnalyticClause* in your query, then SUM acts as an aggregate function.
- If you specify DISTINCT and the *AnalyticClause*, then you can only specify the *QueryPartitionClause*. The *OrderByClause* and *WindowingClause* are not allowed.

Examples

Sum all employee salaries:

```
Command> SELECT SUM(salary) Total FROM employees;

TOTAL
< 691400 >
1 row found.
```

SYS_CONTEXT

Returns information about the current session.

The data type of the return value is VARCHAR2.

SQL syntax

```
SYS_CONTEXT('namespace', 'parameter' [, length ])
```

Parameters

SYS_CONTEXT has the parameters:

Parameter	Description
namespace	Value: USERENV Other values result in a return of NULL.
parameter	Supported values: • ACTION • AUTHENTICATION_METHOD • CLIENT_INFO • CURRENT_SCHEMA • CURRENT_USER • CURRENT_USERID • IDENTIFICATION_TYPE • LANG • LANGUAGE • MODULE • NLS_SORT • SESSION_USER • SESSION_USERID • SID
length	Length in bytes, from 1 to 4000.

These are descriptions of the supported values for *parameter*:

Parameter	Description
ACTION	Identifies the position in the module (application name) and is set through OCI.
AUTHENTICATION_METHOD	Returns the method of authentication for these types of users: • Local database user authenticated by password • External user authenticated by the operating system

Parameter	Description
CLIENT_INFO	Returns the user session information that can be stored by an application through OCI.
CURRENT_SCHEMA	The name of the currently active database schema. This may change during the duration of a session to reflect the owner of any active definer's rights object. When used directly in the body of a view definition, this returns the default schema used when executing the SQL statement that is using the view. It does not respect views used in the SQL statement as having definer's rights. SYS_CONTEXT returns the same value when the CURRENT_SCHEMA parameter is supplied as it does when the CURRENT_USER parameter is supplied.
CURRENT_USER	The name of the database user whose privileges are currently active. This may change during the duration of a session to reflect the owner of any active definer's rights object. When no definer's rights object is active, CURRENT_USER returns the same value as SESSION_USER. When used directly in the body of a view definition, this returns the user that is executing the SQL statement that is using the view. It does not respect views used in the SQL statement as having definer's rights.
CURRENT_USERID	The identifier of the database user whose privileges are currently active.
IDENTIFICATION_TYPE	Returns the way the user was created in the database. Specifically, it reflects the IDENTIFIED clause in the CREATE/ALTER USER syntax. In the list that follows, the syntax used during user creation is followed by the identification type returned: • IDENTIFIED BY *password*: LOCAL • IDENTIFIED EXTERNALLY: EXTERNAL
LANG	The ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	The language and territory currently used by the session, along with the database character set, in this form: *language_territory.characterset*
MODULE	The application name (module) set through OCI.
NLS_SORT	Binary or linguistic sort.
SESSION_USER	The name of the database user at logon. This value remains the same throughout the duration of the session.
SESSION_USERID	The identifier of the database user at logon.
SID	The connection ID of the current connection.

Description

The data type of the return value is VARCHAR2.

Examples

```
SELECT SYS_CONTEXT('USERENV', 'CURRENT_USER') FROM dual;
< TERRY >
1 row found.

SELECT SYS_CONTEXT('USERENV', 'LANGUAGE') FROM dual;
< AMERICAN_AMERICA.AL32UTF8 >
1 row found.
```

```
SELECT SYS_CONTEXT('USERENV', 'IDENTIFICATION_TYPE') FROM dual;
< EXTERNAL >
1 row found.
```

SYSDATE and GETDATE

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

SQL syntax

```
SYSDATE | GETDATE( )
```

Parameters

The SYSDATE and GETDATE functions have no parameters.

Description

- SYSDATE and GETDATE perform identically. SYSDATE is compatible with Oracle Database syntax. 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:MI: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 INSERT...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.

Examples

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.
```

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.
```

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

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

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.
```

This 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.
```

SYSTEM_USER

Returns the name of the current database user as identified by the operating system.

SQL syntax

```
SYSTEM_USER
```

Parameters

SYSTEM_USER has no parameters.

Examples

To return the name of the operating system user:

```
SELECT SYSTEM_USER FROM dual;
```

TAN

The TAN function returns the tangent of *Expression* (an angle expressed in radians).

SQL syntax

```
TAN(Expression)
```

Parameters

TAN has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the TAN function to return the tangent of 135 degrees.

```
Command> SELECT TAN(135 * 3.14159265359/180) "Tangent of 135 degrees" FROM dual;
< -.99999999999996898576939651230133793225994 >
1 row found.
```

TANH

The TANH function returns the hyperbolic tangent of *Expression*.

SQL syntax

TANH(*Expression*)

Parameters

TANH has the parameter:

Parameter	Description
Expression	Operand or column can be any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type.

Description

- If *Expression* is of type NUMBER, the data type returned is NUMBER. For all other numeric data types, the data type returned is BINARY_DOUBLE.
- If the value of *Expression* is NULL, NULL is returned.

Example

Use the TANH function to return the hyperbolic tangent of .5.

```
Command> SELECT TANH(.5) "Hyperbolic tangent of .5" FROM dual;
< .46211715726000975850231848364367254872 1 >
1 row found.
```

TIMESTAMPADD

The TIMESTAMPADD function adds a specified number of intervals to a timestamp and returns the modified timestamp.

SQL syntax

TIMESTAMPADD (*Interval*, *IntegerExpression*, *TimestampExpression*)

Parameters

TIMESTAMPADD has the parameters:

Parameter	Description
Interval	Specified interval. Must be expressed as literal. Valid values are listed in the description section.

Parameter	Description
IntegerExpression	Expression that evaluates to TT_BIGINT.
TimestampExpression	Datetime expressions. Valid data types are ORA_DATE, ORA_TIMESTAMP, TT_DATE, and TT_TIMESTAMP. (The alias DATE and TIMESTAMP data types are also valid). TT_TIME is not supported.

Description

- Valid values for *Interval* are:
  - SQL_TSI_FRAC_SECOND
  - SQL_TSI_SECOND
  - SQL_TSI_MINUTE
  - SQL_TSI_HOUR
  - SQL_TSI_DAY
  - SQL_TSI_WEEK
  - SQL_TSI_MONTH
  - SQL_TSI_QUARTER
  - SQL_TSI_YEAR
- SQL_TSI_FRAC_SECOND is expressed in billionths of a second.
- The return type is the same as the original data type. For example, if your expression is of type TIMESTAMP, then the resulting data type is TIMESTAMP. Only positive timestamp expressions (0001-01-01) are allowed both in the query and the result. For TT_DATE and TT_TIMESTAMP, because the starting range for these data types is 1753-01-01, the timestamp expression must be equal to or greater than this date.
- If *IntegerExpression* or *TimestampExpression* is NULL, then the result is NULL.
- The function computes the total time interval as a product of the *IntegerExpression* and the interval and adds it to the specified *TimestampExpression*. Adding a year advances the timestamp by 12 months and adding a week advances the timestamp by seven days. If the *IntegerExpression* is negative, the specified interval is subtracted from the *TimestampExpression*.
- There is a possibility of precision loss depending on your use of the specified interval and timestamp expression. For example, if your interval is SQL_TSI_HOUR, and you specify 2 for *IntegerExpression* and TT_DATE for *TimestampExpression*, TimesTen treats the two hours as zero days and returns the sum of the original date plus zero days resulting in some loss of precision. If however, your *IntegerExpression* is 48, TimesTen treats the 48 hours as two days and returns the sum of the original date plus two days. In this case, there is no loss of precision.
- If the addition of the timestamp results in an overflow of the specified component (such as more than 60 seconds, or more than 24 hours, or more than 12 months), then the overflow is carried over to the next component. For example, if the seconds component overflows, then the minutes component is advanced.

Examples

Use the TIMESTAMPADD function to add 3 months to timestamp '2009-11-30 10:00:00'. TimesTen increments the year and adjusts the day component to accommodate the 28 days in the month of February.

```
Command> SELECT TIMESTAMPADD (SQL_TSI_MONTH, 3, TIMESTAMP '2010-11-30 10:00:00') FROM dual;
< 2011-02-28 10:00:00 >
1 row found.
```

Use the TIMESTAMPADD function to add 1 second to timestamp '2010-12-31 23:59:59'. TimesTen propagates the overflow through all components of the timestamp and advances the components appropriately.

```
Command> SELECT TIMESTAMPADD (SQL_TSI_SECOND, 1, TIMESTAMP '2010-12-31 23:59:59' FROM dual;
< 2011-01-01 00:00:00 >
1 row found.
```

TIMESTAMPDIFF

The TIMESTAMPDIFF function returns the total number of specified intervals between two timestamps.

SQL syntax

TIMESTAMPDIFF (*Interval*, *TimestampExpression1*, *TimestampExpression2*)

Parameters

TIMESTAMPDIFF has the parameters:

Parameter	Description
Interval	Specified interval. Must be expressed as literal. Valid values are listed in the description section.
TimestampExpression1	Datetime expressions. Valid data types are ORA_DATE, ORA_TIMESTAMP, TT_DATE, and TT_TIMESTAMP. (The alias DATE and TIMESTAMP data types are also valid.) TT_TIME is not supported.
TimestampExpression2	Datetime expressions. Valid data types are ORA_DATE, ORA_TIMESTAMP, TT_DATE, and TT_TIMESTAMP. (The alias DATE and TIMESTAMP data types are also valid.) TT_TIME is not supported.

Description

- Valid values for *Interval* are:
  - SQL_TSI_FRAC_SECOND
  - SQL_TSI_SECOND
  - SQL_TSI_MINUTE
  - SQL_TSI_HOUR
  - SQL_TSI_DAY
  - SQL_TSI_WEEK
  - SQL_TSI_MONTH
  - SQL_TSI_QUARTER
  - SQL_TSI_YEAR
- SQL_TSI_FRAC_SECOND is expressed in billionths of a second.
- *Interval* determines the units in which the difference in timestamps is returned. For example, if you specify SQL_TSI_YEAR, the difference in timestamps is returned in years.

- TimesTen returns the result as the difference between *TimestampExpression2* minus (-) *TimestampExpression1*. The return type is TT_BIGINT.
- Only positive timestamp expressions (0001-01-01) are allowed. For TT_DATE and TT_TIMESTAMP, because the starting range for these data types is 1753-01-01, the timestamp expression must be equal to or greater than this date.
- If *TimestampExpression1* or *TimestampExpression2* is NULL, then the result is NULL.
- If either timestamp expression is a date value and *Interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of the timestamp is set to 0 before TimesTen calculates the difference between the timestamps.
- The function first expresses each of the timestamps in units of the specified *Interval* by converting the higher order interval type to the specified interval type. For example, TimesTen converts years to months if the specified interval is months. Thus, one year is 12 months, one week is seven days, and so on. To find the number of days between two timestamps, the exact number of days is computed. Since months vary in the number of days, TimesTen does not make an assumption about the number of days in a month.
- The function increments the specified interval whenever fractional intervals cross an interval boundary. For example, the difference in years between 2009-12-31 and 2010-01-01 is one year because the fractional year represents a crossing from one year to the next (2009 to 2010). However, the difference between 2010-01-01 and 2010-12-31 is zero years because the fractional interval does not cross a boundary. It falls within the year 2010.
- The function calculates the difference in weeks by first calculating the difference in days and then divides the result by seven before rounding. TimesTen assumes a week starts on a Sunday. Therefore the difference in weeks between 2010-10-21 (a Thursday) and 2010-10-25 (the following Monday) results in a value of one week. The difference in the same dates, if Tuesday denoted the start of the week, would result in zero weeks.

Examples

Use the TIMESTAMPDIFF function to calculate the difference in days between dates 2008-02-01 and 2008-03-01. Because 2008 is a leap year, the result is 29 days. The calculation is precise with no assumption of a 30-day month.

```
Command> SELECT TIMESTAMPDIFF (SQL_TSI_DAY, DATE '2008-02-01', DATE '2008-03-01') FROM dual;
< 29 >
1 row found.
```

Use the TIMESTAMPDIFF function to calculate the difference in months between dates 2009-02-01 and 2009-03-01. Because there is a crossing of the interval month boundary, the function returns 1. In the second example, because days is specified for the interval, the result is 28.

```
Command> SELECT TIMESTAMPDIFF (SQL_TSI_MONTH, DATE '2009-02-01', DATE '2009-03-01')
       FROM dual;
< 1 >
1 row found.

Command> SELECT TIMESTAMPDIFF (SQL_TSI_DAY, DATE '2009-02-01', DATE '2009-03-01')
       FROM dual;
< 28 >
1 row found.
```

Use the TIMESTAMPDIFF function to calculate the difference in months between dates 2009-02-01 and 2009-02-29. Because there is not a crossing of the interval month boundary, the function returns 0.

```
Command> SELECT TIMESTAMPDIFF (SQL_TSI_MONTH, DATE '2009-02-01', DATE '2009-02-28')
         FROM dual;
< 0 >
1 row found.
```

Use the TIMESTAMPDIFF function to illustrate the time difference in fractional seconds between mixed types. The time difference of one hour is returned in nanoseconds (unit for fractional seconds). The time element of the data type is set to 00:00:00.

```
Command> SELECT TIMESTAMPDIFF (SQL_TSI_FRAC_SECOND, TT_TIMESTAMP '2009-12-31 01:00:00.00',
         DATE '2009-12-31') FROM dual;
< -3600000000000 >
1 row found.
```

TO_BLOB

The TO_BLOB function converts VARBINARY or BINARY to a BLOB:

This function is not supported in TimesTen Scaleout.

SQL syntax

```
TO_BLOB ( ValidDataType )
```

Parameters

TO_BLOB has the parameters:

Parameter	Description
ValidDataType	A value that is of VARBINARY or BINARY data type.

Examples

The following example creates a table with a BINARY and a VARBINARY columns. The TO_BLOB function is used on the values of these columns to convert the BINARY and VARBINARY data to a BLOB.

```
Command> CREATE TABLE bvar(col1 BINARY (10), col2 VARBINARY (10));

Command> INSERT INTO bvar (col1, col2) VALUES (0x4D7953514C, 0x39274D);
1 row inserted.

Command> SELECT * FROM bvar;
 < 4D7953514C0000000000, 39274D >
1 row found.

Command> SELECT TO_BLOB(col1), TO_BLOB(col2) FROM bvar;
< 4D7953514C0000000000, 39274D >
1 row found.
```

TO_CHAR

The TO_CHAR function converts a DATE, TIMESTAMP or numeric input value to a VARCHAR2.

SQL syntax

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

Parameters

TO_CHAR has the parameters:

Parameter	Description
Expression1	A DATE, TIMESTAMP, CLOB, NCLOB, 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. See "Datetime Format Models" for information on the datetime format model used for TO_CHAR of data type DATE or TIMESTAMP. See "Format Model for ROUND and TRUNC Date Functions" for information on the datetime format model used for TO_CHAR of data type TT_DATE or TT_TIMESTAMP.
- TO_CHAR supports different number format models depending on the numeric data type specified for the expression. See "Number Format Models" for information on the number format model used for TO_CHAR of data type NUMBER or ORA_FLOAT. See "Format Model for ROUND and TRUNC Date Functions" for information on the number format model used for TO_CHAR of all other numeric data types.

Examples

```
SELECT FIRST 5 first_name,
       TO_CHAR (hire_date, 'MONTH DD, YYYY'),
       TO_CHAR (salary, '$999999.99')
  FROM employees;
< Steven, JUNE      17, 1987,   $24000.00 >
< Neena, SEPTEMBER 21, 1989,   $17000.00 >
< Lex, JANUARY   13, 1993,   $17000.00 >
< Alexander, JANUARY   03, 1990,    $9000.00 >
< Bruce, MAY       21, 1991,    $6000.00 >
5 rows found.

SELECT TO_CHAR(-0.12,'$B99.9999') FROM dual;
<   -$.1200 >
1 row found.

SELECT TO_CHAR(-12, 'B99999PR') FROM dual;
<     12  >
1 row found.

SELECT TO_CHAR(-12,'FM99999') FROM dual;
< -12 >
1 row found.

SELECT TO_CHAR(1234.1,'9,999.999') FROM dual;
<  1,234.100 >
1 row found.
```

TO_CLOB

The TO_CLOB function converts one of the following values to a CLOB:

- Character value contained in one of the following data types: CHAR, VARCHAR2, NVARCHAR2, TT_VARCHAR, TT_NVARCHAR, or NCLOB
- Datetime value contained in a DATE or TIMESTAMP data type
- Number value contained in a NUMBER, BINARY_FLOAT, or BINARY_DOUBLE data type

This function is not supported in TimesTen Scaleout.

SQL syntax

```
TO_CLOB ( ValidDataType )
```

Parameters

TO_CLOB has the parameters:

Parameter	Description
ValidDataType	A value of one of the valid data types mentioned above.

Description

The TO_CLOB function will not operate on values contained in INTERVAL or TIMESTAMP with TIMEZONE data types.

Examples

The following example uses the TO_CLOB function to convert a string.

```
Command> DESCRIBE clob_content;

Table USER1.CLOB_CONTENT:
  Columns:
   *ID                          NUMBER (38) NOT NULL
    CLOB_COLUMN                 CLOB NOT NULL

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

Command> INSERT INTO clob_content (id, clob_column) VALUES (3, EMPTY_CLOB());
1 row inserted.

Command> UPDATE clob_content
         SET clob_column = TO_CLOB('Demonstration of the TO_CLOB function.')
         WHERE id = 3;
1 row updated.
```

TO_DATE

The TO_DATE function converts a CHAR, VARCHAR2, CLOB, or NCLOB argument to a value of DATE data type.

SQL syntax

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

Parameters

TO_DATE has the parameters:

Parameter	Description
Expression1	A CHAR, VARCHAR2, CLOB, or NCLOB 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 HHMISS.
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. See "Datetime Format Models" for more information.

Examples

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

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

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 AM')
         FROM dual;
< DECEMBER  23, 1997 10:56-20 AM >
1 row found.

Command> SELECT TO_CHAR(TO_DATE('12-23-1997 15:56:20',
                     'MM-DD-YYYY HH24:MI:SS'),
                     'MONTH DD, YYYY HH24:MI-SS')
         FROM dual;
< DECEMBER  23, 1997 15:56-20 >
1 row found.
```

TO_TIMESTAMP

The TO_TIMESTAMP function converts a CHAR, VARCHAR2, CLOB, or NCLOB argument to a value of TIMESTAMP data type.

SQL syntax

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

Parameters

TO_TIMESTAMP has the parameters:

Parameter	Description
Expression1	A CHAR, VARCHAR2, CLOB, or NCLOB expression.

Parameter	Description
Expression2	The format string that specifies the format of *Expression1*. If you omit the format string (*Expression2*), then *Expression1* must be in the default format for the TIMESTAMP data type. The default format for the TIMESTAMP data type is SYYYY-MM-DD HH24:MI:SS:FF[9].
Expression3	A CHAR or VARCHAR2 expression to specify the NLS parameter. This value is currently ignored.

Description

- The TO_TIMESTAMP function converts a CHAR, VARCHAR2, CLOB, or NCLOB expression (passed to the function as *Expression1*) to a value of the TIMESTAMP data type. The return data type is TIMESTAMP.
- You can use a valid datetime format element for the format string in *Expression2*. See "Datetime Format Models" for more information.

Examples

Example 1: This example shows the return data type for the TO_TIMESTAMP function, which has the maximum fractional second precision of 9.

```
Command> describe SELECT TO_TIMESTAMP ('2021-05-07 10:11:12.123456') FROM dual;

Prepared Statement:
  Columns:
    EXP                     TIMESTAMP (9) NOT NULL
```

Example 2: This example throws an error when converting the character string to the TIMESTAMP data type. *Expression1* indicates a fractional second precision of 6 for the TIMESTAMP data type, but the format string (*Expression2*) indicates a value of 2 (FF2). The value cannot be truncated, resulting in a conversion error.

```
Command> SELECT TO_TIMESTAMP('2021-01-01 10:11:12.123456',
           'YYYY-MM-DD HH:MI:SS.FF2') FROM dual;
 2813: Error converting from character string '2021-01-01 10:11:12.123456'
       to timestamp
The command failed.
```

Example 3: These examples illustrate that the value of *n* for FF[*n*] should be large enough to accommodate the fractional seconds of *Expression1* (123456, in this example), such that there is no truncation. If you do not specify a value for *n*, as in the second example, the default is 9.

```
Command> SELECT TO_TIMESTAMP ('2021-05-07 10:10:10.123456',
        'YYYY-MM-DD HH:MI:SS.FF6') FROM dual;
< 2021-05-07 10:10:10.123456 >
1 row found.

Command> select to_timestamp('2021-05-07 10:10:10.123456',
        'YYYY-MM-DD HH:MI:SS.FF') FROM dual;
< 2021-05-07 10:10:10.123456000 >
1 row found.
```

Example 4: These examples show the result when *Expression1* is a character string and a format string is specified.

```
Command> SELECT TO_TIMESTAMP ('2021-05-07 10:10:10.123456',
        'YYYY-MM-DD HH:MI:SS.FF6') FROM dual;
```

```
< 2021-05-07 10:10:10.123456 >
1 row found.

Command> SELECT TO_TIMESTAMP ('2021-05-07 23:00:00.123456',
    'YYYY-MM-DD HH24:MI:SS.FF6') FROM dual;
< 2021-05-07 23:00:00.123456 >
1 row found.
```

Example 5: This example uses the FF format string. The FF format uses the maximum precision of 9 as shown in the result.

```
Command> SELECT TO_TIMESTAMP ('10-Sep-02 10:10:10.123000',
    'DD-Mon-RR HH12:MI:SS.FF') FROM dual;
< 2002-09-10 10:10:10.123000000 >
1 row found.
```

Example 6: These examples show the result when there is no format string. The default format is used for *Expression1*. Note that the fractional seconds precision of *Expression1* (123456789, in this example) is optional.

```
Command> SELECT TO_TIMESTAMP ('2021-MAY-07 101010.123456789') FROM dual;
< 2021-05-07 10:10:10.123456789 >
1 row found.

Command> SELECT TO_TIMESTAMP ('2021-MAY-07 101010') FROM dual;
< 2021-05-07 10:10:10.000000000 >
1 row found.

Command> SELECT TO_TIMESTAMP ('2021-MAY-07 101010.12') FROM dual;
< 2021-05-07 10:10:10.120000000 >
1 row found.
```

Example 7: This example illustrates the usage of the TO_CHAR function with the TO_TIMESTAMP to return the TIMESTAMP data type in an explicit timestamp format.

```
Command> SELECT TO_CHAR(TO_TIMESTAMP ('2021-05-07 13:11:12.123456',
    'YYYY-MM-DD HH24:MI:SS.FF6'),'DD/MON/YYYY HH24:MI:SS.FF6 PM') FROM dual;
< 07/MAY/2021 13:11:12.123456 PM >
1 row found.
```

Example 8: This example creates a table with a column of type TIMESTAMP(9). After describing the table, the example inserts one row without using the TO_TIMESTAMP function, and then inserts a second row using the TO_TIMESTAMP function. A SELECT on the table shows the results from the two insert operations.

```
Command> CREATE TABLE ts_table(col1 TIMESTAMP(9));
Command> describe ts_table;

Table SAMPLEUSER.TS_TABLE:
  Columns:
    COL1                    TIMESTAMP (9)

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

Command> INSERT INTO ts_table VALUES('2021-05-04 11:12:13.999999');
1 row inserted.
Command> INSERT INTO ts_table VALUES(to_timestamp('04-05-2021 11:12:13.123456789',
    'DD-MM-YYYY HH:MI:SS.FF9'));
1 row inserted.

Command> SELECT * FROM ts_table;
```

```
< 2021-05-04 11:12:13.999999000 >
< 2021-05-04 11:12:13.123456789 >
2 rows found.
```

Example 9: These examples illustrate the use of the AM and the PM format strings.

```
Command> SELECT TO_TIMESTAMP ('10-Sep-02 10:10:10.123456 PM',
       'DD-Mon-RR HH12:MI:SS.FF6 PM') FROM dual;
< 2002-09-10 22:10:10.123456 >
1 row found.

Command> SELECT TO_CHAR(TO_TIMESTAMP ('10-Sep-02 10:10:10.123456 PM',
       'DD-Mon-RR HH12:MI:SS.FF6 PM'),'DD-Mon-RR HH12:MI:SS.FF6 PM')
       FROM dual;
< 10-Sep-02 10:10:10.123456 PM >
1 row found.
```

The following example creates the ts_table2 table, defining col2 with the TIMESTAMP(9) data type. After describing the table, insert operations are done, illustrating the use of inserting data into a TIMESTAMP column using AM and PM.

```
Command> CREATE TABLE ts_table2 (col1 number primary key, col2 timestamp(9));
Command> describe ts_table2;

Table SAMPLEUSER.TS_TABLE2:
  Columns:
   *COL1                           NUMBER NOT NULL
    COL2                           TIMESTAMP (9)

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

Command> INSERT INTO ts_table2 VALUES (100,
       TO_TIMESTAMP('10-FEB-20 12.46.48.802050 PM',
       'DD-MON-RR HH:MI:SS.FF AM'));
1 row inserted.

Command> SELECT TO_CHAR(col2) FROM ts_table2;
< 2021-02-10 12:46:48.802050000 >
1 row found.

Command> SELECT TO_CHAR (col2, 'DD-MON-RR HH:MI:SS.FF AM') from ts_table2;
< 10-FEB-20 12:46:48.802050000 PM >
1 row found.
```

TO_LOB

The TO_LOB function converts supplied TT_VARCHAR and VARCHAR2 data types to a CLOB and VARBINARY data types to a BLOB.

This function is not supported in TimesTen Scaleout.

SQL syntax

TO_LOB (*ValidDataType*)

Parameters

TO_LOB has the parameters:

Parameter	Description
ValidDataType	A value that is of TT_VARCHAR, VARCHAR2, or BINARY data types.

Description

You can use this function only on a TT_VARCHAR, VARCHAR2, or VARBINARY column, and only with the CREATE TABLE AS SELECT or INSERT...SELECT statements on tables with a defined LOB column.

Examples

The following example shows how to use the TO_LOB function within the INSERT...SELECT statement on a table with a LOB column.

```
Command> CREATE TABLE clb(c CLOB);
Command> CREATE TABLE vc (v VARCHAR2(2000));
Command> INSERT INTO vc(v)
       VALUES ('Showing the functionality of the TO_LOB function');
1 row inserted.

Command> INSERT INTO clb
       SELECT TO_LOB(v) FROM vc;
1 row inserted.

Command> SELECT * FROM clb;
< Showing the functionality of the TO_LOB function >
1 row found.
```

Because of the restriction mentioned above, you cannot use the TO_LOB function in all cases where you can use the TO_CLOB or TO_BLOB functions. The following example demonstrates the error you receive when you try to use the TO_LOB function in this manner:

```
Command> SELECT TO_LOB(col1) FROM bvar;
 2610: Operand data type 'BINARY' invalid for operator
 'TO_LOB' in expr ( TO_LOB( BVAR.COL1 ))
The command failed.
```

TO_NCLOB

The TO_NCLOB function converts one of the following values to a NCLOB:

- Character value contained in one of the following data types: CHAR, VARCHAR2, NVARCHAR2, TT_VARCHAR, TT_NVARCHAR, or NCLOB
- Datetime value contained in a DATE or TIMESTAMP data type
- Number value contained in a NUMBER, BINARY_FLOAT, or BINARY_DOUBLE data type

This function is not supported in TimesTen Scaleout.

SQL syntax

TO_NCLOB (*ValidDataType*)

Parameters

TO_NCLOB has the parameters:

Parameter	Description
ValidDataType	A value of one of the valid data types mentioned above.

Examples

The following converts the data in the VARCHAR2 job_title column to be of data type NCLOB.

```
Command> SELECT TO_NCLOB(job_title) FROM jobs;

< Public Accountant >
< Accounting Manager >
< Administration Assistant >
< President >
< Administration Vice President >
< Accountant >
< Finance Manager >
< Human Resources Representative >
< Programmer >
< Marketing Manager >
< Marketing Representative >
< Public Relations Representative >
< Purchasing Clerk >
< Purchasing Manager >
< Sales Manager >
< Sales Representative >
< Shipping Clerk >
< Stock Clerk >
< Stock Manager >
19 rows found.
```

TO_NUMBER

Converts an expression to a value of NUMBER type.

SQL syntax

TO_NUMBER (*Expression*[, *Format]*)

Parameters

TO_NUMBER has the parameters:

Parameter	Description
Expression	The expression to be converted, where the value can be of type CHAR, VARCHAR2, NCHAR, NVARCHAR2, BINARY_FLOAT, BINARY_DOUBLE, CLOB, or NCLOB.
Format	If specified, the format is used to convert *Expression* to a value of NUMBER type. The format string identifies the number format model. The format and can be either a constant or a parameter.

Description

You can use a number format model with the TO_NUMBER function. For more information on number format models, see "Number Format Models".

Examples

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

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

TRIM

The TRIM function trims leading or trailing characters (or both) from a character string.

SQL syntax

There are four syntax options for TRIM:

- You can specify one of the TRIM qualifiers (LEADING or TRAILING or BOTH) with the *Trim_character*:

  TRIM (LEADING|TRAILING|BOTH *Trim_character* FROM *Expression*)

- You can specify one of the TRIM qualifiers (LEADING or TRAILING or BOTH) without the *Trim_character*. In this case, *Trim_character* defaults to a blank.

  TRIM (LEADING|TRAILING|BOTH FROM *Expression*)

- You can specify the *Trim_character* without one of the TRIM qualifiers, which removes both leading and trailing instances of *Trim_character* from *Expression*.

  TRIM (*Trim_character* FROM *Expression*)

- You can specify the *Expression* without a qualifier or a *Trim_character*, which results in leading and trailing blank spaces removed from *Expression*.

  TRIM (*Expression*)

Parameters

TRIM has the parameters:

Parameter	Description
LEADING | TRAILING | BOTH	LEADING | TRAILING| BOTH are qualifiers to TRIM function. LEADING removes all leading instances of *Trim_character* from *Expression*. TRAILING removes all trailing instances of *Trim_character* from *Expression*. BOTH removes leading and trailing instances of *Trim_character* from *Expression*.
[*Trim_character*]	If specified, *Trim_character* represents the CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column used for trimming *Expression*. Must be only one character. If you do not specify *Trim_character*, it defaults to a single blank. If *Trim_character* is a character literal, enclose it in single quotes.
Expression	*Expression* is the CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB operand or column to be trimmed. If *Expression* is a character literal, enclose it in single quotes.

Description

- If *Expression* is of type CHAR or VARCHAR2, the data type returned is VARCHAR2. If *Expression* is of type NCHAR or NVARCHAR2, the data type returned is NVARCHAR2. If *Expression* is of type CLOB, the data type returned is CLOB. If *Expression* is of type NCLOB, the data type returned is NCLOB. The returned data type length is equal to the data type length of *Expression*.
- If *Expression* is a data type defined with CHAR length semantics, the returned length is expressed in CHAR length semantics.
- If either *Trim_character* or *Expression* is NULL, the result is NULL.
- You can specify TT_CHAR, TT_VARCHAR, TT_NCHAR, and TT_NVARCHAR for *Trim_character* and *Expression*. If *Expression* is of type TT_CHAR or TT_VARCHAR, the data type returned is TT_VARCHAR. If *Expression* is of type TT_NCHAR or TT_NVARCHAR, the data type returned is TT_NVARCHAR.
- If *Trim_character* is of type NCHAR or NVARCHAR2 and *Expression* is of type CHAR or VARCHAR2, then *Trim_character* is demoted to CHAR or VARCHAR2 before TRIM is invoked. The conversion of *Trim_character* could be lost. If *Trim_character* is not in the database character set, then the query may produce unexpected results.
- For CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB and NCLOB types:
  - If all the characters in *Expression* are removed by the TRIM function, the result is NULL.
- For TT_CHAR, TT_VARCHAR, TT_NCHAR and TT_NVARCHAR types:
  - If all the characters in *Expression* are removed by the TRIM function, the result is the empty string.

Examples

Use TRIM function with qualifier to remove *Trim_character* '0' from *Expression* '0000TRIM Example0000':

```
Command> SELECT TRIM (LEADING '0' FROM '0000TRIM Example0000') FROM dual;
< TRIM Example0000 >
1 row found.
Command> SELECT TRIM (TRAILING '0' FROM '0000TRIM Example0000') FROM dual;
< 0000TRIM Example >
1 row found.
Command> SELECT TRIM (BOTH '0' FROM '0000TRIM Example0000') FROM dual;
< TRIM Example >
1 row found.
```

Use TRIM function with qualifier to remove blank spaces. Do not specify a *Trim_character*. Default value for *Trim_character* is blank space:

```
Command> SELECT TRIM (LEADING FROM '   TRIM Example   ') FROM dual;
< TRIM Example    >
1 row found.
Command> SELECT TRIM (TRAILING FROM '   TRIM Example   ') FROM dual;
<    TRIM Example >
1 row found.
Command> SELECT TRIM (BOTH FROM '   TRIM Example   ') FROM dual;
< TRIM Example >
1 row found.
```

Use TRIM function with *Trim_character* '0'. Do not specify a qualifier. Leading and trailing '0's are removed from *Expression* '0000TRIM Example0000':

```
Command> SELECT TRIM ('0' FROM '0000TRIM Example0000') FROM dual;
< TRIM Example >
1 row found.
```

Use TRIM function without a qualifier or *Trim_character*. Leading and trailing spaces are removed.

```
< TRIM Example >
1 row found.
Command> SELECT TRIM ('    TRIM Example    ') FROM dual;
```

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

See "Format Model for ROUND and TRUNC Date Functions" for information on the supported format models to use in *fmt*.

Examples

```
Command> SELECT TRUNC (TO_DATE ('27-OCT-92','DD-MON-YY'),'YEAR') FROM dual;
< 2092-01-01 00:00:00 >
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. The value of *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

Examples

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

SELECT TRUNC (15.79,-1) FROM dual;
< 10 >
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 is used by a hash index.

SQL syntax

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

Parameters

TT_HASH has the parameter:

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 depends 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 ttXactAdmin display the hash value as a signed decimal so that the value matches TT_HASH output.

Examples

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;
```

UID

This function returns an integer (TT_INTEGER) that uniquely identifies the session user.

SQL syntax

```
UID
```

Parameters

UID has no parameters.

Examples

```
SELECT UID FROM dual;
< 10 >
1 row found.
```

UNISTR

The UNISTR function 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. 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. TimesTen returns the value in UTF-16 format. You can specify Unicode escapes as part of the string.

Examples

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.
```

USER

Returns the name of the TimesTen user who is currently connected to the database.

SQL syntax

```
USER
```

Parameters

USER has no parameters.

Examples

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

```
SELECT USER FROM dual;
```

VSIZE

The VSIZE function returns the number of bytes in the internal representation of an expression.

SQL syntax

```
VSIZE(Expression)
```

Parameters

VSIZE has the parameter:

Parameter	Description
Expression	Expression that is passed to the VSIZE function. The function returns the number of bytes in the internal representation of the expression.

Description

- If the value of expression is NULL, NULL is returned. Otherwise, the data type returned is NUMBER.
- The VSIZE function does not support LOB data directly. However, LOBs can be passed in as arguments through implicit data conversion.

Examples

Use the VSIZE function to return the number of bytes in the last_name column of the employees in department 10.

```
Command> SELECT last_name, VSIZE (last_name) "BYTES" FROM employees
       WHERE department_id = 10 ORDER BY employee_id;
< Whalen, 6 >
1 row found.
```

This example illustrates how to use the VSIZE function on a column defined with the CLOB data type. This example first creates the vsize_varchar2 table with the col1 column defined with the VARCHAR2(200) data type. It then creates the vsize_clob table with the col1 column defined with the CLOB data type. The same string is inserted into col1 for each table. The VSIZE function is then used to return the number of bytes in the internal representation of the data in col1. For the vsize_clob table, the CAST function is used to cast the CLOB data type as the VARCHAR2(200) data type (for the col1 column). As illustrated, the VSIZE function returns the same result for the same query on the vsize_varchar2 table as on the vsize_clob table.

```
Command> CREATE TABLE vsize_varchar2 (col1 VARCHAR2 (200));
Command> CREATE TABLE vsize_clob (col1 CLOB);
Command> INSERT INTO vsize_varchar2 VALUES ('This is a test to illustrate how to
         use the VSIZE function on a column defined with the CLOB
         data type');
1 row inserted.
Command> INSERT INTO vsize_clob VALUES ('This is a test to illustrate how to
         use the VSIZE function on a column defined with the CLOB
         data type');
1 row inserted.
Command> SELECT VSIZE (col1) FROM vsize_varchar2;
< 102 >
1 row found.
Command> SELECT VSIZE (CAST (col1 AS VARCHAR2 (200))) FROM vsize_clob;
< 102 >
1 row found.
```

This example illustrates the difference between the LENGTH and the VSIZE functions. The LENGTH function returns the length of SYSDATE. The VSIZE function returns the number of bytes in the internal representation of SYSDATE.

```
Command> SELECT SYSDATE FROM dual;
< 2021-03-07 10:47:40 >
1 row found.
Command> SELECT LENGTH (SYSDATE) FROM dual;
< 19 >
1 row found.
Command> SELECT VSIZE (SYSDATE) FROM dual;
< 7 >
1 row found.
```

5

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 INSERT...SELECT and UPDATE, and in 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

Component	Description
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" 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).

Component	Description
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.

  Figure 5-1 shows the values that result from logical operations. A question mark (?) represents the NULL value.

Figure 5-1 Values that result from logical operations

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 does not 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 "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

Component	Description
Expression	See "Expression Specification" for the syntax. 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	See "Subqueries" for the syntax.

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.

Examples

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);
```

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;
```

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 | (*RowValueConstructorList*) |

The syntax for *RowValueConstructorList*:

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

The syntax for *RowValueConstructorElement*:

Expression | NULL

The syntax for *CompOp*:

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

The syntax for one element in the *ValueList* (no parentheses necessary):

ConstantExpression

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

({*ConstantExpression*} [,...])

The syntax for an empty *ValueList*:

( )

Parameters

Component	Description
Expression	Specifies a value to be obtained. The values in *ValueList* must be compatible with the expression. See "Expression Specification" for information on the syntax of expressions.
ConstantExpression	Specifies a constant value or an expression that evaluates to a constant value (such as a number, character string, or date). This includes support for bound values (? or :*DynamicParameter*), NULL, and calls to functions that return constant values.
=	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 *RowValueConstructor* values. The *ValueList* can be an empty list (sometimes generated by SQL generation tools) or consists of *ConstantExpression* entries. The *ValueList* *cannot* include column references, sequences, subqueries, ROWID values, or ROWNUM values. The *ValueList* can be nested if *RowValueConstructor* is a list. For example: (x, y) not in ((1+1, 2), (abs(-1), 5+1))

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)).
- All character data types are compared in accordance with the current value of the NLS_SORT session parameter.
- See "Numeric Data Types" 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.

Examples

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

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

For *ValueList* examples, see the Examples section in "ANY / IN Predicate (Value List)".

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 *anyrow* 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 | (*RowValueConstructorList*) | *Subquery*

The syntax for *RowValueConstructorList*:

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

The syntax for *RowValueConstructorElement*:

Expression | NULL

The syntax for *CompOp*:

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

Parameters

Component	Description
Expression	See "Expression Specification" for information on syntax. 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	See "Subqueries" for information on the syntax for subqueries.

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.

Examples

This example retrieves a list of customers having at least one unshipped order:

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

This is an example of an IN predicate with subquery. It SELECTs customers having at least one unshipped order:

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

This example 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);
```

This example 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);
```

This example 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);
```

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.
```

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 | (*RowValueConstructorList*) |

The syntax for *RowValueConstructorList*:

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

The syntax for *RowValueConstructorElement*:

Expression | NULL

The syntax for *CompOp*:

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

The syntax for one element in the *ValueList* (no parentheses necessary):

ConstantExpression

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

({*ConstantExpression*} [,...])

The syntax for an empty *ValueList*:

( )

Parameters

Component	Description
Expression	Specifies a value to be obtained. The values in *ValueList* must be compatible with the expression. See "Expression Specification" for information on syntax.
ConstantExpression	Specifies a constant value or an expression that evaluates to a constant value (such as a number, character string, or date). This includes support for bound values (? or :*DynamicParameter*), NULL, and calls to functions that return constant values.
=	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.

Component	Description
ValueList	A list of values that are compared against the *RowValueConstructor* values. The *ValueList* can be an empty list (sometimes generated by SQL generation tools) or consists of *ConstantExpression* entries. The *ValueList* *cannot* include column references, sequences, subqueries, ROWID values, or ROWNUM values. The *ValueList* can be nested if *RowValueConstructor* is a list. For example: (x, y) not in ((1+1, 2), (abs(-1), 5+1))

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.
- All character data types are compared in accordance with the current value of the NLS_SORT session parameter.
- See "Numeric Data Types" for information about how TimesTen compares values of different but compatible types.
- When evaluating an empty *ValueList*, the result of *Expression* IN is false.

Examples

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)
```

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

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

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);
```

The following example queries an empty select list for IN condition.

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

The following example uses a list of expressions with IN.

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

The next three examples, using ttIsql, show the use of constant expressions. Assume a table T with a single column named X of type NUMBER, with the following data:

```
Command> SELECT * FROM t;
< 1 >
< 2 >
< 3 >
< 4 >
< 5 >
5 rows found.
```

This first example uses constant expressions in a query:

```
Command> SELECT x FROM t WHERE x in (abs(1-2), TO_NUMBER('1')+2, 3);
< 1 >
< 3 >
2 rows found.
```

This second example also uses dynamic parameters:

```
Command> SELECT x FROM t WHERE x = ANY (1+?, 1+3, ?);

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 '_QMARK_1' (NUMBER) > 1
Enter Parameter 2 '_QMARK_2' (NUMBER) > 5
< 2 >
< 4 >
< 5 >
3 rows found.
```

This third example also uses NULL:

```
Command> SELECT x FROM t WHERE x IN (null, 1, 3+1, 2);
< 1 >
< 2 >
< 4 >
3 rows found.
```

For the next example, consider a table T_DATE with a single column named MYDATE of type DATE, with the following data:

```
Command> SELECT * FROM t_date;
< 2013-08-13 00:00:00 >
< 2013-08-14 00:00:00 >
< 2013-08-15 00:00:00 >
3 rows found.
```

The example uses constant expressions for dates:

```
Command> SELECT mydate FROM t_date
       WHERE mydate IN (DATE '2013-08-12'+1,
                  DATE '2013-08-12'+2);
< 2013-08-13 00:00:00 >
< 2013-08-14 00:00:00 >
2 rows found.
```

For the next example, consider a table MYCHARS with a single column named COL1 of type VARCHAR2(32), with the following data:

```
Command> SELECT * FROM mychars;
< abc >
< def >
```

```
< ghi >
3 rows found.
```

The example uses a function call that returns a constant expression:

```
Command> SELECT col1 FROM mychars WHERE col1 IN (ltrim('abcdef', 'abc'));
< def >
1 row found.
```

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

Parameter	Description
Expression1, *Expression2*, *Expression3*	See "Expression Specification" for information on the syntax. 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.

  See "Comparison Predicate" for information on comparisons.
- The BETWEEN predicate is not supported for NCHAR types.

Examples

Parts sold for under $250.00 and over $1500.00 are discounted 25 percent.

```
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 | (*RowValueConstructorList*) | *ScalarSubquery*

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

Component	Description
Expression	See "Expression Specification" for information on syntax. 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. See "Subqueries" for information on scalar subqueries.
=	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

- All character data types are compared in accordance with the current value of the NLS_SORT session parameter.
- If *RowValueConstructorList* is specified only the operators = and <> are allowed.
- See "Numeric Data Types" for information about how TimesTen compares values of different but compatible types.
- If either side of a comparison predicate evaluates to UNKNOWN or NULL, this implies that neither the predicate nor the negation of the predicate is TRUE.
- The NULL value itself can be used directly as an operand of an operator or predicate. For example, the (1 = NULL) comparison is supported. This is the same as if you cast NULL to the appropriate data type, as follows: (1 = CAST(NULL AS INT)). Both methods are supported and return the same results.

Examples

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

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

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.
```

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 or returns no rows for NOT EXISTS.

SQL syntax

```
[NOT] EXISTS (Subquery)
```

Parameters

The EXISTS predicate has the following parameter:

Parameter	Description
Subquery	See "Subqueries" for information on syntax.

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

Examples

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 = 'unshipped');
```

Get a list of customers having no unshipped orders.

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

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

Parameter	Description
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 want 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.
- See "INF and NAN" for more information on Inf and NaN.

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

Parameter	Description
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 want 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.
- See "INF and NAN" for more information on Inf and NaN.

IS NULL Predicate

The 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* | *LOBDataType*} IS [NOT] NULL

Parameters

Parameter	Description
ColumnName	The name of a column from which a value is to be taken. See "Names, Namespace and Parameters" for information on column names.
Constant	A specific value. See "Constants" for information on constants.
Expression	Expression to test.
LOBDataType	Value to test that is in a CLOB, BLOB, or NCLOB data type.

Examples

Use IS NULL to identify the president of the company, who is the only person without a manager.

```
Command> SELECT * FROM employees
       WHERE manager_id IS NULL;
< 100, Steven, King, SKING, 515.123.4567, 1987-06-17 00:00:00, AD_PRES, 24000,
<NULL>, <NULL>, 90 >
1 row found.
```

The following statement uses IS NULL to identify all locations without a state or province.

```
Command> SELECT * FROM locations
       WHERE state_province IS NULL;
< 1000, 1297 Via Cola di Rie, 00989, Roma, <NULL>, IT >
< 1100, 93091 Calle della Testa, 10934, Venice, <NULL>, IT >
< 1300, 9450 Kamiya-cho, 6823, Hiroshima, <NULL>, JP >
< 2000, 40-5-12 Laogianggen, 190518, Beijing, <NULL>, CN >
< 2300, 198 Clementi North, 540198, Singapore, <NULL>, SG >
< 2400, 8204 Arthur St, <NULL>, London, <NULL>, UK >
6 rows found.
```

LIKE Predicate

A LIKE predicate evaluates to TRUE if the source contains a given pattern. The LIKE predicate matches a portion of one character value to another by searching the source for the pattern specified.

SQL syntax

Source [NOT] LIKE *Pattern*
 [ESCAPE {'*EscapeChar*' | {? | :*DynamicParameter*} }]

The syntax for *Pattern* is as follows:

Expression [|| *Expression*] [...]

Parameters

Parameter	Description
Source	This source is searched for all occurrences of the pattern. The source may be an expression, column, character string resulting from a function, or any combination of these that results in a character string used for the source on which the pattern is matched. The source can be a CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. See "Expressions" for information on expressions. See "Pattern Matching for Strings of NCHAR, NVARCHAR2, and NCLOB Data Types" for information on searching within a national character string within NCHAR, NVARCHAR, or NCLOB.

Parameter	Description
Pattern	Describes a character pattern that you are searching for in the source with one or more expressions. The data type of the pattern should be a character string data type, such as CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. Multiple expressions may be concatenated to form the character string used for the pattern. The pattern consists of characters including digits and special characters. For example, NAME LIKE 'Annie' evaluates to TRUE only for a name of Annie with no spaces. You can also use the predicate to test for a partial match by using one or more of the following symbols: • The symbol _ represents any single character. For example: BOB and TOM both satisfy the predicate NAME LIKE '_O_'. • The symbol % represents any string of zero or more characters. For example: MARIE and RENATE both satisfy the predicate NAME LIKE '%A%'. You can use the _ or % symbols multiple times and in any combination in a pattern. However, you cannot use the symbols literally within a pattern unless you use the ESCAPE clause and precede the symbols with the escape character, described by the *EscapeChar* parameter.
Expression	Any expression included in the pattern may be a column, a dynamic parameter, or the result of a function that evaluates to a character string. See "Expression Specification" for information on expressions.
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 % symbols in the pattern act as wild card characters. If an escape character is specified, 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.
- When providing a combination of expressions, columns, character strings, dynamic parameters, or function results to form the pattern, you can concatenate items together using the || operator to form the final pattern.
- Case is significant in all conditions comparing character expressions that use the LIKE predicate.
- If the value of the expression, the pattern, or the escape character is NULL, the LIKE predicate evaluates to NULL.
- The LIKE predicate may be slower when used on a multibyte character set.

- See "Pattern Matching for Strings of NCHAR, NVARCHAR2, and NCLOB Data Types" for more information on searching within a national character string within NCHAR, NVARCHAR, or NCLOB.

Examples

Find each employee whose last name begins with 'Sm'.

```
Command> SELECT employee_id, last_name,first_name FROM employees
       WHERE last_name LIKE 'Sm%'
       ORDER BY employee_id,last_name,first_name;
< 159, Smith, Lindsey >
< 171, Smith, William >
2 rows found.
```

Find each employee whose last name begins with 'SM'. This query returns no results because there are no employees whose last_name begins with upper case 'SM'.

```
Command> SELECT employee_id, last_name,first_name from employees
       WHERE last_name LIKE 'SM%'
       ORDER BY employee_id,last_name,first_name;
0 rows found.
```

However, by upper casing the source value of the last name column, you can find all names that begin with 'SM'.

```
Command> SELECT employee_id, last_name, first_name FROM employees
       WHERE UPPER(last_name) LIKE ('SM%');
< 159, Smith, Lindsey >
< 171, Smith, William >
2 rows found.
```

Use a dynamic parameter denoted by ? to find each employee whose last name begins with 'Sm' at execution time.

```
Command> SELECT employee_id, last_name,first_name FROM employees
       WHERE last_name like ?
       ORDER BY employee_id,last_name,first_name;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 '_QMARK_1' (VARCHAR2) > 'Sm%'
< 159, Smith, Lindsey >
< 171, Smith, William >
2 rows found.
```

Use a bind variable denoted by :a to find each employee whose last name begins with 'Sm' at execution time.

```
Command> SELECT employee_id, last_name,first_name FROM employees
       WHERE last_name LIKE :a
       ORDER BY employee_id,last_name,first_name;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 'A' (VARCHAR2) > 'Sm%'
```

```
< 159, Smith, Lindsey >
< 171, Smith, William >
2 rows found.
```

For each employee whose last name begins with 'Smit', find the last name of the manager. Display the first name and last name of the employee and the last name of the manager.

```
Command> SELECT e1.first_name || ' ' || e1.last_name||' works for '||e2.last_name
       FROM employees e1, employees e2
       WHERE e1.manager_id = e2.employee_id
       AND e1.last_name like 'Smit';
< Lindsey Smith works for Partners >
< William Smith works for Cambrault >
2 rows found.
```

This query pattern references the last_name column as the pattern for which to search:

```
Command> SELECT e1.first_name || ' ' || e1.last_name||' works for ' ||
           e2.last_name
       FROM employees e1, employees e2
       WHERE e1.manager_id = e2.employee_id AND 'Smith' like e1.last_name;
< Lindsey Smith works for Partners >
< William Smith works for Cambrault >
2 rows found.
```

The pattern can be a column or the result of a function. The following uses the UPPER function on both the source last_name column as well as the 'ma' search string for which you are searching:

```
Command> SELECT last_name, first_name FROM employees
       WHERE UPPER(last_name) LIKE UPPER('ma%');
< Markle, Steven >
< Marlow, James >
< Mallin, Jason >
< Matos, Randall >
< Marvins, Mattea >
< Mavris, Susan >
6 rows found.
```

The following query demonstrates using a dynamic parameter to request the pattern.

```
Command> SELECT first_name || ' ' || last_name
       FROM employees WHERE last_name like ?;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 '_QMARK_1' (VARCHAR2) > 'W%'
< Matthew Weiss >
< Alana Walsh >
< Jennifer Whalen >
3 rows found.
```

The following query demonstrates combining a character string with a dynamic parameter in the pattern.

```
Command> SELECT first_name || ' ' || last_name
       FROM employees WHERE last_name like 'W' || ?;

Type '?' for help on entering parameter values.
```

```
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 '_QMARK_1' (VARCHAR2) > '%'
< Matthew Weiss >
< Alana Walsh >
< Jennifer Whalen >
3 rows found.
```

Pattern Matching for Strings of NCHAR, NVARCHAR2, and NCLOB Data Types

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

Character	Description
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-insensitive 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.

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

This example is equivalent:

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

6 SQL Statements

This chapter provides information about the SQL statements available in TimesTen.

SQL statements are generally considered to be either data manipulation language (DML) statements or data definition language (DDL) statements.

DML statements modify database objects. INSERT, UPDATE and DELETE are examples of DML statements. The SELECT statement retrieves data from one or more tables or views.

DDL statements modify the database schema. CREATE TABLE and DROP TABLE are examples of DDL statements.

In addition to an alphabetical listing of all statements, this chapter also contains:

- Summary of SQL Statements Supported in TimesTen
- Comments Within SQL Statements
- Optimizer Hints

Summary of SQL Statements Supported in TimesTen

Table 6-1 shows a summary of the SQL statements in TimesTen. The second column indicates if the statement is supported in TimesTen Scaleout. Every statement except ALTER SEQUENCE is supported in TimesTen Classic.

Table 6-1 SQL statements supported in TimesTen

SQL statement	Supported in TimesTen Scaleout?
ALTER ACTIVE STANDBY PAIR	N
ALTER CACHE GROUP	Y
ALTER FUNCTION	N
ALTER PACKAGE	N
ALTER PROFILE	Y
ALTER PROCEDURE	N
ALTER REPLICATION	N
ALTER SEQUENCE	Y Not supported in TimesTen Classic.
ALTER SESSION	Y
ALTER TABLE	Y Unsupported clauses: Aging and column-based compression Unsupported data types: LOB columns are not supported in tables. LOB variables are supported in PL/SQL programs.

Table 6-1 (Cont.) SQL statements supported in TimesTen

SQL statement	Supported in TimesTen Scaleout?
ALTER USER	Y
CALL	Y
COMMIT	Y
CREATE ACTIVE STANDBY PAIR	N
CREATE CACHE GROUP	Y static read-only with incremental autorefresh
CREATE FUNCTION	Y
CREATE INDEX	Y
CREATE MATERIALIZED VIEW	Y with restrictions
CREATE PACKAGE	Y
CREATE PACKAGE BODY	Y
CREATE PROCEDURE	Y
CREATE PROFILE	Y
CREATE REPLICATION	N
CREATE SEQUENCE	Y with TimesTen Scaleout specific BATCH clause.
CREATE SYNONYM	Y
CREATE TABLE	Y including CREATE TABLE... AS SELECT Unsupported clauses: Aging and column-based compression Unsupported data types: LOBs and ROWID Distribution clause is not supported for global temporary tables.
CREATE USER	Y
CREATE VIEW	Y
DELETE	Y
DROP ACTIVE STANDBY PAIR	N
DROP CACHE GROUP	Y static read-only cache groups with incremental autorefresh
DROP FUNCTION	Y
DROP INDEX	Y
DROP MATERIALIZED VIEW	Y
DROP PACKAGE [BODY]	Y
DROP PROCEDURE	Y
DROP PROFILE	Y
DROP REPLICATION	N
DROP SEQUENCE	Y
DROP SYNONYM	Y

Table 6-1 (Cont.) SQL statements supported in TimesTen

SQL statement	Supported in TimesTen Scaleout?
DROP TABLE	Y
DROP USER	Y
DROP VIEW	Y
FLUSH CACHE GROUP	N
GRANT	Y
INSERT	Y
INSERT...SELECT	Y
LOAD CACHE GROUP	Y static read-only cache groups with incremental autorefresh
MERGE	N
REFRESH CACHE GROUP	Y static read-only cache groups with incremental autorefresh
REVOKE	Y
ROLLBACK	Y
SELECT	Y
TRUNCATE TABLE	Y, but TRUNCATE TABLE is similar to a DDL statement that invalidates all commands that depend on the table being truncated. It is preferable to use the DELETE statement rather than the TRUNCATE statement to delete all rows in a table.
UNLOAD CACHE GROUP	Y static read-only cache groups with incremental autorefresh
UPDATE	Y

Comments Within SQL Statements

A comment can appear between keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement in two ways:

- Begin the comment with a slash and an asterisk (/*). Proceed with the text of the comment. The text can span multiple lines. End the comment with an asterisk and a slash. (*/). You do not need to separate the opening and terminating characters from the text by a space or line break.
- Begin the comment with two hyphens (--). Proceed with the text of the comment. The text cannot extend to a new line. End the comment with a line break.

Optimizer Hints

Optimizer hints are instructions that are passed to the TimesTen query optimizer. The optimizer considers these hints when choosing the best execution plan for your query. Most of the hints are supported both in TimesTen Scaleout and in TimesTen Classic. There are also hints that are supported only in TimesTen Scaleout. See Optimizer Hints Supported in TimesTen Scaleout Only for information.

TimesTen supports three levels of optimizer hints:

- Statement level optimizer hints: When specified, the optimizer considers the hint for the particular statement. See Statement Level Optimizer Hints for details.
- Transaction level optimizer hints: When specified (by calling the appropriate built-in procedure), the optimizer considers the hint for the entire transaction. See Use Optimizer Hints to Modify the Execution Plan in the *Oracle TimesTen In-Memory Database Operations Guide*.
- Connection level optimizer hints: When specified, the optimizer considers the hint for the entire connection. See Use Optimizer Hints to Modify the Execution Plan in the *Oracle TimesTen In-Memory Database Operations Guide* and OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for details.

The order of precedence for optimizer hints is statement level hints, transaction level hints and then connection level hints. Table 6-2 provides a summary of the statement, transaction, and connection level optimizer hints.

Table 6-2 Summary of Statement, Transaction, and Connection Level Optimizer Hints

Statement Level Optimizer Hint	Transaction Level Optimizer Hint	Connection Level Optimizer Hint
You specify the hint within the comment syntax and after a SQL VERB in a SQL statement.	You specify the hint by calling the ttOptSetFlag, or the ttOptSetOrder, or the ttOptUseIndex built-in procedure.	You specify the hint in the OptimizerHint general connection attribute.
The hint is scoped to the SQL statement.	The hint is scoped to the transaction.	The hint is scoped to the connection.
The autocommit setting has no effect. After the statement containing the hint is executed, the hint has no effect on future statements or queries.	The autocommit setting has an effect. You must set autocommit to off. Doing so ensures the hint is in effect for the duration of your transaction (until you issue a commit or rollback). If you do not set autocommit to off, the statement is executed in its own transaction and the hint only has an effect on the statement.	The autocommit setting has no effect. The hint is in effect for the duration of the connection.
The optimizer considers the hint for the statement only.	The optimizer considers the hint for all statements in the transaction.	The optimizer considers the hint for all statements in the connection.
The hint is supported in the CREATE TABLE...AS SELECT statement.	The hint is not supported in the CREATE TABLE...AS SELECT statement. This is a DDL statement and TimesTen implicitly commits a DDL statement.	The hint is not supported in the CREATE TABLE...AS SELECT statement. This is a DDL statement and TimesTen implicitly commits a DDL statement.

Table 6-2 (Cont.) Summary of Statement, Transaction, and Connection Level Optimizer Hints

Statement Level Optimizer Hint	Transaction Level Optimizer Hint	Connection Level Optimizer Hint
If you specify the hint in a transaction in which transaction level optimizer hints or connection level optimizer hints are specified, the statement level optimizer hint overrides the transaction level hint or the connection level hint for the SQL statement. After TimesTen executes the SQL statement: • The original transaction level optimizer hint remains in effect for the duration of the transaction or • The original connection level optimizer hint remains in effect for the duration of the connection.	The hint is in effect for the duration of the transaction. If you specify a statement level optimizer hint in a SQL statement, the statement level optimizer hint is in effect for the statement and the optimizer does not use the transaction level hint for the statement. After TimesTen executes the statement, the original transaction level optimizer hint remains in effect for the duration of the transaction. A hint specified at this level overrides the same hint specified at the connection level.	The hints are in effect for the duration of the connection. The order of precedence is statement level, transaction level, and then connection level.
You use the statement level optimizer hints if you want to influence the optimizer for a specific statement. You must specify the hint for each statement in which you want to influence the optimizer. This could result in multiple alterations to your statements.	You use the transaction level optimizer hints to influence the optimizer for all statements in a transaction. You do not have to specify a hint for each statement. The hint applies to all statements in the transaction. The hint can be overridden by specifying the hint at the statement level.	You use the connection level optimizer hint to influence the optimizer for all statements in the connection. The hint can be overridden by specifying the hint at the transaction or at the statement level.

Statement Level Optimizer Hints

Statement level optimizer hints are comments in a SQL statement that pass instructions to the TimesTen query optimizer. The optimizer considers these hints when choosing the best execution plan for your query. It analyzes the SQL statements and generates a query plan which is then used by the SQL execution engine to execute the query and return the data.

See Use Optimizer Hints to Modify the Execution Plan in *Oracle TimesTen In-Memory Database Operations Guide* for information about statement level optimizer hints.

SQL Syntax

A SQL statement can have one comment that includes one or more statement level optimizer hints.
These hints are only supported in TimesTen Cache:

- TT_DynamicLoadMultiplePKs
- TT_DynamicLoadRootTbl
- TT_DynamicPassThrough

Some hints are not supported in certain SQL statements:

- TT_CommitDMLOnSuccess is supported in the DELETE, INSERT, and UPDATE statements. It is also valid in the INSERT...SELECT statement and must follow the SELECT keyword. This hint is supported in TimesTen Scaleout only.
- The TT_GridQueryExec and TT_PartialResult hints are supported in the SELECT, INSERT...SELECT, and CREATE TABLE... AS SELECT SQL statements only and these hints must follow the SELECT keyword. These hints are supported in TimesTen Scaleout only.

- The remaining hints are supported in the DELETE, INSERT, MERGE, SELECT, UPDATE, INSERT...SELECT, and CREATE TABLE...AS SELECT SQL statements and these hints must follow the DELETE, INSERT, MERGE, SELECT, or UPDATE keyword.

You embed statement level optimizer hints in comment syntax. TimesTen supports hints in comments that span one line and in comments that span more than one line. If your comment that contains the hint spans one or more lines, use the comment syntax, /*+...*/. If your comment that contains the hint spans one line, use the comment syntax, --+.

Syntax:

```
SQL VERB {/*+ [CommentText] hint [{hint|CommentText} [...]] */ |
       --+ [CommentText] hint [{hint|CommentText} [...]] }
hint::= ScaleoutHint | CacheHint | JoinOrderHint | IndexHint| FlagHint
ScaleoutHint::= TT_CommitDMLOnSuccess({0|1})|TT_GridQueryExec({LOCAL|GLOBAL})|
           TT_PartialResult(0|1)
CacheHint::= TT_DynamicLoadMultiplePKs ({0|1})|TT_DynamicLoadRootTbl ({0|1})|
         TT_DynamicPassthrough(N)
JoinOrderHint::= TT_JoinOrder (CorrelationName CorrelationName [...])
IndexHint::= TT_Index (CorrelationName,IndexName,{0|1} [;...])
FlagHint::= FlagName (0|1)
FlagName::= TT_BranchAndBound|TT_CountAsInt|TT_DynamicLoadEnable|
        TT_DynamicLoadErrorMode| TT_FirstRow|TT_ForceCompile|
        TT_GenPlan|TT_HashGb|TT_HashScan|TT_IndexedOr|TT_MergeJoin|
        TT_NestedLoop|TT_NoRemRowIdOpt|TT_Range|TT_Rowid|TT_RowLock|
        TT_ShowJoinOrder|TT_TblLock|TT_TblScan|TT_TmpHash|TT_TmpRange|
        TT_TmpTable|TT_UseBoyerMooreStringSearch|
```

Parameters

Parameter	Description
SQL VERB	*SQL VERB* refers to one of the keywords: DELETE, INSERT, MERGE, SELECT, or UPDATE. You embed a statement level optimizer hint in comment syntax and if the comment syntax contains a statement level optimizer hint, the comment syntax must follow the *SQL VERB*. The TT_GridQueryExec and TT_PartialResult hints are valid for the SELECT keyword only.
/*+ *hint* */	One or more hints that are embedded in comment syntax. The comment syntax can span one or more lines. The plus sign (+) denotes the start of a statement level optimizer hint. Make sure there is no space between the star (*) and the plus sign (+).
--+ *hint*	One or more hints that are embedded in comment syntax. The comment syntax can only span one line. The plus sign (+) denotes the start of a statement level optimizer hint. Make sure there is no space between the dash (-) and the plus sign (+).

Parameter	Description
hint	A statement level optimizer hint. A SQL statement supports one or more statement level optimizer hints as one comment string. For one SQL statement, you can specify one comment that contains one or more hints and that comment must follow a DELETE, INSERT, MERGE, SELECT, or UPDATE keyword (or for TT_GridQueryExec and TT_PartialResult, the SELECT keyword). TT_CommitDMLOnSuccess must follow a DELETE, INSERT, or UPDATE keyword and in the INSERT...SELECT statement, it must follow the SELECT keyword. If you specify more than one hint within the comment, make sure there is a space between the hints. Statement level optimizer hints are scoped to a SQL statement and have per query semantics. For hints other than TT_GridQueryExec, TT_PartialResult, or TT_CommitDMLOnSuccess: • The name and type of statement level optimizer hints map to the transaction level optimizer hints. Transaction level optimizer hints are invoked by calling the built-in procedures ttOptSetFlag, ttOptSetOrder, or ttOptUseIndex. • Transaction level hints are scoped to the transaction and have transaction semantics. You must set autocommit to 0 and the transaction level optimizer hint is in effect for the duration of your transaction. • For more information, see ttOptSetFlag," "ttOptSetOrder or ttOptUseIndex in the *Oracle TimesTen In-Memory Database Reference*.
CommentText	Text within a comment string. You can use both statement level optimizer hints and commenting text within one comment. Make sure to include a space between the hint and the commenting text.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>ScaleoutHint</td><td>ScaleoutHint refers to the TT_CommitDMLOnSuccess statement level hint as well as the TT_GridQueryExec and the TT_PartialResult statement level optimizer hints. These hints are supported in TimesTen Scaleout only.

See TT_CommitDMLOnSuccess Optimizer Hint for detailed information.
TT_GridQueryExec (LOCAL|GLOBAL) returns query results for the local element (if LOCAL) or for all elements (if GLOBAL). If K-safety is set to 2 and GLOBAL is specified, the results include the data in all elements in a replica set.
LOCAL: Queries are executed in the local element only. If the local element does not have a full copy of the data, TimesTen Scaleout returns partial results.
GLOBAL: TimesTen Scaleout retrieves data from all elements, including copies of the rows from all tables from all replica sets to generate the results. This results in duplicate data returned if K-safety is set to 2 or if tables have a duplicate distribution scheme.
The default is neither local nor global. If you do not specify this hint, the query is executed in one logical data space. Exactly one full copy of the data is used to compute the query.
TT_PartialResult (0|1) returns an error (if 0) or partial results (if 1) when data is not available.
0: Returns error if the required data is not available in the case where all elements in a replica set are not available. If at least one element from each replica set is available or the data required by the query is available, the optimizer returns the query result correctly without error. This is the default.
1: Returns partial results if all elements in a replica set are not available.

<pre>SELECT /*+TT_GridQueryExec(LOCAL)*/
    COUNT(*), elementId# FROM t
    GROUP BY elementId#;

SELECT /*+TT_GridQueryExec(GLOBAL)*/
    COUNT(*), elementId# FROM t
    GROUP BY elementId#;

SELECT /*+TT_PartialResult(0)*/ COUNT
   (*), elementId# FROM t GROUP BY
   elementId#;

SELECT /*+TT_PartialResult(1)*/ COUNT
   (*), elementId# FROM t GROUP BY
   elementId#;</pre></td></tr>
<tr><td>CacheHint</td><td>CacheHint refers to the supported optimizer hints for TimesTen Cache. These hints are TT_DynamicLoadMultiplePKs, TT_DynamicLoadRootTbl, and TT_DynamicPassthrough. These hints are described later in this table (in alphabetical order).</td></tr>
</table>

Parameter	Description
JoinOrderHint::= TT_JoinOrder (*CorrelationName CorrelationName* [...])	*JoinOrderHint* refers to the syntax for the TT_JoinOrder statement level optimizer hint. The TT_JoinOrder hint instructs the optimizer to join your tables in a specified order. The join order is in effect for the statement only. Specify TT_JoinOrder and two or more correlation names. Make sure to enclose the correlation names in parentheses. *CorrelationName* refers to the alias of your table specified in the query or subquery of your SQL statement. Make sure to use the correlation name rather than the actual table name when using this hint. For example, if you are joining the employees and departments tables and you specify the emps correlation name for the employees table and the depts correlation name for the departments table, then to specify the TT_JoinOrder hint in ttIsql: `Command> SELECT /*+ TT_JoinOrder (EMPS DEPTS)*/...` If your TT_JoinOrder hint contains syntax errors, TimesTen ignores the hint. If your TT_JoinOrder hint does not contain a closing parenthesis, then the remainder of the comment string is ignored. So if you specify additional statement level optimizer hints after the TT_JoinOrder hint, and the TT_JoinOrder hint is missing the closing parenthesis, these additional statement level optimizer hints are ignored. You can execute the built-in procedure, ttOptSetOrder, to specify a join order for the duration of your transaction. The TT_JoinOrder statement level optimizer hint is in effect for the statement only. For more information on ttOptSetOrder, see ttOptSetOrder" in the *Oracle TimesTen In-Memory Database Reference*.

Parameter	Description

IndexHint::= TT_INDEX (*CorrelationName* *IndexName* {0|1} [;...])	*IndexHint* refers to the syntax for the TT_INDEX statement level optimizer hint. Use the TT_INDEX hint to direct the optimizer to use or not use an index for your table. The index hint is in effect for the statement only.
	CorrelationName refers to the alias of your table specified in the query or subquery of your SQL statement. Make sure to use the correlation name rather than the actual table name. If you omit the correlation name, the setting affects all tables with the specified index name.
	IndexName refers to the index you want the optimizer to consider or not consider for the table If you omit the index name, the setting applies to all indexes of the specified correlation.
	Specify a value of 0 to ask the optimizer not to consider the index. Specify a value of 1 to ask the optimizer to consider the index.
	For example, To direct the optimizer to use the index emp_name_ix for a query on the employees table and you specify the correlation name, e, for the employees table, then to specify the TT_INDEX hint in ttIsql:
	`Command> SELECT /*+ TT_INDEX`
	`(E,EMP_NAME_IX,1) */ ...`
	Use a semicolon (;) to include more than one *CorrelationName*, *IndexName*, {0|1} for the TT_INDEX hint. You must specify each of the parameters: the *CorrelationName*, the *IndexName*, and either 0 or 1.
	If your TT_Index hint contains syntax errors, TimesTen ignores the hint. If your TT_Index hint does not contain a closing parenthesis, then the remainder of the comment string is ignored. So if you specify additional statement level optimizer hints after the TT_Index hint and the TT_Index hint is missing the closing parenthesis, these additional statement level optimizer hints are ignored.
	You can execute the built-in procedure, ttOptUseIndex, to specify an index hint for the duration of your transaction. The TT_INDEX statement level optimizer hint is in effect for the statement only.
	For more information on ttOptUseIndex, see ttOptUseIndex" in the *Oracle TimesTen In-Memory Database Reference*.

FlagHint	*FlagHint* refers to statement level optimizer flags that you enable or disable to influence the execution plan of the TimesTen query optimizer. These flags map to the flags used in the ttOptSetFlag built-in procedure. Statement level optimizer hint flags are in effect for the statement only whereas transaction level optimizer hint flags are in effect for the duration of your transaction.
TT_BranchAndBound	Flag that maps to the flag BranchAndBound in the ttOptSetFlag built-in procedure.

Parameter	Description

TT_CountAsInt	This hint controls the return data type for the COUNT function (used in a query). Specify 1 to have the return data type be TT_INTEGER. Specify 0 to have the return data type be TT_BIGINT. If you do not specify this hint, the default return data type is TT_BIGINT. This hint is supported at the statement and at the connection levels. See "COUNT" for information on the COUNT function.
	This hint is provided for backward compatibility. If you specify the hint with a value of 1, it may result in an unexpected integer overflow. New applications should not specify this hint. This ensures TimesTen uses the default return data type of TT_BIGINT for the COUNT function.
	This example specifies a value of 1 for the hint. The return data type is TT_INTEGER:
	`Command> describe SELECT /*+TT_CountAsInt(1)*/ COUNT (*) FROM dual;`
	
	`Prepared Statement:`
	`  Columns:`
	`    EXP                  TT_INTEGER NOT NULL`
	This example specifies a value of 0 for the hint. The return data type is TT_BIGINT.
	`Command> describe SELECT /*+TT_CountAsInt(0)*/ COUNT (*) FROM dual;`
	
	`Prepared Statement:`
	`  Columns:`
	`    EXP                  TT_BIGINT NOT NULL`
	This example does not set the optimizer hint. The default return data type is TT_BIGINT.
	`describe SELECT COUNT (*) FROM dual;`
	
	`Prepared Statement:`
	`  Columns:`
	`    EXP                  TT_BIGINT NOT NULL`

TT_DynamicLoadEnable	Flag that maps to the flag DynamicLoadEnable in the ttOptSetFlag built-in procedure.
TT_DynamicLoadErrorMode	Flag that maps to the flag DynamicLoadErrorMode in the ttOptSetFlag built-in procedure.

Parameter	Description
TT_DynamicLoadMultiplePKs{ 0|1}	TimesTen Cache optimizer hint, supported in TimesTen Classic. This hint enables (if set to 1) or disables (if set to 0), the ability to dynamically load multiple cache instances on a single table cache group. The dynamic load operation must be triggered by a qualified SELECT statement that contains a WHERE clause, in which the WHERE clause references multiple primary key values of the root table of the cache group. The default is 1. When both the TT_DynamicLoadMultiplePKs and the TT_DynamicLoadRootTbl hints are specified, the TT_DynamicLoadMultiplePKs take precedence.
TT_DynamicLoadRootTbl	TimesTen Cache optimizer hint, supported in TimesTen Classic. This hint enables (if set to 1) or disables (if set to 0), the ability to dynamically load multiple cache instances on a single table cache group. The dynamic load operation must be triggered by a qualified SELECT statement that contains a WHERE clause, in which the WHERE clause does not reference multiple primary key values of the root table of the cache group. The default is 0. When both the TT_DynamicLoadMultiplePKs and the TT_DynamicLoadRootTbl hints are specified, the TT_DynamicLoadMultiplePKs take precedence.
TT_DynamicPassThrough(N)	TimesTen Cache optimizer hint, supported in TimesTen Classic. If specified, this hint limits the number of rows that can be dynamically loaded into a TimesTen cache instance. Specifically, if a dynamic load operation triggered by a qualified SELECT statement results in a number of rows that is greater than the specified N row limit, the cache instance is not loaded and instead the query is passed to the Oracle database. The dynamic load must be triggered by a qualified SELECT statement and the cache group must not have a WHERE clause. The hint is ignored for non-SELECT statements. Set this hint to the maximum number of rows you want dynamically loaded. If you set the hint to a value less than or equal to 0 or if you do not specify the hint, the dynamic load has no row limit. In this case, there is not a limit in the number of rows can be loaded into the cache instance. See Automatic Passthrough of Dynamic Load to the Oracle Database in the *Oracle TimesTen In-Memory Database Cache Guide* for details.
TT_FirstRow	Flag that maps to the flag FirstRow in the ttOptSetFlag built-in procedure.
TT_ForceCompile	Flag that maps to the flag ForceCompile in the ttOptSetFlag built-in procedure.
TT_GenPlan	Flag that maps to the flag GenPlan in the ttOptSetFlag built-in procedure.
TT_HashGb	Flag that maps to the flag HashGb in the ttOptSetFlag built-in procedure.
TT_HashScan	Flag that maps to the flag Hash in the ttOptSetFlag built-in procedure.
TT_IndexedOr	Flag that maps to the flag IndexedOr in the ttOptSetFlag built-in procedure.

Parameter	Description
TT_MergeJoin	Flag that maps to the flag MergeJoin in the ttOptSetFlag built-in procedure.
TT_NestedLoop	Flag that maps to the flag NestedLoop in the ttOptSetFlag built-in procedure.
TT_NoRemRowIdOpt	Flag that maps to the flag NoRemRowIdOpt in the ttOptSetFlag built-in procedure.
TT_Range	Flag that maps to the flag Range in the ttOptSetFlag built-in procedure.
TT_Rowid	Flag that maps to the flag Rowid in the ttOptSetFlag built-in procedure.
TT_RowLock	Flag that maps to the flag Rowlock in the ttOptSetFlag built-in procedure.
TT_ShowJoinOrder	Flag that maps to the flag ShowJoinOrder in the ttOptSetFlag built-in procedure.
TT_TblLock	Flag that maps to the flag TblLock in the ttOptSetFlag built-in procedure.
TT_TblScan	Flag that maps to the flag Scan in the ttOptSetFlag built-in procedure.
TT_TmpHash	Flag that maps to the flag TmpHash in the ttOptSetFlag built-in procedure.
TT_TmpRange	Flag that maps to the flag TmpRange in the ttOptSetFlag built-in procedure.
TT_TmpTable	Flag that maps to the flag TmpTable in the ttOptSetFlag built-in procedure.
TT_UseBoyerMooreStringSearch	Flag that maps to the flag UseBoyerMooreStringSearch in the ttOptSetFlag built-in procedure.

> **Note**
>
> For descriptions of flags discussed in the preceding table, see ttOptSetFlag" in the *Oracle TimesTen In-Memory Database Reference*

Description

- Embed statement level optimizer hints in comment syntax. Begin the comment with either /* or --. Follow the beginning comment syntax with a plus sign (+). The plus sign (+) signals TimesTen to interpret the comment as a list of hints. The plus sign (+) must follow immediately after the comment delimiter. (For example, after /* or after --). No space is permitted between the comment delimiter and the plus sign (+).

  In the following example, there is a space between the star (*) and the plus sign (+), so the hint is ignored:

  ```
  Command> SELECT /* + TT_TblScan (1) This hint is ignored because there is a
           space between the star (*) and the plus (+) sign. */ ...
  ```

- A *hint* is one of the statement level optimizer hints supported by TimesTen. There can be a space between the plus sign (+) and the hint. If the comment contains multiple hints, separate the hints by at least one space. For example, to specify two hints on one line:

  ```
  Command> SELECT --+ TT_MergeJoin (0) TT_NestedLoop (1)
          ...
  ```

- You can intersperse commenting text with hints in a comment. For example,

  ```
  Command> SELECT /*+ TT_HashScan (1) This demonstrates a hint followed by a
           comment string. */ ...
  ```

- TimesTen ignores hints and does not return an error if:
  - Your hint does not follow the DELETE, INSERT, MERGE, SELECT or UPDATE keyword (or for TT_GridQueryExec or TT_PartialResult, the SELECT keyword). TT_CommitDMLOnSuccess must follow the DELETE, INSERT, UPDATE keyword and for INSERT...SELECT, it must follow the SELECT keyword.
  - Your hint contains misspellings or syntax errors. If you have hints that are within the same comment and some hints are correct syntactically and some hints are incorrect syntactically, TimesTen ignores the incorrect hints and accepts the correct hints.
  - You use either the TT_JoinOrder or TT_Index hint and you do not supply a closing parenthesis, the remainder of the hint string is ignored.
- For hints that conflict with each other, TimesTen uses the rightmost hint in the comment. For example, if the comment string is /*+TT_TblScan (0)...TT_TblScan (1) */, the rightmost hint, TT_TblScan(1), is used.
- Statement level optimizer hints override conflicting transaction level optimizer hints. If you specify a transaction level optimizer hint that conflicts with a statement level optimizer hint, the statement level optimizer hint overrides the conflicting transaction level optimizer hint. For example, if you call ttOptSetFlag, and enable the Range flag and then you issue a SQL query and disable the statement level optimizer flag, TT_Range, TimesTen disables the range flag for the query. After the query is executed, the original range flag setting that was in place in the transaction before the query was executed remains in effect for the duration of the transaction. For more information, see Using Statement Level Optimizer Hints for a SELECT Query. The TT_GridQueryExec, TT_PartialResult, TT_CommitDMLOnSuccess, and TT_CountAsInt hints are not supported at the transaction level.
- Do not use statement level optimizer hints in a subquery.
- The TimesTen query optimizer does not recognize statement level optimizer hints for passthrough statements. TimesTen passes the SQL text for passthrough statements to the Oracle database and the SQL text is processed according to the SQL rules of the Oracle database. Passthrough statements are not supported in TimesTen Scaleout.

SQL Statements that Support Statement Level Optimizer Hints

You can specify statement level optimizer hints in SQL statements. Not all hints are supported in all statements. You must specify the hint within comment syntax and the comment syntax must immediately follow the SQL VERB. (For example, SELECT /*+ *hint* */...) Table 6-3 shows the correct placement of the statement level hint. It also indicates if a hint is not supported in the statement.

Table 6-3 Placement of Statement Level Hints in SQL Statements

SQL Statement	Placement of Hint
CREATE TABLE... AS SELECT	CREATE TABLE...AS SELECT [*hint*]... Do not use transaction level hints with the CREATE TABLE...AS SELECT statement. TT_CommitDMLOnSuccess is not supported in this statement.
DELETE	DELETE [*hint*]... The TT_GridQueryExec and TT_PartialResult hints are not supported in this statement.
INSERT	INSERT [*hint*]... The TT_GridQueryExec and TT_PartialResult hints are not supported in this statement.
INSERT...SELECT	INSERT...SELECT [*hint*]...
MERGE	MERGE [*hint*]... The TT_GridQueryExec and TT_PartialResult hints are not supported.
SELECT	SELECT [*hint*]... Do not specify a hint in a subquery. The TT_CommitDMLOnSuccess hint is not supported in this statement.
SELECT...{UNION|MINUS| INTERSECT} SELECT...	SELECT [*hint*] {UNION|MINUS|INTERSECT} SELECT... The TT_CommitDMLOnSuccess hint is not supported in this statement.
UPDATE	UPDATE [*hint*]... The TT_GridQueryExec and TT_PartialResult hints are not supported in this statement.

Understanding Hints

Use optimizer hints to influence the TimesTen query optimizer in determining the choice of the execution plan for your query.

TT_GridQueryExec, TT_PartialResult and TT_CommitDMLOnSuccess are supported at the connection and statement levels only. This section is not valid for these hints.

To view transaction level optimizer hints, execute the built-in procedure, ttOptSetFlag. For more information on the built-in procedure, ttOptGetFlag, see ttOptGetFlag in *Oracle TimesTen In-Memory Database Reference*.

Examples

For TT_CommitDMLOnSuccess examples, see TT_CommitDMLOnSuccess Optimizer Hint for information.

For TT_GridQueryExec and TT_PartialResult examples:

- See TT_GridQueryExec in the *Oracle TimesTen In-Memory Database Scaleout User's Guide*.
- See TT_PartialResult in the *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

The following examples illustrate usages of statement level and transaction level optimizer hints. The TimesTen optimizer is a cost based query optimizer and generates what it thinks is the most optimal execution plan for your statement. This plan differs from release to release. The plan is based on the indexes that exist on the referenced tables as well as the column and table statistics that are available. When you recompute statistics or change indexes, the TimesTen optimizer may change the execution plan based on the recomputed statistics and index changes. Because the execution plan may vary, these examples are included for demonstration purposes only. Examples include:

- Using Statement Level Optimizer Hints for a SELECT Query
- Using On and Off Hinting
- Using TT_JoinOrder to Specify a Join Order
- Using the TT_INDEX Statement Level Optimizer Hint

Using Statement Level Optimizer Hints for a SELECT Query

View the execution plan for a query. Then use statement level optimizer hints to influence the optimizer to choose a different execution plan. Consider the query:

```
Command> SELECT r.region_name, c.country_name
       FROM regions r, countries c
       WHERE r.region_id = c.region_id
       ORDER BY c.region_id;
```

Use the ttIsql EXPLAIN command to view the plan generated by the optimizer. Note:

- The optimizer performs two range scans using table level locking for both scans.
- The optimizer uses the MergeJoin operation to join the two tables.

```
Command> EXPLAIN SELECT r.region_name, c.country_name
       FROM regions r, countries c
       WHERE r.region_id = c.region_id
       ORDER BY c.region_id;

Query Optimizer Plan:

 STEP:             1
 LEVEL:             2
 OPERATION:         TblLkRangeScan
 TBLNAME:          COUNTRIES
 IXNAME:           COUNTR_REG_FK
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        <NULL>


 STEP:             2
 LEVEL:             2
 OPERATION:         TblLkRangeScan
 TBLNAME:          REGIONS
 IXNAME:           REGIONS
 INDEXED CONDITION:   R.REGION_ID >= C.REGION_ID
 NOT INDEXED:        <NULL>

 STEP:             3
 LEVEL:             1
 OPERATION:         MergeJoin
 TBLNAME:          <NULL>
 IXNAME:           <NULL>
```

```
 INDEXED CONDITION:   C.REGION_ID = R.REGION_ID
 NOT INDEXED:         <NULL>
```

Now use statement level optimizer hints to direct the optimizer to perform the scans using row level locking and to use a NestedLoop operation to join the tables. Set autocommit to on to illustrate that the autocommit setting has no effect because statement level optimizer hints are scoped to the SQL statement.

```
Command> autocommit on;
Command> EXPLAIN SELECT /*+ TT_RowLock (1), TT_TblLock (0), TT_MergeJoin (0),
       TT_NestedLoop (1) */
       r.region_name, c.country_name
       FROM regions r, countries c
       WHERE r.region_id = c.region_id
       ORDER BY c.region_id;

Query Optimizer Plan:

 STEP:              1
 LEVEL:             3
 OPERATION:         RowLkRangeScan
 TBLNAME:           REGIONS
 IXNAME:            REGIONS
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:         <NULL>

 STEP:              2
 LEVEL:             3
 OPERATION:         RowLkRangeScan
 TBLNAME:           COUNTRIES
 IXNAME:            COUNTR_REG_FK
 INDEXED CONDITION:   C.REGION_ID = R.REGION_ID
 NOT INDEXED:         <NULL>

 STEP:              3
 LEVEL:             2
 OPERATION:         NestedLoop
 TBLNAME:           <NULL>
 IXNAME:            <NULL>
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:         <NULL>

 STEP:              4
 LEVEL:             1
 OPERATION:         OrderBy
 TBLNAME:           <NULL>
 IXNAME:            <NULL>
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:         <NULL>
```

Prepare the query again without statement level optimizer hints. The optimizer reverts back to the original execution plan because statement level optimizer hints are scoped to the SQL statement.

```
Command> EXPLAIN SELECT r.region_name, c.country_name
       FROM regions r, countries c
       WHERE r.region_id = c.region_id
       ORDER BY c.region_id;

Query Optimizer Plan:

 STEP:              1
```

```
 LEVEL:           2
 OPERATION:       TblLkRangeScan
 TBLNAME:         COUNTRIES
 IXNAME:          COUNTR_REG_FK
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:     <NULL>

 STEP:            2
 LEVEL:           2
 OPERATION:       TblLkRangeScan
 TBLNAME:         REGIONS
 IXNAME:          REGIONS
 INDEXED CONDITION:   R.REGION_ID >= C.REGION_ID
 NOT INDEXED:     <NULL>

 STEP:            3
 LEVEL:           1
 OPERATION:       MergeJoin
 TBLNAME:         <NULL>
 IXNAME:          <NULL>
 INDEXED CONDITION:   C.REGION_ID = R.REGION_ID
 NOT INDEXED:     <NULL>
```

Using On and Off Hinting

This example illustrates the importance of directing the optimizer to specifically enable or disable hints that perform a similar function. For example, the hash and range hints direct the optimizer to use either a hash or range access path for the table. In order to ensure the optimizer chooses the specific access path, enable one hint and disable all other related hints.

Create a table and create a hash index on the first column of the table and a range index on the second column.

```
Command> CREATE TABLE test (col1 NUMBER, col2 NUMBER);
Command> CREATE HASH INDEX h_index ON test (col1);
Command> CREATE INDEX hr_index ON test (col2);
```

Set autocommit to off and execute the built-in procedure, `ttOptGetFlag`, to review the current transaction level optimizer hint settings for the transaction. A setting of 1 means the flag is enabled.

```
Command> autocommit off;
Command> CALL ttOptGetFlag ('Hash');
< Hash, 1 >
1 row found.
Command> CALL ttOptGetFlag ('Scan');
< Scan, 1 >
1 row found.
```

Use the `ttIsql` `EXPLAIN` command to review the plan for a `SELECT` query using a `WHERE` clause and dynamic parameters. The optimizer uses a hash scan.

```
Command> EXPLAIN SELECT * FROM test WHERE col1 = ? and col2 = ?;

Query Optimizer Plan:

 STEP:            1
 LEVEL:           1
 OPERATION:       RowLkHashScan
 TBLNAME:         TEST
 IXNAME:          H_INDEX
```

```
 INDEXED CONDITION:   TEST.COL1 = _QMARK_1
 NOT INDEXED:         TEST.COL2 = _QMARK_2
```

Use the statement level optimizer hint TT_Range to direct the optimizer to use a range scan. Note that the optimizer ignores the TT_Range hint and uses a hash scan because you did not direct the optimizer to disable the hash scan. Alter the statement and direct the optimizer to use a range scan and not use a hash scan. To accomplish this, enable the statement level optimizer hint TT_Range and disable the statement level optimizer hint TT_HashScan. The optimizer no longer ignores the TT_Range hint.

```
Command> EXPLAIN SELECT --+ TT_Range (1) Single line comment to set TT_Range
       * FROM TEST WHERE col1 = ? and col2 = ?;

Query Optimizer Plan:

 STEP:             1
 LEVEL:            1
 OPERATION:        RowLkHashScan
 TBLNAME:          TEST
 IXNAME:           H_INDEX
 INDEXED CONDITION:   TEST.COL1 = _QMARK_1
 NOT INDEXED:         TEST.COL2 = _QMARK_2

Command> EXPLAIN SELECT /*+ TT_Range (1) TT_HashScan (0)
        Multiple line comment to enable TT_Range and disable TT_HashScan */
        * FROM TEST WHERE col1 = ? and col2 = ?;

Query Optimizer Plan:

 STEP:             1
 LEVEL:            1
 OPERATION:        RowLkRangeScan
 TBLNAME:          TEST
 IXNAME:           HR_INDEX
 INDEXED CONDITION:   TEST.COL2 = _QMARK_2
 NOT INDEXED:         TEST.COL1 = _QMARK_1
```

Prepare the query again without using statement level optimizer hints and without issuing a commit or rollback. The optimizer uses the transaction level optimizer hints settings that were in effect before executing the query. The optimizer uses transaction level optimizer hints because statement level optimizer hints are scoped to the SQL statement.

```
Command> EXPLAIN SELECT * FROM TEST WHERE col1 = ? and col2 = ?;

Query Optimizer Plan:

 STEP:             1
 LEVEL:            1
 OPERATION:        RowLkHashScan
 TBLNAME:          TEST
 IXNAME:           H_INDEX
 INDEXED CONDITION:   TEST.COL1 = _QMARK_1
 NOT INDEXED:         TEST.COL2 = _QMARK_2
```

Using TT_JoinOrder to Specify a Join Order

Use the statement level optimizer hint TT_JoinOrder to direct the optimizer to use a specific join order. First use a transaction level optimizer hint to direct the optimizer to use a specific join order for the transaction. Then use a statement level optimizer hint to direct the optimizer to change the join order for the statement only.

```
Command> CALL ttOptSetOrder ('e d j');
Command> EXPLAIN SELECT *
       FROM employees e, departments d, job_history j
       WHERE e.department_id = d.department_id
       AND e.hire_date = j.start_date;

Query Optimizer Plan:

 STEP:             1
 LEVEL:            3
 OPERATION:        TblLkRangeScan
 TBLNAME:          EMPLOYEES
 IXNAME:           EMP_DEPT_FK
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:      <NULL>

 STEP:             2
 LEVEL:            3
 OPERATION:        TblLkRangeScan
 TBLNAME:          DEPARTMENTS
 IXNAME:           DEPARTMENTS
 INDEXED CONDITION:   D.DEPARTMENT_ID >= E.DEPARTMENT_ID
 NOT INDEXED:      <NULL>

 STEP:             3
 LEVEL:            2
 OPERATION:        MergeJoin
 TBLNAME:          <NULL>
 IXNAME:           <NULL>
 INDEXED CONDITION:   E.DEPARTMENT_ID = D.DEPARTMENT_ID
 NOT INDEXED:      <NULL>

 STEP:             4
 LEVEL:            2
 OPERATION:        TblLkRangeScan
 TBLNAME:          JOB_HISTORY
 IXNAME:           JOB_HISTORY
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:      E.HIRE_DATE = J.START_DATE

 STEP:             5
 LEVEL:            1
 OPERATION:        NestedLoop
 TBLNAME:          <NULL>
 IXNAME:           <NULL>
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:      <NULL>
```

Use the statement level optimizer hint, `TT_JoinOrder`, to direct the optimizer to override the transaction level join order optimizer hint for the SQL statement only.

```
Command> EXPLAIN SELECT --+ TT_JoinOrder (e j d)
        *
        FROM employees e, departments d, job_history j
        WHERE e.department_id = d.department_id
        AND e.hire_date = j.start_date;

Query Optimizer Plan:

 STEP:             1
 LEVEL:            3
 OPERATION:        TblLkRangeScan
```

```
 TBLNAME:          EMPLOYEES
 IXNAME:           EMP_DEPT_FK
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        <NULL>

 STEP:             2
 LEVEL:             3
 OPERATION:         TblLkRangeScan
 TBLNAME:          JOB_HISTORY
 IXNAME:           JOB_HISTORY
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        E.HIRE_DATE = J.START_DATE

 STEP:             3
 LEVEL:             2
 OPERATION:         NestedLoop
 TBLNAME:          <NULL>
 IXNAME:           <NULL>
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        <NULL>

 STEP:             4
 LEVEL:             2
 OPERATION:         TblLkRangeScan
 TBLNAME:          DEPARTMENTS
 IXNAME:           DEPARTMENTS
 INDEXED CONDITION:   D.DEPARTMENT_ID >= E.DEPARTMENT_ID
 NOT INDEXED:        <NULL>

 STEP:             5
 LEVEL:             1
 OPERATION:         MergeJoin
 TBLNAME:          <NULL>
 IXNAME:           <NULL>
 INDEXED CONDITION:   E.DEPARTMENT_ID = D.DEPARTMENT_ID
 NOT INDEXED:        <NULL>
```

Prepare the query again to verify that the join order that was in effect for the transaction remains in effect.

```
Command> EXPLAIN SELECT *
       FROM employees e, departments d, job_history j
       WHERE e.department_id = d.department_id
       AND e.hire_date = j.start_date;

Query Optimizer Plan:

 STEP:             1
 LEVEL:             3
 OPERATION:         TblLkRangeScan
 TBLNAME:          EMPLOYEES
 IXNAME:           EMP_DEPT_FK
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        <NULL>

 STEP:             2
 LEVEL:             3
 OPERATION:         TblLkRangeScan
 TBLNAME:          DEPARTMENTS
 IXNAME:           DEPARTMENTS
 INDEXED CONDITION:   D.DEPARTMENT_ID >= E.DEPARTMENT_ID
 NOT INDEXED:        <NULL>
```

```
 STEP:              3
 LEVEL:               2
 OPERATION:          MergeJoin
 TBLNAME:            <NULL>
 IXNAME:             <NULL>
 INDEXED CONDITION:   E.DEPARTMENT_ID = D.DEPARTMENT_ID
 NOT INDEXED:        <NULL>

 STEP:              4
 LEVEL:               2
 OPERATION:          TblLkRangeScan
 TBLNAME:            JOB_HISTORY
 IXNAME:             JOB_HISTORY
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        E.HIRE_DATE = J.START_DATE

 STEP:              5
 LEVEL:               1
 OPERATION:          NestedLoop
 TBLNAME:            <NULL>
 IXNAME:             <NULL>
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:        <NULL>
```

Using the TT_INDEX Statement Level Optimizer Hint

Perform a query on the employees table that uses the index, emp_name_ix. Then use the statement level optimizer hint TT_INDEX to direct the optimizer not to use this index. First run the ttIsql command, indexes, to view the indexes for the employees table.

```
Command> indexes employees;

Indexes on table TESTUSER.EMPLOYEES:
 EMPLOYEES: unique range index on columns:
  EMPLOYEE_ID
  (referenced by foreign key index JHIST_EMP_FK on table TESTUSER.JOB_HISTORY)
 TTUNIQUE_0: unique range index on columns:
  EMAIL
 EMP_DEPT_FK: non-unique range index on columns:
  DEPARTMENT_ID
  (foreign key index references table TESTUSER.DEPARTMENTS(DEPARTMENT_ID))
 EMP_JOB_FK: non-unique range index on columns:
  JOB_ID
  (foreign key index references table TESTUSER.JOBS(JOB_ID))
 EMP_NAME_IX: non-unique range index on columns:
  LAST_NAME
  FIRST_NAME
 5 indexes found.

5 indexes found on 1 table.
```

Use the ttIsql command, EXPLAIN, to view the execution plan for a SELECT query on the employees table that uses a WHERE clause on the last_name column.

```
Command> EXPLAIN SELECT e.first_name
       FROM employees e
       WHERE e.last_name BETWEEN 'A' AND 'B';

Query Optimizer Plan:

 STEP:              1
```

```
 LEVEL:            1
 OPERATION:        RowLkRangeScan
 TBLNAME:          EMPLOYEES
 IXNAME:           EMP_NAME_IX
 INDEXED CONDITION:   E.LAST_NAME >= 'A' AND E.LAST_NAME <= 'B'
 NOT INDEXED:      <NULL>
```

Use the statement level optimizer hint, TT_INDEX, to direct the optimizer not to use the index, emp_name_ix.

```
Command> EXPLAIN SELECT --+ TT_INDEX (E,EMP_NAME_IX,0)
         e.first_name
         FROM employees e
         WHERE e.last_name BETWEEN 'A' AND 'B';

Query Optimizer Plan:

 STEP:             1
 LEVEL:            1
 OPERATION:        TblLkRangeScan
 TBLNAME:          EMPLOYEES
 IXNAME:           EMPLOYEES
 INDEXED CONDITION:   <NULL>
 NOT INDEXED:      E.LAST_NAME <= 'B' AND E.LAST_NAME >= 'A'
```

Optimizer Hints Supported in TimesTen Scaleout Only

These optimizer hints are only supported in TimesTen Scaleout. They are valid at the statement and at the connection levels.

- TT_GridQueryExec Optimizer Hint
- TT_PartialResult Optimizer Hint
- TT_CommitDMLOnSuccess Optimizer Hint

See OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for information on hints at the connection level and "Statement Level Optimizer Hints" in this book for information on statement level optimizer hints.

TT_GridQueryExec Optimizer Hint

The TT_GridQueryExec optimizer hint enables you to specify whether the query should return data from the local element or from all elements, including the elements in a replica set when K-safety is set to 2.

If you do not specify this hint, the query is executed in one logical data space. It is neither local nor global. This means that exactly one full copy of the data is used to compute the query. Use this hint in cases where obtaining some result is more important than obtaining the correct result (for example, where one or more replica sets are unavailable). Valid options for this hint are LOCAL and GLOBAL.

For more information, see:

- TT_GridQueryExec in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for information on using this hint.
- OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for information on using this hint at the connection level.
- "Statement Level Optimizer Hints" for information on using this hint at the statement level.

This example illustrates how to use the TT_GridQueryExec(GLOBAL) hint on the dual table to determine the ids of all elements, replica sets, and dataspaces.

```
Command> SELECT /*+TT_GridQueryExec(GLOBAL)*/ elementId#, replicasetId#,
         dataspaceId# FROM dual ORDER BY elementId#,replicasetId#,dataspaceId#;

ELEMENTID#, REPLICASETID#, DATASPACEID#
< 1, 1, 1 >
< 2, 1, 2 >
< 3, 2, 1 >
< 4, 2, 2 >
< 5, 3, 1 >
< 6, 3, 2 >
6 rows found.
```

See TT_GridQueryExec in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for more examples.

TT_PartialResult Optimizer Hint

The TT_PartialResult optimizer hint enables you to specify whether the query should return partial results if some data is not available.

Use TT_PartialResult(1) to direct the query to return partial results if all elements in a replica set are not available.

Use TT_PartialResult(0) to direct the query to return an error if the required data is not available in the case where all elements in a replica set are not available. If at least one element from each replica set is available or the data required by the query is available, the optimizer returns the query result correctly without error.

The default is TT_PartialResult(0).

For more information, see:

- TT_PartialResult in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for information on using this hint and for examples.
- OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for information on using this hint at the connection level.
- "Statement Level Optimizer Hints" for information on using this hint at the statement level.

TT_CommitDMLOnSuccess Optimizer Hint

Use the TT_CommitDMLOnSuccess hit to enable or disable a commit operation as part of DML execution.

- At the statement level, TT_CommitDMLOnSuccess is used in a DML statement (DELETE, INSERT, INSERT... SELECT, and UPDATE) to enable or disable the commit behavior of the transaction when the DML operation is executed. For the INSERT...SELECT statement, specify TT_CommitDMLOnSuccess after the SELECT keyword.

  TT_CommitDMLOnSuccess is valid in DML operations only. It is not valid for queries or DDL operations and, if specified in a non-DML statement, is ignored and no error is returned. See Statement Level Optimizer Hints for information on the syntax and semantics.
- At the connection level, TT_CommitDMLOnSuccess is also used to enable or disable the commit behavior of the transaction when a DML operation is executed. However, you specify TT_CommitDMLOnSuccess as a parameter to the OptimizerHint connection attribute. See

OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for information on using TT_CommitDMLOnSuccess at the connection level.

At both levels, valid options are 0 and 1. If you do not specify TT_CommitDMLOnSuccess, there are no changes to the normal commit behavior. The order of precedence is statement level followed by connection level.

The TT_CommitDMLOnSuccess commit behavior at the statement level is:

- TT_CommitDMLOnSuccess(1) commits the current transaction if the DML statement in which the hint is specified is executed successfully. If there are open cursors at commit time, all cursors are closed and the transaction is committed. If the statement with this hint fails, the transaction is not committed.
- TT_CommitDMLOnSuccess(0) disables the commit of the current transaction if the DML statement in which the hint is specified is executed successfully.

Table 6-4 shows the commit behavior when not setting TT_CommitDMLOnSuccess as well as setting TT_CommitDMLOnSuccess to 0 and 1 at the statement and connection levels. The table shows the commit behavior when autocommit is set to 0.

Table 6-5 shows the commit behavior when not setting TT_CommitDMLOnSuccess as well as setting TT_CommitDMLOnSuccess to 0 and 1 at the statement and connection levels. The table shows the commit behavior when autocommit is set to 1.

Table 6-4 TT_CommitDMLOnSuccess Commit Behavior: Autocommit 0

Blank	Not Set at Connection Level	Set to 0 at Connection Level	Set to 1 at Connection Level
Not set at statement level	• If there are no open cursors, and DML execution is successful, the transaction is not committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is not committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed due to the connection level setting of 1. • If there are open cursors and DML execution is successful, the cursors are closed and the transaction is committed due to the connection level setting of 1. • If DML execution is not successful, the transaction is not committed.

Table 6-4 (Cont.) TT_CommitDMLOnSuccess Commit Behavior: Autocommit 0

Blank	Not Set at Connection Level	Set to 0 at Connection Level	Set to 1 at Connection Level
Set to 0 at statement level	• If there are no open cursors, and DML execution is successful, the transaction is not committed up until the DML statement that includes the hint due to the autocommit 0 setting. This DML statement is not committed due to disabling the commit behavior at the statement level. (This is the same behavior as when TT_CommitDMLOnSuccess is not set at the statement level.) • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. (This is the same behavior as when TT_CommitDMLOnSuccess is not set at the statement level.) • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is not committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed up until the DML statement that includes the hint. This DML statement is not committed due to disabling the commit behavior at the statement level. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.
Set to 1 at statement level	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.

Table 6-5 TT_CommitDMLOnSuccess Commit Behavior: Autocommit 1

Blank	Not Set at Connection Level	Set to 0 at Connection Level	Set to 1 at Connection Level
Not Set at Statement Level	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, the cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.
Set to 0 at Statement Level	• If there are no open cursors, and DML execution is successful, the transaction is committed. (This is the same behavior as when TT_CommitDMLOnSuccess is not set at the statement level.) • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. (This is the same behavior as when TT_CommitDMLOnSuccess is not set at the statement level.) • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, the cursors are not closed and the transaction is not committed. • If DML execution is not successful, the transaction is not committed.

Table 6-5 (Cont.) TT_CommitDMLOnSuccess Commit Behavior: Autocommit 1

Blank	Not Set at Connection Level	Set to 0 at Connection Level	Set to 1 at Connection Level
Set to 1 at Statement Level	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.	• If there are no open cursors, and DML execution is successful, the transaction is committed. • If there are open cursors and DML execution is successful, cursors are closed and the transaction is committed. • If DML execution is not successful, the transaction is not committed.

For more information, see:

- Using the TT_CommitDMLOnSuccess Hint in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for additional information.
- OptimizerHint in the *Oracle TimesTen In-Memory Database Reference* for information on using TT_CommitDMLOnSuccess at the connection level.
- Statement Level Optimizer Hints for information on the syntax for TT_CommitDMLOnSuccess at the statement level.

These examples illustrate the use of the TT_CommitDMLOnSuccess optimizer hint:

- Setting TT_CommitDMLOnSuccess to 1
- Using TT_CommitDMLOnSuccess at Connection Level

Setting TT_CommitDMLOnSuccess to 1

This example first creates the mytable table. It then sets autocommit to 0 and inserts a row into the mytable table. A second connection (conn2) connects to the database and issues a SELECT query against the mytable table. The query returns 0 rows. The ttIsql use command returns the application to the first connection (database1) and issues a second INSERT operation, setting TT_CommitDMLOnSuccess to 1 at the statement level. A second ttIsql use command returns the application to the conn2 connection. A SELECT query shows two rows have been inserted into the mytable table. This example illustrates that issuing TT_CommitDMLOnSuccess(1) commits the transaction after the successful execution of the second INSERT operation (which set the hint).

```
Command> CREATE TABLE mytable (col1 TT_INTEGER, col2 VARCHAR2(4000));
Command> autocommit 0;
Command> INSERT INTO mytable VALUES (10, 'ABC');
1 row inserted.
```

Establish a second connection (conn2)

```
Command> connect as conn2;
Using the connection string of connection database1 to connect...
```

```
...
(Default setting AutoCommit=1)
```

Issue a SELECT query and expect 0 rows due to autocommit set to 0.

```
conn2: Command> SELECT * FROM mytable;
0 rows found.
```

Return to the first connection (database1) and issue an INSERT operation with TT_CommitDMLOnSuccess set to 1.

```
conn2: Command> use database1;
database1: Command> INSERT /*+TT_CommitDMLOnSuccess(1)*/
                 INTO mytable VALUES (10, 'ABC');
1 row inserted.
```

Return to the second connection (conn2) and issue a SELECT query. Expect 2 rows (due to the two INSERT statements. (The transaction is committed due to the TT_CommitDMLOnSuccess statement level hint set to 1 and the successful execution of the two INSERT operations.)

```
database1: Command> use conn2
conn2: Command> SELECT * FROM mytable;
< 10, ABC >
< 10, ABC >
2 rows found.
```

Using TT_CommitDMLOnSuccess at Connection Level

This example first creates the mytable table. It then uses PL/SQL to insert 1000 rows into the table. There is a second connection to the database (conn2) and this connection connects with TT_CommitDMLOnSuccess set to 1 at the connection level. Various operations are performed to illustrate the behavior of TT_CommitDMLOnSuccess at both the statement and connection levels.

```
Command>  CREATE TABLE mytable (col1 TT_INTEGER NOT NULL PRIMARY KEY,
          col2 VARCHAR2 (4000));
Command> BEGIN
         FOR i in 1..1000
         LOOP
          INSERT INTO mytable VALUES (i,i);
         END LOOP;
       END;
       /
PL/SQL procedure successfully completed.
```

Establish a second connection (conn2) and connect setting TT_CommitDMLOnSuccess at the connection level to 1.

```
Command> CONNECT adding "OptimizerHint=TT_CommitDMLOnSuccess(1)" as conn2;
Connection successful:
...
```

Set autocommit to 0 and issue a DELETE operation.

```
conn2: Command> autocommit 0;
conn2: Command> DELETE FROM mytable WHERE col1=1000;
1 row deleted.
```

Return to the original connection (database1) and issue a SELECT query to see if the DELETE operation was committed. The operation was committed due to the TT_CommitDMLOnSuccess setting of 1 at the connection level.

```
conn2: Command> use database1;
database1: Command> SELECT * FROM mytable WHERE col1=1000;
0 rows found.
```

Return to the second connection (conn2) and issue an INSERT operation. Then return to the original connection (database1). The transaction containing the INSERT operation was committed.

```
database1: Command> use conn2;
conn2: Command> INSERT INTO mytable VALUES (1000,1000);
1 row inserted.
conn2: Command> use database1
database1: Command> SELECT * FROM mytable WHERE col1=1000;
< 1000, 1000 >
1 row found.
```

Return to the second connection (conn2) and issue a DELETE operation, followed by an INSERT operation, and then a second INSERT operation where TT_CommitDMLOnSuccess is set to 0 at the statement level (the second INSERT).

```
database1: Command> use conn2;
conn2: Command> DELETE FROM mytable WHERE col1=1000;
1 row deleted.
conn2: Command> INSERT INTO mytable VALUES (1001,1001);
1 row inserted.
conn2: Command> INSERT /*+TT_CommitDMLOnSuccess(0)*/ INTO mytable
        VALUES (1002,1002);
1 row inserted.
```

Issue a SELECT query and notice the results of the query. The one DELETE operation and the two INSERT operations were successful.

```
conn2: Command> SELECT * FROM mytable where col1 >= 1000;
< 1001, 1001 >
< 1002, 1002 >
2 rows found.
```

Return to the original connection (database1) and issue the same SELECT query. Observe that the one DELETE statement and the first INSERT operation were committed. This is due to the TT_CommitDMLOnSuccess setting of 1 at the connection level. The second INSERT statement was not committed due to the TT_CommitDMLOnSuccess setting of 0 for this second INSERT statement.

```
conn2: Command> use database1;
database1: Command> SELECT * FROM mytable where col1 >= 1000;
< 1001, 1001 >
1 row found.
```

Return to the second connection (conn2) and issue a third INSERT operation. Then issue a SELECT query and observe the results.

```
database1: Command> use conn2;
conn2: Command> INSERT INTO mytable VALUES (1003,1003);
1 row inserted.
conn2: Command> SELECT * FROM mytable where col1 >= 1000 ORDER BY col1;
< 1001, 1001 >
< 1002, 1002 >
< 1003, 1003 >
3 rows found.
```

Return to the original connection (database1) and issue the same SELECT query. Note the results are the same as in the conn2 connection. The transaction is committed due to the

TT_CommitDMLOnSuccess setting of 1 at the connection level and the successful execution of the second and third INSERT operations.

```
conn2: Command> use database1
database1: Command> SELECT * FROM mytable where col1 >= 1000 ORDER BY col1;
< 1001, 1001 >
< 1002, 1002 >
< 1003, 1003 >
3 rows found.
```

ALTER ACTIVE STANDBY PAIR

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

You can change an active standby pair by:

- Adding or dropping a subscriber database
- Altering store attributes

  Only the PORT and TIMEOUT attributes can be set for subscribers.
- Including tables, sequences or cache groups in the replication scheme
- Excluding tables, sequences or cache groups from the replication scheme

See Making Other Changes to an Active Standby Pair in *Oracle TimesTen In-Memory Database Replication Guide*.

Required Privilege

ADMIN

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL Syntax

```
ALTER ACTIVE STANDBY PAIR {
    SubscriberOperation |
    StoreOperation | InclusionOperation |
    NetworkOperation } [...];
```

Syntax for *SubscriberOperation*:

```
{ADD | DROP } SUBSCRIBER FullStoreName
```

Syntax for *StoreOperation*:

```
ALTER STORE FullStoreName SET StoreAttribute
```

Syntax for *InclusionOperation*:

```
  [{ INCLUDE | EXCLUDE }{TABLE [[Owner.]TableName [,...]]|
      CACHE GROUP [[Owner.]CacheGroupName [,...]]|
      SEQUENCE [[Owner.]SequenceName [,...]]} [,...]]
```

Syntax for *NetworkOperation*:

ADD ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*
 { { MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost* }
  PRIORITY *Priority* } [...]
DROP ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*
 { MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost* } [...]

Parameters

Parameter	Description
ADD SUBSCRIBER *FullStoreName*	Indicates a subscriber database. *FullStoreName* is the database 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 database. This operation fails if the replication scheme has only one subscriber. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.
ALTER STORE *FullStoreName* SET *StoreAttribute*	Indicates changes to the attributes of a database. Only the PORT and TIMEOUT attributes can be set for subscribers. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description. See CREATE ACTIVE STANDBY PAIR For information on *StoreAttribute* clauses.
FullStoreName	The database, specified as one of the following: • SELF • The prefix of the database file name For example, if the database path is *directory*/*subdirectory*/data.ds0, then data is the database name that should be used. This is the database 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 the Network in *Oracle TimesTen In-Memory Database Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: "MyHost-500".
{INCLUDE|EXCLUDE} {[TABLE [*Owner*.]*TableName*[,...]]| CACHE GROUP [[*Owner*.]*CacheGroupName*]|[,...] SEQUENCE [[*Owner*.]*SequenceName* [,...]]} [,...]	Includes in or excludes from replication the tables, sequences or cache groups listed. INCLUDE adds the tables, sequences or cache groups to the replication scheme. 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). You cannot use the EXCLUDE clause for AWT cache groups.

Parameter	Description
ADD ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Adds *NetworkOperation* to replication scheme. Enables you to control the network interface that a master store uses for every outbound connection to each of its subscriber stores. In the context of the ADD ROUTE clause, each master database is a subscriber of the other master database and each read-only subscriber is a subscriber of both master databases. Can be specified more than once. For *FullStoreName*, "ON *host*" must be specified.
DROP ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Drops *NetworkOperation* from replication scheme. Can be specified more than once. For *FullStoreName*, "ON *host*" must be specified.
MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost*	*MasterHost* and *SubscriberHost* are the IP addresses for the network interface on the master and subscriber stores. Specify in dot notation or canonical format or in colon notation for IPV6. Clause can be specified more than once. Valid for both ADD and DROP ROUTE MASTER.
PRIORITY *Priority*	Variable expressed as an integer from 1 to 99. Denotes the priority of the IP address. Lower integral values have higher priority. An error is returned if multiple addresses with the same priority are specified. Controls the order in which multiple IP addresses are used to establish peer connections. Required syntax of *NetworkOperation* clause. Follows MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost* clause.

Description

- You must stop the replication agent before altering an active standby pair. The exceptions are for those objects and statements that are automatically replicated and included based on the values of the DDL_REPLICATION_LEVEL and DDL_REPLICATION_ACTION attributes. See ALTER SESSION for more information.
- You can only alter the active standby pair replication scheme on the active database. See Making Other Changes to an Active Standby Pair in *Oracle TimesTen In-Memory Database Replication Guide* for more information.
- Do not use ALTER ACTIVE STANDBY PAIR when using Oracle Clusterware with TimesTen. See Restricted Commands and SQL Statements in *Oracle TimesTen In-Memory Database Replication Guide* for more information.

  Instead, perform the tasks described in Changing the Schema section of the *Oracle TimesTen In-Memory Database Replication Guide*.
- 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 the INCLUDE or EXCLUDE clause to include the listed tables, sequences or cache groups in the replication scheme or to exclude them from the replication scheme. Use one INCLUDE or EXCLUDE clause for each object type (table, sequence or cache group). The ALTER ACTIVE STANDBY statement is not necessary for those objects and statements that are automatically replicated and included based on the values of the

DDL_REPLICATION_LEVEL and DDL_REPLICATION_ACTION attributes. See ALTER SESSION for more information. However, if DDL_REPLICATION_LEVEL is 2 or greater and DDL_REPLICATION_ACTION="EXCLUDE", use the INCLUDE clause to include replicated objects into the replication scheme.

- Do not use the EXCLUDE clause for AWT cache groups.
- When DDL_REPLICATION_LEVEL is 2 or greater, the INCLUDE clause can only be used with empty tables on the active database. The contents of the corresponding tables on the standby and any subscribers will be truncated before the table is added to the replication scheme.

Examples

Add a subscriber to the replication scheme.

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

Drop two subscribers from the replication scheme.

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

Alter the store attributes of the rep3 and rep4 databases.

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

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;
```

Add *NetworkOperation* clause to active standby pair:

```
ALTER ACTIVE STANDBY PAIR
ADD ROUTE MASTER rep1 ON "machine1" SUBSCRIBER rep2 ON "machine2"
MASTERIP "1.1.1.1" PRIORITY 1 SUBSCRIBERIP "2.2.2.2" PRIORITY 1;
```

See Also

CREATE ACTIVE STANDBY PAIR
DROP ACTIVE STANDBY PAIR

ALTER CACHE GROUP

The ALTER CACHE GROUP statement modifies the state, interval and mode of AUTOREFRESH for a cache group.

Updates on the Oracle Database tables 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 defined as READONLY with an AUTOREFRESH clause.

Any values or states set by ALTER CACHE GROUP are persistent. They are stored in the database and survive daemon and cache agent restarts.

Required Privilege

No privilege is required for the cache group owner.

ALTER ANY CACHE GROUP for another user's cache group.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

This statement changes the AUTOREFRESH mode of the cache group, which determines which rows are updated during an autorefresh operation. You cannot use the ALTER CACHE GROUP...SET AUTOREFRESH MODE clause in TimesTen Scaleout.

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

This statement changes the AUTOREFRESH interval on the cache group. You cannot use the ALTER CACHE GROUP...SET AUTOREFRESH INTERVAL clause in TimesTen Scaleout.

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

This statement alters the AUTOREFRESH state:

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

Parameters

Parameter	Description
[*Owner*.]*CacheGroupName*	Name assigned to the new cache group.
AUTOREFRESH	Indicates that changes to the Oracle Database tables should be automatically propagated to TimesTen.
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 the Oracle Database 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*	An integer value that specifies how often AUTOREFRESH should be scheduled, in minutes, seconds or milliseconds. The default value is five minutes. An autorefresh interval set to 0 milliseconds enables continuous autorefresh, where the next autorefresh cycle is scheduled immediately after the last autorefresh cycle has ended. See Automatically Refreshing a Cache Group in the *Oracle TimesTen In-Memory Database Cache Guide* for more information. 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 support log if the wait queue reaches 10.

Parameter	Description
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 canceled, 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 canceled, 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 ALTER CACHE GROUP...SET AUTOREFRESH STATE. 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 operation is part of the transaction, but the refresh itself is not.
- If you issue an ALTER CACHE GROUP... SET AUTOREFRESH STATE OFF statement and there is an autorefresh operation currently running, then:
  - If LockWait interval is 0, the ALTER statement fails with a lock timeout error.
  - If LockWait interval is nonzero, then the current autorefresh transaction is 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.
- If the ALTER CACHE GROUP statement is part of a transaction that is being replicated, and if the replication scheme has the RETURN TWOSAFE attribute, the transaction may fail.
- You cannot execute the ALTER CACHE GROUP statement when performed under the serializable isolation level. An error message is returned when attempted.

See Also

CREATE CACHE GROUP

ALTER FUNCTION

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The ALTER FUNCTION statement recompiles a standalone stored function. Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

To recompile a function that is part of a package, recompile the package using the ALTER PACKAGE statement.

Required privilege

No privilege is required for the PL/SQL function owner.

ALTER ANY PROCEDURE for another user's function.

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
ALTER FUNCTION [Owner.]FunctionName COMPILE
    [CompilerParametersClause [...]]
    [REUSE SETTINGS]
```

Parameters

Parameter	Description
[*Owner*.]*FunctionName*	Name of the function to be recompiled.
COMPILE	Required keyword that causes recompilation of the function. If the function does not compile successfully, use the ttIsql command SHOW ERRORS to display the compiler error messages.
CompilerParametersClause	Use this optional clause to specify a value for one of the PL/SQL persistent compiler parameters. The PL/SQL persistent compiler parameters are PLSQL_OPTIMIZE_LEVEL and NLS_LENGTH_SEMANTICS. You can specify each parameter once in the statement. If you omit a parameter from this clause and you specify REUSE SETTINGS, then if a value was specified for the parameter in an earlier compilation, TimesTen uses that earlier value. If you omit a parameter and either you do not specify REUSE SETTINGS or no value has been specified for the parameter in an earlier compilation, then TimesTen obtains the value for the parameter from the session environment.
REUSE SETTINGS	Use this optional clause to prevent TimesTen from dropping and reacquiring compiler switch settings. When you specify REUSE SETTINGS, TimesTen preserves the existing settings and uses them for the compilation of any parameters for which values are not specified.

Description

- The ALTER FUNCTION statement does not change the declaration or definition of an existing function. To redeclare or redefine a function, use the CREATE FUNCTION statement.
- TimesTen first recompiles objects upon which the function depends, if any of those objects are invalid.
- TimesTen also invalidates any objects that depend on the function, such as functions that call the recompiled function or package bodies that define functions that call the recompiled function.
- If TimesTen recompiles the function successfully, then the function becomes valid. If recompiling the function results in compilation errors, then TimesTen returns an error and

the function remains invalid. Use the ttIsql command SHOW ERRORS to display compilation errors.

- During recompilation, TimesTen drops all persistent compiler settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the REUSE SETTINGS clause.

See also

CREATE FUNCTION

ALTER PACKAGE

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The ALTER PACKAGE statement explicitly recompiles a package specification, package body, or both. Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors.

This statement recompiles all package objects together. You cannot use the ALTER PROCEDURE or ALTER FUNCTION statement to individually recompile a procedure or function that is part of a package.

Required privilege

No privilege is required for the package owner.

ALTER ANY PROCEDURE for another user's package.

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
ALTER PACKAGE [Owner.]PackageName COMPILE
    [PACKAGE|SPECIFICATION|BODY]
    [CompilerParametersClause [...]]
    [REUSE SETTINGS]
```

Parameters

Parameter	Description
[Owner.]PackageName	Name of the package to be recompiled.
COMPILE	Required clause used to force the recompilation of the package specification, package body, or both.
[PACKAGE|SPECIFICATION|BODY]	Specify PACKAGE to recompile both the package specification and the body. Specify SPECIFICATION to recompile the package specification. Specify BODY to recompile the package body. PACKAGE is the default.

Parameter	Description
CompilerParametersClause	Use this optional clause to specify a value for one of the PL/SQL persistent compiler parameters. The PL/SQL persistent compiler parameters are PLSQL_OPTIMIZE_LEVEL and NLS_LENGTH_SEMANTICS. You can specify each parameter once in the statement. If you omit a parameter from this clause and you specify REUSE SETTINGS, then if a value was specified for the parameter in an earlier compilation, TimesTen uses that earlier value. If you omit a parameter and either you do not specify REUSE SETTINGS or no value has been specified for the parameter in an earlier compilation, then TimesTen obtains the value for the parameter from the session environment.
REUSE SETTINGS	Use this optional clause to prevent TimesTen from dropping and reacquiring compiler switch settings. When you specify REUSE SETTINGS, TimesTen preserves the existing settings and uses them for the compilation of any parameters for which values are not specified.

Description

- When you recompile a package specification, TimesTen invalidates local objects that depend on the specification, such as procedures that call procedures or functions in the package. The body of the package also depends on the specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, then TimesTen recompiles it implicitly at runtime.
- When you recompile a package body, TimesTen does not invalidate objects that depend on the package specification. TimesTen first recompiles objects upon which the body depends, if any of those objects are invalid. If TimesTen recompiles the body successfully, then the body become valid.
- When you recompile a package, both the specification and the body are explicitly recompiled. If there are no compilation errors, then the specification and body become valid. If there are compilation errors, then TimesTen returns an error and the package remains invalid.

See also

CREATE PACKAGE

ALTER PROCEDURE

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The ALTER PROCEDURE statement recompiles a standalone stored procedure. Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the package using the ALTER PACKAGE statement.

Required privilege

No privilege is required for the procedure owner.

ALTER ANY PROCEDURE for another user's procedure.

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
ALTER PROCEDURE [Owner.]ProcedureName COMPILE
    [CompilerParametersClause [...]]
    [REUSE SETTINGS]
```

Parameters

Parameter	Description
[*Owner*.]*ProcedureName*	Name of the procedure to be recompiled.
COMPILE	Required keyword that causes recompilation of the procedure. If the procedure does not compile successfully, use the ttIsql command SHOW ERRORS to display the compiler error messages.
CompilerParametersClause	Use this optional clause to specify a value for one of the PL/SQL persistent compiler parameters. The PL/SQL persistent compiler parameters are PLSQL_OPTIMIZE_LEVEL and NLS_LENGTH_SEMANTICS. You can specify each parameter once in the statement. If you omit a parameter from this clause and you specify REUSE SETTINGS, then if a value was specified for the parameter in an earlier compilation, TimesTen uses that earlier value. If you omit a parameter and either you do not specify REUSE SETTINGS or no value has been specified for the parameter in an earlier compilation, then TimesTen obtains the value for the parameter from the session environment.
REUSE SETTINGS	Use this optional clause to prevent TimesTen from dropping and reacquiring compiler switch settings. When you specify REUSE SETTINGS, TimesTen preserves the existing settings and uses them for the compilation of any parameters for which values are not specified.

Description

- The ALTER PROCEDURE statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a procedure, use the CREATE PROCEDURE statement.
- TimesTen first recompiles objects upon which the procedure depends, if any of those objects are invalid.
- TimesTen also invalidates any objects that depend on the procedure, such as procedures that call the recompiled procedure or package bodies that define procedures that call the recompiled procedure.
- If TimesTen recompiles the procedure successfully, then the procedure becomes valid. If recompiling the procedure results in compilation errors, then TimesTen returns an error and

the procedure remains invalid. Use the ttIsql command SHOW ERRORS to display compilation errors.

- During recompilation, TimesTen drops all persistent compiler settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the REUSE SETTINGS clause.

Examples

Query the system view USER_PLSQL_OBJECT_SETTINGS to check PLSQL_OPTIMIZE_LEVEL for procedure *query_emp*. Alter *query_emp* by changing PLSQL_OPTIMIZE_LEVEL to 3. Verify results.

```
Command> SELECT PLSQL_OPTIMIZE_LEVEL FROM user_plsql_object_settings WHERE  name = 'QUERY_EMP';
< 2 >
1 row found.

Command> ALTER PROCEDURE query_emp COMPILE PLSQL_OPTIMIZE_LEVEL = 3;

Procedure altered.

Command> SELECT PLSQL_OPTIMIZE_LEVEL FROM user_plsql_object_settings WHERE  name = 'QUERY_EMP';
< 3 >
1 row found.
```

See also

CREATE PROCEDURE

ALTER PROFILE

The ALTER PROFILE statement adds, modifies, or removes one or more password parameters in a profile.

Required privilege

ADMIN

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
ALTER PROFILE profile LIMIT password_parameters

password_parameters::=
[FAILED_LOGIN_ATTEMPTS password_parameter_options]
[PASSWORD_LIFE_TIME password_parameter_options]
[PASSWORD_REUSE_TIME password_parameter_options]
[PASSWORD_REUSE_MAX password_parameter_options]
[PASSWORD_LOCK_TIME password_parameter_options]
[PASSWORD_GRACE_TIME password_parameter_options]
[{PASSWORD_COMPLEXITY_CHECKER|PASSWORD_VERIFY_FUNCTION} password_checker_options]

password_parameter_options::=
UNLIMITED|DEFAULT|constant

password_checker_options::=
function|NULL|DEFAULT
```

function::=
TT_VERIFY_FUNCTION|TT_STRONG_VERIFY_FUNCTION|TT_STIG_VERIFY_FUNCTION

Parameters

Parameter	Description
profile	Name of the profile.
LIMIT *password_parameters*	The LIMIT clause sets the limits for the password parameters. The LIMIT keyword is required. The password parameters consist of the name of the password parameter and the value (or limit) for the password parameter. This includes the password complexity checker functions. All the parameters (with the exception of FAILED_LOGIN_ATTEMPTS and PASSWORD_REUSE_MAX) set lengths of time and are interpreted in number of days. You can use a decimal value (for example, you can use .0833 to denote approximately one hour). The minimum value is 1 second. The maximum value is 106,751,991 days. The constant value must be expressed in days. For example, to set a value of 5 minutes, specify the constant value of 0.0034722222222222 (5/1440 days). For FAILED_LOGIN_ATTEMPTS and PASSWORD_REUSE_MAX, you must specify an integer. If you do not specify a password parameter after the LIMIT clause, the limit for that password parameter is based on the limit defined in the DEFAULT profile. In addition, if you only specify the LIMIT keyword with no additional parameters, the limits for the profile are based on the limits of the DEFAULT profile.
FAILED_LOGIN_ATTEMPTS	Specifies the number of consecutive failed attempts to connect to the database by a user before that user's account is locked.
PASSWORD_LIFE_TIME	Specifies the number of days that a user can use the same password for authentication. If you also set a value for PASSWORD_GRACE_TIME, then the password expires if it is not changed within the grace period. In such a situation, future connections to the database are rejected.

Parameter	Description
PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX	These two parameters must be used together. • PASSWORD_REUSE_TIME specifies the number of days that must pass before a user can reuse a password. For example, if you specify a value of 30, then after 30 days the user can reuse a previous password. • PASSWORD_REUSE_MAX specifies the number of password changes that are required before the current password can be reused. You must specify a value for both parameters for them to have any effect. Specifically: • If you specify a value for both parameters: A user cannot reuse a password until the password has been changed the number of times specified for PASSWORD_REUSE_MAX during the number of days specified for PASSWORD_REUSE_TIME. For example, if you specify a value of 30 for PASSWORD_REUSE_TIME and a value of 10 for PASSWORD_REUSE_MAX, then the user can reuse the password after 30 days if the password has been changed 10 times. • If you specify a value for one parameter and specify a value of UNLIMITED for the second parameter, then the user can never reuse a password. • If you specify a value of UNLIMITED for both parameters, then TimesTen ignores both values, indicating that the password can be reused.
PASSWORD_LOCK_TIME	Specifies the number of days the user account is locked after the specified number of consecutive failed connection attempts.
PASSWORD_GRACE_TIME	Specifies the number of days after the grace period begins during which TimesTen issues a warning, but allows the connection to the database. If the password is not changed during the grace period, the password expires. This parameter is associated with the PASSWORD_LIFE_TIME parameter.
UNLIMITED	Indicates that there is no limit for the password parameter. If you specify UNLIMITED, it must follow the password parameter. For example, FAILED_LOGIN_ATTEMPTS UNLIMITED.
DEFAULT	Indicates that you want to omit a limit for the password parameter in this profile. A user that is assigned this profile is subject to the limit defined in the DEFAULT profile for this password parameter. If you specify DEFAULT, it must follow the password parameter. For example, FAILED_LOGIN_ATTEMPTS DEFAULT.
constant	Indicates the value of the password parameter if you do not specify UNLIMITED or DEFAULT. If specified, it must follow the password parameter. For example, FAILED_LOGIN_ATTEMPTS 3.

Parameter	Description
{PASSWORD_COMPLEXITY_CHECKER|PASSWORD_VERIFY_FUNCTION} {*function*|NULL|DEFAULT}	Indicates if password verification is done on passwords and, if so, the function used for verification. You can specify either the PASSWORD_COMPLEXITY_CHECKER or the PASSWORD_VERIFY_FUNCTION password parameter. They are synonymous. *function* refers to one of the three supported password complexity checker functions. Specify one of these functions to direct TimesTen to perform password verification. Valid values: • TT_VERIFY_FUNCTION • TT_STRONG_VERIFY_FUNCTION • TT_STIG_VERIFY_FUNCTION NULL indicates that there is not a password verification function assigned for the profile. DEFAULT indicates that the user is subject to the limits defined by the DEFAULT profile. The DEFAULT profile initially has a value of NULL. If you do not specify the PASSWORD_COMPLEXITY_CHECKER password parameter, the value defaults to the limits defined for the DEFAULT profile.

Description

- Use the ALTER PROFILE statement to modify a previously created profile. See "CREATE PROFILE" for information on creating a profile.
- Changes made using the ALTER PROFILE statement takes effect the next time any affected user connected to the database. The exception is when you modify the PASSWORD_COMPLEXITY_CHECKER password parameter. Password verification is only done on newly created passwords (on the password provided in the IDENTIFIED BY clause of the CREATE USER or ALTER USER statement). Therefore, a user can connect to the database with an old password. See "ALTER the PASSWORD_COMPLEXITY_CHECKER password parameter" for an example.
- You can alter the DEFAULT profile. However, you cannot drop the DEFAULT profile. See "Alter the DEFAULT profile" for an example of altering the DEFAULT profile.
- You cannot alter the password parameters of the SYSTEM profile. This profile is assigned to system users, including the instance administrator.
- You can alter the profile to change the password verification that is done on the passwords of users that are assigned the profile. See "About Password Complexity Checker Verification" for information on password verification and the password complexity checker verification functions.

Examples

ALTER the PASSWORD_COMPLEXITY_CHECKER password parameter

This example creates the myprofile_alterpw1 profile and specifies TT_VERIFY_FUNCTION for the PASSWORD_COMPLEXITY_CHECKER password parameter. The example then creates the sampleuser_alterpw1 user and assigns the myprofile_alterpw1 profile to the sampleuser_alterpw1 user. The example alters the profile, specifying TT_STIG_VERIFY_FUNCTION for the PASSWORD_COMPLEXITY_CHECKER password parameter. The sampleuser_alterpw1 attempts to connect to the database with the original password. The connection is successful. TimesTen does not perform password verification on old passwords. The example then uses the ALTER USER statement to change the sampleuser_alterpw1 user password to meet the requirements of the TT_STIG_VERIFY_FUNCTION. The ALTER USER statement succeeds and the user's password is changed.

```
Command> CREATE PROFILE myprofile_alterpw1 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_VERIFY_FUNCTION;

Profile created.

Command> CREATE USER sampleuser_alterpw1
         IDENTIFIED BY "%aabb2L90" PROFILE myprofile_alterpw1;

User created.
```

Alter the myprofile_alterpw1 profile, changing the value of PASSWORD_COMPLEXITY_CHECKER to TT_STIG_VERIFY_FUNCTION. Connect to the database as the sampleuser_alterpw1 user. The connection succeeds.

```
Command> ALTER PROFILE myprofile_alterpw1 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STIG_VERIFY_FUNCTION;

Profile altered.

Command> GRANT CONNECT TO sampleuser_alterpw1;
Command> connect adding "UID=sampleuser_alterpw1;PWD=%aabb2L90" as sampleuser;
Connection successful: DSN=access1;UID=sampleuser_alterpw1;
DataStore=/scratch/sampleuser/mydatabase1;DatabaseCharacterSet=AL32UTF8;
ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Alter the sampleuser_alterpw1 user specifying the same password. The ALTER USER statement fails. The newly created password does not meet the requirements of the TT_STIG_VERIFY_FUNCTION function. Alter the sampleuser_alterpw1 again, specifying a password that meets the requirements of the TT_STIG_VERIFY_FUNCTION function. The ALTER USER statement succeeds. See "TT_STIG_VERIFY_FUNCTION" for information on the TT_STIG_VERIFY_FUNCTION function.

```
Command> ALTER USER sampleuser_alterpw1
         IDENTIFIED BY  "%aabb2L90";
15186: Password complexity check for the specified password failed
15188: TT-20001: Password length less than 15
The command failed.

Command> ALTER USER sampleuser_alterpw1
         IDENTIFIED BY  "%aabb2L##mf5Fn!";

User altered.
```

Alter the DEFAULT profile

This example verifies the values of the password parameters in the DEFAULT profile. It then alters the profile with different values. Users that are assigned the DEFAULT profile will inherit the modified values at the user's next connection to the database.

```
Command> SELECT * FROM dba_profiles WHERE profile='DEFAULT' AND
         resource_type='PASSWORD';
< DEFAULT, FAILED_LOGIN_ATTEMPTS, PASSWORD, 10 >
< DEFAULT, PASSWORD_LIFE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_MAX, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
< DEFAULT, PASSWORD_LOCK_TIME, PASSWORD, .0034 >
< DEFAULT, PASSWORD_GRACE_TIME, PASSWORD, UNLIMITED >
7 rows found.
```

Create the user1 user and do not specify a profile. User1 is assigned the DEFAULT profile. Use the ALTER PROFILE statement to change the value of the FAILED_LOGIN_ATTEMPTS password parameter to 5 and the value of the PASSWORD_LOCK_TIME password parameter to 1 for the DEFAULT profile. Enclose DEFAULT in double quotation marks as DEFAULT is a reserved word. Connect to the database five times as user1 supplying an incorrect password each time. On the sixth attempt, the user1 account is locked.

```
Command> CREATE USER user1 IDENTIFIED BY user1;

User created.
Command> GRANT CONNECT TO user1;
```

Query the dba_users system view to verify that user1 is assigned the DEFAULT profile.

```
Command> SELECT profile FROM dba_users WHERE username='USER1';
< DEFAULT >
1 row found.
```

Use the ALTER PROFILE statement to modify the DEFAULT profile.

```
Command> ALTER PROFILE "DEFAULT" LIMIT
         FAILED_LOGIN_ATTEMPTS 5
         PASSWORD_LOCK_TIME 1;

Profile altered.
```

Query the dba_profiles system view to verify the values are changed (represented in **bold**).

```
Command> SELECT * FROM dba_profiles WHERE profile='DEFAULT' AND
   resource_type='PASSWORD';
< DEFAULT, FAILED_LOGIN_ATTEMPTS, PASSWORD, 5 >
< DEFAULT, PASSWORD_LIFE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_MAX, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
< DEFAULT, PASSWORD_LOCK_TIME, PASSWORD, 1 >
< DEFAULT, PASSWORD_GRACE_TIME, PASSWORD, UNLIMITED >
7 rows found.
```

Attempt to connect to the database as user1. Supply an incorrect password. On the sixth attempt, the user1 account is locked.

```
Command> connect adding "uid=user1;pwd=user1_test1" as user1;
 7001: User authentication failed
The command failed.
none: Command> connect adding "uid=user1;pwd=user1_test2" as user1;
 7001: User authentication failed
The command failed.
none: Command> connect adding "uid=user1;pwd=user1_test3" as user1;
 7001: User authentication failed
```

```
The command failed.
none: Command> connect adding "uid=user1;pwd=user1_test4" as user1;
 7001: User authentication failed
The command failed.
none: Command> connect adding "uid=user1;pwd=user1_test5" as user1;
 7001: User authentication failed
The command failed.
none: Command> connect adding "uid=user1;pwd=user1_test6" as user1;
15179: the account is locked
The command failed.
```

Create a profile then alter the profile

This example creates the `profile1` profile and specifies values for the `FAILED_LOGIN_ATTEMPTS`, the `PASSWORD_LIFE_TIME`, the `PASSWORD_LOCK_TIME`, and the `PASSWORD_GRACE_TIME` password parameters. It then alters the `profile1` profile to modify the `PASSWORD_REUSE_TIME` and the `PASSWORD_REUSE_MAX` password parameters.

```
Command> CREATE PROFILE profile1 LIMIT
         FAILED_LOGIN_ATTEMPTS 3
         PASSWORD_LIFE_TIME 90
         PASSWORD_LOCK_TIME 30
         PASSWORD_GRACE_TIME 10;

Profile created.
```

Query the `dba_profiles` system view to verify the values for the password parameters. Note that the `PASSWORD_REUSE_TIME` and the `PASSWORD_REUSE_MAX` password parameters each have a value of `DEFAULT` (represented in bold). These password parameters were not specified in the `CREATE PROFILE` definition, so TimesTen assigns a value of `DEFAULT` to each parameter. The values for these parameters are derived from the values in the `DEFAULT` profile.

```
Command> SELECT * FROM dba_profiles WHERE profile = 'PROFILE1' AND
         resource_type= 'PASSWORD';
< PROFILE1, FAILED_LOGIN_ATTEMPTS, PASSWORD, 3 >
< PROFILE1, PASSWORD_LIFE_TIME, PASSWORD, 90 >
< PROFILE1, PASSWORD_REUSE_TIME, PASSWORD, DEFAULT >
< PROFILE1, PASSWORD_REUSE_MAX, PASSWORD, DEFAULT >
< PROFILE1, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, DEFAULT >
< PROFILE1, PASSWORD_LOCK_TIME, PASSWORD, 30 >
< PROFILE1, PASSWORD_GRACE_TIME, PASSWORD, 10 >
7 rows found.
```

Alter the `profile1` profile, specifying a value of `20` for the `PASSWORD_REUSE_TIME` password and a value of `15` for the `PASSWORD_REUSE_MAX` password parameter (represented in **bold**). A user assigned this profile can reuse the same password after `20` days if the password has been changed `15` times.

```
Command> ALTER PROFILE profile1 LIMIT
         PASSWORD_REUSE_TIME 20
         PASSWORD_REUSE_MAX 15;

Profile altered.
```

Query the `dba_profiles` system view to verify the values for the password parameters are changed (represented in **bold**).

```
Command> SELECT * FROM dba_profiles WHERE profile = 'PROFILE1' AND
         resource_type= 'PASSWORD';
< PROFILE1, FAILED_LOGIN_ATTEMPTS, PASSWORD, 3 >
< PROFILE1, PASSWORD_LIFE_TIME, PASSWORD, 90 >
```

< PROFILE1, **PASSWORD_REUSE_TIME**, PASSWORD, **20** >
< PROFILE1, **PASSWORD_REUSE_MAX**, PASSWORD, **15** >
< PROFILE1, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, DEFAULT >
< PROFILE1, PASSWORD_LOCK_TIME, PASSWORD, 30 >
< PROFILE1, PASSWORD_GRACE_TIME, PASSWORD, 10 >
7 rows found.

See also

CREATE PROFILE
DROP PROFILE
CREATE USER
ALTER USER

ALTER REPLICATION

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The ALTER REPLICATION statement adds, alters, or drops replication elements and changes the replication attributes of participating databases involved in a classic replication scheme.

Most ALTER REPLICATION operations are supported only when the replication agent is stopped (ttAdmin -repStop). However, it is possible to dynamically add a subscriber database to a replication scheme while the replication agent is running. See Altering a Classic Replication Scheme in *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Required Privilege

ADMIN

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL Syntax

The ALTER REPLICATION statement has the syntax:

```
ALTER REPLICATION [Owner.]ReplicationSchemeName
 ElementOperation [...] | StoreOperation |
 NetworkOperation [...]
```

Specify *ElementOperation* one or more times:

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

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

CheckConflicts can only be set when replicating TABLE elements. See CHECK CONFLICTS for syntax requirements.

Syntax for *ReturnServiceAttribute* is:

```
{ RETURN RECEIPT [BY REQUEST] | NO RETURN }
```

StoreOperation clauses:

```
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
CONFLICT REPORTING SUSPEND AT Value
CONFLICT REPORTING RESUME AT Value
TABLE DEFINITION CHECKING {EXACT|RELAXED}
```

Specify *NetworkOperation* one or more times:

```
ADD ROUTE MASTER FullStoreName SUBSCRIBER FullStoreName
 { { MASTERIP MasterHost | SUBSCRIBERIP SubscriberHost }
   PRIORITY Priority } [...]

DROP ROUTE MASTER FullStoreName SUBSCRIBER FullStoreName
 { MASTERIP MasterHost | SUBSCRIBERIP SubscriberHost } [...]
```

Parameters

Parameter	Description
[*Owner*.]*ReplicationSchemeName*	Name assigned to the classic replication scheme.

Parameter	Description
ADD ELEMENT *ElementName*	Adds a new element to the existing classic 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 characters. If the element is a DATASTORE, all tables are included in the database. SEQUENCE elements that are part of the database do not have their return services modified by this statement.
ADD ELEMENT *ElementName* DATASTORE {INCLUDE | EXCLUDE} {TABLE [[*Owner*.]*TableName* [,...]]| SEQUENCE [[*Owner*.]*SequenceName*[,...]]} [,...]	Adds a new DATASTORE element to the existing classic 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 characters. INCLUDE includes in the database only the tables and sequences listed. Use one INCLUDE clause for each object type (table or sequence). EXCLUDE includes in the database all tables and sequences *except* the tables and sequences listed. Use one EXCLUDE clause for each object type (table or sequence). 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 database. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.
ALTER ELEMENT * IN *FullStoreName* SET { MASTER | PROPAGATOR } *FullStoreName*	Makes a change to all elements for which *FullStoreName* is the MASTER or PROPAGATOR. *FullStoreName* is the database 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 database 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* {INCLUDE|EXCLUDE} {TABLE [*Owner*.]*TableName* | SEQUENCE [*Owner*.]*SequenceName*} [,...]	*ElementName* is the name of the element to be altered. INCLUDE adds to the database the tables and sequences listed. Use one INCLUDE clause for each object type (table or sequence). EXCLUDE removes from the database the tables and sequences listed. Use one EXCLUDE clause for each object type (table or sequence). If the element is a sequence, RETURN attributes are not applied, no conflict checking is supported and sequences that cycle return an error.

Parameter	Description
ALTER SUBSCRIBER *FullStoreName* SET RETURN RECEIPT [BY REQUEST]|NO RETURN	Indicates an alteration to a subscriber database to enable, disable, or change the return receipt service. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.
CheckConflicts	Check for replication conflicts when simultaneously writing to bidirectionally replicating TABLE elements between databases. You cannot check for conflicts when replicating elements of type DATASTORE. See CHECK CONFLICTS.
COMPRESS TRAFFIC {ON | OFF}	Compress replicated traffic to reduce the amount of network bandwidth. ON specifies that all replicated traffic for the database defined by STORE be compressed. OFF (the default) specifies no compression. See Compressing Replicated Traffic in *Oracle TimesTen In-Memory Database Replication Guide* for details.
CONFLICT REPORTING SUSPEND AT *Value*	Suspends conflict resolution reporting. *Value* is a non-negative integer. Conflict reporting is suspended when the rate of conflict exceeds *Value*. The default is 0, which means that the conflict reporting is always on and will not be suspended. This clause is valid for table level replication.
CONFLICT REPORTING RESUME AT *Value*	Resumes conflict resolution reporting. *Value* is a non-negative integer. Conflict reporting is resumed when the rate of conflict falls below *Value*. The default is 1. This clause is valid for table level replication.
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. If DISABLE RETURN is specified but RESUME RETURN is not specified, the return services remain off until the replication agent for the database has been restarted.
DURABLE COMMIT {ON | OFF}	Overrides the DurableCommits general connection attribute setting. DURABLE COMMIT ON enables durable commits regardless of whether the replication agent is running or stopped.
DROP ELEMENT * IN *FullStoreName*	Deletes the replication description of all elements for which *FullStoreName* is the MASTER. *FullStoreName* is the database 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 database. This operation fails if the classic replication scheme has only one subscriber. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.

Parameter	Description
FAILTHRESHOLD *Value*	The number of log files that can accumulate for a subscriber database. If this value is exceeded, the subscriber is set to the Failed state. The value 0 means "No Limit." This is the default. See Setting the Transaction Log Failure Threshold in *Oracle TimesTen In-Memory Database Replication Guide* for more information.
FullStoreName	The database, specified as one of the following: • SELF • The prefix of the database file name For example, if the database path is *directory*/*subdirectory*/data.ds0, then data is the database name. This is the database 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 the Network in *Oracle TimesTen In-Memory Database 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 attempts to perform a COMMIT to end 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 database on which applications update the specified element. The MASTER database sends updates to its SUBSCRIBER databases. *FullStoreName* is the database 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 *Oracle TimesTen In-Memory Database Replication Guide*.
PORT *PortNumber*	The TCP/IP port number on which the replication agent on this database listens for connections. If not specified, the replication agent allocates a port number automatically. All TimesTen databases that replicate to each other must use the same port number.
PROPAGATOR *FullStoreName*	The database that receives replicated updates and passes them on to other databases.

Parameter	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*. If DISABLE RETURN is specified but RESUME RETURN is not specified, the return services remain off until the replication agent for the database has been restarted.
RETURN RECEIPT [BY REQUEST]	Enables the return receipt service, so that applications that commit a transaction to a master database 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 *Oracle TimesTen In-Memory Database Replication Guide*.
RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED	Sets return services on or off when replication is disabled (stopped or paused state). OFF disables return services when replication is disabled and is the default for RETURN RECEIPT service. ON allows return services to continue to be enabled when replication is disabled and is the default for RETURN TWOSAFE service.
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master database 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 *Oracle TimesTen In-Memory Database Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgment. The default value is 10 seconds. A value of 0 (zero) means there is no timeout. Your application can override this timeout setting by calling the ttRepSyncSet procedure with the returnWait parameter.
SET {MASTER | PROPAGATOR} *FullStoreName*	Sets the given database to be the MASTER or PROPAGATOR of the given elements. The *FullStoreName* must the be database's file base name.
SUBSCRIBER *FullStoreName*	A database that receives updates from the MASTER databases. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.

Parameter	Description
TABLE DEFINITION CHECKING {EXACT|RELAXED}	Specifies type of table definition checking that occurs on the subscriber: • EXACT - The tables must be identical on master and subscriber. • RELAXED - The tables must have the same key definition, number of columns and column data types. The default is RELAXED. **Note:** If you use TABLE DEFINITION CHECKING EXACT, use ttMigrate -exactUpgrade if you migrate the database. If you use TABLE DEFINITION CHECKING RELAXED, use ttMigrate -relaxedUpgrade if you migrate the database.
TIMEOUT *Seconds*	The maximum number of seconds the replication agent waits for a response from remote replication agents. The default is 120 seconds. **Note:** For large transactions that may cause a delayed response from the remote replication agent, the agent scales the timeout to increasingly larger values, as needed, based on the size of the transaction. This scaling will not occur, and the agent may time out waiting for responses, if you set TIMEOUT to less than or equal to 60 seconds for large transactions. Also see Setting Wait Timeout for Response from Remote Replication Agents in *Oracle TimesTen In-Memory Database Replication Guide*.
ADD ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Adds *NetworkOperation* to replication scheme. Enables you to control the network interface that a master store uses for every outbound connection to each of its subscriber stores. Can be specified more than once. For *FullStoreName*, ON "*host*" must be specified.
DROP ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Drops *NetworkOperation* from the classic replication scheme. Can be specified more than once. For *FullStoreName*, ON *"host"* must be specified.
MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost*	*MasterHost* and *SubscriberHost* are the IP addresses for the network interface on the master and subscriber stores. Specify in dot notation or canonical format or in colon notation for IPV6. Clause can be specified more than once. Valid for both ADD and DROP ROUTE MASTER.
PRIORITY *Priority*	Variable expressed as an integer from 1 to 99. Denotes the priority of the IP address. Lower integral values have higher priority. An error is returned if multiple addresses with the same priority are specified. Controls the order in which multiple IP addresses are used to establish peer connections. Required syntax of *NetworkOperation* clause. Follows MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost* clause.

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 database. SET NAME does not admit the use of * IN *FullStoreName*. The *FullStoreName* must be the database's file base name. For example, if the database file name is data.ds0, then data is the file base name.

- ALTER ELEMENT SET MASTER may be used to change the master database for replication elements. The * IN *FullStoreName* option must be used for the MASTER operation. That is, a master database 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 databases after the failure of a (common) MASTER.
- To transfer ownership of the master elements to the subscriber:
  1. Manually drop the replicated elements by executing an ALTER REPLICATION DROP ELEMENT statement for each replicated table.
  2. 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 database, and the currently designated MASTER cannot be deleted from the replication scheme.

- Stop the replication agent before you use the *NetworkOperation* clause.
- You cannot alter the following replication schemes with the ALTER REPLICATION statement:
  - Any active standby pair. Instead, use ALTER ACTIVE STANDBY PAIR.
  - A Clusterware-managed active standby pair. Instead, perform the tasks described in Changing the Schema section of the *Oracle TimesTen In-Memory Database Replication Guide*.

Examples

This example sets up a classic replication scheme for an additional table westleads that is updated on database west and replicated to database east.

```
ALTER REPLICATION r1
  ADD ELEMENT e3 TABLE westleads
    MASTER west ON "westcoast"
    SUBSCRIBER east ON "eastcoast";
```

This example adds an additional subscriber (backup) to table westleads.

```
ALTER REPLICATION r1
  ALTER ELEMENT e3
    ADD SUBSCRIBER backup ON "backupserver";
```

This example changes the element name of table westleads from e3 to newelementname.

```
ALTER REPLICATION r1
  ALTER ELEMENT e3
    SET NAME newelementname;
```

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;
```

This element changes the port number for east.

```
ALTER REPLICATION r1
  ALTER STORE east ON "eastcoast" SET PORT 22251;
```

This example adds my.tab1 table to the ds1 database element in my.rep1 replication scheme.

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

This example adds ds1 database to my.rep1 replication scheme. Include my.tab2 table in the database.

```
ALTER REPLICATION my.rep1
  ADD ELEMENT ds1 DATASTORE
    MASTER rep2
    SUBSCRIBER rep1, rep3
    INCLUDE TABLE my.tab2;
```

This example adds ds2 database to a replication scheme but excludes my.tab1 table.

```
ALTER REPLICATION my.rep1
  ADD ELEMENT ds2 DATASTORE
    MASTER rep2
    SUBSCRIBER rep1
    EXCLUDE TABLE my.tab1;
```

Add *NetworkOperation* clause:

```
ALTER REPLICATION r
ADD ROUTE MASTER rep1 ON "machine1" SUBSCRIBER rep2 ON "machine2"
MASTERIP "1.1.1.1" PRIORITY 1 SUBSCRIBERIP "2.2.2.2"
   PRIORITY 1
MASTERIP "3.3.3.3" PRIORITY 2 SUBSCRIBERIP "4.4.4.4" PRIORITY 2;
```

Drop *NetworkOperation* clause:

```
ALTER REPLICATION r
DROP ROUTE MASTER repl ON "machine1" SUBSCRIBER rep2 ON "machine2"
MASTERIP "1.1.1.1" SUBSCRIBERIP "2.2.2.2"
MASTERIP "3.3.3.3" SUBSCRIBERIP "4.4.4.4";
```

See Also

- ALTER ACTIVE STANDBY PAIR
- CREATE ACTIVE STANDBY PAIR
- CREATE REPLICATION
- DROP ACTIVE STANDBY PAIR
- DROP REPLICATION

To drop a table from a database, see Altering a Replicated Table in a Classic Replication Scheme in *Oracle TimesTen In-Memory Database Replication Guide*.

ALTER SEQUENCE

This statement is supported in TimesTen Scaleout only.

Use the ALTER SEQUENCE statement to change the batch value of a sequence.

Required Privilege

No privilege is required for the sequence owner.

ALTER ANY SEQUENCE privilege for another user's sequence.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
ALTER SEQUENCE [Owner.]SequenceName BATCH BatchValue
```

Parameters

Parameter	Description
SEQUENCE [*Owner*.]*SequenceName*	Name of the sequence to be altered.
BATCH *BatchValue*	Valid with TimesTen Scaleout only. Configures the range of unique sequence values that are stored at each element of the grid. The default value is 10 million.

Description

- Use this statement to change the batch value for a sequence in TimesTen Scaleout. The change affects future sequence numbers.
- This statement cannot be used to alter any other values supported in the CREATE SEQUENCE statement. In this case, 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.

See Using Sequences in *Oracle TimesTen In-Memory Database Scaleout User's Guide* for more information.

Examples

To change the batch value for the sequence:

```
ALTER SEQUENCE myseq BATCH 2000;
Sequence altered
```

See also

CREATE SEQUENCE
DROP SEQUENCE

ALTER SESSION

The ALTER SESSION statement changes session parameters dynamically. This overrides the setting of the equivalent connection attribute for the current session, as applicable.

Required Privilege

None

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. However, these parameters are not supported:

- DDL_REPLICATION_ACTION
- DDL_REPLICATION_LEVEL
- REPLICATION_TRACK

SQL Syntax

```
ALTER SESSION SET
 {COMMIT_BUFFER_SIZE_MAX = n |
  DDL_REPLICATION_ACTION={'INCLUDE'|'EXCLUDE'} |
  DDL_REPLICATION_LEVEL={1|2|3} |
  ISOLATION_LEVEL = {SERIALIZABLE | READ COMMITTED} |
  NLS_SORT = {BINARY| SortName} |
  NLS_LENGTH_SEMANTICS = {BYTE|CHAR} |
  NLS_NCHAR_CONV_EXCP = {TRUE|FALSE} |
  PLSQL_TIMEOUT = n |
  PLSQL_OPTIMIZE_LEVEL = {0|1|2|3}|
  PLSQL_CONN_MEM_LIMIT = n |
  PLSQL_CCFLAGS = 'name1:value1, name2:value2,..., nameN:valueN' |
  PLSQL_SESSION_CACHED_CURSORS = n |
  REPLICATION_TRACK = TrackNumber |
  }
```

Parameters

Parameter	Description
COMMIT_BUFFER_SIZE_MAX= *n*	Changes the maximum size of the commit buffer when a connection is in progress. *n* is expressed as an integer and represents the maximum size of the commit buffer (in MB). Change takes effect starting with the *next* transaction. Call the ttConfiguration built-in procedure to see the currently configured maximum size of the commit buffer. A value of 0 means the buffer is configured with a default size. The default size is 128 KB. For more information on the commit buffer and transaction reclaim operations, see Transaction Reclaim Operations in the *Oracle TimesTen In-Memory Database Operations Guide* and CommitBufferSizeMax in the *Oracle TimesTen In-Memory Database Reference*. **Note:** The equivalent connection attribute is CommitBufferSizeMax.

Parameter	Description
DDL_REPLICATION_ACTION={'INCLUDE'|'EXCLUDE'}	To include a table or sequence in the active standby pair when either is created, set DDL_REPLICATION_ACTION to INCLUDE. If you do not want to include a table or sequence in the active standby pair when either is created, set DDL_REPLICATION_ACTION to EXCLUDE. The default is INCLUDE. If set to EXCLUDE: • A subsequent ALTER ACTIVE STANDBY PAIR ... INCLUDE TABLE is required to be executed on the active database to add the table to the replication scheme. All tables must be empty on all active standby databases and subscribers as the table contents will be truncated when this statement is executed. • A subsequent ALTER ACTIVE STANDBY PAIR ... INCLUDE SEQUENCE is required to be executed on the active database to add the sequence to the replication scheme. This attribute is valid only if DDL_REPLICATION_LEVEL is 2 or greater. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information. **Note:** The equivalent connection attribute is DDLReplicationAction.
DDL_REPLICATION_LEVEL={1|2|3}	Indicates whether DDL is replicated across all databases in an active standby pair. The value can be one of the following: • 1: Default. Add or drop a column to or from a replicated table on the active database using ALTER TABLE. The change is replicated to the table in the standby database. • 2: Supports replication of the creation or dropping of tables, synonyms or indexes from the active database to the standby database. This does include creating or dropping global temporary tables, but does not include CREATE TABLE AS SELECT. The statement is replicated only when the index is created on an empty table. • 3: Supports replication of all DDL supported by level 2 as well as replication of creation or dropping of views and sequences (not including materialized views) and changes to the cache administration user ID and password settings when you call the ttCacheUidPwdSet built-in procedure. **Note**: After you have defined cache groups, you cannot change the cache administration user ID, but can still change the cache administration password. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information. **Note:** The equivalent connection attribute is DDLReplicationLevel.
ISOLATION_LEVEL = {SERIALIZABLE|READ COMMITTED}	Sets isolation level. Change takes effect starting with the *next* transaction. For a descriptions of the isolation levels, see Transaction Isolation Levels in the *Oracle TimesTen In-Memory Database Operations Guide*. **Note:** The equivalent connection attribute is Isolation.

Parameter	Description
NLS_SORT={BINARY| *SortName*}	Indicates which collation sequence to use for linguistic comparisons. Append _CI or _AI to either BINARY or the *SortName* value to do case-insensitive or accent-insensitive sorting. If you do not specify NLS_SORT, the default is BINARY. For a complete list of supported values for *SortName*, see Linguistic Sort Rules Support Linguistic Conventions in *Oracle TimesTen In-Memory Database Operations Guide*. For more information on case-insensitive or accent-insensitive sorting, see Case-Insensitive and Accent-Insensitive Linguistic Sorts in *Oracle TimesTen In-Memory Database Operations Guide*.
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 Character Set Length Semantics Affect Data Storage in *Oracle TimesTen In-Memory Database Operations Guide*.
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.
PLSQL_TIMEOUT= *n*	Controls how long PL/SQL procedures run before being automatically terminated. *n* represents the time, in seconds. Specify 0 for no time limit or any positive integer. The default is 30. When you modify this value, the new value impacts PL/SQL program units that are currently running as well as any other program units subsequently executed in the same connection. See Choose SQL and PL/SQL Timeout Values in the *Oracle TimesTen In-Memory Database Operations Guide* for information on setting timeout values.
PLSQL_OPTIMIZE_LEVEL = {0|1|2|3}	Specifies the optimization level used to compile PL/SQL library units. The higher the setting, the more effort the compiler makes to optimize PL/SQL library units. Possible values are 0, 1, 2 or 3. The default is 2. For more information, see PLSQL_OPTIMIZE_LEVEL in *Oracle TimesTen In-Memory Database Reference*.
PLSQL_CONN_MEM_LIMIT = *n*	Specifies the maximum amount of process heap memory that PL/SQL can use for this connection, where *n* is an integer expressed in MB. The default is 100. For more information, see PLSQL_CONN_MEM_LIMIT in *Oracle TimesTen In-Memory Database Reference*.
PLSQL_CCFLAGS = '*name1:value1, name2:value2, ..., nameN:valueN*'	Specifies inquiry directives to control conditional compilation of PL/SQL units, which enables you to customize the functionality of a PL/SQL program depending on conditions that are checked. For example, to activate debugging features: `PLSQL_CCFLAGS = 'DEBUG:TRUE'`

Parameter	Description
PLSQL_SESSION_CACHED_CURSORS= *n*	Specifies the maximum number of session cursors to cache. The default is 50. The range of values is 1 to 65535. The PLSQL_SESSION_CACHED_CURSORS setting in TimesTen behaves the same as the SESSION_CACHED_CURSORS setting in Oracle RDBMS.
REPLICATION_TRACK = *TrackNumber*	When managing track-based parallel replication, you can assign a connection to a replication track. All transactions issued by the connection are assigned to this track, unless the track is altered. If the number specified is for a non-existent replication track X, the transaction is assigned to a track number computed as X modulo ReplicationParallelism. You cannot change tracks in the middle of a transaction unless all preceding operations have been read operations. For more information, see Specifying Replication Tracks Within an Automatic Parallel Replication Environment in *Oracle TimesTen In-Memory Database Replication Guide*. The equivalent connection attribute is ReplicationTrack.

Description

- The ALTER SESSION statement affects commands that are subsequently executed by the session. ALTER SESSION does not do an implicit commit.
- In cases of client failover, if an ALTER SESSION statement is issued in the failed connection, the setting is not seen or carried over to the new connection. You must re-issue the ALTER SESSION statement and re-specify the value for that parameter. For more information on client failover, in TimesTen Classic, see Using Automatic Client Failover in the *Oracle TimesTen In-Memory Database Operations Guide* and, in TimesTen Scaleout, see Client Connection Failover in the *Oracle TimesTen In-Memory Database Scaleout User's Guide*.
- Operations involving character comparisons support linguistic sensitive collating sequences. Case-insensitive sorts may affect DISTINCT value interpretation.
- Implicit and explicit conversions between CHAR and NCHAR are supported.
- You can use the SQL string functions with the supported character sets. For example, UPPER and LOWER functions support non-ASCII CHAR and VARCHAR2 characters as well as NCHAR and NVARCHAR2 characters.
- Choice of character set could have an impact on memory consumption for CHAR and VARCHAR2 column data.
- The character sets of all databases involved in a replication scheme must match.
- To add an existing table to an active standby pair, set DDL_REPLICATION_LEVEL to 2 or greater and DDL_REPLICATION_ACTION to INCLUDE. Alternatively, you can use the ALTER ACTIVE STANDBY PAIR ... INCLUDE TABLE statement if DDL_REPLICATION_ACTION is set to EXCLUDE. In this case, the table must be empty and present on all databases before executing the ALTER ACTIVE STANDBY PAIR ... INCLUDE TABLE statement as the table contents will be truncated when this statement is executed.
- To add an existing sequence or view to an active standby pair, set DDL_REPLICATION_LEVEL to 3. To include the sequence in the replication scheme, DDL_REPLICATION_ACTION must be set to INCLUDE. This does not apply to materialized views.

- Objects are replicated only when the receiving database is of a TimesTen release that supports that level of replication, and is configured for an active standby pair replication scheme. For example, replication of sequences (requiring DDL_REPLICATION_LEVEL=3) to a database release prior to 11.2.2.7.0 is not supported. The receiving database must be of at least release 11.2.1.8.0 for replication of objects supported by DDL_REPLICATION_LEVEL=2.

Examples

Use the ALTER SESSION statement to change COMMIT_BUFFER_SIZE_MAX to 500 MB. First, call ttConfiguration to display the current connection setting. Use the ALTER SESSION statement to change the COMMIT_BUFFER_SIZE_MAX setting to 500. Call ttConfiguration to display the new setting.

```
Command>  CALL ttConfiguration ('CommitBufferSizeMax');
< CommitBufferSizeMax, 0 >
1 row found.
Command> ALTER SESSION SET COMMIT_BUFFER_SIZE_MAX = 500;

Session altered.

Command> CALL ttConfiguration ('CommitBufferSizeMax');
< CommitBufferSizeMax, 500 >
1 row found.
```

Use the ALTER SESSION statement to change PLSQL_TIMEOUT to 60 seconds. Use a second ALTER SESSION statement to change PLSQL_OPTIMIZE_LEVEL to 3. Then call ttConfiguration to display the new values.

```
Command> ALTER SESSION SET PLSQL_TIMEOUT = 60;
Session altered.
Command> ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;
Session altered.

Command> CALL TTCONFIGURATION ();
< CkptFrequency, 600 >
< CkptLogVolume, 0 >
< CkptRate, 0 >
...
< PLSQL_OPTIMIZE_LEVEL, 3 >
< PLSQL_TIMEOUT, 60 >
...
47 rows found.
```

In this example, set PLSQL_TIMEOUT to 20 seconds. Attempt to execute a program that loops indefinitely. In 20 seconds, execution is terminated and an error is returned.

```
Command> ALTER SESSION SET PLSQL_TIMEOUT = 20;

Command> DECLARE v_timeout NUMBER;
       BEGIN
         LOOP
           v_timeout :=0;
           EXIT WHEN v_timeout < 0;
         END LOOP;
       END;
       /
 8509: PL/SQL execution terminated; PLSQL_TIMEOUT exceeded
```

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;
(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 >
< 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.
```

The following example enables automatic parallel replication with disabled commit dependencies. It uses the ALTER SESSION statement to change the replication track number to 5 for the current connection. To enable automatic parallel replication with disabled commit dependencies for replication schemes, set ReplicationApplyOrdering to 2. Then, always set REPLICATION_TRACK to a number less than or equal to ReplicationParallelism. For example, the ReplicationParallelism connection attribute could be set to 6, which is higher than the value of 5 set for REPLICATION_TRACK.

```
Command> ALTER SESSION SET REPLICATION_TRACK = 5;
Session altered.
```

The following example enables replication of adding and dropping columns, tables, synonyms and indexes by setting the following on the active database in an alter standby replication pair: DDL_REPLICATON_LEVEL set to 2 and DDLReplicationAction set to 'INCLUDE'.

```
Command > ALTER SESSION SET DDL_REPLICATION_LEVEL=2;
Session altered.

Command > ALTER SESSION SET DDL_REPLICATION_ACTION='INCLUDE';
Session altered.
```

> **Note**
>
> The equivalent connection attributes for DDL_REPLICATION_LEVEL and DDL_REPLICATION_ACTION are DDLReplicationLevel and DDLReplicationAction, respectively.

ALTER TABLE

The ALTER TABLE statement changes an existing table definition.

The ALTER TABLE statement is supported in TimesTen Scaleout and in TimesTen Classic. However, there are differences in syntax and semantics. For simplicity, the supported syntax, parameters, description (semantics), and examples for TimesTen Scaleout and for TimesTen Classic are separated into the usage with TimesTen Scaleout and the usage with TimesTen Classic. While there is repetition in the usages, it is presented this way in order to allow you to progress from syntax to parameters to semantics to examples for each usage.

Review the required privilege section and then see:

- ALTER TABLE: Usage with TimesTen Scaleout
- ALTER TABLE: Usage with TimesTen Classic

Required Privilege

No privilege is required for the table owner.

ALTER ANY TABLE for another user's table.

For ALTER TABLE...ADD FOREIGN KEY, the owner of the altered table must have the REFERENCES privilege on the table referenced by the foreign key clause.

After reviewing this section, see:

- ALTER TABLE: Usage with TimesTen Scaleout
- ALTER TABLE: Usage with TimesTen Classic

ALTER TABLE: Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. Column-based compression and aging are not supported.

See:

- SQL Syntax for ALTER TABLE: TimesTen Scaleout
- Parameters for ALTER TABLE ADD CONSTRAINT PRIMARY KEY: TimesTen Scaleout
- Parameters for ALTER TABLE ADD UNIQUE CONSTRAINT: TimesTen Scaleout

- Additional Parameters for ALTER TABLE: TimesTen Scaleout
- Description for ALTER TABLE ADD PRIMARY KEY: TimesTen Scaleout
- Description for ALTER TABLE ADD UNIQUE: TimesTen Scaleout
- Additional ALTER TABLE Information: TimesTen Scaleout
- Examples: Add Primary Key Constraint Using Global Indexes in TimesTen Scaleout
- Examples: Add Unique Constraint Using Global Indexes in TimesTen Scaleout
- Additional Examples for ALTER TABLE: TimesTen Scaleout

ALTER TABLE: Usage with TimesTen Classic

See:

- SQL Syntax for ALTER TABLE: TimesTen Classic
- Parameters for ALTER TABLE: TimesTen Classic
- Description for ALTER TABLE: TimesTen Classic
- Examples for ALTER TABLE: TimesTen Classic

SQL Syntax for ALTER TABLE: TimesTen Scaleout

To change the distribution key in TimesTen Scaleout:

```
ALTER TABLE [Owner.]TableName DistributionClause;
```

To add a primary key constraint and optionally specify a global or local index:

Note: The (*CreateIndexStmt*) is the clause used to represent the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a primary key constraint. See "CREATE INDEX" for details.

```
ALTER TABLE [Owner.]TableName ADD CONSTRAINT ConstraintName
  PRIMARY KEY (ColumnName [,... ]) [(UsingIndexClause2)];

UsingIndexClause2::= USING INDEX {GLOBAL|LOCAL} [USE HASH INDEX PAGES=RowPages|CURRENT]|
USING INDEX (CreateIndexStmt)
```

> **Note**
>
> You cannot use ALTER TABLE to drop a primary key constraint. To drop the constraint, you must drop and recreate the table.

To add a unique constraint on a column and optionally specify a global or local index:

Note: The (*CreateIndexStmt*) is the clause used to represent the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a unique constraint. See CREATE INDEX for details.

```
ALTER TABLE Owner.]TableName
ADD UNIQUE (ColumnName)
[UsingIndexClause1];

UsingIndexClause1::= USING INDEX {GLOBAL | LOCAL}| USING INDEX (CreateIndexStmt)
```

To add one column:

```
ALTER TABLE [Owner.]TableName
  ADD [COLUMN] ColumnName ColumnDataType
    [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL]
  [COMPRESS (CompressColumns [,...])];
```

To add multiple columns:

```
ALTER TABLE [Owner.]TableName
 ADD (ColumnName ColumnDataType
     [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL] [,... ] );
```

To add a NOT NULL column (note that the DEFAULT clause is required):

```
ALTER TABLE [Owner.]TableName
  ADD [COLUMN] ColumnName ColumnDataType
    NOT NULL [ENABLE] DEFAULT DefaultVal [[NOT] INLINE] [UNIQUE];
```

To add multiple NOT NULL columns (note that the DEFAULT clause is required):

```
ALTER TABLE [Owner.]TableName
ADD (ColumnName ColumnDataType
  NOT NULL [ENABLE] DEFAULT DefaultVal [[NOT] INLINE] [UNIQUE] [,...]);
```

To remove columns.

```
ALTER TABLE [Owner.]TableName
DROP {[COLUMN] ColumnName | (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 range index with the USE RANGE INDEX clause:

```
ALTER TABLE [Owner.]TableName USE RANGE INDEX;
```

To change the default value of a column:

```
ALTER TABLE [Owner.]TableName
MODIFY (ColumnName DEFAULT DefaultVal);
```

To drop a unique constraint on a column:

```
ALTER TABLE Owner.]TableName 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);

Parameters for ALTER TABLE ADD CONSTRAINT PRIMARY KEY: TimesTen Scaleout

Parameter	Description
ALTER TABLE [*Owner*.]TableName	Start of ALTER TABLE statement. Name of table required. Owner i optional.
ADD CONSTRAINT *ConstraintName* PRIMARY KEY	Clause indicating that the table is to be altered by adding a primary key constraint. *ConstraintName* is the name of the constraint. Once you add the primary key constraint, you cannot drop it. You must drop the table.
(*ColumnName* [,…])	(*ColumnName*) is required and specifies the column(s) to use for the primary key constraint.
[*UsingIndexClause2*]	*UsingIndexClause2* is optional and is described in the remainder of this table. You cannot specify two USING INDEX clauses in the ALTER TABLE definition.
USING INDEX {GLOBAL|LOCAL}	Part of [*UsingIndexClause2*]: If specified, indicates if a global or local index is to be created for the primary key.
USE HASH INDEX PAGES = *RowPages*|CURRENT	Part of the USING INDEX {GLOBAL|LOCAL} clause and is optional. If specified, indicates a unique hash index is to be created for the primary key. If not specified, a unique range index is created. Can be used for both global and local indexes. The PAGES clause is required and enables you to specify the expected page count value for the table. If you specify *RowPages*, the number of pages is used to calculate the page count value. To determine the value for *RowPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for RowPages (256000/256=1000). The value for *RowPages* must be a positive constant and must be greater than 0. If you specify CURRENT, the current number of rows in the table is used to calculate the page count value. TimesTen recommends that you do not specify PAGES=CURRENT if there are no rows in your table.
USING INDEX (*CreateIndexStmt*)	Part of the [*UsingIndexClause2*] clause. When this USING INDEX clause is specified, the (*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a primary key constraint. See CREATE INDEX for information on the CREATE INDEX statement.

Parameters for ALTER TABLE ADD UNIQUE CONSTRAINT: TimesTen Scaleout

Parameter	Description
ALTER TABLE [*Owner*.]TableName	Start of ALTER TABLE statement. Name of table required. Owner is optional.
ADD UNIQUE (*ColumnName* [,...])	Clause indicating that the table is to be altered by adding a unique constraint. (*ColumnName*) is required and specifies the column(s) to be used for the unique constraint.
[*UsingIndexClause1*]	*UsingIndexClause1* is optional and is described in the remainder of this table. You cannot specify two USING INDEX clauses in the ALTER TABLE definition. This clause enables you to define a global or local index for the PRIMARY KEY
USING INDEX {GLOBAL|LOCAL}	Part of [*UsingIndexClause1*]. If specified, indicates if a global or local index is to be created for the unique constraint.
USING INDEX (*CreateIndexStmt*)	Part of the [*UsingIndexClause1*] clause. When this USING INDEX clause is specified, the (*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a unique constraint. See CREATE INDEX for information on the CREATE INDEX statement.

Additional Parameters for ALTER TABLE: TimesTen Scaleout

Parameter	Description
[*Owner*.] *TableName*	Identifies the table to be altered.
DistributionClause	See "CREATE TABLE" for information on syntax.
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. Currently, you cannot assign a default value for the ROWID data type. Altering the default value of a column has no impact on existing rows. **Note:** To add a NOT NULL column to a table that is part of a replication scheme, DDL_REPLICATON_LEVEL must be 3 or greater.

Parameter	Description
ColumnName	Name of the column participating in the ALTER TABLE statement. A new column cannot have the same name as an existing column or another new column. If you add a NOT NULL column, you must include the DEFAULT clause.
ColumnDataType	Type of the column to be added. Some types require additional parameters. See "Data Types" for the data types that can be specified.
NOT NULL [ENABLE]	If you add a column, you can specify NOT NULL. If you specify NOT NULL, then you must include the DEFAULT clause. Optionally, you can specify ENABLE after the NOT NULL clause. Because NOT NULL constraints are always enabled, you are not required to specify ENABLE.
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.
DROP UNIQUE (*ColumnName*)	Indicates that unique constraint is to dropped. *ColumnName* is the name of the constraint.
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 clause.
FOREIGN KEY	Specifies that a foreign key is to be added.
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	Changes primary key to use a hash index. If the primary key already uses a hash index, then this clause is equivalent to the SET PAGES clause.
USE RANGE INDEX	Changes primary key to use a range index. If the primary key already uses a range index, TimesTen ignores this clause.

Parameter	Description
SET PAGES = *RowPages* | CURRENT	Resizes the hash index to reflect the expected number of pages in the table. If you specify CURRENT, the current number of rows in the table is used to calculate the page count value. If you specify *RowPages*, the number of pages is used. To determine the value for *RowPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for *RowPages* (256000/256=1000). The value for *RowPages* must be a positive constant and must be greater than 0. TimesTen recommends that you do not specify PAGES=CURRENT if there are no rows in your table. If your estimate is too small, performance may be degraded. See Column Definition: TimesTen Scaleout for more information on hash indexes.

Description for ALTER TABLE ADD PRIMARY KEY: TimesTen Scaleout

You have the option of specifying an additional clause after the PRIMARY KEY clause in your ALTER TABLE definition. This clause enables you to specify a global or local index for the primary key constraint.

- The USING INDEX {GLOBAL | LOCAL} clause is one option that enables you to specify a global or local index for the primary key constraint. You must specify the GLOBAL or the LOCAL keyword. You can optionally specify the USE HASH INDEX clause after the USING INDEX {GLOBAL | LOCAL} clause if you want to define a hash index.
- The USING INDEX (*CreateIndexStmt*) clause is your other option for specifying a global or local index. The(*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a primary key constraint. See CREATE INDEX for information on the CREATE INDEX statement.

> **Note**
>
> You cannot use both the USING INDEX {GLOBAL | LOCAL} and the USING INDEX (*CreateIndexStmt*) in the ALTER TABLE definition. Specify one clause or the other or specify neither.

See CREATE INDEX for information on global and local indexes and their use in TimesTen Scaleout.

Description for ALTER TABLE ADD UNIQUE: TimesTen Scaleout

You have the option of specifying an additional clause after the UNIQUE clause in your ALTER TABLE definition. This clause enables you to specify a global or local index for the unique constraint.

- The USING INDEX {GLOBAL | LOCAL} clause is one option that enables you to specify a global or local index for the primary key constraint. You must specify the GLOBAL or the LOCAL keyword.

- The USING INDEX (*CreateIndexStmt*) clause is your other option for specifying a global or local index. The(*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a unique constraint. See CREATE INDEX for information on the CREATE INDEX statement.

> **Note**
>
> You cannot use both the USING INDEX {GLOBAL | LOCAL} and the USING INDEX (*CreateIndexStmt*) in the ALTER TABLE definition. Specify one clause or the other or specify neither.

See CREATE INDEX for information on global and local indexes and their use in TimesTen Scaleout.

Additional ALTER TABLE Information: TimesTen Scaleout

- You can alter tables to change defaults or add and drop columns and constraints. However, you cannot change the distribution scheme unless the table is empty. In addition, you cannot drop a constraint that is named in the DISTRIBUTE BY REFERENCE clause. See CREATE TABLE for information on the distribution schemes. See Altering Tables in *Oracle TimesTen In-Memory Database Scaleout User's Guide* for more information.
- The ALTER TABLE statement cannot be used to alter a temporary table.
- The ALTER TABLE ADD [COLUMN] *ColumnName* statement adds one or more new columns to an existing table. When you add one or more columns, 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.
- You cannot use the ALTER TABLE statement to add a column, drop a column, or add a constraint for cache group tables.
- Only one partition is added to the table per statement regardless of the number of columns added.
- You can ALTER a table to add a NOT NULL column with a default value. The DEFAULT clause is required.

  You cannot use the column as a primary key column. Specifically, you cannot specify the column in the statement: ALTER TABLE ADD *ConstraintName* PRIMARY KEY (*ColumnName* [,...]).
- 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 1000. In addition, the total number of partitions in a table cannot exceed 1000, one of which is used by TimesTen.
- Do not specify the ADD CONSTRAINT ... PRIMARY KEY clause on a global temporary 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 -relaxedUpgrade command, and restoring the table using the ttRestore -r -relaxedUpgrade command. Dropping the added column does not recover the lost performance or decrease the number of partitions.
- When you use the ALTER TABLE DROP statement to remove one or more columns from an existing table, 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. You cannot drop columns that are in the table's primary key. You cannot drop columns that are in any of the table's foreign keys until you have dropped all foreign keys. You cannot drop columns that are indexed 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 RANGE 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 RANGE INDEX clause is specified.
- If ON DELETE CASCADE is specified on a foreign key constraint for a child table, a user can delete rows from a parent table for which the user has the DELETE privilege without requiring explicit DELETE privilege on the child table.
- To change the ON DELETE CASCADE triggered action, drop then redefine the foreign key constraint.
- 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 causes cascaded deletes in the child table. This, in turn, triggers 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.
- 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 views or temporary tables.
- You cannot use ALTER TABLE to drop a primary key constraint. You would have to drop and recreate the table in order to drop the constraint.

Examples: Add Primary Key Constraint Using Global Indexes in TimesTen Scaleout

These examples show various uses of the syntax for using global indexes with ALTER TABLE ADD PRIMARY KEY.

Create a table. Alter the table and add a primary key constraint. Specify the USING INDEX GLOBAL clause. Drop the table.

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
         a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD CONSTRAINT pk PRIMARY KEY (c,b) USING INDEX GLOBAL;
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  PK: global unique range index on columns:
    C
    B
  1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab1;
```

Create a table. Alter the table adding a primary key constraint. Specify the USING INDEX GLOBAL with the USE HASH INDEX PAGES clause. Drop the table.

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
         a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD CONSTRAINT pk PRIMARY KEY (c,b)
         USING INDEX GLOBAL USE HASH INDEX PAGES =200;
Command> INDEXES mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  PK: global unique hash index on columns:
    C
    B
  1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab1;
```

Create a table. Alter the table adding a primary key constraint. Specify the USING INDEX (CreateIndexStmt) clause. The (CreateIndexStmt) clause is the TimesTen CREATE INDEX statement. See CREATE INDEX for information on this statement.

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
         a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD CONSTRAINT pk PRIMARY KEY (c,b)
```

```
        USING INDEX (CREATE GLOBAL UNIQUE HASH INDEX myglobalix ON mytab1 (c,b) PAGES =200);
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  MYGLOBALIX: global unique hash index on columns:
    C
    B
  1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab1;
```

This example illustrates that you cannot use the `USING INDEX GLOBAL|LOCAL` clause with the `USING INDEX (CreateIndexStmt)` clause.

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
        a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD CONSTRAINT pk PRIMARY KEY (c,b)
        USING INDEX GLOBAL USE HASH INDEX PAGES = 200
        USING INDEX (CREATE GLOBAL UNIQUE HASH INDEX myglobalix ON mytab1 (c,b) PAGES
=200);
 1001: Syntax error in SQL statement  before or at: "USING", character position: 102
...USING INDEX GLOBAL USE HASH INDEX PAGES = 200 USING INDEX (CREATE G...
                                                 ^^^^^
The command failed.
```

Examples: Add Unique Constraint Using Global Indexes in TimesTen Scaleout

These examples show various uses of the syntax for using global indexes with `ALTER TABLE ADD UNIQUE CONSTRAINT`.

Create a table. Alter the table adding a unique constraint. Drop the table. Create the table again adding a unique constraint and specifying the `USING INDEX GLOBAL` clause.

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
        a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD UNIQUE (a);
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  TTUNIQUE_6E6: unique range index on columns:
    A
  1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab1;
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
        a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD UNIQUE (a) USING INDEX GLOBAL;
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  $GUA8C5B4ECE6D8: global unique range index on columns:
    A
  1 index found.
```

```
1 index found on 1 table.
Command> DROP TABLE mytab1;
```

Create a table. Alter the table adding a unique constraint and use the USING INDEX (CreateIndexStmt) clause to create a local unique index. Alter the table a second time adding another unique constaint. Use the USING INDEX (CreateIndexStmt) clause to create a global unique index.

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
         a TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> ALTER TABLE mytab1 ADD UNIQUE (b) USING INDEX (CREATE UNIQUE INDEX
myuniqueidxB ON mytab1 (b));
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  MYUNIQUEIDXB: unique range index on columns:
    B
  1 index found.

1 index found on 1 table.
Command> ALTER TABLE mytab1 ADD UNIQUE (c) USING INDEX (CREATE GLOBAL UNIQUE INDEX
myuniqueidxC ON mytab1 (c));
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  MYUNIQUEIDXB: unique range index on columns:
    B
  MYUNIQUEIDXC: global unique range index on columns:
    C
  2 indexes found.

2 indexes found on 1 table.
Command> DROP TABLE mytab1;
```

Additional Examples for ALTER TABLE: TimesTen Scaleout

Table 6-6 shows the rules associated with altering tables. Supporting examples follow.

Table 6-6 ALTER TABLE Rules

ALTER Statement	Comment
ALTER TABLE t1 ADD CONSTRAINT c1 PRIMARY KEY (p);	The primary key constraint is added to the table. The distribution key is not changed.
CREATE TABLE t1 (c1 NUMBER, c2 VARCHAR2 (10)); ALTER TABLE t1 DISTRIBUTE BY HASH (c1);	The operation succeeds if the table is empty. If the table is not empty, the operation fails because the distribution key cannot be changed on tables that are not empty.

Table 6-6 (Cont.) ALTER TABLE Rules

ALTER Statement	Comment
ALTER TABLE t1 ADD CONSTRAINT c1 FOREIGN KEY (f1)REFERENCES t2 (c2);	The operation succeeds. The distribution of the t1 table is not related to the c1 constraint.
CREATE TABLE t1...CONSTRAINT fk1... DISTRIBUTE BY REFERENCE(fk1); ALTER TABLE t1 DROP CONSTRAINT(fk1);	The operation fails. The foreign key is used to distribute the table.

These examples support the information in the Table 6-6 table:

- Use ALTER TABLE to Add a Primary Key Constraint
- Add a Primary Key Constraint on Table Distributed on Unique Column
- Use ALTER TABLE to Change a Distribution Key
- Add a Foreign Key Constraint That Is Not Part of a Distribution Key
- Attempt to Drop a Foreign Key Constraint Used as a Distribution Key

Use ALTER TABLE to Add a Primary Key Constraint

This example creates the mytable table without a primary key or distribution clause. The table is distributed by hash on a hidden column. Then the ALTER TABLE statement is used to add a primary key constraint. The operation succeeds but the distribution key is not changed.

```
Command> CREATE TABLE mytable (col1 NUMBER NOT NULL, col2 VARCHAR2 (32));
Command> describe mytable;

Table SAMPLEUSER.MYTABLE:
  Columns:
    COL1                        NUMBER NOT NULL
    COL2                        VARCHAR2 (32) INLINE
  DISTRIBUTE BY HASH

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

Now alter the table to add the primary key. The operation succeeds. The distribution scheme and distribution key do not change.

```
Command> ALTER TABLE mytable ADD CONSTRAINT c1 PRIMARY KEY (col1);
Command> describe mytable;

Table SAMPLEUSER.MYTABLE:
  Columns:
   *COL1                        NUMBER NOT NULL
    COL2                        VARCHAR2 (32) INLINE
  DISTRIBUTE BY HASH

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

Add a Primary Key Constraint on Table Distributed on Unique Column

This example creates the mytab table and distributes the data by hash on the id2 unique column. The example then alters the mytab table adding the primary key constraint on the id column. A ttIsql describe command shows the table remains distributed by hash on the id2 column.

```
Command> CREATE TABLE mytab (id TT_INTEGER NOT NULL, id2 TT_INTEGER UNIQUE,
         id3 TT_INTEGER) distribute by hash (id2);
Command> ALTER TABLE mytab ADD CONSTRAINT c1 PRIMARY KEY (id);              Command> describe mytab;

Table SAMPLEUSER.MYTAB:
  Columns:
   *ID                             TT_INTEGER NOT NULL
    ID2                            TT_INTEGER UNIQUE
    ID3                            TT_INTEGER
  DISTRIBUTE BY HASH (ID2)

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

Use ALTER TABLE to Change a Distribution Key

This example shows that you can use the ALTER TABLE statement to change the distribution key, but only if the table is empty.

```
Command> CREATE TABLE mytable2 (col1 NUMBER NOT NULL, col2 VARCHAR2 (32))
        DISTRIBUTE BY HASH (col1,col2);
Command> describe mytable2;

Table SAMPLEUSER.MYTABLE2:
  Columns:
    COL1                            NUMBER NOT NULL
    COL2                            VARCHAR2 (32) INLINE
  DISTRIBUTE BY HASH (COL1, COL2)

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

Use the ALTER TABLE statement to change the distribution key to col1. The operation succeeds because the table is empty.

```
Command> ALTER TABLE mytable2 DISTRIBUTE BY HASH (col1);
Command> describe mytable2;

Table SAMPLEUSER.MYTABLE2:
  Columns:
    COL1                            NUMBER NOT NULL
    COL2                            VARCHAR2 (32) INLINE
  DISTRIBUTE BY HASH (COL1)

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

Insert a row of data and attempt to change the distribution key back to col1, col2. The operation fails because the table is not empty.

```
Command> INSERT INTO mytable2 VALUES (10, 'test');
1 row inserted.
Command> commit;
Command> ALTER TABLE mytable2 DISTRIBUTE BY HASH (col1,col2);
 1069: Table not empty. Alter table distribution is only permitted on empty
```

```
tables.
The command failed.
```

Add a Foreign Key Constraint That Is Not Part of a Distribution Key

This example first describes the accounts and accounts2 tables. The example then alters the accounts2 table, adding a foreign key constraint. Since this constraint is not part of the accounts2 table distribution, the operation succeeds.

```
Command> describe accounts;

Table SAMPLEUSER.ACCOUNTS:
  Columns:
   *ACCOUNT_ID                      NUMBER (10) NOT NULL
    PHONE                           VARCHAR2 (15) INLINE NOT NULL
    ACCOUNT_TYPE                    CHAR (1) NOT NULL
    STATUS                          NUMBER (2) NOT NULL
    CURRENT_BALANCE                 NUMBER (10,2) NOT NULL
    PREV_BALANCE                    NUMBER (10,2) NOT NULL
    DATE_CREATED                    DATE NOT NULL
    CUST_ID                         NUMBER (10) NOT NULL
  DISTRIBUTE BY REFERENCE (FK_CUSTOMER)

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

Command> describe accounts2;

Table SAMPLEUSER.ACCOUNTS2:
  Columns:
   *ACCOUNTS2_ID                    NUMBER (10) NOT NULL
    ACCOUNT_ORIG_ID                 NUMBER (10) NOT NULL
    STATUS                          NUMBER (2) NOT NULL
  DISTRIBUTE BY HASH (ACCOUNTS2_ID)

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

Command> ALTER TABLE accounts2 ADD CONSTRAINT accounts2_fk FOREIGN KEY
          (account_orig_id) REFERENCES accounts (account_id);
```

Use the ttIsql indexes command to show the accounts2_fk constraint is created successfully.

```
Command> indexes accounts2;

Indexes on table SAMPLEUSER.ACCOUNTS2:
  ACCOUNTS2: unique range index on columns:
    ACCOUNTS2_ID
  ACCOUNTS2_FK: non-unique range index on columns:
    ACCOUNT_ORIG_ID
    (foreign key index references table SAMPLEUSER.ACCOUNTS(ACCOUNT_ID))
  2 indexes found.

2 indexes found on 1 table.
```

Attempt to Drop a Foreign Key Constraint Used as a Distribution Key

This example attempts to drop the fk_accounts constraint. Since the constraint is used as the distribution key, the operation fails.

```
Command> describe transactions;
```

```
Table SAMPLEUSER.TRANSACTIONS:
  Columns:
   *TRANSACTION_ID                NUMBER (10) NOT NULL
   *ACCOUNT_ID                    NUMBER (10) NOT NULL
   *TRANSACTION_TS                TIMESTAMP (6) NOT NULL
    DESCRIPTION                   VARCHAR2 (60) INLINE
    OPTYPE                        CHAR (1) NOT NULL
    AMOUNT                        NUMBER (6,2) NOT NULL
  DISTRIBUTE BY REFERENCE (FK_ACCOUNTS)

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

Command> ALTER TABLE transactions DROP CONSTRAINT fk_accounts;
 1072: Dropping a table's reference by distribution foreign key is not allowed.
The command failed.
```

SQL Syntax for ALTER TABLE: TimesTen Classic

To add one column:

```
ALTER TABLE [Owner.]TableName ADD [COLUMN] ColumnName ColumnDataType
  [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL]
  [COMPRESS (CompressColumns [,...])];
```

To add multiple columns:

```
ALTER TABLE [Owner.]TableName ADD (ColumnName ColumnDataType
  [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL] [,... ] )
  [COMPRESS (CompressColumns [,...])];
```

To add a NOT NULL column (note that the DEFAULT clause is required):

```
ALTER TABLE [Owner.]TableName
  ADD [COLUMN] ColumnName ColumnDataType
    NOT NULL [ENABLE] DEFAULT DefaultVal [[NOT] INLINE] [UNIQUE]
  [COMPRESS (CompressColumns [,...])]
```

To add multiple NOT NULL columns (note that the DEFAULT clause is required):

```
ALTER TABLE [Owner.]TableName
  ADD (ColumnName ColumnDataType
       NOT NULL [ENABLE] DEFAULT DefaultVal [[NOT] INLINE] [UNIQUE] [,...])
  [COMPRESS (CompressColumns [,...])]
```

The *CompressColumns* syntax is as follows:

```
{ColumnDefinition | (ColumnDefinition [,...])} BY DICTIONARY
   [MAXVALUES = CompressMax]
```

To remove columns.

```
ALTER TABLE [Owner.]TableName
  DROP {[COLUMN] ColumnName | (ColumnName [,... ] )}
```

> **Note**
>
> If removing columns in a compressed column group, all columns in the compressed column group must be specified.

To add a primary key constraint using a range index:

ALTER TABLE [*Owner*.]*TableName* ADD CONSTRAINT *ConstraintName*
PRIMARY KEY (*ColumnName* [,...])

To add a primary key constraint using a hash index:

ALTER TABLE [*Owner*.]*TableName* ADD CONSTRAINT *ConstraintName*
PRIMARY KEY (*ColumnName* [,...])
USE HASH INDEX PAGES = *RowPages* | CURRENT

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*

> **Note**
>
> You cannot use ALTER TABLE to drop a primary key constraint. To drop the constraint, drop and recreate the table.

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 range index with the USE RANGE INDEX clause:

ALTER TABLE [*Owner*.]*TableName*
USE RANGE 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
   {SECOND[S] | MINUTE[S] | HOUR[S] | DAY[S]}
    [CYCLE num2 {SECOND[S] | 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
```

To change the lifetime for time-based aging:

```
ALTER TABLE [Owner.]TableName
SET AGING LIFETIME num1 {SECOND[S] | MINUTE[S] | HOUR[S] | DAY[S]}
```

To change the cycle for time-based aging:

```
ALTER TABLE [Owner.]TableName
SET AGING CYCLE num2 {SECOND[S] | MINUTE[S] | HOUR[S] | DAY[S]}
```

Parameters for ALTER TABLE: TimesTen Classic

Parameter	Description
[*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. Currently, you cannot assign a default value for the ROWID data type. Altering the default value of a column has no impact on existing rows. **Note:** To add a NOT NULL column to a table that is part of a replication scheme, DDL_REPLICATON_LEVEL must be 3 or greater.

Parameter	Description
ColumnName	Name of the column participating in the ALTER TABLE statement. A new column cannot have the same name as an existing column or another new column. If you add a NOT NULL column, you must include the DEFAULT clause.
ColumnDataType	Type of the column to be added. Some types require additional parameters. See Data Types for the data types that can be specified.
NOT NULL [ENABLE]	If you add a column, you can specify NOT NULL. If you specify NOT NULL, then you must include the DEFAULT clause. Optionally, you can specify ENABLE after the NOT NULL clause. Because NOT NULL constraints are always enabled, you are not required to specify ENABLE.
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.
COMPRESS (*CompressColumns* [,...])	Defines a compressed column group for a table that is enabled for compression. This can include one or more columns in the table. If you define multiple columns for a compression group, you must specify the columns as INLINE. An out-of-line column cannot be in a multi-column compression group. Each compressed column group is limited to a maximum of 16 columns. See Column-Based Compression of Tables (TimesTen Classic) for details on compression columns.
BY DICTIONARY	Defines a compression dictionary for each compressed column group.
MAXVALUES = *CompressMax*	*CompressMax* is the total number of distinct values in the table and sets the size for the compressed column group pointer column to 1, 2, or 4 bytes and sets the size for the maximum number of entries in the dictionary table. For the dictionary table, NULL is counted as one unique value. *CompressMax* takes an integer between 1 and 2^{32}-1. The maximum size defaults to size of 2^{32}-1 if the MAXVALUES clause is omitted, which uses 4 bytes for the pointer column. An error is thrown if the value is greater than 2^{32}-1. See Column-Based Compression of Tables (TimesTen Classic) for details on maximum sizing for compression dictionaries.

Parameter	Description
ADD CONSTRAINT *ConstraintName* PRIMARY KEY (*ColumnName* [,...]) [USE HASH INDEX PAGES = *RowPages* | CURRENT]	Adds a primary key constraint to the table. Columns of the primary key must be defined as NOT NULL. Specify *ConstraintName* as the name of the index used to enforce the primary key constraint. Specify *ColumnName* as the name(s) of the NOT NULL column(s) used for the primary key. Specify the USE HASH INDEX clause to use a hash index for the primary key. If not specified, a range index is used for the primary key constraint. If you specify CURRENT, the current number of rows in the table is used to calculate the page count value. If you specify *RowPages*, the number of pages is used. To determine the value for *RowPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for RowPages (256000/256=1000). The value for *RowPages* must be a positive constant and must be greater than 0. TimesTen recommends that you do not specify PAGES=CURRENT if there are no rows in your table. If your estimate is too small, performance may be degraded. See Column Definition: TimesTen Classic for more information on hash indexes. **Note:** Before you use ADD CONSTRAINT to add a named primary key constraint, be aware that you cannot use ALTER TABLE to drop a primary key constraint. You would have to drop and recreate the table in order to drop the constraint.
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 clause.
FOREIGN KEY	Specifies that a foreign key is to be added.
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	Changes primary key to use a hash index. If the primary key already uses a hash index, then this clause is equivalent to the SET PAGES clause.
USE RANGE INDEX	Changes primary key to use a range index. If the primary key already uses a range index, TimesTen ignores this clause.

Parameter	Description
SET PAGES = *RowPages* | CURRENT	Resizes the hash index to reflect the expected number of pages in the table. If you specify CURRENT, the current number of rows in the table is used to calculate the page count value. If you specify *RowPages*, the number of pages is used. To determine the value for *RowPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for *RowPages* (256000/256=1000). The value for *RowPages* must be a positive constant and must be greater than 0. TimesTen recommends that you do not specify PAGES=CURRENT if there are no rows in your table. If your estimate is too small, performance may be degraded. See Column Definition: TimesTen Classic for more information on hash indexes.
ADD AGING LRU [ON | OFF]	Adds least recently used (LRU) aging to an existing table that has no aging policy defined. The LRU aging policy defines the type of aging (least recently used (LRU)), the aging state (ON or OFF) and the LRU aging attributes. Set the aging state to either ON or OFF. ON indicates that the aging state is enabled and aging is done automatically. OFF indicates that the aging state is disabled and aging is not done automatically. In both cases, the aging policy is defined. The default is ON. LRU attributes are defined by calling the ttAgingLRUConfig and/or the ttAgingTableLRUConfig built-in procedures. LRU attributes are not defined at the SQL level. See ttAgingLRUConfig and ttAgingTableLRUConfig in the *Oracle TimesTen In-Memory Database Reference* and Implementing an Aging Policy in Your Tables in the *Oracle TimesTen In-Memory Database Operations Guide* for more information.

Parameter	Description
ADD AGING USE *ColumnName*...[ON| OFF]	Adds time-based aging to an existing table that has no aging policy defined. The time-based aging policy defines the type of aging (time-based), the aging state (ON or OFF) and the time-based aging attributes. Set the aging state to either ON or OFF. ON indicates that the aging state is enabled and aging is done automatically. OFF indicates that the aging state is disabled and aging is not done automatically. In both cases, the aging policy is defined. The default is ON. Time-based aging attributes are defined at the SQL level and are specified by the LIFETIME and CYCLE clauses. Specify *ColumnName* as the name of the column used for time-based aging. Define the column as NOT NULL and of data type TIMESTAMP or DATE. The value of this column is subtracted from SYSDATE, truncated using the specified unit (minute, hour, day) and then compared to the LIFETIME value. If the result is greater than the LIFETIME value, then the row is a candidate for aging. The values of the column used for aging 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 (the column cannot be NULL). You can define your aging column with a data type of TT_TIMESTAMP or TT_DATE. If you choose data type TT_DATE, then you must specify the LIFETIME unit as days. For more information about time-based aging, see Implementing an Aging Policy in Your Tables in *Oracle TimesTen In-Memory Database Operations Guide*.
LIFETIME *Num1* {SECOND[S] | MINUTE[S] | HOUR[S] | DAY[S]	Specify the LIFETIME clause after the ADD AGING USE *ColumnName* clause if you are adding the time-based aging policy to an existing table. Specify the LIFETIME clause after the SET AGING clause to change the LIFETIME setting. The LIFETIME clause specifies the minimum amount of time data is kept in cache. Specify *Num1* as a positive integer constant to indicate the unit of time expressed in seconds, minutes, hours or days that rows should be kept in cache. Rows that exceed the LIFETIME value are aged out (deleted from the table). If you define your aging column with data type TT_DATE, then you must specify DAYS as the LIFETIME unit. The concept of time resolution is supported. If 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. A LIFETIME of 3 days is different than a LIFETIME of 72 hours (3*24) or a LIFETIME of 432 minutes (3*24*60).

Parameter	Description
CYCLE *Num2* {SECOND[S]| MINUTE[S]| HOUR[S]|DAY[S]}	Specify the optional CYCLE clause after the LIFETIME clause if you are adding the time-based aging policy to an existing table. CYCLE is a time-based aging attribute. The CYCLE clause indicates how often the system should examine rows to see if data exceeds the specified LIFETIME value and should be aged out (deleted). Specify *Num2* as a positive integer constant. If you do not specify the CYCLE clause, then the default value is 5 minutes. If you specify 0 for *Num2*, then the aging thread wakes up every second. If the aging state is OFF, then aging is not done automatically and the CYCLE clause is ignored. Specify the CYCLE clause after the SET AGING clause to change the CYCLE setting.
SET AGING {ON|OFF}	Changes the aging state. The aging policy must be previously defined. ON enables automatic aging. OFF disables automatic aging. To control aging with an external scheduler, then disable aging and invoke the ttAgingScheduleNow built-in procedure.
DROP AGING	Drops the aging policy from the table. After you define an aging policy, you cannot alter it. Drop aging, then redefine.
SET AGING LIFETIME *Num1* {SECOND[S]| MINUTE[S]|HOUR[S] |DAY[S]}	Use this clause to change the lifetime for time-based aging. *Num1* must be a positive integer constant. If you defined your aging column with data type TT_DATE, then you must specify DAYS as the LIFETIME unit.
SET AGING CYCLE *Num2* {SECOND[S]| MINUTE[S]| HOUR[S]|DAY[S]}	Use this clause to change the cycle for time-based aging. *Num2* must be a positive integer constant.

Description for ALTER TABLE: TimesTen Classic

- The ALTER TABLE statement cannot be used to alter a temporary table.
- The ALTER TABLE ADD [COLUMN] *ColumnName* statement adds one or more new columns to an existing table. When you add one or more columns, the new columns are added to the end of all existing rows of the table in one new partition.
- The ALTER TABLE ADD or DROP COLUMN statement can be used to add or drop columns from replicated tables.

  Do not use ALTER TABLE to alter a replicated table that is part of a TWOSAFE BY REQUEST transaction.
- Columns referenced by materialized views cannot be dropped.
- You cannot use the ALTER TABLE statement to add a column, drop a column, or add a constraint for cache group tables.

- Only one partition is added to the table per statement regardless of the number of columns added.
- You can ALTER a table to add a NOT NULL column with a default value. The DEFAULT clause is required. Restrictions include:
  - You cannot use the column as a primary key column. Specifically, you cannot specify the column in the statement: ALTER TABLE ADD *ConstraintName* PRIMARY KEY (*ColumnName* [,...]).
  - You cannot use the column for time-based aging. Specifically, you cannot specify the column in the statement ALTER TABLE ADD AGING USE *ColumnName*.

> **Note**
>
> To add a NOT NULL column to a table that is part of a replication scheme, DDL_REPLICATON_LEVEL must be 3 or greater.

- 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 1000. In addition, the total number of partitions in a table cannot exceed 1000, one of which is used by TimesTen.
- Use the ADD CONSTRAINT ... PRIMARY KEY clause to add a primary key constraint to a regular table or to a detailed or materialized view table. Do not use this clause on a table that already has a primary key.
- If you use the ADD CONSTRAINT... PRIMARY KEY clause to add a primary key constraint, and you do not specify the USE HASH INDEX clause, then a range index is used for the primary key constraint.
- If a table is replicated and the replication agent is active, you cannot use the ADD CONSTRAINT ... PRIMARY KEY clause. Stop the replication agent first.
- Do not specify the ADD CONSTRAINT ... PRIMARY KEY clause on a global temporary table.
- Do not specify the ADD CONSTRAINT ... PRIMARY KEY clause on a cache group table because cache group tables defined with a primary key must be defined in the CREATE CACHE GROUP statement.
- 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 -relaxedUpgrade command, and restoring the table using the ttRestore -r -relaxedUpgrade command. Dropping the added column does not recover the lost performance or decrease the number of partitions.
- When you use the ALTER TABLE DROP statement to remove one or more columns from an existing table, 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. You cannot drop columns that are in the table's primary key. You cannot drop columns that are in any of the table's foreign keys until you have dropped all foreign keys. You cannot drop columns that are indexed 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 RANGE 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 RANGE INDEX clause is specified.
- Make sure to stop the replication agent before adding or dropping a foreign key on a replicated table.
- If ON DELETE CASCADE is specified on a foreign key constraint for a child table, a user can delete rows from a parent table for which the user has the DELETE privilege without requiring explicit DELETE privilege on the child table.
- To change the ON DELETE CASCADE triggered action, drop then redefine the foreign key constraint.
- 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 causes cascaded deletes in the child table. This, in turn, triggers 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 is done at runtime. If there is an error, the receiver thread stops working.
  - All tables in the delete cascade tree have to be replicated if any table in the tree is replicated. This restriction is 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.

- 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 views or temporary tables.
  - You cannot use ALTER TABLE to drop a primary key constraint. You would have to drop and recreate the table in order to drop the constraint.
- After you have defined an aging policy for the table, you cannot change the policy from LRU to time-based or from time-based to LRU. You must first drop aging and then alter the table to add a new aging policy.
- The aging policy must be defined to change the aging state.
- The following rules determine if a row is accessed or referenced for LRU aging:
  - 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.
- 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.
- Call the ttAgingScheduleNow procedure to schedule the aging process right away regardless if the aging state is ON or OFF.
- For the time-based aging policy, you cannot add or modify the aging column. This is because you cannot add or modify a NOT NULL column.
- Aging restrictions:
  - You cannot drop the column that is used for time-based aging.
  - Tables that are related by foreign keys must have the same aging policy.
  - For LRU aging, if a child row is not a candidate for aging, neither this child row nor its parent row are 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.
- Restrictions for column-based compression of tables:
  - You can add compressed column groups with the ALTER TABLE statement only if the table was enabled for compression at table creation. You can add uncompressed columns to any table, including tables enabled for compression. Refer to Column-Based Compression of Tables (TimesTen Classic) for more details on adding compressed column groups to a table.
  - You cannot modify columns of a compressed column group.
  - You can drop all columns within a compressed column group with the ALTER TABLE command; when removing columns in a compressed column group, all columns in the compressed column group must be specified for removal.
  - You cannot use ALTER TABLE to modify an existing uncompressed column to make it compressed. For example:

```
Command> create table mytab (a varchar2 (30), b int, c int) compress ((a,b)
        by dictionary);
Command> alter table mytab add (d int) compress (c by dictionary);
2246: Cannot change compression clause for already defined column C
The command failed.
```

Understanding Partitions When Using ALTER TABLE in TimesTen

When you create a table, an initial partition is created. If you ALTER the table, and add additional columns, secondary partitions are created. There is one secondary partition created for each ALTER TABLE statement. For a column in secondary partitions, you cannot create a primary key constraint on the column or use the column for time-based aging.

You can use ttMigrate -r -relaxedUpgrade to condense multiple partitions. This means the initial partition plus one or more secondary partitions are condensed into a single partition called the initial partition. Once you condense the partitions, you can then ALTER the table and add a primary key constraint on the column or use the column for time-based aging. This is because the columns are no longer in secondary partitions but are now in the initial partition.

If your database is involved in replication and you want to condense multiple partitions, you must use the StoreAttribute TABLE DEFINITION CHECKING RELAXED (of the CREATE REPLICATION statement). Run ttMigrate -r -relaxedUpgrade on both the master and subscriber or on either the master or subscriber by using -duplicate.

Use ttSchema to view partition numbers for columns. ttSchema displays secondary partition number 1 as partition 1, secondary partition number 2 as partition 2 and so on.

As an example, create a table *MyTab* with 2 columns. Then ALTER the table adding 2 columns (*Col3* and *Col4*) with the NOT NULL DEFAULT clause.

```
Command> CREATE TABLE MyTab (Col1 NUMBER, Col2 VARCHAR2 (30));
Command> ALTER TABLE MyTab ADD (Col3 NUMBER NOT NULL DEFAULT 10, Col4 TIMESTAMP
         NOT NULL DEFAULT TIMESTAMP '2012-09-03 12:00:00');
```

Use ttSchema to verify *Col3* and *Col4* are in secondary partition 1.

```
ttschema -DSN sampledb_1122
-- Database is in Oracle type mode
create table TESTUSER.MYTAB (
     COL1 NUMBER,
     COL2 VARCHAR2(30 BYTE) INLINE,
     COL3 NUMBER NOT NULL DEFAULT 10,
     COL4 TIMESTAMP(6) NOT NULL DEFAULT TIMESTAMP '2012-09-03 12:00:00');
-- column COL3 partition 1
-- column COL4 partition 1
```

Attempt to add a primary key constraint on *Col3* and time-based aging on *Col4*. You see errors because you can neither add a primary key constraint nor add time-based aging to a column that is not in the initial partition.

```
Command> ALTER TABLE MyTab ADD CONSTRAINT PriKey PRIMARY KEY (Col3);
 2419: All columns in a primary key constraint must be in the initial partition;
column COL3 was added by ALTER TABLE
The command failed.

Command> ALTER TABLE MyTab ADD AGING USE Col4 LIFETIME 3 DAYS;
 3023: Aging column must be in the initial partition; column COL4 was added by
ALTER TABLE
The command failed.
```

Use ttMigrate with the -relaxedUpgrade option to condense the partitions. Then use ttSchema to verify the partitions are condensed and there are no columns in secondary partition 1.

```
ttMigrate -c dsn=sampledb_1122 test.migrate

Saving user PUBLIC
User successfully saved.

Saving table TESTUSER.MYTAB
  Saving rows...
  0/0 rows saved.
Table successfully saved.

ttDestroy sampledb_1122

ttMigrate -r -relaxedUpgrade
 dsn=sampledb_1122 test.migrate

Restoring table TESTUSER.MYTAB
  Restoring rows...
  0/0 rows restored.
Table successfully restored.

ttSchema DSN=sampledb_1122
-- Database is in Oracle type mode
create table TESTUSER.MYTAB (
        COL1 NUMBER,
        COL2 VARCHAR2(30 BYTE) INLINE,
        COL3 NUMBER NOT NULL DEFAULT 10,
        COL4 TIMESTAMP(6) NOT NULL DEFAULT TIMESTAMP '2012-09-03 12:00:00');
```

Now add a primary key constraint on *Col3* and time-based aging on *Col4*. The results are successful because *Col3* and *Col4* are in the initial partition as a result of ttMigrate. Use ttSchema to verify results.

```
Command> ALTER TABLE MyTab ADD CONSTRAINT PriKey PRIMARY KEY (Col3);
Command> ALTER TABLE MyTab ADD AGING USE Col4 LIFETIME 3 DAYS;

ttschema sampledb_1122
-- Database is in Oracle type mode
create table TESTUSER.MYTAB (
        COL1 NUMBER,
        COL2 VARCHAR2(30 BYTE) INLINE,
        COL3 NUMBER NOT NULL DEFAULT 10,
        COL4 TIMESTAMP(6) NOT NULL DEFAULT TIMESTAMP '2012-09-03 12:00:00')
    AGING USE COL4 LIFETIME 3 days CYCLE 5 minutes ON;

    alter table TESTUSER.MYTAB add constraint PRIKEY primary key (COL3);
```

Examples for ALTER TABLE: TimesTen Classic

Add returnrate column to parts table.

```
ALTER TABLE parts ADD COLUMN returnrate DOUBLE;
```

Add numsssign and prevdept columns to contractor table.

```
ALTER TABLE contractor
  ADD ( numassign INTEGER, prevdept CHAR(30) );
```

Remove addr1 and addr2 columns from employee table.

```
ALTER TABLE employee DROP ( addr1, addr2 );
```

Drop the UNIQUE title column of the books table.

```
ALTER TABLE books DROP UNIQUE (title);
```

Add the x1 column to the t1 table with a default value of 5:

```
ALTER TABLE t1 ADD (x1 INT DEFAULT 5);
```

Change the default value of column x1 to 2:

```
ALTER TABLE t1 MODIFY (x1 DEFAULT 2);
```

Alter table primarykeytest to add the primary key constraint c1. Use the ttIsql INDEXES command to show that the primary key constraint c1 is created and a range index is used:

```
Command> CREATE TABLE primarykeytest (col1 TT_INTEGER NOT NULL);
Command> ALTER TABLE primarykeytest ADD CONSTRAINT c1
       PRIMARY KEY (col1);
Command> INDEXES primarykeytest;

Indexes on table SAMPLEUSER.PRIMARYKEYTEST:
  C1: unique range index on columns:
    COL1
  1 index found.

1 index found on 1 table.
```

Alter table prikeyhash to add the primary key constraint c2 using a hash index. Use the ttIsql INDEXES command to show that the primary key constraint c2 is created and a hash index is used:

```
Command> CREATE TABLE prikeyhash (col1 NUMBER (3,2) NOT NULL);
Command> ALTER TABLE prikeyhash ADD CONSTRAINT c2
         PRIMARY KEY (col1) USE HASH INDEX PAGES = 20;
Command> INDEXES prikeyhash;

Indexes on table SAMPLEUSER.PRIKEYHASH:
  C2: unique hash index on columns:
    COL1
  1 index found.

1 table found.
```

Attempt to add a primary key constraint on a table already defined with a primary key. You see an error:

```
Command> CREATE TABLE oneprikey (col1 VARCHAR2 (30) NOT NULL,
       col2 TT_BIGINT NOT NULL, col3 CHAR (15) NOT NULL,
       PRIMARY KEY (col1,col2));
Command> ALTER TABLE oneprikey ADD CONSTRAINT c2
       PRIMARY KEY (col1,col2);
 2235: Table can have only one primary key
The command failed.
```

Attempt to add a primary key constraint on a column that is not defined as NOT NULL. You see an error:

```
Command> CREATE TABLE prikeynull (col1 CHAR (30));
Command> ALTER TABLE prikeynull ADD CONSTRAINT c3
       PRIMARY KEY (col1);
 2236: Nullable column cannot be part of a primary key
The command failed.
```

This example illustrates the use of range and hash indexes. It creates the pkey table with col1 as the primary key. A range index is created by default. The table is then altered to change the index on col1 to a hash index. The table is altered again to change the index back to a range index.

```
Command> CREATE TABLE pkey (col1 TT_INTEGER PRIMARY KEY, col2 VARCHAR2 (20));
Command> INDEXES pkey;
Indexes on table SAMPLEUSER.PKEY:
  PKEY: unique range index on columns:
   COL1
 1 index found.
1 index found on 1 table.
```

Alter the pkey table 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.
```

Alter the pkey table to use a range index with the USE RANGE INDEX clause:

```
Command> ALTER TABLE pkey USE RANGE INDEX;
Command> INDEXES pkey;
Indexes on table SAMPLEUSER.PKEY:
  PKEY: unique range index on columns:
    COL1
  1 index found.
1 table found.
```

This example generates an error when attempting to alter a table to define either a range or hash index on a column without a primary key.

```
Command> CREATE TABLE myindex (Ccl1 CHAR (20));
Command> ALTER TABLE myindex USE RANGE INDEX;
 2810: The table has no primary key so cannot change its index type
The command failed.
Command> ALTER TABLE myindex USE HASH INDEX PAGES = CURRENT;
 2810: The table has no primary key so cannot change its index type
The command failed.
```

These examples show how time resolution works with aging. In this example, lifetime is three days.

- If (SYSDATE - ColumnValue) <= 3, do not age out the row.
- If (SYSDATE - ColumnValue) > 3, then the row is a candidate for aging.
- If (SYSDATE - ColumnValue) = 3 days, 22 hours, then row is not aged out because lifetime was specified in days. The row would be aged out if lifetime had been specified as 72 hours.

This example alters a table by 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 *)
```

This example alters a table by adding time-based aging. The table has no previous aging policy. The agingcolumn column is used for aging. 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
```

This example illustrates that after 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 *)
```

This 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 to use an external scheduler to control the aging process. Set aging state to OFF and then call the ttAgingScheduleNow procedure 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 *)
```

Call ttAgingScheduleNow to invoke aging with an external scheduler:

```
Command> CALL ttAgingScheduleNow ('agingdemo4');
```

Attempt to alter a table adding the aging column and then use that column for time-based aging. An 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.
```

Attempt to alter the LIFETIME clause for a table defined with time-based aging. The aging column is defined with data type TT_DATE. An error is generated because the LIFETIME unit is not expressed in DAYS.

```
Command> CREATE TABLE aging1 (col1 TT_DATE NOT NULL) AGING USE
       col1 LIFETIME 2 DAYS;
Command> ALTER TABLE aging1 SET AGING LIFETIME 2 HOURS;
 2977: Only DAY lifetime unit is allowed with a TT_DATE column
The command failed.
```

Alter the employees table to add a new compressed column of state, which contains the full name of the state. Note that the employees table already has a compressed column group consisting of job_id and manager_id.

```
Command> ALTER TABLE employees
       ADD COLUMN state VARCHAR2(20)
       COMPRESS (state BY DICTIONARY);

Command> DESCRIBE employees;
Table MYSCHEMA.EMPLOYEES:
  Columns:
   *EMPLOYEE_ID                     NUMBER (6) NOT NULL
    FIRST_NAME                      VARCHAR2 (20) INLINE
    LAST_NAME                       VARCHAR2 (25) INLINE NOT NULL
    EMAIL                           VARCHAR2 (25) INLINE NOT NULL
    PHONE_NUMBER                    VARCHAR2 (20) INLINE
    HIRE_DATE                       DATE NOT NULL
    JOB_ID                          VARCHAR2 (10) INLINE NOT NULL
```

```
  SALARY                   NUMBER (8,2)
  COMMISSION_PCT           NUMBER (2,2)
  MANAGER_ID               NUMBER (6)
  DEPARTMENT_ID            NUMBER (4)
  STATE                    VARCHAR2 (20) INLINE
 COMPRESS ( ( JOB_ID, MANAGER_ID ) BY DICTIONARY,
       STATE BY DICTIONARY )

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

The following example drops the compressed column state from the employees table:

```
Command> ALTER TABLE employees
 DROP state;
Command> DESCRIBE employees;
Table MYSCHEMA.EMPLOYEES:
 Columns:
 *EMPLOYEE_ID              NUMBER (6) NOT NULL
  FIRST_NAME               VARCHAR2 (20) INLINE
  LAST_NAME                VARCHAR2 (25) INLINE NOT NULL
  EMAIL                    VARCHAR2 (25) INLINE NOT NULL
  PHONE_NUMBER             VARCHAR2 (20) INLINE
  HIRE_DATE                DATE NOT NULL
  JOB_ID                   VARCHAR2 (10) INLINE NOT NULL
  SALARY                   NUMBER (8,2)
  COMMISSION_PCT           NUMBER (2,2)
  MANAGER_ID               NUMBER (6)
  DEPARTMENT_ID            NUMBER (4)
 COMPRESS ( ( JOB_ID, MANAGER_ID ) BY DICTIONARY )

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

See Also

CREATE TABLE
DROP TABLE

ALTER USER

The ALTER USER statement enables you to change a user's password. It also enables you to change the profile for the user, to lock or unlock the user's account, and to expire the user's password. A user with the ADMIN privilege can perform these operations.

This statement also enables you to change a user from internal to external or from external to internal.

Required Privilege

No privilege is required to change the user's own password.

ADMIN privilege is required for all other operations.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

This is the syntax for ALTER USER...IDENTIFIED BY. Ensure to specify at least one of these clauses: IDENTIFIED BY, PROFILE, ACCOUNT, or PASSWORD EXPIRE.

```
ALTER USER user [IDENTIFIED BY {password | "password"}]
  [PROFILE profile] [ACCOUNT {LOCK|UNLOCK}] [PASSWORD EXPIRE];
```

This is the syntax for ALTER USER...IDENTIFIED EXTERNALLY. Ensure to specify at least one of these clauses: IDENTIFIED EXTERNALLY, PROFILE, or ACCOUNT.

```
ALTER USER user [IDENTIFIED EXTERNALLY]
  [PROFILE profile] [ACCOUNT {LOCK|UNLOCK}];
```

Parameters

Parameter	Description
user	Name of the user to alter.
IDENTIFIED BY password|"*password*"	Specifies an internal user and the password for the internal user. The password you can specify is dependent on the profile assigned to the user. Specifically, the value of the PASSWORD_COMPLEXITY_CHECKER password parameter determines the complexity of the password. If the value is TT_VERIFY_FUNCTION, TT_STRONG_VERIFY_FUNCTION, or TT_STIG_VERIFY_FUNCTION, the password must meet specific password verification requirements. For example, if the value is TT_VERIFY_FUNCTION, the password cannot contain the name of the database. See About Password Complexity Checker Verification for details.
IDENTIFIED EXTERNALLY	Specifies the user is an external user.
PROFILE *profile*	Use the PROFILE clause to specify the name of the profile (designated by *profile*) that you want to assign to the user. The profile sets the limits for the password parameters for the user. See CREATE PROFILE for information on these password parameters. You can specify a PROFILE clause for external users, but the password parameters have no effect for these users.
ACCOUNT [LOCK|UNLOCK]	Specify ACCOUNT LOCK to lock the user's account and disable connections to the database. Specify ACCOUNT UNLOCK to unlock the user's account and enable connections to the database. The default is ACCOUNT UNLOCK.
PASSWORD EXPIRE	Specify PASSWORD EXPIRE if you want the user's password to expire. This setting forces a user with ADMIN privileges to change the password before the user can connect to the database. This clause is not valid for an externally identified user (as denoted by the IDENTIFIED EXTERNALLY clause).

Description

- Database users can be internal or external.

  - Internal users are defined for a TimesTen database.
  - External users are defined by the operating system. External users cannot be assigned a TimesTen password.
- Password requirements:
  - Cannot exceed 30 characters.
  - Is case-sensitive.
  - Must start with a letter. A password cannot start with a digit or a special character unless the password is enclosed in double quotation marks.
  - If a special character is used, the password must be contained in double quotation marks. The exceptions are the # and the @ special characters. A password that contains the # or the @ special character does not need to be enclosed in double quotation marks.
  - Cannot contain a semi-colon (;) or a double quotation mark (").
- Use the PROFILE clause to change the profile for a user. See CREATE PROFILE for details.
- Use the ACCOUNT LOCK or ACCOUNT UNLOCK to change the lock settings for the user account.
- Use the PASSWORD EXPIRE clause to expire the user's password and force a password change before the user can connect to the database.
- You can alter a user over a client/sever connection if the connection is encrypted with TLS. See Transport Layer Security for TimesTen Client/Server in the *Oracle TimesTen In-Memory Database Security Guide* for details.
- When replication is configured, this statement is replicated.

Examples

Illustrate Password Verification When Altering User

This example creates the myprofile_strongpw profile and specifies a value of TT_STRONG_VERIFY_FUNCTION for the PASSWORD_COMPLEXITY_CHECKER password parameter. The example then creates the sampleuser_pwchange user and assigns the myprofile_strongpw profile to this user. The specified password meets the requirements of the TT_STRONG_VERIFY_FUNCTION function and the user is created. See TT_STRONG_VERIFY_FUNCTION for more information on the TT_STRONG_VERIFY_FUNCTION function.

```
Command> CREATE PROFILE myprofile_strongpw LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STRONG_VERIFY_FUNCTION;

Profile created.

Command> CREATE USER sampleuser_pwchange
         IDENTIFIED BY "5&AbbN*60" PROFILE myprofile_strongpw;

User created.
```

Now alter the myprofile_strongpw profile, changing the value of the PASSWORD_COMPLEXITY_CHECKER password parameter to TT_STIG_VERIFY_FUNCTION. Use the ALTER USER statement to expire the password for the sampleuser_pwchange user. Attempt to connect to the database as the sampleuser_pwchange user. The connection fails, as the password is expired.

```
Command> ALTER PROFILE myprofile_strongpw LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STIG_VERIFY_FUNCTION;

Profile altered.

Command> ALTER USER sampleuser_pwchange PASSWORD EXPIRE;

User altered.

Command> GRANT CONNECT TO sampleuser_pwchange;
Command> connect adding "UID=sampleuser_pwchange;PWD=5&AbbN*60" as sampleuser;
15180: the password has expired
The command failed.
```

Use the ALTER USER statement to change the password for the sampleuser_pwchange user. The ALTER USER statement succeeds, as the password meets the requirements of the TT_STIG_VERIFY_FUNCTION function. Attempt to connect to the database as the sampleuser_pwchange user. The connection is successful. See TT_STIG_VERIFY_FUNCTION for more information on the TT_STIG_VERIFY_FUNCTION function.

```
access1: Command> ALTER USER sampleuser_pwchange
         IDENTIFIED BY "bd@<!BCvvKASn67";

User altered.

Command> connect adding "UID=sampleuser_pwchange;PWD=bd@<!BCvvKASn67"
         as sampleuser;
Connection successful: DSN=access1;UID=sampleuser_pwchange;
DataStore=/scratch/sampleuser/mydatabase1;DatabaseCharacterSet=AL32UTF8;
ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Change the User's Profile

This example creates the user1 user and assigns the user1 user the profile1 profile. The example then uses the ALTER USER statement to change the user1 user's profile to profile2.

```
Command> CREATE USER user1 IDENTIFIED BY user1 PROFILE profile1;

User created.

Command> ALTER USER user1 PROFILE profile2;

User altered.
```

Query the dba_users system view to verify the user1 profile has been changed to profile2.

```
Command> SELECT profile FROM dba_users WHERE username = 'USER1';
< PROFILE2 >
1 row found.
```

Lock and Unlock a User's Account

This example creates the user2 user. It then uses the ALTER USER statement to lock and then unlock the user2 user's account.

```
Command> CREATE USER user2 IDENTIFIED BY user2 PROFILE profile1;

User created.

Command> ALTER USER user2 ACCOUNT LOCK;

User altered.
```

Grant the CONNECT privilege to user2;

```
Command> GRANT CONNECT TO user2;
```

Attempt to connect to the database as user2. The user2 account is locked so the connection fails.

```
Command> connect adding "UID=user2;PWD=user2" as user2;
15179: the account is locked
The command failed.
```

As the instance administrator, reconnect to the database and use the ALTER USER statement to unlock the user2 account.

```
none: Command> use database1
database1: Command> ALTER USER user2 ACCOUNT UNLOCK;

User altered.
```

Attempt to connect to the database as the user2 user. The connection succeeds.

```
database1: Command> connect adding "UID=user2;PWD=user2" as user2;
Connection successful: DSN=database1;UID=user3;DataStore=/scratch/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Expire a User's Password

This example uses the ALTER USER statement to change the user2 user's account to expire the password. A user with ADMIN privilege must change the user2 password before user2 can connect to the database.

```
Command> ALTER USER user2 PASSWORD EXPIRE;

User altered.
```

Attempt to connect to the database as user2. The user2 password must be changed before the user2 user can connect to the database.

```
Command>  connect adding "UID=user2;PWD=user2" as user2;
15180: the password has expired
The command failed.
```

As the instance administrator, reconnect to the database and use the ALTER USER statement to change the user2 password.

```
none: Command> use database1
database1: Command>  ALTER USER user2 IDENTIFIED BY newuser2password;

User altered.
```

Attempt to connect to the database a the `user2` user. The connection succeeds.

```
database1: Command> connect adding "UID=user2;PWD=newuser2password" as user2;
Connection successful: DSN=database1;UID=user4;DataStore=/scratch/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Change a User from External to Internal and Internal to External

This example uses the `ALTER USER` statement to change the `user2` internal user to an external user and then back to an internal user.

```
Command> ALTER USER user2 IDENTIFIED EXTERNALLY;

User altered.
```

Use the `ALTER USER` statement to change the `user2` external user back to an internal user.

```
Command> ALTER USER user2 IDENTIFIED BY user2_password_change;

User altered.
```

See Also

- CREATE PROFILE
- CREATE USER
- DROP USER
- GRANT
- REVOKE

CALL

Use the `CALL` statement to execute a TimesTen built-in procedure or to execute a PL/SQL procedure or function that is standalone or part of a package from within SQL.

Required Privilege

The privileges required for executing each TimesTen built-in procedure are listed in the description of each procedure in the Built-In Procedures section in the *Oracle TimesTen In-Memory Database Reference*.

No privileges are required for an owner calling its own PL/SQL procedure or function that is standalone or part of a package using the `CALL` statement. For all other users, the `EXECUTE` privilege on the procedure or function or on the package in which it is defined is required.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

To call a TimesTen built-in procedure:

```
CALL TimesTenBuiltIn [( arguments )]
```

When calling PL/SQL procedures or functions that are standalone or part of a package, you can either call these by name or as the result of an expression.

To call a PL/SQL procedure:

CALL [*Owner*.][*Package*.]*ProcedureName* [(*arguments*)]

To call a PL/SQL function that returns a parameter, one of the following are appropriate:

CALL [*Owner*.][*Package*.]*FunctionName* [(*arguments*)] INTO :*return_param*

> **Note**
>
> A user's own PL/SQL procedure or function takes precedence over a TimesTen built-in procedure with the same name.

Parameters

Parameter	Description
TimesTenBuiltIn	Name of the TimesTen built-in procedure. For a full list of TimesTen built-in procedures, see Built-In Procedures in the *Oracle TimesTen In-Memory Database Reference*.
[*Owner*.]*ProcedureName*	Name of the PL/SQL procedure. You can optionally specify the owner of the procedure.
[*Owner*.]*FunctionName*	Name of the PL/SQL function. You can optionally specify the owner of the function.
arguments	Specify 0 or more arguments for the PL/SQL procedure or function.
INTO	If the routine is a function, the INTO clause is required.
return_param	Specify the host variable that stores the return value of the function.

Description

Detailed information on how to execute PL/SQL procedures or functions with the CALL statement in TimesTen is provided in Executing Procedures and Functions in the *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*, Using CALL to Execute Procedures and Functions in the *Oracle TimesTen In-Memory Database C Developer's Guide*, or Using CALL to Execute Procedures and Functions in the *Oracle TimesTen In-Memory Database Java Developer's Guide*.

Examples

The following is the definition of the mytest function:

```
create or replace function mytest return number is
begin
  return 1;
end;
/
```

Perform the following to execute the mytest function in a CALL statement:

```
Command> variable n number;
Command> call mytest() into :n;
```

```
Command> print n;
N                    : 1
```

The following example creates a function that returns the salary of the employee whose employee ID is specified as input, then calls the function and displays the result that was returned.

```
Command> CREATE OR REPLACE FUNCTION get_sal
         (p_id employees.employee_id%TYPE) RETURN NUMBER IS
           v_sal employees.salary%TYPE := 0;
        BEGIN
          SELECT salary INTO v_sal FROM employees
            WHERE employee_id = p_id;
          RETURN v_sal;
        END get_sal;
        /

Function created.

Command> variable n number;
Command> call get_sal(100) into :n;
Command> print n;
N                    : 24000
```

COMMIT

The COMMIT statement ends the current transaction and makes permanent all changes performed in the transaction.

Required privilege

None

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
COMMIT [WORK]
```

Parameters

The COMMIT statement enables the following optional keyword:

Parameter	Description
[WORK]	Optional clause supported for compliance with the SQL standard. COMMIT and COMMIT WORK are equivalent.

Description

- Until you commit a transaction:
  - You can see any changes you have made during the transaction but other users cannot see the changes. After you commit the transaction, the changes are visible to other users' statements that execute after the commit.
  - You can roll back (undo) changes made during the transaction with the ROLLBACK statement.

- This statement releases transaction locks.
- For passthrough, the Oracle Database transaction will also be committed.
- A commit closes all open cursors.

Examples

Insert a row into regions table of the HR schema and commit transaction. First set autocommit to 0:

```
Command> SET AUTOCOMMIT 0;
Command> INSERT INTO regions VALUES (5,'Australia');
1 row inserted.
Command> COMMIT;
Command> SELECT * FROM regions;
< 1, Europe >
< 2, Americas >
< 3, Asia >
< 4, Middle East and Africa >
< 5, Australia >
5 rows found.
```

See also

ROLLBACK

CREATE ACTIVE STANDBY PAIR

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

This statement creates an active standby pair. It includes an active master database, a standby master database, and may also include one or more read-only subscribers. The active master database replicates updates to the standby master database, which propagates the updates to the subscribers.

Required Privilege

ADMIN

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL Syntax

```
CREATE ACTIVE STANDBY PAIR
 FullStoreName, FullStoreName [ReturnServiceAttribute]
   [SUBSCRIBER FullStoreName [,...]]
   [STORE FullStoreName [StoreAttribute [...]]]
   [NetworkOperation [...] ]
   [{ INCLUDE | EXCLUDE }{TABLE [[Owner.]TableName [,...]]|
      CACHE GROUP [[Owner.]CacheGroupName [,...]]|
      SEQUENCE [[Owner.]SequenceName [,...]]} [,...]]
```

Syntax for *ReturnServiceAttribute*:

```
{ RETURN RECEIPT [BY REQUEST] |
  RETURN TWOSAFE [BY REQUEST] |
  NO RETURN }
```

Syntax for *StoreAttribute*:

```
  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
  TABLE DEFINITION CHECKING {RELAXED|EXACT}
```

Syntax for *NetworkOperation*:

```
ROUTE MASTER FullStoreName SUBSCRIBER FullStoreName
  { { MASTERIP MasterHost | SUBSCRIBERIP SubscriberHost }
    PRIORITY Priority } [...]
```

Parameters

Parameter	Description
FullStoreName	The database, specified as one of the following: • SELF • The prefix of the database file name For example, if the database path is *directory/subdirectory/*data.ds0, then data is the database name that should be used. This is the database 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 the Network in *Oracle TimesTen In-Memory Database 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 an active master database are blocked until the transaction is received by the standby master database. Specifying 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 *Oracle TimesTen In-Memory Database Replication Guide*.

Parameter	Description
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to an active master database are blocked until the transaction is committed on the standby master database. Specifying 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 *Oracle TimesTen In-Memory Database Replication Guide*.
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*. Specifying SUBSCRIBER is the same as specifying ALL. Both settings refer to the standby master database. This failure policy can be specified for either the RETURN RECEIPT or RETURN TWOSAFE service.
RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED	Sets return services on or off when replication is disabled (stopped or paused state). OFF disables return services when replication is disabled and is the default for RETURN RECEIPT service. ON allows return services to continue to be enabled when replication is disabled and is the default for RETURN TWOSAFE service. See Establishing Return Service Failure and Recovery Policies in *Oracle TimesTen In-Memory Database Replication Guide*.
RESUME RETURN *Milliseconds*	If DISABLE RETURN has disabled return service blocking, this attribute sets the policy for when to re-enable the return service.
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 *Oracle TimesTen In-Memory Database Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgment. A value of 0 (zero) means that there is no waiting. The default value is 10 seconds. The application can override this timeout setting by using the returnWait parameter in the ttRepSyncSet built-in procedure.
SUBSCRIBER *FullStoreName* [,...]]	A database that receives updates from a master database. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.
STORE *FullStoreName* [*StoreAttribute* [...]]	Defines the attributes for the specified database. Attributes include PORT, TIMEOUT and FAILTHRESHOLD. *FullStoreName* is the database file name specified in the DataStore attribute of the DSN description.

Parameter	Description
TABLE DEFINITION CHECKING {EXACT|RELAXED}	StoreAttribute clause. Specifies type of table definition checking that occurs on the subscriber: • EXACT - The tables must be identical on master and subscriber. • RELAXED - The tables must have the same key definition, number of columns and column data types. The default is RELAXED. **Note:** If you use TABLE DEFINITION CHECKING EXACT, use ttMigrate -exactUpgrade if you migrate the database. If you use TABLE DEFINITION CHECKING RELAXED, use ttMigrate -relaxedUpgrade if you migrate the database.
{INCLUDE | EXCLUDE} {TABLE [[*Owner*.]*TableName*[,...]]| CACHE GROUP [[*Owner*.]*CacheGroupName* [,...]]| SEQUENCE [[*Owner*.]*SequenceName* [,...]]} [,...]	An active standby pair replicates an entire database by default. INCLUDE includes only the listed tables, sequences or cache groups for the replication scheme. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE removes tables or sequences or cache groups from the replication scheme. Use one EXCLUDE clause for each object type (table, sequence or cache group). Do not use the EXCLUDE clause for AWT cache groups.
COMPRESS TRAFFIC {ON | OFF}	Compress replicated traffic to reduce the amount of network bandwidth. ON specifies that all replicated traffic for the database defined by STORE be compressed. OFF (the default) specifies no compression. See Compressing Replicated Traffic in *Oracle TimesTen In-Memory Database Replication Guide* for details.
DURABLE COMMIT {ON | OFF}	Overrides the DurableCommits general connection attribute setting. DURABLE COMMIT ON enables durable commits regardless of whether the replication agent is running or stopped. It also enables durable commits when the ttRepStateSave built-in procedure has marked the standby database as failed.
FAILTHRESHOLD *Value*	The number of log files that can accumulate for a subscriber database. If this value is exceeded, the subscriber is set to the Failed state.The value 0 means "No Limit." This is the default. See Setting the Transaction Log Failure Threshold in *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Parameter	Description
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 valid only 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 only reissue the commit. The transaction may not be rolled back. This is the default. COMMIT: On timeout, the commit function attempts to perform a COMMIT to end the transaction locally. No more operations are possible on the same transaction. This setting can be overridden for specific transactions by calling the localAction parameter in the ttRepSyncSet procedure.
MASTER *FullStoreName*	The database on which applications update the specified element. The MASTER database sends updates to its SUBSCRIBER databases. The *FullStoreName* must be the database specified in the DataStore attribute of the DSN description.
PORT *PortNumber*	The TCP/IP port number on which the replication agent for the database listens for connections. If not specified, the replication agent automatically allocates a port number. In an active standby pair, the standby master database listens for updates from the active master database. Read-only subscribers listen for updates from the standby master database.
ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Denotes the *NetworkOperation* clause. If specified, enables you to control the network interface that a master store uses for every outbound connection to each of its subscriber stores. In the context of the ROUTE clause, you can define the following: • A route for the active database to the standby database and for the standby database to the active database for when failover occurs • A route for a read-only subscriber to the active and standby databases When using active standby pairs, ROUTE should be specified at least twice for an active standby pair with no read only subscribers. Then, ROUTE should be specified twice more for each read only subscriber on the active standby pair. For *FullStoreName*, ON "*host*" must be specified.
MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost*	*MasterHost* and *SubscriberHost* are the IP addresses for the network interface on the master and subscriber stores. Specify in dot notation or canonical format or in colon notation for IPV6. Clause can be specified more than once.

Parameter	Description
PRIORITY *Priority*	Variable expressed as an integer from 1 to 99. Denotes the priority of the IP address. Lower integral values have higher priority. An error is returned if multiple addresses with the same priority are specified. Controls the order in which multiple IP addresses are used to establish peer connections. Required syntax of *NetworkOperation* clause. Follows MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost* clause.
TIMEOUT *Seconds*	The maximum number of seconds the replication agent waits for a response from remote replication agents. The default is 120 seconds. In an active standby pair, the active master database sends messages to the standby master database. The standby master database sends messages to the read-only subscribers. **Note:** For large transactions that may cause a delayed response from the remote replication agent, the agent scales the timeout based on the size of the transaction. This scaling is disabled if you set TIMEOUT to less than or equal to 60 seconds. Also see Setting Wait Timeout for Response from Remote Replication Agents in *Oracle TimesTen In-Memory Database Replication Guide*.

Description

- After you create an active standby pair, make one of your databases the active database. To accomplish this, call ttRepStateSet ('ACTIVE'). Then use ttRepAdmin to duplicate the active database to the second database. When the operation is successful, the second database becomes the standby database. For more information, see Setting Up an Active Standby Pair with No Cache Groups in *Oracle TimesTen In-Memory Database Replication Guide*.
- The SUBSCRIBER clause lists one or more read-only subscriber databases. You can designate up to 127 subscriber databases.
- Replication between the active master database and the standby master database 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.
- If the active standby pair has the RETURN TWOSAFE attribute and replicates a cache group, a transaction may fail if:
  - The transaction that is being replicated contains an ALTER TABLE statement or an ALTER CACHE GROUP statement.
  - The transaction contains an INSERT, UPDATE or DELETE statement on a replicated table, replicated cache group or an asynchronous writethrough cache group.
- You can use an active standby pair to replicate read-only cache groups and asynchronous writethrough (AWT) cache groups. You cannot use an active standby pair to replicate synchronous writethrough (SWT) cache groups or user managed cache groups.
- You cannot use the EXCLUDE clause for AWT cache groups.
- You cannot execute the CREATE ACTIVE STANDBY PAIR statement when Oracle Clusterware is used with TimesTen.

Examples

This example creates an active standby pair whose master databases 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 databases.

```
CREATE ACTIVE STANDBY PAIR rep1, rep2 RETURN RECEIPT
  SUBSCRIBER rep3
  STORE rep1 PORT 21000 TIMEOUT 30
  STORE rep2 PORT 22000 TIMEOUT 30;
```

Specify *NetworkOperation* clause to control network interface:

```
CREATE ACTIVE STANDBY PAIR rep1,rep2
ROUTE MASTER rep1 ON "machine1" SUBSCRIBER rep2 ON "machine2"
MASTERIP "1.1.1.1" PRIORITY 1 SUBSCRIBERIP "2.2.2.2" PRIORITY 1
ROUTE MASTER rep2 ON "machine2" SUBSCRIBER rep1 ON "machine1"
MASTERIP "2.2.2.2" PRIORITY 1 SUBSCRIBERIP "1.1.1.1" PRIORITY 1;
```

See Also

ALTER ACTIVE STANDBY PAIR
DROP ACTIVE STANDBY PAIR

CREATE CACHE GROUP

The CREATE CACHE GROUP statement:

- Creates the table defined by the cache group.
- Loads all new information associated with the cache group in the appropriate system tables.

A *cache group* is a set of tables related through foreign keys that cache data from tables in an Oracle database. There is one root table that 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 database table.
- The rows are stored in a TimesTen table with the same name as the Oracle database table.

If a database has more than one cache group, the cache groups must correspond to different Oracle database (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.

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. System managed cache group types include:

- Read-only cache groups are updated in the Oracle database, and the updates are propagated from the Oracle database to the cache.

- Asynchronous writethrough (AWT) cache groups are updated in the cache and the updates are propagated to the Oracle database. Transactions continue executing on the cache without waiting for a commit on the Oracle database.
- Synchronous writethrough (SWT) cache groups are updated in the cache and the updates are propagated to the Oracle database. Transactions are committed on the cache after notification that a commit has occurred on the Oracle database.

Because TimesTen manages system managed cache groups, including loading and unloading the cache group, certain statements and clauses cannot be used in the definition of these cache groups, including:

- WHERE clauses in AWT and SWT cache group definitions
- READONLY, PROPAGATE and NOT PROPAGATE in cache table definitions
- AUTOREFRESH in AWT and SWT cache group definitions

The FLUSH CACHE GROUP and REFRESH CACHE GROUP operations are not allowed for AWT and SWT cache groups.

You must stop the replication agent before creating an AWT cache group.

A *user managed cache group* must be managed by the application or user. PROPAGATE in a user managed cache group is synchronous. The table-level READONLY keyword can only be used for user managed cache groups.

In addition, both TimesTen and Oracle Database must be able to parse all WHERE clauses.

Cache groups can be explicitly or dynamically loaded.

In cache groups that are explicitly loaded, new cache instances are loaded manually into the TimesTen cache tables from the Oracle database tables using a LOAD CACHE GROUP or REFRESH CACHE GROUP statement or automatically using an autorefresh operation.

In a dynamic cache group, new cache instances can be loaded manually into the TimesTen cache tables by using a LOAD CACHE GROUP or on demand using a dynamic load operation. In a dynamic load operation, data is automatically loaded into the TimesTen cache tables from the cached Oracle database tables when a SELECT, UPDATE, DELETE or INSERT statement is issued on one of the cache tables, where the data is not present in the cache table but does exist in the cached Oracle database table. A manual refresh or automatic refresh operation on a dynamic cache group can result in the updating or deleting of existing cache instances, but not in the loading of new cache instances.

Any cache group type (read-only, asynchronous writethrough, synchronous writethrough, user managed) can be defined as an explicitly loaded cache group.

Any cache group type can be defined as a dynamic cache group *except* a user managed cache group that has both the AUTOREFRESH cache group attribute and the PROPAGATE cache table attribute.

Data in a dynamic cache group is aged out because LRU aging is defined by default. Use the ttAgingLRUConfig and/or the ttAgingTableLRUConfig built-in procedures to override the space usage thresholds for LRU aging. You can also define time-based aging on a dynamic cache group to override LRU aging.

For more information on static and dynamic cache groups, see Cache Groups and Cache Tables in *Oracle TimesTen In-Memory Database Cache Guide*.

Required Privilege

CREATE CACHE GROUP or CREATE ANY CACHE GROUP *and* CREATE TABLE (if all tables in the cache group are owned by the current user) *or* CREATE ANY TABLE (if at least one of the tables in the cache group is not owned by the current user).

Usage with TimesTen Scaleout

Static read-only cache groups with incremental autorefresh are supported.

SQL Syntax: TimesTen Scaleout

For static read-only cache groups in TimesTen Scaleout:

```
CREATE READONLY CACHE GROUP [Owner.]CacheGroupName
 [AUTOREFRESH
   [MODE INCREMENTAL]
   [INTERVAL IntervalValue {MINUTE[S] | SECOND[S] | MILLISECOND[S]}]
   [STATE {ON|OFF|PAUSED}]
  ]
  FROM
   [Owner.]TableName (ColumnDefinition[,...][,PRIMARY KEY(ColumnName[,...])])
  [UNIQUE HASH ON (HashColumnName[,...]) PAGES=PrimaryPages]
  [ParentDistributionClause]
   [WHERE ExternalSearchCondition]
  [,[Owner.]TableName (ColumnDefinition[,...]
   [,PRIMARY KEY(ColumnName[,...])]
   [,FOREIGN KEY(ColumnName[,...])
       REFERENCES RefTableName (ColumnName [,...])[ON DELETE CASCADE]])
 [UNIQUE HASH ON (HashColumnName[,...]) PAGES=PrimaryPages]
 [ChildDistributionClause]
  [WHERE ExternalSearchCondition]
 [,...]
 ]
```

The syntax for the distribution clause for a parent:

```
ParentDistributionClause::= DISTRIBUTE BY HASH [(ColumnName [,...])] | DUPLICATE
```

The syntax for the distribution clause for a child:

```
ChildDistributionClause::= DISTRIBUTE BY HASH [(ColumnName [,...])] |
 DISTRIBUTE BY REFERENCE [(ForeignKeyConstraint)] | DUPLICATE
```

SQL Syntax: TimesTen Classic

For read-only cache groups:

```
CREATE [DYNAMIC] [HYBRID] READONLY CACHE GROUP [Owner.]CacheGroupName
 [AUTOREFRESH
  [MODE {INCREMENTAL | FULL}]
  [INTERVAL IntervalValue {MINUTE[S] | SECOND[S] | MILLISECOND[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]
 [AGING {LRU|
      USE ColumnName
      LIFETIME Num1 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
      [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
      }[ON|OFF]
 ]
 [WHERE ExternalSearchCondition]
} [,...]
```

For asynchronous writethrough cache groups:

```
CREATE [DYNAMIC] [ASYNCHRONOUS] WRITETHROUGH CACHE GROUP   [Owner.]CacheGroupName
 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 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
      [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
    }[ON|OFF]
 ]
} [,...]
```

For synchronous writethrough cache groups:

```
CREATE [DYNAMIC] SYNCHRONOUS WRITETHROUGH
CACHE GROUP [Owner.]CacheGroupName
 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 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
      [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
    }[ON|OFF]
 ]
} [,...]
```

For user managed cache groups:

```
CREATE [DYNAMIC][USERMANAGED] CACHE GROUP [Owner.]CacheGroupName
 [AUTOREFRESH
  [MODE {INCREMENTAL | FULL}]
  [INTERVAL IntervalValue {MINUTE[S] | SECOND[S] | MILLISECOND[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]
[AGING {LRU|
     USE ColumnName
        LIFETIME Num1 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
        [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
     }[ON|OFF]
]
[WHERE ExternalSearchCondition]
} [,...]
```

Parameters

Following are the parameters for the cache group definition before the FROM keyword:

Parameter	Description
[*Owner*.]*CacheGroupName*	Owner and name assigned to the new cache group.
DYNAMIC	Supported in TimesTen Classic only. If specified, a dynamic cache group is created.
HYBRID	Supported in TimesTen Classic only. If specified, a dynamic read-only cache group where the root table does not exist in the Oracle database.
AUTOREFRESH	The AUTOREFRESH parameter automatically propagates changes from the Oracle database to the cache group.
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 the Oracle database 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. In TimesTen Scaleout, MODE INCREMENTAL is supported.
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. An autorefresh interval set to 0 milliseconds enables continuous autorefresh, where the next autorefresh cycle is scheduled immediately after the last autorefresh cycle has ended. See Automatically Refreshing a Cache Group in the *Oracle TimesTen In-Memory Database Cache Guide* for more information. 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 support log if the wait queue reaches 10.
STATE [ON | OFF | PAUSED]	Specifies whether autorefresh should be ON or OFF or PAUSED when the cache group is created. You can alter this setting later by using the ALTER CACHE GROUP statement. 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 database 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 following parameters.

Parameter	Description
[*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 it is nullable. Each table must have at least one column.
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.
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	Supported in TimesTen Classic only. Specifies whether changes to the cached table are automatically propagate to the corresponding Oracle database 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*	Sizes the hash index to reflect the expected number of pages in your table. To determine the value for *PrimaryPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for *PrimaryPages* (256000/256=1000). The value for *PrimaryPages* must be a positive constant and must be greater than 0. If your estimate for *PrimaryPages* is too small, performance may be degraded. For more information on hash indexes, see CREATE TABLE.
WHERE *ExternalSearchCondition*	The WHERE clause evaluated by the Oracle database 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 Database. See Using a WHERE Clause in *Oracle TimesTen In-Memory Database Cache Guide*.

Parameter	Description
ParentDistributionClause	In TimesTen Scaleout, distribution clause for a parent table in a static read-only cache group with incremental autorefresh. These distribution schemes are supported for parent tables: • DISTRIBUTE BY HASH [(*ColumnName* [,...])] • DUPLICATE
ChildDistributionClause	In TimesTen Scaleout, distribution clause for a child table in a static read-only cache group with incremental autorefresh. These distribution schemes are supported for child tables: • DISTRIBUTE BY HASH [(*ColumnName* [,...])] • DISTRIBUTE BY REFERENCE [(*ForeignKeyConstraint*)] • DUPLICATE
AGING LRU [ON | OFF]	Supported in TimesTen Classic only. If specified, defines the LRU aging policy on the root table. The LRU aging policy applies to all tables in the cache group. The LRU aging policy defines the type of aging (least recently used (LRU)), the aging state (ON or OFF) and the LRU aging attributes. Set the aging state to either ON or OFF. ON indicates that the aging state is enabled and aging is done automatically. OFF indicates that the aging state is disabled and aging is not done automatically. In both cases, the aging policy is defined. The default is ON. In dynamic cache groups, LRU aging is ON by default. However, you can specify time-based aging or set LRU aging to OFF at the syntax level. LRU aging cannot be specified on a cache group with the autorefresh attribute, unless the cache group is dynamic. LRU attributes are defined by calling the ttAgingLRUConfig and/or the ttAgingTableLRUConfig built-in procedures. LRU attributes are not defined at the SQL level. See ttAgingLRUConfig and ttAgingTableLRUConfig in the *Oracle TimesTen In-Memory Database Reference* and Implementing an Aging Policy in Your Tables in the *Oracle TimesTen In-Memory Database Operations Guide* for more information.

Parameter	Description
AGING USE *ColumnName*...[ON|OFF]	Supported in TimesTen Classic only. If specified, defines the time-based aging policy on the root table. The time-based aging policy applies to all tables in the cache group. The time-based aging policy defines the type of aging (time-based), the aging state (ON or OFF) and the time-based aging attributes. Set the aging state to either ON or OFF. ON indicates that the aging state is enabled and aging is done automatically. OFF indicates that the aging state is disabled and aging is not done automatically. In both cases, the aging policy is defined. The default is ON. Time-based aging attributes are defined at the SQL level and are specified by the LIFETIME and CYCLE clauses. Specify *ColumnName* as the name of the column used for time-based aging. Define the column as NOT NULL and of data type TIMESTAMP or DATE. The value of this column is subtracted from SYSDATE, truncated using the specified unit (second, minute, hour, day) and then compared to the LIFETIME value. If the result is greater than the LIFETIME value, then the row is a candidate for aging. The values of the column used for aging 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 (the column cannot be NULL). For more information about time-based aging, see Implementing Aging in a Cache Group for TimesTen Classic in *Oracle TimesTen In-Memory Database Cache Guide*.
LIFETIME *Num1* {SECOND[S]|MINUTE[S]|HOUR[S]DAY[S]}	Supported in TimesTen Classic only. LIFETIME is a time-based aging attribute and is a required clause. Specify the LIFETIME clause after the AGING USE *ColumnName* clause. The LIFETIME clause specifies the minimum amount of time data is kept in cache. Specify *Num1* as a positive integer constant to indicate the unit of time expressed in seconds, minutes, hours or days that rows should be kept in cache. Rows that exceed the LIFETIME value are aged out (deleted from the table). The concept of time resolution is supported. If 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. A LIFETIME of 3 days is different than a LIFETIME of 72 hours (3*24) or a LIFETIME of 432 minutes (3*24*60).

Parameter	Description
[CYCLE *Num2* {SECOND[S] | MINUTE[S] | HOUR[S]|DAY[S]}]	Supported in TimesTen Classic only. CYCLE is a time-based aging attribute and is optional. Specify the CYCLE clause after the LIFETIME clause. The CYCLE clause indicates how often the system should examine rows to see if data exceeds the specified LIFETIME value and should be aged out (deleted). Specify *Num2* as a positive integer constant. If you do not specify the CYCLE clause, then the default value is 5 minutes. If you specify 0 for *Num2,* then the aging thread wakes up every second. If the aging state is OFF, then aging is not done automatically and the CYCLE clause is ignored.

Cache Groups in TimesTen Scaleout

TimesTen Scaleout supports static read-only cache groups with incremental autorefresh. You can specify a distribution scheme on a parent table and on one or more child tables. The distribution scheme specifies how data is distributed across the elements of the database.

A distribution scheme is denoted by the DISTRIBUTE BY clause:

- For a single table cache group, the default distribution scheme is HASH.
- If you do not specify a column in the DISTRIBUTE BY clause, the primary key columns are used as the key columns for the distribution scheme.
- For a multiple table cache group, you can specify either the HASH or the DUPLICATE distribution scheme on the parent table. If you define the DUPLICATE distribution scheme, you can only specify HASH or DUPLICATE on the child tables.
- For a multiple table cache group, HASH is the default distribution scheme for the parent table and all child tables default to the REFERENCE distribution scheme. If you specify DUPLICATE on the parent table, and do not specify a distribution scheme for the child tables, the default distribution scheme for the child tables is DUPLICATE.
- If the foreign key on a child table is identical to the primary key on the parent table, the HASH distribution scheme is used for the child table as an optimization.
- It is best practice to distribute child tables by reference.

The following are not supported:

- Full autorefresh mode
- Aging
- Materialized views
- Global indexes

Cache Groups in TimesTen Classic

Dynamic Hybrid Read-Only Cache Groups

A dynamic hybrid read-only cache group is a dynamic read-only cache group where the root table does not exist in the Oracle database. The root table is automatically created in the TimesTen database from the cache group definition. The cache group definition includes the description of this root table, as if it existed in the Oracle database.

Description and restrictions:

- The root table must not exist in the Oracle database.
- The root table in the TimesTen database must have a primary key.
- The root table can only contain columns in the primary key. The primary key must be referenced by at least one child table.
- For dynamic load triggering, you can use a derived table in the FROM clause of the SELECT statement. You can also specify more than one table of the same hybrid cache group in the SELECT query.
- If you issue a SELECT query on the root table in the TimesTen database, this SELECT operation does not trigger a dynamic load.
- You cannot specify time-based aging on this type of cache group. LRU aging is enabled by default.
- The WHERE clause in the cache group definition is not supported.
- You cannot use the LOAD CACHE GROUP statement to manually load the cache group.
- The UNLOAD CACHE GROUP ...WITH ID statement is not supported.

See Hybrid Cache Group in the *Oracle TimesTen In-Memory Database Cache Guide* for more information on dynamic hybrid read-only cache groups.

Description of Cache Groups

- Two cache groups cannot have the same owner name and group name. If you do not specify the owner name, your schema becomes the owner name for the new cache group.
- Neither a cache table name nor a cache group name can contain #.
- Dynamic parameters are not allowed in the WHERE clause.
- Oracle Database temporary tables cannot be cached.
- Each table must correspond to a table in the Oracle database.
- In the Oracle database, you can define a parent/child relationship and then insert a null value into the foreign key column of the child table. This means this row in the child table references a null parent. You can then create a cache group and cache the parent/child relationship of the Oracle database tables. However, if you load data from the Oracle database tables into the cache group, the row that contains the null value of the foreign key column is not loaded. TimesTen recommends that you do not create cache groups if the tables you cache define a parent/child relationship in which the foreign key represents a null parent.
- You cannot use lowercase delimited identifiers to name your cache tables. Table names in TimesTen are case-insensitive and are stored as uppercase. The name of the cache table must be the same as the Oracle database table name. Uppercase table names on TimesTen will not match mixed case table names on the Oracle database. As a workaround, create a synonym for your table in the Oracle database and use that synonym as the table name for the cache group. This workaround is not available for read-only cache groups or cache groups with the AUTOREFRESH parameter set.
- Each column in the cache table must match each column in the Oracle database table, both in name and in data type. See Mappings Between Oracle Database and TimesTen Data Types in *Oracle TimesTen In-Memory Database Cache 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. Tables that are not in the cache group can only be referenced with a subquery.
- Generally, you do not have to fully qualify the column names in the WHERE clause of the CREATE CACHE GROUP, LOAD CACHE GROUP, UNLOAD CACHE GROUP, REFRESH CACHE GROUP or FLUSH CACHE GROUP statements. However, since TimesTen automatically generates queries that join multiple tables in the same cache group, a column must be fully qualified if there is more than one table in the cache group that contains columns with the same name.
- By default, a range index is created to enforce the primary key for a cache group table. Use the UNIQUE HASH clause if you want 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 range 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 for more information on the UNIQUE HASH clause.
  - Use ALTER TABLE to change the representation of the primary key index for a table.
- For cache group tables with the PROPAGATE attribute and for tables of SWT and AWT cache groups, foreign keys specified with ON DELETE CASCADE must be a proper subset of foreign keys with ON DELETE CASCADE in the Oracle database tables.
- You cannot execute the CREATE CACHE GROUP statement when performed under the serializable isolation level. An error message is returned when attempted.

The AUTOREFRESH parameter automatically propagates changes from the Oracle database to TimesTen cache groups. For static cache groups, deletes, updates and inserts are automatically propagated from the Oracle database to the cache group. For dynamic cache groups, only deletes and updates are propagated. Inserts to the specified Oracle database tables are not propagated to dynamic cache groups. They are dynamically loaded into TimesTen Cache when referenced by the application. They can also be explicitly loaded by the application.

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 statement. Once a cache group has been specified as either AUTOREFRESH or PROPAGATE, you cannot change these attributes. If you are creating a read-only cache group, you do not need to specify the autorefresh clause. A read-only cache group defaults to incremental autorefresh.

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 changes and periodically updates only the rows that have changed in the specified Oracle database tables. The first incremental refresh is always a full refresh, unless the autorefresh state is PAUSED. The default mode is INCREMENTAL.

FULL AUTOREFRESH is more efficient when most of the Oracle database table rows have been changed. INCREMENTAL AUTOREFRESH is more efficient when there are fewer changes.

TimesTen schedules an autorefresh operation when the transaction that contains a statement with AUTOREFRESH specified 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 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.

The NOT PROPAGATE attribute cannot be used with the AUTOREFRESH attribute.

Aging in cache groups:

- You can implement sliding windows with time-based aging. See Configuring a Sliding Window in TimesTen Classic in *Oracle TimesTen In-Memory Database Cache Guide*.
- After you have defined an aging policy for the table, you cannot change the policy from LRU to time-based or from time-based to LRU. You must first drop aging and then alter the table to add a new aging policy.
- The aging policy must be defined to change the aging state.
- 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 database. 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 the database. If you use both LRU and time-based aging, then the thread wakes up based on a combined consideration of both types.
- Call the ttAgingScheduleNow procedure to schedule the aging process right away regardless if the aging state is ON or OFF.
- The following rules determine if a row is accessed or referenced for LRU aging:
  - 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.
- 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.
- use
  For LRU aging, if a child row is not a candidate for aging, then neither this child row nor its parent row are 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.
- Specify either the LRU aging or time-based aging policy on the root table. The policy applies to all tables in the cache group.
- For the time-based aging policy, you cannot add or modify the aging column. This is because you cannot add or modify a NOT NULL column.
- Restrictions on defining aging for a cache group:
  - LRU aging is not supported on a cache group defined with the autorefresh attribute, unless it is a dynamic cache group.
  - The aging policy cannot be added, altered, or dropped for read-only cache groups or cache groups with the AUTOREFRESH attribute while the cache agent is active. Stop the cache agent first.

– You cannot drop the column that is used for time-based aging.

Examples: TimesTen Classic

These examples are specific to TimesTen Classic. For information and examples on using cache groups in TimesTen Scaleout, see Using Cache Groups in TimesTen Scaleout in the *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

Create a read-only 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));
```

Create an asynchronous writethrough cache group:

```
CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP cstomers
FROM
customer (custid INT NOT NULL,
    name CHAR(100) NOT NULL,
    addr CHAR(100),
    zip INT,
    PRIMARY KEY(custid));
```

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));
```

Create a user managed 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);
```

Create a cache group with time-based aging. Specify `agetimestamp` as the column for aging. Specify `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 *)
```

Use a synonym for a mixed case delimited identifier table name in the Oracle database so the mixed case table name can be cached in TimesTen. First attempt to cache the mixed case Oracle database table name. You see the error "Could not find '*NameofTable*' in Oracle":

```
Command> AUTOCOMMIT 0;
Command> PASSTHROUGH 3;
Command> CREATE TABLE "MixedCase" (col1 NUMBER PRIMARY KEY NOT NULL);
Command> INSERT INTO "MixedCase" VALUES (1);
1 row inserted.
Command> COMMIT;
Command> CREATE CACHE GROUP MixedCase1 from "MixedCase"
       (col1 NUMBER PRIMARY KEY NOT NULL);
 5140: Could not find SAMPLEUSER.MIXEDCASE in Oracle.  May not have privileges.
The command failed.
```

Now, using the PassThrough attribute, create the synonym "MIXEDCASE" in the Oracle database and use that synonym as the table name.

```
Command> AUTOCOMMIT 0;
Command> PASSTHROUGH 3;
Command> CREATE SYNONYM "MIXEDCASE" FOR "MixedCase";
Command> COMMIT;
Command> CREATE CACHE GROUP MixedCase2 FROM "MIXEDCASE"
        (col1 NUMBER PRIMARY KEY NOT NULL);
Warning  5147: Cache group contains synonyms
Command> COMMIT;
```

Attempt to use a synonym name with a read-only cache group or a cache group with the AUTOREFRESH attribute. You see an error:

```
Command> AUTOCOMMIT 0;
Command> PASSTHROUGH 3;
Command> CREATE SYNONYM "MIXEDCASE_AUTO" FOR "MixedCase";
Command> COMMIT;
Command> CREATE READONLY CACHE GROUP MixedCase3 AUTOREFRESH MODE
        INCREMENTAL INTERVAL 10 MINUTES FROM "MIXEDCASE_AUTO"
        (Col1 NUMBER PRIMARY KEY NOT NULL);
 5142: Autorefresh is not allowed on cache groups with Oracle synonyms
The command failed.
```

See Also

ALTER CACHE GROUP
ALTER TABLE
DROP CACHE GROUP
FLUSH CACHE GROUP
LOAD CACHE GROUP
UNLOAD CACHE GROUP

CREATE FUNCTION

The CREATE FUNCTION statement creates a standalone stored function.

Required Privilege

CREATE PROCEDURE (if owner) or CREATE ANY PROCEDURE (if not owner).

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE [OR REPLACE] FUNCTION [Owner.]FunctionName
    [(arguments [IN|OUT|IN OUT][NOCOPY] DataType [DEFAULT expr][,...])]
    RETURN DataType
    [InvokerRightsClause][AccessibleByClause][DETERMINISTIC]
    {IS|AS} PlsqlFunctionBody

InvokerRightsClause::=
AUTHID {CURRENT_USER|DEFINER}

AccessibleByClause::=
ACCESSIBLE BY (accessor[,...])

accessor::=
[UnitKind][Owner.]UnitName
```

You can specify *InvokerRightsClause*, *AccessibleByClause*, or DETERMINISTIC in any order.

Parameters

Parameter	Description
OR REPLACE	Specify OR REPLACE to recreate the function if it already exists. Use this clause to change the definition of an existing function without dropping and recreating it. When you recreate a function, TimesTen recompiles it.
FunctionName	Name of function.
arguments	Name of argument or parameter. You can specify 0 or more parameters for the function. If you specify a parameter, you must specify a data type for the parameter. The data type must be a PL/SQL data type.

Parameter	Description
IN|OUT|IN OUT	Parameter modes. IN is a read-only parameter. You can pass the parameter's value into the function but the function cannot pass the parameter's value out of the function and back to the calling PL/SQL block. The value of the parameter cannot be changed. OUT is a write-only parameter. Use an OUT parameter to pass a value back from the function to the calling PL/SQL block. You can assign a value to the parameter. IN OUT is a read/write parameter. You can pass values into the function and return a value back to the calling program (either the original, unchanged value or a new value set within the function. IN is the default.
NOCOPY	Specify NOCOPY to instruct TimesTen to pass the parameter as fast as possible. You can enhance performance when passing a large value such as a record, an index-by-table, or a varray to an OUT or IN OUT parameter. IN parameters are always passed NOCOPY.
DEFAULT *expr*	Use this clause to specify a default value for the parameter. You can specify := in place of the keyword DEFAULT.
RETURN *DataType*	Required clause. A function must return a value. You must specify the data type of the return value of the function. Do not specify a length, precision, or scale for the data type. The data type is a PL/SQL data type.
InvokerRightsClause	Lets you specify whether the SQL statements in PL/SQL functions or procedures execute with definer's or invoker's rights. The AUTHID setting affects the name resolution and privilege checking of SQL statements that a PL/SQL procedure or function issues at runtime, as follows: • Specify DEFINER so that SQL name resolution and privilege checking operate as though the owner of the procedure or function (the definer, in whose schema it resides) is running it. DEFINER is the default. • Specify CURRENT_USER so that SQL name resolution and privilege checking operate as though the current user (the invoker) is running it. For more information, see Definer's Rights and Invoker's Rights (AUTHID Clause) in the *Oracle TimesTen In-Memory Database Security Guide*.
AccessibleByClause	Use this clause to specify one or more *accessors* (PL/SQL units) that can invoke the function directly. The list of accessors that can access the function is called a *white list*. A white list gives you the ability to add an extra layer of security to your PL/SQL objects. Specifically, you can restrict access to the function to only those objects on the white list. *AccessibleByClause* can appear only once in the CREATE FUNCTION statement. Syntax: ACCESSIBLE BY (*accessor* [,...])

Parameter	Description
accessor	Used in *AccessibleByClause*. An accessor is a PL/SQL unit that can invoke the function. An accessor can appear more than once in the *AccessibleByClause* clause. Syntax: [*UnitKind*][*Owner*.]*UnitName*
UnitKind	Used in the *accessor* clause (which is part of the *AccessibleByClause* clause). Specifies the kind of PL/SQL unit that can invoke the function. *UnitKind* is optional, but if specified, valid options are: • FUNCTION • PROCEDURE • PACKAGE
[*Owner*.]*UnitName*	Used in the *accessor* clause (which is part of the *AccessibleByClause* clause). Specifies the name of the PL/SQL unit that can invoke the function. If you specify *UnitKind*, then *UnitName* must be a name of a unit of that kind. For example, if you specify PROCEDURE for *UnitKind*, then *UnitName* must be the name of a procedure. *UnitName* is required. You can optionally specify *Owner*. If you specify *Owner*, then *UnitName* must reside in that owner's schema. If you do not specify *Owner*, *UnitName* must be in the schema that contains the function.
DETERMINISTIC	Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its parameters.
IS|AS	Specify either IS or AS to declare the body of the function.
plsql_function_spec	Specifies the function body.

Description

- *AccessibleByClause*:
  - The compiler checks the validity of the syntax of the ACCESSIBLE BY clause, but does not check that the accessor exists. Therefore, you can define an accessor that does yet exist in the owner's schema.
  - When you invoke the function, the compiler first does the normal permission checks on the invocation. If any check fails, the invocation fails, even if the invoker is an accessor. If all normal permission checks on the invocation succeed, and the function has no ACCESSIBLE BY clause, the invocation succeeds. If the function has an ACCESSIBLE BY clause, the invocation succeeds only if the invoker is an accessor.
- When you create or replace a function, the privileges granted on the function remain the same. If you drop and recreate the object, the object privileges that were granted on the original object are revoked.
- In a replication environment, the CREATE FUNCTION statement is not replicated. For more information, see Creating a New PL/SQL Object in an Existing Active Standby Pair and Adding a PL/SQL Object to an Existing Classic Replication Scheme in the *Oracle TimesTen In-Memory Database Replication Guide*.
- TimesTen does not support:

- parallel_enable_clause

  You can specify this clause, but it has no effect.
- call_spec clause
- AS EXTERNAL clause

Examples

Using the Accessible By Clause

This example creates the ProtectedFunction function. The ACCESSIBLE BY clause is used to restrict the invocation of the function to the CallingProc1 and CallingProc2 procedures. Note that for CallingProc1, the type of PL/SQL unit is not specified and for CallingProc2, the type of PL/SQL unit is specified (PROCEDURE).

```
Command> CREATE OR REPLACE FUNCTION ProtectedFunction (a IN NUMBER)
           RETURN NUMBER
         ACCESSIBLE BY (CallingProc1, PROCEDURE CallingProc2)
         AS
         BEGIN
           RETURN a * 1;
         END;
         /

Function created.
```

Create the CallingProc1 and CallingProc2 procedures.

```
Command> CREATE OR REPLACE PROCEDURE CallingProc1 AS
           a NUMBER:=1;
         BEGIN
           a:=ProtectedFunction(a);
           DBMS_OUTPUT.PUT_LINE ('Calling Procedure: '|| a);
         END;
         /

Procedure created.

Command> CREATE OR REPLACE PROCEDURE CallingProc2
         AS
           a NUMBER:=2;
         BEGIN
           a:=ProtectedFunction(a);
           DBMS_OUTPUT.PUT_LINE ('Calling Procedure: '|| a);
         END;
         /

Procedure created.
```

Call the procedures. CallingProc1 and CallingProc2 are in the white list, resulting in successful execution.

```
Command> SET SERVEROUTPUT ON
Command> exec CallingProc1;
Calling Procedure: 1

PL/SQL procedure successfully completed.

Command> exec CallingProc2;
Calling Procedure: 2
```

```
PL/SQL procedure successfully completed.
```

Illustrating the Syntax for Creating a PL/SQL Function

Create function get_sal with one input parameter. Return salary as type NUMBER.

```
Command> CREATE OR REPLACE FUNCTION get_sal
       (p_id employees.employee_id%TYPE) RETURN NUMBER IS
       v_sal employees.salary%TYPE := 0;
      BEGIN
       SELECT salary INTO v_sal FROM employees
         WHERE employee_id = p_id;
       RETURN v_sal;
      END get_sal;
      /

Function created.
```

CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table or a materialized view.

Required Privilege

For a global index (supported in TimesTen Scaleout) and for a local index (supported in TimesTen Scaleout and in TimesTen Classic:

- If the owner, no privilege is required.
- If not the owner, the CREATE ANY INDEX system privilege or the INDEX object privilege is required.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. You can create a global or a local index.

SQL Syntax

The syntax to create a range index follows. Use the GLOBAL keyword, the optional INCLUDE clause, and the optional *IndexDistributionClause* clause to create a global index. A global index is supported only in TimesTen Scaleout:

```
CREATE [GLOBAL][UNIQUE] INDEX [Owner.]IndexName ON
[Owner.]TableName (ColumnName [ASC | DESC][,... ])
[INCLUDE (ColumnName[,…])]
[IndexDistributionClause]
```

The syntax to create a hash index follows. Use the GLOBAL keyword to create a global index. The optional INCLUDE clause, and the optional *IndexDistributionClause* clause can only be used with a global index. A global index is supported only in TimesTen Scaleout:

```
CREATE [GLOBAL][UNIQUE] HASH INDEX [Owner.]IndexName ON
[Owner.]TableName (ColumnName [ASC | DESC][,... ] )
[INCLUDE (ColumnName [,…])]
[ PAGES = RowPages | CURRENT ]
[IndexDistributionClause]
```

The syntax for *IndexDistributionClause* can only be used for a global index (supported in TimesTen Scaleout):

IndexDistributionClause::=
DISTRIBUTE BY HASH [(*ColumnName* [,…])]

Parameters

Parameter	Description
GLOBAL	The GLOBAL keyword is only supported in TimesTen Scaleout. In TimesTen Scaleout: • Specify GLOBAL to create a global index. A global index maps all rows in the table to a hash distribution scheme. When you create a global index, TimesTen Scaleout creates a materialized view with a local index and a hash distribution scheme to the index key columns. • If you do not specify GLOBAL, a local index is created in each database element. The index in this database element maps to rows in the table in the database element.
UNIQUE	You can specify UNIQUE for both range and hash indexes. If you specify UNIQUE each possible combination of index key column values can occur in only one row of the table.
HASH	Specify HASH to create a hash index. Specify UNIQUE with HASH to create a unique hash index.
[*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.
ON [*Owner*.] *TableName*	Designates the table or materialized view 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 32 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 clause is currently ignored.
INCLUDE (*ColumnName* [,…])	The INCLUDE clause is only supported in TimesTen Scaleout and can only be used when defining a global index. Use the INCLUDE clause to add non-index columns to the index definition. These non-index columns can be used to satisfy some queries without accessing the base table. This is especially useful if such columns are accessed frequently in queries (both for equality and for range conditions).

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>PAGES = {RowPages | CURRENT}</td><td>The PAGES clause sizes the hash index to reflect the expected number of pages in the table. If you do not specify the PAGES clause when defining a hash index, the default is CURRENT.

If you specify CURRENT, the current number of rows in the table is used to calculate the page count value. If you specify RowPages, the number of pages is used. To determine the value for RowPages, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for RowPages (256000/256=1000).

The value for RowPages must be a positive constant and must be greater than 0.

Do not specify PAGES=CURRENT if there are no rows in your table. This is because when rows are added to the table, the hash index does not perform optimally.</td></tr>
<tr><td>IndexDistributionClause :: = DISTRIBUTE BY HASH [(ColumnName [,…])]</td><td>You can specify the DISTRIBUTE BY HASH clause only if you are defining a global index. This clause is optional.

If you do not specify this clause, the column(s) defined in the global index definition form the distribution key.

If you do specify this clause, you can optionally specify the ColumnName clause:
If specified, you must then specify one or more columns for the distribution key. These columns must include one or more of the columns specified in the global index definition.
If not specified, the column(s) defined in the global index definition form the distribution key.
Note
The distribution key of the global index cannot be the same as the distribution key of the table.</td></tr>
</table>

Indexes in TimesTen Scaleout

TimesTen Scaleout supports global and local indexes:

- Global index: Maps all rows in the table to a hash distribution scheme. When you create a global index, TimesTen Scaleout creates a materialized view with a local index and a hash distribution scheme to the index key columns. The local index that is created on the materialized view further optimizes query performance.
- Local index: Is created in each database element. The index in this database element maps to rows in the table in this database element. Queries on index columns that do not include the distribution key columns on the table require communication with an element in every replica set.

See Understanding Indexes in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for information on using indexes in TimesTen Scaleout.

Choosing a Global or a Local index in TimesTen Scaleout

Consider the following when deciding whether to use a global or a local index:

- Use a global index for:

  - Unique columns: A global unique index optimizes query execution by performing unique constraint checks more efficiently. TimesTen Scaleout uses the distribution key columns for uniqueness verification instead of accessing all replica sets. However, if the distribution key is a subset of the index key, create a local index.
  - Queries that have an equality predicate that do not include all of the columns in the distribution key of the table.
  - A group of columns that are frequently joined in queries with primary key columns.
  - Non-index columns that are frequently used in queries: Define a global index with the INCLUDE clause to include those non-index columns. In such a case, the table does not need to be accessed to satisfy the query.
  - An index where the index key is a prefix of the distribution key of the table.
- Use a local index for:
  - Non-unique columns: If the index key consists of only non-unique columns, create a local non-unique index.
  - An index key that has the same columns as the distribution key for the table.
  - The situation where the distribution key of the table is a prefix of the index key.
  - Queries that have an equality predicate that includes all columns in the distribution key of the table.

See Understanding Indexes in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for more information.

Description of Global Indexes

Global index usage:

- You must specify the GLOBAL keyword to create a global index. An index is local by default.
- Global indexes by default are distributed by hash and can only be distributed by hash. Local indexes are not distributed.
- When you create a global index, TimesTen Scaleout internally creates its own materialized view and its own local index on that materialized view.
- Global indexes result in more efficient query execution with joins. However DML operations are slower due to the maintenance of the internal materialized view (that is created when you define a global index).
- When a new element is added in the grid, the schema is replicated on the new element. In addition, the rows are redistributed, and the indexes are rebuilt. This includes the global indexes. Similarly, when an element is removed from the grid, the rows are redistributed and the indexes are rebuilt.

Distribution scheme of table:

- You can define a global index on a table distributed by hash and on a table distributed by reference. Global indexes on parent and child (first-level reference) tables are supported. However, you cannot define a global index on grandchild tables or any tables that are not first-level reference tables.
- You cannot define a global index on a table distributed by duplicate.

Restrictions on global indexes:

- The column list for the distribution key cannot contain the ROWID pseudocolumn or a column of type ROWID.

- Not supported on a global temporary table.
- Not supported on readonly cache groups.
- Not supported on a materialized view.

Syntax and Semantic Rules for Global Indexes

You must specify the GLOBAL keyword to create a global index. If you do not specify the GLOBAL keyword, a local index is created. Global indexes are distributed by hash on index key columns.

If you specify the GLOBAL keyword, you can optionally specify these clauses that are specific to global indexes:

- INCLUDE clause: Optional clause that enables you to include non-key columns in the index. If such columns are frequently accessed by queries that use the index, this may improve performance.
- IndexDistributionClause: Optional clause that enables you to specify what columns to use for the hash distribution. If you do not specify this clause, then the index columns form the distribution key. The distribution key of the index cannot be the same as the distribution key of the table.
- Examples:

  Global range index:

```
Command> CREATE GLOBAL INDEX globalindex1 ON mytab (a) INCLUDE (b,c) DISTRIBUTE BY
HASH (a);
Command> indexes mytab;

Indexes on table SAMPLEUSER.MYTAB:
  MYTAB: unique range index on columns:
    C
    B
  GLOBALINDEX1: global non-unique range index on columns:
    A
    Included columns:
      B
      C
  2 indexes found.

2 indexes found on 1 table.
Command> drop table mytab;
```

  Global hash index:

```
Command> CREATE TABLE mytab (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
           a TT_INTEGER NOT NULL, PRIMARY KEY (c,b)) DISTRIBUTE BY HASH (a,b);
Command> CREATE GLOBAL HASH INDEX globalhashindex1 ON mytab(a) INCLUDE (b,c) PAGES=200
DISTRIBUTE BY HASH (a);
Command> indexes MYTAB;

Indexes on table SAMPLEUSER.MYTAB:
  MYTAB: unique range index on columns:
    C
    B
  GLOBALHASHINDEX1: global non-unique hash index on columns:
```

```
A
Included columns:
 B
 C
2 indexes found.

2 indexes found on 1 table.
```

See Examples: TimesTen Scaleout for additional examples.

General Description of Indexes in TimesTen Scaleout

- TimesTen creates a nonunique range index by default. Specify CREATE UNIQUE INDEX to create a unique range index.
- To create a nonunique hash index, specify CREATE HASH INDEX. To create a unique hash index, specify CREATE UNIQUE HASH INDEX.
- 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.
- 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 *Oracle TimesTen In-Memory Database Operations Guide*.
- If you create indexes that are redundant, TimesTen generates warnings or errors. Call ttRedundantIndexCheck to see the list of redundant indexes for your tables.
- To change the size or type of a hash index, drop the hash index and create a new index.
- A hash index is created with a fixed size that remains constant for the life of the table. To resize the hash index, drop and recreate the index. If the hash index has insufficient pages it results in hash collisions which slows down the index look-up. Hash key comparison is a fast operation, so a small number of hash collisions should not cause a performance problem for TimesTen.

  To ensure that your hash index is sized correctly, your application must indicate the expected size of your table with the value of the *RowPages* parameter of the SET PAGES clause. Compute this value by dividing the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for the value of RowPages (256000/256=1000).
- The maximum number of columns that can be specified for an index is 32.

Using Indexes in Query Processing

Proper indexes can improve query performance. Some queries can benefit from the use of indexes and some queries do not benefit from the use of indexes. Additionally, the choice of indexes for your queries is important.

A range index is ideal for processing range searches and exact matches, especially if most of the values in the index columns are unique. For example, if a range index is defined on columns (C1,C2), the index can be used to process the following types of predicates. *ConstantOrParam* refers to a constant value or dynamic parameter and *range* refers to the operators >,<,>=, or <=:

- C1 = *ConstantOrParam* AND C2 = *ConstantOrParam*
- C1 = *ConstantOrParam* AND C2 *range ConstantOrParam*
- C1 = *ConstantOrParam*
- C1 *range ConstantOrParam*

A range index efficiently processes equality and range predicates and efficiently processes sort and group operations. Use range indexes on index columns with many unique values. The order of columns you specify in a range index is relevant. The order of expressions in the predicate of a query that uses the range index is not relevant. When your query is processed, only one range index is used for each scan of your table even if you have defined multiple range indexes on your table.

A hash index efficiently processes equality predicates. You must size your hash index correctly for optimal performance. Use the PAGES parameter to size your hash index. If you specify a PAGES value that is too small, a large number of hash collisions may result, leading to performance degradation for statements that access the hash index. The order of columns specified in the hash index is not relevant and the order of expressions in the predicate of the query that uses the hash index is not relevant. If either a hash index or a range index can be used to process a particular equality predicate, the hash index is chosen because a lookup in a hash index is faster than a scan of a range index.

You can influence the indexes used by the optimizer by setting statement level or transaction level optimizer hints. see Statement Level Optimizer Hints for information on statement level optimizer hints. For more information on transaction level optimizer hints, see ttOptSetFlag, ttOptSetOrder, or ttOptUseIndex in the *Oracle TimesTen In-Memory Database Reference*.

Examples: TimesTen Scaleout

These examples illustrate the syntax requirements for creating a global index. You must specify the GLOBAL keyword to create a global index.

Illustrate Global Index Syntax

This example illustrates the supported syntax for a global index.

Create a table with three columns (c,b,a) and define a primary key on two of those columns (c,b). Distribute the table by hash on columns (a,b).

Create a global unique range index on columns (c,b).

```
Command> CREATE TABLE mytab1 (c TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL, a
TT_INTEGER NOT NULL,
PRIMARY KEY (c,b)) DISTRIBUTE BY HASH (a,b);
Command> CREATE GLOBAL UNIQUE INDEX mygix1 ON mytab1 (c,b);
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
  MYGIX1: global unique range index on columns:
    C
    B
  MYTAB1: unique range index on columns:
```

```
    C
    B
  2 indexes found.

2 indexes found on 1 table.
```

Create a second global range index and specify the INCLUDE clause.

```
Command> CREATE GLOBAL INDEX mygix2 ON MYTAB1(b) include (a);
Command> indexes mytab1

Indexes on table SAMPLEUSER.MYTAB1:
  MYGIX1: global unique range index on columns:
    C
    B
  MYTAB1: unique range index on columns:
    C
    B
  MYGIX2: global non-unique range index on columns:
    B
    Included columns:
      A
  3 indexes found.

3 indexes found on 1 table.
```

Drop the second index and recreate it. Distribute the index by hash on column b.

```
Command> DROP INDEX mygix2;
Command> CREATE GLOBAL INDEX mygix2 ON MYTAB1(b) INCLUDE (a) DISTRIBUTE BY HASH(b);
Command> INDEXES mytab1

Indexes on table SAMPLEUSER.MYTAB1:
  MYGIX1: global unique range index on columns:
    C
    B
  MYTAB1: unique range index on columns:
    C
    B
  MYGIX2: global non-unique range index on columns:
    B
    Included columns:
      A
  3 indexes found.

3 indexes found on 1 table.
```

Create a global hash index.

```
Command> CREATE GLOBAL HASH INDEX mygix3 ON mytab1(a) PAGES =200;
Command> indexes mytab1;

Indexes on table SAMPLEUSER.MYTAB1:
```

```
  MYGIX1: global unique range index on columns:
    C
    B
  MYTAB1: unique range index on columns:
    C
    B
  MYGIX3: global non-unique hash index on columns:
    A
  MYGIX2: global non-unique range index on columns:
    B
    Included columns:
      A
  4 indexes found.

4 indexes found on 1 table.
```

Distribution Key of Global Index is Same as Distribution Key of Table

This example illustrates that you cannot create a global index whose distribution key is the same as the distribution key of the table. In this example, the `mytab1` table is distributed by hash on columns `(a,b)`. An attempt to create a global index, with columns `(a,b)` as the distribution key, results in an error.

```
Command> CREATE TABLE mytab1 (a TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
         c TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> CREATE GLOBAL INDEX gix1 ON mytab1(a,b,c) DISTRIBUTE BY HASH (a,b);
 2253: Distribution key for global index cannot be same as that of the table or
 other global index. Consider creating a local index.
The command failed.
```

Global Index Creates Its Own Materialized View and its Own Local Index

This example illustrates that when you create a global index, TimesTen Scaleout creates its own internal materialized view and its own local index. Create the `mytab2` table distributed by hash on columns `(a,b)`. Create a global non-unique range index distributed by hash on columns `(b,a)`. Run the `ttIsql indexes` command to show the `gix2` global index is created as well as an internal local index on the internal materialized view. Then, run the `ttIsql views` command to show an internal materialized view is also created as a result of creating the global index. Run the `ttIsql describe` command to show the internal materialized view. Note that you cannot explicitly drop the internal materialized view or the internal local index.

```
Command> CREATE TABLE mytab2 (a TT_INTEGER NOT NULL, b TT_INTEGER NOT NULL,
         c TT_INTEGER NOT NULL) DISTRIBUTE BY HASH (a,b);
Command> CREATE GLOBAL INDEX gix2 ON mytab2(a,b,c) DISTRIBUTE BY HASH (b,a);
Command> indexes;

Indexes on materialized view SAMPLEUSER.$GV9B55D3955D52:
  $GVI9B55D3955D52: non-unique range index on columns:
    A
    B
    C
  1 index found.

Indexes on table SAMPLEUSER.MYTAB2:
  GIX2: global non-unique range index on columns:
    A
    B
    C
  1 index found.
```

```
2 indexes found on 2 tables.
Command> views;
  SAMPLEUSER.$GV9B55D3955D52
1 view found.
Command> describe SAMPLEUSER.$GV9B55D3955D52;

Materialized view SAMPLEUSER.$GV9B55D3955D52:
  Global index: GIX2 (table: MYTAB2)
  Columns:
    A                          TT_INTEGER NOT NULL
    B                          TT_INTEGER NOT NULL
    C                          TT_INTEGER NOT NULL
  DISTRIBUTE BY HASH (B, A)

1 view found.
```

Examples: TimesTen Classic

Create a table and then create a unique hash index on *col2*. Do not specify the PAGES clause. If PAGES is not specified, the current table page count is used for the size of the hash table. Use INDEXES to verify the index was created. Insert a row in the table, set SHOWPLAN to 1 and then verify the optimizer uses the hash index.

```
Command> CREATE TABLE tab (col1 NUMBER PRIMARY KEY NOT NULL, col2 VARCHAR2 (30));
Command> CREATE UNIQUE HASH INDEX hash1 ON tab (col2);
Command> INDEXES;

Indexes on table TESTUSER.TAB:
  HASH1: unique hash index on columns:
    COL2
  TAB: unique range index on columns:
    COL1
  2 indexes found.

2 indexes found on 1 table.
Command> INSERT INTO tab VALUES (10, 'ABC');
Command> SHOWPLAN 1;
Command> SELECT * FROM tab where col2 = 'ABC';

Query Optimizer Plan:

  STEP:             1
  LEVEL:            1
  OPERATION:        RowLkHashScan
  TBLNAME:          TAB
  IXNAME:           HASH1
  INDEXED CONDITION:   TAB.COL2 = 'ABC'
  NOT INDEXED:      <NULL>

< 10, ABC >
1 row found.
```

Create a table and create a nonunique hash index on *col1*. Use PAGES = CURRENT to use the current table page count to size the hash index. Use INDEXES to verify the nonunique hash index is created.

```
Command> CREATE TABLE tab2 (col1 NUMBER);
Command> CREATE HASH INDEX hash_index ON tab2 (col1) PAGES = CURRENT;
Command> INDEXES;

Indexes on table TESTUSER.TAB2:
```

```
  HASH_INDEX: non-unique hash index on columns:
    COL1
  1 index found.

1 index found on 1 table.
```

Create table and create unique hash index on *col3*. Use PAGES = 100 to specify a page count of 100 for the size of the hash table. Use INDEXES to verify the unique hash index is created.

```
Command> CREATE TABLE tab3 (col1 NUMBER, col2 NUMBER, col3 TT_INTEGER);
Command> CREATE UNIQUE HASH INDEX unique_hash1 on tab3 (col3) PAGES = 100;
Command> INDEXES;

Indexes on table TESTUSER.TAB3:
  UNIQUE_HASH1: unique hash index on columns:
    COL3
  1 index found.

1 index found on 1 table.
```

The regions table in the HR schema creates a unique index on region_id. Issue the ttIsql INDEXES command on table regions. You see the unique range index regions.

```
Command> INDEXES REGIONS;

Indexes on table SAMPLEUSER.REGIONS:
  REGIONS: unique range index on columns:
    REGION_ID
    (referenced by foreign key index COUNTR_REG_FK on table SAMPLEUSER.COUNTRIES)
  1 index found.

1 index found on 1 table.
```

Attempt to create a unique index i on table regions indexing on column region_id. You see a warning message.

```
Command> CREATE UNIQUE INDEX i ON regions (region_id);
Warning  2232: New index I is identical to existing index REGIONS;
consider dropping index I
```

Call ttRedundantIndexCheck to see warning message for this index:

```
Command> CALL ttRedundantIndexCheck ('regions');
< Index SAMPLEUSER.REGIONS.I is identical to index SAMPLEUSER.REGIONS.REGIONS;
consider dropping index SAMPLEUSER.REGIONS.I >
1 row found.
```

Create table redundancy and define columns co11 and col2. Create two user indexes on col1 and col2. You see an error message when you attempt to create the second index r2. Index r1 is created. Index r2 is not created.

```
Command> CREATE TABLE redundancy (col1 CHAR (30), col2 VARCHAR2 (30));
Command> CREATE INDEX r1 ON redundancy (col1, col2);
Command> CREATE INDEX r2 ON redundancy (col1, col2);
 2231: New index R2 would be identical to existing index R1
The command failed.
```

Issue the ttIsql command INDEXES on table redundancy to show that only index r1 is created:

```
Command> INDEXES redundancy;

Indexes on table SAMPLEUSER.REDUNDANCY:
```

```
  R1: non-unique range index on columns:
    COL1
    COL2
  1 index found.

1 index found on 1 table.
```

This unique index ensures that all part numbers are unique.

```
CREATE UNIQUE INDEX purchasing.partnumindex
ON purchasing.parts (partnumber);
```

Create a linguistic index named `german_index` on table `employees1`. 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 range index on columns:
    NLSSORT(ID,'NLS_SORT=GERMAN')
  GERMAN_INDEX2: non-unique range index on columns:
    NLSSORT(ID2,'nls_sort=german_ci')
  2 indexes found.
1 table found.
```

See Also

DROP INDEX

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*. The view is refreshed synchronously with regard to changes in the detail tables.

Required Privileges

User executing the statement must have `CREATE MATERIALIZED VIEW` (if owner) or `CREATE ANY MATERIALIZED VIEW` (if not owner) privilege.

Owner of the materialized view must have:

- `SELECT` privilege on the detail tables.
- `CREATE TABLE` privilege.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. You must specify the `DISTRIBUTE BY HASH` clause and you must define a distribution key. The `DISTRIBUTE BY REFERENCE` and `DUPLICATE` clauses are not supported.

SQL Syntax: TimesTen Scaleout

```
CREATE MATERIALIZED VIEW [Owner.]ViewName
DISTRIBUTE BY HASH (ColumnName [,...])
```

AS *SelectQuery*
[PRIMARY KEY (*ColumnName* [,...])]
[UNIQUE HASH ON (*HashColumnName* [,...]) PAGES = *PrimaryPages*]

SQL Syntax: TimesTen Classic

CREATE MATERIALIZED VIEW [*Owner*.]*ViewName*
AS *SelectQuery*
[PRIMARY KEY (*ColumnName* [,...])]
[UNIQUE HASH ON (*HashColumnName* [,...]) PAGES = *PrimaryPages*]

Parameters

Parameter	Description
[*Owner*.]*ViewName*	Name assigned to the new view.
DISTRIBUTE BY HASH (*ColumnName* [,...])	TimesTen Scaleout only. You must specify the DISTRIBUTE BY HASH clause and you must specify one or more columns for the distribution key (even if you have specified a primary key). The detail table must be distributed by hash. DISTRIBUTE BY REFERENCE or DUPLICATE clauses are not supported. This clause must appear before the AS *SelectQuery* clause.
SelectQuery	Select column from the detail tables to be used in the view.
ColumnName	Name of the column(s) that forms the primary key for the view to be created. Up to 32 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.
UNIQUE HASH ON	Hash index for the table. Only unique hash indexes are created. This parameter is used for equality predicates. UNIQUE HASH ON requires that a primary key be defined.
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.
PAGES = *PrimaryPages*	Sizes the hash index to reflect the expected number of pages in your table. To determine the value for *PrimaryPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for *PrimaryPages* (256000/256=1000). The value for *PrimaryPages* must be a positive constant and must be greater than 0. If your estimate for *PrimaryPages* is too small, performance may be degraded. See CREATE TABLE for information on hash indexes.

Description and Restrictions for CREATE MATERIALIZED VIEW: TimesTen Scaleout

Description and restrictions include:

- The SQL optimizer may re-write a query against a base table to use an available materialized view if the use of the materialized view is expected to improve the execution time of the query.
- You must specify the DISTRIBUTE BY HASH clause and you must specify it with a distribution key (even if you have specified a primary key and intend to use the primary key as the distribution key).
- You must specify the DISTRIBUTE BY HASH clause before the AS *SelectQuery* clause.
- You can only specify the DISTRIBUTE BY HASH clause. The DISTRIBUTE BY REFERENCE and DUPLICATE clauses are not supported.
- The *SelectQuery* must be restricted to single table SELECT statements.
- You cannot specify the GROUP BY or the WHERE clause in the *SelectQuery*.
- You cannot use SQL functions in the *SelectQuery*.
- You cannot use an expression in the *SelectQuery*.
- The detail table of the materialized view cannot have a foreign key with a cascade delete clause.
- The distribution key columns must be in the project list of the *SelectQuery*.
- There are no DDL rewrites. For example, if you create a unique index on the detail table, a corresponding index on the materialized view (which is distributed on the unique column) is not created.

Description: TimesTen Scaleout and TimesTen Classic

The restrictions and requirements on the defining query include:

- Each expression in the select list must have a unique name.
- Do not use non-materialized views to define a materialized view.
- Do not define CLOB, BLOB, or NCLOB data types for columns in the select list of the materialized view query.
- The detail tables cannot belong to a cache group and the detail tables cannot have compression.
- Do not use SELECT FOR UPDATE.
- Do not reference system tables or views.
- Do not use nested definitions for a materialized view.
- Do not use dynamic parameters.
- Do not use ROWNUM.
- Do not use analytic functions.
- Do not use GROUPING SETS, ROLLUP, or CUBE.
- Do not use the SYSDATE function.
- Do not use the functions SYSTEM_USER, USER, CURRENT_USER, or SESSION_USER.
- Do not use NEXTVAL or CURRVAL.

- 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.
- Do not use the WITH *subquery* clause.

The restrictions (not on the defining query) include:

- Do not have a hash-based primary key that contains any aggregate columns of the materialized view.
- A materialized view cannot be replicated directly using TimesTen replication. You can replicate the detail tables. You must define the same materialized view on both sides of replication. TimesTen automatically updates the corresponding materialized views.
- You cannot define a foreign key if the referencing or referenced table is a materialized view.

The following restrictions and requirements on the defining query are:

- The view definition must include all columns in the group by list in the select list.
- An aggregate view must include a COUNT (*) or COUNT (non-nullable column) in the select list.
- Do not use derived tables or JOIN tables.
- Do not use SELECT DISTINCT or an aggregate distinct function.
- Do not use the set operators UNION, MINUS, or INTERSECT.
- Do not use SUM of nullable expressions.
- Use only simple columns as group by columns.
- Group by columns cannot belong to self join tables.
- Do not use these clauses:
  - HAVING
  - ORDER BY
  - DISTINCT
  - FIRST
  - JOIN
- Do not use the TT_HASH function.
- You can use SUM and COUNT but do not use expressions involving SUM and COUNT. Do not use AVG, which is treated as SUM/COUNT.
- Do not specify MIN or MAX functions in the select list.
- For joins:
  - Join predicates cannot have an OR.
  - Do not specify Cartesian product joins (joins with no join predicate).
  - For outer joins, outer join each inner table with at most one table.

Additional considerations include:

- 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.

- By default, a range index is created to enforce the primary key for a materialized view. Alternatively, 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 range index for that view by omitting the UNIQUE HASH clause.
  - If 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 for more information about the UNIQUE HASH clause.
- You can use ALTER TABLE to change the representation of the primary key index or resize a hash index of a materialized view.
- You cannot add or drop columns in the materialized view with the ALTER TABLE statement. To change the structure of the materialized view, drop and recreate the view.
- You can create indexes on the materialized view with the CREATE INDEX SQL statement.

The owner of a materialized view must have the SELECT privilege on its detail tables. The SELECT privilege is implied by the SELECT ANY TABLE and ADMIN system privileges. When the SELECT privilege or a higher-level system privilege on the detail tables is revoked from the owner of the materialized view, the materialized view becomes *invalid*.

Selecting from an invalid materialized view fails with an error. Updates to the detail tables of an invalid materialized view do not update the materialized view.

You can identify invalid materialized views by using the ttIsql describe command and by inspecting the STATUS column of the SYS.DBA_OBJECTS, SYS.ALL_OBJECTS or SYS.USER_OBJECTS system tables. See *Oracle TimesTen In-Memory Database System Tables and Views Reference*.

If the revoked privilege is restored, you can make an invalid materialized view valid again by dropping and recreating the materialized view.

For more information, see Object Privileges for Materialized Views in *Oracle TimesTen In-Memory Database Security Guide*.

Examples for CREATE MATERIALIZED VIEW: TimesTen Scaleout

Syntax example:

```
Command> CREATE MATERIALIZED VIEW mv
         DISTRIBUTE BY HASH (phone)
         AS SELECT phone FROM accounts;
1010 rows materialized.
```

Examples: TimesTen Classic

Create 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;
```

Create 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 ON (x1) PAGES=100;
```

Create 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(+);
```

The following example creates a materialized view `empmatview2` based on selected columns `employee_id` and `email` from table `employees`. After the materialized view is created, create an index on the materialized view column `mvemp_id` of the materialized view `empmatview2`.

```
CREATE MATERIALIZED VIEW empmatview2
   AS SELECT employee_id mvemp_id, email mvemail
        FROM employees;
107 rows materialized.

CREATE INDEX empmvindex ON empmatview2 (mvemp_id);
```

See Also

- CREATE INDEX
- CREATE TABLE
- DROP MATERIALIZED VIEW

CREATE PACKAGE

The `CREATE PACKAGE` statement creates the specification for a standalone package, which is an encapsulated collection of related procedures, functions, and other program objects stored together in your database. The package specification declares these objects. The package body defines these objects.

Required Privilege

`CREATE PROCEDURE` (if owner) or `CREATE ANY PROCEDURE` (if not owner).

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE [OR REPLACE] PACKAGE [Owner.]PackageName
     [InvokerRightsClause] [AccessibleByClause]
     {IS|AS}
     PlsqlPackageSpec

InvokerRightsClause::=
AUTHID {CURRENT_USER | DEFINER}

AccessibleByClause::=
ACCESSIBLE BY (accessor[,...])

accessor::=
[UnitKind][Owner.]UnitName
```

You can specify *InvokerRightsClause* or *AccessibleByClause* in any order.

Parameters

Parameter	Description
OR REPLACE	Specify OR REPLACE to recreate the package specification if it already exists. Use this clause to change the specification of an existing package without dropping and recreating the package. When you change a package specification, TimesTen recompiles it.
PackageName	Name of the package.
InvokerRightsClause	Lets you specify whether the SQL statements in PL/SQL functions or procedures execute with definer's or invoker's rights. The AUTHID setting affects the name resolution and privilege checking of SQL statements that a PL/SQL procedure or function issues at runtime, as follows: • Specify DEFINER so that SQL name resolution and privilege checking operate as though the owner of the procedure or function (the definer, in whose schema it resides) is running it. DEFINER is the default. • Specify CURRENT_USER so that SQL name resolution and privilege checking operate as though the current user (the invoker) is running it. For more information, see Definer's Rights and Invoker's Rights (AUTHID Clause) in the *Oracle TimesTen In-Memory Database Security Guide*.
AccessibleByClause	Use this clause to specify one or more *accessors* (PL/SQL units) that can invoke the package directly. The list of accessors that can access the package is called a *white list*. A white list gives you the ability to add an extra layer of security to your PL/SQL objects. Specifically, you can restrict access to the package to only those objects on the white list. *AccessibleByClause* can appear only once in the CREATE PACKAGE statement. Syntax: ACCESSIBLE BY (*accessor* [,...])
accessor	Used in *AccessibleByClause*. An accessor is a PL/SQL unit that can invoke the package. An accessor can appear more than once in the *AccessibleByClause* clause. Syntax: [*UnitKind*][*Owner*.]*UnitName*
UnitKind	Used in the *accessor* clause (which is part of the *AccessibleByClause* clause). Specifies the kind of PL/SQL unit that can invoke the package. *UnitKind* is optional, but if specified, valid options are: • FUNCTION • PROCEDURE • PACKAGE

Parameter	Description
[*Owner*.]*UnitName*	Used in the *accessor* clause (which is part of the *AccessibleByClause* clause). Specifies the name of the PL/SQL unit that can invoke the package. If you specify *UnitKind*, then *UnitName* must be a name of a unit of that kind. For example, if you specify PROCEDURE for *UnitKind*, then *UnitName* must be the name of a procedure. *UnitName* is required. You can optionally specify *Owner*. If you specify *Owner*, then *UnitName* must reside in that owner's schema. If you do not specify *Owner*, *UnitName* must be in the schema that contains the package.
IS|AS	Specify either IS or AS to declare the body of the function.
PlsqlPackageSpec	Specifies the package specification. Can include type definitions, cursor declarations, variable declarations, constant declarations, exception declarations and PL/SQL subprogram declarations.

Description

- *AccessibleByClause*:
  - *AccessibleByClause* is valid at the top-level package definition. You cannot specify *AccessibleByClause* in the individual procedures or functions within the package. In addition, you cannot specify *AccessibleByClause* in the CREATE PACKAGE BODY statement.
  - You can use this clause to restrict access to helper packages. For example, assume your PL/SQL package defines an API for a given functionality and that functionality is implemented using a set of helper procedures and functions. You want to limit applications to only be able to call the API procedure or function that is defined in your package, and to not be able to call the helper procedures and functions directly. You can use the ACCESSIBLE BY clause to achieve this.
  - The compiler checks the validity of the syntax of the ACCESSIBLE BY clause, but does not check that the accessor exists. Therefore, you can define an accessor that does yet exist in the owner's schema.
  - When you invoke the package, the compiler first does the normal permission checks on the invocation. If any check fails, the invocation fails, even if the invoker is an accessor. If all normal permission checks on the invocation succeed, and the package has no ACCESSIBLE BY clause, the invocation succeeds. If the package has an ACCESSIBLE BY clause, the invocation succeeds only if the invoker is an accessor.
- When you create or replace a package, the privileges granted on the package remain the same. If you drop and recreate the object, the object privileges that were granted on the original object are revoked.
- In a replicated environment, the CREATE PACKAGE statement is not replicated. For more information, see Creating a New PL/SQL Object in an Existing Active Standby Pair and Adding a PL/SQL Object to an Existing Classic Replication Scheme in the *Oracle TimesTen In-Memory Database Replication Guide*.

Examples

Illustrating the Correct Usage of the Accessible By Clause

This example illustrates the correct usage of the *AccessibleByClause*. The clause is specified at the top-level of the CREATE PACKAGE statement. Note that the CallingProc procedure does not need to exist.

```
Command> CREATE OR REPLACE PACKAGE ProtectedPkg
           ACCESSIBLE BY (PROCEDURE CallingProc)
         AS
           PROCEDURE ProtectedProc;
         END;
         /

Package created.
```

Illustrating the Incorrect Usage of the Accessible By Clause

These examples show the incorrect use of the AccessibleByClause. The first example attempts to use AccessibleByClause in the packaged procedure, resulting in a compilation error. The second example attempts to use AccessibleByClause in the CREATE PACKAGE BODY statement, resulting in a compilation error.

This example uses the ACCESSIBLE BY clause in the packaged procedure.

```
Command> CREATE OR REPLACE PACKAGE ProtectedPkg1
         AS
           PROCEDURE ProtectedProc1
           ACCESSIBLE BY (PROCEDURE CallingProc)
         END;
         /

Warning: Package created with compilation errors.

Command> SHOW ERRORS
Errors for PACKAGE PROTECTEDPKG1:

LINE/COL ERROR
-------- -----------------------------------------------------------------
0/0      PLS-00157: Only schema-level programs allow ACCESSIBLE BY
```

This example uses the ACCESSIBLE BY clause in the CREATE PACKAGE BODY statement.

```
Command> CREATE OR REPLACE PACKAGE ProtectedPkg3
           ACCESSIBLE BY (PROCEDURE CallingProc3)
         AS
           PROCEDURE ProtectedProc3;
         END;
         /

Package created.

Command> CREATE OR REPLACE PACKAGE BODY ProtectedPkg3
           ACCESSIBLE BY (PROCEDURE CallingProc3)
         AS
           PROCEDURE ProtectedProc3 AS
           BEGIN
             NULL;
           END;
           ;
         /

Warning: Package body created with compilation errors.

Command> SHOW ERRORS
Errors for PACKAGE BODY PROTECTEDPKG3:

LINE/COL ERROR
-------- -----------------------------------------------------------------
```

```
2/1      PLS-00103: Encountered the symbol "ACCESSIBLE" when expecting one of the
following:

   is as compress compiled wrapped
```

Ensuring Only the API Can Access the Helper Package

This example walks through a series of steps to illustrate the use of the *AccessibleByClause*. The example creates the SampleAPI package and the SampleHelper package. The ACCESSIBLE BY clause is specified on the SampleHelper to ensure that only the SampleAPI package can access the SampleHelper package.

Steps:

1. Create the SampleHelper package. Specify the ACCESSIBLE BY clause, giving the SampleAPI package access to the SampleHelper package. The SampleAPI package is in the white list.

   ```
   Command> CREATE OR REPLACE PACKAGE SampleHelper
              ACCESSIBLE BY (SampleAPI)
            AS
              PROCEDURE SampleH1;
              PROCEDURE SampleH2;
            END;
            /

   Package created.
   ```

2. Create the SampleHelper package body.

   ```
   Command> CREATE OR REPLACE PACKAGE BODY SampleHelper
            AS
              PROCEDURE SampleH1 AS
              BEGIN
                DBMS_OUTPUT.PUT_LINE('Sample helper procedure SampleH1');
              END;
              PROCEDURE SampleH2 AS
              BEGIN
                DBMS_OUTPUT.PUT_LINE('Sample helper procedure SampleH2');
              END;
            END;
            /

   Package body created.
   ```

3. Create the SampleAPI package.

   ```
   Command> CREATE OR REPLACE PACKAGE SampleAPI
            AS
              PROCEDURE p1;
              PROCEDURE p2;
            END;
            /

   Package created.
   ```

4. Create the SampleAPI package body. The p1 procedure references the SampleHelper.SampleH1 procedure. The p2 procedure references the SampleHelper.SampleH2 procedure.

   ```
   Command> CREATE OR REPLACE PACKAGE BODY SampleAPI
            AS
              PROCEDURE p1 AS
              BEGIN
                DBMS_OUTPUT.PUT_LINE('SampleAPI procedure p1');
                SampleHelper.SampleH1;
   ```

```
    END;
    PROCEDURE p2 AS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('SampleAPI procedure p2');
      SampleHelper.SampleH2;
    END;
  END;
  /

Package body created.
```

5. Call the SampleAPI.p1 and the SampleAPI.p2 procedures. The SampleAPI package is in the white list of the SampleHelper package, resulting in successful execution.

```
Command> SET SERVEROUTPUT ON
Command> BEGIN
           SampleAPI.p1;
           SampleAPI.p2;
         END;
         /
SampleAPI procedure p1
Sample helper procedure SampleH1
SampleAPI procedure p2
Sample helper procedure SampleH2

PL/SQL procedure successfully completed.
```

6. Call the SampleHelper.SampleH1 procedure directly. An error is returned due to insufficient access privileges.

```
Command> BEGIN
           SampleHelper.SampleH1;
         END;
        /
 8503: ORA-06550: line 2, column 3:
PLS-00904: insufficient privilege to access object SAMPLEHELPER
 8503: ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
The command failed.
```

CREATE PACKAGE BODY

The CREATE PACKAGE BODY statement creates the body of a standalone package. A package is an encapsulated collection of related procedures, functions, and other program objects stored together in your database. A package specification declares these objects. A package body defines these objects.

Required Privilege

CREATE PROCEDURE (if owner) or CREATE ANY PROCEDURE (if not owner).

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE [OR REPLACE] PACKAGE BODY [Owner.]PackageBody
    {IS|AS} plsql_package_body
```

Parameters

Parameter	Description
OR REPLACE	Specify OR REPLACE to recreate the package body if it already exists. Use this clause to change the body of an existing package without dropping and recreating it. When you change a package body, TimesTen recompiles it.
PackageBody	Name of the package body.
IS|AS	Specify either IS or AS to declare the body of the function.
plsql_package_body	Specifies the package body which consists of PL/SQL subprograms.

Description

In a replicated environment, the CREATE PACKAGE BODY statement is not replicated. For more information, see Creating a New PL/SQL Object in an Existing Active Standby Pair and Adding a PL/SQL Object to an Existing Classic Replication Scheme in the *Oracle TimesTen In-Memory Database Replication Guide*.

When you create or replace a package body, the privileges granted on the package body remain the same. If you drop and recreate the object, the object privileges that were granted on the original object are revoked.

CREATE PROCEDURE

The CREATE PROCEDURE statement creates a standalone stored procedure.

Required Privilege

CREATE PROCEDURE (if owner) or CREATE ANY PROCEDURE (if not owner).

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE [OR REPLACE] PROCEDURE [Owner.]ProcedureName
   [(arguments [IN|OUT|IN OUT][NOCOPY] DataType [DEFAULT expr][,...])]
   [InvokerRightsClause][AccessibleByClause] [DETERMINISTIC]
   {IS|AS} plsql_procedure_body

InvokerRightsClause::=
AUTHID {CURRENT_USER|DEFINER}

AccessibleByClause::=
ACCESSIBLE BY(accessor[,...])

accessor::=
[UnitKind][Owner.]UnitName
```

You can specify *InvokerRightsClause*, *AccessibleByClause*, or DETERMINISTIC in any order.

Parameters

Parameter	Description
OR REPLACE	Specify OR REPLACE to recreate the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping and recreating it. When you recreate a procedure, TimesTen recompiles it.
ProcedureName	Name of procedure.
arguments	Name of argument/parameter. You can specify 0 or more parameters for the procedure. If you specify a parameter, you must specify a data type for the parameter. The data type must be a PL/SQL data type.
[IN|OUT|IN OUT]	Parameter modes. IN is a read-only parameter. You can pass the parameter's value into the procedure but the procedure cannot pass the parameter's value out of the procedure and back to the calling PL/SQL block.The value of the parameter cannot be changed. OUT is a write-only parameter. Use an OUT parameter to pass a value back from the procedure to the calling PL/SQL block. You can assign a value to the parameter. IN OUT is a read/write parameter. You can pass values into the procedure and return a value back to the calling program (either the original, unchanged value or a new value set within the procedure. IN is the default.
NOCOPY	Specify NOCOPY to instruct TimesTen to pass the parameter as fast as possible. Can enhance performance when passing a large value such as a record, an index-by-table, or a varray to an OUT or IN OUT parameter. IN parameters are always passed NOCOPY.
DEFAULT *expr*	Use this clause to specify a DEFAULT value for the parameter. You can specify := in place of the keyword DEFAULT.
InvokerRightsClause	Lets you specify whether the SQL statements in PL/SQL functions or procedures execute with definer's or invoker's rights. The AUTHID setting affects the name resolution and privilege checking of SQL statements that a PL/SQL procedure or function issues at runtime, as follows: • Specify DEFINER so that SQL name resolution and privilege checking operate as though the owner of the procedure or function (the definer, in whose schema it resides) is running it. DEFINER is the default. • Specify CURRENT_USER so that SQL name resolution and privilege checking operate as though the current user (the invoker) is running it. For more information, see Definer's Rights and Invoker's Rights (AUTHID Clause) in the *Oracle TimesTen In-Memory Database Security Guide*.

Parameter	Description
AccessibleByClause	Use this clause to specify one or more *accessors* (PL/SQL units) that can invoke the procedure directly. The list of accessors that can access the procedure is called a *white list*. A white list gives you the ability to add an extra layer of security to your PL/SQL objects. Specifically, you can restrict access to the procedure to only those objects on the white list. The *AccessibleByClause* can appear only once in the CREATE PROCEDURE statement. Syntax: ACCESSIBLE BY (*accessor* [,...])
accessor	Used in the *AccessibleByClause*. An accessor is a PL/SQL unit that can invoke the procedure. An accessor can appear more than once in the *AccessibleByClause*. Syntax: [*UnitKind*][*Owner*.]*UnitName*
UnitKind	Used in the *accessor* clause (which is part of the *AccessibleByClause* clause). Specifies the kind of PL/SQL unit that can invoke the procedure. *UnitKind* is optional, but if specified, valid options are: • FUNCTION • PROCEDURE • PACKAGE
[*Owner*.]*UnitName*	Used in the *accessor* clause (which is part of the *AccessibleByClause* clause). Specifies the name of the PL/SQL unit that can invoke the procedure. If you specify *UnitKind*, then *UnitName* must be a name of a unit of that kind. For example, if you specify PROCEDURE for *UnitKind*, then *UnitName* must be the name of a procedure. *UnitName* is required. You can optionally specify *Owner*. If you specify *Owner*, then *UnitName* must reside in that owner's schema. If you do not specify *Owner*, *UnitName* must be in the schema that contains the procedure.
DETERMINISTIC	Specify DETERMINISTIC to indicate that the procedure returns the same result value whenever it is called with the same values for its parameters.
IS|AS	Specify either IS or AS to declare the body of the procedure.
plsql_procedure_body	Specifies the procedure body.

Description

- *AccessibleByClause*:
  - The compiler checks the validity of the syntax of the *AccessibleByClause*, but does not check that the accessor exists. Therefore, you can define an accessor that does yet exist in the owner's schema.
  - When you invoke the procedure, the compiler first does the normal permission checks on the invocation. If any check fails, the invocation fails, even if the invoker is an accessor. If all normal permission checks on the invocation succeed, and the procedure has no AccessibleByClause, the invocation succeeds. If the procedure has an AccessibleByClause, the invocation succeeds only if the invoker is an accessor.
- When you create or replace a procedure, the privileges granted on the procedure remain the same. If you drop and recreate the object, the object privileges that were granted on the original object are revoked.

- The namespace for PL/SQL procedures is distinct from the TimesTen built-in procedures. You can create a PL/SQL procedure with the same name as a TimesTen built-in procedure.
- TimesTen does not support:
  - call_spec clause
  - AS EXTERNAL clause
- In a replicated environment, the CREATE PROCEDURE statement is not replicated. For more information, see Creating a New PL/SQL Object in an Existing Active Standby Pair and Adding a PL/SQL Object to an Existing Classic Replication Scheme in the *Oracle TimesTen In-Memory Database Replication Guide*.

Examples

Using the Accessible By Clause

This example creates the ProtectedProc procedure and uses the ACCESSIBLE BY clause to restrict access to the CallingProc procedure. The CallingProc procedure does not yet exist. The example then creates the CallingProc procedure, which calls the ProtectedProc procedure. The CallingProc procedure is successfully created, as it is specified in the ACCESSIBLE BY clause. The example then attempts to call the ProtectedProc procedure directly, resulting in an error. It concludes with attempting to create the AnotherCallingProc procedure that references the ProtectedProc procedure, but the AnotherCallingProc procedure is not in the white list. A compilation error results.

Steps to illustrate the example:

1. Create the ProtectedProc procedure, specifying the ACCESSIBLE BY clause. The CallingProc procedure is in the white list. It does not yet exist.

```
Command> CREATE OR REPLACE PROCEDURE ProtectedProc
          ACCESSIBLE BY (CallingProc)
        AS
        BEGIN
          DBMS_OUTPUT.PUT_LINE ('ProtectedProc');
        END;
        /

Procedure created.
```

2. Create the CallingProc procedure, referencing the ProtectedProc procedure.

```
Command> CREATE OR REPLACE PROCEDURE CallingProc
        AS
        BEGIN
          DBMS_OUTPUT.PUT_LINE ('CallingProc');
          ProtectedProc;
        END;
        /

Procedure created.
```

3. Call the CallingProc procedure. The procedure is successfully executed.

```
Command> SET SERVEROUTPUT ON
Command> exec CallingProc;
CallingProc
ProtectedProc

PL/SQL procedure successfully completed.
```

4. Attempt to call the ProtectedProc procedure directly. An error is thrown due to insufficient access privileges.

```
Command> exec ProtectedProc;
 8503: ORA-06550: line 1, column 7:
PLS-00904: insufficient privilege to access object PROTECTEDPROC
 8503: ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
The command failed.
```

5. Create the AnotherCallingProc procedure that references the ProtectedProc procedure. The AnotherCallingProc is not in the white list (not listed in the ACCESSIBLE BY clause of ProtectedProc), resulting in a compilation error.

```
Command> CREATE OR REPLACE PROCEDURE AnotherCallingProc
       AS
       BEGIN
         DBMS_OUTPUT.PUT_LINE ('AnotherCallingProc');
         ProtectedProc;
       END;
       /

Warning: Procedure created with compilation errors.

Command> SHOW ERRORS
Errors for PROCEDURE ANOTHERCALLINGPROC:

LINE/COL ERROR
-------- -----------------------------------------------------------------
5/1      PL/SQL: Statement ignored
5/1      PLS-00904: insufficient privilege to access object PROTECTEDPROC
```

Using the Accessor Clause

This example illustrates the uses of the accessor clause through a sequence of steps.

1. Create the SampleUser1 and SampleUser2 users and grant ADMIN privileges to both users.

```
Command> CREATE USER SampleUser1 IDENTIFIED BY SampleUser1;

User created.

Command> CREATE USER SampleUser2 IDENTIFIED BY SampleUser2;

User created.
Command> GRANT ADMIN TO SampleUser1, SampleUser2;
```

2. Create the SampleUser1.ProtectedProc procedure, specifying the ACCESSIBLE BY clause. The CallingProc procedure is specified in the white list without an owner. The owner of the CallingProc procedure is assumed to be in the same schema as the owner of the procedure with the ACCESSIBLE BY clause. Thus, CallingProc is assumed to be in the SampleUser1 schema.

```
Command> CREATE OR REPLACE PROCEDURE SampleUser1.ProtectedProc
       ACCESSIBLE BY (CallingProc)
       AS
       BEGIN
         DBMS_OUTPUT.PUT_LINE ('SampleUser1 ProtectedProc');
       END;
       /

Procedure created.
```

3. Connect as SampleUser1. Create the CallingProc procedure, referencing the SampleUser1.ProtectedProc procedure.

```
Command> Connect adding "uid=SampleUser1;pwd=SampleUser1PW" as SampleUser1;
Connection successful:
DSN=database1;UID=SampleUser1;DataStore=/scratch/sampleuser1/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;
PermSize=128;
(Default setting AutoCommit=1)

sampleuser1: Command> CREATE OR REPLACE PROCEDURE CallingProc
              AS
              BEGIN
                DBMS_OUTPUT.PUT_LINE ('SampleUser1 CallingProc');
                ProtectedProc;
              END;
              /

Procedure created.
```

4. From the SampleUser1 connection, call the CallingProc procedure. The call succeeds.

```
sampleuser1: Command> SET SERVEROUTPUT ON
sampleuser1: Command> exec CallingProc;
SampleUser1 CallingProc
SampleUser1 ProtectedProc

PL/SQL procedure successfully completed.
```

5. Connect to SampleUser2. Create the CallingProc procedure, referencing the SampleUser1.ProtectedProc procedure. A compilation error results.

```
SampleUser1: Command> connect adding "uid=Sampleuser2;pwd=SampleUser2PW"
              as SampleUser2;
Connection successful:
DSN=database1;UID=Sampleuser2;DataStore=/scratch/sampleuser2/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;
PermSize=128;
(Default setting AutoCommit=1)

sampleuser2: Command> CREATE OR REPLACE PROCEDURE CallingProc
              AS
              BEGIN
                DBMS_OUTPUT.PUT_LINE ('SampleUser2 CallingProc');
                SampleUser1.ProtectedProc;
              END;
              /

Warning: Procedure created with compilation errors.

sampleuser2: Command> SHOW ERRORS
Errors for PROCEDURE CALLINGPROC:

LINE/COL ERROR
-------- -----------------------------------------------------------------
5/1      PL/SQL: Statement ignored
5/1      PLS-00904: insufficient privilege to access object PROTECTEDPROC
```

6. Switch to the SampleUser1 connection. Recreate the ProtectedProc procedure.

```
sampleuser2: Command> use SampleUser1
sampleuser1: Command> CREATE OR REPLACE PROCEDURE ProtectedProc
              ACCESSIBLE BY (CallingProc, SampleUser2.CallingProc)
              AS
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('SampleUser1 ProtectedProc');
END;
/

Procedure created.
```

7. From the SampleUser2 connection, call the CallingProc procedure. The SampleUser2.CallingProc is in the white list of the SampleUser1.ProtectedProc procedure, resulting in successful execution.

```
sampleuser1: Command> use SampleUser2;
sampleuser2: Command> SET SERVEROUTPUT ON
sampleuser2: Command> exec CallingProc
SampleUser2 CallingProc
SampleUser1 ProtectedProc

PL/SQL procedure successfully completed.
```

Using the CREATE PROCEDURE Statement to Retrieve Information

Create a procedure query_emp to retrieve information about an employee. Pass the employee_id 171 to the procedure and retrieve the last_name and salary into two OUT parameters.

```
Command> CREATE OR REPLACE PROCEDURE query_emp
          (p_id IN employees.employee_id%TYPE,
           p_name  OUT employees.last_name%TYPE,
           p_salary OUT employees.salary%TYPE) IS
         BEGIN
          SELECT last_name, salary INTO p_name, p_salary
          FROM employees
          WHERE employee_id = p_id;
         END query_emp;
         /

Procedure created.
```

CREATE PROFILE

The CREATE PROFILE statement creates a profile, which is a set of limits on the database resources. If you assign a profile to a user, that user cannot exceed the limits specified in the profile.

Required Privilege

ADMIN

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE PROFILE profile LIMIT password_parameters

password_parameters::=
[FAILED_LOGIN_ATTEMPTS password_parameter_options]
[PASSWORD_LIFE_TIME password_parameter_options]
[PASSWORD_REUSE_TIME password_parameter_options]
[PASSWORD_REUSE_MAX password_parameter_options]
[PASSWORD_LOCK_TIME password_parameter_options]
[PASSWORD_GRACE_TIME password_parameter_options]
```

[{PASSWORD_COMPLEXITY_CHECKER|PASSWORD_VERIFY_FUNCTION} *password_checker_options*]

password_parameter_options::=
UNLIMITED|DEFAULT|constant

password_checker_options::=
function|NULL|DEFAULT

function::
TT_VERIFY_FUNCTION|TT_STRONG_VERIFY_FUNCTION|TT_STIG_VERIFY_FUNCTION

Parameters

Parameter	Description
profile	Name of the profile.
LIMIT *password_parameters*	The LIMIT clause sets the limits for the password parameters. The LIMIT keyword is required. The password parameters consist of the name of the password parameter and the value (or limit) for the password parameter. This includes the password complexity checker functions. All the parameters (with the exception of FAILED_LOGIN_ATTEMPTS and PASSWORD_REUSE_MAX) set lengths of time and are interpreted in number of days. You can use a decimal value (for example, you can use .0833 to denote approximately one hour). The minimum value is 1 second. The maximum value is 106,751,991 days. The constant value must be expressed in days. For example, to set a value of 5 minutes, specify the constant value of 0.0034722222222222 (5/1440 days). For FAILED_LOGIN_ATTEMPTS and PASSWORD_REUSE_MAX, you must specify an integer. If you do not specify a password parameter after the LIMIT clause, the limit for that password parameter is based on the limit defined in the DEFAULT profile. In addition, if you only specify the LIMIT keyword with no additional parameters, the limits for the profile are based on the limits of the DEFAULT profile.
FAILED_LOGIN_ATTEMPTS	Specifies the number of consecutive failed attempts to connect to the database by a user before that user's account is locked.
PASSWORD_LIFE_TIME	Specifies the number of days that a user can use the same password for authentication. If you also set a value for PASSWORD_GRACE_TIME, then the password expires if it is not changed within the grace period. In such a situation, future connections to the database are rejected.

Parameter	Description
PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX	These two parameters must be used together. • PASSWORD_REUSE_TIME specifies the number of days that must pass before a user can reuse a password. For example, if you specify a value of 30, then after 30 days the user can reuse a previous password. • PASSWORD_REUSE_MAX specifies the number of password changes that are required before the current password can be reused. You must specify a value for both parameters for them to have any effect. Specifically: • If you specify a value for both parameters: A user cannot reuse a password until the password has been changed the number of times specified for PASSWORD_REUSE_MAX during the number of days specified for PASSWORD_REUSE_TIME. For example, if you specify a value of 30 for PASSWORD_REUSE_TIME and a value of 10 for PASSWORD_REUSE_MAX, then the user can reuse the password after 30 days if the password has been changed 10 times. • If you specify a value for one parameter and specify a value of UNLIMITED for the second parameter, then the user can never reuse a password. • If you specify a value of UNLIMITED for both parameters, then TimesTen ignores both values, indicating that the password can be reused.
PASSWORD_LOCK_TIME	Specifies the number of days the user account is locked after the specified number of consecutive failed connection attempts.
PASSWORD_GRACE_TIME	Specifies the number of days after the grace period begins during which TimesTen issues a warning, but allows the connection to the database. If the password is not changed during the grace period, the password expires. This parameter is associated with the PASSWORD_LIFE_TIME parameter.
UNLIMITED	Indicates that there is no limit for the password parameter. If you specify UNLIMITED, it must follow the password parameter. For example, FAILED_LOGIN_ATTEMPTS UNLIMITED.
DEFAULT	Indicates that you want to omit a limit for the password parameter in this profile. A user that is assigned this profile is subject to the limit defined in the DEFAULT profile for this password parameter. If you specify DEFAULT, it must follow the password parameter. For example, FAILED_LOGIN_ATTEMPTS DEFAULT.
constant	Indicates the value of the password parameter if you do not specify UNLIMITED or DEFAULT. If specified, it must follow the password parameter. For example, FAILED_LOGIN_ATTEMPTS 3.

Parameter	Description
{PASSWORD_COMPLEXITY_CHECKER|PASSWORD_VERIFY_FUNCTION} {*function*|NULL|DEFAULT}	Indicates if password verification is done on passwords and, if so, the function used for verification. You can specify either the PASSWORD_COMPLEXITY_CHECKER or the PASSWORD_VERIFY_FUNCTION password parameter. They are synonymous. *function* refers to one of the three supported password complexity checker functions. Specify one of these functions to direct TimesTen to perform password verification. Valid values: • TT_VERIFY_FUNCTION • TT_STRONG_VERIFY_FUNCTION • TT_STIG_VERIFY_FUNCTION NULL indicates that there is not a password verification function assigned for the profile. DEFAULT indicates that the user is subject to the limits defined by the DEFAULT profile. The DEFAULT profile initially has a value of NULL. If you do not specify the PASSWORD_COMPLEXITY_CHECKER password parameter, the value defaults to the limits defined for the DEFAULT profile.

Description: PROFILE Statement

- Use the CREATE PROFILE statement to create a profile for the password resources, which is a set of limits for the password parameters. If you assign the profile to a user, the user cannot exceed the limits specified for the profile. If you do not assign a profile to a user, TimesTen assigns the DEFAULT profile. See Password Management in the *Oracle TimesTen In-Memory Database Security Guide* for more information on password management and profiles.
- To specify the password parameter limits for a user, do the following:
  - Use the CREATE PROFILE statement to create a profile that defines the password parameter limits.
  - Use the CREATE USER or ALTER USER statement to assign the profile to the user.
- There is a DEFAULT profile that defines a limit for each of the password parameters. This profile initially defines UNLIMITED for these parameters (which indicates that no limit has been set for the parameter). The exceptions are:
  - FAILED_LOGIN_ATTEMPTS: Set to 10.
  - PASSWORD_LOCK_TIME: Set to 0.0034722222222222 days (equal to 5 minutes, 5/1440 days)
  - PASSWORD_COMPLEXITY_CHECKER: Set to NULL.

  You can change these limits by using the ALTER PROFILE statement and specifying "DEFAULT" for the profile name. (Note that DEFAULT must be enclosed in double quotation marks.) See ALTER PROFILE for information.
- If a user is not assigned a profile, the user is subject to the limits defined in the DEFAULT profile. If a user is assigned a profile and that profile omits a limit on the password parameter or specifies DEFAULT for the password parameter, then the user is subject to the limits on those password parameters as defined by the DEFAULT profile.

- The instance administrator is assigned a system profile. You cannot alter or drop the profile of an instance administrator.

About Password Complexity Checker Verification

Password complexity checker verification ensures the password for a user is complex enough to deter intruders who try to guess passwords. When you specify a password complexity checker function in the CREATE PROFILE statement, and then assign this profile to a user, the user must create a password that meets the requirements defined for the password complexity checker function. These requirements depend on the specific password complexity checker function that you specify.

TimesTen provides the TT_VERIFY_FUNCTION, the TT_STRONG_VERIFY_FUNCTION, and the TT_STIG_VERIFY_FUNCTION password complexity checker functions to manage the complexity of the passwords. These functions are stored in the SYS schema.

The characters of interest for the password complexity checker functions are letters, digits, and special characters:

- **letter**: Uppercase and lowercase letters
- **digit**: 0-9 numbers
- **special**: A character that is neither a letter nor a digit.

```
`~!@#$%^&*()_-+={}[]\/<>,.?':|(space)
```

> **Note**
>
> – If you use one or more of the special characters, the entire password must be enclosed in double quotation marks ("). The exceptions are the # and the @ special characters. (A password that contains the # or the @ does not need to be enclosed in double quotation marks.)
>
> – The password cannot contain a semicolon (;) or a double quotation mark (").
>
> – The password must begin with a letter unless you enclose the entire password in double quotation marks.

You cannot define your own function for password complexity checker verification. The complexity of the password is checked when you use the IDENTIFIED BY clause in the CREATE USER or ALTER USER statements.

Here are the password complexity checker functions:

- TT_VERIFY_FUNCTION
- TT_STRONG_VERIFY_FUNCTION
- TT_STIG_VERIFY_FUNCTION

TT_VERIFY_FUNCTION

The TT_VERIFY_FUNCTION does the following password complexity checker verification:

- A password:
  - Must have at least 8 characters.

  - Of these 8 characters, must contain at least one letter, at least one digit, and at least one special character. .
- A password cannot contain:
  - Username or the username reversed
  - Database name
  - Oracle or TimesTen

> **Note**
>
> The comparisons are case insensitive.

TT_STRONG_VERIFY_FUNCTION

The TT_STRONG_VERIFY_FUNCTION performs the following password complexity checker verification:

- Must have at least 9 characters.
- Of these 9 characters, must contain at least two uppercase letters, at least two lowercase letters, at least two digits, and at least two special characters.

TT_STIG_VERIFY_FUNCTION

The TT_STIG_VERIFY_FUNCTION performs the following password complexity checker verification:

- Must have at least 15 characters.
- Of these 15 characters, must contain at least one uppercase letter, at least one lowercase letter, at least one digit, and at least one special character.

Description: Password Complexity Checker Verification

- EXECUTE privilege on TT_VERIFY_FUNCTION, TT_STRONG_VERIFY_FUNCTION, and TT_STIG_VERIFY_FUNCTION is required. TimesTen grants the EXECUTE privilege on these functions to PUBLIC by default.
- The SYSTEM and the DEFAULT profiles are assigned a value of a NULL by default. A NULL value indicates that there is no password complexity checker function for these profiles, and as such, there is no password complexity checker verification done.
- You cannot modify the SYSTEM profile to specify a password complexity checker function. Passwords for system users do not undergo password complexity checker verification.
- You can use the ALTER PROFILE statement to modify the DEFAULT profile to specify a password complexity checker function. You specify such a function in the PASSWORD_COMPLEXITY_CHECKER (or PASSWORD_VERIFY_FUNCTION) clause.
- The TT_VERIFY_FUNCTION, TT_STRONG_VERIFY_FUNCTION, and TT_STIG_VERIFY_FUNCTION functions in TimesTen are equivalent to the ORA12C_VERIFY_FUNCTION, ORA12C_STRONG_VERIFY_FUNCTION, and ORA12C_STIG_VERIFY_FUNCTION functions in Oracle Database.
- If you use the ttMigrate utility to downgrade to an earlier major release (such as the 18.1 release), the PASSWORD_COMPLEXITY_CHECKER value is set to NULL for each profile in the database.

Restrictions on the Password Complexity Checker Functions

There are restrictions on the password complexity checker functions:

- You cannot specify the SYS schema for the password complexity checker function. For example:

```
Command> CREATE PROFILE my_profile LIMIT
         PASSWORD_COMPLEXITY_CHECKER SYS.TT_VERIFY_FUNCTION;
15187: Cannot specify schema name for password complexity checker function
The command failed.
```

- You cannot define your own password complexity checker function. Use only the TT_VERIFY_FUNCTION,the TT_STRONG_VERIFY_FUNCTION, or the TT_STIG_VERIFY_FUNCTION password complexity checker functions.
- The password complexity checker verification is only done on a newly created password. You specify this new password by using the IDENTIFIED BY clause of the CREATE USER or the ALTER USER statement. TimesTen does not verify differences between an old and a new password.
- Multi-byte characters are not supported when specifying passwords. TimesTen does not validate passwords with multi-byte characters.

Examples

These examples illustrate various uses of the CREATE PROFILE statement. The examples also show how to use the supported clauses:

- Examples illustrating password complexity checker verification:
  - Specify TT_VERIFY_FUNCTION for PASSWORD_COMPLEXITY_CHECKER
  - Specify TT_STRONG_VERIFY_FUNCTION for PASSWORD_COMPLEXITY_CHECKER
  - Specify TT_STIG_VERIFY_FUNCTION for PASSWORD_COMPLEXITY_CHECKER
  - Modify PASSWORD_COMPLEXITY_CHECKER Value for SYSTEM and DEFAULT
  - Create a Profile and Attempt to Specify an Invalid Password Complexity Checker Function
- Additional examples:
  - Create a Profile and Set Limits on the Password Parameters
  - Create a Profile and Specify FAILED_LOGIN_ATTEMPTS
  - Determine the Password Parameter Values in the DEFAULT Profile
  - Specify PASSWORD_LIFE_TIME and PASSWORD_GRACE_TIME
  - Create a Profile Specifying Only the LIMIT Keyword
  - Specify UNLIMITED for PASSWORD_REUSE_TIME
  - Specify DEFAULT for PASSWORD_REUSE_TIME
  - Specify PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX

Specify TT_VERIFY_FUNCTION for PASSWORD_COMPLEXITY_CHECKER

This example first creates a profile and specifies the TT_VERIFY_FUNCTION function. The example then queries the dba_profiles system view to verify the TT_VERIFY_FUNCTION has been assigned to this profile. A user who is assigned this profile must specify a password that meets the password verification requirements for this function.

```
Command> CREATE PROFILE myprofile_pw1 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_VERIFY_FUNCTION;

Profile created.

Command> SELECT * FROM dba_profiles WHERE profile = 'MYPROFILE_PW1';
< MYPROFILE_PW1, FAILED_LOGIN_ATTEMPTS, PASSWORD, DEFAULT >
< MYPROFILE_PW1, PASSWORD_LIFE_TIME, PASSWORD, DEFAULT >
< MYPROFILE_PW1, PASSWORD_REUSE_TIME, PASSWORD, DEFAULT >
< MYPROFILE_PW1, PASSWORD_REUSE_MAX, PASSWORD, DEFAULT >
< MYPROFILE_PW1, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, TT_VERIFY_FUNCTION >
< MYPROFILE_PW1, PASSWORD_LOCK_TIME, PASSWORD, DEFAULT >
< MYPROFILE_PW1, PASSWORD_GRACE_TIME, PASSWORD, DEFAULT >
< MYPROFILE_PW1, TEMP_SPACE_PER_SESSION_MAX, MEMORY, DEFAULT >
8 rows found.
```

Create the sampleuser_pw1 user and assign the myprofile_pw1 profile to this user. Specify a password that meets the requirements of the TT_VERIFY_FUNCTION. See TT_VERIFY_FUNCTION for information on the TT_VERIFY_FUNCTION function.

```
Command> CREATE USER sampleuser_pw1
         IDENTIFIED BY "A1!XXcg3" PROFILE myprofile_pw1;

User created.
```

Attempt to create the sampleuser_pw2. Assign the myprofile_pw1 profile to the user. Specify a password that contains the username reversed in uppercase. The CREATE USER statement fails. The password cannot contain the username reversed. Note that the comparison is case insensitive.

```
Command> CREATE USER sampleuser_pw2
         IDENTIFIED BY "2WP_RESUELPMAS" PROFILE myprofile_pw1;
15186: Password complexity check for the specified password failed
15188: TT-20002: Password contains the username reversed
The command failed.
```

Specify TT_STRONG_VERIFY_FUNCTION for PASSWORD_COMPLEXITY_CHECKER

This example creates the myprofile_pw2 profile and specifies TT_STRONG_VERIFY_FUNCTION for the PASSWORD_COMPLEXITY_CHECKER password parameter. The example then creates the sampleuser_pw2 user, assigns the myprofile_pw2 profile to the user. The password meets the requirements of the TT_STRONG_VERIFY_FUNCTION function. See

TT_STRONG_VERIFY_FUNCTION for more information on the TT_STRONG_VERIFY_FUNCTION function.

```
Command> CREATE PROFILE myprofile_pw2 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STRONG_VERIFY_FUNCTION;

Profile created.
```

Create the sampleuser_pw2, assign the myprofile_pw2 profile to the user. The password meets the requirements of the TT_STRONG_VERIFY_FUNCTION function. The user is successfully created.

```
Command> CREATE USER sampleuser_pw2
         IDENTIFIED BY "!ddFF6C2?" PROFILE myprofile_pw2;

User created.
```

Specify TT_STIG_VERIFY_FUNCTION for PASSWORD_COMPLEXITY_CHECKER

This example creates the myprofile_pw3 profile and specifies TT_STIG_VERIFY_FUNCTION for the PASSWORD_COMPLEXITY_CHECKER password parameter. The example then creates the sampleuser_pw3 user, assigns the myprofile_pw3 profile to the user. The password meets the requirements of the TT_STIG_VERIFY_FUNCTION function. See TT_STIG_VERIFY_FUNCTION for more information on the TT_STIG_VERIFY_FUNCTION function.

```
Command> CREATE PROFILE myprofile_pw3 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STIG_VERIFY_FUNCTION;

Profile created.
```

Create the sampleuser_pw3, assign the myprofile_pw3 profile to the user. The password meets the requirements of the TT_STIG_VERIFY_FUNCTION function. The user is successfully created.

```
Command> CREATE USER sampleuser_pw3
         IDENTIFIED BY "!ddBBKKUYT165>m" PROFILE myprofile_pw3;

User created.
```

Modify PASSWORD_COMPLEXITY_CHECKER Value for SYSTEM and DEFAULT

This example queries the dba_profiles system view to check the value of the PASSWORD_COMPLEXITY_CHECKER password parameter for the SYSTEM and the DEFAULT profiles. The value is NULL by default.

```
Command> SELECT * FROM dba_profiles WHERE
         resource_name='PASSWORD_COMPLEXITY_CHECKER' AND
         profile IN ('DEFAULT','SYSTEM');
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
< SYSTEM, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
2 rows found.
```

Attempt to modify the PASSWORD_COMPLEXITY_CHECKER password parameter for the SYSTEM profile. An error results as this password parameter cannot be modified.

```
Command> ALTER PROFILE SYSTEM LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STRONG_VERIFY_FUNCTION;
15176: Profile SYSTEM cannot be altered
The command failed.
```

Attempt to modify the PASSWORD_COMPLEXITY_CHECKER password parameter for the DEFAULT profile. The modification is successful. A user who is assigned the DEFAULT profile, or is not assigned a profile, must specify a password that meets the password verification requirements for the TT_STRONG_VERIFY_FUNCTION function.

```
Command> ALTER PROFILE "DEFAULT" LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STRONG_VERIFY_FUNCTION;

Profile altered
```

Query the dba_profiles view to verify the TT_STRONG_VERIFY_FUNCTION has been assigned to the DEFAULT profile.

```
Command> SELECT * FROM dba_profiles WHERE
         resource_name='PASSWORD_COMPLEXITY_CHECKER' AND
         profile = 'DEFAULT';
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, TT_STRONG_VERIFY_FUNCTION >
1 row found.
```

Create a Profile and Attempt to Specify an Invalid Password Complexity Checker Function

This example specifies an invalid password complexity checker function for the PASSWORD_COMPLEXITY_CHECKER clause. Even though this function resides in the SYS schema, an error results, as you can only specify one of the three supported password complexity checker functions.

```
Command> CREATE PROFILE myprofile1 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_COMPLEXITY_CHECK;
 8529: Invalid password complexity checker function TT_COMPLEXITY_CHECK
The command failed.
```

Create a Profile and Set Limits on the Password Parameters

This example creates the profile1 profile and sets various limits on the password parameters. It then queries the dba_profiles system view to verify the limits.

```
Command> CREATE PROFILE profile1 LIMIT
         FAILED_LOGIN_ATTEMPTS 5
         PASSWORD_LIFE_TIME 60
         PASSWORD_REUSE_TIME 60
         PASSWORD_REUSE_MAX 5
         PASSWORD_LOCK_TIME 1
         PASSWORD_GRACE_TIME 10;

Profile created.
```

Query the dba_profiles system view to verify the limits. Note that since the PASSWORD_COMPLEXITY_CHECKER password parameter was not specified in the CREATE PROFILE statement, the value of PASSWORD_COMPLEXITY_CHECKER is DEFAULT (the value comes from the value that is in the DEFAULT profile).

```
Command> SELECT * FROM dba_profiles WHERE profile = 'PROFILE1' AND
         resource_type='PASSWORD';
< PROFILE1, FAILED_LOGIN_ATTEMPTS, PASSWORD, 5 >
< PROFILE1, PASSWORD_LIFE_TIME, PASSWORD, 60 >
< PROFILE1, PASSWORD_REUSE_TIME, PASSWORD, 60 >
< PROFILE1, PASSWORD_REUSE_MAX, PASSWORD, 5 >
< PROFILE1, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, DEFAULT >
< PROFILE1, PASSWORD_LOCK_TIME, PASSWORD, 1 >
< PROFILE1, PASSWORD_GRACE_TIME, PASSWORD, 10 >
7 rows found.
```

Create a Profile and Specify FAILED_LOGIN_ATTEMPTS

This example creates the profile2 profile and specifies a value of 1 for FAILED_LOGIN_ATTEMPTS. The example then creates the user2 user and assigns user2 the profile2 profile. The user2 user attempts to connect to the database, but specifies an invalid password. The connection fails. After five minutes, the user2 user attempts to reconnect to the database. The connection succeeds due to the 0.0034722222222222 (equal to 5 minutes) value for PASSWORD_LOCK_TIME (specified in the DEFAULT profile).

```
Command> CREATE PROFILE profile2 LIMIT FAILED_LOGIN_ATTEMPTS 1;

Profile created.

Command> CREATE USER user2 IDENTIFIED BY user2 PROFILE profile2;

User created.
```

Grant admin privilege to user2.

```
Command> GRANT ADMIN TO user2;
```

Attempt to connect to the database. The connection fails due to an invalid password specified in the connection string.

```
Command> connect adding "UID=user2;PWD=user3" as user2;
 7001: User authentication failed
The command failed.
```

Attempt to connect again specifying the correct password in the connection string. The connection fails due to:

- One previous failed connection attempt
- An attempt to connect to the database before the five minute password lock time.

```
none: Command> use database1
database1: Command> connect adding "UID=user2;PWD=user2" as user2;
15179: the account is locked
The command failed.
```

After five minutes, attempt to connect to the database again. The connection succeeds.

```
none: Command> use database1
database1: Command> connect adding "UID=user2;PWD=user2" as user2;
Connection successful: DSN=database1;UID=user2;DataStore=/scratch/database1;
```

```
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Determine the Password Parameter Values in the DEFAULT Profile

This example queries the dba_profiles system view to determine the password parameter values for the DEFAULT profile.

```
Command> SELECT * FROM dba_profiles WHERE profile = 'DEFAULT' AND
        resource_type='PASSWORD';
< DEFAULT, FAILED_LOGIN_ATTEMPTS, PASSWORD, 10 >
< DEFAULT, PASSWORD_LIFE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_MAX, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
< DEFAULT, PASSWORD_LOCK_TIME, PASSWORD, .0034 >
< DEFAULT, PASSWORD_GRACE_TIME, PASSWORD, UNLIMITED >
7 rows found.
```

Specify PASSWORD_LIFE_TIME and PASSWORD_GRACE_TIME

This example creates the profile4 profile and specifies a value of 0.0034722222222222 (equal to 5 minutes) for the PASSWORD_LIFE_TIME password parameter and a value of 0.01041667 (equal to 15 minutes) for the PASSWORD_GRACE_TIME password parameter. It then creates the user4 user and assigns the profile4 profile to user4. The example continues with attempts to connect to the database as user4.

```
Command> CREATE PROFILE profile4 LIMIT
        PASSWORD_LIFE_TIME 0.0034722222222222
        PASSWORD_GRACE_TIME 0.01041667;

Profile created.
```

Query the dba_profiles system view to verify the values for the password parameters.

```
Command> SELECT * FROM dba_profiles WHERE profile = 'PROFILE4' AND
        resource_type='PASSWORD';
< PROFILE2, FAILED_LOGIN_ATTEMPTS, PASSWORD, DEFAULT >
< PROFILE2, PASSWORD_LIFE_TIME, PASSWORD, .0034 >
< PROFILE2, PASSWORD_REUSE_TIME, PASSWORD, DEFAULT >
< PROFILE2, PASSWORD_REUSE_MAX, PASSWORD, DEFAULT >
< PROFILE2, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, DEFAULT >
< PROFILE2, PASSWORD_LOCK_TIME, PASSWORD, DEFAULT >
< PROFILE2, PASSWORD_GRACE_TIME, PASSWORD, .0104 >
7 rows found.
```

Create the user4 user and assign user4 the profile4 profile. Grant the CONNECT privilege to user4.

```
Command> CREATE USER user4 IDENTIFIED BY user4 PROFILE profile4;

User created.

Command> GRANT CONNECT TO user4;
```

Connect to the database as user4. The connection succeeds.

```
Command> connect adding "UID=user4;PWD=user4" as user4;
Connection successful: DSN=access1;UID=user4;DataStore=/scratch/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Disconnect from the database. After 5 minutes, reconnect to the database as user4. The connection succeeds but a warning is issued. The password lifetime is 5 minutes and the password grace time is 15 minutes.

```
user4: Command> disconnect user4;
Disconnecting from user4...
none: Command> use database1
database1: Command> connect adding "UID=user4;PWD=user4" as user4;

Warning 15182: Password will expire within 0.010417 days

Connection successful: DSN=access1;UID=user4;DataStore=/scratch/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Disconnect from the database. After 15 minutes, reconnect to the database as user4. The connection fails as the password grace time of 15 minutes has ended.

```
user4: Command> disconnect user4;
Disconnecting from user4...
none: Command> use database1
database1: Command> connect adding "UID=user4;PWD=user4" as user4;
15180: the password has expired
The command failed.
```

Create a Profile Specifying Only the LIMIT Keyword

This example creates the profile5 profile and specifies just the LIMIT keyword. The example then queries the dba_profiles system view to illustrate the password parameter limits for the profile5 profile are all set to a value of DEFAULT.

```
Command> CREATE PROFILE profile5 LIMIT;

Profile created.
Command> SELECT * FROM dba_profiles WHERE profile = 'PROFILE5' AND
         resource_type='PASSWORD
< PROFILE5, FAILED_LOGIN_ATTEMPTS, PASSWORD, DEFAULT >
< PROFILE5, PASSWORD_LIFE_TIME, PASSWORD, DEFAULT >
< PROFILE5, PASSWORD_REUSE_TIME, PASSWORD, DEFAULT >
< PROFILE5, PASSWORD_REUSE_MAX, PASSWORD, DEFAULT >
< PROFILE5, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, DEFAULT >
< PROFILE5, PASSWORD_LOCK_TIME, PASSWORD, DEFAULT >
< PROFILE5, PASSWORD_GRACE_TIME, PASSWORD, DEFAULT >
7 rows found.
```

Specify UNLIMITED for PASSWORD_REUSE_TIME

This example creates the profile6 profile and specifies a PASSWORD_REUSE_TIME of UNLIMITED. The password cannot be reused.

```
Command> CREATE PROFILE profile6 LIMIT
         PASSWORD_REUSE_MAX 2
         PASSWORD_REUSE_TIME UNLIMITED;

Profile created.
```

Create the user6 user and assign user6 the profile6 profile. Change the user6 password two times. Attempt to reuse the user6 password. The attempt fails due to the PASSWORD_REUSE_TIME value of UNLIMITED.

```
Command> CREATE USER user6 IDENTIFIED BY user6 PROFILE profile6;
```

```
User created.

Command> ALTER USER user6 IDENTIFIED BY user6_test1;

User altered.

Command> ALTER USER user6 IDENTIFIED BY user6_test2;

User altered.

Command> ALTER USER user6 IDENTIFIED BY user6;
15183: Password cannot be reused
The command failed.
```

Specify DEFAULT for PASSWORD_REUSE_TIME

This example creates the profile7 profile, specifying the value of DEFAULT for the PASSWORD_REUSE_TIME password parameter and the value of 3 for the PASSWORD_REUSE_MAX password parameter. TimesTen uses the value in the DEFAULT profile for the PASSWORD_REUSE_TIME password parameter.

```
Command> CREATE PROFILE profile7 LIMIT
         PASSWORD_REUSE_TIME DEFAULT
         PASSWORD_REUSE_MAX 3;

Profile created.
```

Query the dba_profiles system view to verify the password parameter values for the profile7 profile. Note the value of DEFAULT for PASSWORD_REUSE_TIME and a value of 3 for PASSWORD_REUSE_MAX (represented in **bold**).

```
Command> SELECT * FROM dba_profiles WHERE profile = 'PROFILE7' AND
         resource_type = 'PASSWORD';
< PROFILE7, FAILED_LOGIN_ATTEMPTS, PASSWORD, DEFAULT >
< PROFILE7, PASSWORD_LIFE_TIME, PASSWORD, DEFAULT >
< PROFILE7, PASSWORD_REUSE_TIME, PASSWORD, DEFAULT >
< PROFILE7, PASSWORD_REUSE_MAX, PASSWORD, 3 >
< PROFILE7, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, DEFAULT >
< PROFILE7, PASSWORD_LOCK_TIME, PASSWORD, DEFAULT >
< PROFILE7, PASSWORD_GRACE_TIME, PASSWORD, DEFAULT >
7 rows found.
```

Query the dba_profiles system view to verify the password parameter values for the DEFAULT profile. Note the value of UNLIMITED for PASSWORD_REUSE_TIME (represented in **bold**).

```
Command> SELECT * FROM dba_profiles WHERE profile = 'DEFAULT' AND
         resource_type = 'PASSWORD';
< DEFAULT, FAILED_LOGIN_ATTEMPTS, PASSWORD, 10 >
< DEFAULT, PASSWORD_LIFE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_MAX, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
< DEFAULT, PASSWORD_LOCK_TIME, PASSWORD, .0034 >
< DEFAULT, PASSWORD_GRACE_TIME, PASSWORD, UNLIMITED >
7 rows found.
```

Create the user7 user and assign the profile7 profile to user7. Change the user7 password three times. The user7 password cannot be reused due to the value of UNLIMITED for the PASSWORD_REUSE_TIME parameter.

```
Command> CREATE USER user7 IDENTIFIED BY user7 PROFILE profile7;

User created.

Command> ALTER USER user7 IDENTIFIED BY user7_test1;

User altered.

Command> ALTER USER user7 IDENTIFIED BY user7_test2;

User altered.

Command> ALTER USER user7 IDENTIFIED BY user_test3;

User altered.

Command> ALTER USER user7 IDENTIFIED BY user7;
15183: Password cannot be reused
The command failed.
```

Specify PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX

This example creates the profile8 profile, specifying a value of 0.0020833 (equal to approximately 2 minutes) for the PASSWORD_REUSE_TIME password parameter and a value of 2 for the PASSWORD_REUSE_MAX password parameter. The example then creates the user8 user and assigns user8 the profile8 profile. The user8 password is changed two times within two minutes. Then, still within the two minutes, the original user8 password (user8_pwd) is reused. The ALTER USER operation fails. Even though the password is changed 2 times, the original password can only be reused after 0.00208333 days (equal to approximately two minutes). After two minutes, the original user8 password (user8_pwd) is reused again. The ALTER USER operation succeeds. The user's password was changed two times and more than two minutes had passed.

```
Command> CREATE PROFILE profile8 LIMIT
         PASSWORD_REUSE_TIME 0.00208333
         PASSWORD_REUSE_MAX 2;

Profile created.
```

Create the user8 user and assign user8 the profile8 profile.

```
Command> CREATE USER user8 IDENTIFIED BY user8_pwd PROFILE profile8;

User created.
```

Immediately alter the user, changing the password two times.

```
Command> ALTER USER user8 IDENTIFIED BY user8_test1;

User altered.

Command> ALTER USER user8 IDENTIFIED BY user8_test2;

User altered.
```

Within two minutes, attempt to reuse the original user8_pwd password (represented in **bold**). The ALTER USER operation fails as the original password can only be reused after two minutes.

```
Command> ALTER USER user8 IDENTIFIED BY user8_pwd;
15183: Password cannot be reused
The command failed.
```

After two minutes, attempt to reuse the original user8_pwd password (represented in **bold**). The ALTER USER operation succeeds. The original password can be reused as the password was changed two times and two minutes had expired.

```
Command> ALTER USER user8 IDENTIFIED BY user8_pwd;

User altered.
```

See Also

- ALTER PROFILE
- DROP PROFILE
- CREATE USER
- ALTER USER
- DROP USER
- GRANT
- REVOKE

CREATE REPLICATION

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The CREATE REPLICATION statement:

- Defines a classic replication scheme on a participating database.
- Installs the specified configuration in the executing database's replication system tables.
- Typically consists of one or more replication element specifications and zero or more STORE specifications.

TimesTen SQL configuration for replication also provides a programmable way to configure a classic replication scheme. 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 databases, list the replication configuration, and view replication status.

Required Privilege

ADMIN

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

Definitions

A *replication element* is an entity that TimesTen synchronizes between databases. A replication element can be a whole table or a database. A database can include most types of tables and sequences. It can include only specified tables and sequences, or include all tables except specified tables and sequences. It cannot include temporary tables or views, whether materialized or nonmaterialized.

A *replication scheme* is a set of replication elements, as well as the databases that maintain copies of these elements.

For more detailed information on SQL configuration for classic replication, see Defining a Classic Replication Scheme in the *Oracle TimesTen In-Memory Database 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 [[Owner.]TableName[,...]] |
       SEQUENCE [[Owner.]SequenceName[,...]]} [,...]]
[ STORE FullStoreName [StoreAttribute [... ]]] [...]
[ NetworkOperation[...]]
```

See CHECK CONFLICTS for *CheckConflicts* syntax.

Syntax for *ReturnServiceAttribute*:

```
{ RETURN RECEIPT [BY REQUEST] |
 RETURN TWOSAFE [BY REQUEST] |
 NO RETURN }
```

Syntax for *StoreAttribute*:

```
 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
 CONFLICT REPORTING SUSPEND AT Value
 CONFLICT REPORTING RESUME AT Value
 TABLE DEFINITION CHECKING {RELAXED|EXACT}
```

Syntax for *NetworkOperation*:

```
ROUTE MASTER FullStoreName SUBSCRIBER FullStoreName
 { { MASTERIP MasterHost | SUBSCRIBERIP SubscriberHost }
    PRIORITY Priority } [...]
```

Parameters

Parameter	Description
[*Owner*.]*ReplicationSchemeName*	Name assigned to the new classic replication scheme. Classic replication schemes should have names that are unique from all other database objects.
CheckConflicts	Check for replication conflicts when simultaneously writing to bidirectionally replicated databases. See CHECK CONFLICTS for information on *CheckConflicts*.

Parameter	Description
COMPRESS TRAFFIC {ON | OFF}	Compress replicated traffic to reduce the amount of network bandwidth. ON specifies that all replicated traffic for the database defined by STORE be compressed. OFF (the default) specifies no compression. See Compressing Replicated Traffic in *Oracle TimesTen In-Memory Database Replication Guide* for details.
CONFLICT REPORTING SUSPEND AT *Value*	Suspends conflict resolution reporting. *Value* is a non-negative integer. The default is 0 and means never suspend. Conflict reporting is suspended when the rate of conflict exceeds *Value*. If you set *Value* to 0, conflict reporting suspension is turned off. This clause is valid for table level replication.
CONFLICT REPORTING RESUME AT *Value*	Resumes conflict resolution reporting. *Value* is a non-negative integer. Conflict reporting is resumed when the rate of conflict falls below *Value*. The default is 1. This clause is valid for table level replication.
DATASTORE	Define entire database as element. This type of element can only be defined for a master database that is not configured with an element of type TABLE in the same or a different replication scheme.
{INCLUDE|EXCLUDE} {[TABLE[*Owner*.]*TableName* [,...]]| SEQUENCE [[*Owner*.]*SequenceName*[,...]]} [,...]	INCLUDE includes in the DATASTORE element only the tables or sequences listed. Use one INCLUDE clause for each object type (table or sequence). EXCLUDE includes in the DATASTORE element all tables or sequences except for those listed. Use one EXCLUDE clause for each object type (table or sequence).
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. If DISABLE RETURN is specified but RESUME RETURN is not specified, the return services remain off until the replication agent for the database has been restarted.
DURABLE COMMIT {ON|OFF}	Overrides the DurableCommits general connection attribute setting. DURABLE COMMIT ON enables durable commits regardless of whether the replication agent is running or stopped.

Parameter	Description
ELEMENT *ElementName*	The entity that TimesTen synchronizes between databases. TimesTen supports the entire database (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 characters. Each *ElementName* must be unique within a classic replication scheme. Also, you cannot define two element descriptions for the same element. See Defining Replication Elements in *Oracle TimesTen In-Memory Database Replication Guide* for details.
FAILTHRESHOLD *Value*	The number of log files that can accumulate for a subscriber database. If this value is exceeded, the subscriber is set to the Failed state.The value 0 means "No Limit." This is the default. See Setting the Transaction Log Failure Threshold in *Oracle TimesTen In-Memory Database Replication Guide*.
FullStoreName	The database, specified as one of the following: • SELF • The prefix of the database file name For example, if the database path is *directory/subdirectory/*data.ds0, then data is the database name that should be used. This is the database 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 the Network in *Oracle TimesTen In-Memory Database 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 only reissue the commit. The transaction may not be rolled back. This is the default. COMMIT: On timeout, the commit function attempts to perform a COMMIT to end the transaction locally. No more operations are possible on the same transaction. This setting can be overridden for specific transactions by calling the localAction parameter in the ttRepSyncSet procedure.
MASTER *FullStoreName*	The database on which applications update the specified element. The MASTER database sends updates to its SUBSCRIBER databases. The *FullStoreName* must be the database specified in the DataStore attribute of the DSN description.

Parameter	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 *Oracle TimesTen In-Memory Database Replication Guide*.
PORT *PortNumber*	The TCP/IP port number on which the replication agent for the database listens for connections. If not specified, the replication agent automatically allocates a port number.
PROPAGATOR *FullStoreName*	The database that receives replicated updates and passes them on to other databases. The *FullStoreName* must be the database 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*. If DISABLE RETURN is specified but RESUME RETURN is not specified, the return services remain off until the replication agent for the database has been restarted.
RETURN RECEIPT [BY REQUEST]	Enables the return receipt service, so that applications that commit a transaction to a master database 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 *Oracle TimesTen In-Memory Database Replication Guide*.
RETURN SERVICES {ON|OFF} WHEN [REPLICATION] STOPPED	Sets return services on or off when replication is disabled (stopped or paused state). OFF disables return services when replication is disabled and is the default for RETURN RECEIPT service. ON allows return services to continue to be enabled when replication is disabled and is the default for RETURN TWOSAFE service.
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master database are blocked until the transaction is committed on all subscribers. **Note**: This service can only be used in a bidirectional replication scheme where the elements are defined as DATASTORE. Specifying 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 *Oracle TimesTen In-Memory Database Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgment. The default value is 10 seconds. A value of 0 (zero) means that there is no timeout. Your application can override this timeout setting by calling the returnWait parameter in the ttRepSyncSet procedure.

Parameter	Description
SEQUENCE [*Owner*.]*SequenceName*	Define the sequence specified by [*Owner*.]*SequenceName* as element. See Defining Replication Elements in *Oracle TimesTen In-Memory Database Replication Guide* for details.
STORE *FullStoreName*	Defines the attributes for a given database. Attributes include PORT, TIMEOUT and FAILTHRESHOLD. The *FullStoreName* must be the database specified in the DataStore attribute of the DSN description.
SUBSCRIBER *FullStoreName*	A database that receives updates from the MASTER databases. The *FullStoreName* must be the database 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 *Oracle TimesTen In-Memory Database Replication Guide* for details.
TIMEOUT *Seconds*	The maximum number of seconds the replication agent waits for a response from remote replication agents. The default is 120 seconds. **Note:** For large transactions that may cause a delayed response from the remote replication agent, the agent scales the timeout based on the size of the transaction. This scaling is disabled if you set TIMEOUT to less than or equal to 60 seconds. Also see Setting Wait Timeout for Response from Remote Replication Agents in *Oracle TimesTen In-Memory Database Replication Guide*.
TRANSMIT {DURABLE | NONDURABLE}	Specifies whether to flush the master log to the file system 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 the file system before they are sent to subscribers. This setting can only be used if the specified element is a DATASTORE. This is the default for RETURN TWOSAFE transactions. TRANSMIT DURABLE specifies that records are to be flushed to the file system 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 DATASTORE Element in *Oracle TimesTen In-Memory Database Replication Guide* for more information.
TABLE DEFINITION CHECKING {EXACT|RELAXED}	Specifies type of table definition checking that occurs on the subscriber: • EXACT: The tables must be identical on master and subscriber. • RELAXED: The tables must have the same key definition, number of columns and column data types. The default is RELAXED. **Note:** If you use TABLE DEFINITION CHECKING EXACT, use ttMigrate -exactUpgrade if you migrate the database. If you use TABLE DEFINITION CHECKING RELAXED, use ttMigrate -relaxedUpgrade if you migrate the database.

Parameter	Description
ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Denotes the *NetworkOperation* clause. If specified, enables you to control the network interface that a master store uses for every outbound connection to each of its subscriber stores. Can be specified more than once. For *FullStoreName*, ON "*host*" must be specified.
MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost*	*MasterHost* and *SubscriberHost* are the IP addresses for the network interface on the master and subscriber stores. Specify in dot notation or canonical format or in colon notation for IPV6. Clause can be specified more than once.
PRIORITY *Priority*	Variable expressed as an integer from 1 to 99. Denotes the priority of the IP address. Lower integral values have higher priority. An error is returned if multiple addresses with the same priority are specified. Controls the order in which multiple IP addresses are used to establish peer connections. Required syntax of *NetworkOperation* clause. Follows MASTERIP *MasterHost* | SUBSCRIBERIP *SubscriberHost* clause.

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 clause of the CREATE REPLICATION or ALTER REPLICATION statement has the following parameters:

Parameter	Description
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.

Parameter	Description
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 databases.
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 databases must use the same UPDATE BY specification. See Enabling System Timestamp Column Maintenance and Enabling User Timestamp Column Maintenance in *Oracle TimesTen In-Memory Database Replication Guide* for more information. The 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). The 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 databases on the same host must be unique for each classic replication scheme for each TimesTen instance.
- Replication elements can only be updated (by normal application transactions) through the MASTER database. PROPAGATOR and SUBSCRIBER databases are read-only.
- If you define a classic replication scheme that permits multiple databases to update the same table, see Resolving Replication Conflicts in *Oracle TimesTen In-Memory Database Replication Guide* for recommendations on how to avoid conflicts when updating rows.
- SELF is intended for classic replication schemes where all participating databases are local. Do not use SELF for a distributed classic replication scheme in a production environment, where spelling out the host name for each database in a script enables it to be used at each participating database.

- Each attribute for a given STORE may be specified only once, or not at all.
- Specifying the PORT of a database for one classic replication scheme specifies it for all classic replication schemes. All other connection attributes are specific to the classic replication scheme specified in the command.
- For replication schemes, *DataStoreName* is always the prefix of the TimesTen database checkpoint file names. These are the files with the.ds0 and.ds1 suffixes that are saved on the file system 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 *Oracle TimesTen In-Memory Database Replication Guide*.
- If you specify a report format for an element and then drop the element, the corresponding report files are not deleted.
- Use the CONFLICT REPORTING SUSPEND AT clause to specify a high water mark threshold at which the reporting of conflict resolution is suspended.
- Use the CONFLICT REPORTING RESUME AT clause to specify a low water mark threshold where the reporting of conflict resolution is resumed. When the rate of conflict falls below the low water mark threshold, conflict resolution reporting is resumed.
- The state of whether conflict reporting is suspended or not by a replication agent does not persist across the local replication agent and the peer agent stop and restart.
- Do not use the CREATE REPLICATION statement to replicate cache groups. Only active standby pairs can replicate cache groups. See the CREATE ACTIVE STANDBY PAIR statement.

Examples

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";
```

Replicate the entire masterds database to the subscriber, subscriber1ds. The FAILTHRESHOLD specifies that a maximum of 10 log files can accumulate on masterds before it decides that subscriber1ds has failed.

```
CREATE REPLICATION repl.wholestore
  ELEMENT e DATASTORE
    MASTER masterds ON "server1"
    SUBSCRIBER subscriber1ds ON "server2"
  STORE masterds FAILTHRESHOLD 10;
```

Bidirectionally replicate the entire westds and eastds databases 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;
```

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";
```

Replicate the contents of the customerswest table from the west database to the ROUNDUP database and the customerseast table from the east database. 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;
```

Replicate the contents of the repl.tab table from the centralds database to the propds database, which propagates the changes to the backup1ds and backup2ds databases.

```
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"
              bakcup2ds ON "backupsystem2";
```

Bidirectionally replicate the contents of the repl.accounts table between the eastds and westds databases. Each database 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 database, it includes a timestamp column (tstamp). The CHECK CONFLICTS clause establishes automatic timestamp comparison to detect any update conflicts between the two databases. 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";
```

Replicate the contents of the repl.accounts table from the activeds database to the backupds database, 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 database 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 database's replication agent responds in under 100 milliseconds. Finally, the bandwidth between databases is limited, so configure replication to compress the data when it is replicated from the activeds database.

```
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;
```

Illustrate conflict reporting suspend and conflict reporting resume clauses for table level replication. Use these clauses for table level replication not database replication. Issue repschemes command to show that replication scheme is created.

```
Command> CREATE TABLE repl.accounts (tstamp BINARY (8) NOT NULL
        PRIMARY KEY, tstamp1 BINARY (8));
Command> CREATE REPLICATION repl.r2
        ELEMENT elem_accounts_1 TABLE repl.accounts
        CHECK CONFLICTS BY ROW TIMESTAMP
        COLUMN tstamp1
        UPDATE BY SYSTEM
```

```
        ON EXCEPTION ROLLBACK WORK
        MASTER westds ON "west1"
        SUBSCRIBER eastds ON "east1"
  ELEMENT elem_accounts_2 TABLE repl.accounts
  CHECK CONFLICTS BY ROW TIMESTAMP
  COLUMN tstamp1
  UPDATE BY SYSTEM
  ON EXCEPTION ROLLBACK WORK
  MASTER eastds ON "east1"
  SUBSCRIBER westds ON "west1"
  STORE westds
  CONFLICT REPORTING SUSPEND AT 20
  CONFLICT REPORTING RESUME AT 10;
Command> REPSCHEMES;

Replication Scheme REPL.R2:

  Element: ELEM_ACCOUNTS_1
  Type: Table REPL.ACCOUNTS
  Conflict Check Column: TSTAMP1
  Conflict Exception Action: Rollback Work
  Conflict Timestamp Update: System
  Conflict Report File: (none)
  Master Store: WESTDS on WEST1 Transmit Durable
  Subscriber Store: EASTDS on EAST1

  Element: ELEM_ACCOUNTS_2
  Type: Table REPL.ACCOUNTS
  Conflict Check Column: TSTAMP1
  Conflict Exception Action: Rollback Work
  Conflict Timestamp Update: System
  Conflict Report File: (none)
  Master Store: EASTDS on EAST1 Transmit Durable
  Subscriber Store: WESTDS on WEST1

  Store: EASTDS on EAST1
    Port: (auto)
    Log Fail Threshold: (none)
    Retry Timeout: 120 seconds
    Compress Traffic: Disabled

  Store: WESTDS on WEST1
    Port: (auto)
    Log Fail Threshold: (none)
    Retry Timeout: 120 seconds
    Compress Traffic: Disabled
    Conflict Reporting Suspend: 20
    Conflict Reporting Resume: 10

1 replication scheme found.
```

Example of *NetworkOperation* clause with 2 MASTERIP and SUBSCRIBERIP clauses:

```
CREATE REPLICATION r ELEMENT e DATASTORE
MASTER rep1 SUBSCRIBER rep2 RETURN RECEIPT
MASTERIP "1.1.1.1" PRIORITY 1 SUBSCRIBERIP "2.2.2.2"
    PRIORITY 1
MASTERIP "3.3.3.3" PRIORITY 2 SUBSCRIBERIP "4.4.4.4"
    PRIORITY 2;
```

Example of *NetworkOperation* clause. Use the default sending interface but a specific receiving network:

```
CREATE REPLICATION r
ELEMENT e DATASTORE
MASTER rep1 SUBSCRIBER rep2
ROUTE MASTER rep1 ON "machine1" SUBSCRIBER rep2 ON "machine2"
SUBSCRIBERIP "rep2nic2" PRIORITY 1;
```

Example of using the *NetworkOperation* clause with multiple subscribers:

```
CREATE REPLICATION r ELEMENT e DATASTORE
MASTER rep1 SUBSCRIBER rep2,rep3
ROUTE MASTER rep1 ON "machine1" SUBSCRIBER rep2 ON "machine2"
MASTERIP "1.1.1.1" PRIORITY 1 SUBSCRIBERIP "2.2.2.2"
   PRIORITY 1
ROUTE MASTER Rep1 ON "machine1" SUBSCRIBER Rep3 ON "machine2"
MASTERIP "3.3.3.3" PRIORITY 2 SUBSCRIBERIP "4.4.4.4";
```

See Also

ALTER ACTIVE STANDBY PAIR
ALTER REPLICATION
CREATE ACTIVE STANDBY PAIR
DROP ACTIVE STANDBY PAIR
DROP REPLICATION

CREATE SEQUENCE

The CREATE SEQUENCE statement creates a new sequence number generator that can subsequently be used by multiple users to generate unique integers. 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 continues to generate numbers after the minimum or maximum is reached.

Required Privilege

CREATE SEQUENCE (if owner) or CREATE ANY SEQUENCE (if not owner).

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. The BATCH clause is supported in TimesTen Scaleout only.

SQL Syntax

```
CREATE SEQUENCE [Owner.]SequenceName
     [INCREMENT BY IncrementValue]
     [MINVALUE MinimumValue]
     [MAXVALUE MaximumValue]
     [CYCLE]
     [CACHE CacheValue]
     [START WITH StartValue]
     [BATCH BatchValue]
```

Parameters

Parameter	Description
SEQUENCE [*Owner*.]*SequenceName*	Name of the sequence number generator.

Parameter	Description
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 minimum 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 are 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 minimum 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*.
BATCH *BatchValue*	Valid with TimesTen Scaleout only. Configures the range of unique sequence values that are stored at each element of the grid. The default value is 10 million.

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.
- Do not create a sequence with the same name as a view or materialized view.
- Sequences with the CYCLE attribute cannot be replicated (TimesTen Classic).
- In TimesTen Classic, in which there is a replicated environment for an active standby pair, if DDL_REPLICATION_LEVEL is 3 or greater when you execute CREATE SEQUENCE on the active database, the sequence is replicated to all databases in the replication scheme. To include the sequence in the replication scheme, set DDL_REPLICATION_ACTION to INCLUDE. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Usage with TimesTen Scaleout

- The CREATE SEQUENCE statement creates a global object. Once you create the sequence, the sequence values are retrieved from any element of the database.
- Sequence values are unique, but across elements the values might not be returned in monotonic order. Within a single element, sequence values are in monotonic order. But over time, across elements, sequence values are not returned monotonically. However, the monotonic property is guaranteed within an element.
- The batch value is the range of unique sequence values stored in the element. Each element has its own batch. An element will get a new batch when its local batch is consumed. There is one element that owns the sequence and is responsible for allocating batch sequence blocks to other elements.
- For the BATCH clause:
  - Use this clause to specify the range of sequence values that are stored on each element of the grid.
  - The default is 10 million.
  - *BatchValue* must be greater than or equal to *CacheValue*.
  - The maximum value for BatchValue is dependent on the maximum value of the signed integer for the platform.
- Each element in a replica set has its own batch.
- An element's batch sequence values are recoverable. Cache values are not recoverable.

See Using Sequences in *Oracle TimesTen In-Memory Database Scaleout User's Guide* for detailed information and examples.

Using CURRVAL and NEXTVAL in TimesTen Scaleout

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.

If you execute 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.

NEXTVAL and CURRVAL can be used in the following.

- The *SelectList* of a SELECT statement, but not the *SelectList* of a subquery
- The *SelectList* of an INSERT...SELECT statement
- The SET clause of an UPDATE statement

See Using Sequences in *Oracle TimesTen In-Memory Database Scaleout User's Guide* for information on the usage of CURRVAL and NEXTVAL in a grid and for examples.

Using CURRVAL and NEXTVAL in TimesTen Classic

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.

The current value of a sequence is a connection-specific value. If there are two concurrent connections to the same database, each connection has its own CURRVAL of the same sequence set to its last NEXTVAL reference. 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. In the case of recovery, sequences are not rolled back. It is possible that the range of values of a sequence can have gaps; however, each sequence value is still unique.

If you execute 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.

> **Note**
>
> NEXTVAL cannot be used in a query on a standby node of an active standby pair.

NEXTVAL and CURRVAL can be used in the following.

- The *SelectList* of a SELECT statement, but not the *SelectList* of a subquery
- The *SelectList* of an INSERT...SELECT statement
- The SET clause of an UPDATE statement

Examples: TimesTen Scaleout

For detailed examples, see Using Sequences in the *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

Syntax example:

```
Command> CREATE SEQUENCE mysequence BATCH 100;
Command> describe mysequence;

Sequence SAMPLEUSER.MYSEQUENCE:
  Minimum Value: 1
  Maximum Value: 9223372036854775807
  Current Value: 1
  Increment:     1
  Cache:         20
  Cycle:         Off
  Batch:         100

1 sequence found.
```

Examples: TimesTen Classic

Create a sequence.

```
CREATE SEQUENCE mysequence INCREMENT BY 1 MINVALUE 2
    MAXVALUE 1000;
```

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; */
```

To create a sequence and generate a sequence number:

```
CREATE SEQUENCE seq INCREMENT BY 1;
INSERT INTO student VALUES (seq.NEXTVAL, 'Sally');
```

To use a sequence in an UPDATE SET clause:

```
UPDATE student SET studentno = seq.NEXTVAL WHERE name = 'Sally';
```

To use a sequence in a query:

```
SELECT seq.CURRVAL FROM student;
```

See Also

ALTER SEQUENCE
DROP SEQUENCE

CREATE SYNONYM

The CREATE SYNONYM statement creates a public or private synonym for a database object. A synonym is an alias for a database object. The object can be a table, view, synonym, sequence, PL/SQL stored procedure, PL/SQL function, PL/SQL package, materialized view or cache group.

A *private* synonym is owned by a specific user and exists in that user's schema. A private synonym is accessible to users other than the owner only if those users have appropriate privileges on the underlying object and specify the schema along with the synonym name.

A *public* synonym is accessible to all users as long as the user has appropriate privileges on the underlying object.

CREATE SYNONYM is a DDL statement.

Synonyms can be used in these SQL statements:

- DML statements: SELECT, DELETE, INSERT, UPDATE, MERGE
- Some DDL statements: GRANT, REVOKE, CREATE TABLE ... AS SELECT, CREATE VIEW ... AS SELECT, CREATE INDEX, DROP INDEX
- Some cache group statements: LOAD CACHE GROUP, UNLOAD CACHE GROUP, REFRESH CACHE GROUP, FLUSH CACHE GROUP

Required Privilege

CREATE SYNONYM (if owner) or CREATE ANY SYNONYM (if not owner) to create a private synonym.

CREATE PUBLIC SYNONYM to create a public synonym.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

CREATE [OR REPLACE] [PUBLIC] SYNONYM [*Owner1*.]*synonym* FOR [*Owner2*.]*object*

Parameters

Parameter	Description
[OR REPLACE]	Specify OR REPLACE to recreate the synonym if it already exists. Use this clause to change the definition of an existing synonym without first dropping it.
[PUBLIC]	Specify PUBLIC to create a public synonym. Public synonyms are accessible to all users, but each user must have appropriate privileges on the underlying object in order to use the synonym. When resolving references to an object, TimesTen uses a public synonym only if the object is not prefaced by a schema name.
[*Owner1*.]*synonym*	Specify the owner of the synonym. You cannot specify an owner if you have specified PUBLIC. If you omit both PUBLIC and *Owner1*, TimesTen creates the synonym in your own schema. Specify the name for the synonym, which is limited to 30 bytes.
[*Owner2*.]*object*	Specify the owner in which the object resides. Specify the object name for which you are creating a synonym. If you do not qualify *object* with *Owner2*, the object is in your own schema. The *Owner2* and *object* do not need to exist when the synonym is created.

Description

- The schema object does not need to exist when its synonym is created.
- Do not create a public synonym with the same name as a TimesTen built-in procedure.
- In order to use the synonym, appropriate privileges must be granted to a user for the object aliased by the synonym before using the synonym.
- A private synonym cannot have the same name as tables, views, sequences, PLSQL packages, functions, procedures, and cache groups that are in the same schema as the private synonym.
- A public synonym may have the same name as a private synonym or an object name.
- If the PassThrough attribute is set so that a query needs to executed in the Oracle database, the query is sent to the Oracle database without any changes. If the query uses a synonym for a table in a cache group, then a synonym with the same name must be defined for the corresponding Oracle database table for the query to be successful.
- When an object name is used in the DML and DDL statements in which a synonym can be used, the object name is resolved as follows:

1. Search for a match within the current schema. If no match is found, then:
2. Search for a match with a public synonym name. If no match is found, then:
3. Search for a match in the SYS schema. If no match is found, then:
4. The object does not exist.

   TimesTen creates a public synonym for some objects in the SYS schema. The name of the public synonym is the same as the object name. Thus steps 2 and 3 in the object name resolution can be switched without changing the results of the search.

- In a replicated environment for an active standby pair, if DDL_REPLICATION_LEVEL is 2 or greater when you execute CREATE SYNONYM on the active database, the synonym is replicated to all databases in the replication scheme. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Examples

As user ttuser, create a synonym for the jobs table. Verify that you can retrieve the information using the synonym. Display the contents of the SYS.USER_SYNONYMS system view.

```
Command> CREATE SYNONYM synjobs FOR jobs;
Synonym created.

Command> SELECT FIRST 2 * FROM jobs;
< AC_ACCOUNT, Public Accountant, 4200, 9000 >
< AC_MGR, Accounting Manager, 8200, 16000 >
2 rows found.
Command> SELECT FIRST 2 * FROM synjobs;
< AC_ACCOUNT, Public Accountant, 4200, 9000 >
< AC_MGR, Accounting Manager, 8200, 16000 >
2 rows found.

Command> SELECT * FROM sys.user_synonyms;
< SYNJOBS, TTUSER, JOBS, <NULL> >
1 row found.
```

Create a public synonym for the employees table.

```
Command> CREATE PUBLIC SYNONYM pubemp FOR employees;
Synonym created.
```

Verify that pubemp is listed as a public synonym in the SYS.ALL_SYNONYMS system view.

```
Command> SELECT * FROM sys.all_synonyms;
< PUBLIC, TABLES, SYS, TABLES, <NULL> >
...
< TTUSER, SYNJOBS, TTUSER, JOBS, <NULL> >
< PUBLIC, PUBEMP, TTUSER, EMPLOYEES, <NULL> >
57 rows found.
```

Create a synonym for the tab table in the terry schema. Describe the synonym.

```
Command> CREATE SYNONYM syntab FOR terry.tab;
Synonym created.
Command> DESCRIBE syntab;

Synonym TTUSER.SYNTAB:
  For Table TERRY.TAB
  Columns:
    COL1                    VARCHAR2 (10) INLINE
```

```
  COL2                  VARCHAR2 (10) INLINE

1 Synonyms found.
```

Redefine the synjobs synonym to be an alias for the employees table by using the OR REPLACE clause. Describe synjobs.

```
Command> CREATE OR REPLACE synjobs FOR employees;
Synonym created.

Command> DESCRIBE synjobs;

Synonym TTUSER.SYNJOBS:
  For Table TTUSER.EMPLOYEES
  Columns:
   *EMPLOYEE_ID                NUMBER (6) NOT NULL
    FIRST_NAME                 VARCHAR2 (20) INLINE
    LAST_NAME                  VARCHAR2 (25) INLINE NOT NULL
    EMAIL                      VARCHAR2 (25) INLINE UNIQUE NOT NULL
    PHONE_NUMBER               VARCHAR2 (20) INLINE
    HIRE_DATE                  DATE NOT NULL
    JOB_ID                     VARCHAR2 (10) INLINE NOT NULL
    SALARY                     NUMBER (8,2)
    COMMISSION_PCT             NUMBER (2,2)
    MANAGER_ID                 NUMBER (6)
    DEPARTMENT_ID              NUMBER (4)

1 Synonyms found.
```

See also

DROP SYNONYM

CREATE TABLE

The CREATE TABLE statement defines a table.

The CREATE TABLE statement is supported in TimesTen Scaleout and in TimesTen Classic. However, there are differences in syntax and semantics. For simplicity, the supported syntax, parameters, description (semantics), and examples for TimesTen Scaleout and for TimesTen Classic are separated into the usage with TimesTen Scaleout and the usage with TimesTen Classic. While there is repetition in the usages, it is presented this way in order to allow you to progress from syntax to parameters to semantics to examples for each usage.

Review the required privilege section and then see:

- CREATE TABLE: Usage with TimesTen Scaleout
- CREATE TABLE: Usage with TimesTen Classic

Required Privilege

CREATE TABLE (if owner) or CREATE ANY TABLE (if not owner).

The owner of the created table must have the REFERENCES privilege on tables referenced by the REFERENCE clause.

In TimesTen Classic:

- ADMIN privilege is required if replicating a new table across an active standby pair when DDL_REPLICATION_LEVEL=2 or greater and DDL_REPLICATION_ACTION=INCLUDE.

- These attributes cause the CREATE TABLE to implicitly execute an ALTER ACTIVE STANDBY PAIR... INCLUDE TABLE statement. See ALTER SESSION for more details.

After reviewing this section, see:

- CREATE TABLE: Usage with TimesTen Scaleout
- CREATE TABLE: Usage with TimesTen Classic

CREATE TABLE: Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. Column-based compression and aging are not supported. The using index clause and the distribution clause is not supported for global temporary tables.

See:

- SQL Syntax for CREATE TABLE: TimesTen Scaleout
- Parameters for CREATE TABLE: TimesTen Scaleout
- Column Definition: TimesTen Scaleout
- Description for USING INDEX Clauses in CREATE TABLE: TimesTen Scaleout
- Additional Information for CREATE TABLE: TimesTen Scaleout
- Examples: Global and Local Indexes in TimesTen Scaleout
- Additional Examples: TimesTen Scaleout

CREATE TABLE: Usage with TimesTen Classic

See:

- SQL Syntax for CREATE TABLE: TimesTen Classic
- Parameters for CREATE TABLE: TimesTen Classic
- Column Definition: TimesTen Classic
- Description for CREATE TABLE: TimesTen Classic
- Examples: TimesTen Classic

SQL Syntax for CREATE TABLE: TimesTen Scaleout

The syntax for a persistent table:

```
CREATE TABLE [Owner.]TableName
(
   ColumnDefinition [,...]
    [PRIMARY KEY (ColumnName [,...]) [UsingIndexClause1]]|
    [[CONSTRAINT ForeignKeyName]
       FOREIGN KEY ([ColumnName] [,...])
       REFERENCES RefTableName
          [(ColumnName [,...])] [ON DELETE CASCADE]] [...]

)
[UNIQUE HASH ON (HashColumnName [,...])
   PAGES = PrimaryPages]
[DistributionClause]
[AS SelectQuery]
```

The syntax for the *UsingIndexClause1* is shown below. Note: The *CreateIndexStmt* is the TimesTen CREATE INDEX statement. See CREATE INDEX for details. You must create a unique index as this is a requirement for a primary key.

```
UsingIndexClause1::= USING INDEX {GLOBAL | LOCAL}| USING INDEX (CreateIndexStmt)
```

The syntax for the *DistributionClause*:

```
DistributionClause::= DISTRIBUTE BY HASH [(ColumnName [,...])] |
DISTRIBUTE BY REFERENCE [(ForeignKeyConstraint)] | DUPLICATE
```

> **Note**
>
> You cannot specify a PRIMARY KEY in both the *ColumnDefinition* clause and the PRIMARY KEY clause. If you are specifying the *UsingIndexClause1* clause, you must specify PRIMARY KEY and PRIMARY KEY must be specified after the *ColumnDefinition* clause. The *UsingIndexClause1* clause cannot be specified as part of the *ColumnDefinition* clause.

Syntax for global temporary table:

The *UsingIndexClause1* and the *DistributionClause* are not supported for global temporary tables. The syntax 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 for CREATE TABLE: TimesTen Scaleout

Parameter	Description
CREATE TABLE [*Owner*.]*TableName*	CREATE TABLE indicates you want to create a table. You must specify a name for the table and optionally the owner of the table.
(*ColumnDefinition*)	*ColumnDefinition* indicates the column name, data type, and so on. *ColumnDefinition* is described in Column Definition: TimesTen Scaleout. If you specify the AS *SelectQuery* clause, *ColumnDefinition* is optional.
PRIMARY KEY (*ColumnName* [,...])	The placement of the PRIMARY KEY keyword after the *ColumnDefinition* indicates the PRIMARY KEY is specified after the columns are defined. This enables you to specify more than one column for the primary key.

Parameter	Description
[*UsingIndexClause1*]	*UsingIndexClause1* is optional and is described in the next two rows of this table. You cannot specify two USING INDEX clauses in the CREATE TABLE definition. This clause enables you to define a global or local index for the PRIMARY KEY.
USING INDEX {GLOBAL|LOCAL}	Part of [*UsingIndexClause1*]. If specified, indicates if a global or local index is to be created for the primary key.
USING INDEX (*CreateIndexStmt*)	Part of the [*UsingIndexClause1*] clause. When this USING INDEX clause is specified, the (*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a 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 range 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. If you specify the AS *SelectQuery* clause, you cannot define a foreign key on the table you are creating.

Parameter	Description
[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 HASH ON	Hash index for the table. 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. If you specify the AS *SelectQuery* clause, you must define *HashColumnName* on the table you are creating.
PAGES = *PrimaryPages*	Sizes the hash index to reflect the expected number of pages in your table. To determine the value for *PrimaryPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for *PrimaryPages* (256000/256=1000). The value for *PrimaryPages* must be a positive constant and must be greater than 0. If your estimate for *PrimaryPages* is too small, performance may be 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.
AS *SelectQuery*	If specified, creates a new table from the contents of the result set of the *SelectQuery*. The rows returned by *SelectQuery* are inserted into the table. Data types and data type lengths are derived from *SelectQuery*. *SelectQuery* is a valid SELECT statement that may or may not contain a subquery. You can specify a statement level optimizer hint after the SELECT verb. See Statement Level Optimizer Hints for information on statement level optimizer hints.

Parameter	Description
DistributionClause	Supported in TimesTen Scaleout only. There are three options: • DISTRIBUTE BY HASH [(*ColumnName* [,...])] • DUPLICATE • DISTRIBUTE BY REFERENCE [(*ForeignKeyConstraint*)] The DISTRIBUTE BY HASH clause specifies a hash distribution scheme which distributes data based on the hash of the primary key or the hash of the user-defined distribution column(s). Rows are distributed across the replica sets and each row exists in a replica set. The distribution key is optional. If specified, it consists of one or more columns and these columns are used to distribute the data. The DUPLICATE clause specifies a duplicate distribution scheme which distributes identical copies of data in a table to all elements of the database. All rows of a table exist in each element. The DISTRIBUTE BY REFERENCE clause specifies a reference distribution scheme which distributes the data of a child table based on the location of the parent row defined by the foreign key constraint. A child table row exists in the same replica set as its parent table. The foreign key constraint is optional in the DISTRIBUTE BY REFERENCE clause. However, if you define more than one foreign key constraint, you must specify one of the foreign key constraints in the DISTRIBUTE BY REFERENCE clause. If you do not specify a clause, the default is DISTRIBUTE BY HASH. You must specify the *DistributionClause* before the AS *SelectQuery* clause. You cannot update the distribution key columns.

Parameter	Description
GLOBAL TEMPORARY	Specifies that the table being created is a global 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 global 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. Operations on temporary tables do generate log records. The amount of log they generate is less than for permanent tables. The *DistributionClause* is not supported. TRUNCATE TABLE is not supported with global temporary tables. Local temporary tables are not supported. No object privileges are needed to access global temporary tables. Do not specify the AS *SelectQuery* clause with global temporary tables.
ColumnName	Name of the column in a table. If the name is used in the primary key definition, it 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. If you specify the AS *SelectQuery* clause, you do not have to specify the *ColumnName*. Do not specify the data type with the AS *SelectQuery* clause.

Column Definition: TimesTen Scaleout

SQL Syntax

You can only use the keyword, ENABLE, when defining columns in the CREATE TABLE statement.

The syntax is as follows:

```
ColumnName ColumnDataType
 [DEFAULT DefaultVal]
 [[NOT] INLINE]
 [PRIMARY KEY | UNIQUE |
 NULL [UNIQUE] |
```

```
NOT NULL [ENABLE] [PRIMARY KEY | UNIQUE]
]
```

Column Definition Parameters

The column definition has the following parameters:

Parameter	Description
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. A table can have a maximum of 1000 columns. If you specify the AS *SelectQuery* clause, *ColumnName* is optional. The number of column names must match the number of columns in *SelectQuery*.
ColumnDataType	Type of data the column can contain. Some data types require that you indicate a length. See Data Types for the data types that can be specified. If you specify the AS *SelectQuery* clause, do not specify *ColumnDataType*.
DEFAULT *DefaultVal*	Indicates that if a value is not specified for the column in an INSERT statement, the default value *DefaultVal* is inserted into the column. The default value specified must have a type that is compatible with the data type of the column. A default value can be as long as the data type of the associated column allows. You cannot assign a default value for the ROWID data type or for columns in read-only cache groups. In addition, you cannot use a function within the DEFAULT clause. The following are the supported data types for *DefaultVal*: • NULL • Constant expression (an expression that is evaluated to a constant value) • SYSDATE and GETDATE (SYSDATE and GETDATE) • SYSTEM_USER (SYSTEM_USER) If the default value is one of the users, the data type of the column must be either CHAR or VARCHAR2 and the width of the column must be at least 30 characters. If you specify the AS *SelectQuery* clause, optionally, you can specify the DEFAULT clause on the table you are creating.
INLINE|NOT INLINE	By default, variable-length columns whose declared column length is greater than 128 bytes are stored out of line. Variable-length columns whose declared column length is less than or equal to 128 bytes are stored inline. The default behavior can be overridden during table creation through the use of the INLINE and NOT INLINE keywords. If you specify the AS *SelectQuery* clause, optionally, you can specify the INLINE | NOT INLINE clause on the table you are creating.
NULL	Indicates that the column can contain NULL values. If you specify the AS *SelectQuery* clause, optionally, you can specify NULL on the table you are creating. If you specify NULL, you cannot specify ENABLE.

Parameter	Description
NOT NULL [ENABLE]	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. If you specify the AS *SelectQuery* clause, optionally, you can specify NOT NULL [ENABLE] on the table you are creating. If you specify NOT NULL, you can optionally specify ENABLE. Because NOT NULL constraints are always enabled, you are not required to specify ENABLE. You can only use the keyword, ENABLE, when defining columns in the CREATE TABLE statement.
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 range index to enforce uniqueness. So 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. If you specify the AS *SelectQuery* clause, optionally, you can specify UNIQUE on the table you are creating.
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. If you specify the AS *SelectQuery* clause, optionally, you can specify PRIMARY KEY on the table you are creating.

Description for USING INDEX Clauses in CREATE TABLE: TimesTen Scaleout

You have the option of specifying an additional clause after the PRIMARY KEY clause in your CREATE TABLE definition. This clause enables you to specify a global or local index for the primary key constraint.

- The USING INDEX {GLOBAL | LOCAL} clause is one option that enables you to specify a global or local index for the primary key constraint. You must specify the GLOBAL or the LOCAL keyword. You can optionally specify the USE HASH INDEX clause after the USING INDEX {GLOBAL | LOCAL} clause if you want to define a hash index.
- The USING INDEX (*CreateIndexStmt*) clause is your other option for specifying a global or local index. The(*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must create a unique index as that is the requirement for a primary key constraint. If you use the CREATE INDEX statement to create a hash index, see CREATE INDEX for information on the CREATE INDEX statement.

> **Note**
>
> You cannot use both the USING INDEX {GLOBAL | LOCAL} and the USING INDEX (*CreateIndexStmt*) in the CREATE TABLE definition. Specify one clause or the other or specify neither.

When you create a hash index, the hash index must be sized. TimesTen provides the PAGES= clause for this purpose. Consider these options:

- If you specify the USING INDEX...*CreateIndexStmt* clause to create a HASH index, you have the option of specifying the PAGES= clause. If you do not specify the PAGES= clause, TimesTen uses PAGES=CURRENT as the default to size the hash index. (See CREATE INDEX for details on the CREATE INDEX statement.)
- If you specify the UNIQUE HASH ON clause (part of the CREATE TABLE definition), you must specify the PAGES= clause to size the hash index.
- If you specify both the USING INDEX...*CreateIndexStmt* and the UNIQUE HASH ON clause (part of the CREATE TABLE definition), TimesTen uses the value specified in the UNIQUE HASH ON...PAGES= clause to size the hash index. TimesTen also issues a warning that there was a different number of pages specified for the hash index and it is using the value specified in the UNIQUE HASH ON...PAGES= clause.

In this example, the PAGES= clause is specified in both the *CreateIndexStmt* clause (PAGES=200) and the UNIQUE HASH ON clause (PAGES=400). TimesTen issues a warning and uses PAGES=400 to size the hash index:

```
Command> CREATE TABLE mytab (col1 TT_INTEGER, col2 TT_INTEGER, PRIMARY KEY (col1, col2)
         USING INDEX (CREATE GLOBAL UNIQUE HASH INDEX myindex on mytab (col1,col2) PAGES=200))
         UNIQUE HASH ON (col1,col2) PAGES=400 DISTRIBUTE BY HASH (col1);
Warning  2252: Different number of pages specified for hash index MYINDEX in table and index definition.
Index created with pages = 400
```

Restrictions:

- The USING INDEX clause cannot be used for foreign key constraints on a table.
- The USING INDEX clause cannot be used with views.

See CREATE INDEX for information on global and local indexes and their use in TimesTen Scaleout.

Additional Information for CREATE TABLE: TimesTen Scaleout

- TimesTen Scaleout distributes data by one of three distribution schemes:
  - Hash: TimesTen Scaleout distributes data based on the hash of the primary key column(s) or one or more columns you specify in the DISTRIBUTED BY HASH clause. A given row is stored in a replica set. Rows are evenly distributed across the replica sets. Hash is the default distribution scheme as it is appropriate for most tables.
  - Reference: TimesTen Scaleout distributes data of a child table based on the location of the parent table that is identified by the foreign key. A given row of a child table is present in the same replica set as its parent table. This distribution scheme optimizes joins by distributing related data within a single replica set. You can distribute the parent table by hash or reference. The parent is called the root table if it is distributed by hash. You must define the child (foreign) key columns as NOT NULL.
  - Duplicate: TimesTen Scaleout distributes full identical copies of data to all elements of the database. All rows are present in all elements. This distribution scheme optimizes the performance of reads by storing identical data in every data instance. This distribution scheme is appropriate for tables that are relatively small, frequently read, and infrequently modified.

  See Defining the Distribution Scheme for Tables and Defining Table Distribution Schemes in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for more information.
- For tables with a hash distribution scheme:
  - The distribution key is used if specified.

  - The primary key is used if the distribution key is not specified.
  - A hidden column is used if there is no primary key or distribution key. Data is distributed randomly and evenly.

  You should specify a distribution key if there is a primary key defined on the table, but the primary key is not the best way to distribute the data. If there is no primary key, but there is a unique column, then you may want to distribute the data on this unique column. If there is no primary key and no unique column, then do not specify a distribution key. TimesTen Scaleout distributes the data on the hidden column.

- If the distribution scheme is by reference:
  - Only a single foreign key constraint can be referenced in the DISTRIBUTE BY REFERENCE clause. There may be multiple foreign key constraints in the child table, but only one can be used to determine the reference distribution.
  - A referenced foreign key constraint must be named in the constraint clause if there is more than one.
  - The foreign key constraint in the reference distribution clause must reference the primary key or a unique key of the parent table. If the parent table is the root, the referenced key must be the distribution key.
  - You can create a foreign key relationship to a non distribution key column of the parent table, but you cannot then distribute by reference based on this foreign key relationship.
  - You cannot update the foreign key column that is used in the DISTRIBUTE BY REFERENCE clause.
- If you are planning to load your tables with data, consider creating your tables without indexes. After the data is loaded, you can then create your indexes. This reduces the time it take to load the data into the tables. The exception is if you are using foreign keys and reference tables.
- You can use the CREATE TABLE...AS SELECT statement to create a new table based on the definition of the original table. Note that primary key constraints are not carried over to the new table so how the data is distributed changes if you do not define a primary key constraint on the new table.

  See Use CREATE TABLE...AS SELECT for more information.
- You cannot update the distribution key column(s) unless you update the column(s) to the same value.
- All columns participating in the primary key are NOT NULL.
- A PRIMARY KEY that is specified in the *ColumnDefinition* can only be specified for one column.
- You cannot specify a PRIMARY KEY in both the *ColumnDefinition* clause and the PRIMARY KEY clause.
- For both primary key and foreign key constraints, duplicate column names are not allowed in the constraint column list.
- You cannot update primary key column(s) unless you update the column(s) to the same value.
- There are performance considerations when you define out of line columns instead of inline columns:
  - Accessing data is slower because TimesTen does not store data contiguously with out of line columns.
  - Populating data is slower because TimesTen generates more logging operations.

  - Deleting data is slower because TimesTen performs more reclaim and logging operations.
  - Storing a column requires less overhead.
- If ON DELETE CASCADE is specified on a foreign key constraint for a child table, a user can delete rows from a parent table for which the user has the DELETE privilege without requiring explicit DELETE privilege on the child table.
- To change the ON DELETE CASCADE triggered action, drop then redefine the foreign key constraint.
- 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.
- Use the ALTER TABLE statement to change the representation of the primary key index for a table.
- If you specify the AS *SelectQuery* clause:
  - Data types and data type lengths are derived from the *SelectQuery*. Do not specify data types on the columns of the table you are creating.
  - TimesTen defines on columns in the new table NOT NULL constraints that were explicitly created on the corresponding columns of the selected table if *SelectQuery* selects the column rather than an expression containing the column.
  - NOT NULL constraints that were implicitly created by TimesTen on columns of the selected table (for example, primary keys) are carried over to the new table. You can override the NOT NULL constraint on the selected table by defining the new column as NULL. For example:

    ```
    CREATE TABLE newtable (newcol NULL) AS SELECT (col) FROM tab;
    ```
  - NOT INLINE/INLINE attributes are carried over to the new table.
  - Unique keys, foreign keys, indexes and column default values are not carried over to the new table.
  - If all expressions in *SelectQuery* are columns, rather than expressions, then you can omit the columns from the table you are creating. In this case, the name of the columns are the same as the columns in *SelectQuery*. If the *SelectQuery* contains an expression rather than a simple column reference, either specify a column alias or name the column in the CREATE TABLE statement.
  - Do not specify foreign keys on the table you are creating.
  - Do not specify the SELECT FOR UPDATE clause in *SelectQuery*.
  - The ORDER BY clause is not supported when you use the AS *SelectQuery* clause.
  - *SelectQuery* cannot contain set operators UNION, MINUS, INTERSECT.
- By default, a range 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 using a table's primary key, then choose a range index for that table by omitting the UNIQUE HASH clause.
  - If 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.

- A hash index is created with a fixed size that remains constant for the life of the table or until the hash index is resized with the ALTER TABLE statement or when the index is dropped and recreated. A smaller hash index results in more hash collisions. A larger hash index reduces collisions but can waste memory. Hash key comparison is a fast operation, so a small number of hash collisions should not cause a performance problem for TimesTen.

  To ensure that your hash index is sized correctly, your application must indicate the expected size of your table with the value of the *RowPages* parameter of the SET PAGES clause. Compute this value by dividing the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for the value of RowPages (256000/256=1000).
- At most 16 columns are allowed in a hash key.
- 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 causes cascaded deletes in the child table. This, in turn, triggers 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, the following rule is also enforced.
  - If a table is reached by a "delete" path, then all its children are also reached by a "delete" path.
- The data in a global temporary table is private to the current connection and does not need to be secured between users. Thus, global temporary tables do not require object privileges.

Examples: Global and Local Indexes in TimesTen Scaleout

These examples show various uses of the syntax for using global indexes with CREATE TABLE...PRIMARY KEY.

Create a table specifying a primary key. Use the USING INDEX GLOBAL clause to create a global range index. The index must be unique as is a requirement for a primary key.

```
Command> CREATE TABLE mytab (c TT_INTEGER, b TT_INTEGER, a TT_INTEGER,
         PRIMARY KEY (c,b) USING INDEX GLOBAL) DISTRIBUTE BY HASH (a,b);
Command> indexes mytab;

Indexes on table SAMPLEUSER.MYTAB:
  MYTAB: global unique range index on columns:
    C
    B
  1 index found.

1 index found on 1 table
Command> DROP TABLE mytab;
```

Create a table specifying a primary key. Use the USING INDEX LOCAL clause to create a local range index. The index must be unique as is a requirement for a primary key.

```
Command> CREATE TABLE mytab (c TT_INTEGER, b TT_INTEGER, a TT_INTEGER,
 PRIMARY KEY (c,b) USING INDEX LOCAL DISTRIBUTE BY HASH (a,b);
Command> indexes mytab;

Indexes on table SAMPLEUSER.MYTAB:
 MYTAB: unique range index on columns:
  C
  B
 1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab;
```

Create a table specifying a primary key. Use the USING INDEX (*CreateIndexStmt*) clause to create a global range index. The (*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must specify the UNIQUE keyword when creating the index. This is a requirement for a primary key. See CREATE INDEX for information on this statement.

```
Command> CREATE TABLE mytab (c TT_INTEGER, b TT_INTEGER, a TT_INTEGER,
      PRIMARY KEY (c,b) USING INDEX (CREATE GLOBAL UNIQUE INDEX GlobalUniqueIdx ON mytab
(c,b))) DISTRIBUTE  BY HASH (a,b);
Command> indexes mytab;

Indexes on table SAMPLEUSER.MYTAB:
 GLOBALUNIQUEIDX: global unique range index on columns:
  C
  B
 1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab;
```

Create a table specifying a primary key. Use the USING INDEX (*CreateIndexStmt*) clause to create a global range index. The (*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. The CREATE INDEX definition specifies the INCLUDE clause to include additional column(s) in the index definition. You must specify the UNIQUE keyword when creating the index. This is a requirement for a primary key. See CREATE INDEX for information on this statement.

```
Command> CREATE TABLE mytab (c TT_INTEGER, b TT_INTEGER, a TT_INTEGER,
      PRIMARY KEY (c,b) USING INDEX (CREATE GLOBAL UNIQUE INDEX GlobalUniqueIdx
      ON mytab (c,b) INCLUDE (a))) DISTRIBUTE  BY HASH (a,b);
Command> indexes mytab;

Indexes on table SAMPLEUSER.MYTAB:
 GLOBALUNIQUEIDX: global unique range index on columns:
  C
  B
  Included columns:
   A
```

```
  1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab;
```

Create a table specifying a primary key. Use the USING INDEX (*CreateIndexStmt*) clause to create a global unique hash index. The (*CreateIndexStmt*) clause indicates that you want to define the index according to the TimesTen CREATE INDEX statement. The parentheses ( ) are required. You must specify the UNIQUE keyword when creating the index. This is a requirement for a primary key. See CREATE INDEX for information on this statement.

```
Command> CREATE TABLE mytab (c TT_INTEGER, b TT_INTEGER, a TT_INTEGER,
         PRIMARY KEY (c,b) USING INDEX (CREATE GLOBAL UNIQUE HASH INDEX GlobalUniqueIdx
         ON mytab (c,b))) DISTRIBUTE  BY HASH (a,b);
Command> indexes mytab;

Indexes on table SAMPLEUSER.MYTAB:
  GLOBALUNIQUEIDX: global unique hash index on columns:
    C
    B
  1 index found.

1 index found on 1 table.
Command> DROP TABLE mytab;
```

Additional Examples: TimesTen Scaleout

These examples illustrate how to create tables with the duplicate, hash, and reference distribution schemes.

- Create the account_type Table
- Create the account_status Table
- Create the customers Table
- Create the accounts Table
- Create the transactions Table
- View the Tables
- View the Definition of the Accounts Table

These examples illustrate how to create tables with the DISTRIBUTE BY REFERENCE distribution scheme:

- DISTRIBUTE BY REFERENCE with One Foreign Key
- Table with More Than One Foreign Key
- Foreign Key Relationship Not On Distribution Key of the Parent Table
- Using First and Second Level Child Foreign Key Relationship

"Use CREATE TABLE...AS SELECT" shows how to use the CREATE TABLE...AS SELECT clause in TimesTen Scaleout.

Create the account_type Table

This example runs ttIsql to create the account_type table and use a duplicate distribution scheme to distribute the data. This table contains few rows and uses a duplicate distribution scheme to optimize reads. Copies of the data in the table are distributed to all elements of the database.

```
Command> CREATE TABLE account_type ( type CHAR(1) NOT NULL PRIMARY KEY,
     description VARCHAR2(100) NOT NULL) DUPLICATE;
```

Create the account_status Table

This example runs ttIsql to create the account_status table and use a duplicate distribution scheme. The table size is small and uses a distribution scheme to optimize reads. Copies of the data in the table are distributed to all elements of the database.

```
Command> CREATE TABLE account_status(status NUMBER(2) NOT NULL PRIMARY KEY,
     description VARCHAR2(100) NOT NULL) DUPLICATE;
```

Create the customers Table

This example runs ttIsql to create the customers table and distributes the table by hash. The data in the table is distributed to each element based on the hash of the cust_id column (the primary key).

```
Command> CREATE TABLE customers(cust_id NUMBER(10,0) NOT NULL PRIMARY KEY,
     first_name VARCHAR2(30) NOT NULL,last_name VARCHAR2(30) NOT NULL,
     addr1 VARCHAR2(64),addr2 VARCHAR2(64), zipcode VARCHAR2(5),
     member_since DATE NOT NULL)
    DISTRIBUTE BY HASH;
```

Create the accounts Table

This example runs ttIsql to create the accounts table and defines three primary/foreign key relationships. The accounts table is distributed by reference and the data is distributed based on the fk_customer foreign key constraint. This scheme optimizes the performance of joins by distributing the data in the accounts table based on the location of the corresponding value of the customers.cust_id parent column (of the fk_customer foreign key constraint). The row of a child table exists in the same replica set as the parent table. If the join is performed on the primary or foreign key, the data is stored on one element, so TimesTen Scaleout does not have to access different elements.

```
Command> CREATE TABLE accounts(account_id NUMBER(10,0) NOT NULL PRIMARY KEY,
     phone VARCHAR2(15) NOT NULL,account_type CHAR(1) NOT NULL,
     status NUMBER(2) NOT NULL,current_balance NUMBER(10,2) NOT NULL,
     prev_balance NUMBER(10,2) NOT NULL,date_created DATE NOT NULL,
     cust_id NUMBER(10,0) NOT NULL,
    CONSTRAINT fk_customer FOREIGN KEY (cust_id)
    REFERENCES customers(cust_id),CONSTRAINT fk_acct_type
    FOREIGN KEY (account_type)
    REFERENCES account_type(type),
    CONSTRAINT fk_acct_status
    FOREIGN KEY (status)
    REFERENCES account_status(status)
    )
    DISTRIBUTE BY REFERENCE (fk_customer);
```

Create the transactions Table

This example runs ttIsql to create the transactions table. The transactions table is distributed by reference and the data is distributed based on the fk_accounts foreign key constraint. This scheme optimizes the performance of joins by distributing the data in the transaction table based on the location of the corresponding value of the accounts.account_id parent column (of the fk_accounts foreign key constraint). The row of a child table exists in the same replica set as the parent table. If the join is performed on the primary or foreign key, the data is stored on one element, so TimesTen Scaleout does not have to access different elements.

The accounts parent table is also distributed by reference. This defines a two level distribute by reference distribution hierarchy.

```
Command> CREATE TABLE transactions(transaction_id NUMBER(10,0) NOT NULL,
         account_id NUMBER(10,0) NOT NULL ,
         transaction_ts TIMESTAMP NOT NULL,
         description VARCHAR2(60),
         optype CHAR(1) NOT NULL,
         amount NUMBER(6,2) NOT NULL,
        PRIMARY KEY (account_id, transaction_id, transaction_ts),
        CONSTRAINT fk_accounts FOREIGN KEY (account_id)
        REFERENCES accounts(account_id)
        )
        DISTRIBUTE BY REFERENCE (fk_accounts);
```

View the Tables

This example runs the ttIsql tables command to view the tables in the database.

```
Command> tables;
  SAMPLEUSER.ACCOUNTS
  SAMPLEUSER.ACCOUNT_STATUS
  SAMPLEUSER.ACCOUNT_TYPE
  SAMPLEUSER.CUSTOMERS
  SAMPLEUSER.TRANSACTIONS
5 tables found.
```

View the Definition of the Accounts Table

This example runs the ttIsql describe command to view the definition of the accounts table.

```
Command> describe accounts;

Table SAMPLEUSER.ACCOUNTS:
  Columns:
   *ACCOUNT_ID                     NUMBER (10) NOT NULL
    PHONE                          VARCHAR2 (15) INLINE NOT NULL
    ACCOUNT_TYPE                   CHAR (1) NOT NULL
    STATUS                         NUMBER (2) NOT NULL
    CURRENT_BALANCE                NUMBER (10,2) NOT NULL
    PREV_BALANCE                   NUMBER (10,2) NOT NULL
    DATE_CREATED                   DATE NOT NULL
    CUST_ID                        NUMBER (10) NOT NULL
  DISTRIBUTE BY REFERENCE (FK_CUSTOMER)

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

DISTRIBUTE BY REFERENCE with One Foreign Key

This example illustrates that you do not have to specify the foreign key constraint in the DISTRIBUTE BY REFERENCE clause. There is only one foreign key.

First create the Orders table and distribute by hash.

```
Command> CREATE TABLE Orders
         (OrderId TT_INTEGER NOT NULL PRIMARY KEY,
          OrderDate DATE NOT NULL,
          discount BINARY_FLOAT)
        DISTRIBUTE BY HASH;
```

Create the OrderDetails table with one foreign key constraint. There is no need to name the constraint in the distribution clause.

```
Command> CREATE TABLE OrderDetails
         (OrderId TT_INTEGER NOT NULL,
          PartId TT_INTEGER NOT NULL,
          Quantity TT_INTEGER NOT NULL,
        FOREIGN KEY (OrderId)
        REFERENCES Orders (OrderId))
        DISTRIBUTE BY REFERENCE;
```

Run the ttIsql describe command to view the tables.

```
Command> describe Orders;

Table SAMPLEUSER.ORDERS:
  Columns:
   *ORDERID                         TT_INTEGER NOT NULL
    ORDERDATE                       DATE NOT NULL
    DISCOUNT                        BINARY_FLOAT
  DISTRIBUTE BY HASH (ORDERID)

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

Command> describe OrderDetails;

Table SAMPLEUSER.ORDERDETAILS:
  Columns:
    ORDERID                         TT_INTEGER NOT NULL
    PARTID                          TT_INTEGER NOT NULL
    QUANTITY                        TT_INTEGER NOT NULL
  DISTRIBUTE BY REFERENCE

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

Table with More Than One Foreign Key

This example illustrates that if a table contains more than one foreign key constraint, the DISTRIBUTE BY REFERENCE clause must name the foreign key constraint that will be used as the reference. The customers2 table is the parent and is distributed by hash. The OrderDetails2 table contains two foreign key constraints and this table is distributed by reference on the c1_1 constraint. This constraint must be included in the DISTRIBUTED BY REFERENCE clause.

```
Command> CREATE TABLE customers2 (CustomerId TT_INTEGER NOT NULL PRIMARY KEY,
         LastOrderDate DATE NOT NULL,PromotionDiscount BINARY_FLOAT)
        DISTRIBUTE BY HASH;

Command> CREATE TABLE OrderDetails2 (OrderId TT_INTEGER NOT NULL,
         CustomerId TT_INTEGER NOT NULL, Quantity TT_INTEGER NOT NULL,
        CONSTRAINT c1_1 FOREIGN KEY (OrderId)
        REFERENCES Orders (OrderId),
        CONSTRAINT c2_2 FOREIGN KEY (CustomerId)
        REFERENCES Customers2 (CustomerId))
        DISTRIBUTE BY REFERENCE (c1_1);
```

Foreign Key Relationship Not On Distribution Key of the Parent Table

This example creates the orders2 parent table with the OrderId primary key and the CouponId unique key. The table is distributed by hash. Since no distribution key is specified, the data is distributed by hash on the OrderId primary key. The coupons child table establishes a foreign key relationship on the CouponId unique key. Since this key is not the distribution key of the orders2 parent table, TimesTen Scaleout throws an error.

```
Command> CREATE TABLE Orders2 (OrderId TT_INTEGER NOT NULL PRIMARY KEY,
         CouponId TT_INTEGER NOT NULL UNIQUE, OrderDate DATE NOT NULL,
         discount BINARY_FLOAT)
        DISTRIBUTE BY HASH;

Command> CREATE TABLE Coupons (CouponId TT_INTEGER NOT NULL,
         discount BINARY_FLOAT,
        CONSTRAINT CouponC1 FOREIGN KEY (CouponId)
        REFERENCES Orders2 (CouponId) )
        DISTRIBUTE BY REFERENCE (CouponC1);
 1067: The Parent keys for a distribute by reference table with hash distributed
parent must include the distribution keys of the parent.
The command failed.
```

Using First and Second Level Child Foreign Key Relationship

This example creates the Coupons2 parent table and distributes the data by hash. The Orders3 child table is created as a first level foreign key relationship and the parent table (Coupons2) is the root table. The OrderDetails3 child table is created as a second level foreign key relationship and the parent table (Orders3) is a reference table.

```
Command> CREATE TABLE Coupons2 (CouponId TT_INTEGER NOT NULL PRIMARY KEY,
         discount BINARY_FLOAT)
        DISTRIBUTE BY HASH;

Command> CREATE TABLE Orders3 (OrderId TT_INTEGER NOT NULL PRIMARY KEY,
         CouponId TT_INTEGER NOT NULL, OrderDate DATE NOT NULL,
         discount BINARY_FLOAT, CONSTRAINT c1_coupons FOREIGN KEY (CouponId)
        REFERENCES Coupons2 (CouponId))
        DISTRIBUTE BY REFERENCE (c1_coupons);

Command> CREATE TABLE OrderDetails3 (OrderId TT_INTEGER NOT NULL,
         PartId TT_INTEGER NOT NULL, quantity TT_INTEGER NOT NULL,
        CONSTRAINT c1_orders FOREIGN KEY (OrderId)
        REFERENCES Orders3 (OrderId))
        DISTRIBUTE BY REFERENCE (C1_orders);
```

Use CREATE TABLE...AS SELECT

This example creates the NewCustomers table based on the customers table. It defines a primary key constraint to maintain the same distribution scheme and ensure the data is distributed on the primary key.

```
Command> CREATE TABLE NewCustomers(cust_id PRIMARY KEY, first_name, last_name,
         addr1, addr2, zipcode, member_since) AS SELECT * FROM customers;
0 rows inserted.
Command> describe NewCustomers;

Table SAMPLEUSER.NEWCUSTOMERS:
  Columns:
   *CUST_ID                    NUMBER (10) NOT NULL
    FIRST_NAME                 VARCHAR2 (30) INLINE NOT NULL
    LAST_NAME                  VARCHAR2 (30) INLINE NOT NULL
```

```
  ADDR1                  VARCHAR2 (64) INLINE
  ADDR2                  VARCHAR2 (64) INLINE
  ZIPCODE                VARCHAR2 (5) INLINE
  MEMBER_SINCE           DATE NOT NULL
 DISTRIBUTE BY HASH (CUST_ID)

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

Run ttIsql describe to view the original customers table:

```
Command> describe Customers;

Table SAMPLEUSER.CUSTOMERS:
 Columns:
  *CUST_ID               NUMBER (10) NOT NULL
  FIRST_NAME             VARCHAR2 (30) INLINE NOT NULL
  LAST_NAME              VARCHAR2 (30) INLINE NOT NULL
  ADDR1                  VARCHAR2 (64) INLINE
  ADDR2                  VARCHAR2 (64) INLINE
  ZIPCODE                VARCHAR2 (5) INLINE
  MEMBER_SINCE           DATE NOT NULL
 DISTRIBUTE BY HASH (CUST_ID)

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

SQL Syntax for CREATE TABLE: TimesTen Classic

You cannot specify a PRIMARY KEY in both the *ColumnDefinition* clause and the PRIMARY KEY clause.

The syntax for a persistent table:

```
CREATE TABLE [Owner.]TableName
(
  {{ColumnDefinition} [,...]
   [PRIMARY KEY (ColumnName [,...]) |
   [[CONSTRAINT ForeignKeyName]
     FOREIGN KEY ([ColumnName] [,...])
     REFERENCES RefTableName
       [(ColumnName [,...])] [ON DELETE CASCADE]] [...]
  }
)
[ColumnBasedCompression]
[UNIQUE HASH ON (HashColumnName [,...])
  PAGES = PrimaryPages]
[AGING {LRU|
      USE ColumnName
         LIFETIME Num1 {SECOND[S] | MINUTE[S] | HOUR[S] |DAY[S]}
           [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
     }[ON|OFF]
]
[AS SelectQuery]
```

The syntax for a global 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 for CREATE TABLE: TimesTen Classic

Parameter	Description
[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 Database table owner names must always match TimesTen table owner names. See Basic Names for rules on defining names.

Parameter	Description
GLOBAL TEMPORARY	Specifies that the table being created is a global 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 global 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. When DDL_REPLICATION_LEVEL is 2 or greater, the creation of a global temporary table is replicated in an active standby pair, but the global temporary table is not included in the replication scheme. Temporary tables are automatically excluded from active standby pairs or when the DATASTORE element has been specified. 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. Truncate table is not supported with global temporary tables. Local temporary tables are not supported. No object privileges are needed to access global temporary tables. Do not specify the AS *SelectQuery* clause with global temporary tables.
ColumnDefinition	An individual column in a table. Each table must have at least one column. If you specify the AS *SelectQuery* clause, *ColumnDefinition* is optional.

Parameter	Description
ColumnName	Name of the column in a table. Is used in various clauses of the CREATE TABLE statement. If the name is used in the primary key definition, it 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. If you specify the AS *SelectQuery* clause, you do not have to specify the *ColumnName*. Do not specify the data type with the AS *SelectQuery* clause.
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. You 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.

Parameter	Description
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 range 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. If you specify the AS *SelectQuery* clause, you cannot define a foreign key on the table you are creating.
[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.
ColumnBasedCompression	Defines compression at the column level, which stores data more efficiently. Eliminates redundant storage of duplicate values within columns and improves the performance of SQL queries that perform full table scans. See Column-Based Compression of Tables (TimesTen Classic) for details.
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. This parameter is used for equality predicates. UNIQUE HASH ON requires that a primary key be defined.

Parameter	Description
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. If you specify the AS *SelectQuery* clause, you must define *HashColumnName* on the table you are creating.
PAGES = *PrimaryPages*	Sizes the hash index to reflect the expected number of pages in your table. To determine the value for *PrimaryPages*, divide the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for *PrimaryPages* (256000/256=1000). The value for *PrimaryPages* must be a positive constant and must be greater than 0. If your estimate for *PrimaryPages* is too small, performance may be 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 specified, defines the LRU aging policy for the table. The LRU aging policy defines the type of aging (least recently used (LRU)), the aging state (ON or OFF) and the LRU aging attributes. Set the aging state to either ON or OFF. ON indicates that the aging state is enabled and aging is done automatically. OFF indicates that the aging state is disabled and aging is not done automatically. In both cases, the aging policy is defined. The default is ON. LRU attributes are defined by calling the ttAgingLRUConfig and/or the ttAgingTableLRUConfig built-in procedures. LRU attributes are not defined at the SQL level. See ttAgingLRUConfig and ttAgingTableLRUConfig in the *Oracle TimesTen In-Memory Database Reference* and Implementing an Aging Policy in Your Tables in the *Oracle TimesTen In-Memory Database Operations Guide* for more information.

Parameter	Description
[AGING USE *ColumnName*...[ON|OFF]]	If specified, defines the time-based aging policy for the table. The time-based aging policy defines the type of aging (time-based), the aging state (ON or OFF) and the time-based aging attributes. Set the aging state to either ON or OFF. ON indicates that the aging state is enabled and aging is done automatically. OFF indicates that the aging state is disabled and aging is not done automatically. In both cases, the aging policy is defined. The default is ON. Time-based aging attributes are defined at the SQL level and are specified by the LIFETIME and CYCLE clauses. Specify *ColumnName* as the name of the column used for time-based aging. Define the column as NOT NULL and of data type TIMESTAMP or DATE. The value of this column is subtracted from SYSDATE, truncated using the specified unit (second, minute, hour, day) and then compared to the LIFETIME value. If the result is greater than the LIFETIME value, then the row is a candidate for aging. The values of the column that you use for aging 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 (the column cannot be NULL). You can define your aging column with a data type of TT_TIMESTAMP or TT_DATE. If you choose data type TT_DATE, then you must specify the LIFETIME unit as days. If you specify the AS *SelectQuery* clause, you must define the *ColumnName* on the table you are creating. For more information about time-based aging, see Implementing an Aging Policy in Your Tables in *Oracle TimesTen In-Memory Database Operations Guide*.

Parameter	Description
LIFETIME *Num1* {SECOND[S]| MINUTE[S]| HOUR[S]| DAY[S]}	LIFETIME is a time-based aging attribute and is a required clause. Specify the LIFETIME clause after the AGING USE *ColumnName* clause. The LIFETIME clause specifies the minimum amount of time data is kept in cache. Specify *Num1* as a positive integer constant to indicate the unit of time expressed in seconds, minutes, hours or days that rows should be kept in cache. Rows that exceed the LIFETIME value are aged out (deleted from the table). If you define your aging column with data type TT_DATE, then you must specify DAYS as the LIFETIME unit. The concept of time resolution is supported. If 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. A LIFETIME of 3 days is different than a LIFETIME of 72 hours (3*24) or a LIFETIME of 432 minutes (3*24*60).
[CYCLE *Num2* {SECOND[S] |MINUTE[S]| HOUR[S]| DAY[S]}]	CYCLE is a time-based aging attribute and is optional. Specify the CYCLE clause after the LIFETIME clause. The CYCLE clause indicates how often the system should examine rows to see if data exceeds the specified LIFETIME value and should be aged out (deleted). Specify *Num2* as a positive integer constant. If you do not specify the CYCLE clause, then the default value is 5 minutes. If you specify 0 for *Num2*, then the aging thread wakes up every second. If the aging state is OFF, then aging is not done automatically and the CYCLE clause is ignored.
AS *SelectQuery*	If specified, creates a new table from the contents of the result set of the *SelectQuery*. The rows returned by *SelectQuery* are inserted into the table. Data types and data type lengths are derived from *SelectQuery*. *SelectQuery* is a valid SELECT statement that may or may not contain a subquery. See SELECT for information on the SELECT statement. You can specify a statement level optimizer hint after the SELECT verb. For more information on statement level optimizer hints, see Statement Level Optimizer Hints.

Column Definition: TimesTen Classic

SQL Syntax

You can only use the keyword, ENABLE, when defining columns in the CREATE TABLE statement.

For all data types other than LOBs, the syntax is as follows:

```
ColumnName ColumnDataType
 [DEFAULT DefaultVal]
 [[NOT] INLINE]
 [PRIMARY KEY | UNIQUE |
 NULL [UNIQUE] |
 NOT NULL [ENABLE] [PRIMARY KEY | UNIQUE]
]
```

For LOB data types, you cannot create a primary key or unique constraint on LOB columns. In addition, LOB data types are stored out of line, so the INLINE attribute cannot be specified.

LOB data types are not supported with TimesTen Scaleout.

For all LOB data types, the syntax is:

```
ColumnName ColumnDataType
 [DEFAULT DefaultVal] [[NOT] NULL [ENABLE]] |
 [[NOT] NULL [ENABLE]] [DEFAULT DefaultVal]
```

Parameters

The column definition has the following parameters:

Parameter	Description
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. A table can have a maximum of 1000 columns. If you specify the AS *SelectQuery* clause, *ColumnName* is optional. The number of column names must match the number of columns in *SelectQuery*.
ColumnDataType	Type of data the column can contain. Some data types require that you indicate a length. See Data Types for the data types that can be specified. If you specify the AS *SelectQuery* clause, do not specify *ColumnDataType*.
DEFAULT *DefaultVal*	Indicates that if a value is not specified for the column in an INSERT statement, the default value *DefaultVal* is inserted into the column. The default value specified must have a type that is compatible with the data type of the column. A default value can be as long as the data type of the associated column allows. You cannot assign a default value for the ROWID data type or for columns in read-only cache groups. In addition, you cannot use a function within the DEFAULT clause. The following are the supported data types for *DefaultVal*: • NULL • Constant expression (an expression that is evaluated to a constant value) • SYSDATE and GETDATE (SYSDATE and GETDATE) • SYSTEM_USER (SYSTEM_USER) If the default value is one of the users, the data type of the column must be either CHAR or VARCHAR2 and the width of the column must be at least 30 characters. If you specify the AS *SelectQuery* clause, optionally, you can specify the DEFAULT clause on the table you are creating.

Parameter	Description
INLINE|NOT INLINE	By default, variable-length columns whose declared column length is greater than 128 bytes are stored out of line. Variable-length columns whose declared column length is less than or equal to 128 bytes are stored inline. The default behavior can be overridden during table creation through the use of the INLINE and NOT INLINE keywords. If you specify the AS *SelectQuery* clause, optionally, you can specify the INLINE | NOT INLINE clause on the table you are creating.
NULL	Indicates that the column can contain NULL values. If you specify the AS *SelectQuery* clause, optionally, you can specify NULL on the table you are creating. If you specify NULL, you cannot specify ENABLE.
NOT NULL [ENABLE]	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. If you specify the AS *SelectQuery* clause, optionally, you can specify NOT NULL [ENABLE] on the table you are creating. If you specify NOT NULL, you can optionally specify ENABLE. Because NOT NULL constraints are always enabled, you are not required to specify ENABLE. You can only use the keyword, ENABLE, when defining columns in the CREATE TABLE statement.
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 range index to enforce uniqueness. So 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. If you specify the AS *SelectQuery* clause, optionally, you can specify UNIQUE on the table you are creating.
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. If you specify the AS *SelectQuery* clause, optionally, you can specify PRIMARY KEY on the table you are creating.

Description for CREATE TABLE: TimesTen Classic

- If you are planning to load your tables with data, consider creating your tables without indexes. After the data is loaded, you can then create your indexes. This reduces the time it take to load the data into the tables.
- All columns participating in the primary key are NOT NULL.
- A PRIMARY KEY that is specified in the *ColumnDefinition* can only be specified for one column.
- You cannot specify a PRIMARY KEY in both the *ColumnDefinition* clause and the PRIMARY KEY clause.
- For both primary key and foreign key constraints, duplicate column names are not allowed in the constraint column list.
- You cannot update primary key column(s) unless you update the column(s) to the same value.

- There are performance considerations when you define out of line columns instead of inline columns:
  - Accessing data is slower because TimesTen does not store data contiguously with out of line columns.
  - Populating data is slower because TimesTen generates more logging operations.
  - Deleting data is slower because TimesTen performs more reclaim and logging operations.
  - Storing a column requires less overhead.
- If ON DELETE CASCADE is specified on a foreign key constraint for a child table, a user can delete rows from a parent table for which the user has the DELETE privilege without requiring explicit DELETE privilege on the child table.
- To change the ON DELETE CASCADE triggered action, drop then redefine the foreign key constraint.
- 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.
- Use the ALTER TABLE statement to change the representation of the primary key index for a table.
- If you specify the AS *SelectQuery* clause:
  - Data types and data type lengths are derived from the *SelectQuery*. Do not specify data types on the columns of the table you are creating.
  - TimesTen defines on columns in the new table NOT NULL constraints that were explicitly created on the corresponding columns of the selected table if *SelectQuery* selects the column rather than an expression containing the column.
  - NOT NULL constraints that were implicitly created by TimesTen on columns of the selected table (for example, primary keys) are carried over to the new table. You can override the NOT NULL constraint on the selected table by defining the new column as NULL. For example:

    ```
    CREATE TABLE newtable (newcol NULL) AS SELECT (col) FROM tab;
    ```
  - NOT INLINE/INLINE attributes are carried over to the new table.
  - Unique keys, foreign keys, indexes and column default values are not carried over to the new table.
  - If all expressions in *SelectQuery* are columns, rather than expressions, then you can omit the columns from the table you are creating. In this case, the name of the columns are the same as the columns in *SelectQuery*. If the *SelectQuery* contains an expression rather than a simple column reference, either specify a column alias or name the column in the CREATE TABLE statement.
  - Do not specify foreign keys on the table you are creating.
  - Do not specify the SELECT FOR UPDATE clause in *SelectQuery*.
  - The ORDER BY clause is not supported when you use the AS *SelectQuery* clause.
  - *SelectQuery* cannot contain set operators UNION, MINUS, INTERSECT.
  - In a replicated environment, be aware of the following.

    To include a new table, including global temporary tables, into an active standby pair when the table is created, set DDL_REPLICATION_LEVEL to 2 or greater and

DDL_REPLICATION_ACTION to INCLUDE before executing the CREATE TABLE statement on the active database. In this configuration, the table is included in the active standby pair and is replicated to all databases in the replication scheme.

If DDL_REPLICATION_ACTION is set to EXCLUDE, then the new table is not included in the active standby pair but is replicated to all databases in the replication scheme. Any DML issued on that table will not be replicated, as the table will not be part of the replication scheme. To enable DML replication for the table, you must execute the ALTER ACTIVE STANDBY PAIR ... INCLUDE TABLE statement to include the table. In this case, the table must be empty and present on all databases before executing ALTER ACTIVE STANDBY PAIR ... INCLUDE TABLE, as the table contents will be truncated when this statement is executed.

See ALTER SESSION for more information.

- By default, a range 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 using a table's primary key, then choose a range index for that table by omitting the UNIQUE HASH clause.
  - If 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.
- A hash index is created with a fixed size that remains constant for the life of the table or until the hash index is resized with the ALTER TABLE statement or when the index is dropped and recreated. A smaller hash index results in more hash collisions. A larger hash index reduces collisions but can waste memory. Hash key comparison is a fast operation, so a small number of hash collisions should not cause a performance problem for TimesTen.

  To ensure that your hash index is sized correctly, your application must indicate the expected size of your table with the value of the *RowPages* parameter of the SET PAGES clause. Compute this value by dividing the number of expected rows in your table by 256. For example, if your table has 256,000 rows, specify 1000 for the value of RowPages (256000/256=1000).
- At most 16 columns are allowed in a hash key.
- 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 causes cascaded deletes in the child table. This, in turn, triggers 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, the following 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 is done at runtime. If there is an error, the receiver thread stops working.
  - All tables in the delete cascade tree have to be replicated if any table in the tree is replicated. This restriction is 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 data in a global temporary table is private to the current connection and does not need to be secured between users. Thus, global temporary tables do not require object privileges.
- After you have defined an aging policy for the table, you cannot change the policy from LRU to time-based or from time-based to LRU. You must first drop aging and then alter the table to add a new aging policy.
- The aging policy must be defined to change the aging state.
- For the time-based aging policy, you cannot add or modify the aging column. This is because you cannot add or modify a NOT NULL column.
- 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 database. 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.
- The following rules determine if a row is accessed or referenced for LRU aging:
  - 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.
- 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.
- Call the ttAgingScheduleNow procedure to schedule the aging process immediately regardless of the aging state.
- 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 column that is used for time-based aging.
  - The aging policy and aging state must be the same in all sites of replication.
  - Tables that are related by foreign keys must have the same aging policy.
  - For LRU aging, if a child row is not a candidate for aging, neither this child row nor its parent row are 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.

Column-Based Compression of Tables (TimesTen Classic)

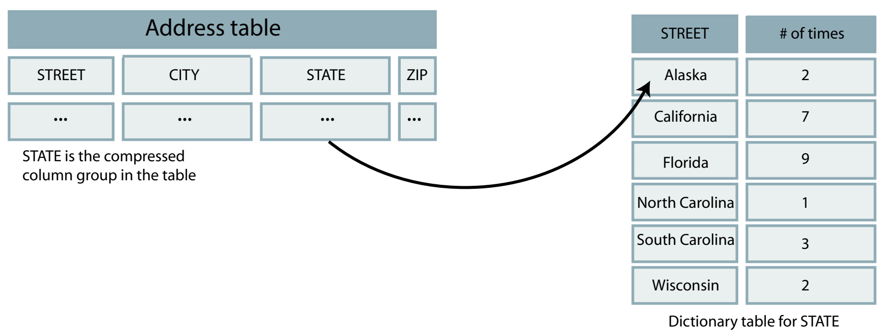
You can compress tables at the column level, which stores data more efficiently. This eliminates redundant storage of duplicate values within columns and improves the performance of SQL queries that perform full table scans.

You can define one or more columns in a table to be compressed together, which is called a compressed column group. You can define one or more compressed column groups in each table.

A dictionary table is created for each compressed column group that contains a column with all the distinct values of the compressed column group. The compressed column group now contains a pointer to the row in the dictionary table for the appropriate value. The width of this pointer can be 1, 2, or 4 bytes long depending on the maximum number of entries you defined for the dictionary table. So if the sum of the widths of the columns in a compressed column group is wider than the 1, 2, or 4 byte pointer width, and if there are a lot of duplicate values of those column values, you have reduced the amount of space used by the table.

Figure 6-1 shows the compressed column group in the table pointing to the appropriate row in the dictionary table.

Figure 6-1 Column-Based Compression



The dictionary table has a column of pointers to each of the distinct values. When the user configures the maximum number of distinct entries for the compressed column group, the size of the compressed column group is set as follows:

- 1 byte for a maximum number of entries of 255 (2^8-1). When the maximum number is between 1 and 255, the dictionary size is set to 255 (2^8-1) values and the compressed column group pointer column is 1 byte.
- 2 bytes for a maximum number of entries of 65,535 (2^{16}-1). When the maximum number is between 256 and 65,535, the dictionary size is set to 65,535 (2^{16}-1) values and the compressed column group pointer column is 2 bytes.
- 4 bytes for a maximum number of entries of 4,294,967,295 (2^{32}-1). When the maximum number is between 65,536 and 4,294,967,295, the dictionary size is set to 4,294,967,295 (2^{32}-1) values and the compressed column group pointer column is 4 bytes. This is the default.

Syntax: Column-Based Compression (TimesTen Classic)

The syntax for *ColumnBasedCompression* is:

```
[COMPRESS (CompressColumns [,...])]
```

The *CompressColumns* syntax is as follows:

```
{ColumnDefinition | (ColumnDefinition [,...])} BY DICTIONARY
  [MAXVALUES = CompressMax]
```

Parameters

ColumnBasedCompression syntax has the following parameters:

Parameter	Description
COMPRESS (*CompressColumns* [,...])	Defines a compressed column group for a table that is enabled for compression. This can include one or more columns in the table. However, a column can be included in only one compressed column group. Only INLINE columns are supported when you specify multiple columns in a compressed column group. An out-of-line column cannot be in a multi-column compression group. Each compressed column group is limited to a maximum of 16 columns.
BY DICTIONARY	Defines a compression dictionary for each compressed column group.
MAXVALUES = *CompressMax*	*CompressMax* is the total number of distinct values in the table and sets the size for the compressed column group pointer column to 1, 2, or 4 bytes and sets the size for the maximum number of entries in the dictionary table. For the dictionary table, NULL is counted as one unique value. *CompressMax* takes an integer between 1 and 429497295 (2^{32}-1). The maximum size defaults to size of 2^{32}-1 if the MAXVALUES clause is omitted, which uses 4 bytes for the pointer column. An error is thrown if the value is greater than 2^{32}-1.

Description: Column-Based Compression (TimesTen Classic)

- Compressed column groups can be added at the time of table creation or added later using ALTER TABLE. You can drop a compressed column group with the ALTER TABLE statement, but you must drop the entire group.
- You can create indexes on any columns in the table and on columns that exist in separate compression column groups. However, you cannot create single column compression groups on unique columns or on single column primary keys. You also cannot create unique indexes or primary keys where all the indexes or primary keys are in the same compression group.
- LOB columns cannot be compressed.
- Compression is not supported on columns in replicated tables, cache group tables, or on global temporary tables. You cannot create a table with the CREATE TABLE AS SELECT statement when defining column-based compression for that table in that statement.
- You cannot create materialized views on tables enabled for compression.
- Column-based compression is not supported with TimesTen Scaleout.

Examples: TimesTen Classic

A range 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 range index on columns:
    PARTNUMBER
  1 index found.
1 index found on 1 table.
```

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;
```

A range 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 range index on columns:
    MEMBERNAME
    CLUB
  1 index found on 1 table.
```

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));
```

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;
```

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;
```

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));
```

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);
```

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));
```

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);
```

This example creates the rangex table and defines col1 as the primary key. A range index is created by default.

```
Command> CREATE TABLE rangex (col1 TT_INTEGER PRIMARY KEY);
Command> INDEXES rangex;
Indexes on table SAMPLEUSER.RANGEX:
  RANGEX: unique range index on columns:
    COL1
  1 index found
1 index found on 1 table.
```

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;
    ;
```

This example shows how time resolution works with aging.

If lifetime is three days (resolution is in days):

- If (SYSDATE - *ColumnValue*) <= 3, do not age.
- If (SYSDATE - *ColumnValue*) > 3, then the row is a candidate for aging.
- If (SYSDATE - *ColumnValue*) = 3 days, 22 hours, then the row is not aged out if you specified a lifetime of three days. The row would be aged out if you had specified a lifetime of 72 hours.

This 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 *)
```

This example creates a table with time-based aging. Lifetime is three days. Cycle is not specified, so the default is five 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 *)
```

This example generates an error message. It illustrates that after you create an aging policy, you cannot change it. You must drop aging and redefine aging.

```
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 *)
```

Attempt to create a table with time-based aging. Define aging column with data type TT_DATE and LIFETIME 3 hours. An error is generated because the LIFETIME unit must be expressed as DAYS.

```
Command> CREATE TABLE aging1 (col1 TT_INTEGER PRIMARY KEY,
        col2 TT_DATE NOT NULL) AGING USE col2 LIFETIME 3 HOURS;
 2977: Only DAY lifetime unit is allowed with a TT_DATE column
The command failed.
```

Use AS *SelectQuery* clause to create the table emp. Select last_name from the employees table where employee_id between 100 and 105. You see six rows inserted into emp. First issue the SELECT statement to see rows that should be returned.

```
Command> SELECT last_name FROM employees
       WHERE employee_id BETWEEN 100 AND 105;
< King >
< Kochhar >
< De Haan >
< Hunold >
< Ernst >
< Austin >
6 rows found.
Command> CREATE TABLE emp AS SELECT last_name FROM employees
       WHERE employee_id BETWEEN 100 AND 105;
6 rows inserted.
Command> SELECT * FROM emp;
< King >
< Kochhar >
< De Haan >
< Hunold >
< Ernst >
< Austin >
6 rows found.
```

Use AS *SelectQuery* to create table totalsal. Sum salary and insert result into totalsalary. Define alias s for *SelectQuery* expression.

```
Command> CREATE TABLE totalsal AS SELECT SUM (salary) s FROM employees;
1 row inserted.
Command> SELECT * FROM totalsal;
< 691400 >
1 row found.
```

Use AS *SelectQuery* to create table defined with column commission_pct. Set default to .3. First describe table employees to show that column commission_pct is of type NUMBER (2,2). For table c_pct, column commission_pct inherits type NUMBER (2,2) from column commission_pct of employees table.

```
Command> DESCRIBE employees;
Table SAMPLEUSER.EMPLOYEES:
  Columns:
   *EMPLOYEE_ID                    NUMBER (6) NOT NULL
    FIRST_NAME                     VARCHAR2 (20) INLINE
    LAST_NAME                      VARCHAR2 (25) INLINE NOT NULL
    EMAIL                          VARCHAR2 (25) INLINE UNIQUE NOT NULL
    PHONE_NUMBER                   VARCHAR2 (20) INLINE
    HIRE_DATE                      DATE NOT NULL
    JOB_ID                         VARCHAR2 (10) INLINE NOT NULL
    SALARY                         NUMBER (8,2)
    COMMISSION_PCT                 NUMBER (2,2)
    MANAGER_ID                     NUMBER (6)
    DEPARTMENT_ID                  NUMBER (4)

1 table found.
(primary key columns are indicated with *)
Command> CREATE TABLE c_pct (commission_pct DEFAULT .3) AS SELECT
       commission_pct FROM employees;
107 rows inserted.
Command> DESCRIBE c_pct;

Table SAMPLEUSER.C_PCT:
  Columns:
    COMMISSION_PCT                 NUMBER (2,2) DEFAULT .3
```

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

The following example creates the employees table where the job_id is compressed.

```
Command> CREATE TABLE EMPLOYEES
        (EMPLOYEE_ID NUMBER (6) PRIMARY KEY,
         FIRST_NAME VARCHAR2(20),
         LAST_NAME VARCHAR2(25) NOT NULL,
         EMAIL VARCHAR2(25) NOT NULL,
         PHONE_NUMBER VARCHAR2(20),
         HIRE_DATE DATE NOT NULL,
         JOB_ID VARCHAR2(10) NOT NULL,
         SALARY NUMBER (8,2),
         COMMISSION_PCT NUMBER (2,2),
         MANAGER_ID NUMBER(6),
         DEPARTMENT_ID NUMBER(4))
        COMPRESS (JOB_ID BY DICTIONARY);

Command> DESCRIBE EMPLOYEES;

Table MYSCHEMA.EMPLOYEES:
  Columns:
   *EMPLOYEE_ID                     NUMBER (6) NOT NULL
    FIRST_NAME                      VARCHAR2 (20) INLINE
    LAST_NAME                       VARCHAR2 (25) INLINE NOT NULL
    EMAIL                           VARCHAR2 (25) INLINE NOT NULL
    PHONE_NUMBER                    VARCHAR2 (20) INLINE
    HIRE_DATE                       DATE NOT NULL
    JOB_ID                          VARCHAR2 (10) INLINE NOT NULL
    SALARY                          NUMBER (8,2)
    COMMISSION_PCT                  NUMBER (2,2)
    MANAGER_ID                      NUMBER (6)
    DEPARTMENT_ID                   NUMBER (4)
  COMPRESS ( JOB_ID BY DICTIONARY )

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

The following example shows that there are three dictionary table sizes. The value you specify for the maximum number of entries is rounded up to the next size. For example, specifying 400 as the maximum number of job IDs creates a dictionary table that can have at most 65535 entries. The default size of 2^{32}-1 is not shown in the DESCRIBE output.

```
Command> CREATE TABLE employees
        (employee_id NUMBER(6) PRIMARY KEY,
         first_name VARCHAR2(20),
         last_name VARCHAR2(25),
         email VARCHAR2(25) NOT NULL,
         job_id VARCHAR2(10) NOT NULL,
         manager_id NUMBER(6),
         department_id NUMBER(4))
        COMPRESS (last_name BY DICTIONARY MAXVALUES=70000,
          job_id BY DICTIONARY MAXVALUES=400,
          department_id BY DICTIONARY MAXVALUES=100);

Command> DESCRIBE employees;
Table MYSCHEMA.EMPLOYEES:
  Columns:
   *EMPLOYEE_ID                     NUMBER (6) NOT NULL
    FIRST_NAME                      VARCHAR2 (20) INLINE
    LAST_NAME                       VARCHAR2 (25) INLINE
```

```
  EMAILS                    VARCHAR2 (25) INLINE NOT NULL
  JOB_ID                    VARCHAR2 (10) INLINE NOT NULL
  MANAGER_ID                NUMBER (6)
  DEPARTMENT_ID             NUMBER (4)
 COMPRESS ( LAST_NAME BY DICTIONARY,
        JOB_ID BY DICTIONARY MAXVALUES=65535,
        DEPARTMENT_ID BY DICTIONARY MAXVALUES=255 )

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

See also:

ALTER TABLE
DROP TABLE
TRUNCATE TABLE
UPDATE

CREATE USER

The CREATE USER statement creates a user in the TimesTen database.

Required Privilege

ADMIN

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE USER user IDENTIFIED BY {password | "password"}
 [PROFILE profile] [ACCOUNT {LOCK|UNLOCK}] [PASSWORD EXPIRE]
```

or

```
CREATE USER user IDENTIFIED EXTERNALLY
 [PROFILE profile] [ACCOUNT {LOCK|UNLOCK}]
```

Parameters

Parameter	Description
user	Name of the user.

Parameter	Description
IDENTIFIED BY {*password* | "*password*"}	Identification clause for an internal user. You must supply a password for an internal user. The password you can specify is dependent on the profile assigned to the user. Specifically, the value of the PASSWORD_COMPLEXITY_CHECKER password parameter determines the complexity of the password. If the value is TT_VERIFY_FUNCTION, TT_STRONG_VERIFY_FUNCTION, or TT_STIG_VERIFY_FUNCTION, the password must meet specific password verification requirements. For example, if the value is TT_VERIFY_FUNCTION, the password cannot contain the name of the database. See About Password Complexity Checker Verification for details.
IDENTIFIED EXTERNALLY	Identifies an external user (the operating system user). To perform database operations as an external user, the external user name must match the user name authenticated by the operating system or network. A password is not required by TimesTen as the user has been authenticated by the operating system at login time.
PROFILE *profile*	Use the PROFILE clause to specify the name of the profile (designated by *profile*) that you want to assign to the user. The profile sets the limits for the password parameters for the user. See CREATE PROFILE for information on these password parameters. If you omit the PROFILE clause, TimesTen assigns the DEFAULT profile to the user. If you create an external user (denoted by specifying the EXTERNALLY keyword), you can specify a PROFILE clause, but the password parameters have no effect on external users. Additionally, if you do not specify the PROFILE clause for an external user, TimesTen assigns the DEFAULT profile to the user (but the password parameters have no effect).
ACCOUNT [LOCK|UNLOCK]	Specify ACCOUNT LOCK to lock the user's account and disable connections to the database. Specify ACCOUNT UNLOCK to unlock the user's account and enable connections to the database. The default is ACCOUNT UNLOCK.
PASSWORD EXPIRE	Specify PASSWORD EXPIRE if you want the user's password to expire. This setting forces a user with ADMIN privileges to change the password before the user can connect to the database. In order to change the expired password, a user with ADMIN privileges must use the ALTER USER statement with the IDENTIFIED BY clause to change the password. Once the password is changed, the user can log in to the database with the new password. Note that even if the newly created user is granted ADMIN privileges, that newly created user cannot login to the database and therefore cannot initially change the password. See ALTER USER for information. This clause is not valid for an externally identified user (as denoted by the IDENTIFIED EXTERNALLY clause).

Description

- Database users can be internal or external.
  - Internal users are defined for a TimesTen database.
  - External users are defined by the operating system. External users cannot be assigned a TimesTen password.
- Password requirements:
  - Cannot exceed 30 characters.
  - Is case-sensitive.
  - Must start with a letter. A password cannot start with a digit or a special character unless the password is enclosed in double quotation marks.
  - If a special character is used, the password must be contained in double quotation marks. The exceptions are the # and the @ special characters. A password that contains the # or the @ special character does not need to be enclosed in double quotation marks.
  - Cannot contain a semi-colon (;) or a double quotation mark (").
- When a user is created, the user has the privileges granted to PUBLIC and no additional privileges.
- Use the PROFILE clause to assign a profile to a user. If you assign the profile to an internal user, the user cannot exceed the limits specified for the profile. If you do not assign a profile to an internal user, a DEFAULT profile is assigned to that user. See "CREATE PROFILE" for details.
- Use the ACCOUNT LOCK or ACCOUNT UNLOCK to lock or unlock the user account.
- Use the PASSWORD EXPIRE clause to expire the user's password and force a password change before the user can connect to the database.
- You can create a user over a client/sever connection if the connection is encrypted with TLS. See Transport Layer Security for TimesTen Client/Server in the *Oracle TimesTen In-Memory Database Security Guide* for details.
- In TimesTen, user brad is the same as user "brad". In both cases, the name of the user is created as BRAD.
- User names are TT_CHAR data type.
- This statement is replicated.

Examples

Create Users and Observe Password Verification

This example creates the user_pw1 user and does not assign a profile to the user1_pw user. The user is subject to the limits of the DEFAULT profile. The PASSWORD_COMPLEXITY_CHECKER password parameter is set to NULL for the DEFAULT profile. Therefore, there is no password verification performed on this user's password. The example then alters the DEFAULT profile, changing the value of the PASSWORD_COMPLEXITY_CHECKER to TT_VERIFY_FUNCTION. The user1_p1 user can still connect to the database with the original password. Password verification is performed only on newly created users.

```
Command> CREATE USER user_pw1 IDENTIFIED BY user1_pw1;

User created.
```

Query the dba_profiles system view to check the limits of the password parameters for the DEFAULT profile. The PASSWORD_COMPLEXITY_CHECKER password parameter has a value of NULL.

```
Command> SELECT * FROM dba_profiles WHERE profile = 'DEFAULT';
< DEFAULT, FAILED_LOGIN_ATTEMPTS, PASSWORD, 10 >
< DEFAULT, PASSWORD_LIFE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_REUSE_MAX, PASSWORD, UNLIMITED >
< DEFAULT, PASSWORD_COMPLEXITY_CHECKER, PASSWORD, NULL >
< DEFAULT, PASSWORD_LOCK_TIME, PASSWORD, .0034 >
< DEFAULT, PASSWORD_GRACE_TIME, PASSWORD, UNLIMITED >
< DEFAULT, TEMP_SPACE_PER_SESSION_MAX, MEMORY, UNLIMITED >
8 rows found.
```

Alter the DEFAULT profile, changing the value of the PASSWORD_COMPLEXITY_CHECKER parameter to TT_VERIFY_FUNCTION. Attempt to connect to the database as the user_pw1 user. The connection is successful, as password verification is only performed on newly created passwords.

```
Command> ALTER PROFILE "DEFAULT" LIMIT
       PASSWORD_COMPLEXITY_CHECKER TT_VERIFY_FUNCTION;

Profile altered.

Command> connect adding "UID=user_pw1;PWD=user_pw1" as user1;
Connection successful: DSN=access1;UID=user_pw1;
DataStore=/scratch/user1/mydatabase1;DatabaseCharacterSet=AL32UTF8;
ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Create the user_pw2 user and specify user_pw2 for the password. The CREATE USER statement fails. Password verification is performed on the password for user_pw2, as the password is a newly created password. Create the user_pw2 user again, specifying a password that meets the requirements of the TT_VERIFY_FUNCTION function. The CREATE USER statement is successful, and the user is created. See TT_VERIFY_FUNCTION for more information on the TT_VERIFY_FUNCTION function.

```
Command> CREATE USER user_pw2 IDENTIFIED BY user_pw2;
15186: Password complexity check for the specified password failed
15188: TT-20002: Password contains the username
The command failed.

Command> CREATE USER user_pw2 IDENTIFIED BY abc75#n4;

User created.
```

Create User with TT_STRONG_VERIFY_FUNCTION Password Qequirements

This example illustrates the password verification requirements for the TT_STRONG_VERIFY_FUNCTION function. Create the profile_pw3 profile and specify a value of TT_STRONG_VERIFY_FUNCTION for the PASSWORD_COMPLEXITY_CHECKER password parameter. Create the user_pw3 user and assign this user the profile_pw3 profile. Experiment with different passwords to confirm that the password meets the requirements of the TT_STRONG_VERIFY_FUNCTION function. If the password meets the requirements, the CREATE USER statement is successful and the user is created. See TT_STRONG_VERIFY_FUNCTION.

```
Command> CREATE PROFILE profile_pw3 LIMIT
         PASSWORD_COMPLEXITY_CHECKER TT_STRONG_VERIFY_FUNCTION;

Profile created.
```

Create the user_pw3 user and experiment with various passwords. Recall that special characters must be enclosed in double quotation marks (with the exception of # and @).

```
Command> CREATE USER user_pw3 IDENTIFIED BY abcABC1#
         PROFILE profile_pw3;
15186: Password complexity check for the specified password failed
15188: TT-20001: Password length less than 9
The command failed.

Command> CREATE USER user_pw3 IDENTIFIED BY abcABCD1#
         PROFILE profile_pw3;
15186: Password complexity check for the specified password failed
15188: TT-20001: Password must contain at least 2 digit(s)
The command failed.

Command> CREATE USER user_pw3 IDENTIFIED BY abcABCD11#
         PROFILE profile_pw3;
15186: Password complexity check for the specified password failed
15188: TT-20001: Password must contain at least 2 special character(s)
The command failed.

Command> CREATE USER user_pw3 IDENTIFIED BY "!abcABCD11#"
         PROFILE profile_pw3;

User created.
```

Create a User and Assign a Profile

This example creates the user1 user and assigns the profile1 profile to the user.

```
Command> CREATE USER user1 IDENTIFIED BY user1 PROFILE profile1;

User created.
```

Create a User and Do Not Assign a Profile

This example creates the user2 user and does not assign a profile. The user2 user is assigned the values of the password parameters in the DEFAULT profile.

```
Command> CREATE USER user2 identified by user2;

User created.
```

Query the dba_users system view to verify the user2 user is assigned the DEFAULT profile.

```
Command> SELECT profile FROM dba_users WHERE username='USER2';
< DEFAULT >
1 row found.
```

Create a User and Lock the User Account

This example creates the user3 user and locks the user3 account. The user3 account must be unlocked by a user with the ADMIN privilege before the user3 user can connect to the database.

```
Command> CREATE USER user3 IDENTIFIED BY user3 ACCOUNT LOCK;

User created.
```

Grant the CONNECT privilege to user3;

```
Command> GRANT CONNECT TO user3;
```

Attempt to connect to the database as user3. The user3 account is locked so the connection fails.

```
Command> connect adding "UID=user3;PWD=user3" as user3;
15179: the account is locked
The command failed.
```

As the instance administrator, reconnect to the database and use the ALTER USER statement to unlock the user3 account.

```
none: Command> use database1
database1: Command> ALTER USER user3 ACCOUNT UNLOCK;

User altered.
```

Attempt to connect to the database a the user3 user. The connection succeeds.

```
database1: Command> connect adding "UID=user3;PWD=user3" as user3;
Connection successful: DSN=database1;UID=user3;DataStore=/scratch/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Lock the User Account and Enforce a Password Change

This example creates the user4 user. The user4 user is assigned the profile1 profile. The user4 account is locked and the password for user4 must be changed before the user4 user can connect to the database.

```
Command> CREATE USER user4 identified by user4 PROFILE profile1
       ACCOUNT LOCK PASSWORD EXPIRE;

User created.
```

Attempt to connect to the database as user4. The user4 account is locked and the password must be changed before the user4 user can connect to the database.

```
Command>  connect adding "UID=user4;PWD=user4" as user4;
15179: the account is locked
The command failed.
```

As the instance administrator, reconnect to the database and use the ALTER USER statement to unlock the user4 account.

```
none: Command> use database1
database1: Command> ALTER USER user4 ACCOUNT UNLOCK;

User altered.
```

Grant the CONNECT privilege to user4. Then change the user4's expired password. (This example changes the password to user4_changed, represented in **bold**.)

```
database1: Command> GRANT CONNECT TO user4;
database1: Command> ALTER USER user4 IDENTIFIED BY user4_changed;

User altered.
```

Attempt to connect to the database as the user4 user. The connection succeeds. The account is unlock and the password is changed.

```
database1: Command> connect adding "UID=user4;PWD=user4_changed" as user4;
Connection successful: DSN=database1;UID=user4;DataStore=/scratch/database1;
DatabaseCharacterSet=AL32UTF8;ConnectionCharacterSet=AL32UTF8;PermSize=128;
(Default setting AutoCommit=1)
```

Create an External User

This example creates the user5 user as an external user.

```
Command> CREATE USER user5 IDENTIFIED EXTERNALLY;

User created.
```

See Also

- ALTER USER
- DROP USER
- CREATE PROFILE
- ALTER PROFILE
- GRANT
- REVOKE

CREATE VIEW

The CREATE VIEW statement creates a view of the tables specified in the *SelectQuery* clause. A view is a logical table that is based on one or more *detail tables*. The view itself contains no data. It is sometimes called a *nonmaterialized view* to distinguish it from a materialized view, which does contain data that has already been calculated from detail tables.

In a replicated environment for an active standby pair, if DDL_REPLICATION_LEVEL is 3 or greater when you execute CREATE VIEW on the active database, the view is replicated to all databases in the replication scheme. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Required Privilege

The user executing the statement must have the CREATE VIEW privilege (if owner) or CREATE ANY VIEW (if not the owner) for another user's view.

The owner of the view must have the SELECT privilege on the detail tables.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
CREATE VIEW [Owner.]ViewName AS SelectQuery
```

Parameters

Parameter	Description
[Owner.]ViewName	Name of view
SelectQuery	Selects column from the detail tables to be used in the view. You can also create indexes on the view.

Restrictions on the SELECT Query

There are several restrictions on the query that is used to define the view.

- A SELECT * query in a view definition is expanded when the view is created. Any columns added after a view is created do not affect the view.
- Do not use the following in a SELECT statement that is used to create a view:
  - FIRST
  - ORDER BY

    If used, this is ignored by CREATE VIEW. The result will not be sorted.
  - Arguments
- 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.
- Do not use SELECT FOR UPDATE to create 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.
- 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 query to form a supported 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.
- Use the DROP VIEW statement to drop a view.
- A view cannot be altered with an ALTER TABLE statement.
- Referencing a view can fail because of dropped or altered detail tables.

Examples

Create a nonmaterialized view from the employees table.

```
Command> CREATE VIEW v1 AS SELECT employee_id, email FROM employees;
Command> SELECT FIRST 5 * FROM v1;
```

```
< 100, SKING >
< 101, NKOCHHAR >
< 102, LDEHAAN >
< 103, AHUNOLD >
< 104, BERNST >
5 rows found.
```

Create a nonmaterialized view `tview` with column `max1` from an aggregate query on the table `t1`.

```
CREATE VIEW tview (max1) AS SELECT MAX(x1) FROM t1;
```

See also

CREATE MATERIALIZED VIEW
CREATE TABLE
DROP VIEW

DELETE

The `DELETE` statement deletes rows from a table.

Required Privilege

No privilege is required for the table owner.

`DELETE` on the table for another user's table.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
DELETE [hint] [FIRST NumRows] FROM [Owner.]TableName [CorrelationName]
[WHERE SearchCondition]
[RETURNING|RETURN Expression[,...]INTO DataItem[,...]]
```

Parameters

Parameter	Description
hint	Specifies a statement level optimizer hint for the `DELETE` statement. See Statement Level Optimizer Hints for information on optimizer hints.
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* identifies a table to be deleted. *CorrelationName* specifies an alias for the immediately preceding table. Use the correlation name to reference the table elsewhere in the `DELETE` statement. The scope of the *CorrelationName* is the SQL statement in which it is used. It must conform to the syntax rules for a basic name. See Basic Names for details.

Parameter	Description
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.
Expression	Valid expression syntax. See Expressions for details.
DataItem	Host variable or PL/SQL variable that stores the retrieved *Expression* value.

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 your table has out of line columns and there are millions of rows to delete, consider calling the ttCompact built-in procedure to free memory.
- The DELETE operation fails if it violates any foreign key constraint. See CREATE TABLE for a description of the foreign key constraint.
- 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.
- If ON DELETE CASCADE is specified on a foreign key constraint for a child table, a user can delete rows from a parent table for which the user has the DELETE privilege without requiring explicit DELETE privilege on the child table.
- Restrictions on the RETURNING clause:
  - Each *Expression* must be a simple expression. Aggregate functions are not supported.
  - You cannot return a sequence number into an OUT parameter.
  - ROWNUM and subqueries cannot be used in the RETURNING clause.
  - Parameters in the RETURNING clause cannot be duplicated anywhere in the DELETE statement.
  - Using the RETURNING clause to return multiple rows requires PL/SQL BULK COLLECT functionality. See FORALL and BULK COLLECT Operations in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for information about BULK COLLECT.
  - In PL/SQL, you cannot use a RETURNING clause with a WHERE CURRENT operation.

Examples

Rows for orders whose quantity is less than 50 are deleted.

```
DELETE FROM purchasing.orderitems
WHERE quantity < 50;
```

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);
```

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';
```

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.
```

Delete an employee from employees. Declare empid and name as variables with the same data types as employee_id and last_name. Delete the row, returning employee_id and last_name into the variables. Verify that the correct row was deleted.

```
Command> VARIABLE empid NUMBER(6) NOT NULL;
Command> VARIABLE name VARCHAR2(25) INLINE NOT NULL;
Command> DELETE FROM employees WHERE last_name='Ernst'
         RETURNING employee_id, last_name INTO :empid,:name;
1 row deleted.
Command> PRINT empid name;
EMPID            : 104
NAME             : Ernst
```

DROP ACTIVE STANDBY PAIR

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

This statement drops an active standby pair replication scheme.

Required privilege

ADMIN

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
DROP ACTIVE STANDBY PAIR
```

Parameters

DROP ACTIVE STANDBY PAIR 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 database on which the statement was issued.

You cannot execute the DROP ACTIVE STANDBY PAIR statement when Oracle Clusterware is used with TimesTen.

See also

ALTER ACTIVE STANDBY PAIR
CREATE ACTIVE STANDBY PAIR

DROP CACHE GROUP

This statement is supported in TimesTen Scaleout.

In TimesTen Classic:

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.

Required privilege

No privilege is required for the cache group owner.

If not the cache group owner, DROP ANY CACHE GROUP *and*

DROP ANY TABLE if at least one table in the cache group is not owned by the current user.

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
DROP CACHE GROUP [Owner.]CacheGroupName
```

Parameters

Parameter	Description
[Owner.]CacheGroupName	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 Database objects for read-only cache groups and cache groups with the AUTOREFRESH attribute are uninstalled by the cache agent. If the cache agent is not running during the DROP CACHE GROUP operation, the Oracle Database objects are uninstalled on the next startup of the cache agent.
- You cannot execute the DROP CACHE GROUP statement when performed under the serializable isolation level. An error message is returned when attempted.
- 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 fails with a lock timeout error.

- If LockWait interval is nonzero, then the current autorefresh transaction is preempted (rolled back), and the DROP statement continues. This affects all cache groups with the same autorefresh interval.

Examples

```
DROP CACHE GROUP westerncustomers;
```

See also

ALTER CACHE GROUP
CREATE CACHE GROUP

DROP FUNCTION

The DROP FUNCTION statement removes a standalone stored function from the database. Do not use this statement to remove a function that is part of a package.

Required privilege

No privilege is required for the function owner.

DROP ANY PROCEDURE for another user's function.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
DROP FUNCTION [Owner.]FunctionName
```

Parameters

Parameter	Description
[Owner.]FunctionName	Name of the function to be dropped.

Description

- When you drop a function, TimesTen invalidates objects that depend on the dropped function. If you subsequently reference one of these objects, TimesTen attempts to recompile the object and returns an error message if you have not recreated the dropped function.
- Do not use this statement to remove a function that is part of a package. Either drop the package or redefine the package without the function using the CREATE PACKAGE statement with the OR REPLACE clause.
- To use the DROP FUNCTION statement, you must have PL/SQL enabled in your database. If you do not have PL/SQL enabled in your database, an error is thrown.

Examples

The following statement drops the function myfunc and invalidates all objects that depend on myfunc:

```
Command> DROP FUNCTION myfunc;

Function dropped.
```

If PL/SQL is not enabled in your database, TimesTen returns an error:

```
Command> DROP FUNCTION myfunc;
 8501: PL/SQL feature not installed in this TimesTen database
The command failed.
```

See also

CREATE FUNCTION

DROP INDEX

The DROP INDEX statement deletes the specified index. The index can be global (TimesTen Scaleout or local (TimesTen Scaleoutor TimesTen Classic.

Required Privilege

No privilege is required for the index owner. DROP ANY INDEX is required for an index owned by another user.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout. Use the DROP INDEX statement to drop both global and local indexes.

SQL Syntax

DROP INDEX [*Owner*.]*IndexName* [FROM [*Owner*.]*TableName*]

Parameters

Parameter	Description
[*Owner*.]*IndexName*	Name of the index to be dropped. You can include the name of the owner of the table for the index. For TimesTen Scaleout, the index can be global or local.
[*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 the index associated with it, use the ALTER TABLE statement.
- If an index is created on a UNIQUE column constraint, it can only be dropped by dropping the constraint. Use the ALTER TABLE DROP UNIQUE statement for this purpose. Also, see CREATE TABLE for more information about the UNIQUE column constraint.
- If a DROP INDEX operation is or was active in an uncommitted transaction, other transactions that are performing DML operations, that do not access that index, are blocked.
- If an index is dropped, any prepared statement that uses the index is automatically prepared again 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.
- 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.
- 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 an instance of the table that is not empty.
- If the index is replicated across an active standby pair and if `DDL_REPLICATION_LEVEL` is 2 or greater, use the `DROP INDEX` statement to drop the index from the active standby pair in the replication scheme. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Examples

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

DROP MATERIALIZED VIEW

The `DROP MATERIALIZED VIEW` statement removes the specified materialized view, including any hash indexes and any range indexes associated with it.

Required privilege

View owner or `DROP ANY MATERIALIZED VIEW` (if not owner) *and*

Table owner or `DROP ANY TABLE` (if not owner) *and*

Index owner or `DROP ANY INDEX` (if not owner) if there is an index on the view.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
DROP MATERIALIZED VIEW [Owner.]ViewName
```

Parameters

Parameter	Description
[*Owner*.]*ViewName*	Identifies the materialized view to be dropped.

Description

When you execute a DROP MATERIALIZED VIEW operation, the detail tables are updated and locked. An error may result if the detail table was already locked by another transaction.

Examples

The following statement drops the custorder materialized view.

```
DROP MATERIALIZED VIEW custorder;
```

See also

CREATE MATERIALIZED VIEW

DROP PACKAGE [BODY]

The DROP PACKAGE statement removes a stored package from the database. Both the specification and the body are dropped. DROP PACKAGE BODY removes only the body of the package.

Required privilege

No privilege is required for the package owner.

DROP ANY PROCEDURE for another user's package.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
DROP PACKAGE [BODY] [Owner.]PackageName
```

Parameters

Parameter	Description
PACKAGE [BODY]	Specify BODY to drop only the body of the package. Omit BODY to drop both the specification and body of the package.
[*Owner*.]*PackageName*	Name of the package to be dropped.

Description

- When you drop only the body of the package, TimesTen does not invalidate dependent objects. However, you cannot execute one of the procedures or stored functions declared in the package specification until you recreate the package body.
- TimesTen invalidates any objects that depend on the package specification. If you subsequently reference one of these objects, then TimesTen tries to recompile the object and returns an error if you have not recreated the dropped package.
- Do not use this statement to remove a single object from the package. Instead, recreate the package without the object using the CREATE PACKAGE and CREATE PACKAGE BODY statements with the OR REPLACE clause.

- To use the DROP PACKAGE [BODY] statement, you must have PL/SQL enabled in your database. If you do not have PL/SQL enabled in your database, TimesTen returns an error.

Example

The following statement drops the body of package samplePackage:

```
Command> DROP PACKAGE BODY SamplePackage;
Package body dropped.
```

To drop both the specification and body of package samplepackage:

```
Command> DROP PACKAGE samplepackage;
Package dropped.
```

See also

CREATE PACKAGE

DROP PROCEDURE

The DROP PROCEDURE statement removes a standalone stored procedure from the database. Do not use this statement to remove a procedure that is part of a package.

Required privilege

No privilege is required for the procedure owner.

DROP ANY PROCEDURE for another user's procedure.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
DROP PROCEDURE [Owner.]ProcedureName
```

Parameters

Parameter	Description
[*Owner*.]*ProcedureName*	Name of the procedure to be dropped.

Description

- When you drop a procedure, TimesTen invalidates objects that depend on the dropped procedure. If you subsequently reference one of these objects, TimesTen attempts to recompile the object and returns an error message if you have not recreated the dropped procedure.
- Do not use this statement to remove a procedure that is part of a package. Either drop the package or redefine the package without the procedure using the CREATE PACKAGE statement with the OR REPLACE clause.
- To use the DROP PROCEDURE statement, you must have PL/SQL enabled in your database. If you do not have PL/SQL enabled in your database, an error is thrown.

Examples

The following statement drops the procedure myproc and invalidates all objects that depend on myproc:

```
Command> DROP PROCEDURE myproc;
Procedure dropped.
```

If PL/SQL is not enabled in your database, TimesTen returns an error:

```
Command> DROP PROCEDURE myproc;

 8501: PL/SQL feature not installed in this TimesTen databaseThe command failed.
```

See also

CREATE PROCEDURE

DROP PROFILE

The DROP PROFILE statement removes a profile from the database.

Required privilege

ADMIN

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

DROP PROFILE *profile* [CASCADE]

Parameters

Parameter	Description
profile	Name of the profile to be dropped.
CASCADE	Specify CASCADE to de-assign the profile from any users to whom the profile is assigned. TimesTen reassigns the DEFAULT profile to such users. You must specify CASCADE to drop a profile that is currently assigned to users.

Description

- Use this statement to drop an existing profile. You cannot drop the DEFAULT profile. See "CREATE PROFILE" for information on the DEFAULT profile.
- If you create a profile that is not currently assigned to a user, you do not need to specify CASCADE to drop the profile. If, however, the profile is currently assigned to a user, you must specify CASCADE to drop the profile.

Example

This example creates the test_profile profile and the test_profile_assign_to_user profile. It then creates the test_user user and assigns the test_profile_assign_to_user profile to that user. The example attempts to drop the test_profile profile. The operation succeeds as there are no users assigned to this profile. The example then attempts to drop the test_profile_assign_to_user profile. The operation succeeds if CASCADE is specified. After the test_profile_assign_to_user profile is dropped, the test_user user is assigned the DEFAULT profile.

1. Create the test_profile profile. Set FAILED_LOGIN_ATTEMPTS to a value of 5.

```
Command> CREATE PROFILE test_profile LIMIT FAILED_LOGIN_ATTEMPTS 5;

Profile created.
```

2. Create the test_profile_assign_to_user profile. Set FAILED_LOGIN_ATTEMPTS to a value of 3.

```
Command> CREATE PROFILE test_profile_assign_to_user
       LIMIT FAILED_LOGIN_ATTEMPTS 3;

Profile created.
```

3. Create the test_user user and assign the test_profile_assign_to_user profile to this user.

```
Command> CREATE USER test_user identified by test_user_pwd
       PROFILE test_profile_assign_to_user;

User created.
```

4. Drop the test_profile profile. The DROP PROFILE operation succeeds. There are no users assigned to this test_profile profile.

```
Command> DROP PROFILE test_profile;

Profile dropped.
```

5. Attempt to drop the test_profile_assign_to_user profile. The DROP PROFILE operation fails. There is a user assigned to this profile. Repeat the DROP PROFILE operation again, but this time specify CASCADE. The DROP PROFILE operation succeeds.

```
Command> DROP PROFILE test_profile_assign_to_user;
15178: Profile TEST_PROFILE_ASSIGN_TO_USER has users assigned, cannot drop without CASCADE
The command failed.

Command> DROP PROFILE test_profile_assign_to_user CASCADE;

Profile dropped.
```

6. Query the DBA_USERS system view to verify that the test_user user has been assigned the DEFAULT profile.

```
Command> SELECT profile FROM dba_users WHERE username = 'TEST_USER';

PROFILE
< DEFAULT >
1 row found.
```

See also

CREATE PROFILE
ALTER PROFILE
CREATE USER
ALTER USER

DROP REPLICATION

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The DROP REPLICATION statement destroys a classic replication scheme and removes it from the executing database.

Required privilege

ADMIN

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
DROP REPLICATION [Owner.]ReplicationSchemeName
```

Parameters

Parameter	Description
[*Owner.*]*ReplicationSchemeName*	Name assigned to the classic replication scheme.

Description

Dropping the last replication scheme on a database does not delete the replicated tables. These tables exist and persist at a database whether any replication schemes are defined.

Examples

The following statement erases the executing database's knowledge of a classic replication scheme, r:

```
DROP REPLICATION r;
```

See also

ALTER REPLICATION
CREATE REPLICATION

DROP SEQUENCE

The DROP SEQUENCE statement removes an existing sequence number generator.

If the sequence is replicated across an active standby pair and if DDL_REPLICATION_LEVEL is 3 or greater, the DROP SEQUENCE statement drops the sequence from the active standby pair for all databases in the replication scheme. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Required Privilege

No privilege is required for the sequence owner.

DROP ANY SEQUENCE for another user's sequence.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
DROP SEQUENCE [Owner.]SequenceName
```

Parameters

Parameter	Description
[*Owner*.]*SequenceName*	Name of the sequence number generator.

Description

- Sequences can be dropped while they are in use.
- If you are using TimesTen Scaleout, you can modify the batch value with the ALTER SEQUENCE statement. Otherwise, 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.
- If the sequence is part of a replication scheme, use the ALTER REPLICATION statement to drop the sequence from the replication scheme. Then use the DROP SEQUENCE statement to drop the sequence.

Examples

The following statement drops mysequence:

```
DROP SEQUENCE mysequence;
```

See also

CREATE SEQUENCE

DROP SYNONYM

The DROP SYNONYM statement removes a synonym from the database.

If the synonym is replicated across an active standby pair and if DDL_REPLICATION_LEVEL is 2 or greater, the DROP SYNONYM statement drops the synonym from the active standby pair for all databases in the replication scheme. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Required Privilege

No privilege is required to drop the private synonym by its owner. The DROP ANY SYNONYM privilege is required to drop another user's private synonym.

The DROP PUBLIC SYNONYM privilege is required to drop a PUBLIC synonym.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

To drop a private synonym, use the following syntax:

```
DROP SYNONYM [Owner.]SynonymName
```

To drop a public synonym:

```
DROP PUBLIC SYNONYM SynonymName
```

Parameters

Parameter	Description
PUBLIC	Specify PUBLIC to drop a public synonym.
Owner	Optionally, specify the owner for a private synonym. If you omit the owner, the private synonym must exist in the current user's schema.
SynonymName	Specify the name of the synonym to be dropped.

Examples

Drop the public synonym pubemp:

```
DROP PUBLIC SYNONYM pubemp;
Synonym dropped.
```

Drop the private synjobs synonym:

```
DROP SYNONYM synjobs;
Synonym dropped.
```

As user terry with DROP ANY SYNONYM privilege, drop the private syntab synonym owned by ttuser.

```
DROP SYNONYM ttuser.syntab;
Synonym dropped.
```

See also

CREATE SYNONYM

DROP TABLE

The DROP TABLE statement removes the specified table, including any hash indexes and any range indexes associated with it.

Required Privilege

No privilege is required for the table owner.

DROP ANY TABLE for another user's table.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
DROP TABLE [Owner.]TableName
```

Parameters

Parameter	Description
[*Owner*.]*TableName*	Identifies the table to be dropped.

Description

- If you attempt to drop a table that is in use, an error results.
- If DROP TABLE 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 can do one of the following:
  - Use the DROP REPLICATION statement to drop the replication scheme before issuing the DROP TABLE statement.
  - If DDL_REPLICATION_LEVEL is 2 or greater, the DROP TABLE statement drops the table from the active standby pair for all databases in the replication scheme.

    If DDL_REPLICATION_LEVEL is 1, stop the replication agent and use the ALTER ACTIVE STANDBY PAIR ... EXCLUDE TABLE statement to exclude the table from the replication scheme. Then use the DROP TABLE statement to drop the table.

    See "Making DDL Changes in an Active Standby Pair" in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.
- A temporary table cannot be dropped by a connection if some other connection has some non-empty instance of the table.

Examples

```
CREATE TABLE vendorperf
  (ordernumber INTEGER,
   delivday TT_SMALLINT,
   delivmonth TT_SMALLINT,
   delivyear TT_SMALLINT,
   delivqty TT_SMALLINT,
   remarks VARCHAR2(60));
CREATE UNIQUE INDEX vendorperfindex ON vendorperf (ordernumber);
```

The following statement drops the table and index.

```
DROP TABLE vendorperf;
```

DROP USER

The DROP USER statement removes a user from the database.

Required privilege

ADMIN

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
DROP USER user
```

Parameters

Parameter	Description
user	Name of the user that is being dropped from the database.

Description

Before you can drop a user:

- The user must exist either internally or externally in the database.
- You must drop objects that the user owns.
- When replication is configured, this statement is replicated.

Examples

Drop user terry from the database:

```
DROP USER terry;
User dropped.
```

See also

CREATE USER
ALTER USER
GRANT
REVOKE

DROP VIEW

The DROP VIEW statement removes the specified view.

If the view is replicated across an active standby pair and if DDL_REPLICATION_LEVEL is 3 or greater, the DROP VIEW statement drops the view from the active standby pair for all databases in the replication scheme. See Making DDL Changes in an Active Standby Pair in the *Oracle TimesTen In-Memory Database Replication Guide* for more information.

Required Privilege

View owner or DROP ANY VIEW (if not owner)

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
DROP VIEW [Owner.]ViewName
```

Parameters

Parameter	Description
[*Owner*.]*ViewName*	Identifies the view to be dropped.

Examples

The following statement drops the custorder view.

```
DROP VIEW custorder;
```

See Also

[CREATE VIEW](#)

FLUSH CACHE GROUP

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The FLUSH CACHE GROUP statement flushes data from TimesTen cache tables to Oracle Database tables. This statement is available only for user managed cache groups.

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 Database) is turned off. Instead of propagating every transaction upon commit, many transactions can be committed before changes are propagated to Oracle Database. For each cache instance ID, if the cache instance exists in the Oracle database, the operation in the Oracle database consists of an update. If the cache instance does not exist in the Oracle database, TimesTen inserts it.

This is useful, for example, in a shopping cart application in which many changes may be made to the cart, which uses TimesTen as a high-speed cache, before the order is committed to the master Oracle database 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 does not exist in the Oracle database table 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 the Oracle database 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 the Oracle database tables.

Required privilege

No privilege is required for the cache group owner.

FLUSH or FLUSH ANY CACHE GROUP for another user's cache group.

INSERT, DELETE, UPDATE privileges on underlying tables.

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
FLUSH CACHE GROUP [Owner.]Cache
[WHERE ConditionalExpression]
```

or

```
FLUSH CACHE GROUP [Owner.]CacheGroupName
WITH ID (ColumnValueList)
```

Parameters

Parameter	Description
[*Owner.*]*CacheGroupName*	Name of the cache group to be flushed.
WHERE *ConditionalExpression*	Use the WHERE clause to specify a search condition to qualify the target rows of the cache operation. If you use more than one table in the WHERE clause and the tables have columns with the same names, fully qualify the table names.
WITH ID *ColumnValueList*	The WITH ID clauses enables you to use primary key values to flush the cache instance. Specify *ColumnValueList* as either a list of literals or binding parameters to represent the primary key values.

Description

- WHERE clauses are generally used to apply the operation to a set of cache instances, rather than to a single cache instance or to all cache instances. The flush operation uses the WHERE clause to determine which cache instances to send to the Oracle database.
- Generally, you do not have to fully qualify the column names in the WHERE clause of the FLUSH CACHE GROUP statement. However, since TimesTen automatically generates queries that join multiple tables in the same cache group, a column must be fully qualified if there is more than one table in the cache group that contains columns with the same name. Without an owner name, all tables referenced by cache group WHERE clauses are owned by the current login name executing the cache group operation.
- When the WHERE clause is omitted, the entire contents of the cache group is flushed to the Oracle database tables. When the WHERE clause is included, it is allowed to include only the root table.
- Following the execution of a FLUSH CACHE GROUP statement, the ODBC function SQLRowCount(), the JDBC method getUpdateCount(), and the OCI function OCIAttrGet() with the OCI_ATTR_ROW_COUNT argument return the number of cache instances that were flushed.
- Use the WITH ID clause to specify binding parameters.

Restrictions

Do not use the WITH ID clause when flushing:

- Static user managed cache group with the AUTOREFRESH attribute
- AWT or SWT cache groups

Examples

```
FLUSH CACHE GROUP marketbasket;

FLUSH CACHE GROUP marketbasket
WITH ID(10);
```

See also

CREATE CACHE GROUP

GRANT

The GRANT statement assigns one or more privileges to a user.

Required privilege

ADMIN to grant system privileges.

ADMIN or the object owner to grant object privileges.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

GRANT {*SystemPrivilege* [,...] | ALL [PRIVILEGES]} [...] TO {*user* |PUBLIC} [,...]

or

GRANT {{*ObjectPrivilege* [,...] | ALL [PRIVILEGES]} ON {[*Owner*.]*object*}[,...]} TO
{*user* | PUBLIC} [,...]

Parameters

The following parameters are for granting system privileges:

Parameter	Description
SystemPrivilege	This is the system privilege to grant. See "System Privileges" for a list of acceptable values.
ALL [PRIVILEGES]	Assigns all system privileges to the user.
user	Name of the user to whom privileges are being granted. The user name must first have been introduced to the TimesTen database by a CREATE USER statement.
PUBLIC	Specifies that the privilege is granted to all users.

The following parameters are for granting object privileges:

Parameter	Description
ObjectPrivilege	This is the object privilege to grant. See "Object Privileges" for a list of acceptable values.
ALL [PRIVILEGES]	Assigns all object privileges to the user.

Parameter	Description
[*Owner*.]*object*	*object* is the name of the object on which privileges are being granted. *Owner* is the owner of the object. If *Owner* is not specified, the user who is granting the privilege is the owner.
user	Name of the user to whom privileges are being granted. The user must exist in the database.
PUBLIC	Specifies that the privilege is granted to all users.

Description

- One or more system privileges can be granted to a user by a user with ADMIN privilege.
- One or more object privileges can be granted to a user by the owner of the object.
- One or more object privileges can be granted to a user on any object by a user with ADMIN privilege.
- To remove a privilege from a user, use the REVOKE statement.
- You cannot grant system privileges and object privileges in the same statement.
- Only one object can be specified in an object privilege statement.
- When replication is configured, this statement is replicated.

Examples

Grant the ADMIN privilege to the user terry:

```
GRANT admin TO terry;
```

Assuming the grantor has ADMIN privilege, grant the SELECT privilege to user terry on the customers table owned by user pat:

```
GRANT SELECT ON pat.customers TO terry;
```

Grant an object privilege to user terry:

```
GRANT SELECT ON emp_details_view TO terry;
```

See also

CREATE USER
ALTER USER
DROP USER
REVOKE
"The PUBLIC Role"

INSERT

The INSERT statement adds rows to a table.

The following expressions can be used in the VALUES clause of an INSERT statement:

- TO_CHAR
- TO_DATE
- *Sequence* NEXTVAL and *Sequence* CURRVAL

- CAST
- DEFAULT
- SYSDATE and GETDATE
- User and Session Functions
- Expressions
- SYSTEM_USER

Required privilege

No privilege is required for the table owner.

INSERT for another user's table.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
INSERT [hint] INTO [Owner.]TableName [(Column [,...])]
VALUES (SingleRowValues)
[RETURNING|RETURN Expression[,...] INTO DataItem[,...]]
```

The *SingleRowValues* parameter has the syntax:

```
{NULL|{?|:DynamicParameter}|{Constant}| DEFAULT}[,...]
```

Parameters

Parameter	Description
hint	Specifies a statement level optimizer hint for the INSERT statement. See "Statement Level Optimizer Hints" for information on statement level optimizer hints.
Owner	The owner of the table into which data is inserted.
TableName	Name of the table into which data is inserted.
Column	Each column in this list is assigned a value from *SingleRowValues*. If you omit one or more of the table's columns from this list, then the value of the omitted column in the inserted row is the column default value as specified when the table was created or last altered. If any omitted column has a NOT NULL constraint and has no default value, then the database returns an error. If you omit a list of columns completely, then you must specify values for all columns in the table.
? *:DynamicParameter*	Placeholder 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" for information on constants.
DEFAULT	Specifies that the column should be updated with the default value.
Expression	Valid expression syntax. See Expressions for information on expressions.
DataItem	Host variable or PL/SQL variable that stores the retrieved *Expression* value.

Description

- If you omit any of the table's columns from the column name list, the INSERT statement places the default value in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns and there is no default value, the INSERT statement 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.
- The INSERT operation fails if it violates a foreign key constraint. See "CREATE TABLE" for a description of the foreign key constraint.
- Restrictions on the RETURNING clause:
  - Each *Expression* must be a simple expression. Aggregate functions are not supported.
  - You cannot return a sequence number into an OUT parameter.
  - ROWNUM and subqueries cannot be used in the RETURNING clause.
  - Parameters in the RETURNING clause cannot be duplicated anywhere in the INSERT statement.
  - In PL/SQL, you cannot use a RETURNING clause with a WHERE CURRENT operation.

Examples

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');
```

For dynamic parameters :pno and :pname, values are supplied at runtime.

```
INSERT INTO purchasing.parts (partnumber, partname)
  VALUES (:pno, :pname);
```

Return the annual salary and job_id of a new employee. Declare the variables sal and jobid with the same data types as salary and job_id. Insert the row into employees. Print the variables for verification.

```
Command> VARIABLE sal12 NUMBER(8,2);
Command> VARIABLE jobid VARCHAR2(10) INLINE NOT NULL;

Command> INSERT INTO employees(employee_id, last_name, email, hire_date,
           job_id, salary)
         VALUES (211,'Doe','JDOE',sysdate,'ST_CLERK',2400)
         RETURNING salary*12, job_id INTO :sal12,:jobid;
1 row inserted.

PRINT sal12 jobid;
SAL12              : 28800
JOBID              : ST_CLERK
```

See also

CREATE TABLE
INSERT...SELECT

INSERT...SELECT

The INSERT...SELECT statement inserts the results of a query into a table.

Required privilege

No privilege is required for the object owner.

INSERT and SELECT for another user's object.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
INSERT INTO [Owner.]TableName [(ColumnName [,...])] InsertQuery
```

Parameters

Parameter	Description
[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...SELECT statement 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...SELECT statement 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". You can specify a statement level optimizer hint after the SELECT verb. See "Statement Level Optimizer Hints" for more information on statement level optimizer hints.

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. See "Using CURRVAL and NEXTVAL in TimesTen Classic " for more details.
- 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*.
- A RETURNING clause cannot be used in an INSERT...SELECT statement.
- The SELECT subquery in a UNION, UNION ALL, MINUS, or INTERSECT must have the same number of projected expressions.

Examples

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 Oracle database tables into a TimesTen cache group.

Required Privilege

No privilege is required for the cache group owner.

LOAD or LOAD ANY CACHE GROUP for another user's cache group.

INSERT, DELETE, UPDATE privileges on underlying tables.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
LOAD CACHE GROUP [Owner.]GroupName
[WHERE ConditionalExpression]
COMMIT EVERY n ROWS
[PARALLEL NumThreads [READERS NumReaders]]
```

or

```
LOAD CACHE GROUP [Owner.]GroupName
WITH ID (ColumnValueList)
```

> **Note**
>
> The WITH ID clause is not supported in TimesTen Scaleout.

Parameters

Parameter	Description
[*Owner*.]*GroupName*	Name assigned to the cache group.
WHERE *ConditionalExpression*	Use the WHERE clause to specify a search condition to qualify the target rows of the cache operation. If you use more than one table in the WHERE clause and the tables have columns with the same names, fully qualify the table names.

Parameter	Description
COMMIT EVERY *n* ROWS	Use the COMMIT EVERY *n* ROWS clause to indicate the frequency (based on the number of rows that are loaded into the cache group) at which a commit is issued during the load operation. This clause is required if you do not specify the WITH ID clause. *n* represents the number of rows that are loaded into the cache group before the transaction is committed. Express *n* as an integer (where *n* >= 0). If you specify 0 for *n*, the LOAD CACHE GROUP statement is executed as one transaction.
[PARALLEL *NumThreads*]	Provides parallel loading for cache group tables. Specifies the number of loading threads to run concurrently. One thread performs the bulk fetch from the Oracle database and the other threads (*NumThreads* - 1 threads) perform the inserts into TimesTen. Each thread uses its own connection or transaction. The minimum value for *NumThreads* is 2. The maximum value is 10. If you specify a value greater than 10, TimesTen assigns the value 10.
[READERS *NumReaders*]	This option specifies the total number of threads from the *NumThreads* parameter to use for bulk fetching from the Oracle database. For example, if you specify a *NumThreads* parameter of 8 and a readers option of 3, then 3 threads are used to bulk fetch data from the Oracle database and 5 threads are used to load data into the TimesTen database. Express *NumReaders* as an integer where 0 < *NumReaders* < *NumThreads*.
WITH ID *ColumnValueList*	The WITH ID clauses enables you to use primary key values to load the cache instance. Specify *ColumnValueList* as either a list of literals or binding parameters to represent the primary key values. The WITH ID clause is not supported in TimesTen Scaleout.

Description

- LOAD CACHE GROUP loads all new cache instances from the Oracle database that satisfy the cache group definition and are not yet present in the cache group.
- Before issuing the LOAD CACHE GROUP statement, ensure that the replication agent is running if the cache group is replicated or is an AWT cache group. Make sure the cache agent is running.
- LOAD CACHE GROUP is executed in its own transaction, and must be the first operation in a transaction.
- LOAD CACHE GROUP only loads new (inserted) rows on the Oracle database tables into the corresponding TimesTen cache tables.
- Errors cause a rollback. When cache instances are committed periodically, errors abort the remainder of the load. The load is rolled back to the last commit.
- If the LOAD CACHE GROUP statement fails when you specify COMMIT EVERY *n* ROWS (where *n* >= 0), the content of the target cache group could be in an inconsistent state since some loaded rows are already committed. Some cache instances may be partially loaded. Use the UNLOAD CACHE GROUP statement to unload the cache group, then reload the cache group.
- Generally, you do not have to fully qualify the column names in the WHERE clause of the LOAD CACHE GROUP statement. However, since TimesTen automatically generates queries

that join multiple tables in the same cache group, a column must be fully qualified if there is more than one table in the cache group that contains columns with the same name.

- When loading a read-only cache group:
  - The AUTOREFRESH state must be paused.
  - The LOAD CACHE GROUP statement cannot have a WHERE clause (except on a dynamic cache group).
  - The cache group must be empty.
- The automatic refresh state of a cache group may change after a LOAD CACHE GROUP operation completes. See Loading and Refreshing a Dynamic Cache Group with Autorefresh in the *Oracle TimesTen In-Memory Database Cache Guide* for information.
- Following the execution of a LOAD CACHE GROUP statement, the ODBC function SQLRowCount(), the JDBC method getUpdateCount(), and the OCI function OCIAttrGet() with the OCI_ATTR_ROW_COUNT argument return the number of cache instances that were loaded.
- Use the WITH ID clause as follows:
  - In place of the WHERE clause for faster loading of the cache instance
  - To specify binding parameters
  - To roll back the load transaction upon failure

Restrictions

- The LOAD CACHE GROUP...WITH ID clause is not supported in TimesTen Scaleout.
- Do not reference child tables in the WHERE clause.
- Do not specify the PARALLEL clause in the following circumstances:
  - With the WITH ID clause
  - With the COMMIT EVERY 0 ROWS clause
  - When database level locking is enabled (connection attribute LockLevel is set to 1)
- Do not use the WITH ID clause when loading these types of cache groups:
  - Static read-only cache group
  - Static user managed cache group with the autorefresh attribute
  - User managed cache group with the AUTOREFRESH and PROPAGATE attributes
- Do not use the WITH ID clause with the COMMIT EVERY *n* ROWS clause.

Examples

```
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;
```

Use the HR schema to illustrate the use of the PARALLEL clause with the LOAD CACHE GROUP statement. The COMMIT EVERY *n* ROWS clause is required. Issue the CACHEGROUPS command.

You see cache group cg2 is defined and the autorefresh state is paused. Unload cache group cg2, then specify the LOAD CACHE GROUP statement with the PARALLEL clause to provide parallel loading. You see 25 cache instances loaded.

```
Command> CACHEGROUPS;

Cache Group SAMPLEUSER.CG2:

  Cache Group Type: Read Only
  Autorefresh: Yes
  Autorefresh Mode: Incremental
  Autorefresh State: Paused
  Autorefresh Interval: 1.5 Minutes

  Root Table: SAMPLEUSER.COUNTRIES
  Table Type: Read Only

  Child Table: SAMPLEUSER.LOCATIONS
  Table Type: Read Only

  Child Table: SAMPLEUSER.DEPARTMENTS
  Table Type: Read Only

1 cache group found.

Command> UNLOAD CACHE GROUP cg2;
25 cache instances affected.
Command> COMMIT;
Command> LOAD CACHE GROUP cg2 COMMIT EVERY 10 ROWS PARALLEL 2;
25 cache instances affected.
Command> COMMIT;
```

The following example loads only the cache instances for customers whose customer number is greater than or equal to 5000 into the TimesTen cache tables in the new_customers cache group from the corresponding Oracle database tables:

```
LOAD CACHE GROUP new_customers WHERE (oratt.customer.cust_num >= 5000) COMMIT
EVERY 256 ROWS;
```

See also

- REFRESH CACHE GROUP
- UNLOAD CACHE GROUP

MERGE

This statement is not supported in TimesTen Scaleout.

In TimesTen Classic:

The MERGE statement enables 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 which 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.

Required privilege

No privilege is required for the owner of the target table and the source table.

INSERT or UPDATE on a target table owned by another user and SELECT on a source table owned by another user.

Usage with TimesTen Scaleout

This statement is not supported with TimesTen Scaleout.

SQL syntax

```
MERGE [hint] INTO [Owner.]TargetTableName [Alias] USING
   {[Owner.]SourceTableName|(Subquery)}[Alias] ON (Condtion)
   {MergeUpdateClause MergeInsertClause |
    MergeInsertClause MergeUpdateClause |
    MergeUpdateClause | MergeInsertClause
   }
```

The syntax for *MergeUpdateClause* is as follows:

```
WHEN MATCHED THEN UPDATE SET SetClause [WHERE Condition1]
```

The syntax for *MergeInsertClause* is as follows:

```
WHEN NOT MATCHED THEN INSERT [Columns [,...]] VALUES
   ( {{Expression | DEFAULT|NULL} [,...] }) [WHERE Condition2]
```

Parameters

Parameter	Description
hint	Specifies a statement level optimizer hint for the MERGE statement. See "Statement Level Optimizer Hints" for information on statement level optimizer hints.
[*Owner*.]*TargetTableName*	Name of the target table. This is the table in 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 to use joins or aggregates. Optionally, you can specify an alias for the table name or the subquery.
ON (*Condition*)	Specify the condition 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.

Parameter	Description
SET *SetClause*	Clause used with the UPDATE statement. See "UPDATE" for information on the UPDATE statement.
[WHERE *Condition1*]	For each row that matches the ON (*Condition*), *Condition1* is evaluated. If the condition is true when evaluated, 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. See "INSERT" for information on the INSERT statement.
[WHERE *Condition2*]	If specified, *Condition2* is evaluated. If the condition is true when evaluated, 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* and *MergeInsertClause* together or separately. If you specify both, they can be in either order.
- If DUAL is the only table specified in the USING clause and it is not referenced elsewhere in the MERGE statement, specify DUAL as a simple table rather than using it in a subquery. In this simple case, to help performance, specify a key condition on a unique index of the target table in the ON clause.
- 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.
  - The RETURNING clause cannot be used in a MERGE statement.

Examples

In this example, dual is specified as a simple table. There is a key condition on the UNIQUE index of the target table specified in the ON clause. The DuplicateBindMode attribute is set to 1 in this example. (The default is 0.)

```
Command> CREATE TABLE mergedualex (col1 TT_INTEGER NOT NULL,
         col2 TT_INTEGER, PRIMARY KEY (col1));
Command> MERGE INTO mergedualex USING dual ON (col1 = :v1)
       WHEN MATCHED THEN UPDATE SET col2 = col2 + 1
```

```
    WHEN NOT MATCHED THEN INSERT VALUES (:v1, 1);
Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1  'V1' (TT_INTEGER) > 10
1 row merged.
Command> SELECT * FROM mergedualex;
< 10, 1 >
1 row found.
```

In this example, a table called contacts is created with columns employee_id and manager_id. One row is inserted into contacts with values 101 and NULL for employee_id and manager_id, respectively. The MERGE statement is used to insert rows into contacts 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 contacts:

```
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 replaces data in the TimesTen cache tables with the most current committed data from the Oracle database cached tables.

Required Privilege

CREATE SESSION on the Oracle Database schema and SELECT on the Oracle Database tables.

No privilege for the cache group is required for the cache group owner.

REFRESH or REFRESH ANY CACHE GROUP for another user's cache group.

INSERT, DELETE, UPDATE privileges on underlying tables.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL Syntax

```
REFRESH CACHE GROUP [Owner.]GroupName
[WHERE ConditionalExpression]
COMMIT EVERY n ROWS
[PARALLEL NumThreads]
```

or

```
REFRESH CACHE GROUP [Owner.]GroupName
WITH ID (ColumnValueList)
```

> **Note**
>
> The WITH ID clause is not supported in TimesTen Scaleout.

Parameters

Parameter	Description
[*Owner*.]*GroupName*	Name assigned to the cache group.
WHERE *ConditionalExpression*	Use the WHERE clause to specify a search condition to qualify the target rows of the cache operation. If you use more than one table in the WHERE clause and the tables have columns with the same names, fully qualify the table names.
COMMIT EVERY *n* ROWS	Use the COMMIT EVERY *n* ROWS clause to indicate the frequency (based on the number of rows that are refreshed in the cache group) at which a commit is issued during the refresh operation. This clause is required if you do not specify the WITH ID clause. *n* represents the number of rows that are refreshed before the transaction is committed. Express *n* as an integer (where *n* >= 0). If you specify 0 for *n*, the REFRESH CACHE GROUP statement is executed as one transaction.
[PARALLEL *NumThreads*]	Provides parallel loading for cache group tables. Specifies the number of loading threads to run concurrently. One thread performs the bulk fetch from the Oracle database and the other threads (*NumThreads* -1 threads) perform the inserts into TimesTen. Each thread uses its own connection or transaction. The minimum value for *NumThreads* is 2. The maximum value is 10. If you specify a value greater than 10, TimesTen assigns the value 10.
WITH ID *ColumnValueList*	The WITH ID clauses enables you to use primary key values to refresh the cache instance. Specify *ColumnValueList* as either a list of literals or binding parameters to represent the primary key values. The WITH ID clause is not supported in TimesTen Scaleout.

Description

- A REFRESH CACHE GROUP statement must be executed in its own transaction.
- Before issuing the REFRESH CACHE GROUP statement, ensure that the replication agent is running if the cache group is replicated or is an AWT cache group. Make sure the cache agent is running.
- The REFRESH CACHE GROUP statement replaces data in the TimesTen cached tables with the most current committed data from the cached Oracle database tables, including data that already exists in the TimesTen cached tables. For an explicitly loaded cache group, a refresh operation is equivalent to issuing an UNLOAD CACHE GROUP statement followed by a LOAD CACHE GROUP statement. Operations on all rows in the Oracle database tables including inserts, updates, and deletes are applied to the cache tables. For dynamic cache groups, a refresh operation refreshes only rows that are updated or deleted on the Oracle database tables into the cache tables. For more information on explicitly loaded and dynamic cache groups, see Transmitting Changes Between the TimesTen and Oracle Databases in *Oracle TimesTen In-Memory Database Cache Guide*.
- When refreshing a read-only cache group:
  - The AUTOREFRESH state must be paused.
  - If the cache group is a read-only dynamic cache group, do not use the PARALLEL clause.
- If the automatic refresh state of a cache group (dynamic or explicitly loaded) is PAUSED, the state is changed to ON after an unconditional REFRESH CACHE GROUP statement issued on the cache group completes.
- If the automatic refresh state of a dynamic cache group is PAUSED, the state remains PAUSED after a REFRESH CACHE GROUP...WITH ID statement completes.
- Generally, you do not have to fully qualify the column names in the WHERE clause of the REFRESH CACHE GROUP statement. However, since TimesTen automatically generates queries that join multiple tables in the same cache group, a column must be fully qualified if there is more than one table in the cache group that contains columns with the same name.
- If the REFRESH CACHE GROUP statement fails when you specify COMMIT EVERY *n* ROWS (where *n* >= 0), the content of the target cache group could be in an inconsistent state since some loaded rows are already committed. Some cache instances may be partially loaded. Use the UNLOAD CACHE GROUP statement to unload the cache group, then use the LOAD CACHE GROUP statement to reload the cache group.
- Following the execution of a REFRESH CACHE GROUP statement, the ODBC function SQLRowCount(), the JDBC method getUpdateCount(), and the OCI function OCIAttrGet() with the OCI_ATTR_ROW_COUNT argument return the number of cache instances that were refreshed.
- Use the WITH ID clause:
  - In place of the WHERE clause for faster refreshing of the cache instance
  - To specify binding parameters
  - To roll back the refresh transaction upon failure

Restrictions

- The REFRESH CACHE GROUP...WITH ID clause is not supported in TimesTen Scaleout.

- Do not specify the PARALLEL clause:
  - With the WITH ID clause
  - With the COMMIT EVERY *n* ROWS clause
  - When database level locking is enabled (connection attribute LockLevel is set to 1)
  - For read-only dynamic cache groups
- Do not use the WITH ID clause when refreshing these types of cache groups:
  - Static read-only cache groups
  - Static user managed cache groups with the autorefresh attribute
  - User managed cache groups with the autorefresh and propagate attributes
- Do not use the WITH ID clause with the COMMIT EVERY *n* ROWS clause.
- Do not use the WHERE clause with dynamic or read-only cache groups.

Examples

```
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;
```

Use the HR schema to illustrate the use of the PARALLEL clause with the REFRESH CACHE GROUP statement. The COMMIT EVERY *n* ROWS is required. Issue the CACHEGROUPS command. You see cache group cg2 is defined and the autorefresh state is paused. Specify the REFRESH CACHE GROUP statement with the PARALLEL clause to provide parallel loading. You see 25 cache instances refreshed.

```
Command> CACHEGROUPS;

Cache Group SAMPLEUSER.CG2:

  Cache Group Type: Read Only
  Autorefresh: Yes
  Autorefresh Mode: Incremental
  Autorefresh State: Paused
  Autorefresh Interval: 1.5 Minutes

  Root Table: SAMPLEUSER.COUNTRIES
  Table Type: Read Only

  Child Table: SAMPLEUSER.LOCATIONS
  Table Type: Read Only

  Child Table: SAMPLEUSER.DEPARTMENTS
  Table Type: Read Only

1 cache group found.
Command> REFRESH CACHE GROUP cg2 COMMIT EVERY 20 ROWS PARALLEL 2;
25 cache instances affected.
```

See also

[ALTER CACHE GROUP]
[CREATE CACHE GROUP]

- DROP CACHE GROUP
- FLUSH CACHE GROUP
- LOAD CACHE GROUP
- UNLOAD CACHE GROUP

REVOKE

The REVOKE statement removes one or more privileges from a user.

Required privilege

ADMIN to revoke system privileges.

ADMIN or object owner to revoke object privileges.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

REVOKE {*SystemPrivilege* [,...] | ALL [PRIVILEGES]} FROM {*User* |PUBLIC} [,...]

or

REVOKE {{*ObjectPrivilege* [,...] | ALL [PRIVILEGES]} ON {[*Owner*.*Object*}} [,...]
FROM {*user* | PUBLIC}[,...]

Parameters

The following parameters are for revoking system privileges:

Parameter	Description
SystemPrivilege	This is the system privilege to revoke. See "System Privileges" for a list of acceptable values.
ALL [PRIVILEGES]	Revokes all system privileges from the user.
User	Name of the user from whom privileges are being revoked. The user name must first have been introduced to the TimesTen database by a CREATE USER statement.
PUBLIC	Specifies that the privilege is revoked for all users.

The following parameters are for revoking object privileges:

Parameter	Description
ObjectPrivilege	This is the object privilege to revoke. See "Object Privileges" for a list of acceptable values.
ALL [PRIVILEGES]	Revokes all object privileges from the user.
User	Name of the user from whom privileges are to be revoked. The user name must first have been introduced to the TimesTen database through a CREATE USER statement.
[*Owner*.]*Object*	*Object* is the name of the object on which privileges are being revoked. *Owner* is the owner of the object. If *Owner* is not specified, then the user who is revoking the privilege is known as the owner.

Parameter	Description
PUBLIC	Specifies that the privilege is revoked for all users.

Description

- Privileges on objects cannot be revoked from the owner of the objects.
- Any user who can grant a privilege can revoke the privilege even if they were not the user who originally granted the privilege.
- Privileges must be revoked at the same level they were granted. You cannot revoke an object privilege from a user who has the associated system privilege. For example, if you grant SELECT ANY TABLE to a user and then try to revoke SELECT ON BOB.TABLE1, the revoke fails unless you have specifically granted SELECT ON BOB.TABLE1 in addition to SELECT ANY TABLE.
- If a user has been granted all system privileges, you can revoke a specific privilege. For example, you can revoke ALTER ANY TABLE from a user who has been granted all system privileges.
- If a user has been granted all object privileges, you can revoke a specific privilege on a specific object from the user. For example, you can revoke the DELETE privilege on table CUSTOMERS from user TERRY even if TERRY has previously been granted all object privileges.
- You can revoke all privileges from a user even if the user has not previously been granted all privileges.
- You cannot revoke a specific privilege from a user who has not been granted the privilege.
- You cannot revoke privileges on objects owned by a user.
- You cannot revoke system privileges and object privileges in the same statement.
- You can specify only one object in an object privilege statement.
- Revoking the SELECT privilege on a detail table or a system privilege that includes the SELECT privilege from user2 on a detail table owned by user1 causes associated materialized views owned by user2 to be marked invalid.
- When replication is configured, this statement is replicated.

Examples

Revoke the ADMIN and DDL privileges from the user terry:

```
REVOKE admin, ddl FROM terry;
```

Assuming the revoker has ADMIN privilege, revoke the UPDATE privilege from terry on the customers table owned by pat:

```
REVOKE update ON pat.customers FROM terry;
```

See also

ALTER USER
CREATE USER
DROP USER
GRANT

ROLLBACK

Use the ROLLBACK statement to undo work done in the current transaction.

Required privilege

None

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
ROLLBACK [WORK]
```

Parameters

The ROLLBACK statement enables the following optional keyword:

Parameter	Description
[WORK]	Optional clause supported for compliance with the SQL standard. ROLLBACK and ROLLBACK WORK are equivalent.

Description

When the PassThrough connection attribute is specified with a value greater than zero, the Oracle database transaction will also be rolled back.

A rollback closes all open cursors.

Examples

Insert a row into the regions table of the HR schema and then roll back the transaction. First set AUTOCOMMIT to 0:

```
Command> SET AUTOCOMMIT 0;
Command> INSERT INTO regions VALUES (5,'Australia');
1 row inserted.
Command> SELECT * FROM regions;
< 1, Europe >
< 2, Americas >
< 3, Asia >
< 4, Middle East and Africa >
< 5, Australia >
5 rows found.
Command> ROLLBACK;
Command> SELECT * FROM regions;
< 1, Europe >
< 2, Americas >
< 3, Asia >
< 4, Middle East and Africa >
4 rows found.
```

See also

COMMIT

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*, *result set*, or *query result*.

Required privilege

No privilege is required for the object owner.

SELECT for another user's object.

SELECT...FOR UPDATE also requires UPDATE privilege for another user's object.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

The general syntax for a SELECT statement is the following:

```
[WithClause] SELECT [hint][FIRST NumRows | ROWS m TO n] [ALL | DISTINCT] SelectList
  FROM TableSpec [,...]
  [WHERE SearchCondition]
  [GROUP BY GroupByClause [,...] [HAVING SearchCondition]]
  [ORDER BY OrderByClause [,...]]
  [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 as follows:

```
SELECT [hint] [ROWS m TO n] [ALL] SelectList
  FROM TableSpec [,...]
    [WHERE SearchCondition]
    [GROUP BY GroupByClause [,...] [HAVING SearchCondition] [,...]]
  {UNION [ALL] | MINUS | INTERSECT}
  SELECT [ROWS m TO n] [ALL] SelectList
    FROM TableSpec [,...]
     [WHERE SearchCondition]
     [GROUP BY GroupByClause [,...] [HAVING SearchCondition [,...] ] ]
     [ORDER BY OrderByClause [,...] ]
```

The syntax for *OrderByClause* is as follows:

```
{ColumnID|ColumnAlias|Expression} [ASC|DESC] [NULLS { FIRST|LAST }]
```

Parameters

Parameter	Description
[*WithClause*]	The WITH clause, also known as subquery factoring, enables you to assign a name to a subquery block, which can subsequently be referenced multiple times within the top-level SELECT statement. See "WithClause" for information on the syntax for the *WithClause*.
hint	Specifies a statement level optimizer hint for the SELECT statement. See "Statement Level Optimizer Hints" for information on statement level optimizer hints.

Parameter	Description
FIRST *NumRows*	Specifies the number of rows to retrieve. *NumRows* must be either a positive INTEGER value 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 value 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 the default.
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. You cannot use SELECT ... on a LOB column.
SelectList	Specifies how the columns of the query result are to be derived. See "SelectList" for the syntax for a select list.
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. See "TableSpec" for the *TableSpec* syntax.
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 conditions 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 "Search Conditions" for more information on search conditions.
GROUP BY *GroupByClause* [,...]	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. See "GROUP BY Clause" for information on the syntax for the GROUP BY clause.
HAVING *SearchCondition*	The HAVING clause can be used in a SELECT statement to filter groups of an aggregate result. The existence of a HAVING clause in a SELECT statement turns the query into an aggregate query. All columns referenced outside the sources of aggregate functions in any clause except the WHERE clause must be included in the GROUP BY clause. Subqueries can be specified in the HAVING clause.

Parameter	Description
(+)	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 are 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 the value NULL.
ORDER BY *OrderByClause* [,...]	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. The ORDER BY clause supports column aliases, which 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.
ColumnID	Must correspond to a column in the select list. You can identify a column to be sorted by specifying its name or 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 this 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*]} [,...]
ASC|DESC	For each column designated in the ORDER BY clause, you can specify whether the sort order is to be ascending or descending. If neither ASC (ascending) nor DESC (descending) is specified, ascending order is used. All character data types are sorted according to the current value of the NLS_SORT session parameter.
NULLS { FIRST|LAST }	Valid with ORDER BY clause and is optional. If you specify ASC or DESC, NULLS FIRST or NULLS LAST must follow ASC or DESC. Specify NULLS FIRST to have rows with NULL values returned first in your ordered query. Specify NULLS LAST to have rows with NULL values returned last in your ordered query. NULLS LAST is the default when rows are returned in ascending order. NULLS FIRST is the default when rows are returned in descending order. If you specify the ORDER BY clause without the ASC or DESC clause and without the NULLS FIRST or NULLS LAST clause, the default ordering sequence is ascending NULLS LAST.

Parameter	Description
FOR UPDATE [OF [[*Owner*.] *TableName*.] *ColumnName* [,...]] [NOWAIT | WAIT *Seconds*]	FOR UPDATE • FOR UPDATE maintains a lock on a row until the end of the current transaction, regardless of isolation. All other transactions are excluded from performing any operation on that row 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. When table locks are used, the table is locked in update mode whether or not there are any qualifying rows. • If the serializable isolation level and row locking are enabled, nonqualifying rows are downgraded to shared mode. If a read-committed isolation level and row locking are turned on, nonqualifying rows are unlocked. • SELECT...FOR UPDATE locks are not blocked by SELECT locks. FOR UPDATE [OF [[*Owner*.]*TableName*.]*ColumnName* [,...]] • This mode optionally includes the name of the column or columns in the table to be locked for update. [NOWAIT | WAIT *Seconds*] • This specifies how to proceed if the selected rows are locked. It does not apply to table-level locks or database-level locks. • NOWAIT specifies that there is no waiting period for locks. 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 fractions of 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.

Parameter	Description
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 is compatible. If UNION ALL is specified, duplicate rows from both SELECT statements are retained. Otherwise, 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 as follows: • Neither SELECT statement can specify FIRST *NumRows*. • The SELECT subquery in a UNION, UNION ALL, MINUS, or INTERSECT must have the same number of projected expressions. This is true for INSERT...SELECT as well. • 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 statement 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 you use a correlation name, the correlation name must conform to the syntax rules for a basic name. All correlation names within one SELECT statement 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 select list and the WHERE clause to qualify columns from that table. See "TableSpec" for more information about correlation names.
- SELECT...FOR UPDATE is supported in a SELECT statement that specifies a subquery, but it can be specified only in the outermost query.
- 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.
- Use the SELECT...INTO statement in PL/SQL. If you use the SELECT...INTO statement outside of PL/SQL, TimesTen accepts, but silently ignores, the syntax.

Examples

This example shows the use of a column alias (max_salary) in the SELECT statement:

```
SELECT MAX(salary) AS max_salary
FROM employees
WHERE employees.hire_date > '2000-01-01 00:00:00';
< 10500 >
1 row found.
```

This example uses two tables, orders and lineitems.

The orders table and lineitems table are created as follows:

```
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';
```

Consider this query:

```
SELECT * FROM tablea, tableb
WHERE tablea.column1 = tableb.column1 AND tableb.column2 > 5
FOR UPDATE;
```

The query locks all rows in tablea where:

- The value of tablea.column1 equals at least one tableb.column1 value where tableb.column2 is greater than 5.

The query also locks all rows in tableb where:

- The value of tableb.column2 is greater than 5.
- The value of tableb.column1 equals at least one tablea.column1 value.

If no WHERE clause is specified, all rows in both tables are locked.

This example demonstrates the (+) join operator:

```
SELECT * FROM t1, t2
WHERE t1.x = t2.x(+);
```

The following 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;
```

The following query is valid:

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) AND (t3.y = 4 OR t3.y = 5);
```

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);
```

The following query is valid:

```
SELECT * FROM t1, t2, t3
WHERE t1.x = t2.x(+) AND t1.y IN (4,5);
```

The following query results in an inner join. The condition without the (+) operator is treated as an inner join condition.

```
SELECT * FROM t1, t2
WHERE t1.x = t2.x(+) AND t1.y = t2.y;
```

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 is 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;
```

For more join examples, see "JoinedTable".

The following example returns the current sequence value in the student table.

```
SELECT SEQ.CURRVAL FROM student;
```

The following query produces a derived table because it contains a SELECT statement in the FROM clause.

```
SELECT * FROM t1, (SELECT MAX(x2) maxx2 FROM t2) tab2
WHERE t1.x1 = tab2.maxx2;
```

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);
```

In the following, select all orders that have the same price as the highest price in their category.

```
SELECT * FROM orders WHERE price = (SELECT MAX(price)
FROM stock WHERE stock.cat=orders.cat);
```

The next example illustrates the use of the INTERSECT set operator. There is a department_id value 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 do not include the row in the departments table whose department_id value 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.
```

The next example illustrates the use of the MINUS set operator by combining rows returned by the first query but not the second. The row containing the NULL department_id value 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.
```

The following 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.
```

The following example locks the employees table for update and waits 10 seconds for the lock to be available. An error is returned if the lock is not acquired in 10 seconds. The first five rows are selected.

```
Command> SELECT FIRST 5 last_name FROM employees FOR UPDATE WAIT 10;
< King >
< Kochhar >
```

```
< De Haan >
< Hunold >
< Ernst >
5 rows found.
```

The next 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.
```

In the following, use the `HR` schema to illustrate the use of a subquery with the `FOR UPDATE` clause.

```
Command> SELECT employee_id, job_id FROM job_history
       WHERE (employee_id, job_id) NOT IN (SELECT employee_id, job_id
       FROM employees)
       FOR UPDATE;
< 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.
```

In the following, use a dynamic parameter placeholder for `SELECT ROWS` *m* `TO` *n* and `SELECT FIRST`.

```
Command> SELECT ROWS ? TO ? employee_id FROM employees;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 (TT_INTEGER) > 1
Enter Parameter 2 (TT_INTEGER) > 3
< 100 >
< 101 >
< 102 >
3 rows found.
Command> SELECT ROWS :a TO :b employee_id FROM employees;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 (TT_INTEGER) > 1
Enter Parameter 2 (TT_INTEGER) > 3
< 100 >
```

```
< 101 >
< 102 >
3 rows found.
Command> SELECT FIRST ? employee_id FROM employees;

Type '?' for help on entering parameter values.
Type '*' to end prompting and abort the command.
Type '-' to leave the parameter unbound.
Type '/;' to leave the remaining parameters unbound and execute the command.

Enter Parameter 1 (TT_INTEGER) > 3
< 100 >
< 101 >
< 102 >
3 rows found.
```

The following example illustrates the use of NULLS LAST in the ORDER BY clause. Query the employees table to find employees with a commission percentage greater than .30 or a commission percentage that is NULL. Select the first seven employees and order by commission_pct and last_name. Order commision_pct in descending order and use NULLS LAST to display rows with NULL values last in the query. Output commission_pct and last_name.

```
Command> SELECT FIRST 7 commission_pct,last_name
       FROM employees where commission_pct > .30
       OR commission_pct IS NULL
       ORDER BY commission_pct DESC NULLS LAST,last_name;
< .4, Russell >
< .35, King >
< .35, McEwen >
< .35, Sully >
< <NULL>, Atkinson >
< <NULL>, Austin >
< <NULL>, Baer >
7 rows found.
```

WithClause

Syntax

WithClause has the following syntax:

WITH *QueryName* AS (*Subquery*) [, *QueryName* AS (*Subquery*)] ...

Parameters

WithClause has the following parameter:

Parameter	Description
QueryName AS (*Subquery*)	Specifies an alias for a subquery that can be used multiple times within the SELECT statement.

Description

Subquery factoring provides the WITH clause that enables you to assign a name to a subquery block, which can subsequently be referenced multiple times within the main SELECT query. The query name is visible to the main query and any subquery contained in the main query.

The WITH clause can only be defined as a prefix to the main SELECT statement.

Subquery factoring is useful in simplifying complex queries that use duplicate or complex subquery blocks in one or more places. In addition, TimesTen uses subquery factoring to optimize the query by evaluating and materializing the subquery block once and providing the result for each reference in the SELECT statement.

You can specify the set operators: UNION, MINUS, INTERSECT in the main query.

Restrictions using the WITH clause:

- Do not use the WITH clause in a view or materialized view definition.
- Recursive subquery factoring is not supported.
- Do not use the WITH clause in subqueries or derived tables.
- You cannot provide a column parameter list for the query alias. For example, TimesTen does not support: WITH *w1*(*c1*,*c2*) AS ...

Example

The following example creates the query names *dept_costs* and *avg_cost* for the initial query block, then uses these names in the body of the main query.

```
Command> WITH dept_costs AS (
       SELECT department_name, SUM(salary) dept_total
       FROM employees e, departments d
       WHERE e.department_id = d.department_id
       GROUP BY department_name),
        avg_cost AS (
       SELECT SUM(dept_total)/COUNT(*) avg
       FROM dept_costs)
       SELECT * FROM dept_costs
       WHERE dept_total >
       (SELECT avg FROM avg_cost)
       ORDER BY department_name;

> DEPARTMENT_NAME DEPT_TOTAL
------------------------------
Sales 304500
Shipping 156400
```

SelectList

SQL syntax

The *SelectList* parameter of the SELECT statement has the following syntax:

```
{* | [Owner.]TableName.* |
  { Expression | [[Owner.]TableName.]ColumnName |
    [[Owner.]TableName.]ROWID | NULL
  }
  [[AS] ColumnAlias] } [,...]
```

Parameters

The *SelectList* parameter of the SELECT statement has the following parameters:

Parameter	Description
*	Includes, as columns of the query result, all columns of all tables specified in the FROM clause.

Parameter	Description
[*Owner*.]*TableName*.*	Includes all columns of the specified table in the result.
Expression	An aggregate query includes a GROUP BY clause or an aggregate function. When the select list is not an aggregate query, the column reference must reference a table in the FROM clause. A column reference in the select list of an aggregate query must reference a column list in the GROUP BY clause. If there is no GROUP BY clause, then the column reference must reference a table in the FROM clause. See "GROUP BY Clause" for more information on the 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. See "Using CURRVAL and NEXTVAL in TimesTen Classic " for more details.
[[*Owner*.]*Table*.] ROWID	Includes the ROWID pseudocolumn from the named owner's indicated table.
NULL	When NULL is specified, the default for the 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.
- TimesTen does not support subqueries in the select list.
- 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.
  - 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 qualifier 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. See "GROUP BY Clause" for details on 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, and so on.
- 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 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, as follows:
  - The WHERE clause eliminates rows before groups are formed.
  - The GROUP BY clause groups the resulting rows. See "GROUP BY Clause" for more details.
  - The select list aggregate functions are computed for each group.

Examples

In the following example, one value, the average number of days you wait for a part, is returned:

```
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 following statement.

```
SELECT partnumber, deliverydays
FROM purchasing.supplyprice
WHERE deliverydays < 20;
```

Multiple rows may be returned for a single part.

The part number and average price of each part are returned by the following statement.

```
SELECT partnumber, AVG(unitprice)
FROM purchasing.supplyprice
GROUP BY partnumber;
```

In the following example, the join returns names and locations of California suppliers. Rows are returned in ascending order by partnumber values. Rows containing duplicate part numbers are returned in ascending order by vendorname values. The FROM clause defines two correlation names (v and s), which are used in both the select list and the WHERE clause. The vendornumber column 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;
```

The following 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';
```

The next example shows how to retrieve the rowid of a specific row. The retrieved rowid value can be used later for another SELECT, DELETE, or UPDATE statement.

```
SELECT rowid
FROM purchasing.vendors
WHERE vendornumber = 123;
```

The following 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 following syntax:

*TableName*Syntax | *JoinedTable* | *DerivedTable*

*TableName*Syntax::= [*Owner*.]*TableName* [*CorrelationName*] |
([Owner.]TableName) [CorrelationName] |
([Owner.]TableName [CorrelationName])

A simple table specification has the following syntax:

[*Owner*.]*TableName* or ([*Owner*.]*TableName*)

Parameters

The *TableSpec* parameter of the SELECT statement has the following parameters:

Parameter	Description
TableNameSyntax	Identifies a table to be referenced. Parentheses are optional.
CorrelationName	*CorrelationName* specifies an alias for the immediately preceding table. When accessing columns of that table elsewhere in the SELECT statement, use the correlation name instead of the actual table name within the statement. The scope of the correlation name is the SQL statement in which it is used. The correlation name must conform to the syntax rules for a basic name. See "Basic Names" for more information. All correlation names within one statement must be unique.
JoinedTable	Specifies the query that defines the table join. See "JoinedTable" for more information.
DerivedTable	Specifies a table derived from the evaluation of a SELECT statement. No FIRST *NumRows* or ROWS *m* TO *n* clauses are allowed in this SELECT statement. See "DerivedTable" for more information.

JoinedTable

The *JoinedTable* parameter specifies a table derived from CROSS JOIN, INNER JOIN, LEFT OUTER JOIN or RIGHT OUTER JOIN.

SQL syntax

The syntax for *JoinedTable* is as follows:

{*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 as follows:

{INNER | LEFT [OUTER] | RIGHT [OUTER]}

Parameters

The *JoinedTable* parameter of the *TableSpec* clause of a SELECT statement has the following parameters:

Parameter	Description
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 following syntax: SELECT *Selectlist* FROM *Table1*, *Table2*
QualifiedJoin	Specifies that the join is 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. These are the supported join types: • INNER • LEFT [OUTER] • RIGHT [OUTER] INNER JOIN returns a result table that combines the rows from two tables that meet *SearchCondition*. LEFT OUTER JOIN returns join rows that match *SearchCondition* and rows from the first table that do not have *SearchCondition* evaluated as true with any row from the second table. RIGHT OUTER JOIN returns join rows that match *SearchCondition* and rows from the second table that do not have *SearchCondition* evaluated as true with any row from the first table.
ON *SearchCondition*	Specifies the search criteria to be used in a JOIN parameter. *SearchCondition* can refer only 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 that defines a materialized view. Thus, a joined table can be used in UNION, MINUS, INTERSECT, a subquery, a nonmaterialized view, or a derived table.
- A subquery cannot be specified in the operand of a joined table. For example, the following statement is *not* supported:

```
SELECT * FROM
  regions INNER JOIN (SELECT * FROM countries) table2
  ON regions.region_id=table2.region_id;
```

- A view can be specified as an operand of a joined table.
- A temporary table cannot be specified as an operand of a joined table.
- OUTER JOIN can be specified in two ways, either using the (+) operator in *SearchCondition* of the WHERE clause or using a JOIN table operation. The two specification methods cannot coexist in the same statement.
- Join order and grouping can be specified with a *JoinedTable* operation, but they cannot be specified with the (+) operator. For example, the following operation *cannot* be specified with the (+) operator:

  ```
  t LEFT JOIN (t2 INNER JOIN t3 ON x2=x3) ON (x1 = x2 - x3)
  ```

Examples

These examples use the regions and countries tables from the HR schema.

The following performs a left outer join.

```
SELECT * FROM regions LEFT JOIN countries
  ON regions.region_id=countries.region_id
  WHERE regions.region_id=3;

< 3, Asia, JP, Japan, 3 >
< 3, Asia, CN, China, 3 >
< 3, Asia, IN, India, 3 >
< 3, Asia, AU, Australia, 3 >
< 3, Asia, SG, Singapore, 3 >
< 3, Asia, HK, HongKong, 3 >
6 rows found.
```

You can also perform a left outer join with the (+) operator, as follows.

```
SELECT * FROM regions, countries
  WHERE regions.region_id=countries.region_id (+)
    AND regions.region_id=3;
```

The following performs a right outer join.

```
SELECT * FROM regions RIGHT JOIN countries
  ON regions.region_id=wountries.region_id
  WHERE regions.region_id=3;

< AU, Australia, 3, 3, Asia >
< CN, China, 3, 3, Asia >
< HK, HongKong, 3, 3, Asia >
< IN, India, 3, 3, Asia >
< JP, Japan, 3, 3, Asia >
< SG, Singapore, 3, 3, Asia >
6 rows found.
```

The next example performs a right outer join with the (+) operator.

```
SELECT * FROM countries, regions
     WHERE regions.region_id (+)=countries.region_id
     AND countries.region_id=3;
< JP, Japan, 3, 3, Asia >
< CN, China, 3, 3, Asia >
< IN, India, 3, 3, Asia >
```

```
< AU, Australia, 3, 3, Asia >
< SG, Singapore, 3, 3, Asia >
< HK, HongKong, 3, 3, Asia >
6 rows found.
```

Note that the right join methods produce the same rows but in a different display order. There should be no expectation of row order for join results.

The following performs an inner join.

```
SELECT * FROM regions INNER JOIN countries
  ON regions.region_id=countries.region_id
  WHERE regions.region_id=2;

< 2, Americas, US, United States of America, 2 >
< 2, Americas, CA, Canada, 2 >
< 2, Americas, BR, Brazil, 2 >
< 2, Americas, MX, Mexico, 2 >
< 2, Americas, AR, Argentina, 2 >
5 rows found.
```

The next example performs a cross join.

```
SELECT * FROM regions CROSS JOIN countries
  WHERE regions.region_id=1;

< 1, Europe, AR, Argentina, 2 >
< 1, Europe, AU, Australia, 3 >
< 1, Europe, BE, Belgium, 1 >
< 1, Europe, BR, Brazil, 2 >
...
< 1, Europe, SG, Singapore, 3 >
< 1, Europe, UK, United Kingdom, 1 >
< 1, Europe, US, United States of America, 2 >
< 1, Europe, ZM, Zambia, 4 >
< 1, Europe, ZW, Zimbabwe, 4 >
25 rows found.
```

See also

- CREATE TABLE
- INSERT
- INSERT...SELECT
- UPDATE

DerivedTable

A derived table is the result of a `SELECT` statement in the `FROM` clause, with an alias.

SQL syntax

The syntax for *DerivedTable* is as follows:

```
(Subquery) [CorrelationName]
```

Parameters

The *DerivedTable* parameter of the *TableSpec* clause of a `SELECT` statement has the following parameters:

Parameter	Description
Subquery	See "Subqueries" for information on subqueries.
CorrelationName	Optionally use *CorrelationName* to specify an alias for the derived table. It must be different from any table name referenced in the query.

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 DUAL is useful for computing a constant expression (an expression that is evaluated to a constant value) with the SELECT statement. Because DUAL has only one row, the constant is returned only once.
- *Subquery* cannot refer to a column from another derived table.
- A derived table cannot be used as a source of a joined table.
- A derived table cannot be used as a target of a DELETE or UPDATE statement.

GROUP BY Clause

Specify the GROUP BY clause if you want the database to group the selected rows based on the value of expressions for each row and return a single row of summary information for each group. If the GROUP BY clause is omitted, the entire query result is treated as one group. If this clause contains CUBE or ROLLUP, the results contain superaggregate groupings in addition to the regular groupings.

The expressions in the GROUP BY clause can do the following:

- Designate single or multiple columns.
- Include arithmetic operations, the ROWID pseudocolumn, or NULL.
- Include a date, a constant, or a dynamic parameter.
- Include ROLLUP or CUBE clauses, where the results produce superaggregate groupings in addition to the regular groupings. Superaggregate groupings are calculated subtotals and totals returned with the regular groupings in the GROUP BY clause.
- Include GROUPING SETS clause to distinguish which superaggregate groupings to produce.

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 the construct *, *TableName*.*, or *Owner.TableName*.*, 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 NULL values in a column are placed in a single group.

> **Note**
>
> To identify and potentially eliminate NULL groupings from the superaggregate groupings, use the GROUPING function. See "GROUPING" for information.

SQL Syntax

The general syntax for the GROUP BY clause is the following:

GROUP BY
{*Expression* | *RollupCubeClause* | *GroupingSetsClause* }[,...]

GroupingSetsClause::= GROUPING SETS
GroupingExpressionList | *RollupCubeClause* [,...]

RollupCubeClause
{ ROLLUP | CUBE } (*GroupingExpressionList*) }

GroupingExpressionList::=
{ *Expression* | *ExpressionList* [, { *Expression* | *ExpressionList* }] ...}

ExpressionList :: = (*Expression* [, *Expression*] ...)

Parameters

Parameter	Description
Expression	Valid expression syntax. See Expressions for more information.
RollupCubeClause	The GROUP BY clause may include one or more ROLLUP or CUBE clauses.
GroupingSetsClause	The GROUP BY clause may include one or more GROUPING SETS clauses. The GROUPING SETS clause enables you to explicitly specify which groupings of data that the database returns. See "GROUPING SETS" for more information.
GroupingExpressionList	The GROUP BY clause can contain multiple expressions or expression lists.
ROLLUP *GroupingExpressionList*	The ROLLUP clause is used to generate super aggregate rows from groups. See "ROLLUP" for more information.
CUBE *GroupingExpressionList*	The CUBE clause groups selected rows based on the values of all possible combinations of the grouping columns in the CUBE clause. See "CUBE" for more information.
ExpressionList	A list of one or more expressions, each separated by a comma.

Examples

The following GROUP BY 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.
```

Query *emp_details_view* to select the first 10 departments and managers within the department and count the number of employees in the department with the same manager. Use the GROUP BY clause to group the result by department and manager.

```
Command> columnlabels on;
Command> SELECT first 10 department_id AS DEPT, manager_id AS MGR,
       COUNT(employee_id) AS NUM_EMP
       FROM emp_details_view
       GROUP BY (department_id, manager_id)
       ORDER BY department_id, manager_id;

DEPT, MGR, NUM_EMP
< 10, 101, 1 >
< 20, 100, 1 >
< 20, 201, 1 >
< 30, 100, 1 >
< 30, 114, 5 >
< 40, 101, 1 >
< 50, 100, 5 >
< 50, 120, 8 >
< 50, 121, 8 >
< 50, 122, 8 >
10 rows found.
```

ROLLUP, CUBE and GROUPING SETS Clauses

The following definitions describe how columns can be grouped within the ROLLUP, CUBE, and GROUPING SETS clauses:

- **Grouping column**: A single column used in a GROUP BY clause. For example, in the following GROUP BY clause, X, Y, and Z are group columns.

  ```
  GROUP BY X, GROUPING SETS(Y, Z)
  ```

- **Composite Column**: A list of grouping columns inside parentheses. For example, in the following clause, (C1, C2) and (C3, C4) are composite columns.

  ```
  GROUP BY ROLLUP( (C1,C2), (C3,C4), C5);
  ```

- **Grouping**: Grouping is a single level of aggregation from within a grouping set. For example, in the following statement, (C1) and (C2, C3) are individual groupings.

  ```
  GROUP BY GROUPING SETS(C1, (C2,C3));
  ```

- **Grouping Set**: A collection of groupings inside parentheses. For example, in the following statement, (C1, (C2, C3)) and (C2, (C4, C5)) are two individual grouping sets.

  ```
  GROUP BY GROUPING SETS(C1, (C2,C3)), GROUPING SETS(C2, (C4, C5));
  ```

- **Concatenated grouping sets**: Separate multiple grouping sets with commas. The result is a cross-product of groupings from each grouping set.
- **Grand Total or Empty set column**: A grand total or empty set grouping computes aggregation by considering all rows as one group. Grand totals are automatically provided in the results for ROLLUP and CUBE clauses; however, you request the grand total in the GROUPING SETS clause by providing empty parentheses, ( ).

Duplicate grouping columns can be used in ROLLUP, CUBE or GROUPING SETS. However, it does result in duplicated result rows.

The ROLLUP, CUBE and GROUPING SETS clauses are not supported in a materialized view definition.

The following sections describe the GROUPING SETS, ROLLUP, and CUBE clauses:

- GROUPING SETS
- ROLLUP
- CUBE

GROUPING SETS

The GROUPING SETS clause enables you to explicitly specify which groupings of data that the database returns. You specify only the desired groups by enclosing them within parentheses, so the database only generates the superaggregate summaries in which you are interested.

The following statement produces three groups: one group returns results for each gender and year columns, a second for a summary superaggregate for each of the months and the last result for the grand total.

```
SELECT GENDER, YEAR, MONTH,
  SUM (NUM_OF_STUDENTS) AS TOTAL
  FROM INSTRUCTOR_SUMMARY
  GROUP BY GROUPING SETS ((GENDER, YEAR),  -- 1ST GROUP
                    (MONTH), -- 2ND GROUP
                    ()); -- 3RD GROUP
```

You can combine multiple GROUPING SETS to generate specific combinations between the multiple GROUPING SETS. The following statement contains two GROUPING SETS clauses:

```
GROUP BY GROUPING SETS (YEAR, MONTH),
      GROUPING SETS (WEEK, DAY);
```

This is equivalent to the following GROUPING SETS statement:

```
GROUP BY GROUPING SETS (YEAR, WEEK),
 (YEAR, DAY),
 (MONTH, WEEK),
 (MONTH, DAY);
```

When a GROUP BY clause has both regular grouping columns and a GROUPING SETS clause, the results are grouped by the regular grouping column as follows:

```
GROUP BY a, b GROUPING SETS(c, d);
```

This is equivalent to the following:

```
GROUP BY GROUPING SETS((a, b, c), (a, b, d));
```

The following example specifies the grouping sets of (region_name, country_name), state_province, and grand totals.

```
Command> SELECT region_name AS Region,
        country_name AS Country,
        state_province AS State,
         COUNT(employee_id) AS "Total Emp"
      FROM regions r, countries c, locations l, departments d, employees e
      WHERE r.region_id = c.region_id AND
        l.country_id = c.country_id AND
        d.location_id = l.location_id AND
        d.department_id = e.department_id
      GROUP BY grouping sets((region_name, country_name), state_province, ())
      ORDER BY region_name, state_province;
```

```
REGION, COUNTRY, STATE, TOTAL EMP
< Americas, Canada, <NULL>, 2 >
< Americas, United States of America, <NULL>, 68 >
< Europe, Germany, <NULL>, 1 >
< Europe, United Kingdom, <NULL>, 35 >
< <NULL>, <NULL>, Bavaria, 1 >
< <NULL>, <NULL>, California, 45 >
< <NULL>, <NULL>, Ontario, 2 >
< <NULL>, <NULL>, Oxford, 34 >
< <NULL>, <NULL>, Texas, 5 >
< <NULL>, <NULL>, Washington, 18 >
< <NULL>, <NULL>, <NULL>, 106 >
< <NULL>, <NULL>, <NULL>, 1 >
12 rows found.
```

ROLLUP

ROLLUP is used within the GROUP BY clause. When used with SUM, ROLLUP generates subtotals from most detailed level (all columns specified in the ROLLUP clause) to the grand total level, by removing one column at each level. These are known as superaggregate rows.

The ROLLUP clause returns the following:

- Regular aggregate rows that would be produced by GROUP BY without using ROLLUP.
- Subtotals following the grouping list specified in the ROLLUP clause. ROLLUP takes as its argument an ordered list of grouping columns. Each subtotal is created for the ordered list of grouping columns dropping the right-most grouping column until it reaches the grand total. For instance, if you specify GROUP BY ROLLUP(x, y, z), the returned superaggregate groups would be as follows: (x,y,z), (x,y), (x), ( ).

  The number of subtotals created is *n*+1 aggregate levels, where *n* is the number of grouping columns. For example, if there are three expressions (*n*=3) in the ROLLUP clause, then *n*+1 = 3+1, resulting in four groupings.
- Grand total row.

You can group columns using composite columns inside parentheses. For example, in the following statement:

```
GROUP BY ROLLUP( (a, b), (c, d), e);
```

The (a, b) and (c, d) composite columns are treated as a unit when the database produces the ROLLUP results. In this example, the grouping sets returned are as follows: ((a, b), (c, d), e), ((a, b), (c, d)), (a, b) and ().

You can execute several ROLLUP clauses within your SELECT statement, as follows:

```
SELECT C1, COUNT(*)
FROM T
GROUP BY ROLLUP(a, b), ROLLUP(c, d);
```

This is equivalent to the following statement:

```
SELECT C1, COUNT(*)
FROM T
GROUP BY GROUPING SETS((a, b),(a),()),
 GROUPING SETS((c, d),(c), ());
```

This example queries the employees table to select the first 10 departments and return the number of employees under each manager in each department. Use ROLLUP to subtotal the number of employees in each department and return a grand total of all employees in the company.

```
Command> SELECT first 10 department_id AS Dept,
         manager_id AS Mgr,
        COUNT(employee_id) AS "Total emp"
        FROM employees
        GROUP BY ROLLUP(department_id, manager_id)
        ORDER BY department_id, manager_id;

DEPT, MGR, TOTAL EMP
< 10, 101, 1 >
< 10, <NULL>, 1 >
< 20, 100, 1 >
< 20, 201, 1 >
< 20, <NULL>, 2 >
< 30, 100, 1 >
< 30, 114, 5 >
< 30, <NULL>, 6 >
< 40, 101, 1 >
< 40, <NULL>, 1 >
10 rows found.
```

The following query returns the number of employees in each region, country and state or province. The rollup returns superaggregate rows for subtotals of all employees in each state or province and in each country and a grand total for all employees in the company. By combining the region and country as its own unit (within parentheses), the rollup does not return all employees for each region.

```
Command> SELECT region_name AS Region,
         country_name AS Country,
         state_province AS State,
         COUNT(employee_id) AS "Total Emp"
        FROM regions r, countries c, locations l, departments d, employees e
        WHERE r.region_id = c.region_id
        AND l.country_id = c.country_id
        AND d.location_id = l.location_id
        AND d.department_id = e.department_id
        GROUP BY rollup((region_name, country_name), state_province)
        ORDER BY region_name;

REGION, COUNTRY, STATE, TOTAL EMP
< Americas, Canada, Ontario, 2 >
< Americas, United States of America, Texas, 5 >
< Americas, United States of America, California, 45 >
< Americas, United States of America, Washington, 18 >
< Americas, Canada, <NULL>, 2 >
< Americas, United States of America, <NULL>, 68 >
< Europe, Germany, Bavaria, 1 >
< Europe, United Kingdom, <NULL>, 1 >
< Europe, United Kingdom, Oxford, 34 >
< Europe, Germany, <NULL>, 1 >
< Europe, United Kingdom, <NULL>, 35 >
< <NULL>, <NULL>, <NULL>, 106 >
12 rows found.
```

CUBE

The `CUBE` clause groups the selected rows based on the values of all possible combinations of the grouping columns in the `CUBE` clause. It returns a single row of summary information for each group. For example, if there are three expressions (*n*=3) in the `CUBE` clause, then $2^n = 2^3$, resulting in eight groupings. Rows grouped on the values of *n* expressions are called regular rows; all others are called superaggregate rows. You can group using composite columns. For example, a commonly requested CUBE operation is for state sales subtotals on all combinations of month, state, and product sold.

If you specify GROUP BY CUBE(a, b, c), the resulting aggregate groupings generated are as follows: (a,b,c), (a,b), (a,c), (b,c), a, b, c, ( ).

This example returns the number of employees for each region and country, issue the following query.

```
Command> SELECT region_name AS Region,
           country_name AS Country,
           COUNT(employee_id) AS "Total Emp"
         FROM regions r, countries c, locations l, departments d, employees e
         WHERE r.region_id = c.region_id
         AND l.country_id = c.country_id
         AND d.location_id = l.location_id
         AND d.department_id = e.department_id
         GROUP BY CUBE(region_name, country_name)
         ORDER BY region_name;

REGION, COUNTRY, TOTAL EMP
< Americas, Canada, 2 >
< Americas, United States of America, 68 >
< Americas, <NULL>, 70 >
< Europe, Germany, 1 >
< Europe, United Kingdom, 35 >
< Europe, <NULL>, 36 >
< <NULL>, Canada, 2 >
< <NULL>, Germany, 1 >
< <NULL>, United Kingdom, 35 >
< <NULL>, United States of America, 68 >
< <NULL>, <NULL>, 106 >
11 rows found.
```

TRUNCATE TABLE

The TRUNCATE TABLE statement is similar to a DELETE statement that deletes all rows.

In TimesTen Classic, the TRUNCATE operation is faster than the DELETE operation in most circumstances, as DELETE removes each row individually.

In TimesTen Scaleout, TRUNCATE TABLE is similar to a DDL statement that invalidates all commands that depend on the table being truncated. It is preferable to use the DELETE statement rather than the TRUNCATE statement to delete all rows in a table.

Required privilege

No privilege is required for the table owner.

DELETE for another user's table.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
TRUNCATE TABLE [Owner.]TableName
```

Parameters

Parameter	Description
[*Owner*.]*TableName*	Identifies the table to be truncated.

Description

- TRUNCATE is a DDL statement. A commit is performed before and after execution of the TRUNCATE statement.
- If your table has out of line columns and there are millions of rows to truncate, consider calling the ttCompact built-in procedure to free memory.
- 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 any detail table of a materialized view, table that is a part of a cache group, or 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".
- 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.

Examples

To delete all the rows from the recreation.clubs table, use:

```
TRUNCATE TABLE recreation.clubs;
```

See also

ALTER TABLE
DROP TABLE

UNLOAD CACHE GROUP

The UNLOAD CACHE GROUP statement removes data from the cache group.

Required privilege

No privilege is required for the cache group owner.

UNLOAD or UNLOAD ANY CACHE GROUP for another user's cache group.

INSERT, DELETE, UPDATE privileges on underlying tables.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

```
UNLOAD CACHE GROUP [Owner.]CacheGroupName
  [WHERE ConditionalExpression]
  [COMMIT EVERY n ROWS]
```

or

```
UNLOAD CACHE GROUP [Owner.]CacheGroupName
WITH ID (ColumnValueList)
```

> **Note**
>
> The WITH ID clause is not supported in TimesTen Scaleout.

Parameters

Parameter	Description
[*Owner*.]*CacheGroupName*	Name assigned to the cache group.
WHERE *ConditionalExpression*	Use the WHERE clause to specify a search condition to qualify the target rows of the cache operation. If you use more than one table in the WHERE clause and the tables have columns with the same names, fully qualify the table names.
COMMIT EVERY *n* ROWS	Use the COMMIT EVERY *n* ROWS clause to indicate the frequency (based on the number of rows that are unloaded) at which a commit is issued during the unload operation. ROWS refers to the number of rows that are deleted from the cache group. For example, if your cache group has one cache instance and the cache instance consists of 1 parent row and 10 child rows, and you issue COMMIT EVERY 2 ROWS, TimesTen issues one commit after the entire cache instance is deleted. TimesTen does not commit in the middle of deleting an cache instance. So once the unload operation reaches its threshold (2 rows in this case), TimesTen issues a commit after all rows are deleted for that cache instance. If you specify this clause, the cache agent must be running and the unload must be the only operation in the transaction. Express *n* as an integer where (*n* >= 0). If you specify 0 for *n*, the UNLOAD CACHE GROUP statement is executed as one transaction and the cache agent does the delete. To improve performance, use this clause when you are performing operations on cache groups that affect large amounts of data. Do not use this clause when you have cache groups with a small amount of data.
WITH ID *ColumnValueList*	The WITH ID clauses enables you to use primary key values to unload the cache instance. Specify *ColumnValueList* as either a list of literals or binding parameters to represent the primary key values. The WITH ID clause is not supported in TimesTen Scaleout.

Description

- The UNLOAD CACHE GROUP statement deletes rows from the TimesTen cache tables without affecting the data in the Oracle database tables.
- If your table has out of line columns and there are millions of rows, consider calling the ttCompact built-in procedure to free memory.
- If the cache group is replicated, an UNLOAD CACHE GROUP statement deletes the entire contents of any replicated cache group as well.
- Execution of the UNLOAD CACHE GROUP statement for an AWT cache group waits until updates on the rows have been propagated to the Oracle database.
- The UNLOAD CACHE GROUP statement can be used for any type of cache group. See "CREATE CACHE GROUP" for information on cache groups.
- Use the UNLOAD CACHE GROUP statement carefully with cache groups that have the AUTOREFRESH 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 the Oracle database.
- Following the execution of an UNLOAD CACHE GROUP statement, the ODBC function SQLRowCount(), the JDBC method getUpdateCount(), and the OCI function OCIAttrGet() with the OCI_ATTR_ROW_COUNT argument return the number of cache instances that were unloaded.
- If you specify the COMMIT EVERY *n* ROWS clause, the cache agent performs the unload operation and commits the transaction after unloading the data. Make sure the cache agent is up and running. If you do not specify the COMMIT EVERY *n* ROWS clause, the unload operation is executed by the application.
- If you specify the COMMIT EVERY *n* ROWS clause, you cannot rollback the unload operation. If the unload operation fails when you specify the COMMIT EVERY *n* ROWS clause (where *n* >= 0), the cache group could be in an inconsistent state since some unloaded rows are already committed. Therefore, some cache instances may be partially unloaded. If this occurs, unload the cache group again.
- Use the WITH ID clause to specify binding parameters.
- The UNLOAD CACHE GROUP operation is executed in its own transaction.

Restrictions

- The UNLOAD CACHE GROUP...WITH ID clause is not supported in TimesTen Scaleout.
- Do not reference child tables in the WHERE clause.
- Do not user the WITH ID clause on static read-only cache groups, or static user-managed cache groups with the autorefresh attribute.
- Do not use the WITH ID clause with the COMMIT EVERY *n* ROWS clause.

Examples

Use the UNLOAD CACHE GROUP... COMMIT EVERY *n* ROWS to unload data from cached tables. The cache agent unloads the data because the COMMIT EVERY *n* ROWS clause is used.

```
Command> UNLOAD CACHE GROUP testcache WHERE sampleuser.orders.order_id > 100
       COMMIT EVERY 100 ROWS;
2 cache instances affected.
```

CREATE and UNLOAD a cache group. The application performs the unload operation because the COMMIT EVERY *n* ROWS clause is not used.

```
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;
```

See also

ALTER CACHE GROUP
CREATE CACHE GROUP
DROP CACHE GROUP
FLUSH CACHE GROUP
LOAD CACHE GROUP

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.

Required privilege

No privilege is required for the table owner.

UPDATE for another user's table.

Usage with TimesTen Scaleout

This statement is supported with TimesTen Scaleout.

SQL syntax

UPDATE [*hint*] [FIRST *NumRows*]
{[*Owner*.]*TableName* [*CorrelationName*]}
SET {*ColumnName* =
{*Expression1* | NULL | DEFAULT}} [,...]
[WHERE *SearchCondition*]
RETURNING|RETURN *Expression2*[,...] INTO *DataItem*[,...]

Parameters

Parameter	Description
hint	Specifies a statement level optimizer hint for the UPDATE statement. See "Statement Level Optimizer Hints" for information on statement level optimizer hints.
FIRST *NumRows*	Specifies the number of rows to update. FIRST *NumRows* is not supported in subquery statements. *NumRows* must be either a positive INTEGER value 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.

Parameter	Description
[*Owner*.]*TableName* [*CorrelationName*]	[*Owner*.]*TableName* identifies the table to be updated. *CorrelationName* specifies an alias for the table and must conform to the syntax rules for a basic name. See Basic Names for details. When accessing columns of that table elsewhere in the UPDATE statement, use the correlation name instead of the actual table name. The scope of the correlation name is the SQL statement in which it is used. All correlation names within one statement must be unique.
SET *ColumnName*	*ColumnName* specifies a 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 columns to be updated as long as the values of the primary key columns are not changed.
Expression1	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 data type of the updated column. *Expression1* can specify a column or sequence CURRVAL or NEXTVAL reference when updating values. See "Using CURRVAL and NEXTVAL in TimesTen Classic " for more details.
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.
Expression2	Valid expression syntax. See Expressions for information.
DataItem	Host variable or PL/SQL variable that stores the retrieved *Expression2* value.

Description

- For TimesTen Scaleout, you cannot update distribution key column(s) unless you update the column(s) to the same value.
- You cannot update primary key column(s) unless you update the column(s) to the original value.
- If the WHERE clause is omitted, all rows of the table are updated as specified by the SET clause.
- TimesTen generates a warning when a character or binary string is truncated during an UPDATE operation.
- Constraint violations (UNIQUE, FOREIGN KEY, NOT NULL) result in the failure of the UPDATE statement.
- The UPDATE operation fails if it violates any foreign key constraint. See CREATE TABLE for a description of foreign key constraints.
- Restrictions on the RETURNING clause:
  - Each *Expression2* must be a simple expression. Aggregate functions are not supported.
  - You cannot return a sequence number into an OUT parameter.
  - ROWNUM and subqueries cannot be used in the RETURNING clause.

- Parameters in the RETURNING clause cannot be duplicated anywhere in the UPDATE statement.
- Using the RETURNING clause to return multiple rows requires PL/SQL BULK COLLECT functionality. See FORALL and BULK COLLECT Operations in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*.
- In PL/SQL, you cannot use a RETURNING clause with a WHERE CURRENT operation.

Examples

Use the UPDATE statement to update *employees* with *department_id* = 110. For *employees* with *department_id* = 110, update the *manager_id* to the *manager_id* of *employees* with *job_id* = 'FI_ACCOUNT'. Use the DISTINCT qualifier in the subquery of the SET clause.

First find the *manager_id* of *employees* with *job_id* = 'FI_ACCOUNT.'

```
Command> SELECT manager_id FROM employees WHERE job_id = 'FI_ACCOUNT';
< 108 >
< 108 >
< 108 >
< 108 >
< 108 >
5 rows found.
```

Next find the *manager_id* of *employees* with *department_id* = 110.

```
Command> SELECT manager_id FROM employees WHERE department_id = 110;
< 101 >
< 205 >
2 rows found.
```

Now update the *manager_id* of *employees* with *department_id* = 110. Use SELECT DISTINCT in the subquery of the SET clause. After the UPDATE, verify the *manager_id* for *employees* with *department_id* = 110 was updated.

```
Command> UPDATE employees SET manager_id =
         (SELECT DISTINCT employees.manager_id
          FROM employees
          WHERE employees.job_id = 'FI_ACCOUNT')
        WHERE employees.department_id = 110;
2 rows updated.

Command> SELECT manager_id FROM employees WHERE department_id = 110;
< 108 >
< 108 >
2 rows found.
```

Use subqueries in the SET clause of the UPDATE statement. Update *employees* with *location_id* = 1700 or *location_id* = 2400. Set *department_id* for these employees to the *department_id* of *location_id* = 2500. (This is *department_id* 80). Set salary for these employees to the maximum salary of their department.

First query the first 5 employees to check their department_id and salary.

```
Command> SELECT FIRST 5 employee_id, department_id, salary
        FROM employees
        ORDER BY employee_id, department_id, salary;
< 100, 90, 24000 >
< 101, 90, 17000 >
< 102, 90, 17000 >
< 103, 60, 9000 >
```

```
< 104, 60, 6000 >
5 rows found.
```

Now use the UPDATE statement to update employees.

```
Command> UPDATE employees e1
       SET department_id =
           (SELECT department_id FROM departments
            WHERE location_id = 2500),
          salary =
           (SELECT MAX(salary) FROM employees e2
             WHERE e1.department_id = e2.department_id)
       WHERE department_id IN
         (SELECT department_id FROM departments
          WHERE location_id = 2400 OR location_id = 1700);
19 rows updated.
```

Query the first five employees again to check that employees with the original *department_id* of 90 have been updated. The *department_id* is now 80 and the salary is 24000.

```
Command> SELECT FIRST 5 employee_id, department_id, salary
       FROM employees
       ORDER BY employee_id, department_id, salary;
< 100, 80, 24000 >
< 101, 80, 24000 >
< 102, 80, 24000 >
< 103, 60, 9000 >
< 104, 60, 6000 >
5 rows found.
```

The following example increases the price of parts costing more than $500 by 25 percent.

```
UPDATE purchasing.parts
SET salesprice = salesprice * 1.25
WHERE salesprice > 500.00;
```

This next example updates the column with the NEXTVAL value from sequence seq.

```
UPDATE student SET studentno = seq.NEXTVAL WHERE name = 'Sally';
```

The following query updates the status of all the customers who have at least one unshipped order.

```
UPDATE customers SET customers.status = 'unshipped'
WHERE customers.id = ANY
   (SELECT orders.custid FROM orders
      WHERE orders.status = 'unshipped');
```

The following statement updates all the duplicate orders, assuming id is not a primary key.

```
UPDATE orders a SET orders.status = 'shipped'
   WHERE EXISTS (SELECT 1 FROM orders b
      WHERE a.id = b.id AND a.rowid < b.rowid);
```

This next example updates job_id, salary and department_id for an employee whose last name is'Jones' in the employees table. The values of salary, last_name and department_id are returned into variables.

```
Command> VARIABLE bnd1 NUMBER(8,2);
Command> VARIABLE bnd2 VARCHAR2(25) INLINE NOT NULL;
Command> VARIABLE bnd3 NUMBER(4);
Command> UPDATE employees SET job_id='SA_MAN', salary=salary+1000,
        department_id=140 WHERE last_name='Jones'
```

```
    RETURNING salary*0.25, last_name, department_id
    INTO :bnd1, :bnd2, :bnd3;
1 row updated.
Command> PRINT bnd1 bnd2 bnd3;
BND1            : 950
BND2            : Jones
BND3            : 140
```

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

A join update statement has the following parameters:

Parameter	Description
[*Owner*.]*TableName*	Identifies the table to be updated.
SET (*ColumnName*[,...])= *Subquery*	Specifies the 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 select list 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 specified using 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 a NULL value in the updated column.

Examples

In this example, if a row from t1 has no match in t2, then its x1 value in the first SELECT and its x1 and y1 values in the second SELECT are set to NULL.

```
UPDATE t1 SET x1=(SELECT x2 FROM t2 WHERE id1=id2);
UPDATE t1 SET (x1,y1)=(SELECT x2,y2 FROM t2 WHERE id1=id2);
```

In order to restrict the UPDATE statement to update only rows from t1 that have a match in t2, a WHERE clause with a 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

7 Privileges

When multiple users can access database objects, authorization can be controlled to these objects with privileges. Every object has an owner. Privileges control if a user can modify an object owned by another user. Privileges are granted or revoked either by the instance administrator, a user with the ADMIN privilege or, for privileges to a certain object, by the owner of the object.

The Granting and Revoking Privileges section in the *Oracle TimesTen In-Memory Database Security Guide* provides a detailed description of how to grant and revoke privileges for the different objects. In addition, the following sections provide a quick reference on all privileges that are required to perform TimesTen operations:

- System Privileges
- Object Privileges
- Privilege Hierarchy
- The PUBLIC Role

System Privileges

A system privilege is the right to perform a particular action or to perform an action on any object of a particular type. Objects include tables, views, materialized views, synonyms, indexes, sequences, cache groups, replication schemes and PL/SQL functions, procedures and packages. Only the instance administrator or a user with ADMIN privilege can grant or revoke system privileges.

Table 7-1 System privileges

Privilege	Description
ADMIN	Enables a user to perform administrative tasks including checkpointing, backups, migration, and user creation and deletion.
ALTER ANY CACHE GROUP	Enables a user to alter any cache group in the database.
ALTER ANY INDEX	Enables a user to alter any index in the database. **Note**: There is no ALTER INDEX statement.
ALTER ANY MATERIALIZED VIEW	Enables a user to alter any materialized view in the database. **Note**: There is no ALTER MATERIALIZED VIEW statement.
ALTER ANY PROCEDURE	Enables a user to alter any PL/SQL procedure, function or package in the database.
ALTER ANY SEQUENCE	Enables a user to alter any sequence in the database.
ALTER ANY TABLE	Enables a user to alter any table in the database.
ALTER ANY VIEW	Enables a user to alter any view in the database. **Note**: There is no ALTER VIEW statement.
CACHE_MANAGER	Enables a user to perform operations related to cache groups.

Table 7-1 (Cont.) System privileges

Privilege	Description
CREATE ANY CACHE GROUP	Enables a user to create a cache group owned by any user in the database.
CREATE ANY INDEX	Enables a user to create an index on any table or materialized view in the database.
CREATE ANY MATERIALIZED VIEW	Enables a user to create a materialized view owned by any user in the database.
CREATE ANY PROCEDURE	Enables a user to create a PL/SQL procedure, function or package owned by any user in the database.
CREATE ANY SEQUENCE	Enables a user to create a sequence owned by any user in the database.
CREATE ANY SYNONYM	Enables a user to create a private synonym owned by any user in the database.
CREATE ANY TABLE	Enables a user to create a table owned by any user in the database.
CREATE ANY VIEW	Enables a user to create a view owned by any user in the database.
CREATE CACHE GROUP	Enables a user to create a cache group owned by that user.
CREATE MATERIALIZED VIEW	Enables a user to create a materialized view owned by that user.
CREATE PROCEDURE	Enables a user to create a PL/SQL procedure, function or package owned by that user.
CREATE PUBLIC SYNONYM	Enables a user to create a public synonym.
CREATE SEQUENCE	Enables a user to create a sequence owned by that user.
CREATE SESSION	Enables a user to create a connection to the database.
CREATE SYNONYM	Enables a user to create a private synonym.
CREATE TABLE	Enables a user to create a table owned by that user.
CREATE VIEW	Enables a user to create a view owned by that user.
DELETE ANY TABLE	Enables a user to delete from any table in the database.
DROP ANY CACHE GROUP	Enables a user to drop any cache group in the database.
DROP ANY INDEX	Enables a user to drop any index in the database.
DROP ANY MATERIALIZED VIEW	Enables a user to drop any materialized view in the database.
DROP ANY PROCEDURE	Enables a user to drop any PL/SQL procedure, function or package in the database.
DROP ANY SEQUENCE	Enables a user to drop any sequence in the database.
DROP ANY SYNONYM	Enables a user to drop a synonym owned by any user in the database.
DROP ANY TABLE	Enables a user to drop any table in the database.
DROP ANY VIEW	Enables a user to drop any view in the database.
DROP PUBLIC SYNONYM	Enables a user to drop a public synonym.
EXECUTE ANY PROCEDURE	Enables a user to execute any PL/SQL procedure, function or package in the database.
FLUSH ANY CACHE GROUP	Enables a user to flush any cache group in the database.

Table 7-1 (Cont.) System privileges

Privilege	Description
INSERT ANY TABLE	Enables a user to insert into any table in the database. It also enables the user to insert into any table using the synonym, public or private, to that table.
LOAD ANY CACHE GROUP	Enables a user to load any cache group in the database.
REFRESH ANY CACHE GROUP	Enables a user to flush any cache group in the database.
SELECT ANY SEQUENCE	Enables a user to select from any sequence or synonym on a sequence in the database.
SELECT ANY TABLE	Enables a user to select from any table, view, materialized view, or synonym in the database.
UNLOAD ANY CACHE GROUP	Enables a user to unload any cache group in the database.
UPDATE ANY TABLE	Enables a user to update any table or synonym in the database.
XLA	Enables a user to connect to a database as an XLA reader.

Object Privileges

An object privilege is the right to perform a particular action on an object or to access another user's object. Objects include tables, views, materialized views, indexes, synonyms, sequences, cache groups, replication schemes and PL/SQL functions, procedures and packages.

An object's owner has all object privileges for that object, and those privileges cannot be revoked. The object's owner can grant object privileges for that object to other database users. A user with ADMIN privilege can grant and revoke object privileges from users who do not own the objects on which the privileges are granted.

Table 7-2 Object privileges

Privilege	Object type	Description
DELETE	Table	Enables a user to delete from a table.
EXECUTE	PL/SQL package, procedure or function	Enables a user to execute a PL/SQL package, procedure or function directly.
FLUSH	Cache group	Enables a user to flush a cache group.
INDEX	Table or materialized view	Enables a user to create an index on a table or materialized view.
INSERT	Table or synonym	Enables a user to insert into a table or into the table through a synonym.
LOAD	Cache group	Enables a user to load a cache group.
REFERENCES	Table or materialized view	Enables a user to create a foreign key dependency on a table or materialized view. The REFERENCES privilege on a parent table implicitly grants SELECT privilege on the parent table.
REFRESH	Cache group	Enables a user to refresh a cache group.

Table 7-2 (Cont.) Object privileges

Privilege	Object type	Description
SELECT	Table, sequence, view, materialized view, or synonym	Enables a user to select from a table, sequence, view, materialized view, or synonym. The SELECT privilege enables a user to perform all operations on a sequence. A user can be granted the SELECT privilege on a synonym or a view without being explicitly granted the SELECT privilege on the originating table.
UNLOAD	Cache group	Enables a user to unload a cache group.
UPDATE	Table	Enables a user to update a table.

Privilege Hierarchy

Some privileges confer other privileges. For example, ADMIN privilege confers all other privileges. The CREATE ANY TABLE system privilege confers the CREATE TABLE object privilege. Table 7-3 shows the privilege hierarchy.

Table 7-3 Privilege hierarchy

Privilege	Confers these privileges
ADMIN	All other privileges including CACHE_MANAGER
CREATE ANY INDEX	INDEX (any table or materialized view)
CREATE ANY MATERIALIZED VIEW	CREATE MATERIALIZED VIEW
CREATE ANY PROCEDURE	CREATE PROCEDURE
CREATE ANY SEQUENCE	CREATE SEQUENCE
CREATE ANY SYNONYM	CREATE SYNONYM
CREATE ANY TABLE	CREATE TABLE
CREATE ANY VIEW	CREATE VIEW
DELETE ANY TABLE	DELETE (any table)
EXECUTE ANY PROCEDURE	EXECUTE (any procedure)
INSERT ANY TABLE	INSERT (any table)
SELECT ANY SEQUENCE	SELECT (any sequence)
SELECT ANY TABLE	SELECT (any table, view or materialized view)
UPDATE ANY TABLE	UPDATE (any table)

Cache group privileges have a separate hierarchy except that ADMIN confers the CACHE_MANAGER privilege.

The CACHE_MANAGER privilege confers these privileges:

- CREATE ANY CACHE GROUP

- ALTER ANY CACHE GROUP
- DROP ANY CACHE GROUP
- FLUSH ANY CACHE GROUP
- LOAD ANY CACHE GROUP
- UNLOAD ANY CACHE GROUP
- REFRESH ANY CACHE GROUP
- FLUSH (object)
- LOAD (object)
- UNLOAD (object)
- REFRESH (object)

The CACHE_MANAGER privilege also includes the ability to start and stop the cache agent and the replication agent.

CREATE ANY CACHE GROUP confers the CREATE CACHE GROUP privilege for any cache group.

The PUBLIC Role

All users of the database have the PUBLIC role. In a newly created TimesTen database, by default PUBLIC has SELECT and EXECUTE privileges on various system tables and views and PL/SQL functions, procedures and packages. You can see the list of objects by using this query:

```
SELECT table_name, privilege FROM sys.dba_tab_privs WHERE grantee='PUBLIC';
```

Privileges that are granted to PUBLIC as part of database creation cannot be revoked. To see a list of these privileges, use this query:

```
SELECT table_name, privilege FROM sys.dba_tab_privs WHERE grantor='SYS';
```

8 Reserved Words

TimesTen reserves words for use in SQL statements.You cannot use reserved words as non-quoted identifiers.

To use one of these words as an identifier (such as a table name or column name), enclose the reserved word in quotes.

The reserved words are:

- AGING
- ALL
- ANY
- AS
- BETWEEN
- BINARY_DOUBLE_INFINITY
- BINARY_DOUBLE_NAN
- BINARY_FLOAT_INFINITY
- BINARY_FLOAT_NAN
- CASE
- CHAR
- COLUMN
- COLUMNAR
- CONNECTION
- CONSTRAINT
- CROSS
- CURRENT_SCHEMA
- CURRENT_USER
- CURSOR
- DATASTORE_OWNER
- DATE
- DECIMAL
- DEFAULT
- DESTROY
- DISTINCT
- DISTRIBUTE
- FIRST

- FLOAT
- FOR
- FOREIGN
- FROM
- GROUP
- HAVING
- INNER
- INTEGER
- INTERSECT
- INTERVAL
- INTO
- IS
- JOIN
- LEFT
- LIKE
- LONG
- MINUS
- NULL
- ON
- ORA_SYSDATE
- ORDER
- PRIMARY
- PROPAGATE
- PUBLIC
- READONLY
- RIGHT
- ROWNUM
- ROWS
- SELECT
- SELF
- SESSION_USER
- SET
- SMALLINT
- SOME
- SYSDATE
- SYSTEM_USER

- TO
- TT_SYSDATE
- UID
- UNION
- UNIQUE
- UPDATE
- USER
- USING
- VARCHAR
- WHEN
- WHERE
- WITH