Oracle® TimesTen In-Memory Database

SQL Reference

Release 11.2.1

E13070-08

October 2010

ORACLE®

Oracle TimesTen In-Memory Database SQL Reference, Release 11.2.1

E13070-08

Copyright © 1996, 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface ix

Audience ix
Related documents ix
Conventions ix
Documentation Accessibility x
Technical support xi

What's New xiii

New features in Release 11.2.1.7.0 xiii
New features in Release 11.2.1.6.0 xiii
New features in Release 11.2.1.4.0 xiii
New features in Release 11.2.1.1.0 xiv

1 Data Types

Type specifications 1-1
ANSI SQL data types 1-5
Types supported for backward compatibility in Oracle type mode 1-7
TimesTen type mapping 1-9
Character data types 1-11
CHAR 1-11
NCHAR 1-12
VARCHAR2 1-13
NVARCHAR2 1-14
Numeric data types 1-15
Exact and approximate numeric data types 1-15
NUMBER 1-15
TT_BIGINT 1-17
TT_INTEGER 1-18
TT_SMALLINT 1-19
TT_TINYINT 1-19
Floating-point numbers 1-19
BINARY_DOUBLE 1-20
BINARY_FLOAT 1-20
FLOAT and FLOAT(*n*) 1-20
BINARY and VARBINARY data types 1-20

Numeric precedence 1-21
ROWID data type 1-21
Datetime data types 1-22
DATE 1-22
TIME 1-22
TIMESTAMP 1-23
TT_DATE 1-23
TT_TIMESTAMP 1-23
TimesTen intervals 1-23
Using interval data types 1-23
Using DATE and TIME data types 1-23
Handling timezone conversions 1-24
Datetime and interval data types in arithmetic operations 1-24
Restrictions on datetime and interval arithmetic operations 1-26
Storage requirements 1-26
Data type comparison rules 1-27
Numeric values 1-27
Date values 1-27
Character values 1-28
Binary and linguistic sorting 1-28
Blank-padded and nonpadded comparison semantics 1-28
Data type conversion 1-28
Implicit data type conversion 1-28
NULL values 1-29
INF and NAN 1-30
Constant values 1-30
Primary key values 1-30
Selecting Inf and NaN (floating-point conditions) 1-30
Expressions involving Inf and NaN 1-31
Overflow and truncation 1-31
Underflow 1-32
Replication limits 1-32
TimesTen type mode (backward compatibility) 1-32
Data types supported in TimesTen type mode 1-33
Oracle data types supported in TimesTen type mode 1-36

2 Names and Parameters

Basic names 2-1
Owner names 2-1
Compound identifiers 2-2
Dynamic parameters 2-2
Duplicate parameter names 2-3
Inferring data type from parameters 2-3

3 Expressions

ROWID specification 3-1
ROWNUM specification 3-1

Expression specification 3-3
Subqueries 3-6
Aggregate functions 3-8
Constants 3-11
Format models 3-16
Number format models 3-17
Number format elements 3-17
Datetime format models 3-20
Datetime format elements 3-21
Format model for ROUND and TRUNC date functions 3-23
Format model for TO_CHAR of TimesTen datetime data types 3-24
ABS 3-26
ADD_MONTHS 3-27
ASCIISTR 3-29
CASE 3-30
CAST 3-32
CHR 3-33
CEIL 3-34
COALESCE 3-35
CONCAT 3-36
DECODE 3-38
EXTRACT 3-39
FLOOR 3-40
GREATEST 3-41
LEAST 3-43
LOWER and UPPER 3-45
LPAD 3-46
LTRIM 3-48
MOD 3-50
NCHR 3-51
NLSSORT 3-52
NUMTODSINTERVAL 3-54
NUMTOYMINTERVAL 3-55
NVL 3-56
POWER 3-57
ROUND (date) 3-58
ROUND (expression) 3-59
RPAD 3-61
RTRIM 3-63
SIGN 3-65
SQRT 3-67
String functions 3-68
SUBSTR, SUBSTRB, SUBSTR4 3-69
INSTR, INSTRB, INSTR4 3-70
LENGTH, LENGTHB, LENGTH4 3-71
SYS_CONTEXT 3-72
SYSDATE and GETDATE 3-74

TO_CHAR ... 3-76
TO_DATE ... 3-78
TO_NUMBER ... 3-79
TRIM ... 3-80
TRUNC (date) ... 3-83
TRUNC (expression) ... 3-84
TT_HASH ... 3-85
UID ... 3-86
UNISTR ... 3-87
USER functions ... 3-88
CURRENT_USER ... 3-89
USER ... 3-90
SESSION_USER ... 3-91
SYSTEM_USER ... 3-92

4 Search Conditions

Search condition general syntax ... 4-2
ALL/ NOT IN predicate (subquery) ... 4-4
ALL/NOT IN predicate (value list) ... 4-6
ANY/ IN predicate (subquery) ... 4-8
ANY/ IN predicate (value list) ... 4-10
BETWEEN predicate ... 4-13
Comparison predicate ... 4-14
EXISTS predicate ... 4-16
IS INFINITE predicate ... 4-18
IS NAN predicate ... 4-19
IS NULL predicate ... 4-20
LIKE predicate ... 4-21
NCHAR and NVARCHAR2 ... 4-24

5 SQL Statements

Comments within SQL statements ... 5-1
ALTER ACTIVE STANDBY PAIR ... 5-2
ALTER CACHE GROUP ... 5-6
ALTER FUNCTION ... 5-8
ALTER PACKAGE ... 5-10
ALTER PROCEDURE ... 5-12
ALTER REPLICATION ... 5-14
ALTER SESSION ... 5-23
ALTER TABLE ... 5-29
ALTER USER ... 5-42
CALL ... 5-44
COMMIT ... 5-46
CREATE ACTIVE STANDBY PAIR ... 5-47
CREATE CACHE GROUP ... 5-53
CREATE FUNCTION ... 5-66
CREATE INDEX ... 5-69

CREATE MATERIALIZED VIEW 5-73
CREATE MATERIALIZED VIEW LOG 5-79
CREATE PACKAGE 5-81
CREATE PACKAGE BODY 5-83
CREATE PROCEDURE 5-84
CREATE REPLICATION 5-87
CHECK CONFLICTS 5-94
CREATE SEQUENCE 5-101
CREATE SYNONYM 5-104
CREATE TABLE 5-107
Column Definition 5-112
CREATE USER 5-122
CREATE VIEW 5-124
DELETE 5-126
DROP ACTIVE STANDBY PAIR 5-129
DROP CACHE GROUP 5-130
DROP FUNCTION 5-131
DROP INDEX 5-132
DROP [MATERIALIZED] VIEW 5-134
DROP MATERIALIZED VIEW LOG 5-135
DROP PACKAGE [BODY] 5-136
DROP PROCEDURE 5-137
DROP REPLICATION 5-138
DROP SEQUENCE 5-139
DROP SYNONYM 5-140
DROP TABLE 5-141
DROP USER 5-142
FLUSH CACHE GROUP 5-143
GRANT 5-145
INSERT 5-147
INSERT...SELECT 5-150
LOAD CACHE GROUP 5-151
MERGE 5-155
REFRESH CACHE GROUP 5-159
REFRESH MATERIALIZED VIEW 5-162
REVOKE 5-163
ROLLBACK 5-165
SELECT 5-166
SelectList 5-176
TableSpec 5-179
DerivedTable 5-180
JoinedTable 5-181
TRUNCATE TABLE 5-184
UNLOAD CACHE GROUP 5-186
UPDATE 5-188
Join update 5-191

6 Privileges

System privileges 6-1
Object privileges 6-3
Privilege hierarchy 6-4
The PUBLIC role 6-5

7 Reserved Words

Index

Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory using standard SQL interfaces.

Audience

This document provides a reference for TimesTen SQL statements, expressions, and functions, including TimesTen SQL extensions. It also describes data types.

To work with this guide, you should understand how database systems work. You should also have knowledge of SQL (Structured Query Language).

Related documents

TimesTen documentation is available on the product distribution media and on the Oracle Technology Network:

`http://www.oracle.com/technetwork/database/timesten/documentation`

Conventions

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

> **Note:** In TimesTen documentation, the terms "data store" and "database" are equivalent. Both terms refer to the TimesTen database unless otherwise noted.

This document uses the following text conventions:

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.

Convention	Meaning
italic monospace	Italic monospace type indicates a variable in a code example that you must replace. For example: `Driver=install_dir/lib/libtten.sl` Replace *install_dir* with the path of your TimesTen installation directory.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line.
%	The percent sign indicates the UNIX shell prompt.
#	The number (or pound) sign indicates the UNIX root prompt.

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

Convention	Meaning
install_dir	The path that represents the directory where the current release of TimesTen is installed.
TTinstance	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique alphanumeric instance name. This name appears in the install path.
bits or *bb*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
release or *rr*	Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.
DSN	The data source name.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/support/contact.html` or visit `http://www.oracle.com/accessibility/support.html` if you are hearing impaired.

Technical support

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

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

What's New

This section lists new features for Release 11.2.1 that are documented in this reference and provides cross-references to additional information.

New features in Release 11.2.1.7.0

This section lists new features for Release 11.2.1.7.0 that are documented in this reference and provides cross-references to additional information:

- The `CALL` statement is now documented to cover how to execute the TimesTen built-in procedures and how to execute PL/SQL procedures and functions. See "CALL" on page 5-44 for more information.
- The `SYS_CONTEXT` expression can now be used to retrieve the connection ID. See "SYS_CONTEXT" on page 3-72 for details.
- Explicit load for a global cache group is now enabled. For details, see "CREATE CACHE GROUP" on page 5-53 for more information.

New features in Release 11.2.1.6.0

This section lists new features for Release 11.2.1.6.0 that are documented in this reference and provides cross-references to additional information:

- The system tables and replication tables are now documented in *Oracle TimesTen In-Memory Database System Tables and Limits Reference*.
- You can modify the `REPLICATION_TRACK` session parameter for parallel replication with the `ALTER SESSION` statement. See `ALTER SESSION` for more details.

New features in Release 11.2.1.4.0

This section lists new features for Release 11.2.1.4.0 that are documented in this reference and provides cross-references to additional information:

- You can create synonyms for database objects including tables, views, sequences, PL/SQL stored procedures, PL/SQL functions, PL/SQL packages, materialized views and cache groups. For more information, see:
  - `CREATE SYNONYM`
  - `DROP SYNONYM`

- CREATE ANY SYNONYM, CREATE PUBLIC SYNONYM, DROP ANY SYNONYM and DROP PUBLIC SYNONYM system privileges in Table 6–1, " System privileges"

- Time-based aging can now be specified for tables and cache groups in units of seconds as well as minutes, hours, and days. See the ALTER TABLE, CREATE TABLE, and CREATE CACHE GROUP statements.
- If privileges on a table owned by one user are revoked from another user, the associated materialized views owned by the second user are marked invalid. See "Invalid materialized views" on page 5-76.

New features in Release 11.2.1.1.0

This section lists new features for Release 11.2.1.1.0 that are documented in this reference and provides cross-references to additional information.

Access Control

- New system privileges and object privileges. See Chapter 6, "Privileges". Also see "Required privileges" for each SQL statement in Chapter 5, "SQL Statements".
- New syntax for ALTER USER, CREATE USER, DROP USER, GRANT and REVOKE statements
- Revised reserved words. See Chapter 7, "Reserved Words".

PL/SQL support

The ALTER SESSION statement has been enhanced.

These statements are new:

- ALTER FUNCTION
- ALTER PACKAGE
- ALTER PROCEDURE
- CREATE FUNCTION
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PROCEDURE
- DROP FUNCTION
- DROP PACKAGE [BODY]
- DROP PROCEDURE

These SQL functions are new:

- SYS_CONTEXT
- UID

Oracle In-Memory Database Cache (IMDB Cache)

These IMDB Cache features are new:

- Dynamic cache groups: In a dynamic cache group, new cache instances are loaded manually into the TimesTen cache tables using a load operation, or on demand using a dynamic load operation. See:

  - CREATE CACHE GROUP
  - LOAD CACHE GROUP
  - REFRESH CACHE GROUP
  - UNLOAD CACHE GROUP
- Cache grid: A cache grid is a collection of TimesTen databases that collectively manage the application data using the relational data model. A cache grid consists of one or more grid members each backed by a TimesTen database. See "CREATE CACHE GROUP" on page 5-53.
- Global cache groups: In a global cache group, data in the cache tables are shared among TimesTen databases within a cache grid. See "CREATE CACHE GROUP" on page 5-53.

Asynchronous materialized views

Materialized views can be refreshed asynchronously. The CREATE MATERIALIZED VIEW statement has been enhanced. These statements are new:

- CREATE MATERIALIZED VIEW LOG
- DROP MATERIALIZED VIEW LOG
- REFRESH MATERIALIZED VIEW

RETURNING ... INTO clause

The DELETE, INSERT and UPDATE statements have been enhanced with the RETURNING...INTO clause.

ROWID data type

The ROWID data type has been implemented. See "ROWID data type" on page 1-21.

Duplicate parameter names

TimesTen offers Oracle-style behavior for duplicated parameter names. See "Duplicate parameter names" on page 2-3.

Bitmap indexes

You can create bitmap indexes. See "CREATE INDEX" on page 5-69.

Set operators in subqueries

Set operators are allowed in subqueries. See "Subqueries" on page 3-6.

Enhanced ":" parameter markers

":" parameter markers have been enhanced. See "Dynamic parameters" on page 2-2.

Multiline C-style comments

You can use multiline C-style comments in SQL statements. See "Comments within SQL statements" on page 5-1.

View and sequence names

A view and a sequence cannot have the same name. See "CREATE VIEW" on page 5-124 and "CREATE SEQUENCE" on page 5-101.

1

Data Types

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

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

TimesTen follows the ODBC standard for type conversion. A discussion of this standard is not included in this guide. For information, refer to ODBC API reference documentation, which is available from Microsoft or a variety of third parties. For example:

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

If you are using IMDB Cache, see "Mappings between Oracle and TimesTen data types" in *Oracle In-Memory Database Cache User's Guide*. This section compares valid data types for creating cache group columns, as well as type conversions for passthrough queries.

Type specifications

TimesTen supports the data types in Table 1–1 in the default Oracle type mode. The type mode is a data store attribute. `TypeMode=0` indicates Oracle type mode. `TypeMode=1` indicates TimesTen mode.

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

Table 1–1 Data types supported in Oracle type mode

Data type	Description
`BINARY(n)`	Fixed-length binary value of *n* bytes. Legal 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)`.
`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

Table 1–1 (Cont.) Data types supported in Oracle type mode

Data type	Description
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
CHAR[ACTER][(*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. Legal 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. 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. Alternatively, specify ORA_CHAR[(*n*[BYTE|CHAR])].
DATE	Stores date and time information: century, year, month, date, hour, minute, and second. Format is: *YYYY-MM-DD HHMMSS*. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. There are no fractional seconds. Alternatively, specify ORA_DATE.
INTERVAL [+/-] *IntervalQualifier*	TimesTen partially supports interval types, expressed with the type INTERVAL and an *IntervalQualifier*. An *IntervalQualifier* can only specify a single field type with no precision. The default leading precision is 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.

Table 1–1 (Cont.) Data types supported in Oracle type mode

Data type	Description
NCHAR[(*n*)]	Fixed-length string of *n* two-byte Unicode characters. The number of bytes required is 2**n* where *n* is the specified number of characters. NCHAR character limits are 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. Alternatively, specify ORA_NCHAR[(*n*)].
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 zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} to (but not including) 1.0×10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0×10^{126}, then TimesTen returns an error.
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 so the maximum size is 2,097,152 (2^{21}). You must specify *n*. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. Alternatively, specify ORA_NVARCHAR2(*n*).
ROWID	An 18-byte character string that represents the address of a table row or materialized view row. Specify a literal ROWID value as a CHAR constant enclosed in single quotes.
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.

Table 1–1 (Cont.) Data types supported in Oracle type mode

Data type	Description
TIMESTAMP [(*fractional_seconds_precision*)]	Stores 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, 1754. 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*] Alternatively, specify ORA_TIMESTAMP[(*fractional_seconds_precision*)]
TT_BIGINT	A signed eight-byte integer in the following range: -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 (2^{63}-1). Use TT_BIGINT rather than the NUMBER data type. TT_BIGINT is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19-digit integers, use NUMBER(*p*) where $p > 19$.
TT_DATE	Stores date information: century, year, month, date. 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).
TT_INTEGER	A signed integer in the range -2,147,483,648 (-2^{31}) to 2,147,483,647 (2^{31}-1). TT_INTEGER is a native signed integer data type. Use TT_INTEGER rather than INTEGER. INTEGER maps to the NUMBER data type. TT_INTEGER is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19-digit integers, use NUMBER(*p*) where $p > 19$.
TT_SMALLINT	A native signed 16-bit integer in the range -32,768 (-2^{15}) to 32,767 (2^{15}-1). Use TT_SMALLINT rather than SMALLINT. SMALLINT maps to the NUMBER data type. TT_SMALLINT is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19-digit integers, use NUMBER(*p*) where $p > 19$.

Table 1–1 (Cont.) Data types supported in Oracle type mode

Data type	Description
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* [*.FFFFFF*]. 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, 1754. 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).
TT_TINYINT	Unsigned integer ranging from 0 to 255 (2^8-1). Use TT_TINYINT rather than the NUMBER data type. TT_TINYINT is more compact and offers faster performance than the NUMBER type. If you need to store greater than 19-digit integers, use NUMBER(*p*) where *p* > 19. Since TT_TINYINT is unsigned, the negation of a TT_TINYINT is a TT_SMALLINT.
VARBINARY(*n*)	Variable-length binary value having maximum length *n* bytes. Legal values for *n* range from 1 to 4194304 (2^{22}). Alternatively, specify TT_VARBINARY(*n*).
VARCHAR[2](*n*[BYTE|CHAR])	Variable-length character string having maximum length *n* bytes or characters. BYTE indicates that the column has byte-length semantics. Legal values for *n* range from a minimum of one byte to a maximum 4194304 (2^{22}) bytes. You must specify *n*. CHAR indicates that the column has character-length semantics. A zero-length string is interpreted as NULL. Nonpadded comparison semantics are used. Do not use the VARCHAR type. Although it is currently synonymous with VARCHAR2, the VARCHAR type is scheduled to be redefined. Alternatively, specify ORA_VARCHAR2(*n*[BYTE|CHAR]).

ANSI SQL data types

TimesTen supports ANSI SQL data types in Oracle type mode. 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) or ORA_FLOAT(126).
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. Alternatively, specify ORA_FLOAT. For example: FLOAT(24) = ORA_FLOAT(24) FLOAT(53) = ORA_FLOAT(53) FLOAT(*n*) = ORA_FLOAT(*n*)
INT[EGER]	NUMBER(38,0) TT_INTEGER is a native 32-bit integer type. Use TT_INTEGER, as this data type is more compact and offers faster performance than the NUMBER type.
NATIONAL CHARACTER(*n*) or NATIONAL CHAR(*n*)	NCHAR(*n*)
NATIONAL CHARACTER VARYING(*n*) or NATIONAL CHAR VARYING(*n*) or NCHAR VARYING(*n*)	NVARCHAR2(*n*)

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

ANSI SQL data type	TimesTen data type
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 ORA_FLOAT(63) or 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 in Oracle type mode

TimesTen supports the data types shown in Table 1–3 for backward compatibility in Oracle type mode.

Table 1–3 Data types supported for backward compatibility in Oracle type mode

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. Legal 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. This is true in Oracle type mode only. TT_CHAR data is padded to the maximum column size with trailing blanks. Blank-padded comparison semantics are used.
TT_DECIMAL[(*p*[,*s*])]	An exact numeric value with a fixed maximum precision (total number of digits) and scale (number of digits to the right of the decimal point). The precision *p* must be between 1 and 40. The scale *s* must be between 0 and *p*. The default precision is 40 and the default scale is 0. Use the NUMBER data type, which offers better performance, rather than TT_DECIMAL.

Table 1–3 (Cont.) Data types supported for backward compatibility in Oracle type mode

Data type	Description
TT_NCHAR[(*n*)]	Fixed-length string of *n* two-byte Unicode characters. The number of bytes required is 2**n* where *n* is the specified number of characters. NCHAR character limits are 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. This is true in Oracle type mode only. 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. This is true in Oracle type mode only. 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. Legal 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. This is true in Oracle type mode only. Blank-padded comparison semantics are used.

TimesTen type mapping

The names of the data types listed in the left column of Table 1–4 are the data types that existed in previous releases of TimesTen. If TypeMode is set to 0 (the default), indicating Oracle type mode, then the name of the data type may be changed to a new name in Oracle type mode. The name of the data type in Oracle type mode is listed in the right column. The table illustrates the mapping of the data type in the left column to the corresponding data type in the right column.

Table 1–4 Data type mapping: TimesTen data type to TimesTen data type in Oracle type mode

TimesTen data type	TimesTen data type in Oracle type mode
BIGINT	TT_BIGINT In Oracle type mode, specify TT_BIGINT. For more information on TT_BIGINT, see "Type specifications" on page 1-1.
BINARY(*n*)	BINARY(*n*) In Oracle type mode, the data type has the same name. For more information on BINARY(*n*), see "Type specifications" on page 1-1.
CHAR[ACTER][(*n*)]	TT_CHAR[(*n*[BYTE|CHAR])] In Oracle type mode, specify TT_CHAR. Character semantics is supported. For more information on type TT_CHAR, see "Types supported for backward compatibility in Oracle type mode" on page 1-7.
DATE	TT_DATE In Oracle type mode, specify TT_DATE. For more information on TT_DATE, see "Type specifications" on page 1-1.
DEC[IMAL][(*p*[,*s*])] or NUMERIC[(*p*[,*s*])]	TT_DECIMAL[(*p*[,*s*])] In Oracle type mode, specify TT_DECIMAL. For more information on TT_DECIMAL, see "Types supported for backward compatibility in Oracle type mode" on page 1-7.
DOUBLE [PRECISION] or FLOAT[(53)]	BINARY_DOUBLE In Oracle type mode, specify BINARY_DOUBLE. For more information on BINARY_DOUBLE, see "Type specifications" on page 1-1.
INT[EGER]	TT_INT[EGER] In Oracle type mode, specify TT_INTEGER. For more information on TT_INTEGER, see "Type specifications" on page 1-1.
INTERVAL *IntervalQualifier*	INTERVAL *IntervalQualifier* In Oracle type mode, the data type has the same name. For more information on interval types, see "Type specifications" on page 1-1.
NCHAR[(*n*)]	TT_NCHAR[(*n*)] In Oracle type mode, specify TT_CHAR. For more information on TT_NCHAR, see "Types supported for backward compatibility in Oracle type mode" on page 1-7.
NVARCHAR(*n*)	TT_NVARCHAR(*n*) In Oracle type mode, specify TT_NVARCHAR. For more information on TT_NVARCHAR, see "Types supported for backward compatibility in Oracle type mode" on page 1-7.
REAL or FLOAT(24)	BINARY_FLOAT In Oracle type mode, specify BINARY_FLOAT. For more information on BINARY_FLOAT, see "Type specifications" on page 1-1.
SMALLINT	TT_SMALLINT In Oracle type mode, specify TT_SMALLINT. For more information on TT_SMALLINT, see "Type specifications" on page 1-1.
TIME	TIME In Oracle type mode, the data type has the same name. For more information on TIME, see "Type specifications" on page 1-1.
TIMESTAMP	TT_TIMESTAMP In Oracle type mode, specify TT_TIMESTAMP. For more information on TT_TIMESTAMP, see "Type specifications" on page 1-1.

Table 1–4 (Cont.) Data type mapping: TimesTen data type to TimesTen data type in Oracle type mode

TimesTen data type	TimesTen data type in Oracle type mode
TINYINT	TT_TINYINT In Oracle type mode, specify TT_TINYINT. For more information on TT_TINYINT, see "Type specifications" on page 1-1.
VARBINARY(*n*)	VARBINARY(*n*) In Oracle type mode, the data type has the same name. For more information on VARBINARY(*n*), see "Type specifications" on page 1-1.
VARCHAR(*n*)	TT_VARCHAR(*n*[BYTE|CHAR]) In Oracle type mode, specify TT_VARCHAR. Character semantics is supported. For more information on TT_VARCHAR, see "Types supported for backward compatibility in Oracle type mode" on page 1-7.

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 byte and multibyte character sets.

The character types are:

- 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, then TimesTen returns an error.

By default, the column length is defined in bytes. Use the CHAR qualifier to define the column length in characters. The size of a character ranges from one byte to four bytes depending on the database character set. The BYTE and CHAR qualifiers override the NLS_LENGTH_SEMANTICS parameter setting. For more information about NLS_LENGTH_SEMANTICS, see "ALTER SESSION" on page 5-23 and "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 following example alters table `typedemo`, adding column `Name4`. Data type is `NCHAR`.

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

Table USER.TYPEDEMO:
  Columns:
```

```
    NAME                            CHAR (20)
    NAME2                           TT_CHAR (20)
    NAME3                           CHAR (10 CHAR)
    NAME4                           NCHAR (10)
1 table found.
```

VARCHAR2

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. For more information on NLS_LENGTH_SEMANTICS, see "ALTER SESSION" on page 5-23 and "Setting globalization support attributes" in *Oracle TimesTen In-Memory Database Operations Guide*.

The NULL value is stored as a single bit inside the tuple for each nullable field. A NOT INLINE VARCHAR2(*n*) whose value is NULL takes (null bit) + four bytes of storage on 32-bit platforms, whereas 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.

Notes:

- 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 following example alters table typedemo adding 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).

Exact and approximate numeric data types

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.

The exact numeric types are:

- `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 zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} up to but not including 1.0×10^{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 non-zero 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 DROP col3;
Command> ALTER TABLE numerics ADD col3 INTEGER;
Command> DESCRIBE numerics;
Table USER1.NUMERICS:
Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          NUMBER (38)
1 table found.
(primary key columns are indicated with *)
Command> ALTER TABLE numerics DROP col3;
Command> ALTER TABLE numerics ADD COLUMN col3 TT_INTEGER;
Command> DESCRIBE numerics;
Table USER1.NUMERICS:
Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          TT_INTEGER
1 table found.
(primary key columns are indicated with *)
Command> ALTER TABLE numerics ADD col4 TT_INT;
Command> DESCRIBE numerics;
Table USER1.NUMERICS:
Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
    COL3                          TT_INTEGER
    COL4                          TT_INTEGER
1 table found.
(primary key columns are indicated with *)
```

TT_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 DROP COLUMN col2;
Command> ALTER TABLE numerics ADD COLUMN col2 TT_SMALLINT;
Command> DESCRIBE numerics;
Table USER1.NUMERICS:
  Columns:
    COL1                          TT_TINYINT
    COL2                          TT_SMALLINT
1 table found.
(primary key columns are indicated with *)
```

TT_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.

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.

```
Command> CREATE TABLE bvar (col1 BINARY (10), col2 VARBINARY (10));
Command> DESCRIBE bvar;
```

```
Table USER1.BVAR:
  Columns:
    COL1                          BINARY (10)
    COL2                          VARBINARY (10) INLINE
1 table found.
(primary key columns are indicated with *)
```

Numeric precedence

The result type of an expression is determined by the operand with the highest type precedence. 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.

ROWID data type

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 specification" on page 3-1.

Specify literal ROWID values in SQL statements as constants enclosed in single quotes. For example:

```
Command> SELECT ROWID, last_name FROM employees
       > WHERE ROWID='BMUFVUAAABTAAAAFi8';
< BMUFVUAAABTAAAAFi8, Hartstein >
1 row found.
```

The ROWID data type can be used as follows:

- As the data type for a table column or materialized view column
- In these types of expressions:
  - Literals
  - Comparisons: <, <=, >, >=, BETWEEN
  - CASE
  - 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:

```
INSERT INTO t2(col1) SELECT ROWID FROM t1;
INSERT INTO t2(col1) SELECT TO_CHAR(ROWID) FROM t1;
INSERT INTO t2(col1) SELECT CAST(ROWID, CHAR(18)) FROM t1;
INSERT INTO t2(col1) SELECT CAST(ROWID, CHAR(18)) FROM t1;
```

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 CASE, COALESCE, DECODE, or NVL 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 SQLTO_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. The SQL TO_CHAR and TO_DATE functions can be used 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`* `[.`*`FFFFFF`*`]`. The fractional seconds precision is 6. The range is from 1753-01-01 00:00:00 (January 1, 1753, midnight) to 9999-12-31 23:59:59 (December 31, 9999, 11:59:59 PM). The `TT_TIMESTAMP` type requires 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 timezone conversions
- Datetime and interval data types in arithmetic operations

Using interval data types

If you are using TimesTen type mode, refer to the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*, Release 6.0.3, for information on interval 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" on page 1-1.

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

Handling timezone conversions

TimesTen 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

If you are using TimesTen type mode, see *Oracle TimesTen In-Memory Database API and SQL Reference Guide*, Release 6.0.3, for information about datetime and interval types in arithmetic operations.

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

- *TimeVal1 - TimeVal2* or *TimestampVal1 - TimestampVal2* or *DateVal1 - DateVal2* returns the difference as INTERVAL DAY TO SECOND.
- *TT_DateVal1 - TT_DateVal2* returns the number of days difference as an integer.
- *DateTimeVal* {+|-} *IntervalVal*
- *IntervalVal* + *DateTimeVal*
- *IntervalVal1* {+|-} *IntervalVal2*

- *IntervalVal* {*|/} *NumericVal*
- *NumericVal* * *IntervalVal*

> **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.

The following table lists the data type, or category of data type, that results from each operation:

Operand 1	Operator	Operand 2	Result type
TIME | DATE | TIMESTAMP	-	TIME | DATE | TIMESTAMP	INTERVAL DAY TO SECOND
TT_DATE	-	TT_DATE	TT_BIGINT (number of days)
Datetime	+ or -	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ or -	Interval	Interval
Interval	* or /	Numeric	Interval
Numeric	*	Interval	Interval

```
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;
SELECT date1 - date2 FROM t1;
```

You cannot compare an INTERVAL YEAR TO MONTH with an INTERVAL DAY TO SECOND:

```
SELECT * FROM t1 WHERE timestamp1 - timestamp2 = NUMTOYMINTERVAL(10, 'YEAR');
```

You cannot compare 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 message 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. For character semantics, the number of bytes stored out of line is dependent 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: On 32-bit platforms, 2*(length of value) + 20 bytes (minimum of 28 bytes). On 64-bit platforms, 2*(length of value) + 24 bytes (minimum of 40 bytes). For INLINE columns: On 32-bit platforms, 2*(length of column) + 4 bytes. On 64-bit platforms, 2*(length of column) + 8 bytes.
ROWID	Twelve bytes.

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

Type	Storage required
TIMESTAMP	Twelve bytes.
TT_BIGINT	Eight bytes.
TT_DATE	Four bytes.
TT_DECIMAL(*p*,*s*)	Approximately *p*/2 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: On 32-bit platforms, length of value + 20 bytes (minimum of 28 bytes). On 64-bit platforms, length of value + 24 bytes (minimum of 40 bytes). For INLINE columns: On 32-bit platforms, length of column + 4 bytes. On 64-bit platforms, length of column + 8 bytes.
VARCHAR2(*n*[BYTE|CHAR])	For NOT INLINE columns: On 32-bit platforms, length of value + 20 bytes (minimum of 28 bytes). NULL value is stored as (null bit) + 4 bytes, or 4.125 bytes. On 64-bit platforms, 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: On 32-bit platforms, *n* + 4 bytes. NULL value is stored as (null bit) + *n* + 4 bytes. On 64-bit platforms, 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.

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 nonpadded comparison semantics

Binary and linguistic sorting

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

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

The default is binary sorting.

Blank-padded and nonpadded comparison semantics

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

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

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

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

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. We recommend explicit conversion.

Implicit data type conversion

The following rules apply:

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

```
Command> CREATE TABLE convdemo (c1 CHAR (10), x1 TT_INTEGER);
Command> CREATE TABLE convdemo2 (c1 NCHAR (10), x2 TT_INTEGER);
Command> INSERT INTO convdemo VALUES ('ABC', 10);
1 row inserted.
Command> INSERT INTO convdemo VALUES ('def', 100);
1 row inserted.
Command> INSERT INTO convdemo2 SELECT * FROM convdemo;
2 rows inserted.
Command> SELECT x1,x2,convdemo.c1, convdemo2.c1
       > FROM convdemo, convdemo2 where Ccnvdemo.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. Any column in a table or any parameter in an expression, regardless of its data type, can contain NULL unless you specify NOT NULL for the column when you create the table.

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

- NULL always sort highest in a sequence of values.
- Two NULL values are not equal to each other except in a GROUP BY or SELECT DISTINCT operation.
- An expression containing a NULL evaluates to NULL. For example, (5-col), where col is NULL, evaluates to NULL.

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

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

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

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 allowable in its type. Arithmetic operations are defined in Chapter 3, "Expressions."
- When aggregate functions are used, overflow can occur when the sum of several numbers exceeds the maximum allowable value of the result type. Aggregate functions are defined in Chapter 3, "Expressions."
- 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:

- 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 non-zero digit is dropped from the fractional part of a numeric value.

Underflow

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

Replication limits

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

- `VARCHAR2` and `VARBINARY` columns cannot exceed four megabytes. For character-length semantics, the limit is four megabytes. The database character set determines how many characters can be represented by four megabytes. The minimum number of characters is 1,000,000/ 4 = 250,000 characters.
- `NVARCHAR2` columns cannot exceed 500,000 characters (four megabytes).

TimesTen type mode (backward compatibility)

TimesTen supports a data type backward compatibility mode called TimesTen type mode. This is specified using the data store creation attribute `TypeMode`, where `TypeMode=1` indicates TimesTen mode.

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

For information on data type usage in TimesTen type mode, refer to *Oracle TimesTen In-Memory Database API and SQL Reference Guide*, Release 6.0.3.

Data types supported in TimesTen type mode

Table 1–6 Data types supported in TimesTen type mode

Data type	Description
BIGINT	A signed eight-byte integer in the range -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 (2^{63}-1). Alternatively, specify TT_BIGINT.
BINARY(*n*)	Fixed-length binary value of *n* bytes. Legal values for *n* range from 1 to 8300. BINARY data is padded to the maximum column size with trailing zeroes.
BINARY_DOUBLE	A 64-bit floating-point number. BINARY_DOUBLE is a double-precision native floating point number. Supports +Inf, -Inf, and NaN values. BINARY_DOUBLE is an approximate numeric value consisting of an exponent and mantissa. You can use exponential or E-notation. BINARY_DOUBLE has binary precision 53. Minimum positive finite value: 2.22507485850720E-308 Maximum positive finite value: 1.79769313486231E+308 Alternatively, specify DOUBLE [PRECISION] or FLOAT[(53)].
BINARY_FLOAT	A 32-bit floating-point number. BINARY_FLOAT is a single-precision native floating-point type. Supports +Inf, -Inf, and NaN values. BINARY_FLOAT is an approximate numeric value consisting of an exponent and mantissa. You can use exponential or E-notation. BINARY_FLOAT has binary precision 24. Minimum positive finite value: 1.17549E-38F Maximum positive finite value: 3.40282E+38F Alternatively, specify REAL or FLOAT(24).
CHAR[ACTER][(*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. Legal values for *n* bytes range from 1 to 8300. 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. A zero-length string is a valid non-null value. CHAR data is padded to the maximum column size with trailing blanks. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28. Alternatively, specify TT_CHAR[(*n* [BYTE|CHAR])].
DATE	Stores date information: century, year, month, date. The format is *YYYY-MM-DD*, where *MM* is expressed as an integer. For example: 2006-10-28. Storage size is four bytes. Valid dates are between 1753-01-01 (January 1,1753) and 9999-12-31 (December 31, 9999). Alternatively, specify TT_DATE.

Table 1–6 (Cont.) Data types supported in TimesTen type mode

Data type	Description
`DEC[IMAL][(p[,s])]` or `NUMERIC[(p[,s])]`	An exact numeric value with a fixed maximum precision (total number of digits) and scale (number of digits to the right of the decimal point). The value of precision *p* must be between 1 and 40. The value of scale *s* must be between 0 and *p*. The default precision is 40 and the default scale is 0.
`INTERVAL [+/-]` *`IntervalQualifier`*	TimesTen partially supports interval types, expressed with `INTERVAL` and an *`IntervalQualifier`*. An *`IntervalQualifier`* can specify only a single field type with no precision. The default leading precision is eight digits for all interval types. The single field type can be one of: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, or `SECOND`. Currently, interval types can be specified only with a constant.
`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. Default and minimum bytes of storage is 2*n* (2). A zero-length string is a valid non-null value. `NCHAR` data is padded to the maximum column size with `U+0020 SPACE`. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28. Alternatively, specify `TT_NCHAR[(n)]`. `NATIONAL CHARACTER` and `NATIONAL CHAR` are synonyms for `NCHAR`.
`SMALLINT`	A native signed 16-bit integer in the range -32,768 (-2^{15}) to 32,767 (2^{15}-1). Alternatively, specify `TT_SMALLINT`.
`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`*. Storage size is eight bytes.
`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`* [*`.FFFFFF`*]. Storage size is eight bytes. Alternatively, specify `TT_TIMESTAMP` or `[TT_]TIMESTAMP(6)`.
`TINYINT`	Unsigned integer ranging from 0 to 255 (2^8-1). Since `TINYINT` is unsigned, the negation of a `TINYINT` is `SMALLINT`. Alternatively, specify `TT_TINYINT`.
`INT[EGER]`	A signed integer in the range -2,147,483,648 (-2^{31}) to 2,147,483,647 (2^{31}-1). Alternatively, specify `TT_INTEGER`.

Table 1–6 (Cont.) Data types supported in TimesTen type mode

Data type	Description
`NVARCHAR(n)`	Variable-length string of *n* two-byte Unicode characters. The number of bytes required is 2**n* where *n* is the specified number of characters. `NVARCHAR` character limits are half the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify *n*. A zero-length string is a valid non-null value. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28. Alternatively, specify `TT_NVARCHAR(n)`. `NATIONAL CHARACTER VARYING`, `NATIONAL CHAR VARYING`, and `NCHAR VARYING` are synonyms for `NVARCHAR`.
`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. Legal values for *n* bytes range from 1 to 4194304 (2^{22}). `CHAR` indicates that the column has character-length semantics. A zero-length string is a valid non-null value. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28. Alternatively, specify `TT_VARCHAR(n[BYTE|CHAR])`.
`VARBINARY(n)`	Variable-length binary value having maximum length *n* bytes. Legal values for *n* range from 1 to 4194304 (2^{22}).

Oracle data types supported in TimesTen type mode

Table 1–7 Oracle data types supported in TimesTen type mode

Data type	Description
NUMBER[(*p* [,*s*])]	Number having precision and scale. The precision value ranges from 1 to 38 decimal. The scale value ranges from -84 to 127. Both precision and scale are optional. If you do not specify a precision or a scale, then maximum precision of 38 and flexible scale are assumed. NUMBER supports scale > precision and negative scale. NUMBER stores zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} up to but not including 1.0×10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0×10^{126}, then TimesTen returns an error. In TimesTen type mode, the NUMBER data type stores 10E-89 as its smallest (closest to zero) value.
ORA_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. Legal values for *n* bytes range from 1 to 8300. 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. A zero-length string is interpreted as NULL. ORA_CHAR data is padded to the maximum column size with trailing blanks. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28.
ORA_DATE	Stores date and time information: century, year, month, date, hour, minute, and second. Format is *YYYY-MM-DD HHMMSS*. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. The storage size is seven bytes. There are no fractional seconds.
ORA_NCHAR[(*n*)]	Fixed-length string of length *n* two-byte Unicode characters. The number of bytes required is 2**n* where *n* is the specified number of characters. NCHAR character limits are half the byte limits so the maximum size is 4150. Default and minimum bytes of storage is 2*n* (2). A zero-length string is interpreted as NULL. ORA_NCHAR data is padded to the maximum column size with U+0020 SPACE. Blank-padded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28.

Table 1–7 (Cont.) Oracle data types supported in TimesTen type mode

Data type	Description
`ORA_NVARCHAR2(n)`	Variable-length string of *n* two-byte Unicode characters. The number of bytes required is 2**n* where *n* is the specified number of characters. `ORA_NVARCHAR2` character limits are half the byte limits so the maximum size is 2,097,152 (2^{21}). You must specify *n*. A zero-length string is interpreted as `NULL`. Nonpadded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28.
`ORA_VARCHAR2(n[BYTE|CHAR])`	Variable-length character string having maximum length *n* bytes or characters. `BYTE` indicates that the column has byte-length semantics. Legal values for *n* bytes range from 1 to 4194304 (2^{22}). You must specify *n*. `CHAR` indicates that the column has character-length semantics. A zero-length string is interpreted as `NULL`. Nonpadded comparison semantics are used. For information on blank-padded and nonpadded semantics, see "Blank-padded and nonpadded comparison semantics" on page 1-28.
`ORA_TIMESTAMP` `[(fractional_seconds_precision)]`	Stores year, month, and day values of the date data type plus hour, minute, and second values of time. *Fractional_seconds_precision* is the number of digits in the fractional part of the seconds field. Valid date range is from January 1, 4712 BC to December 31, 9999 AD. The fractional seconds precision range is 0 to 9. The default is 6. Format is: *YYYY-MM-DD HH:MI:SS* `[.FFFFFFFFF]` Storage size is 12 bytes.

2

Names and Parameters

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

- Basic names
- Owner names
- Compound identifiers
- Dynamic parameters
- Duplicate parameter names
- Inferring data type from parameters

Basic names

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

- The maximum length of a basic name is 30 characters.
- A name can consist of any combination of letters (A to Z a to z), decimal digits (0 to 9), $, #, @, or underscore (_). For identifiers, the first character must be a letter (A-Z a-z) and not a digit or special character. However, for parameter names, the first character can be a letter (A-Z a-z), a decimal digit (0 to 9), or special characters $, #, @, or underscore (_).
- 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 legal 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 table owner names must always match TimesTen table owner names.

Owner names may be specified by the user during table creation, in addition to being automatically determined if they are left unspecified. See "CREATE TABLE" on page 5-107. When creating owner names, follow the same rules as those for creating basic names. See "Basic names" on page 2-1.

Compound identifiers

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

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

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

The following are compound identifiers:

- *Column identifier*: `[[`*Owner*`.]`*TableName*`.]`*ColumnName*
- `[`*Owner*`.]`*IndexName*
- *Table identifier*: `[`*Owner*`.]`*TableName*
- *Row identifier*: `[[`*Owner*`.]`*TableName*`.]`*rowid*

Dynamic parameters

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

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

- 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 assigns a number to each parameter occurrence without regard to name duplication. An Oracle 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 *`		
`FROM t1`		
`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 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

ROWID specification

TimesTen assigns a unique ID called a *rowid* to each row stored in a table. The rowid has data type `ROWID`. You can examine a rowid by querying the `ROWID` pseudocolumn.

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

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

Rowids persists through recovery, backup and restore operations. They do not persist through replication, `ttMigrate` or `ttBulkCp` operations.

See "Expression specification" on page 3-3 for more information on rowids. See "ROWID data type" on page 1-21 for more information about the `ROWID` data type.

ROWNUM specification

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

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

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

The order in which rows are selected depends on the index used and 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
```

Expression specification

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

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

SQL syntax

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

or

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

or

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

or

```
Expression1 | | Expression2
```

or

```
Expression
```

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. Column names are discussed in Chapter 2, "Names and Parameters."
`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.
`?` *:DynamicParameter*	A placeholder for a dynamic parameter. The value of the dynamic parameter is supplied at runtime.
AggregateFunction	A computed value. See "Aggregate functions" on page 3-8.
Constant	A specific value. See "Constants" on page 3-11.
(*Expression*)	Any expression enclosed in parentheses.
Expression1 *Expression2*	*Expression1* and *Expression2*, when used with the bitwise operators, can be of integer or binary types. The data types of the expressions must be compatible. See Chapter 1, "Data Types."
`*`	Multiplies two primaries.
`/`	Divides two primaries.
`+`	Adds two primaries.
–	Subtracts two primaries.

Component	Description
&	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.
||	Concatenates *Expression1* and *Expression2*, where both expressions are character strings. Forms a new string value that contains the values of both expressions. See also "CONCAT" on page 3-36.

Description

- Arithmetic operators can be used between numeric values. See "Numeric data types" on page 1-15.
- 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:
  - Aggregate functions and expressions in parentheses
  - Unary pluses and minuses
  - The * and / operations
  - The + and – operations
  - Elements of equal precedence are evaluated in left-to-right order
- You can enclose expressions in parentheses to control the order of their evaluation. For example:

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

- Type conversion, truncation, underflow, or overflow can occur when some expressions are evaluated. See Chapter 1, "Data Types".
- If either operand in a numeric expression is NULL, the result is NULL.
- Since NVL takes two parameters, both designated as an "expression", TimesTen does not permit NULL in either position. If there is a NULL value in an expression, comparison operators and other predicates evaluate to NULL. See Chapter 4, "Search Conditions" for more information on evaluation of comparison operators and predicates containing NULL values. 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.

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. It 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)
```

Description

TimesTen supports queries with the characteristics listed in each section.

Table subqueries

- A subquery can appear in the WHERE clause or HAVING clause of any statement except one that creates a materialized view. Only one table subquery can be specified in a predicate. These predicates can be specified in a WHERE or HAVING clause, an OR 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.

Scalar subqueries

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.
- Neither outer query nor any scalar subquery should have a DISTINCT modifier.

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);
```

Aggregate functions

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

Aggregate functions can be specified in the select list or the HAVING clause. See "INSERT...SELECT" on page 5-150 for more information. The value of the expression is computed using each row that satisfies the WHERE clause.

SQL syntax

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

Component	Description
Expression	Specifies an argument for the aggregate function. The expression itself cannot be an aggregate function.
AVG	Computes the arithmetic mean of the values in the argument. NULL values are ignored. AVG can be applied only to numeric data types.
MAX	Finds the largest of the values in the argument (ASCII comparison for alphabetic types). NULL values are ignored. MAX can be applied to numeric, character, and BINARY data types.
MIN	Finds the smallest of the values in the argument (ASCII comparison for alphabetic types). NULL values are ignored. MIN can be applied to numeric, character, and BINARY data types.
SUM	Finds the total of all values in the argument. NULL values are ignored. SUM can be applied to numeric data types only.
COUNT *	Counts all rows that satisfy the WHERE clause, including rows containing NULL values. The data type of the result is TT_INTEGER. For more information on the number of rows in a table, see the description for the NUMTUPS field in "SYS.TABLES" in *Oracle TimesTen In-Memory Database System Tables and Limits Reference*.
COUNT *ColumnName*	Counts all rows in a specific column. Rows containing NULL values are not counted. The data type of the result is TT_INTEGER. For more information on the number of rows in a table, see the description for the NUMTUPS field in "SYS.TABLES" in *Oracle TimesTen In-Memory Database System Tables and Limits Reference*.
ALL	Includes any duplicate rows in the argument of an aggregate function. If neither ALL nor DISTINCT is specified, ALL is assumed.
DISTINCT	Eliminates duplicate column values from the argument of an aggregate function. Can be specified for more than one column.

Description

- If an aggregate function is computed over an empty table in which GROUP BY is not used, the results are as follows:
  - COUNT returns 0.
  - AVG, SUM, MAX, and MIN return NULL.

- If an aggregate function is computed over an empty group or an empty grouped table (GROUP BY is used):
  - COUNT returns nothing.
  - AVG, SUM, MAX, and MIN return nothing.
- For SUM:
  - If the source is TT_TINYINT, TT_SMALLINT, or TT_INTEGER, the result data type is TT_INTEGER.
  - If the source is NUMBER, then the result data type is NUMBER with undefined scale and precision.
  - If the source is TT_DECIMAL, then the result data type is TT_DECIMAL with maximum precision.
  - For all other data types, the result data type is the same as the source.
- For MAX and MIN:
  - The result data type is the same as the source.
- For AVG:
  - AVG is evaluated as SUM/COUNT. The result data type is derived using the rule that is applied for the DIV operator.

See Chapter 1, "Data Types" for information about:

- 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.

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 >
```

Calculate the MAX salary for employees in the HR schema:

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

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

```
Command> CREATE TABLE aggregates (col1 TT_TINYINT, col2 TT_SMALLINT,
       > col3 TT_INTEGER, col4 TT_BIGINT, col5 NUMBER (4,2),
       > col6 TT_DECIMAL (6,2), col7 BINARY_FLOAT, col8 BINARY_DOUBLE);
Command> DESCRIBE SELECT SUM (col1) FROM aggregates;
Prepared Statement:
  Columns:
    EXP                             TT_INTEGER
Command> DESCRIBE SELECT SUM (col2) FROM aggregates;
Prepared Statement:
Columns:
```

```
    EXP                            TT_INTEGER
Command> DESCRIBE SELECT SUM (col3) FROM aggregates;
Prepared Statement:
Columns:
    EXP                            TT_INTEGER
Command> DESCRIBE SELECT SUM (col4) FROM Aggregates;
Prepared Statement:
Columns:
    EXP                            TT_BIGINT
Command> DESCRIBE SELECT SUM (col5) FROM aggregates;
Prepared Statement:
Columns:
    EXP                            NUMBER
Command> DESCRIBE SELECT SUM (col6) FROM aggregates;
Prepared Statement:
Columns:
    EXP                            TT_DECIMAL (40,2)
Command> DESCRIBE SELECT SUM (col7) FROM aggregates;
Prepared Statement:
Columns:
    EXP                            BINARY_FLOAT
Command> DESCRIBE SELECT SUM (col8) FROM Aagregates;
Prepared Statement:
  Columns:
    EXP                            BINARY_DOUBLE
```

Constants

A constant is a literal value.

SQL syntax

```
{IntegerValue | FloatValue |FloatingPointLiteral|
    FixedPointValue | 'CharacterString'|
    'NationalCharacterString' | 0xHexadecimalString |
    'DateString' | DateLiteral |'TimeString' |
    TimeLiteral | 'TimestampString' | TimestampLiteral |
    IntervalLiteral | BINARY_FLOAT_INFINITY |
    BINARY_DOUBLE_INFINITY | -BINARY_FLOAT_INFINITY |
    -BINARY_DOUBLE_INFINITY | BINARY_FLOAT_NAN |
     BINARY_DOUBLE_NAN
}
```

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 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 `\uxxxx`, where *xxxx* is the four hex-digit representation of the Unicode character. For example:
	`N'This is an \u0061'`
	
	is equivalent to:
	`N'This is an a'`
	
	The `\u` itself can be escaped with another `\`. The sequence `\\u` is always converted to `\u`. No other escapes are recognized.

HexadecimalString	A string of hexadecimal digits 0 - 9 and A - F (or a - f) compatible with the BINARY, VARBINARY, CHAR and VARCHAR2 data types. A *HexadecimalString* constant must be prefixed with the characters "0x." For example:
	`0xFFFAB0880088343330FFAA7`
	or
	`0x000A001231`

DateString	A string of the format YYYY-MM-DD HH:MI:SS enclosed in single quotation marks ('). For example:
	`'2007-01-27 12:00:00'`
	
	The YYYY field must have a 4-digit value. The MM and DD fields must have 2-digit values. The only spaces allowed are trailing spaces (after the day field). The range is from `'-4713-01-01'` (January 1, 4712 BC) to `'9999-12-31'`, (December 31, 9999). The time component is not required. For example:
	`'2007-01-27'`
	
	For TT_DATE data types, the string is of format YYYY-MM-DD and ranges from `'1753-01-01'` to `'9999-12-31'`.
	If you are using TimesTen type mode, see Oracle TimesTen In-Memory Database Release 6.0.3 documentation for information about *DateString*.

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.
	If you are using TimesTen type mode, see Oracle TimesTen In-Memory Database Release 6.0.3 documentation for information about *`DateLiteral`*.

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

TimeLiteral	Format: `TIME` *`TimeString`*. For example:
	`TIME '20:25:30'`
	
	The `TIME` keyword is case-insensitive.
	Usage examples:
	`INSERT INTO timetable VALUES (TIME '10:00:00');`
	
	`SELECT * FROM timetable WHERE col1 < TIME '10:00:00';`
	
	TimesTen also supports ODBC time-literal syntax.
	For example:
	`{t '12:00:00'}`

Constant	Description
TimestampString	A string of the format YYYY-MM-DD HH:MM:SS[.FFFFFFFFF] -enclosed in single quotation marks('). The range is from '-4713-01-01' (January 1, 4712 BC) to '9999-12-31' (December 31, 9999). The year field must be a 4-digit value. All other fields except for the fractional part must be 2-digit values. The fractional field can consist of 0 to 9 digits. For TT_TIMESTAMP data types, a string of format YYYY-MM-DD HH:MM:SS[.FFFFFF] enclosed in single quotation marks('). The range is from '1753-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'. The fractional field can consist of 0 to 6 digits. If you have a CHAR column called C1, and want to enforce the TIME comparison, you can do the following: `SELECT * FROM testable WHERE C1 = TIME '12:00:00'` In this example, each CHAR value from C1 is converted into a TIME value before comparison, provided that values in C1 conform to the proper TIME syntax. If you are using TimesTen type mode, see Oracle TimesTen In-Memory Database Release 6.0.3 documentation for information on *TimestampString*.

TimestampLiteral	Format: TIMESTAMP *TimestampString*
	
	For example:
	
	`TIMESTAMP '2007-01-27 11:00:00.000000'`
	
	For TIMESTAMP data types, the fraction field supports from 0 to 9 digits of fractional seconds. For TT_TIMESTAMP data types, the fraction field supports from 0 to 6 digits of fractional seconds.
	
	The TIMESTAMP keyword is case-insensitive.
	
	Literal syntax can be used if you want to enforce DATE/TIME/TIMESTAMP comparisons for CHAR and VARCHAR2 data types.
	
	TimesTen also supports ODBC timestamp literal syntax. For example:
	
	`{ts '9999-12-31 12:00:00'}`
	
	If you are using TimesTen type mode, see Oracle TimesTen In-Memory Database Release 6.0.3 documentation for information about *TimestampLiteral*.

IntervalLiteral	Format: INTERVAL [+\-] *CharacterString* *IntervalQualifier*. For example INTERVAL '8' DAY
BINARY_FLOAT_INFINITY| BINARY_DOUBLE_INFINITY	INF (positive infinity) is an IEEE floating-point value that is compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. Use the constant values BINARY_FLOAT_INFINITY or BINARY_DOUBLE_INFINITY to represent positive infinity.
-BINARY_FLOAT_INFINITY| -BINARY_DOUBLE_INFINITY	-INF (negative infinity) is an IEEE floating-point value that is compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. Use the constant values -BINARY_FLOAT_INFINITY and -BINARY_DOUBLE_INFINITY to represent negative infinity.

Constant	Description
BINARY_FLOAT_NAN| BINARY_DOUBLE_NAN	NaN ("not a number") is an IEEE floating-point value that is compatible with the BINARY_FLOAT and BINARY_DOUBLE data types. Use the constant values BINARY_FLOAT_NAN or BINARY_DOUBLE_NAN to represent NaN ("not a number").

Format models

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

Number format models

Use number format models in the following functions:

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

Number format elements

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

The default `american_america` NLS language and territory setting is used.

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.
`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.

Table 3–1 (Cont.) Number format elements

Element	Example	Description
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.
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.

Table 3–1 (Cont.) Number format elements

Element	Example	Description
X	XXXX	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, then TimesTen rounds it to an integer. **Restrictions**: ■ This element accepts only positive values or 0. Negative values return an error. ■ You can precede this element only with 0 (which returns leading zeros) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, then the return always has a leading blank.

Datetime format models

Use datetime format models in the following functions:

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

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

The default `american_america` NLS language and territory setting is used.

Datetime format elements

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

Table 3–2 Datetime format elements

Element	Description
`-/,.;:`"text"	Punctuation and quoted text is reproduced in the result.
`AD` `A.D.`	AD indicator with or without periods.
`AM` `A.M.`	Meridian indicator with or without periods.
`BC` `B.C.`	BC indicator with or without periods.
`D`	Day of week (1-7).
`DAY`	Name of day, padded with blanks to display width of widest name of day.
`DD`	Day of month (1-31).
`DDD`	Day of year.
`DL`	Returns a value in the long date format. In the default `AMERICAN_AMERICA` locale, this is equivalent to specifying the format '*`fmDay`*, *`Month dd`*, *`yyyy`*'. **Restriction**: Specify this format only with the `TS` element, separated by white space.
`DS`	Returns a value in the short date format. In the default `AMERICAN_AMERICA` locale, this is equivalent to specifying the format `'MM/DD/RRRR'`. **Restriction**: Specify this format only with the `TS` element, separated by white space.
`DY`	Abbreviated name of day.
`FM`	Returns a value with no leading or trailing blanks.
`FX`	Requires exact matching between the character data and the format model.
`HH`	Hour of day (1-12).
`HH24`	Hour of day (0-23).
`J`	Julian day: The number of days since January 1, 4712 BC. Numbers specified with `J` must be integers.
`MI`	Minute (0-59).
`MM`	Month (01-12. January = 01).
`MON`	Abbreviated name of month.
`MONTH`	Name of month padded with blanks to display width of the widest name of month.
`RM`	Roman numeral month (I-XII. January = I).
`RR`	Stores 20th century dates in the 21st century using only two digits.

Table 3–2 (Cont.) Datetime format elements

Element	Description
RRRR	Rounds year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you do not want this functionality, then enter the 4-digit year.
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
TS	Returns a value in the short time format. **Restriction**: Specify this format only with the DL or DS element, separated by white space.
X	Local radix character. Example: 'HH:MI:SSXFF'.
Y,YYY	Year with comma in this position.
YYYY SYYYY	4-digit year. S prefixes BC dates with a minus sign.
YYY YY Y	Last 3, 2, or 1 digit (s) of year.

Format model for 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 2 digits of a 4-digit year are between 01 and 99 (inclusive), then the century is one greater than the first 2 digits of that year. If the last 2 digits of a 4-digit year are 00, then the century is the same as the first 2 digits of that year. S prefixes digits with minus sign for B.C. For example, 2002 returns 21; 2000 returns 20.
SYYYY	Year. S prefixes digits with minus sign for B.C.
YYYY	Year
YEAR	Year spelled out in words. For example, TWENTY TEN.
SYEAR	Year spelled out in words. S prefixes digits with minus sign for B.C.
YYY	Last 3 digits of year
YY	Last 2 digits of year
Y	Last digit of year
IYYY	ISO year, 4 digits
IYY	Last 3 digits of ISO year
IY	Last 2 digits of ISO year
I	Last digit of ISO year
Q	Quarter
MONTH	Name of month
MON	Abbreviation of name of month
MM	Month (01-12)
RM	Roman numeral month (I - XII)
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD	Day of year (1-366)
DD	Day of month (1-31)
J	Julian day (the number of days since January 1, 4712 B.C.)
DAY	Starting day of the week
DY	Abbreviation of name of day
D	Day of week (1-7)
HH	Hour of day (1-12)
HH12	Hour of day (1-12)
HH24	Hour of day (0-23)
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)
SS	Second (0-59)
FF	Fractions of a second to a precision of 6 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

Format	Description
,	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.

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 TT_DECIMAL or 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.
```

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 implicitly converted to DATE.
integer	An integer or any value that can be implicitly converted to an integer.

Description

- The return type is always DATE regardless of the data type of *date*. Supported data types are DATE and TIMESTAMP.
- Data types 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 an expression that resolves to a string, in any character set, and returns the ASCII version of the string in the database character set. Non-ASCII characters are converted to Unicode escapes.

SQL syntax

```
ASCIISTR ([N]'String')
```

Parameters

ASCIISTR has the parameter:

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

Description

The ASCIISTR function allows you to see the representation of a string value that is not in 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;TypeMode=0;
(Default setting AutoCommit=1)
Command> SELECT ASCIISTR (n'Aäa') FROM dual;
< A\00E4a >
1 row found.
```

CASE

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

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

SQL syntax

The syntax for a searched CASE expression is:

```
CASE
  {WHEN SearchCondition THEN Expression1}[…]
   [ELSE Expression2]
END
```

The syntax for a simple CASE expression is:

```
CASE Expression
  {WHEN CompExpression THEN Expression1}[…]
   [ELSE Expression2]
END
```

Parameters

CASE has the parameters:

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

Description

CASE expression cannot be specified 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:

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

CAST

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

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

The following conversions are supported:

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

SQL syntax

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

Parameters

CAST has the parameters:

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

Description

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

Examples

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

CHR

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

SQL syntax

CHR(*n*)

Parameters

CHR has the parameter:

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

Description

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

Examples

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

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

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 TT_DECIMAL or 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 2 expressions.

Description

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

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

  is equivalent to:

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

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

Description

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

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

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.
```

The description of the || operator is in "Expression specification" on page 3-3.

DECODE

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

SQL syntax

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

Parameters

DECODE has the parameters:

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

Description

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

Examples

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

```
Command> SELECT location_id,
       > DECODE (country_id, 'IT', 'Italy',
       >                          'Other')
       > FROM locations WHERE location_id < 2000;
LOCATION_ID, EXP
< 1000, Italy >
< 1100, Italy >
< 1200, Japan >
< 1300, Japan >
< 1400, United States >
< 1500, United States >
< 1600, United States >
< 1700, United States >
< 1800, Other >
< 1900, Other >
10 rows found.
```

EXTRACT

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

If you are using TimesTen type mode, see the Oracle TimesTen In-Memory Database Release 6.0.3 documentation for information about the EXTRACT function.

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

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.
```

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 TT_DECIMAL or 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.
```

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.
```

LOWER and UPPER

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

SQL syntax

```
{UPPER | LOWER} (Expression1)
```

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 or NVARCHAR2 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 LPAD function. Must be a NUMBER integer or a value that can be implicitly converted to a NUMBER integer.
Expression2	Sequence of characters left-padded to *Expression1*. If you do not specify *Expression2*, then the default is a single blank. Operand or column can be of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

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.
- 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, and NVARCHAR2 types:
  - If either *Expression1* or *Expression2* is NULL, the result is NULL. If *n* is less than or equal to 0, then 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, then the result is the empty string.

Examples

Use LPAD function to left-pad the string 'LPAD Function' with string 'DEMO-ONLY' plus 2 spaces. Replicate string DEMO-ONLY plus 2 spaces 3 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. You see 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 or NVARCHAR2 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, then 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, or NVARCHAR2.

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. 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, and NVARCHAR2 types:
  - If all the characters in *Expression1* are removed by the LTRIM function, then 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, then 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.
```

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.
Expression2	An INTEGER expression.

Description

- MOD returns the remainder of *Expression1* divided by *Expression2*.
- If *Expression2* is 0, then MOD returns *Expression1*.
- If either *Expression1* or *Expression2* is NULL, MOD returns NULL.
- MOD is treated as a binary arithmetic operation, so the return type is determined according to the rules specified in Chapter 1, "Data Types".
- The MOD function behaves differently from the 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;
```

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.

The following example returns the NCHAR character 187:

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

Example

The following example returns the NCHAR character 187:

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

NLSSORT

Returns the sort key value for the given string.

SQL syntax

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

Parameters

NLSSORT has the following parameters:

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

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 Nnme > 'Gaberd';
0 rows found.
```

```
Command> #Use function in comparison operation
Command> SELECT * FROM nsortdemo WHERE NLSSORT (name, 'NLS_SORT = XDanish') >
> NLSSORT ('Gaberd', 'NLS_SORT = XDanish');
< Gaardiner >
< Gaasten >
2 rows found.
Command> #Use ALTER SESSION to obtain the same results
Command> ALTER SESSION SET NLS_SORT = 'XDanish';
Command> SELECT * FROM nsortdemo ORDER BY name;
< Gaberd >
< Gaardiner >
< Gaasten >
3 rows found.
Command> SELECT * FROM nsortdemo where name > 'Gaberd';
< Gaardiner >
< Gaasten >
2 rows found.
```

The following example creates a linguistic index:

```
Command> CREATE INDEX danishindex
       > ON nsortdemo (NLSSORT (name, 'NLS_SORT =XDanish'));
Command> INDEXES N%;
Indexes on table USER1.NSORTDEMO:
  DANISHINDEX: non-unique T-tree index on columns:
    NLSSORT(NAME,'NLS_SORT = XDanish')
  1 index found.
1 table found.
```

NUMTODSINTERVAL

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

SQL syntax

```
NUMTODSINTERVAL (Expression1, IntervalUnit)
```

Parameters

`NUMTODSINTERVAL` has the parameters:

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

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.
Expression2	The alternate value to use if the value of *Expression1* is NULL.

Description

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

Examples

This example checks for null values of commission_pct and replaces them with 'Not Applicable' for employees whose last name 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 or TT_DECIMAL, 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.
```

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" on page 3-23.

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*, and *Expression1* is of type TT_DECIMAL, the data type returned is NUMBER with maximum precision and scale. Otherwise, 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 2 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.
```

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 or NVARCHAR2 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 or NVARCHAR2 operand or column to be right-padded to *Expression1*. If you do not specify *Expression2*, then 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.
- 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, and NVARCHAR2 types:
  - If either *Expression1* or *Expression2* is NULL, the result is NULL. If *n* is less than or equal to 0, then 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, then 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 5 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 5 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* backwards 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 or NVARCHAR2 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, then 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, or NVARCHAR2.

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. 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, and NVARCHAR2 types:
  - If all the characters in *Expression1* are removed by the RTRIM function, then 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, then 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.
```

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 or TT_DECIMAL, 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.
```

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 or TT_DECIMAL, 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 or TT_DECIMAL, 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.
```

String functions

TimesTen supports these string functions in `SELECT` statements:

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

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

SUBSTR, SUBSTRB, SUBSTR4

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

SQL syntax

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

Parameters

SUBSTR has the parameters:

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

Description

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

Examples

In the first 5 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 5 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.
```

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, then `0` is returned. The position returned is always relative to the beginning of *`CharExpr2`*. `INSTR` returns type `NUMBER`.

If you are using TimesTen type mode, for information on the `INSTR` function, see the Oracle TimesTen In-Memory Database Release 6.0.3 documentation.

SQL syntax

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

Parameters

`INSTR` has the parameters:

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

Description

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

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.
```

LENGTH, LENGTHB, LENGTH4

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

If you are using TimesTen type mode, for information on the `LENGTH` function, see the Oracle TimesTen In-Memory Database Release 6.0.3 documentation.

SQL syntax

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

Parameters

`LENGTH` has the parameter:

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

Description

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

Examples

Determine the length of the string `'William'`:

```
Command> SELECT LENGTH('William') FROM dual;
< 7 >
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: ■ AUTHENTICATION_METHOD ■ CURRENT_USER ■ CURRENT_USERID ■ IDENTIFICATION_TYPE ■ LANG ■ LANGUAGE ■ NLS_SORT ■ SESSION_USER ■ SESSION_USERID ■ SID
length	Number between 1 and 4000 bytes.

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

Parameter	Description
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
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 cursor that is using the view. It does not respect views used in the cursor as being definer's rights.
CURRENT_USERID	The identifier of the database user whose privileges are currently active

Parameter	Description
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*
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;
< TTUSER >
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:MM:SS. The date represents the local current date and time, which is determined by the system on which the statement is executed.

If you are using TimesTen type mode, for information on SYSDATE, see the Oracle TimesTen In-Memory Database Release 6.0.3 documentation.

SQL syntax

```
SYSDATE | GETDATE( )
```

Parameters

The SYSDATE and GETDATE functions have no parameters.

Description

- SYSDATE and GETDATE perform identically. SYSDATE is compatible with Oracle syntax. GETDATE is compatible with Microsoft SQL Server syntax.
- SYSDATE and GETDATE have no arguments, and return a DATE value.
- The SYSDATE or GETDATE value is only retrieved during execution.
- Any required changes to the date (to incorporate a different time zone or Daylight Savings Time, for example) must occur at the system level. The date cannot be altered using SYSDATE or GETDATE.
- The SYSDATE and GETDATE functions return the DATE data type. The DATE format is 'YYYY-MM-DD HH:MM:SS'.
- SYSDATE and GETDATE are built-in functions and can be used anywhere a date expression may be used. They can be used in a 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.
```

TO_CHAR

The TO_CHAR function converts a DATE, TIMESTAMP or numeric input value to a VARCHAR2.

If you are using TimesTen type mode, for information on the TO_CHAR function, see the Oracle TimesTen In-Memory Database Release 6.0.3 documentation.

SQL syntax

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

Parameters

TO_CHAR has the parameters:

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

Description

- TO_CHAR supports different datetime format models depending on the data type specified for the expression. For information on the datetime format model used for TO_CHAR of data type DATE or TIMESTAMP, see "Datetime format models" on page 3-20. For information on the datetime format model used for TO_CHAR of data type TT_DATE or TT_TIMESTAMP, see "Format model for ROUND and TRUNC date functions" on page 3-23.
- TO_CHAR supports different number format models depending on the numeric data type specified for the expression. For information on the number format model used for TO_CHAR of data type NUMBER or ORA_FLOAT, see "Number format models" on page 3-17. For information on the number format model used for TO_CHAR of all other numeric data types, see "Format model for ROUND and TRUNC date functions" on page 3-23.

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_DATE

The TO_DATE function converts a CHAR or VARCHAR2 argument to a value of DATE data type

If you are using TimesTen type mode, for information on the TO_DATE function, see the Oracle TimesTen In-Memory Database Release 6.0.3 documentation.

SQL syntax

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

Parameters

TO_DATE has the parameters:

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

Description

You can use a datetime format model with the TO_DATE function. For more information on datetime format models, see "Datetime format models" on page 3-20.

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') FROM dual;
< DECEMBER ,23 1997 10:56-20 >
1 row found.
```

TO_NUMBER

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

SQL syntax

```
TO_NUMBER (Expression[, format])
```

Parameters

TO_NUMBER has the parameters:

Parameter	Description
Expression	The expression to be converted.
format	If specified, the format is used to convert *Expression* to a value of NUMBER type. The format consists of a format string that identifies the number format model. The format string can be either a constant or a parameter.

Description

You can use a number format model with the TO_NUMBER function. For more information on number format models, see "Number format models" on page 3-17.

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 valid 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*:

  TRIM (LEADING|TRAILING|BOTH FROM *Expression*)

- You can specify the *Trim_character* without one of the TRIM qualifiers:

  TRIM (*Trim_character* FROM *Expression*)

- You can specify the *Expression* without a qualifier or a *Trim_character*:

  TRIM (*Expression*)

Parameters

TRIM has the parameters:

Parameter	Description
TRIM (LEADING | TRAILING | BOTH [*Trim_character*] FROM *Expression*)	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* is optional. If specified, it represents the CHAR, VARCHAR2, NCHAR or NVARCHAR2 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, then enclose it in single quotes. FROM is required. *Expression* is the CHAR, VARCHAR2, NCHAR or NVARCHAR2 operand or column to be trimmed. If *Expression* is a character literal, then enclose it in single quotes.

Parameter	Description
TRIM (*Trim_character* FROM *Expression*)	Removes both leading and trailing instances of *Trim_character* from *Expression*. *Trim_character* is the CHAR, VARCHAR2, NCHAR or NVARCHAR2 operand or column used for trimming *Expression*. Must be only one character. If *Trim_character* is a character literal, then enclose it in single quotes. FROM must follow *Trim_character*. Assumes LEADING | TRAILING | BOTH qualifiers have not been specified. *Expression* is the CHAR, VARCHAR2, NCHAR or NVARCHAR2 operand or column to be trimmed. If *Expression* is a character literal, then enclose it in single quotes.
TRIM (*Expression*)	If you specify *Expression* without a qualifier or *Trim_character*, then leading and trailing blank spaces are removed from *Expression*. *Expression* is the CHAR, VARCHAR2, NCHAR or NVARCHAR2 operand or column to be trimmed. If *Expression* is a character literal, then enclose it in single quotes.

Description

- If you specify the LEADING qualifier, TRIM removes any leading characters equal to *Trim_character* from *Expression*.
- If you specify the TRAILING qualifier, TRIM removes any trailing characters equal to *Trim_character* from *Expression*.
- If you specify the BOTH qualifier (or no qualifier), TRIM removes leading and trailing characters equal to *Trim_character* from *Expression*.
- If you specify only *Expression*, then TRIM removes leading and trailing blank spaces.
- 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. 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, and NVARCHAR2 types:

  - If all the characters in *Expression* are removed by the TRIM function, then 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, then 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

For the permitted format models to use in *fmt*, see "Format model for ROUND and TRUNC date functions" on page 3-23.

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 ttAXactAdmin 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.

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 functions

TimesTen supports these `USER` functions:

- `CURRENT_USER`
- `USER`
- `SESSION_USER`
- `SYSTEM_USER`

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

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;
```

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;
```

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;
```

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;
```

4

Search Conditions

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

Search condition general syntax

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

SQL syntax

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

Parameters

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" on page 4-3 for a description of how these operators work when predicates evaluate to NULL.
BetweenPredicate	Determines whether an expression is within a certain range of values. For example: A BETWEEN B AND C is equivalent to A >= B AND A<= C.
ComparisonPredicate	Compares two expressions or list of two expressions using one of the operators <, <=, >, >=, =, <>.
InPredicate	Determines whether an expression or list of expressions matches an element within a specified set.
ExistsPredicate	Determines whether a subquery returns any row.
LikePredicate	Determines whether an expression contains a particular character string pattern.
NullPredicate	Determines whether a value is NULL.
InfinitePredicate	Determines whether an expression is infinite (positive or negative infinity).
NaNPredicate	Determines whether an expression is the undefined result of an operation ("not a number").
QuantifiedPredicate	Determines whether an expression or list of expressions bears a particular relationship to a specified set.
(*SearchCondition*)	One of the above predicates, enclosed in parentheses.

Description

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

  Figure 4–1 shows the values that result from logical operations. A question mark (`?`) represents the `NULL` value.

Figure 4–1 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 Chapter 3, "Expressions" for information on value extensions during comparison operations.
- See "Numeric data types" on page 1-15 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	The syntax of expressions is defined under "Expression specification" on page 3-3. Both numeric and non-numeric expressions are allowed for ALL predicates, but both expression types must be compatible with each other.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.
Subquery	The syntax of subqueries is defined under "Subqueries" on page 3-6

Description

- The ALL predicate, which returns zero or more rows, uses a *comparison operator* modified with the keyword ALL. See "Numeric data types" on page 1-15 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 | (RowValueConstuctorList) |
```

The syntax for *`RowValueConstructorList`*:

```
RowValueConstructorElement[{, RowValueConstructorElement} ... ]
```

The syntax for *`RowValueConstructorElement`*:

```
Expression | NULL
```

The syntax for *`CompOp`*:

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

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

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

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

```
Constant | ? | :DynamicParameter
```

The syntax for an empty *`ValueList`*:

```
( )
```

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

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

Parameters

Component	Description
`Expression`	Specifies a value to be obtained. The values in *`ValueList`* must be compatible with the expression. For information on the syntax of expressions, see "Expression specification" on page 3-3.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.

Component	Description
<=	Is less than or equal to.
ALL	The predicate is TRUE if all the values in the *ValueList* relate to the expression or list of expressions as indicated by the comparison operator.
ValueList	A list of values that are compared against the expression's or list of expression's value. The *ValueList* cannot contain a column reference or a subquery. The *ValueList* can be nested if the left operand of the *ValueList* is a list. Elements of the *ValueList*: ■ Constant—Indicates a specific value. See "Constants" on page 3-11. ■ ?,:*DynamicParameter*—Placeholder for a dynamic parameter in a prepared SQL statement. The value of the dynamic parameter is supplied when the statement is executed. ■ Empty list, which are sometimes generated by SQL generation tools.

Description

- If X is the value of *Expression*, and (a,b, ..., z) represents the elements in *ValueList*, and OP is a comparison operator, then the following is true:
  - X OP ALL (a,b,...,z) is equivalent to X OP a AND X OP b AND...AND X OP z.
- If X is the value of *Expression* and (a,b,..., z) are the elements in a *ValueList*, then the following is true:
  - X NOT IN (a,b,...,z) is equivalent to NOT (X IN (a,b,...,z)).
- All character data types are compared in accordance with the current value of the NLS_SORT session parameter.
- NULL cannot be specified in *ValueList*.
- See "Numeric data types" on page 1-15 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 ();
```

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 | (RowValueConstuctorList) | Subquery
```

The syntax for *RowValueConstructorList*:

```
RowValueConstructorElement[{, RowValueConstructorElement} ... ]
```

The syntax for *RowValueConstructorElement*:

```
Expression | NULL
```

The syntax for *CompOp*:

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

Parameters

Component	Description
Expression	The syntax of expressions is defined under "Expression specification" on page 3-3. Both numeric and non-numeric expressions are allowed for ANY predicates, but both expression types must be compatible with each other.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.
<=	Is less than or equal to.
Subquery	The syntax of subqueries is defined under "Subqueries" on page 3-6.

Description

The ANY predicate, which returns zero or more rows, uses a *comparison operator* modified with the keyword ANY. See "Numeric data types" on page 1-15 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 `SELECT`s 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 | (RowValueConstuctorList) |
```

The syntax for *RowValueConstructorList*:

```
RowValueConstructorElement[{, RowValueConstructorElement} ... ]
```

The syntax for *RowValueConstructorElement*:

```
Expression | NULL
```

The syntax for *CompOp*:

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

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

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

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

```
Constant | ? | :DynamicParameter
```

The syntax for an empty *ValueList*:

```
( )
```

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

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

Parameters

Component	Description
Expression	Specifies a value to be obtained. The values in *ValueList* must be compatible with the expression. For information on the syntax of expressions, see "Expression specification" on page 3-3.
=	Is equal to.
<>	Is not equal to.
>	Is greater than.
>=	Is greater than or equal to.
<	Is less than.

Component	Description
<=	Is less than or equal to.
{ANY | SOME}	The predicate is TRUE if one or more of the values in the -*ValueList* relate to the expression or list of expressions as indicated by the comparison operator. SOME is a synonym for ANY.
ValueList	A list of values that are compared against the expression's or list of expression's value. The *ValueList* cannot contain a column reference or a subquery. The *ValueList* can be nested if the left operand of the *ValueList* is a list. Elements of the *ValueList*: ■ *Constant*—Indicates a specific value. See "Constants" on page 3-11. ■ ?,:*DynamicParameter*—Placeholder for a dynamic parameter in a prepared SQL statement. The value of the dynamic parameter is supplied when the statement is executed. ■ Empty list, which are sometimes generated by SQL generation tools.

Description

- If X is the value of *Expression*, and (a,b, ..., z) represents the elements in *ValueList*, and OP is a comparison operator, then the following is true:
  - X OP ANY (a,b,...,z) is equivalent to X OP a OR X OP b OR...OR X OP z.
- If X is the value of *Expression* and (a,b,..., z) are the elements in a *ValueList*, then the following is true:
  - X IN (a,b,...,z) is equivalent to X = a OR X = b OR...OR X = z.
- All character data types are compared in accordance with the current value of the NLS_SORT session parameter.
- NULL cannot be specified in *ValueList*.
- See "Numeric data types" on page 1-15 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);
```

To query an empty select list for `IN` condition:

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

Illustrates the use of a list of expressions with in:

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

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*	The syntax for expressions is defined in "Expression specification" on page 3-3. Both numeric and non-numeric expressions are allowed in BETWEEN predicates, but all expressions must be compatible with each other.

Description

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

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

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

  Comparisons are conducted as described in "Comparison predicate" on page 4-14.
- 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 | (RowValueConstuctorList) | 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	The syntax for expressions is defined under "Expression specification" on page 3-3. Both numeric and non-numeric expressions are allowed in comparison predicates, but both expressions must be compatible with each other.
ScalarSubquery	A subquery that returns a single value. Scalar subqueries and their restrictions are defined under "Subqueries" on page 3-6.
=	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" on page 1-15 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 cannot be used directly as an operand of an operator or predicate. If used, it must be cast to the correct datatype. For example, `(1 = NULL)` is not supported; however, `(1 = CAST(NULL AS INT))` is supported. This is in compliance with the ANSI SQL-92 specification.

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 and no rows for NOT EXISTS

SQL syntax

```
[NOT] EXISTS (Subquery)
```

Parameters

The EXISTS predicate has the following parameter:

Parameter	Description
Subquery	The syntax of subqueries is defined under "Subqueries" on page 3-6

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.
- For more information on `Inf` and `NaN`, see "INF and NAN" on page 1-30.

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.
- For more information on `Inf` and `NaN`, see "INF and NAN" on page 1-30.

IS NULL predicate

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

SQL syntax

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

Parameters

Parameter	Description
ColumnName	The name of a column from which a value is to be taken. Column names are discussed in Chapter 2, "Names and Parameters".
Constant	A specific value. See "Constants" on page 3-11.
(*Expression*)	Expression to test.

Examples

Vendors with no personal contact names are identified.

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

LIKE predicate

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

SQL syntax

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

Parameters

Parameter	Description
Expression	The syntax of expressions is presented in Chapter 3, "Expressions".
PatternString	Describes what you are searching for in the expression. The pattern may consist of characters only (including digits and special characters). For example, NAME LIKE 'Annie' evaluates to TRUE only for a name of Annie with no spaces. Upper case and lower case are significant. You can also use the predicate to test for a partial match by using the following symbols in the pattern: _ Represents any single character. For example: BOB and TOM both satisfy the predicate NAME LIKE '_O_'. % Represents any string of zero or more characters. For example, MARIE and RENATE both satisfy the predicate NAME LIKE '%A%'. You can use the _ and % symbols multiple times and in any combination in a pattern. You cannot use these symbols literally within a pattern unless you use the ESCAPE clause and precede the symbols with the escape character, described by the *EscapeChar* parameter.
EscapeChar	Describes an optional escape character which can be used to interpret the symbols _ and % literally in the pattern. The escape character must be a single character. When it appears in the pattern, it must be followed by the escape character itself, the _ symbol or the % symbol. Each such pair represents a single literal occurrence of the second character in the pattern. The escape character is always case sensitive. The escape character cannot be _ or %.
? *DynamicParameter*	Indicates a dynamic parameter in a prepared SQL statement. The parameter value is supplied when the statement is executed.

Description

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

Examples

Find employees 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 employees whose last name begins with 'SM'. This query returns no results because there are no employees 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;
0 rows found.
```

Use a dynamic parameter denoted by ? to find employees 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 employees 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 employees 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 results in an error because the pattern references a column.

```
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;
 1001: Syntax error in SQL statement before or at: "E1", character position: 169
...nager_id = e2.employee_id    AND 'Smith' like e1.last_name
                                                 ^^
The command failed.
```

NCHAR and NVARCHAR2

The LIKE predicate can be used for pattern matching of NCHAR and NVARCHAR2 strings. 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'!';
```

5

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.

DDL statements modify the database schema. `CREATE TABLE` and `DROP TABLE` are examples of DDL statements.

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.

ALTER ACTIVE STANDBY PAIR

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 "Changing the configuration of an active standby pair with cache groups" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*.

Required privilege

ADMIN

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.

Parameter	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. For information on *StoreAttribute* clauses, see "ALTER REPLICATION" on page 5-14.
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 host IP addresses" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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 replication. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE removes the tables, sequences or cache groups from replication. Use one EXCLUDE clause for each object type (table, sequence or cache group).
ADD ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Adds *NetworkOperation* to replication scheme. Allows 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.

Parameter	Description
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.
- You may only alter the active standby pair replication scheme on the active database. See "Changing the configuration of an active standby pair" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for more information.
- Use ADD SUBSCRIBER *FullStoreName* to add a subscriber to the replication scheme.
- Use DROP SUBSCRIBER *FullStoreName* to drop a subscriber from the replication scheme.
- Use ALTER STORE *FullStoreName* SET *StoreAttribute* to change the attributes for the specified database. Only the PORT and TIMEOUT attributes can be set for subscribers.
- 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 clause for each object type (table, sequence or cache group). Use one EXCLUDE clause for each object type (table, sequence or cache group).

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 allows changes to the state, interval and mode of AUTOREFRESH.

Updates on Oracle 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.

For a description of cache group types, see "User managed and system managed cache groups" on page 5-53.

Required privilege

No privilege is required for the cache group owner.

ALTER ANY CACHE GROUP for another user's cache group.

SQL syntax

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

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

This statement changes the AUTOREFRESH interval on the cache group:

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

This statement alters the AUTOREFRESH state:

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

Parameters

Parameter	Description
[*Owner*.]*GroupName*	Name assigned to the new cache group.
AUTOREFRESH	Indicates that changes to Oracle tables should be automatically propagated to TimesTen. For details, see "AUTOREFRESH in cache groups" on page 5-61.
MODE	Determines which rows in the cache are updated during an autorefresh. If the INCREMENTAL clause is specified, TimesTen refreshes only rows that have been changed on Oracle since the last propagation. If the FULL clause is specified or if there is neither FULL nor INCREMENTAL clause specified, TimesTen updates all rows in the cache with each autorefresh. The default mode is INCREMENTAL.

Parameter	Description
INTERVAL *IntervalValue*	Indicates the interval at which autorefresh should occur in units of minutes, seconds or milliseconds. An integer value that specifies how often AUTOREFRESH should be scheduled, in minutes, seconds or milliseconds. The default value is 10 minutes. If the specified interval is not long enough for an AUTOREFRESH to complete, a runtime warning is generated and the next AUTOREFRESH waits until the current one finishes. An informational message is generated in the support log if the wait queue reaches 10.
STATE	Specifies whether AUTOREFRESH should be changed to on, off or paused. By default, the AUTOREFRESH STATE is ON.
ON	AUTOREFRESH is scheduled to occur at the specified interval.
OFF	A scheduled AUTOREFRESH is cancelled, and TimesTen does not try to maintain the information necessary for an INCREMENTAL refresh. Therefore if AUTOREFRESH is turned on again at a later time, the first refresh is FULL.
PAUSED	A scheduled AUTOREFRESH is cancelled, but TimesTen tries to maintain the information necessary for an INCREMENTAL refresh. Therefore if AUTOREFRESH is turned on again at a later time, a full refresh may not be necessary.

Description

- A refresh does not occur immediately after issuing 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 non-zero, 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.

See also

CREATE CACHE GROUP

ALTER FUNCTION

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.

SQL syntax

```
ALTER FUNCTION [Owner.]FunctionName COMPILE
      [compiler_parameters_clause [...]]
      [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.
compiler_parameters_clause	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, PLSCOPE_SETTINGS 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

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.

SQL syntax

```
ALTER PACKAGE [Owner.]PackageName COMPILE
      [PACKAGE|SPECIFICATION|BODY]
      [compiler_parameters_clause [...]]
      [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.
compiler_parameters_clause	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, PLSCOPE_SETTINGS 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

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.

SQL syntax

```
ALTER PROCEDURE [Owner.]ProcedureName COMPILE
      [compiler_parameters_clause [...]]
      [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.
compiler_parameters_clause	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, PLSCOPE_SETTINGS 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` and `PLSCOPE_SETTINGS` for procedure *`query_emp`*. Alter *`query_emp`* by changing `PLSQL_OPTIMIZE_LEVEL` to 3. Verify results.

```
Command> SELECT PLSQL_OPTIMIZE_LEVEL, PLSCOPE_SETTINGS
       > FROM user_plsql_object_settings WHERE  name = 'QUERY_EMP';
< 2, IDENTIFIERS:NONE >
1 row found.

Command> ALTER PROCEDURE query_emp COMPILE PLSQL_OPTIMIZE_LEVEL = 3;

Procedure altered.

Command> SELECT PLSQL_OPTIMIZE_LEVEL, PLSCOPE_SETTINGS
       > FROM user_plsql_object_settings WHERE  name = 'QUERY_EMP';
< 3, IDENTIFIERS:NONE >
1 row found.
```

See also

`CREATE PROCEDURE`

ALTER REPLICATION

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

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 Replication" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for more information.

Required privilege

ADMIN

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[,...]] |
                CACHE GROUP [[Owner.]CacheGroupName[,...]]|
                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 |
                CACHE GROUP [Owner.]CacheGroupName |
                SEQUENCE [Owner.]SequenceName}[,...]
DROP ELEMENT { ElementName | * IN FullStoreName }
```

`CheckConflicts` can only be set when replicating `TABLE` elements. The syntax is described in "CHECK CONFLICTS" on page 5-94.

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 replication scheme.
ADD ELEMENT *ElementName*	Adds a new element to the existing replication scheme. *ElementName* is an identifier of up to 30 characters. With DATASTORE elements, the *ElementName* must be unique with respect to other DATASTORE element names within the first 20 chars. If the element is a DATASTORE, all tables and cache groups are included in the 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* [,...]]| CACHE GROUP [[*Owner*.]*CacheGroupName* [,...]]| SEQUENCE [[*Owner*.]*SequenceName*[,...]]} [,...]	Adds a new DATASTORE element to the existing replication scheme. *ElementName* is an identifier of up to 30 characters. With DATASTORE elements, the *ElementName* must be unique with respect to other DATASTORE element names within the first 20 chars. INCLUDE includes in the database only the tables and cache groups listed. Use one INCLUDE clause for each object type (table, cache group or sequence). EXCLUDE includes in the database all tables and cache groups *except* the tables, cache groups and sequences listed. Use one EXCLUDE clause for each object type (table, cache group or sequence). 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.

Parameter	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`* |
`CACHE GROUP [`*`Owner.`*`]`*`CacheGroupName`* |
`SEQUENCE [`*`Owner.`*`]`*`SequenceName`*`}`
`[,...]`
	`ElementName` is the name of the element to be altered.
	`INCLUDE` adds to the database the tables and cache groups listed. Use one `INCLUDE` clause for each object type (table or cache group).
	`EXCLUDE` removes from the database the tables and cache groups listed. Use one `EXCLUDE` clause for each object type (table, cache group 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.

`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" on page 5-94.
`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 TimesTen to TimesTen 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. Use this clause for table-level replication.

Parameter	Description
`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. Use this clause 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. See "Managing return service timeout errors and replication state changes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.
`DURABLE COMMIT {ON | OFF}`	Set to override the `DurableCommits` setting on a database and enable durable commit when return service blocking has been disabled by `DISABLE RETURN`.
`DROP ELEMENT * IN` *`FullStoreName`*	Deletes the replication description of all elements for which *`FullStoreName`* is the `MASTER`. *`FullStoreName`* is the 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 your replication scheme has only one subscriber. *`FullStoreName`* is the database file name specified in the `DataStore` attribute of the DSN 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 log failure threshold" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for more information.

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. 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 host IP addresses" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: "MyHost-500".
LOCAL COMMIT ACTION {NO ACTION | COMMIT}	Specifies the default action to be taken for a RETURN TWOSAFE transaction in the event of a timeout. NO ACTION: On timeout, the commit function returns to the application, leaving the transaction in the same state it was in when it entered the commit call, with the exception that the application is not able to update any replicated tables. The application can only reissue the commit. The transaction may not be rolled back. This is the default. COMMIT: On timeout, the commit function 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 TimesTen to TimesTen 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 TimesTen to TimesTen Replication Guide*.
RETURN SERVICES {ON | OFF} WHEN [REPLICATION] STOPPED	Set the return service failure policy so that return service blocking is either enabled or disabled when the replication agent is in the "stop" or "pause" state. OFF is the default when using the RETURN RECEIPT service. ON is the default when using the RETURN TWOSAFE service. See "Managing return service timeout errors and replication state changes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master 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 TimesTen to TimesTen Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgement. The default value is 10 seconds. A value of '0' means there is no 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 EXACT.
TIMEOUT *Seconds*	The amount of time a database waits for a response from another database before resending the message. Default: 120 seconds.
ADD ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Adds *NetworkOperation* to replication scheme. Allows 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 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:

- Manually drop the replicated elements by executing an `ALTER REPLICATION DROP ELEMENT` statement for each replicated table.
- Use `ALTER REPLICATION ADD ELEMENT` to add each table back to the replication scheme, with the newly designated `MASTER` / `SUBSCRIBER` roles.
- `ALTER REPLICATION ALTER ELEMENT SET MASTER` does not automatically retain the old master as a subscriber in the scheme. If this is desired, execute an `ALTER REPLICATION ALTER ELEMENT ADD SUBSCRIBER` statement.

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

- Stop the replication agent before you use the *`NetworkOperation`* clause.

Examples

This example sets up replication 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 `my.cg1` cache group to `ds1` database in `my.rep1` replication scheme.

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

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

```
ALTER REPLICATION my.rep1
  ADD ELEMENT ds1 DATASTORE
     MASTER rep2
     SUBSCRIBER rep1, rep3
     INCLUDE TABLE my.tab2
     INCLUDE CACHE GROUP my.cg2, my.cg3;
```

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

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

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 *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*.

ALTER SESSION

The `ALTER SESSION` statement changes session parameters dynamically.

Required privilege

None

SQL syntax

```
ALTER SESSION SET
  {NLS_SORT = {BINARY| SortName} |
   NLS_LENGTH_SEMANTICS = {BYTE|CHAR} |
   NLS_NCHAR_CONV_EXCP = {TRUE|FALSE} |
   ISOLATION_LEVEL = {SERIALIZABLE | READ COMMITTED} |
   PLSQL_TIMEOUT = n |
   PLSQL_OPTIMIZE_LEVEL = {0|1|2|3}|
   PLSCOPE_SETTINGS = {'IDENTIFIERS:ALL'|'IDENTIFIERS:NONE'} |
   PLSQL_CONN_MEM_LIMIT = n
   REPLICATION_TRACK = <TrackNumber>
   } ...
```

Parameters

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 if you want 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 sorts" 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 "Length semantics and 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`.
`ISOLATION_LEVEL = {SERIALIZABLE|READ COMMITTED}`	Sets isolation level. Change takes effect starting with *next* transaction. For a descriptions of the isolation levels, see *Oracle TimesTen In-Memory Database Operations Guide*.

Parameter	Description
`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. If PL/SQL is not enabled in your database and you specify this attribute, TimesTen throws an error.
`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. If PL/SQL is not enabled in your database and you specify this attribute, TimesTen returns an error. For more information, see "PLSQL_OPTIMIZE_LEVEL" in *Oracle TimesTen In-Memory Database Reference*.
`PLSCOPE_SETTINGS = '{IDENTIFIERS:ALL |IDENTIFIERS:NONE}'`	Controls whether or not the PL/SQL compiler generates cross-reference information. Specify `IDENTIFIERS:ALL` to generate cross-reference information. The default is `IDENTIFIERS:NONE`. If PL/SQL is not enabled in your database and you specify this attribute, TimesTen returns an error. For more information, see "PLSCOPE_SETTINGS" 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. *n* is an integer expressed in megabytes. The default is 100. If PL/SQL is not enabled in your database and you specify this attribute, TimesTen returns an error. For more information, see "PLSQL_CONN_MEM_LIMIT" in *Oracle TimesTen In-Memory Database Reference*.
`REPLICATION_TRACK =` *<TrackNumber>*	When parallel replication is configured, specify the track to which the transactions belong for the current connection. All transactions on replicated tables are associated with a track. The track number setting is constant for the lifetime of the connection, unless specifically reset. The default track number is 0. 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 "Increasing replication throughput for other replication schemes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*.

Description

- The `ALTER SESSION` statement affects commands that are subsequently executed by the session. The new session parameters take effect immediately.
- The `NLS_SORT` setting affects materialized views and cache group maintenance. Use the `NLSSORT()` SQL function rather than relying on the `NLS_SORT` setting.

- Character length and byte length semantics are supported to resolve potential ambiguity regarding column length and storage size. Multibyte encoding character sets are supported (For example, UTF-8 or AL32UTF8). Multibyte encodings require varying amounts of storage per character depending on the character. For example, a UTF-8 character may require from 1 to 4 bytes.

  If, for example, a column is defined as CHAR (10), all 10 characters fit in this column regardless of character set encoding. However, for UTF-8 character set encoding, up to 40 bytes are required. TimesTen supports character length and byte length semantics to avoid such ambiguity.
- Operations involving character comparisons support linguistic sensitive collating sequences. Case-insensitive sorts may affect DISTINCT value interpretation. Supported collating sequence sensitive operations:
  - MIN,MAX
  - BETWEEN
  - =,!=,>,>=,<,<=
  - DISTINCT
  - CASE
  - GROUP BY
  - HAVING
  - ORDER BY
  - IN
  - LIKE
- Primary key indexes are based on the BINARY collating sequence. You cannot use nonbinary NLS_SORT with equality searches on the primary key index.
- Implicit and explicit conversions between CHAR and NCHAR are supported.
- Conversions between CHAR and NCHAR are not allowed when using the TIMESTEN8 character set.
- You can use the SQL string functions with the supported character sets. For example, UPPER and LOWER functions support non-ASCII CHAR and VARCHAR2 characters as well as NCHAR and NVARCHAR2 characters.
- TIMESTEN8 character set restrictions:
  - Character set conversions are not allowed.
  - BINARY is the only acceptable collating sequence.
  - CHAR semantics are ignored. Characters are single-byte.
  - UPPER and LOWER functions support ASCII characters only. Results for non-ASCII characters are undefined. TimesTen does not return an error.
- NLS_SORT settings other than BINARY could have a performance impact on character operations.
- Choice of character set could have an impact on memory consumption for CHAR and VARCHAR2 column data.
- The character sets of all databases involved in a replication scheme must match.

Examples

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
```

Call `ttConfiguration` to display the current `PLSCOPE_SETTINGS` value. Use the `ALTER SESSION` statement to change the `PLSCOPE_SETTINGS` value to `IDENTIFIERS:ALL`. Create a dummy procedure `p`. Query the system view `SYS.USER_PLSQL_OBJECT_SETTINGS` to confirm that the new setting is applied to procedure `p`.

```
Command> CALL TTCONFIGURATION ();
< CkptFrequency, 600 >
< CkptLogVolume, 0 >
< CkptRate, 0 >
...
< PLSCOPE_SETTINGS, IDENTIFIERS:NONE >
...
47 rows found.

Command> ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';

Session altered.

Command> CREATE OR REPLACE PROCEDURE p IS
       > BEGIN
       >  NULL;
```

```
       > END;
       > /

Procedure created.

Command> SELECT PLSCOPE_SETTINGS FROM SYS.USER_PLSQL_OBJECT_SETTINGS WHERE
       > NAME = 'p';
< IDENTIFIERS:ALL >
1 row found.
```

The following example uses the `ALTER SESSION` statement to change the `NLS_SORT` setting from `BINARY` to `BINARY_CI` to `BINARY_AI`. The database and connection character sets are `WE8ISO8859P1`.

```
Command> connect "dsn=cs;ConnectionCharacterSet=WE8ISO8859P1";
Connection successful: DSN=cs;UID=user;DataStore=/datastore/user/cs;
DatabaseCharacterSet=WE8ISO8859P1;
ConnectionCharacterSet=WE8ISO8859P1;PermSize=32;TypeMode=0;
(Default setting AutoCommit=1)
Command>#Create the Table
Command> CREATE TABLE collatingdemo (letter VARCHAR2 (10));
Command>#Insert values
Command> INSERT INTO collatingdemo VALUES ('a');
1 row inserted.
Command> INSERT INTO collatingdemo VALUES ('A');
1 row inserted.
Command> INSERT INTO collatingdemo VALUES ('Y');
1 row inserted.
Command> INSERT INTO collatingdemo VALUES ('ä');
1 row inserted.
Command>#SELECT
Command> SELECT * FROM collatingdemo;
< a >
< A >
< Y >
< ä >
4 rows found.
Command>#SELECT with ORDER BY
Command> SELECT * FROM collatingdemo ORDER BY letter;
< A >
< Y >
< a >
< ä >
4 rows found.
Command>#set NLS_SORT to BINARY_CI and SELECT
Command> ALTER SESSION SET NLS_SORT = BINARY_CI;
Command> SELECT * FROM collatingdemo ORDER BY letter;
< a >
< A >
< Y >
< Ä >
< 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 parallel replication and uses the `ALTER SESSION` statement to change the replication track number to 5 for the current connection. The connection attributes `ReplicationParallelism` is set to a value higher than 5 and `ReplicationApplyOrdering` is set to 1.

```
Command> ALTER SESSION SET REPLICATION_TRACK = 5;

Session altered.
```

ALTER TABLE

The `ALTER TABLE` statement changes an existing table definition.

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.

SQL syntax

To add columns:

```
ALTER TABLE [Owner.]TableName
 ADD [COLUMN] ColumnName ColumnDataType
  [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL]
```

or

```
ALTER TABLE [Owner.]TableName
 ADD (ColumnName ColumnDataType
  [DEFAULT DefaultVal] [[NOT] INLINE] [UNIQUE] [NULL] [, ... ] )
```

To remove columns:

```
ALTER TABLE [Owner.]TableName
  DROP [COLUMN] ColumnName
```

or

```
ALTER TABLE [Owner.]TableName
  DROP (ColumnName [, ... ] )
```

To add a 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
```

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}
```

Change the primary key to use a range index with the `USE TREE INDEX` clause:

```
ALTER TABLE [Owner.]TableName
USE TREE INDEX
```

To change the default value of a column:

```
ALTER TABLE [Owner.]TableName
MODIFY (ColumnName DEFAULT DefaultVal)
```

To add or drop a unique constraint on a column:

```
ALTER TABLE Owner.]TableName
{ADD | DROP} UNIQUE (ColumnName)
```

To remove the default value of a column that is nullable, by changing it to `NULL`:

```
ALTER TABLE [Owner.]TableName
MODIFY (ColumnName DEFAULT NULL)
```

To add LRU aging:

```
ALTER TABLE [Owner.]TableName
ADD AGING LRU [ON | OFF]
```

To add time-based aging:

```
ALTER TABLE [Owner.]TableName
ADD AGING USE ColumnName LIFETIME num1
    {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

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.
ColumnName	Name of the column for which the unique constraint or default value is to be changed. A new column cannot have the same name as an existing column or another new column.
ColumnDataType	Type of the column to be added. Some types require additional parameters. See Chapter 1, "Data Types" for the data types that can be specified.
INLINE|NOT INLINE	By default, variable-length columns whose declared column length is > 128 bytes are stored out of line. Variable-length columns whose declared column length is <= 128 bytes are stored inline. The default behavior can be overridden during table creation through the use of the INLINE and NOT INLINE keywords.
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. Specify either *RowPages* (as a positive constant) or CURRENT to calculate the page count value. If you specify CURRENT, the current number of rows in the table is used to calculate the page count value. See "Column Definition" on page 5-112 for a description of hash indexes and pages.
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 or dropped. See "FOREIGN KEY" on page 5-109.
REFERENCES	Specifies that the foreign key references another table.
RefTableName	The name of the table that the foreign key references.

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.
`USE HASH INDEX PAGES = {`*`RowPages`*` | CURRENT}`	Specifies that a hash index is to be used for the primary key. If the primary key already uses a hash index, then this clause is equivalent to the `SET PAGES` clause.
`USE TREE INDEX`	Specifies that a range index is to be used for the primary key. If the primary key already uses a range index, TimesTen ignores this clause.
`SET PAGES`	Resizes the hash index based on the expected number of row pages in the table. Each row page can contain up to 256 rows of data. This number determines the number of hash buckets created for the hash index. The minimum is 1. If your estimate is too small, performance may be degraded. You can specify a constant (*`RowPages`*) or the current number of row pages. See "Column Definition" on page 5-112 for a description of hash indexes and pages.
`RowPages`	The number of row pages expected.
`CURRENT`	Use the number of row pages currently in use.
`ADD AGING LRU [ON | OFF]`	Adds 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` procedure. LRU attributes are not defined at the SQL level. For more information about LRU aging, see "Implementing aging in your tables" in *Oracle TimesTen In-Memory Database Operations Guide*.

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 aging 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. If you want 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

- The ALTER TABLE statement cannot be used to alter a temporary table.
- The ALTER TABLE statement cannot be used to alter a replicated table that is part of a TWOSAFE BY REQUEST transaction. If DDLCommitBehavior=1, this operation results in error 8051. If DDLCommitBehavior=0, the operation succeeds because a commit is performed before the ALTER TABLE operation, resulting in the ALTER TABLE operation being in a new transaction which is not part of the TWOSAFE BY REQUEST transaction.
- The ALTER TABLE ADD [COLUMN] *ColumnName* statement adds one or more new columns to an existing table. The new columns are added to the end of all existing rows of the table in one new partition.
- Columns referenced by materialized views cannot be dropped.
- Only one partition is added to the table per statement regardless of the number of columns added.
- The new columns cannot be declared NOT NULL.
- NULL is the initial value for all added columns, unless a default value is specified for the new column.

- The total number of columns in the table cannot exceed 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 -noRepUpgrade` command, and restoring the table using the `ttRestore -r -noRepUpgrade` command. Dropping the added column does not recover the lost performance or decrease the number of partitions.
- The `ALTER TABLE DROP` statement removes one or more columns from an existing table. The dropped columns are removed from all current rows of the table. Subsequent SQL statements must not attempt to make any use of the dropped columns. 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 TREE INDEX` if your application performs range queries over a table's primary key.
- Use `ALTER TABLE...USE HASH INDEX` if your application performs exact match lookups on a table's primary key.

- An error is generated if a table has no primary key and either the `USE HASH INDEX` clause or the `USE TREE INDEX` clause is specified.
- 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.
  - You must stop the replication agent before adding or dropping a foreign key on a replicated table.
- The `ALTER TABLE ADD/DROP CONSTRAINT` statement has the following restrictions:
  - When a foreign key is dropped, TimesTen also drops the index associated with the foreign key. Attempting to drop an index associated with a foreign key using the regular `DROP INDEX` statement results in an error.
  - Foreign keys cannot be added or dropped on tables in a cache group.
  - Foreign keys cannot be added or dropped on tables that participate in TimesTen replication. If the operation is attempted on a table that is either being replicated or is a replicated table, TimesTen returns an error.
  - Foreign keys cannot be added or dropped on views or temporary tables.

- 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.

Examples

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 T-tree index on columns:
    COL1
  1 index found.

1 table found.
```

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 T-tree index on columns:
```

```
    COL1
 1 index found.
1 table found.
```

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 TREE INDEX` clause:

```
Command> ALTER TABLE pkey USE TREE INDEX;
Command> INDEXES pkey;
Indexes on table SAMPLEUSER.PKEY:
  PKEY: unique T-Tree index on columns:
    COL1
  1 index found.
1 table found.
```

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 illegalindex (Ccl1 CHAR (20));
Command> ALTER TABLE illegalindex USE TREE INDEX;
 2810: The table has no primary key so cannot change its index type
The command failed.
Command> ALTER TABLE illegalindex USE HASH INDEX PAGES = CURRENT;
 2810: The table has no primary key so cannot change its index type
The command failed.
```

These examples show how time resolution works with aging. In this example, lifetime is 3 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 if you want 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.
```

See also

CREATE TABLE
DROP TABLE
"Implementing aging in your tables" in *Oracle TimesTen In-Memory Database Operations Guide*

ALTER USER

The `ALTER USER` statement allows a user to change the user's own password. A user with the `ADMIN` privilege can change another user's password.

This statement also allows a user to change another 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 to change another user's password.

`ADMIN` privilege is required to change users from internal to external and from external to internal.

SQL syntax

```
ALTER USER user IDENTIFIED BY {password | "password"}
ALTER USER user IDENTIFIED EXTERNALLY
```

Parameters

Parameter	Description
user	Name of the user whose password is being changed.
IDENTIFIED BY	Identification clause.
password | "*password*"	Specifies the password that identifies the internal user to the TimesTen database.
EXTERNALLY	Identifies the operating system *user* to the TimesTen database. To perform database operations as an external user, the process needs a TimesTen external user name that matches the user name authenticated by the operating system or network. A password is not required by TimesTen because the user has been authenticated by the operating system at login time.

Description

- Database users can be internal or external.
  - Internal users are defined for a TimesTen database.
  - External users are defined by an external authority, such as the operating system. External users cannot be assigned a TimesTen password.
- If you are an internal user connected as *user*, execute this statement to change your TimesTen password.
- Passwords are case-sensitive.
- You cannot alter a user across a client/server connection. You must use a direct connection when altering a user.

Examples

To change the password for internal user `terry` to `"12345"` from its current setting, use:

```
ALTER USER terry IDENTIFIED BY "12345";
User altered.
```

To change user `terry` to an external user:

```
ALTER USER terry IDENTIFIED EXTERNALLY;
User altered.
```

To change user `terry` back to an internal user, provide a password:

```
ALTER USER terry IDENTIFIED BY "secret";
User altered.
```

See also

CREATE USER
DROP USER
GRANT
REVOKE

CALL

Use the `CALL` statement to invoke 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 invoking 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.

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:

```
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.

For syntax on how to call a PL/SQL procedure or function that is the result of an expression, see the "CALL" statement description in the *Oracle Database SQL Language Reference*.

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.

Parameter	Description
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 "How to execute PL/SQL 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. A transaction is a sequence of SQL statements treated as a single unit.

Required privilege

None

SQL syntax

```
COMMIT [WORK]
```

Parameters

The COMMIT statement allows 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 transaction will also be committed.
- A commit closes all open cursors.

Examples

Insert 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 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

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 [,...]]} [,...]]
```

The syntax for *ReturnServiceAttribute* is

```
{ RETURN RECEIPT [BY REQUEST] |
  RETURN TWOSAFE [BY REQUEST] |
  NO RETURN }
```

Syntax for *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 ]
```

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 host IP addresses" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: "MyHost-500".
RETURN RECEIPT [BY REQUEST]	Enables the return receipt service, so that applications that commit a transaction to 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 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 TimesTen to TimesTen Replication Guide*.
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 TimesTen to TimesTen 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. See "Managing return service timeout errors and replication state changes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.

Parameter	Description
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 TimesTen to TimesTen Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgement. A value of 0 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.
{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 to replication. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE removes tables, sequences or cache groups from the replication scheme. Use one EXCLUDE clause for each object type (table, sequence or cache group).
DURABLE COMMIT {ON | OFF}	Set to override the DurableCommits setting on a database and enable durable commit when return service blocking has been disabled by DISABLE RETURN.
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 log failure threshold" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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 reissue the commit or rollback the call. 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, allows 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, 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.
`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.

Parameter	Description
TIMEOUT *Seconds*	Set the maximum number of seconds a database waits before re-sending a message to an unresponsive database. 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.

Description

- CREATE ACTIVE STANDBY PAIR is immediately followed by the names of the two master databases. The master databases are later designated as ACTIVE and STANDBY using the ttRepStateSet built-in procedure. See "Setting up an active standby pair with no cache groups" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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
- Using an active standby pair to replicate read-only cache groups and asynchronous writethrough (AWT) cache groups is supported.
- You cannot use an active standby pair to replicate synchronous writethrough (SWT) cache groups. If you are using an active standby pair to replicated a database with SWT cache groups, you must either drop or exclude the SWT cache groups.

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;
```

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 table.
- The rows are stored in a TimesTen table with the same name as the Oracle table.

If a database has more than one cache group, the cache groups must correspond to different Oracle (and TimesTen) tables.

Cache group instance refers to a row in the root table and all the child table rows related directly or indirectly to the root table rows.

User managed and system managed cache groups

A cache group can be either system managed or user managed.

A *system managed cache group* is fully managed by TimesTen and has fixed properties. System managed cache group types include:

- Read-only cache groups are updated in the Oracle database, and the updates are propagated from Oracle 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 Oracle.
- 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 Oracle.

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 must be able to parse all `WHERE` clauses.

Explicitly loaded cache groups and dynamic cache groups

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 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 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 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` built-in procedure 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 explicitly loaded and dynamic cache groups, see *Oracle In-Memory Database Cache User's Guide*. For more information about the dynamic load operation, see "Dynamically loading a cache group" in *Oracle In-Memory Database Cache User's Guide*.

Local and global cache groups

You can create either local or global cache groups.

In a local cache group, data in the cache tables are not shared across TimesTen databases even if the databases are members of the same cache grid. Therefore, the databases can have overlapping data or the same data. Any cache group type can be defined as a local cache group. A local cache group can be either dynamically or explicitly loaded.

In a global cache group, data in the cache tables are shared among TimesTen databases within a cache grid. Updates to the same data by different grid members are coordinated by the grid. Only an AWT cache group can be defined as a global cache group.

For more information on local and global cache groups, see "Defining Cache Groups" in the *Oracle In-Memory Database Cache User's Guide*. In addition, see "Example of data sharing among the grid members" in *Oracle In-Memory Database Cache User's 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).

SQL syntax

There are CREATE CACHE GROUP statements for each type of cache group:

- CREATE READONLY CACHE GROUP
- CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP
- CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP
- CREATE USERMANAGED CACHE GROUP

There is one CREATE CACHE GROUP statement to create a global cache group:

- CREATE WRITETHROUGH GLOBAL CACHE GROUP

CREATE READONLY CACHE GROUP

For read-only cache groups, the syntax is:

```
CREATE [DYNAMIC] READONLY CACHE GROUP [Owner.]GroupName
 [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]
 [WHERE ExternalSearchCondition]
 [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]
 ]
} [,...];
```

CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP

For asynchronous writethrough cache groups, the syntax is:

```
CREATE [DYNAMIC] [ASYNCHRONOUS] WRITETHROUGH CACHE GROUP   [Owner.]GroupName
  FROM
   {[Owner.]TableName (
     {ColumnDefinition[,...]}
    [,PRIMARY KEY(ColumnName[,...])]
     [FOREIGN KEY(ColumnName [,...])
          REFERENCES RefTableName (ColumnName [,...])]
                         [ ON DELETE CASCADE ]
 UNIQUE HASH ON (HashColumnName[,...]) PAGES=PrimaryPages]
 [AGING {LRU|
     USE ColumnName
          LIFETIME Num1 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
         [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
     }[ON|OFF]
 ]
} [,...];
```

CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP

For synchronous writethrough cache groups, the syntax is:

```
CREATE [DYNAMIC] SYNCHRONOUS WRITETHROUGH
CACHE GROUP [Owner.]GroupName
 FROM
   {[Owner.]TableName (
     {ColumnDefinition[,...]}
    [,PRIMARY KEY(ColumnName[,...])]
     [FOREIGN KEY(ColumnName [,...])
            REFERENCES RefTableName (ColumnName [,...])}]
                              [ ON DELETE CASCADE ]
 [UNIQUE HASH ON (HashColumnName[,...]) PAGES=PrimaryPages]
 [AGING {LRU|
     USE ColumnName
         LIFETIME Num1 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
         [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
     }[ON|OFF]
 ]
} [,...];
```

CREATE USERMANAGED CACHE GROUP

For user managed cache groups, the syntax is:

```
CREATE [DYNAMIC][USERMANAGED] CACHE GROUP [Owner.]GroupName
 [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]
 [WHERE ExternalSearchCondition]
 [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]
 ]
} [,...];
```

CREATE WRITETHROUGH GLOBAL CACHE GROUP

The following syntax demonstrates how to create a global cache group to cache data within a cache grid. Specify the `DYNAMIC` attribute to enable dynamic load from the Oracle database for the cache group.

```
CREATE [DYNAMIC] [ASYNCHRONOUS] WRITETHROUGH GLOBAL CACHE GROUP  [Owner.]GroupName
  FROM
   {[Owner.]TableName (
     {ColumnDefinition[,...]}
    [,PRIMARY KEY(ColumnName[,...])]
     [FOREIGN KEY(ColumnName [,...])
          REFERENCES RefTableName (ColumnName [,...])]
```

```
                        [ ON DELETE CASCADE ]
 UNIQUE HASH ON (HashColumnName[,...]) PAGES=PrimaryPages]
 [AGING {LRU|
     USE ColumnName
          LIFETIME Num1 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}
         [CYCLE Num2 {SECOND[S] | MINUTE[S] |HOUR[S] |DAY[S]}]
     }[ON|OFF]
 ]
} [,...];
```

Parameters

Following are the parameters for the cache group definition before the FROM keyword:

Parameter	Description
[*Owner.*]*GroupName*	Owner and name assigned to the new cache group.
[DYNAMIC]	If specified, a dynamic cache group is created.
AUTOREFRESH	The AUTOREFRESH parameter automatically propagates changes from the Oracle database to the cache group. For details, see "AUTOREFRESH in cache groups" on page 5-61.
MODE [INCREMENTAL | FULL]	Determines which rows in the cache are updated during an autorefresh. If the INCREMENTAL clause is specified, TimesTen refreshes only rows that have been changed on Oracle since the last propagation. If the FULL clause is specified, TimesTen updates all rows in the cache with each autorefresh. The default autorefresh mode is INCREMENTAL.
INTERVAL *IntervalValue*	Indicates the interval at which autorefresh should occur in units of minutes, seconds or milliseconds. *IntervalValue* is an integer value that specifies how often autorefresh should be scheduled, in MINUTES, SECONDS or MILLISECONDS. The default *IntervalValue* value is 5 minutes. If the specified interval is not long enough for an autorefresh to complete, a runtime warning is generated and the next autorefresh waits until the current one finishes. An informational message is generated in the 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 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 or not it is nullable. Each table must have at least one column. See "Column Definition" on page 5-112.

Parameter	Description
PRIMARY KEY (*ColumnName*[,...])	Specifies that the table has a primary key. Primary key constraints are required for a cache group. *ColumnName* is the name of the column that forms the primary key for the table to be created. Up to 16 columns can be specified for the primary key. Cannot be specified with UNIQUE in one specification.
FOREIGN KEY (*ColumnName*[,...])	Specifies that the table has a foreign key. *ColumnName* is the name of the column that forms the foreign key for the table to be created. See "FOREIGN KEY" on page 5-109.
REFERENCES *RefTableName* (*ColumnName*[,...])	Specifies the table which the foreign key is associated with. *RefTableName* is the name of the referenced table and *ColumnName* is the name of the column referenced in the table.
[ON DELETE CASCADE]	Enables the ON DELETE CASCADE referential action. If specified, when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign key values are also deleted.
READONLY	Specifies that changes cannot be made on the cached table.
PROPAGATE|NOT PROPAGATE	Specifies whether changes to the cached table are automatically propagate to the corresponding Oracle table at commit time.
UNIQUE HASH ON (*HashColumnName*)	Specifies that a hash index is created on this table. *HashColumnName* identifies the column that is to participate in the hash key of this table. The columns specified in the hash index must be identical to the columns in the primary key.
PAGES=*PrimaryPages*	Specifies the expected number of pages in the table. The *PrimaryPages* number determines the number of hash buckets created for the hash index. The minimum is 1. If your estimate is too small, performance is degraded. See "CREATE TABLE" on page 5-107 for more information.
WHERE *ExternalSearchCondition*	The WHERE clause evaluated by Oracle for the cache group table. This WHERE clause is added to every LOAD and REFRESH operation on the cache group. It may not directly reference other tables. It is parsed by both TimesTen and Oracle. See "Using a WHERE clause" in *Oracle In-Memory Database Cache User's Guide*.

Parameter	Description
`AGING LRU [ON | OFF]`	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 set `ON` by default. You can specify time-based aging instead. Aging is disabled by default on an explicitly loaded global cache group. LRU aging is not supported for cache groups with autorefresh. LRU attributes are defined by calling the `ttAgingLRUConfig` procedure. LRU attributes are not defined at the SQL level. For more information about LRU aging, see "Implementing aging on a cache group" in *Oracle In-Memory Database Cache User's Guide*.
`AGING USE` *`ColumnName`*`...[ON|OFF]`	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`). Aging is disabled by default on an explicitly loaded global cache group. For more information about time-based aging, see "Implementing aging on a cache group" in *Oracle In-Memory Database Cache User's 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). 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.

Description

- Two cache groups cannot have the same owner name and group name. If you do not specify the owner name, your login becomes the owner name for the new cache group.
- Dynamic parameters are not allowed in the WHERE clause.
- Oracle temporary tables cannot be cached.
- Each table must correspond to a table in the Oracle database.
- 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 table name. Uppercase table names on TimesTen will not match mixed case table names on Oracle. As a workaround, create a synonym for your table in Oracle 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 table, both in name and in data type. See "Mappings between Oracle and TimesTen data types" in *Oracle In-Memory Database Cache User's 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, UNLOAD, REFRESH or FLUSH statements. However, since TimesTen automatically generates queries that join multiple tables in the same cache group, a column needs to be fully qualified if there is more than one table in the cache group that contains columns with the same name.
- By default, a range index is created to enforce the primary key for a cache group table. Use the UNIQUE HASH clause to specify a hash index for the primary key.
  - If your application performs range queries over a cache group table's primary key, then choose a 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" on page 5-107 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 tables.

AUTOREFRESH in cache groups

The AUTOREFRESH parameter automatically propagates changes from the Oracle database to TimesTen cache groups. For explicitly loaded 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 tables are not propagated to dynamic cache groups. They are dynamically loaded into IMDB 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 command. Once a cache group has been specified as either AUTOREFRESH or PROPAGATE, you cannot change these attributes.

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 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 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 *Oracle In-Memory Database Cache User's 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 all tables. 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.
- 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.
  - Aging is disabled by default on an explicitly loaded global 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.

Cache grid

To cache data in a cache grid, you must create an asynchronous writethrough global cache group. Before you can create this cache group, the TimesTen database must be associated with a cache grid. For more information on creating and using a cache grid and creating and using global cache groups, see "Cache grid" and "Global cache group" in *Oracle In-Memory Database Cache User's Guide*.

Examples

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 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).

SQL syntax

```
CREATE [OR REPLACE] FUNCTION [Owner.]FunctionName
     [(arguments [IN|OUT|IN OUT][NOCOPY] datatype [DEFAULT expr][,...])]
     RETURN datatype [invoker_rights_clause] [DETERMINISTIC]
     {IS|AS} plsql_function_body
```

The syntax for the *`invoker_rights_clause`*:

```
AUTHID {CURRENT_USER|DEFINER}
```

You can specify *`invoker_rights_clause`* or `DETERMINISTIC` in any order.

Parameters

Parameter	Description
`OR REPLACE`	Specify `OR REPLACE` to re-create the function if it already exists. Use this clause to change the definition of an existing function without dropping and re-creating it. When you re-create 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. For more information on PL/SQL data types, see *Oracle TimesTen In-Memory Database PL/SQL Packages Reference*.
`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.

Parameter	Description
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. For more information on NOCOPY, see *Oracle Database SQL Language Reference*.
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. For more information on PL/SQL data types, see *Oracle TimesTen In-Memory Database PL/SQL Packages Reference*.
invoker_rights_clause	Lets you specify whether the function executes with the privileges of the user who owns it or with the privileges of the CURRENT_USER. Specify CURRENT_USER to indicate that the function executes with the privileges of the CURRENT_USER. Specify DEFINER to indicate that the function executes with the privileges of the owner of the function. DEFINER is the default. For more information, see *Oracle Database SQL Language Reference*.
DETERMINISTIC	Specify DETERMINISTIC to indicate that the function should return the same result value whenever it is called with the same values for its parameters. For more information on the DETERMNISTIC clause, see *Oracle Database SQL Language Reference.*
IS|AS	Specify either IS or AS to declare the body of the function.
plsql_function_spec	Specifies the function body.

Restrictions

TimesTen does not support:

- *parallel_enable_clause*. You can specify the clause, but it has no effect.
- *call_spec* clause
- AS EXTERNAL

The CREATE FUNCTION statement is not replicated.

When you create or replace a function, the privileges granted on the function remain the same. If you drop and re-create the object, the object privileges that were granted on the original object are revoked.

Examples

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.
```

See also

Oracle TimesTen In-Memory Database PL/SQL Packages Reference and *Oracle Database SQL Language Reference*

CREATE INDEX

The `CREATE INDEX` statement creates either a range index or a bitmap index on one or more columns of a table or materialized view and assigns a name to the new index.

Required privilege

No privilege is required for table or materialized view owner.

`INDEX` for another user's table or materialized view.

SQL syntax

```
CREATE [UNIQUE|BITMAP] INDEX [Owner.]IndexName ON
[Owner.]TableName ({ColumnName [ASC | DESC]}
[, ... ] )
```

Parameters

Parameter	Description
`UNIQUE`	Prohibits duplicates in the index. If `UNIQUE` is specified, each possible combination of index key column values can occur in only one row of the table. If `UNIQUE` is omitted, duplicate values are allowed. When you create a unique index, all existing rows must have unique values in the indexed columns. If you specify `UNIQUE`, TimesTen creates a range index. A range index: ▪ Speeds up range searches (but can also be used for efficient equality searches) ▪ Is optimized for in-memory data management ▪ Provides efficient sorting by column value
`BITMAP`	Specify `CREATE BITMAP INDEX` to create an index where the information about rows with each unique value is encoded in a bitmap. Each bit in the bitmap corresponds to a row in the table. Use a bitmap index for columns that do not have many unique values.
`[Owner.] IndexName`	Name to be assigned to the new index. A table cannot have two indexes with the same name. If the owner is specified, it must be the same as the owner of the table.
`[Owner.] TableName`	Designates the table 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 16 columns in order from major index key to minor index key.
`[ASC|DESC]`	Specifies the order of the index to be either ascending (the default) or descending. In TimesTen, this parameter is currently ignored.

Description

- If you do not specify `UNIQUE` or `BITMAP`, TimesTen creates a range index.
- Specify a bitmap index on each column to increase the performance of complex queries that specify multiple predicates on multiple columns connected by the `AND` or `OR` operator. At runtime, TimesTen finds bitmaps of rows that satisfy each

predicate and bitmaps from different predicates are combined using bitwise logical operation and then the resultant bitmaps are converted to qualified rows.

- Bitmap indexes are used to satisfy these predicates:
  - Equality predicates. For example: `'x1 = 1'`
  - Range predicates. For example: `'y1 > 10'` and `'z1 BETWEEN 1 and 10'`
  - `AND` predicates. For example: `'x1 > 10 AND y1 > 10'`
  - `OR` predicates. For example: `'x1 > 10 OR y1 > 10'`
  - Complex predicates with AND or OR. For example: `'(x1 > 10 AND y1 > 10) OR (z1 > 10)'`
  - `NOT EQUAL` predicate with `AND`. For example: `'x1 = 1 and y1 != 1'`
- Bitmap indexes:
  - `COUNT` (*) optimization counts rowids from bitmaps.
  - Are used to optimize queries that group by a prefix of columns of the bitmap index.
  - Are used to optimize distinct queries and order by queries.
  - Are used in a `MERGE` join.
- The `CREATE INDEX` statement enters the definition of the index in the system catalog and initializes the necessary data structures. Any rows in the table are then added to the index. In TimesTen, performance is the same regardless of whether the table is created, indexed and populated or created, then populated and indexed.
- If `UNIQUE` is specified, all existing rows must have unique values in the indexed column(s).
- The new index is maintained automatically until the index is deleted by a `DROP INDEX` statement or until the table associated with it is dropped.
- Any prepared statements that reference the table with the new index are automatically prepared again the next time they are executed. Then the statements can take advantage, if possible, of the new index.
- `NULL` compares higher than all other values for sorting.
- An index on a temporary table cannot be created by a connection if any other connection has a non-empty instance of the table.
- If you are using linguistic comparisons, you can create a linguistic index. A linguistic index uses sort key values and storage is required for these values. Only one unique value for `NLS_SORT` is allowed for an index. For more information on linguistic indexes and linguistic comparisons, see "Using linguistic indexes" in *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.

Examples

Create a table and then create a bitmap index on the table. Use the `ttIsql SHOWPLAN` command to verify that the bitmap index is used in the query:

```
Command> CREATE TABLE tab1 (id NUMBER);
```

```
Command> INSERT INTO tab1 VALUES (10);
1 row inserted.
Command> INSERT INTO tab1 VALUES (20);
1 row inserted.
Command> CREATE BITMAP INDEX bitmap_id ON tab1 (id);
Command> COMMIT;
Command> SET AUTOCOMMIT OFF;
Command> SHOWPLAN 1;
Command> SELECT * FROM tab1 WHERE id = 10;

Query Optimizer Plan:

  STEP:                1
  LEVEL:               1
  OPERATION:           RowLkBitmapScan
  TBLNAME:             TAB1
  IXNAME:              BITMAP_ID
  INDEXED CONDITION:   TAB1.ID = 10
  NOT INDEXED:         <NULL>

< 10 >
1 row found.
```

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 T-tree index on columns:
    REGION_ID
    (referenced by foreign key index COUNTR_REG_FK on table SAMPLEUSER.COUNTRIES)
  1 index found.

1 table found.
```

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 `col1` 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 T-tree index on columns:
    COL1
    COL2
  1 index found.

1 table found.
```

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`. If you want to have more than one linguistic sort, create a second linguistic index.

```
Command> CREATE TABLE employees1 (id CHARACTER (21),
id2 character (21));
Command> CREATE INDEX german_index ON employees1
  (NLSSORT(id, 'NLS_SORT=GERMAN'));
Command> CREATE INDEX german_index2 ON employees1
  NLSSORT(id2, 'nls_sort=german_ci'));
Command> indexes employees1;
Indexes on table SAMPLEUSER.EMPLOYEES1:
  GERMAN_INDEX: non-unique T-tree index on columns:
    NLSSORT(ID,'NLS_SORT=GERMAN')
  GERMAN_INDEX2: non-unique T-tree index on columns:
    NLSSORT(ID2,'nls_sort=german_ci')
  2 indexes found.
1 table found.
```

See also

`DROP INDEX`

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 can be refreshed synchronously or asynchronously with regard to changes in the detail tables. If you create an asynchronous materialized view, you must first create a materialized view log on the detail table. See "CREATE MATERIALIZED VIEW LOG" on page 5-79.

Required privilege

- User executing the statement must have CREATE MATERIALIZED VIEW (if owner) or CREATE ANY MATERIALIZED VIEW (if not owner).
- Owner of the materialized view must have SELECT on the detail tables.
- Owner of the materialized view must have CREATE TABLE.

SQL syntax

```
CREATE MATERIALIZED VIEW ViewName
      [REFRESH
        { FAST | COMPLETE } |
              [NEXT SYSDATE[+NUMTODSINTERVAL(IntegerLiteral,IntervalUnit)]]
        |  NEXT SYSDATE[+NUMTODSINTERVAL(IntegerLiteral,IntervalUnit) ]
      ]
      AS SelectQuery
      [PRIMARY KEY (ColumnName [,...])]
      [UNIQUE HASH ON (HashColumnName [,...]) PAGES = PrimaryPages]
```

Parameters

Parameter	Description
ViewName	Name assigned to the new view.
REFRESH	Specifies an asynchronous materialized view.
FAST | COMPLETE	Refresh methods. FAST specifies incremental refresh. COMPLETE specifies full refresh.
NEXT SYSDATE	If NEXT SYSDATE is specified without NUMTODSINTERVAL, the materialized view is refreshed incrementally every time a detail table is modified. The refresh occurs in a separate transaction immediately after the transaction that modifies the detail table has been committed. You cannot specify a full refresh (COMPLETE) every time a detail table is modified. If NEXT SYSDATE is omitted, then the materialized view will not be refreshed automatically. It must be refreshed manually. If NEXT SYSDATE is provided without FAST or COMPLETE specified, COMPLETE is the default refresh method.

Parameter	Description
[+NUMTODSINTERVAL(*IntegerLiteral*, *IntervalUnit*)]	If specified, the materialized view is refreshed at the specified interval. *IntegerLiteral* must be an integer. *IntervalUnit* must be one of the following values: 'DAY', 'HOUR', 'MINUTE', 'SECOND'. If [NEXT SYSDATE[+NUMTODSINTERVAL(*IntegerLiteral*), *IntervalUnit*] is not specified, the materialized view is not refreshed automatically. You can manually refresh the view by using the REFRESH MATERIALIZED VIEW statement.
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 16 columns can be specified for the primary key. Each result column name of a viewed table must be unique. The column name definition cannot contain the table or owner component.
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.
PrimaryPages	Specifies the expected number of pages in the table. This number determines the number of hash buckets created for the hash index. The minimum is 1. If your estimate is too small, performance is degraded. See "CREATE TABLE" on page 5-107 section for more information.

Description

Restrictions on synchronous materialized view and detail tables:

- A materialized view is read-only and cannot be updated directly. A materialized view is updated only when changes are made to the associated detail tables. Therefore a materialized view cannot be the target of a DELETE, UPDATE or INSERT statement.
- Materialized views defined on replicated tables may result in replication failures or inconsistencies if the materialized view is specified so that overflow or underflow conditions occur when the materialized view is updated.
- Detail tables can be replicated, but materialized views themselves cannot be replicated. If detail tables are replicated, TimesTen automatically updates the corresponding views.
- A materialized view and its detail tables cannot be part of a cache group.
- Referential constraints cannot be defined on materialized views.
- If REFRESH is specified, at least one of the refresh options of refresh method (FAST or COMPLETE) or the refresh interval (NEXT SYSDATE) must be specified. If you

omit REFRESH, the materialized view is updated synchronously with updates from the detail tables.

- By default, a range index is created to enforce the primary key for a materialized view. Use the UNIQUE HASH clause to specify a hash index for the primary key.
  - If your application performs range queries over a materialized view's primary key, then choose a 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" on page 5-107 for more information about the UNIQUE HASH clause.
- Use ALTER TABLE to change the representation of the primary key index or resize a hash index.
- You cannot add or drop columns in the materialized view with the ALTER TABLE statement. To change the structure of the materialized view, drop and re-create the view.
- You can create indexes on the materialized view with the CREATE INDEX SQL statement.
- Use the DROP [MATERIALIZED] VIEW statement to drop a materialized view.

There are several restrictions on the query that is used to define the materialized view:

- A SELECT * query in a materialized view definition is expanded when the view is created. Columns added to the detail table after a materialized view is created do not affect the materialized view.
- Temporary tables cannot be used in a materialized view definition. Nonmaterialized views and derived tables cannot be used to define a materialized view.
- All columns in the GROUP BY list must be included in the select list.
- Aggregate view must include a COUNT(*) in the select list.
- SUM and COUNT are allowed, but not expressions involving them, including AVG.
- The following *cannot* be used in a SELECT statement that is creating a materialized view:
  - DISTINCT
  - FIRST
  - HAVING
  - ORDER BY
  - UNION
  - UNION ALL
  - MINUS
  - INTERSECT
  - JOIN
  - User functions: USER, CURRENT_USER, SESSION_USER
  - Subqueries

  - NEXTVAL and CURRVAL
  - Derived tables and joined tables
- Each expression in the select list must have a unique name. The name of a simple column expression is that column's name unless a column alias is defined. ROWID is considered an expression and needs an alias.
- No SELECT FOR UPDATE or SELECT FOR INSERT statements can be used on a view.
- Each inner table can only be outer joined with at most one table.
- Self joins are allowed. A self join is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition.

There are no additional restrictions on asynchronous materialized views with full (COMPLETE) refresh.

In addition to the restrictions in a SELECT statement that is creating a materialized view, the following restrictions apply when creating asynchronous materialized views with incremental (FAST) refresh:

- Aggregate functions are not supported
- Outer joins are not supported.
- The SELECT list must include the ROWID or the primary key columns for all the detail tables.
- The materialized view log must be created for each detail table in the asynchronous material view with incremental refresh before creating the asynchronous materialized view.
- The materialized view log must include all the columns used in the asynchronous materialized views.
- TimesTen creates a unique index for a asynchronous materialized views that are refreshed incrementally. The index is created on the primary key or ROWID of the detail tables included in the SELECT list.

Invalid materialized views

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*.

You can select from an invalid asynchronous materialized view. Refreshing an invalid asynchronous materialized view fails with an error.

Selecting from an invalid synchronous materialized view fails with an error. Updates to the detail tables of an invalid synchronous 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 Limits Reference* .

If the revoked privilege is restored, you can make an invalid materialized view valid again by:

- Dropping and then re-creating the materialized view
- Refreshing an invalid asynchronous materialized view if it was originally specified with complete refreshes

For more information, see "Object privileges for materialized views" in *Oracle TimesTen In-Memory Database Operations Guide*.

Examples

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 (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(+);
```

Create an asynchronous materialized view called `empmatview` with incremental refresh. The materialized view will be refreshed immediately after updates to `employees` have been committed. The columns in `empmatview` are `employee_id` and `email`. You must create a materialized view log before you create an asynchronous materialized view.

```
CREATE MATERALIZED VIEW empmatview
  REFRESH FAST NEXT SYSDATE
  AS SELECT employee_id, email FROM employees;
107 rows materialized.
```

Create an asynchronous materialized view called `empmatview1` with complete refresh. A full refresh of the materialized view occurs every 10 days. The columns in `empmatview` are `employee_id` and `email`. You must create a materialized view log before you create an asynchronous materialized view.

```
CREATE MATERIALIZED VIEW empmatview1
    REFRESH COMPLETE NEXT SYSDATE+NUMTODSINTERVAL(10,'day')
    AS SELECT employee_id, email FROM employees;
107 rows materialized.
```

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 MATERIALIZED VIEW LOG
CREATE TABLE
CREATE VIEW
DROP [MATERIALIZED] VIEW
REFRESH MATERIALIZED VIEW
```

CREATE MATERIALIZED VIEW LOG

The `CREATE MATERIALIZED VIEW LOG` statement creates a log in which changes to the detail table are recorded. The log is required for an asynchronous materialized view that is refreshed incrementally. The log must be created before the materialized view is created. The log is a table in the user's schema called `MVLOG$_`*`detailTableID`*, where *`detailTableID`* is a system-generated ID.

This statement also creates other objects for internal use:

- A global temporary table called `MVLGT$_`*`detailTableID`*
- A sequence called `MVSEQ$_`*`detailTableID`*

The objects are dropped when the `DROP MATERIALIZED VIEW LOG` statement is executed.

Required privileges

`SELECT` on the detail table *and*

`CREATE TABLE` or `CREATE ANY TABLE` (if not owner).

SQL syntax

```
CREATE MATERIALIZED VIEW LOG ON tableName
  [ WITH
       { PRIMARY KEY[, ROWID] |
         ROWID[, PRIMARY KEY } [(columnName[,...])]
       | (columnName[,...])
  ]
```

Parameters

Parameter	Description
`tableName`	Name of the detail table for the materialized view
`[(`*`columnName`*`[,...])`	List of columns for which changes will be recorded in the log. You cannot include the primary key columns in the column list when you specify the `PRIMARY KEY` option.

Description

- Use the `WITH` clause to indicate the keys and columns for which changes will be recorded in the materialized view log. If you specify the `WITH` clause, the following applies:
  - Specify either the `PRIMARY KEY` or `ROWID` when using the `WITH` clause. However, if the `WITH` clause is specified without either option, it defaults implicitly to use `PRIMARY KEY`. In addition, the materialized view log defaults to use `PRIMARY KEY` if the `WITH` clause is omitted altogether.
  - Specify `PRIMARY KEY` to record changes in the primary key columns.
  - Specify the `ROWID` option to record the rowid of all changed rows. The `ROWID` option is useful when the table does not have a primary key or when you do not want to use the primary key when you create the materialized view.

  - You can specify both PRIMARY KEY and ROWID. The materialized view log may be used by more than one asynchronous materialized view using the specified table as the detail table. However, you can only specify one PRIMARY KEY clause, one ROWID clause and one column list for a materialized view log.
- Only one materialized view log is created for a table, even if the table is the detail table for more than one materialized view with FAST refreshes. Make sure to include all the columns that are used in different asynchronous materialized views with FAST refresh.
- A materialized view log cannot be created using a materialized view as the table or for tables in cache groups.
- A materialized view log cannot be altered to add or drop columns.

Examples

Create a materialized view log on the employees table. Include employee_id (the primary key) and email in the log.

```
CREATE MATERIALIZED VIEW LOG ON employees WITH PRIMARY KEY (email);
```

You can create the same materialized view log on the employees table without specifying PRIMARY KEY, which is the default and so is implied, as follows:

```
CREATE MATERIALIZED  VIEW LOG ON employees WITH (email);
```

To create a materialized view log on the employees table with only the primary key, execute the following:

```
CREATE MATERIALIZED VIEW LOG ON employees;
```

Create a materialized view log on the employees table. Include employee_id (the primary key) and row id in the log.

```
Command> create materialized view log on employees with primary key, rowid;
```

Create a materialized view log on the employees table. Include row id in the log.

```
Command> create materialized view log on employees with rowid;
```

Create a materialized view log on the employees table. Include row id, primary key (employee_id) and email in the log.

```
Command> create materialized view log on employees with rowid, primary key
(email);
```

Create a materialized view log on the employees table. Include primary key, by default), and two other columns of email and last_name in the log.

```
Command> create materialized view log on employees with (email, last_name);
```

See also

CREATE MATERIALIZED VIEW
DROP MATERIALIZED VIEW LOG

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).

SQL syntax

```
CREATE [OR REPLACE] PACKAGE [Owner.]PackageName
      [invoker_rights_clause] {IS|AS}
      plsql_package_spec
```

The syntax for the *invoker_rights_clause*:

```
AUTHID {CURRENT_USER | DEFINER}
```

Parameters

Parameter	Description
OR REPLACE	Specify OR REPLACE to re-create 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.
invoker_rights_clause	Lets you specify whether the package executes with the privileges and in the database of the user who owns it or with the privileges and in the database of the CURRENT_USER. Specify CURRENT_USER to indicate that the package executes with the privileges of the CURRENT_USER. Specify DEFINER to indicate that the package executes with the privileges of the owner of the database in which the package resides. DEFINER is the default. For more information, see *Oracle Database SQL Language Reference*.
IS|AS	Specify either IS or AS to declare the body of the function.
plsql_package_spec	Specifies the package specification. Can include type definitions, cursor declarations, variable declarations, constant declarations, exception declarations and PL/SQL subprogram declarations.

Description

The CREATE PACKAGE statement is not replicated.

When you create or replace a package, the privileges granted on the package remain the same. If you drop and re-create the object, the object privileges that were granted on the original object are revoked.

See also

Oracle TimesTen In-Memory Database PL/SQL Packages Reference and *Oracle Database SQL Language Reference*

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).

SQL syntax

```
CREATE [OR REPLACE] PACKAGE BDOY [Owner.]PackageBody
      {IS|AS} plsql_package_body
```

Parameters

Parameter	Description
`OR REPLACE`	Specify `OR REPLACE` to re-create 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

The `CREATE PACKAGE BODY` statement is not replicated.

When you create or replace a package body, the privileges granted on the package body remain the same. If you drop and re-create the object, the object privileges that were granted on the original object are revoked.

See also

Oracle TimesTen In-Memory Database PL/SQL Packages Reference and *Oracle Database SQL Language Reference*

CREATE PROCEDURE

The `CREATE PROCEDURE` statement creates a standalone stored procedure.

Required privilege

`CREATE PROCEDURE` (if owner) or `CREATE ANY PROCEDURE` (if not owner).

SQL syntax

```
CREATE [OR REPLACE] PROCEDURE [Owner.]ProcedureName
     [(arguments [IN|OUT|IN OUT][NOCOPY] datatype [DEFAULT expr][,...])]
     [invoker_rights_clause] [DETERMINISTIC]
     {IS|AS} plsql_procedure_body
```

The syntax for the invoker_rights_clause:

```
AUTHID {CURRENT_USER|DEFINER}
```

You can specify *`invoker_rights_clause`* or `DETERMINISTIC` in any order.

Parameters

Parameter	Description
`OR REPLACE`	Specify `OR REPLACE` to re-create the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping and recreating it. When you re-create 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. For more information on PL/SQL data types, see *Oracle Database SQL Language Reference*.
`[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.

Parameter	Description
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. For more information on NOCOPY, see *Oracle Database SQL Language Reference*
DEFAULT *expr*	Use this clause to specify a DEFAULT value for the parameter. You can specify := in place of the keyword DEFAULT.
invoker_rights_clause	Lets you specify whether the procedure executes with the privileges and in the database of the user who owns it or with the privileges and in the database of the CURRENT_USER. Specify CURRENT_USER to indicate that the procedure executes with the privileges of the CURRENT_USER. Specify DEFINER to indicate that the procedure executes with the privileges of the owner of the database in which the procedure resides. DEFINER is the default. For more information, see *Oracle Database SQL Language Reference*.
DETERMINISTIC	Specify DETERMINISTIC to indicate that the procedure should return the same result value whenever it is called with the same values for its parameters. For more information on the DETERMNISTIC clause, see *Oracle Database SQL Language Reference.*
IS|AS	Specify either IS or AS to declare the body of the procedure.
plsql_procedure_body	Specifies the procedure body.

Restrictions

TimesTen does not support:

- *call_spec* clause
- AS EXTERNAL clause

The CREATE PROCEDURE statement is not replicated.

Description

- 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.
- When you create or replace a procedure, the privileges granted on the procedure remain the same. If you drop and re-create the object, the object privileges that were granted on the original object are revoked.

Examples

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.
```

See also

Oracle TimesTen In-Memory Database PL/SQL Packages Reference and *Oracle Database SQL Language Reference*

CREATE REPLICATION

TimesTen SQL configuration for replication provides a programmable way to configure replication. The configuration can be embedded in C, C++ or Java code. Replication can be configured locally or from remote systems using client/server.

In addition, you need to use the `ttRepAdmin` utility to maintain operations not covered by the supported SQL statements. Use `ttRepAdmin` to change replication state, duplicate databases, list the replication configuration and view replication status.

The `CREATE REPLICATION` statement:

- Defines a replication scheme at 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.

Required privilege

`ADMIN`

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 cache groups. It can include only specified tables and cache groups, or include all tables except specified tables and cache groups. It cannot include temporary tables or views, whether materialized or nonmaterialized.

A *replication scheme* is a set of replication elements, as well as the databases that maintain copies of these elements.

When replicating cache groups:

- When replicating cache groups between databases, both cache groups must be identical, with the exception of the settings for `AUTOREFRESH` and `PROPAGATE`.
- When replicating a cache group with `AUTOREFRESH`, the cache group on the subscriber must set the autorefresh `STATE` to `OFF`. In a bidirectional replication scheme, one of the cache groups must set the autorefresh `STATE` to `OFF`.
- If a master cache group specifies `PROPAGATE`, the subscriber cache group must set the autorefresh `STATE` to `OFF`.

For more detailed information on SQL configuration for replication, see *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*.

SQL syntax

```
CREATE REPLICATION [Owner.]ReplicationSchemeName
{ ELEMENT ElementName
  { DATASTORE | { TABLE [Owner.]TableName [CheckConflicts]} |
       SEQUENCE [Owner.]SequenceName}
     { MASTER | PROPAGATOR } FullStoreName
     [TRANSMIT { NONDURABLE | DURABLE }]
     { SUBSCRIBER FullStoreName [,...]
        [ReturnServiceAttribute] } [, ...] }
```

```
     [...]
     [{INCLUDE | EXCLUDE}
          {TABLE [[Owner.]TableName[,...]] |
           CACHE GROUP [[Owner.]CacheGroupName[,...]] |
           SEQUENCE [[Owner.]SequenceName[,...]} [,...]]
[ STORE FullStoreName [StoreAttribute [... ]]] [...]
[ NetworkOperation[...]]
```

Syntax for *CheckConflicts* is described in "CHECK CONFLICTS" on page 5-94.

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 replication scheme. 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" on page 5-94.
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 TimesTen to TimesTen 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. Use this clause for table-level replication.

Parameter	Description
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. Use this clause 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. See "Defining replication elements" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*.
{INCLUDE|EXCLUDE} {[TABLE[*Owner*.]*TableName* [,...]]| CACHE GROUP [[*Owner*.]*CacheGroupName* [,...]]| SEQUENCE [[*Owner*.]*SequenceName*[,...]]} [,...]	INCLUDE includes in the DATASTORE element only the tables, sequences or cache groups listed. Use one INCLUDE clause for each object type (table, sequence or cache group). EXCLUDE includes in the DATASTORE element all tables, sequences or cache groups except for those listed. Use one EXCLUDE clause for each object type (table, sequence or cache group).
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. See "Managing return service timeout errors and replication state changes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.
DURABLE COMMIT {ON|OFF}	Set to override the DurableCommits setting on a database and enable durable commit when return service blocking has been disabled by DISABLE RETURN.
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 chars. Each *ElementName* must be unique within a replication scheme. Also, you cannot define two element descriptions for the same element. See "Defining replication elements" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.

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 log failure threshold" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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 host IP addresses" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*. Host names containing special characters must be surrounded by double quotes. For example: `"MyHost-500"`. Host names can be up to 30 characters long.
LOCAL COMMIT ACTION{NO ACTION|COMMIT}	Specifies the default action to be taken for a return twosafe transaction in the event of a timeout. **Note**: This attribute is only valid when the `RETURN TWOSAFE` or `RETURN TWOSAFE BY REQUEST` attribute is set in the `SUBSCRIBER` clause. `NO ACTION`: On timeout, the commit function returns to the application, leaving the transaction in the same state it was in when it entered the commit call, with the exception that the application is not able to update any replicated tables. The application can reissue the commit or rollback the call. 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.
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 TimesTen to TimesTen 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.

Parameter	Description
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 TimesTen to TimesTen Replication Guide*
RETURN SERVICES {ON|OFF} WHEN [REPLICATION] STOPPED	Set the return service failure policy so that return service blocking is either unchanged or disabled when the replication agent is in the Stop or Pause state. OFF is the default when using the return receipt service. ON is the default when using the return twosafe service See "Managing return service timeout errors and replication state changes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.
RETURN TWOSAFE [BY REQUEST]	Enables the return twosafe service, so that applications that commit a transaction to a master 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 TimesTen to TimesTen Replication Guide*.
RETURN WAIT TIME *Seconds*	Specifies the number of seconds to wait for return service acknowledgement. The default value is 10 seconds. A value of '0' means that there is no timeout. Your application can override this timeout setting by calling the returnWait parameter in the ttRepSyncSet procedure.
SEQUENCE [*Owner*.]*SequenceName*	Define the sequence specified by [*Owner*.]*SequenceName* as element. See "Defining replication elements" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for details.

Parameter	Description
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 TimesTen to TimesTen Replication Guide* for details.
TIMEOUT *Seconds*	The amount of time a database waits for a response from another database before resending the message. Default: 120 seconds.
TRANSMIT {DURABLE | NONDURABLE}	Specifies whether to flush the master log to disk before sending a batch of committed transactions to the subscribers. TRANSMIT NONDURABLE specifies that records in the master log are not to be flushed to disk before they are sent to subscribers. This setting can only be used if the specified element is a DATASTORE. This is the default for RETURN TWOSAFE transactions. TRANSMIT DURABLE specifies that records are to be flushed to disk before they are sent to subscribers. This is the default for asynchronous and RETURN RECEIPT transactions. **Note**: TRANSMIT DURABLE has no effect on RETURN TWOSAFE transactions. **Note**: TRANSMIT DURABLE cannot be set for active standby pairs. See "Setting transmit durability on database elements" and "Replicating the entire master database with TRANSMIT NONDURABLE" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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 EXACT.
ROUTE MASTER *FullStoreName* SUBSCRIBER *FullStoreName*	Denotes the *NetworkOperation* clause. If specified, allows 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.

Parameter	Description
`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.
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 TimesTen to TimesTen Replication Guide* for more information. The default is UPDATE BY SYSTEM.

Parameter	Description
`ON EXCEPTION {ROLLBACK[WORK |NO ACTION}`	Specifies how to resolve a detected conflict. `ROW TIMESTAMP` conflict detection has the resolution options: ▪ `ROLLBACK [WORK]`: Abort the transaction that contains the conflicting action. ▪ `NO ACTION`: Complete the transaction without performing the conflicting action (`UPDATE`, `INSERT` or `DELETE`). Default is `ON EXCEPTION ROLLBACK [WORK]`.
`REPORT TO 'FileName'`	Specifies the file to log updates that fail the timestamp comparison. *`FileName`* is a SQL character string that cannot exceed 1,000 characters. (SQL character string literals are single-quoted strings that may contain any sequence of characters, including spaces.) The same file can be used to log failed updates for multiple tables.
`[FORMAT {XML|STANDARD}]`	Optionally specifies the conflict report format for an element. The default format is `STANDARD`.
`NO REPORT`	Specify to suppress logging of failed timestamp comparisons.

Description

- The names of all databases on the same host must be unique for each replication scheme for each TimesTen instance.
- Replication elements can only be updated (by normal application transactions) through the `MASTER` database. `PROPAGATOR` and `SUBSCRIBER` databases are read-only.
- If you define a replication scheme that permits multiple databases to update the same table, see "Resolving Replication Conflicts" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for recommendations on how to avoid conflicts when updating rows.
- `SELF` is intended for replication schemes where all participating databases are local. Do not use `SELF` for a distributed replication scheme in a production environment, where spelling out the hostname for each database in a script allows 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 replication scheme specifies it for all replication schemes. All other connection attributes are specific to the 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 disk by checkpoint operations.
- If a row with a default `NOT INLINE VARCHAR` value is replicated, the receiver creates a copy of this value for each row instead of pointing to the default value if and only if the default value of the receiving node is different from the sending node.
- To use timestamp comparison on replicated tables, you must specify a nullable column of type `BINARY(8)` to hold the timestamp value. Define the timestamp column when you create the table. You cannot add the timestamp column with the

ALTER TABLE statement. In addition, the timestamp column cannot be part of a primary key or index.

- If you specify the XML report format, two XML documents are generated:
  - *FileName*.xml: This file contains the DTD for the report and the root node for the report. It includes the document definition and the include directive.
  - *FileName*.include: This file is included in *FileName*.xml and contains all the actual conflicts.
  - The *FileName*.include file can be truncated. Do not truncate the *FileName*.xml file.
  - For a complete description of the XML format, including examples of each conflict, see "Reporting conflicts to an XML file" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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. When the number of conflicts per second exceeds the specified high water mark threshold, conflict resolution reporting (if configured and reported by the report file) and SNMP are suspended and an SNMP trap is emitted to indicate that it has been 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. A SNMP trap is emitted to indicate the resumption of conflict resolution. This trap provides the number of unreported conflicts during the time when conflict resolution was suspended.
- 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 dynamic read-only cache groups asynchronously. Use 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).

SQL syntax

```
CREATE SEQUENCE [Owner.]SequenceName
       [INCREMENT BY IncrementValue]
       [MINVALUE MinimumValue]
       [MAXVALUE MaximumValue]
       [CYCLE]
       [CACHE CacheValue]
       [START WITH StartValue]
```

Parameters

Parameter	Description
`SEQUENCE [Owner.]SequenceName`	Name of the sequence number generator.
`INCREMENT BY IncrementValue`	The incremental value between consecutive numbers. This value can be either a positive or negative integer. It cannot be 0. If the value is positive, it is an ascending sequence. If the value is negative, it is descending. The default value is 1. In a descending sequence, the range starts from `MAXVALUE` to `MINVALUE`, and vice versa for ascending sequence.
`MINVALUE MinimumValue`	Specifies the 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.

Parameter	Description
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*.

Description

- All parameters in the CREATE SEQUENCE statement must be integer values.
- If you do not specify a value in the parameters, TimesTen defaults to an ascending sequence that starts with 1, increments by 1, has the default maximum value and does not cycle.
- There is no ALTER SEQUENCE statement in TimesTen. To alter a sequence, use the DROP SEQUENCE statement and then create a new sequence with the same name. For example, to change the MINVALUE, drop the sequence and re-create it with the same name and with the desired MINVALUE.
- Do not create a sequence with the same name as a view or materialized view.

Incrementing SEQUENCE values with CURRVAL and NEXTVAL

To refer to the SEQUENCE values in a SQL statement, use CURRVAL and NEXTVAL.

- CURRVAL returns the value of the last call to NEXTVAL if there is one in the current session, otherwise it returns an error.
- NEXTVAL increments the current sequence value by the specified increment and returns the value for each row accessed.
- NEXTVAL and CURRVAL can be used in:
  - The *SelectList* of a SELECT statement, but not the *SelectList* of a subquery
  - The *SelectList* of an INSERT...SELECT statement
  - The SET clause of an UPDATE statement
- In a single SQL statement with multiple NEXTVAL references, TimesTen only increments the sequence once, returning the same value for all occurrences of NEXTVAL.
- If a SQL statement contains both NEXTVAL and CURRVAL, NEXTVAL is executed first. CURRVAL and NEXTVAL have the same value in that SQL statement.
- The current value of a sequence is a connection-specific value. If there are two concurrent connections to the same database, each connection has its own CURRVAL of the same sequence set to its last NEXTVAL reference.
- In the case of recovery, sequences are not rolled back. It is possible that the range of values of a sequence can have gaps. Each sequence value is still unique.
- When the maximum value is reached, SEQUENCE either wraps or issues an error statement, depending on the value of the CYCLE option of the CREATE SEQUENCE.

Note: Sequences with the CYCLE attribute cannot be replicated.

Examples

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

`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.

SQL syntax

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema1.]synonym FOR [schema2.]object
```

Parameters

Parameter	Description
`[OR REPLACE]`	Specify `OR REPLACE` to re-create 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.
`[`*`schema1`*`.]`*`synonym`*	Specify the schema to contain the synonym. You cannot specify a schema for the synonym if you have specified `PUBLIC`. If you omit both `PUBLIC` and *`schema1`*, TimesTen creates the synonym in your own schema. Specify the name for the synonym, which is limited to 30 bytes.

Parameter	Description
[*schema2*.]*object*	Specify the schema in which the object resides. Specify the object name for which you are creating a synonym. If you do not qualify *object* with *schema2*, the schema object is in your own schema. The *schema2* 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.
- 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 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.

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.

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.

SQL syntax

The syntax for a persistent table is:

```
CREATE TABLE [Owner.]TableName
(
    {{ColumnDefinition} [,...]
     [PRIMARY KEY (ColumnName [,...]) |
     [[CONSTRAINT ForeignKeyName]
        FOREIGN KEY ([ColumnName] [,...])
        REFERENCES RefTableName
            [(ColumnName [,...])] [ON DELETE CASCADE]] [...]
    }
)
[UNIQUE HASH ON (HashColumnName [,...])
    PAGES = PrimaryPages]
[AGING {LRU|
         USE ColumnName
             LIFETIME Num1 {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 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

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 table owner names must always match TimesTen table owner names. For rules on creating names, see "Basic names" on page 2-1.
GLOBAL TEMPORARY	Specifies that the table being created is a temporary table. A temporary table is similar to a persistent table but it is effectively materialized only when referenced in a connection. A global temporary table definition is persistent and is visible to all connections, but the table instance is local to each connection. It is created when a command referencing the table is compiled for a connection and dropped when the connection is disconnected. All instances of the same temporary table have the same name but they are identified by an additional connection ID together with the table name. Global temporary tables are allocated in temp space. The contents of a temporary table cannot be shared between connections. Each connection sees only its own content of the table and compiled commands that reference temporary tables are not shared among connections. Temporary tables cannot be used as part of a cache group or a replication scheme. Temporary tables are automatically excluded when DATASTORE level replication is defined. A cache group table cannot be defined as a temporary table. Changes to temporary tables cannot be tracked with XLA. Operations on temporary tables do generate log records. The amount of log they generate is less than for permanent tables. 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. See "Column Definition" on page 5-112. If you specify the AS *SelectQuery* clause, *ColumnDefinition* is optional.
ColumnName	Names of the columns that form 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, then 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.

Parameter	Description
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.
[ON DELETE CASCADE]	Enables the ON DELETE CASCADE referential action. If specified, when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign key values are also deleted.
UNIQUE	UNIQUE provides a way of identifying a column where each row must contain a unique value.
UNIQUE HASH ON	Hash index for the table. Only unique hash indexes are created. This parameter is used for equality predicates. 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.
PrimaryPages	Specifies the expected number of pages in the table. This number affects the number of buckets that are allocated for the table's hash index. The minimum is 1. If your estimate is too small, performance is degraded.

Parameter	Description
`[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` procedure. LRU attributes are not defined at the SQL level. For more information about LRU aging, see "Implementing aging in your tables" in *Oracle TimesTen In-Memory Database Operations Guide*.
`[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 aging 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.

Column Definition

SQL syntax

```
ColumnName ColumnDataType
 [DEFAULT DefaultVal]
 [[NOT] INLINE]
 [PRIMARY KEY | UNIQUE |
 NULL [UNIQUE] |
 NOT NULL [PRIMARY KEY | UNIQUE] ]
```

Parameters

The column definition has the 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*.
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 compatible type with the column's data type. A default value can be as long as the data type of the associated column allows. Currently, you cannot assign a default value for the ROWID data type. In addition, you cannot assign a default value for columns in read-only cache groups. Legal data types for *DefaultVal* can be one of: ■ NULL ■ *ConstantValue*. See "Constants" on page 3-11 ■ SYSDATE and GETDATE ■ INSERT ■ *Expression*. See "Expressions" on page 3-1 ■ SYSTEM_USER If the default value is one of the users, the column's data type must be either CHAR or VARCHAR2 and the column's width 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.
ColumnDataType	Type of data the column can contain. Some data types require that you indicate a length. See Chapter 1, "Data Types" for the data types that can be specified. If you specify the AS *SelectQuery* clause, do not specify *ColumnDataType*.
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.

Parameter	Description
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.
NOT NULL	Indicates that the column cannot contain NULL values. If NOT NULL is specified, any statement that attempts to place a NULL value in the column is rejected. If you specify the AS *SelectQuery* clause, optionally, you can specify NOT NULL on the table you are creating.
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. This means that a column with a unique constraint can use more memory and time during execution than a column without the constraint. Cannot be used with PRIMARY KEY. 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

- TimesTen supports one hash index per table. A hash index is defined on the primary key of a table.
- 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.
- Use the ALTER TABLE statement to change the representation of the primary key index for a table.
- A hash index is created with a fixed number of buckets that remains constant for the life of the table or until the hash index is resized using an ALTER TABLE statement to change hash index size. Fewer buckets in the hash index result in more hash collisions. More buckets reduce collisions but can waste memory. Hash key comparison is a fast operation, so a small number of hash collisions does not cause a performance problem for TimesTen.

  The bucket count is derived as the ratio of the maximum table cardinality, derived from the value of PAGES, to the value 20.To ensure that the hash index is sized correctly, an application must indicate the expected size of the table. This is done with the PAGES parameter. The PAGES parameter should be the expected number of rows in the table, divided by 256. (Since 256 is the number of rows TimesTen stores on each page, the value provided is the expected number of pages in the table.) The application may specify a larger value for PAGES, and therefore fewer rows per bucket on average, if memory use is not an overriding concern.
- At most 16 columns are allowed in a hash key.

- All columns participating in the primary key are `NOT NULL`.
- A unique hash index can be specified only for the primary key.
- A `PRIMARY KEY` that is specified in the *`ColumnDefinition`* can only be specified for one column.
- `PRIMARY KEY` cannot be specified in both the *`ColumnDefinition`* parameters and `CREATE TABLE` parameters.
- For both primary key and foreign key constraints, duplicate column names are not allowed in the constraint column list.
- You cannot create a table that has a foreign key referencing a cached table.
- `UNIQUE` column constraint and default column values are not supported with materialized views.
- 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.
  - You must stop the replication agent before adding or dropping a foreign key on a replicated table.

- The data in a global temporary 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.
- 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*.
- *SelectQuery* cannot contain set operators UNION, MINUS, INTERSECT.
- Global temporary tables are not supported.

Examples

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 T-tree index on columns:
   PARTNUMBER
1 index found.
1 table found.
```

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 T-tree index on columns:
```

```
    MEMBERNAME
    CLUB
  1 index found.
1 table found.
```

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 T-tree index on columns:
    COL1
  1 index found.
1 table found.
```

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 3 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. The row is not aged out if you specified a lifetime of 3 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 3 days. Cycle is not specified, so the default is 5 minutes. Aging state is `OFF`.

```
CREATE TABLE agingdemo2
       (agingid NUMBER NOT NULL PRIMARY KEY
        ,name  VARCHAR2 (20)
        ,agingcolumn TIMESTAMP NOT NULL
        )
        AGING USE agingcolumn LIFETIME 3 DAYS OFF;
Command> DESCRIBE agingdemo2;
Table USER.AGINGDEMO2:
  Columns:
     *AGINGID NUMBER NOT NULL
     NAME VARCHAR2 (20) INLINE
     AGINGCOLUMN TIMESTAMP (6) NOT NULL
  Aging use AGINGCOLUMN lifetime 3 days cycle 5 minutes off
1 table found.
(primary key columns are indicated with *)
```

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 6 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 employee_id 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 *)
```

See also

ALTER TABLE
DROP TABLE
TRUNCATE TABLE
UPDATE

CREATE USER

The `CREATE USER` statement creates a user of a TimesTen database.

Required privilege

`ADMIN`

SQL syntax

```
CREATE USER user IDENTIFIED BY {password | "password"}
CREATE USER user IDENTIFIED EXTERNALLY
```

Parameters

Parameter	Description
user	Name of the user that is being added to the database.
IDENTIFIED	Identification clause.
BY {*password* |"*password*"}	Internal users must be given a TimesTen password. To perform database operations using an internal user name, the user must supply this password.
EXTERNALLY	Identifies the operating system *user* to the TimesTen database. To perform database operations as an external user, the process needs a TimesTen external user name that matches the user name authenticated by the operating system or network. A password is not required by TimesTen because the user has been authenticated by the operating system at login time.

Description

- Database users can be internal or external.
  - Internal users are defined for a TimesTen database.
  - External users are defined by an external authority such as the operating system. External users cannot be assigned a TimesTen password.
- Passwords are case-sensitive.
- When a user is created, the user has the privileges granted to `PUBLIC` and no additional privileges.
- You cannot create a user across a client/server connection. You must use a direct connection when creating a user.
- 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.

Examples

To create the internal user `terry` with the password `"secret"`, use:

```
CREATE USER terry IDENTIFIED BY "secret";
User created.
```

Verify that user `terry` has been created:

```
Command> SELECT * FROM sys.all_users WHERE username='TERRY';
< TERRY, 11, 2009-05-12 10:28:04.610353 >
1 row found.
```

To identify the external user `pat` to the TimesTen database, use:

```
CREATE USER pat IDENTIFIED EXTERNALLY;
User created.
```

See also

ALTER USER
DROP USER
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.

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.

SQL syntax

```
CREATE VIEW ViewName AS SelectQuery
```

Parameters

Parameter	Description
`ViewName`	Name assigned to the new view.
`SelectQuery`	Selects column from the detail tables to be used in the view. 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.
- The following cannot be used in a `SELECT` statement that is used to create a view:
  - `DISTINCT`
  - `FIRST`
  - `ORDER BY`, if used, is ignored by the `CREATE VIEW` statement. The result will not be sorted.
  - Arguments
  - Temporary tables
- Each expression in the select list must have a unique name. A name of a simple column expression would be that column's name unless a column alias is defined. `ROWID` is considered an expression and needs an alias.
- No `SELECT FOR UPDATE` or `SELECT FOR INSERT` statements can be used to create a view.
- Certain TimesTen query restrictions are not checked when a nonmaterialized 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 legal query without the derived table, the content of this derived table is materialized. For example, if both the view and the referencing select specify aggregates, the view is materialized before its result can be joined with other tables of the select.

- Use the `DROP [MATERIALIZED] 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 [MATERIALIZED] 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.

SQL syntax

```
DELETE [FIRST NumRows] FROM [Owner.]TableName [CorrelationName]
[WHERE SearchCondition]
[RETURNING|RETURN Expression[,...]INTO DataItem[,...]]
```

Parameters

Parameter	Description
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" on page 2-1.
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 Chapter 3, "Expressions".
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.
- The DELETE operation fails if it violates any foreign key constraint. See "CREATE TABLE" on page 5-107 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 *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

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 drops an active standby pair replication scheme.

Required privilege

ADMIN

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 node on which the statement was issued.

See also

ALTER ACTIVE STANDBY PAIR
CREATE ACTIVE STANDBY PAIR

DROP CACHE GROUP

The `DROP CACHE GROUP` statement drops the table associated with the cache group, and removes the cache group definition from the `CACHE_GROUP` system table.

Required privilege

No privilege is required for the cache group owner or `DROP ANY CACHE GROUP` if not the cache group owner *and*

`DROP ANY TABLE` if at least one table in the cache group is not owned by the current user.

SQL syntax

```
DROP CACHE GROUP [Owner.]GroupName
```

Parameters

Parameter	Description
`[Owner.]GroupName`	Name of the cache group to be deleted.

Description

- If you attempt to delete a cache group table that is in use, TimesTen returns an error.
- Asynchronous writethrough cache groups cannot be dropped while the replication agent is running.
- Automatically installed Oracle objects for 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 objects are uninstalled on the next startup of the cache agent.
- If you issue a `DROP CACHE GROUP` statement, and there is an autorefresh operation currently running, then:
  - If `LockWait` interval is 0, the `DROP CACHE GROUP` statement fails with a lock timeout error.
  - If `LockWait` interval is non-zero, 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.

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 re-created 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 removes the specified index.

Required privilege

No privilege is required for the index owner.

`DROP ANY INDEX` for another user's index.

SQL syntax

```
DROP INDEX [Owner.]IndexName [FROM [Owner.]TableName]
```

Parameters

Parameter	Description
`[Owner.]IndexName`	Name of the index to be dropped. It may include the name of the owner of the table that has the index.
`[Owner.]TableName`	Name of the table upon which the index was created.

Description

- If you attempt to drop a "busy" index—an index that is in use or that enforces a foreign key—an error results. To drop a foreign key and the index associated with it, use the `ALTER TABLE` statement.
- If an index is created through a `UNIQUE` column constraint, it can only be dropped by dropping the constraint with an `ALTER TABLE DROP UNIQUE` statement. See "CREATE TABLE" on page 5-107 for more information about the `UNIQUE` column constraint.
- If a `DROP INDEX` operation is or was active in an uncommitted transaction, other transactions doing DML operations that do not access that index are blocked.
- If an index is dropped, any prepared statement that uses the index is prepared again automatically the next time the statement is executed.
- If no table name is specified, the index name must be unique for the specified or implicit owner. The implicit owner, in the absence of a specified table or owner, is the current user running the program.
- If no index owner is specified and a table is specified, the default owner is the table owner.
- If a table is specified and no owner is specified for it, the default table owner is the current user running the program.
- The table and index owners must be the same.
- An index on a temporary table cannot be dropped by a connection if some other connection has an instance of the table that is not empty.
- If the index is used as the unique index for replication, you must drop the replication scheme before you can drop the index.

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 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.

SQL syntax

```
DROP [MATERIALIZED] VIEW ViewName
```

Parameters

Parameter	Description
`MATERIALIZED`	Specifies that the view is materialized.
`ViewName`	Identifies the view to be dropped.

Description

When you perform a `DROP VIEW` operation on a materialized view, the detail tables are updated and locked. An error may result if the detail table was already locked by another transaction.

Examples

The following statement drops the `custorder` view.

```
DROP VIEW custorder;
```

See also

`CREATE MATERIALIZED VIEW`
`CREATE VIEW`

DROP MATERIALIZED VIEW LOG

The `DROP MATERIALIZED VIEW LOG` statement removes the materialized view log for a detail table. It also drops the global temporary table that was created by the `CREATE MATERIALIZED VIEW LOG` statement.

Required privilege

No privilege is required for the table owner.

`DROP ANY TABLE` for another user's table.

SQL syntax

```
DROP MATERIALIZED VIEW LOG ON TableName
```

Parameters

Parameter	Description
TableName	Name of the detail table for which the materialized view log was created.

Description

This statement drops the materialized view log for the specified detail table. The materialized view log cannot be dropped if there is an asynchronous materialized view that depends on the log for refreshing.

Examples

```
DROP MATERIALIZED VIEW LOG ON employees;
```

See also

`CREATE MATERIALIZED VIEW LOG`
`CREATE MATERIALIZED VIEW`
`DROP [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.

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 re-create 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 re-created the dropped package.
- Do not use this statement to remove a single object from the package. Instead, re-create 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.

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 re-created 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 database
The command failed.
```

See also

`CREATE PROCEDURE`

DROP REPLICATION

The `DROP REPLICATION` statement destroys a replication scheme and removes it from the executing database.

Required privilege

`ADMIN`

SQL syntax

```
DROP REPLICATION [Owner.]ReplicationSchemeName
```

Parameters

Parameter	Description
`[Owner.]ReplicationSchemeName`	Name assigned to the replication scheme.

Description

Dropping the last replication scheme at a database does not delete the replicated tables. These tables exist and persist at a database whether or not any replication schemes are defined.

Examples

The following statement erases the executing database's knowledge of replication scheme, `r`:

```
DROP REPLICATION r;
```

See also

`ALTER REPLICATION`
`CREATE REPLICATION`

DROP SEQUENCE

The `DROP SEQUENCE` statement removes an existing sequence number generator.

Required privilege

No privilege is required for the sequence owner.

`DROP ANY SEQUENCE` for another user's sequence.

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.
- There is no `ALTER SEQUENCE` statement in TimesTen. To alter a sequence, use the `DROP SEQUENCE` statement and then create a new sequence with the same name. For example, to change the `MINVALUE`, drop the sequence and re-create 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.

Required privilege

To drop a private synonym, either the synonym must be in your own schema or you must have the `DROP ANY SYNONYM` privilege.

To drop a `PUBLIC` synonym, you must have the `DROP PUBLIC SYNONYM` privilege.

SQL syntax

```
DROP [PUBLIC] SYNONYM [schema.]synonym
```

Parameters

Parameter	Description
`[PUBLIC]`	Specify `PUBLIC` to drop a public synonym.
`[schema.]synonym`	Specify the name of the synonym and, optionally, if the synonym is a private synonym, the schema. If you omit PUBLIC and the schema name, the synonym must be contained in the user's own schema.

Description

`DROP SYNONYM` is a DDL statement.

Examples

Drop the public synonym `pubemp`:

```
DROP PUBLIC SYNONYM pubemp;
Synonym dropped.
```

Drop the `synjobs` synonym:

```
DROP SYNONYM synjobs;
Synonym dropped.
```

As user `terry` with `DROP ANY SYNONYM` privilege, drop the `syntab` synonym created 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.

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 a `DROP TABLE` operation is or was active in an uncommitted transaction, other transactions doing DML operations that do not access that table are allowed to proceed.
- If the table is a replicated table, you can use the `DROP REPLICATION` statement to drop the replication scheme before issuing the `DROP TABLE` statement. You can also use the `ALTER REPLICATION` statement to drop the table from the replication scheme. Once you drop the table from the replication scheme, you can use the `DROP TABLE` statement to drop the table.
- 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`

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.

Examples

Drop user `terry` from the database:

```
DROP USER terry;
User dropped.
```

See also

CREATE USER
ALTER USER
GRANT
REVOKE

FLUSH CACHE GROUP

The `FLUSH CACHE GROUP` statement flushes data from TimesTen cache tables to Oracle tables. This statement is available only for user managed cache groups. For a description of cache group types, see "User managed and system managed cache groups" on page 5-53.

There are two variants to this operation: one that accepts a `WHERE` clause, and one that accepts a `WITH ID` clause.

`FLUSH CACHE GROUP` is meant to be used when commit propagation (from TimesTen to Oracle) is turned off. Instead of propagating every transaction upon commit, many transactions can be committed before changes are propagated to Oracle. For each cache instance ID, if the cache instance exists in 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 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 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 table. Deletes must be propagated either manually or by turning commit propagation on. Attempts to flush deleted records are silently ignored. No error or warning is issued. Records from tables that are specified as `READ ONLY` or `PROPAGATE` cannot be flushed to Oracle tables.

Required privileges

No privilege is required for the cache group owner.

`FLUSH` or `FLUSH ANY CACHE GROUP` for another user's cache group.

SQL syntax

```
FLUSH CACHE GROUP [Owner.]GroupName
[WHERE ConditionalExpression];
```

or

```
FLUSH CACHE GROUP [Owner.]GroupName
WITH ID (ColumnValueList)
```

Parameters

Parameter	Description
`[Owner.]GroupName`	Name of the cache group to be flushed.
`ConditionalExpression`	A search condition to qualify the target rows of the operation.

Parameter	Description
`WITH ID` *`ColumnValueList`*	The `WITH ID` clauses allows 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 instances, rather than to a single instance or to all instances. The flush operation uses the `WHERE` clause to determine which instances to send to the Oracle database.
- All table names used in cache group `WHERE` clauses should be fully qualified with an owner name to allow other users to execute the same `WHERE` clauses against the same cache group. Without an owner name, all tables referenced by cache group `WHERE` clauses are 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 Oracle 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 on AWT or SWT cache groups, user managed cache groups with the propagate attribute, or autorefreshed and propagated user managed cache groups unless the cache group is a dynamic cache group.
- Do not use the `WITH ID` clause with the `COMMIT EVERY` *`n`* `ROWS` clause.

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.

SQL syntax

```
GRANT {SystemPrivilege [,...] | ALL [PRIVILEGES]} [...] TO {user |PUBLIC} [,...]

GRANT {{ObjectPrivilege [,...] | ALL [PRIVILEGES]} ON {[owner.]object}[,...]} TO
{user | PUBLIC} [,...]
```

Parameters

The following parameters are for granting system privileges:

Parameter	Description
SystemPrivilege	See "System privileges" on page 6-1 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	See "Object privileges" on page 6-3 for a list of acceptable values.
ALL [PRIVILEGES]	Assigns all object privileges to the user.
[*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.

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" on page 6-5

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 functions
- Expressions
- SYSTEM_USER

Required privilege

No privilege is required for the table owner.

INSERT for another user's table.

SQL syntax

```
INSERT INTO [Owner.]TableName [(Column [,...])]
VALUES (SingleRowValues)
[RETURNING|RETURN Expression[,...] INTO DataItem[,...]]
```

The *SingleRowValues* parameter has the syntax:

```
{NULL|{?|:DynamicParameter}|{Constant}| DEFAULT}[,...]
```

Parameters

Parameter	Description
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*	Place holder for a dynamic parameter in a prepared SQL statement. The value of the dynamic parameter is supplied when the statement is executed.
Constant	A specific value. See "Constants" on page 3-11.

Parameter	Description
DEFAULT	Specifies that the column should be updated with the default value.
Expression	Valid expression syntax. See Chapter 3, "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" on page 5-107 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');
```

:pno and :pname are dynamic parameters whose 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
Chapter 3, "Expressions"

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.

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" on page 5-166.

Description

- The column types of the result set must be compatible with the column types of the target table.
- You can specify a sequence `CURRVAL` or `NEXTVAL` when inserting values.
- The target table cannot be referenced in the `FROM` clause of the *`InsertQuery`*.
- In the *`InsertQuery`*, the `ORDER BY` clause is allowed. The sort order may be modified using the `ORDER BY` clause when the result set is inserted into the target table, but the order is not guaranteed.
- The `INSERT` operation fails if there is an error in the *`InsertQuery`*.
- A `RETURNING` clause cannot be used in an `INSERT...SELECT` statement.

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 an Oracle table into a TimesTen cache group. The load operation is local. It is not propagated across cache grid members.

Required privilege

No privilege is required for the cache group owner.

`LOAD CACHE GROUP` or `LOAD ANY CACHE GROUP` for another user's cache group.

SQL syntax

```
LOAD CACHE GROUP [Owner.]GroupName
[WHERE ConditionalExpression]
COMMIT EVERY n ROWS
[PARALLEL NumThreads]
```

or

```
LOAD CACHE GROUP [Owner.]GroupName
WITH ID (ColumnValueList)
```

Parameters

Parameter	Description
`[Owner.]GroupName`	Name assigned to the cache group.
`ConditionalExpression`	A search condition to qualify the target rows of the operation.
`n`	The number of rows to insert into the cache group before committing the work. It must be a nonnegative integer. If it is 0, the entire 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 Oracle 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 allows 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.

Description

- `LOAD CACHE GROUP` loads all new instances from Oracle 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.
- `LOAD CACHE GROUP` is executed in its own transaction, and must be the first operation in a transaction.

- For an explicitly loaded cache group, `LOAD CACHE GROUP` does not update cache instances that are already present in the TimesTen cache tables. Therefore, `LOAD CACHE GROUP` loads only inserts on Oracle tables into the corresponding TimesTen cache tables.
- For a dynamic cache group, `LOAD CACHE GROUP` loads rows that have been inserted, updated and deleted on Oracle tables into the cache tables. For more information about explicitly loaded and dynamic cache groups, see *Oracle In-Memory Database Cache User's Guide*.
- The transaction size is the number of rows inserted before committing the work. The value of *n* in `COMMIT EVERY n ROWS` must be nonnegative and is rounded up to the nearest multiple of 256 for performance reasons.
- Errors cause a rollback. When rows are committed periodically, errors abort the remainder of the load. The load is rolled back to the last commit.
- If the `LOAD` statement fails when you specify `COMMIT EVERY n ROWS` (where *n* is greater than 0), the content of the target cache group could be in an inconsistent state. Some cache instances may be partially loaded. Use the `UNLOAD` statement to unload the cache group, then load again.
- Table names in subqueries in the `WHERE` clause of the `LOAD CACHE GROUP` statement must be fully qualified.
- 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.
- If the automatic refresh state of a cache group (explicitly loaded or dynamic) is `PAUSED`, the state is changed to `ON` after a `LOAD CACHE GROUP` statement that was issued on the cache group completes.
- If the automatic refresh state of a dynamic cache group is `PAUSED` and the cache tables are populated, the state remains `PAUSED` after a `LOAD CACHE GROUP` statement that was issued on the cache group completes.
- 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
  - If you want to roll back the load transaction upon failure

Restrictions

- 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:
  - Explicitly loaded read-only cache group
  - Explicitly loaded 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.
- The `WITH ID` clause cannot be used to acquire a cache instance from another cache grid member.

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 (where *n* is greater than 0) 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 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

The MERGE statement allows you to select rows from one or more sources for update or insertion into a target table. You can specify conditions that are used to evaluate 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.

SQL syntax

```
MERGE 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
[*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 if you want to use joins or aggregates. Optionally, you can specify an alias for the table name or the subquery.

Parameter	Description
ON (*Condition*)	You specify the condition that is used to evaluate each row of the target table to determine if the row should be considered for either a merge insert or a merge update. If the condition is true when evaluated, then the *MergeUpdateClause* is considered for the target row using the matching row from the *SourceTableName*. An error is generated if more than one row in the source table matches the same row in the target table. If the condition is not true when evaluated, then the *MergeInsertClause* is considered for that row.
SET *SetClause*	Clause used with the UPDATE statement. For information on the UPDATE statement, see "UPDATE" on page 5-188.
[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. For more information on the INSERT statement, see "INSERT" on page 5-147.
[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 the contacts table with values 101 and NULL for employee_id and manager_id, respectively. The MERGE statement is used to insert rows into the contacts table using the data in the employees table. A SELECT FIRST 3 rows is used to illustrate that in the case where employee_id is equal to 101, manager_id is updated to 100. The remaining 106 rows from the employees table are inserted into the contacts table:

```
Command> CREATE TABLE contacts (employee_id NUMBER (6) NOT NULL PRIMARY KEY,
       > manager_id NUMBER (6));
Command> SELECT employee_id, manager_id FROM employees WHERE employee_id =101;
< 101, 100 >
1 row found.
Command> INSERT INTO contacts VALUES (101,null);
1 row inserted.
Command> SELECT COUNT (*) FROM employees;
< 107 >
1 row found.
Command> MERGE INTO contacts c
       >   USING employees e
       >   ON (c.employee_id = e.employee_id)
       > WHEN MATCHED THEN
       >   UPDATE SET c.manager_id = e.manager_id
       > WHEN NOT MATCHED THEN
       >   INSERT (employee_id, manager_id)
       >   VALUES (e.employee_id, e.manager_id);
107 rows merged.
Command> SELECT COUNT (*) FROM contacts;
< 107 >
1 row found.
Command> SELECT FIRST 3 employee_id,manager_id FROM employees;
< 100, <NULL> >
< 101, 100 >
< 102, 100 >
3 rows found.
```

```
Command> SELECT FIRST 3 employee_id, manager_id FROM contacts;
< 100, <NULL> >
< 101, 100 >
< 102, 100 >
3 rows found.
```

REFRESH CACHE GROUP

The `REFRESH CACHE GROUP` statement is equivalent to an `UNLOAD CACHE GROUP` statement followed by a `LOAD CACHE GROUP` statement.

Required privilege

- `CREATE SESSION` on the Oracle schema and `SELECT` on the Oracle tables.
- No privilege for the cache group is required for the cache group owner.
- `REFRESH CACHE GROUP` or `REFRESH ANY CACHE GROUP` for another user's cache group.

SQL syntax

```
REFRESH CACHE GROUP [Owner.]GroupName
[WHERE ConditionalExpression]
COMMIT EVERY n ROWS
[PARALLEL NumThreads]
```

or

```
REFRESH CACHE GROUP [Owner.]GroupName
WITH ID (ColumnValueList)
```

Parameters

Parameter	Description
`[Owner.]GroupName`	Name assigned to the cache group.
`ConditionalExpression`	A search condition to qualify the target rows of the operation.
`n`	The number of rows to insert into the cache group before committing the work. The value must be a nonnegative integer. If the value is 0, the entire 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 Oracle 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 allows 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.

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.

- `REFRESH CACHE GROUP` replaces all or specified cache instances in the TimesTen cache tables with the most current data from the corresponding Oracle tables even if an instance is already present in the cache tables. For explicitly loaded cache groups, a refresh operation is equivalent to an `UNLOAD CACHE GROUP` statement followed by a `LOAD CACHE GROUP` statement. Operations on all rows in the Oracle 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 Oracle tables into the cache tables. For more information on explicitly loaded and dynamic cache groups, see *Oracle In-Memory Database Cache User's Guide*.
- When refreshing a read-only cache group:
  - The `AUTOREFRESH` statement must be paused, *and*
  - The `REFRESH` statement cannot have a `WHERE` clause unless the cache group is a dynamic cache group.
- 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.
- Table names in subqueries in the `WHERE` clause of the `REFRESH CACHE GROUP` statement must be fully qualified.
- If the `REFRESH CACHE GROUP` statement fails when you specify `COMMIT EVERY` *`n`* `ROWS` (where *`n`* is greater than 0), the content of the target cache group could be in an inconsistent state. 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
  - If you want to roll back the refresh transaction upon failure

Restrictions

- Do not specify the `PARALLEL` clause:
  - 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 refreshing these types of cache groups:
  - Explicitly loaded read-only cache groups
  - Explicitly loaded 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.

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` (where *n* is greater than 0) 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

REFRESH MATERIALIZED VIEW

The `REFRESH MATERIALIZED VIEW` statement refreshes an asynchronous materialized view manually.

Required privilege

Required privilege on the materialized view log tables:

- No privilege is required for the owner of the materialized view log tables.
- `SELECT ANY TABLE` if not the owner of materialized view log tables.

Required privilege on the materialized view:

- No privilege is required for the owner of the materialized view.
- `SELECT ANY TABLE` if not the owner of materialized view.

SQL syntax

```
REFRESH MATERIALIZED VIEW ViewName
```

Parameters

Parameter	Description
ViewName	Name of the asynchronous materialized view

Description

This statement refreshes the specified asynchronous materialized view. It is executed in a separate thread as a separate transaction and committed. The user transaction is not affected, but the user thread waits for the refresh operation to be completed before returning to the user. If you have not specified a refresh interval for an asynchronous materialized view, using this statement is the only way to refresh the view. If you have specified a refresh interval, you can still use this statement to refresh the view manually.

Since the refresh operation is always performed in a separate transaction, the refresh operation does not wait for any uncommitted user transactions to commit. Only the committed rows are considered for the refresh operation. This is true for the manual refresh statement as well as the automatic refresh that takes place at regular intervals.

If the `CREATE MATERIALIZED VIEW` statement for the view specified a `FAST` refresh, then the `REFRESH MATERIALIZED VIEW` statement uses the incremental refresh method. Otherwise this statement uses the full refresh method.

Examples

```
REFRESH MATERALIZED VIEW bookorders;
```

See also

`CREATE MATERIALIZED VIEW`
`DROP [MATERIALIZED] VIEW`

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.

SQL syntax

```
REVOKE {SystemPrivilege [ , ... ] | ALL [PRIVILEGES]} FROM {user |PUBLIC} [,...]

REVOKE {{ObjectPrivilege [,...] | ALL [PRIVILEGES]} ON {[owner.object]} [,...]
FROM {user | PUBLIC}[,...]
```

Parameters

The following parameters are for revoking system privileges:

Parameter	Description
SystemPrivilege	See "System privileges" on page 6-1 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	See "Object privileges" on page 6-3 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.
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. See "Invalid materialized views" on page 5-76.

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
"The PUBLIC role" on page 6-5

ROLLBACK

Use the ROLLBACK statement to undo work done in the current transaction.

Required privilege

None

SQL syntax

```
ROLLBACK [WORK]
```

Parameters

The ROLLBACK statement allows 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 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.

SQL syntax

The general syntax for a SELECT statement is the following:

```
SELECT [FIRST NumRows | ROWS m TO n] [ALL | DISTINCT] SelectList
  FROM TableSpec [,...]
  [WHERE SearchCondition]
  [GROUP BY Expression [,...]]
  [HAVING SearchCondition]
  [ORDER BY {ColumnID|ColumnAlias|Expression} [ASC | DESC]]
  [,...]
  [FOR UPDATE [OF [[Owner.]TableName.]ColumnName [,...] ]
    [NOWAIT | WAIT Seconds] ]
```

The syntax for a SELECT statement that contains the set operators UNION, UNION ALL, MINUS, or INTERSECT is as follows:

```
SELECT [ROWS m TO n] [ALL] SelectList
  FROM TableSpec [,...]
    [WHERE SearchCondition]
    [GROUP BY Expression [,...]]
    [HAVING SearchCondition] [,...]
  {UNION [ALL] | MINUS | INTERSECT}
  SELECT [ROWS m TO n] [ALL] SelectList
    FROM TableSpec [,...]
     [WHERE SearchCondition]
     [GROUP BY Expression [,...]]
     [HAVING SearchCondition] [,...]
     [ORDER BY {ColumnID|ColumnAlias|Expression} [ASC | DESC]]
```

Parameters

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.

Parameter	Description
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.
SelectList	Specifies how the columns of the query result are to be derived. The syntax of select list is presented under "SelectList" on page 5-176.
FROM *TableSpec*	Identifies the tables referenced in the SELECT statement. The maximum number of tables per query is 24. *TableSpec* identifies a table from which rows are selected. The table can be a derived table, which is the result of a SELECT statement in the FROM clause. The syntax of *TableSpec* is presented under "TableSpec" on page 5-179.
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 Chapter 4, "Search Conditions" for more information on search conditions.
GROUP BY *Expression* [,...]	The GROUP BY clause identifies one or more expressions to be used for grouping when aggregate functions are specified in the select list and when you want to apply the function to groups of rows. The expression can be of various complexities. For example, it can designate single or multiple columns. It can include aggregate functions, arithmetic operations, the ROWID pseudocolumn, or NULL. It can also be a date or user function, a constant, or a dynamic parameter. When you use the GROUP BY clause, the select list can contain *only* aggregate functions and columns referenced in the GROUP BY clause. If the select list contains the construct *, *TableName*.*, or *Owner*.*TableName*.*, then the GROUP BY clause must contain all columns that the * includes. NULL values are considered equivalent in grouping rows. If all other columns are equal, all NULL values in a column are placed in a single group. If the GROUP BY clause is omitted, the entire query result is treated as one group.

Parameter	Description
HAVING	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.
(+)	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	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. For each column, 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. 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*]} [,...]

Parameter	Description
FOR UPDATE [OF [[*Owner*.] *TableName*.] *ColumnName* [,...]] [NOWAIT | WAIT *Seconds*]	FOR UPDATE ■ FOR UPDATE maintains a lock on an element (usually a row) until the end of the current transaction, regardless of isolation. All other transactions are excluded from performing any operation on that element until the transaction is committed or rolled back. ■ FOR UPDATE may be used with joins and the ORDER BY, GROUP BY, and DISTINCT clauses. Update locks are obtained on either tables or rows, depending on the table/row locking method chosen by the optimizer. ■ Rows from all tables that satisfy the WHERE clause are locked in UPDATE mode unless the FOR UPDATE OF clause is specified. This clause specifies which tables to lock. ■ If using row locks, all qualifying rows in all tables from the table list in the FROM clause are locked in update mode. Qualifying rows are those rows that satisfy the WHERE clause. 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*. ■ 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. (See "Basic names" on page 2-1.) 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" on page 5-179 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.

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.hiredate > '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:

- 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:

- tableb.column2 is greater than 5.
- tableb.column1 equals at least one tablea.column1 value.

If no WHERE clause is specified, all rows in both tables 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" on page 5-181.

This 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);
```

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 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.
```

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.
```

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.
```

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.
```

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.
[*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.
[[*Owner*.]*Table*.] *ColumnName*	Includes a particular column from the named owner's indicated table. You can also specify the CURRVAL or NEXTVAL column of a sequence.
[[*Owner*.]*Table*.] ROWID	Includes the ROWID 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 diagram.
- 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 option that operate on different columns in the same table. If the GROUP BY clause is not specified, the function is applied over all rows that satisfy the query. If the GROUP BY clause is specified, the function is applied once for each group defined by the GROUP BY clause. When you use aggregate functions with the GROUP BY clause, the select list can contain aggregate functions, arithmetic expressions, and columns in the GROUP BY clause.
  - A result column containing a fixed value can be created by specifying a constant or an expression involving only constants.
- In addition to specifying how the result columns are derived, the select list also controls their relative position from left to right in the query result. The first result column specified by the select list becomes the leftmost column in the query result, 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 if you want to refer to a column defined by an arithmetic expression or an aggregate.
- To join a table with itself, define multiple correlation names for the table in the FROM clause and use the correlation names in the select list and the WHERE clause to qualify columns from that table.
- When you use the GROUP BY clause, one answer is returned per group in accordance with the select list, as follows:
  - The WHERE clause eliminates rows before groups are formed.
  - The GROUP BY clause groups the resulting rows.
  - The select list aggregate functions are computed for each group.

Examples

In this example, one value, the average number of days you wait for a part, is returned by the statement:

```
SELECT AVG(deliverydays)
FROM purchasing.supplyprice;
```

The part number and delivery time for all parts that take fewer than 20 days to deliver are returned by the 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:

```
{[Owner.]TableName [CorrelationName] | JoinedTable | DerivedTable}
```

A simple table specification has the following syntax:

```
[Owner.]TableName
```

Parameters

The *TableSpec* parameter of the SELECT statement has the following parameters:

Parameter	Description
[*Owner*.]*TableName*	Identifies a table to be referenced.
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" on page 2-1. All correlation names within one statement must be unique.
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.
JoinedTable	Specifies the query that defines the table join. The syntax of *JoinedTable* is presented under "JoinedTable" on page 5-181.

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	For information on subqueries, see "Subqueries" on page 3-6.
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 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.

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.

Perform 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;
```

For more examples of joins specified with the (+) operator, see "Examples" on page 5-171.

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

TRUNCATE TABLE

The TRUNCATE TABLE statement is similar to a DELETE statement that deletes all rows. However, it is faster than DELETE in most circumstances, as DELETE removes each row individually.

Required privilege

No privilege is required for the table owner.

DELETE for another user's table.

SQL syntax

TRUNCATE TABLE [*Owner*.]*TableName*

Parameters

Parameter	Description
[*Owner*.]*TableName*	Identifies the table to be truncated.

Description

- TRUNCATE is a DDL statement and thus is controlled by the DDLCommitBehavior attribute. If DDLCommitBehavior=0 (the default), then a commit is performed before and after execution of the TRUNCATE statement. If DDLCommitBehavior=1, then TRUNCATE is part of a transaction and these transactional rules apply:
  - TRUNCATE operations can be rolled back.
  - Subsequent INSERT statements are not allowed in the same transaction as a TRUNCATE statement.
- Concurrent read committed read operations are allowed, and semantics of the reads are the same as for read committed reads in presence of DELETE statements
- TRUNCATE is allowed even when there are child tables. However, child tables need to be empty for TRUNCATE to proceed. If any of the child tables have any rows in them, TimesTen returns an error indicating that a child table is not empty.
- TRUNCATE is not supported with 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" on page 1-15.
- 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 deletes all rows from the cache group. The unload operation is local. It is not propagated across cache grid members.

Required privilege

No privilege is required for the cache group owner.

`UNLOAD CACHE GROUP` or `UNLOAD ANY CACHE GROUP` for another user's cache group.

SQL syntax

```
UNLOAD CACHE GROUP [Owner.]GroupName
[WHERE ConditionalExpression]
```

or

```
UNLOAD CACHE GROUP [Owner.]GroupName
WITH ID (ColumnValueList);
```

Parameters

Parameter	Description
`[Owner.]GroupName`	Name assigned to the cache group.
`ConditionalExpression`	A search condition to qualify the target rows of the operation.
`WITH ID ColumnValueList`	The `WITH ID` clauses allows 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.

Description

- This statement causes the entire content of the cache group to be deleted from the database.
- If the cache group is replicated, an `UNLOAD CACHE GROUP` statement deletes the entire content of any replica cache group as well.
- The `UNLOAD CACHE GROUP` statement can be used for any type of cache group. For a description of cache group types, see "User managed and system managed cache groups" on page 5-53.
- 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 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.
- Use the `WITH ID` clause to specify binding parameters.

Restrictions

- Do not reference child tables in the `WHERE` clause.
- Do not use the `WITH ID` clause on read-only cache groups or user managed cache groups with the autorefresh attribute unless the cache group is a dynamic cache group.

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);
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.

SQL syntax

```
UPDATE [FIRST NumRows]
{[Owner.]TableName [CorrelationName]}
SET {ColumnName =
{Expression1 | NULL | DEFAULT}} [,...]
[ WHERE SearchCondition ]
RETURNING|RETURN Expression2[,...] INTO DataItem[,...]
```

Parameters

Parameter	Description
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.
[*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 according to "Basic names" on page 2-1. 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.
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.

Parameter	Description
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 Chapter 3, "Expressions".
DataItem	Host variable or PL/SQL variable that stores the retrieved *Expression2* value.

Description

- 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.
- A table on which a unique constraint is defined cannot be updated to contain duplicate rows.
- The UPDATE operation fails if it violates any foreign key constraint. See "CREATE TABLE" on page 5-107 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 *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

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
    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

6

Privileges

The following sections describe 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, 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 6–1 System privileges

Privilege	Description
`ADMIN`	Allows a user to perform administrative tasks including checkpointing, backups, migration, and user creation and deletion.
`ALTER ANY CACHE GROUP`	Allows a user to alter any cache group in the database.
`ALTER ANY INDEX`	Allows a user to alter any index in the database. **Note**: There is no `ALTER INDEX` statement.
`ALTER ANY MATERIALIZED VIEW`	Allows a user to alter any materialized view in the database. **Note**: There is no `ALTER MATERIALIZED VIEW` statement.
`ALTER ANY PROCEDURE`	Allows a user to alter any PL/SQL procedure, function or package in the database.
`ALTER ANY SEQUENCE`	Allows a user to alter any sequence in the database. **Note**: There is no `ALTER SEQUENCE` statement.
`ALTER ANY TABLE`	Allows a user to alter any table in the database.
`ALTER ANY VIEW`	Allows a user to alter any view in the database. **Note**: There is no `ALTER VIEW` statement.
`CACHE_MANAGER`	Allows a user to perform operations related to cache groups.

Table 6–1 (Cont.) System privileges

Privilege	Description
CREATE ANY CACHE GROUP	Allows a user to create a cache group owned by any user in the database.
CREATE ANY INDEX	Allows a user to create an index on any table or materialized view in the database.
CREATE ANY MATERIALIZED VIEW	Allows a user to create a materialized view owned by any user in the database.
CREATE ANY PROCEDURE	Allows a user to create a PL/SQL procedure, function or package owned by any user in the database.
CREATE ANY SEQUENCE	Allows a user to create a sequence owned by any user in the database.
CREATE ANY SYNONYM	Allows a user to create a private synonym owned by any user in the database.
CREATE ANY TABLE	Allows a user to create a table owned by any user in the database.
CREATE ANY VIEW	Allows a user to create a view owned by any user in the database.
CREATE CACHE GROUP	Allows a user to create a cache group owned by that user.
CREATE MATERIALIZED VIEW	Allows a user to create a materialized view owned by that user.
CREATE PROCEDURE	Allows a user to create a PL/SQL procedure, function or package owned by that user.
CREATE PUBLIC SYNONYM	Allows a user to create a public synonym.
CREATE SEQUENCE	Allows a user to create a sequence owned by that user.
CREATE SESSION	Allows a user to create a connection to the database.
CREATE SYNONYM	Allows a user to create a private synonym.
CREATE TABLE	Allows a user to create a table owned by that user.
CREATE VIEW	Allows a user to create a view owned by that user.
DELETE ANY TABLE	Allows a user to delete from any table in the database.
DROP ANY CACHE GROUP	Allows a user to drop any cache group in the database.
DROP ANY INDEX	Allows a user to drop any index in the database.
DROP ANY MATERIALIZED VIEW	Allows a user to drop any materialized view in the database.
DROP ANY PROCEDURE	Allows a user to drop any PL/SQL procedure, function or package in the database.
DROP ANY SEQUENCE	Allows a user to drop any sequence in the database.
DROP ANY SYNONYM	Allows a user to drop a synonym owned by any user in the database.
DROP ANY TABLE	Allows a user to drop any table in the database.
DROP ANY VIEW	Allows a user to drop any view in the database.
DROP PUBLIC SYNONYM	Allows a user to drop a public synonym.
EXECUTE ANY PROCEDURE	Allows a user to execute any PL/SQL procedure, function or package in the database.
FLUSH ANY CACHE GROUP	Allows a user to flush any cache group in the database.

Table 6–1 (Cont.) System privileges

Privilege	Description
INSERT ANY TABLE	Allows a user to insert into any table in the database.
LOAD ANY CACHE GROUP	Allows a user to load any cache group in the database.
REFRESH ANY CACHE GROUP	Allows a user to flush any cache group in the database.
SELECT ANY SEQUENCE	Allows a user to select from any sequence in the database.
SELECT ANY TABLE	Allows a user to select from any table, view or materialized view in the database.
UNLOAD ANY CACHE GROUP	Allows a user to unload any cache group in the database.
UPDATE ANY TABLE	Allows a user to update any table in the database.
XLA	Allows 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, 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 6–2 Object privileges

Privilege	Object type	Description
DELETE	Table	Allows a user to delete from a table.
EXECUTE	PL/SQL package, procedure or function	Allows a user to execute a PL/SQL package, procedure or function directly.
FLUSH	Cache group	Allows a user to flush a cache group.
INDEX	Table or materialized view	Allows a user to create an index on a table or materialized view.
INSERT	Table	Allows a user to insert into a table.
LOAD	Cache group	Allows a user to load a cache group
REFERENCES	Table or materialized view	Allows 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	Allows a user to refresh a cache group

Table 6–2 (Cont.) Object privileges

Privilege	Object type	Description
SELECT	Table, sequence, view or materialized view	Allows a user to select from a table, sequence, view or materialized view. The SELECT privilege allows a user to perform all operations on a sequence. A user can be granted the SELECT privilege on a view without having the SELECT privilege on its detail table.
UNLOAD	Cache group	Allows a user to unload a cache group
UPDATE	Table	Allows 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 6–3 shows the privilege hierarchy.

Table 6–3 Privilege hierarchy

Privilege	Confers these privileges
ADMIN	All other privileges including CACHE_MANAGER
CREATE ANY INDEX	INDEX ON (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 and to perform cache grid operations. The built-in procedures and utilities for these operations are documented in *Oracle TimesTen In-Memory Database Reference*.

`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';
```

7

Reserved Words

TimesTen reserves words for use in SQL statements.

To use one of these words as an identifier (such as a table name or column name), enclose the reserved word in quotes. Otherwise, syntax errors may occur.

Reserved words
AGING
ALL
ANY
AS
BETWEEN
BINARY_DOUBLE_INFINITY
BINARY_DOUBLE_NAN
BINARY_FLOAT_INFINITY
BINARY_FLOAT_NAN
CASE
CHAR
COLUMN
CONNECTION
CONSTRAINT
CROSS
CURRENT_SCHEMA
CURRENT_USER
CURSOR
DATASTORE_OWNER
DATE
DECIMAL
DEFAULT
DESTROY
DISTINCT
FIRST

Reserved words
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

Reserved words
UID
UNION
UNIQUE
UPDATE
USER
USING
VARCHAR
WHEN
WHERE

Index

+ , *See* addition
/ , *See* dividing

Symbols

%
  in LIKE pattern strings, 4-21
& operator, 3-4
* , *See* multiplying
+ operator
  WHERE clause, 5-167
^ operator, 3-4
_
  in LIKE pattern string, 4-21
| operator, 3-4
|| operator, 3-4
~ operator, 3-4

A

ABS function, 3-26
ADD column, 5-34
ADD ELEMENT
  replication, 5-15
ADD SUBSCRIBER
  replication, 5-15
ADD_MONTHS function, 3-27
addition, 3-3
ADMIN system privilege
  definition, 6-1
aggregate functions
  ALL, 3-8
  and overflow, 1-32
  AVG, 3-8
  COUNT *, 3-8
  COUNT ColumnName, 3-8
  DISTINCT, 3-8
  in expressions, 3-3
  in query, 5-177
  MAX, 3-8
  MIN, 3-8
  over empty, ungrouped table, 3-8
  SQL syntax, 3-8
  SUM, 3-8
aging
  cache groups, 5-62
AGING reserved word, 7-1
ALL
  defined, 3-8
  in SELECT statements, 5-167
ALL/ NOT IN predicate (subquery), 4-4
ALL reserved word, 7-1
ALL/NOT IN predicate (value list), 4-6
ALTER ACTIVE STANDBY PAIR statement, 5-2
ALTER ANY CACHE GROUP system privilege
  definition, 6-1
ALTER ANY INDEX system privilege
  definition, 6-1
ALTER ANY MATERIALIZED VIEW system privilege
  definition, 6-1
ALTER ANY PROCEDURE system privilege
  definition, 6-1
ALTER ANY SEQUENCE system privilege
  definition, 6-1
ALTER ANY TABLE system privilege
  definition, 6-1
ALTER ANY VIEW system privilege
  definition, 6-1
ALTER CACHE GROUP statement, 5-6
  AUTOREFRESH cache groups, 5-6
  AUTOREFRESH state, 5-62
  READONLY cache groups, 5-6
ALTER ELEMENT clause
  DROP MASTER, 5-21
  DROP SUBSCRIBER, 5-20
  replication, 5-16
ALTER FUNCTION statement, 5-8
ALTER PACKAGE statement, 5-10
ALTER PROCEDURE statement, 5-12
ALTER REPLICATION statement, 5-14
ALTER SESSION statement, 5-23
ALTER SUBSCRIBER clause, 5-16
ALTER TABLE statement
  ADD column, 5-34
  defined, 5-29
  DROP column, 5-35
  PRIMARY KEY, 5-35
  table names, 5-31
ALTER USER statement, 5-42
altering
  active standby pair, 5-2

replication, 5-14
session parameters, 5-23
tables, 5-29
ANSI SQL data types, 1-5
ANY/ IN predicate (subquery), 4-8
ANY/ IN predicate (value list), 4-10
ANY predicate
defined, 4-8
example, 4-8
operators, 4-8
SQL syntax, 4-8
ANY reserved word, 7-1
approximate data types, 1-15
arithmetic operations
and overflow, 1-32
arithmetic operators
in expressions, 3-4
AS reserved word, 7-1
ASC | DESC clause of CREATE INDEX statement
defined, 5-69
ASCII characters, 3-12
ASCIISTR function, 3-29
asynchronous materialized view
creating, 5-73
attributes
altering, 5-2
AUTOREFRESH clause
ALTER CACHE GROUP statement, 5-62
FULL, 5-61
in cache groups, 5-61
INCREMENTAL, 5-61
INTERVAL, 5-7
STATE, 5-7
AVG function
defined, 3-8

B

basic names
definition, 2-1
objects having, 2-1
rules, 2-1
BETWEEN predicate
defined, 4-13
in search conditions, 4-2
SQL syntax, 4-13
BETWEEN reserved word, 7-1
BIGINT data type, 1-33
BINARY data type, 1-1, 1-20, 1-26, 1-33
BINARY_DOUBLE data type, 1-1, 1-20, 1-26, 1-33
BINARY_DOUBLE_INFINITY reserved word, 7-1
BINARY_DOUBLE_NAN reserved word, 7-1
BINARY_FLOAT data type, 1-2, 1-20, 1-26, 1-33
BINARY_FLOAT_INFINITY reserved word, 7-1
BINARY_FLOAT_NAN, 7-1
bitwise AND operator, 3-4
bitwise NOT operator, 3-4
bitwise OR operator, 3-4
bucket count, 5-113

C

cache grid, 5-63
cache group
definition, 5-53
cache group instance
definition, 5-53
cache groups
aging, 5-62
ALTER CACHE GROUP statement, 5-6
CREATE CACHE GROUP statement, 5-53
DROP CACHE GROUP statement, 5-130
dynamic, 5-54
explicitly loaded, 5-54
FLUSH CACHE GROUP statement, 5-143
global, 5-54
LOAD CACHE GROUP statement, 5-151
local, 5-54
restrictions, 5-60
system managed, 5-53
UNLOAD CACHE GROUP statement, 5-186
user managed, 5-53
user manager, 5-53
CACHE_MANAGER privilege hierarchy, 6-4
CACHE_MANAGER system privilege
definition, 6-1
CALL statement, 5-44
CASE function, 3-30
CASE reserved word, 7-1
CAST function, 3-32
CEIL function, 3-34
CHAR data type, 1-2, 1-11, 1-26, 1-33
CHAR reserved word, 7-1
CHAR VARYING data type, 1-6
CHARACTER
values in constants, 3-11
character data
and truncation, 1-32
character data types, 1-11
character string, 3-11
CHARACTER VARYING data type, 1-6
CHECK CONFLICTS clause
replication, 5-94
syntax, 5-94
CHR function, 3-33
COALESCE function, 3-35
column alias
in SELECT statement, 5-168, 5-176
column definition, 5-112
column name
in expressions, 3-3
column names
in INSERT SELECT statements, 5-150
in NULL predicates, 4-20
column reference
in SELECT statements, 5-168
syntax, 5-168
COLUMN reserved word, 7-1
columns, 5-112
defining, 5-108
in tables, 5-108

COMMIT statement, 5-46
comparing data types in search conditions, 4-3
comparison predicate
  example, 4-15
  in search conditions, 4-2
  operators, 4-14
  SQL syntax, 4-4, 4-8, 4-14
compound identifiers, 2-2
CONCAT function, 3-36
concatenate operator, 3-4
conflict resolution
  check conflicts, 5-16
  replication, 5-94
CONNECTION reserved word, 7-1
constants
  character strings, 3-11
  CHARACTER values, 3-11
  DATE values, 3-12, 3-13
  defined, 3-11
  fixed point values, 3-11
  FLOAT values, 3-11
  HEXIDECIMAL values, 3-12
  in expressions, 3-3
  in NULL predicates, 4-20
  INTEGER values, 3-11
  SQL syntax, 3-11
  TIME values, 3-13
  TIMESTAMP values, 3-14
CONSTRAINT reserved word, 7-1
constraints, defining, 5-107
correlation name
  definition, 5-126
  in SELECT statement, 5-179
correlation names
  in SELECT statements, 5-177
COUNT * function
  defined, 3-8
COUNT ColumnName function
  defined, 3-8
CREATE ACTIVE STANDBY PAIR statement, 5-47
CREATE ANY CACHE GROUP system privilege
  definition, 6-2
CREATE ANY INDEX system privilege
  definition, 6-2
CREATE ANY MATERIALIZED VIEW system privilege
  definition, 6-2
CREATE ANY PROCEDURE system privilege
  definition, 6-2
CREATE ANY SEQUENCE system privilege
  definition, 6-2
CREATE ANY SYNONYM system privilege
  definition, 6-2
CREATE ANY TABLE system privilege
  definition, 6-2
CREATE ANY VIEW system privilege
  definition, 6-2
CREATE CACHE GROUP system privilege
  definition, 6-2
CREATE FUNCTION, 5-66
  defined, 5-66
CREATE GLOBAL TEMPORARY TABLE, 5-107, 5-108
CREATE INDEX statement
  defined, 5-69
  example, 5-72
  index name, 5-69
  table names, 5-69
  tables without rows, 5-70
  UNIQUE clause, 5-69
CREATE MATERIALIZED VIEW
  defined, 5-73
CREATE MATERIALIZED VIEW LOG statement, 5-79
CREATE MATERIALIZED VIEW system privilege
  definition, 6-2
CREATE PACKAGE
  defined, 5-81
CREATE PACKAGE BODY, 5-83
CREATE PACKAGE BODY statement
  defined, 5-83
CREATE PACKAGE statement, 5-81
CREATE PROCEDURE, 5-84
  defined, 5-84
CREATE PROCEDURE system privilege
  definition, 6-2
CREATE PUBLIC SYNONYM system privilege
  definition, 6-2
CREATE REPLICATION statement, 5-87
CREATE SEQUENCE statement, 5-101
  defined, 5-101
CREATE SEQUENCE system privilege
  definition, 6-2
CREATE SESSION system privilege
  definition, 6-2
CREATE SYNONYM statement, 5-104
CREATE SYNONYM system privilege
  definition, 6-2
CREATE TABLE statement
  defined, 5-107
  examples, 5-116
  FOREIGN KEY, 5-109
  hash column name, 5-109
  maximum columns, 5-108, 5-112
  maximum page number, 5-109
  PRIMARY KEY, 5-108
CREATE TABLE system privilege
  definition, 6-2
CREATE USER statement, 5-122
CREATE VIEW statement, 5-124
CREATE VIEW system privilege
  definition, 6-2
creating, 5-172
  active standby pairs, 5-47
  cache groups, 5-53
  constraints, 5-107
  functions, 5-66
  indexes, 5-69
  materialized views, 5-73

procedures, 5-84
sequences, 5-101
tables, 5-107
users, 5-122
views, 5-124
CROSS reserved word, 7-1
CURRENT_SCHEMA, 7-1
CURRENT_USER function, 3-89
CURRENT_USER reserved word, 7-1
CURSOR reserved word, 7-1

D

d (ODBC-date-literal syntax), 3-13
DATA data type
using, 1-23
Data Definition Language (DDL), 5-1
Data Manipulation Language (DML), 5-1
data overflow, 1-31
data truncation, 1-31
data types
ANSI SQL, 1-5
approximate types, 1-15
backward compatibility support, 1-7
character types, 1-11
comparison rules, 1-27
conversion, 1-28
effect of, 1-1
exact and approximate, 1-15
exact types, 1-15
modes, 1-1
specifications, 1-1
storage requirements, 1-26
TimesTen/Oracle compatibility, 1-7, 1-9
TIMEZONE unsupported, 1-24
DATASTORE clause
in CREATE REPLICATION statement, 5-89
DATASTORE_OWNER reserved word, 7-1
DATE and TIME data types
using, 1-23
DATE data type, 1-2, 1-22, 1-23, 1-26, 1-33
DATE literal
ODBC date-literal syntax, 3-13
values in constants, 3-12, 3-13
date literal
defined, 3-13
DATE reserved word, 7-1
date string
defined, 3-12
datetime and interval types
arithmetic operations, 1-24
datetime data types, 1-22
using, 1-23
datetime format model
for TO_CHAR of TT_TIMESTAMP and TT_DATE, 3-23
datetime format models, 3-20
DECIMAL data type, 1-7
DECIMAL reserved word, 7-1
DECODE function, 3-38
DEFAULT column value, 5-112, 5-148
DEFAULT reserved word, 7-1
defining, 5-112
DELETE ANY TABLE system privilege
definition, 6-2
DELETE object privilege
definition, 6-3
DELETE statement
and DROP TABLE statement, 5-126
defined, 5-126
search conditions, 5-126
deleting
indexes, 5-141
rows, 5-126
tables, 5-141
derived table, 5-180
derived tables, 5-172, 5-180
described, 5-180
in SELECT statement, 5-180
in TableSpec, 5-167
restrictions, 5-75
DESTROY reserved word, 7-1
detail tables, 5-74, 5-115, 5-184
and ON DELETE CASCADE, 5-36
in materialized views, 5-74
restrictions, 5-156
VIEWS, 5-124
DISTINCT modifier
and subqueries, 3-6
defined, 3-8
in SELECT statement, 5-167
DISTINCT reserved word, 7-1
dividing, 3-3
dividing expressions, 3-3
DOUBLE PRECISION data type, 1-6
DROP ACTIVE STANDBY PAIR statement, 5-129
DROP ANY CACHE GROUP system privilege
definition, 6-2
DROP ANY INDEX system privilege
definition, 6-2
DROP ANY MATERIALIZED VIEW system privilege
definition, 6-2
DROP ANY PROCEDURE system privilege
definition, 6-2
DROP ANY SEQUENCE system privilege
definition, 6-2
DROP ANY SYNONYM system privilege
definition, 6-2
DROP ANY TABLE system privilege
definition, 6-2
DROP ANY VIEW system privilege
definition, 6-2
DROP CACHE GROUP statement, 5-130
DROP column, 5-35
DROP ELEMENT clause
replication, 5-17
DROP FUNCTION statement, 5-131
DROP INDEX statement, 5-132
DROP MATERIALIZED VIEW LOG statement, 5-135

DROP MATERIALIZED VIEW statement, 5-134
DROP PACKAGE statement, 5-136
DROP PROCEDURE statement, 5-137
DROP PUBLIC SYNONYM system privilege
  definition, 6-2
DROP REPLICATION statement, 5-138
DROP SEQUENCE statement, 5-139
DROP SYNONYM statement, 5-140
DROP TABLE statement, 5-141
DROP USER statement, 5-142
DROP VIEW statement, 5-134
dropping
  active standby pairs, 5-129
  cache groups, 5-130
  functions, 5-131
  indexes, 5-132, 5-141
  procedures, 5-136, 5-137
  replication schemes, 5-138
  sequences, 5-139
  tables, 5-141
  views, 5-134
duplicate parameters, 2-3
DURABLE clause
  in CREATE REPLICATION statement, 5-92
dynamic cache groups, 5-54
dynamic parameters
  example, 3-5
  in expressions, 3-3, 3-5
  in LIKE predicate, 4-21
  in single row inserts, 5-147
  names, 2-2
  naming rules, 2-2

E

ELEMENT clause
  in CREATE REPLICATION statement, 5-89
escape character
  in LIKE predicate, 4-21
escaped Unicode characters, 3-12
exact data types, 1-15
exclusive OR operator, 3-4
EXECUTE ANY PROCEDURE system privilege
  definition, 6-2
EXECUTE object privilege
  definition, 6-3
EXISTS predicate, 4-16
  defined, 4-16
  in search conditions, 4-2
  SQL syntax, 4-16
expressions
  arithmetic operators in, 3-4
  bitwise AND operator, 3-4
  bitwise NOT operator, 3-4
  bitwise OR operator, 3-4
  concatenate operators, 3-4
  exclusive OR operator, 3-4
  in aggregate functions, 3-8
  in BETWEEN predicates, 4-13
  in comparison predicate, 4-4, 4-8, 4-14
  in IS INFINITE predicate, 4-18
  in LIKE predicates, 4-21
  in NAN predicates, 4-19
  in NULL predicates, 4-20
  in UPDATE statements, 5-188
  ROWID, 3-1
  ROWNUM, 3-1
  specification, 3-3
  SQL syntax, 3-3
EXTRACT function, 3-39

F

FAILTHRESHOLD clause
  in ALTER REPLICATION statement, 5-17
  in CREATE ACTIVE STANDBY PAIR statement, 5-49
  in CREATE REPLICATION statement, 5-90
FIRST reserved word, 7-1
fixed point value
  constants, 3-11
  defined, 3-11
FLOAT and FLOAT (n) data types, 1-20
FLOAT data type, 1-6
FLOAT reserved word, 7-2
float value
  defined, 3-11
FLOAT values
  in constants, 3-11
floating-point numbers, 1-19
FLOOR function, 3-40
FLUSH ANY CACHE GROUP system privilege
  definition, 6-2
FLUSH CACHE GROUP statement, 5-143
FLUSH object privilege
  definition, 6-3
FOR reserved word, 7-2
FOREIGN KEY option
  in CREATE TABLE statement, 5-109
FOREIGN reserved word, 7-2
format model
  for ROUND and TRUNC date functions, 3-23
  for TO_CHAR of TimesTen types, 3-23
format model for TO_CHAR of TimesTen types, 3-23
format models, 3-16
FROM reserved word, 7-2
fully qualified name, 2-2
functions
  creating, 5-66

G

GETDATE function, 3-74
global cache groups, 5-54
GLOBAL TEMPORARY TABLE, 5-107, 5-108
global temporary table
  object privilege, 5-108
GRANT, 5-145
GRANT statement, 5-145

GREATEST, 3-41
GROUP, 7-2
GROUP BY
in aggregate functions, 3-8
in SELECT statements, 5-167

H

hash index
examples, 5-116
for table, 5-109
materialized view, 5-74
hash indexes
pages, 5-109
HashColumnName option
in CREATE TABLE statement, 5-109
HAVING clause
in SELECT statements, 5-168
HAVING reserved word, 7-2
HEXIDECIMAL
values in constants, 3-12
hexidecimal string
defined, 3-12

I

IN predicate
in search conditions, 4-2
index names
in CREATE INDEX, 5-69
in DROP INDEX, 5-132
INDEX object privilege
definition, 6-3
index owner (not specified), 5-132
indexes
creating, 5-69
dropping, 5-141
owner not specified, 5-132
INF and NAN data types, 1-30
INLINE (NOT INLINE), 5-112
INLINE (NOT INLINE) clause
in ALTER TABLE statement, 5-31
INNER reserved word, 7-2
INSERT ANY TABLE system privilege
definition, 6-3
INSERT object privilege
definition, 6-3
INSERT SELECT statement, 5-150
omitted columns, 5-150
rows with defined values, 5-150
INSERT statement, 5-147
INSTR function, 3-70
INSTR4 function, 3-70
INSTRB function, 3-70
INTEGER data type, 1-6, 1-34
INTEGER reserved word, 7-2
integer value
defined, 3-11
INTEGER values
in constants, 3-11
INTERSECT reserved word, 7-2
INTERVAL, 7-2
INTERVAL data type, 1-2, 1-26, 1-34
using, 1-23
interval literal, 3-14
INTO reserved word, 7-2
IS INFINITE predicate, 4-18
IS NAN predicate, 4-19
IS NULL predicate, 4-20
defined, 4-20
SQL syntax, 4-18, 4-19, 4-20
IS reserved word, 7-2

J

join conditions
+ operator, 5-167
JOIN reserved word, 7-2
join types
INNER, 5-181
LEFT, 5-181
RIGHT, 5-181
joined tables, 5-181
joins
joining table to itself, 5-177
outer, 5-167

L

LEAST function, 3-43
LEFT reserved word, 7-2
LENGTH function, 3-71
LENGTH4 function, 3-71
LENGTHB function, 3-71
LIKE predicate
defined, 4-21
in search conditions, 4-2
pattern matching of NCHAR and NVARCHAR strings, 4-24
SQL syntax, 4-21
LIKE reserved word, 7-2
LOAD ANY CACHE GROUP system privilege
definition, 6-3
LOAD CACHE GROUP statement, 5-151
LOAD object privilege
definition, 6-3
local cache groups, 5-54
logical operators
in search conditions, 4-2
LONG reserved word, 7-2
lower case letters in names, 2-1
LOWER function, 3-45
LPAD function, 3-46
LTRIM function, 3-48

M

MASTER clause
in ALTER REPLICATION statement, 5-18
in CREATE ACTIVE STANDBY PAIR statement, 5-50

in CREATE REPLICATION statement, 5-90
materialized view log, 5-79
materialized views
invalid, 5-76
revoking privileges on detail table, 5-164
revoking privileges on the detail table, 5-76
MAX function
defined, 3-8
maximum
columns in CREATE TABLE, 5-108, 5-112
items for DISTINCT option, 5-167
table cardinality, 5-113
tables per query, 5-167
MERGE statement, 5-155
MIN function
defined, 3-8
MINUS, 7-2
MOD function, 3-50
multiplying, 3-3
multiplying expressions, 3-3
MVLOG$_ID
materialized view log, 5-79

N

names
basic names, 2-1
compound identifiers, 2-2
dynamic parameters, 2-2
fully qualified, 2-2
lower case letters, 2-1
owner names, 2-1
simple names, 2-2
used in TimesTen, 2-1
user ID, 2-1
naming dynamic parameters, 2-2
naming rules, 2-1
NATIONAL CHAR data type, 1-6
NATIONAL CHAR VARYING data type, 1-6
NATIONAL CHARACTER data type, 1-6
national character string
constant, 3-12
NCHAR data type, 1-3, 1-12, 1-26, 1-34
defined, 4-24
example, 4-24
NCHAR VARYING data type, 1-6
NCHR function, 3-51
NLSSORT function, 3-52
NO RETURN clause
in CREATE REPLICATION statement, 5-90
NONDURABLE clause
in CREATE REPLICATION statement, 5-92
NOT INLINE (INLINE), 5-112
NOT INLINE (INLINE) clause
in ALTER TABLE statement, 5-31
NOT NULL clause
in CREATE TABLE statement, 5-113
in INSERT SELECT statement, 5-150
NULL predicate
in search conditions, 4-2
NULL reserved word, 7-2
NULL storage, 1-13, 1-27
NULL values
defined, 1-29
in comparison predicates, 4-15
in INSERT statement, 5-148
in search conditions, 4-3
in UPDATE statements, 5-188
sort order in CREATE INDEX, 5-70
sorting, 1-29
SQLBindCol ODBC function, 1-29
SQLBindParameter ODBC function, 1-29
NUMBER data type, 1-3, 1-15
NUMBER data type in TimesTen Mode, 1-36
number format models, 3-17
NUMERIC data type, 1-7
numeric data type truncation, 1-32
numeric data types, 1-15
numeric precedence, 1-21
NUMTODSINTERVAL function, 3-54
NUMTOYMINTERVAL function, 3-55
NVARCHAR2 data type, 1-3, 1-14, 1-26
defined, 4-24
example, 4-24
NVL function, 3-56

O

object privilege, 6-3
ON reserved word, 7-2
operators
comparison, 4-14
in WHERE clause of SELECT statement, 5-167
ORA_CHAR data type, 1-36
ORA_DATE data type, 1-36
ORA_NCHAR data type, 1-36
ORA_NVARCHAR2 data type, 1-37
ORA_SYSDATE reserved word, 7-2
ORA_TIMESTAMP data type, 1-37
ORA_VARCHAR2 data type, 1-37
Oracle data types supported in TimesTen type mode, 1-36
ORDER BY clause
and subqueries, 3-6
in SELECT statement, 5-168
specifying result columns, 5-177
ORDER reserved word, 7-2
OUTER JOIN clause
specifying in SELECT statement, 5-182
outer joins
conditions, 5-167
indicators, 5-167
overflow
during type conversion, 1-32
in aggregate functions, 1-32
in arithmetic operations, 1-32
of data, 1-31
owner names, 2-1
owners of index, 5-132

P

package body
  creating, 5-83
packages
  CREATE PACKAGE BODY statement, 5-83
  CREATE PACKAGE statement, 5-81
  creating, 5-81
parameters
  duplicate, 2-3
  inferring data type, 2-3
pattern matching in LIKE predicate, 4-21
PORT clause
  in CREATE ACTIVE STANDBY PAIR statement, 5-50
  in CREATE REPLICATION statement, 5-90
POWER function, 3-57
predicates
  ANY, 4-8
  BETWEEN, 4-13
  comparison, 4-14
  compatible data types, 4-3
  EXISTS, 4-16
  IS NULL, 4-20
  LIKE, 4-21
  null values, 4-3
  order of evaluation, 4-3
primary
  definition, 3-3
  in expressions, 3-3
PRIMARY KEY clause
  in CREATE TABLE statement, 5-108
PRIMARY reserved word, 7-2
privilege
  object, 6-3
  system, 6-1
privilege hierarchy, 6-4
procedures
  creating, 5-84
PROPAGATE reserved word, 7-2
PROPAGATOR clause
  in ALTER REPLICATION statement, 5-18
PUBLIC, 5-145
PUBLIC reserved word, 7-2
PUBLIC role
  privileges, 6-5
PUBLIC user
  in GRANT statement, 5-145

Q

quantified predicate
  in search conditions, 4-2
queries
  and aggregate functions, 5-177
  results, 5-166
  syntax, 5-166

R

READONLY reserved word, 7-2
REAL data type, 1-7
REFERENCES object privilege
  definition, 6-3
REFRESH ANY CACHE GROUP system privilege
  definition, 6-3
REFRESH CACHE GROUP statement, 5-159
REFRESH MATERIALIZED VIEW statement, 5-162
REFRESH object privilege
  definition, 6-3
refreshing a cache group, 5-7
replication, 5-91
  ADD ELEMENT, 5-15
  ADD SUBSCRIBER, 5-15
  ALTER ELEMENT, 5-16
  ALTER SUBSCRIBER, 5-16
  CHECK CONFLICTS, 5-94
  conflict resolution, 5-94, 5-95
  DATASTORE ELEMENT, 5-89
  DROP ELEMENT, 5-17
  ELEMENT, 5-89
  FAILTHRESHOLD, 5-17, 5-49, 5-90
  MASTER, 5-18, 5-50, 5-90, 5-92
  NO RETURN, 5-90
  PORT, 5-50, 5-90
  PROPAGATOR, 5-18
  restrictions, 5-95
  RETURN RECEIPT, 5-19, 5-91
  SUBSCRIBER, 5-19, 5-50, 5-90, 5-92
  TIMEOUT, 5-20, 5-92
  TIMESTAMP, 5-94, 5-95
  TRANSMIT, 5-92
replication element, 5-87
replication scheme, 5-87
reserved words, 7-1
restrictions, 5-74, 5-115, 5-180, 5-184
result columns in SELECT statement, 5-177
RETURN RECEIPT BY REQUEST clause
  in CREATE REPLICATION statement, 5-91
RETURN RECEIPT clause
  in ALTER REPLICATION statement, 5-16, 5-19
  in CREATE REPLICATION statement, 5-91
RETURN TWOSAFE BY REQUEST clause
  in ALTER REPLICATION statement, 5-19
RETURN TWOSAFE clause
  in ALTER REPLICATION statement, 5-19
  in CREATE REPLICATION statement, 5-91
REVOKE, 5-163
REVOKE statement, 5-163
revoking privileges
  materialized views, 5-164
RIGHT reserved word, 7-2
ROLLBACK statement, 5-165
ROUND (date) function, 3-58
ROUND (expression) function, 3-59
rowid, 1-21, 3-1
ROWID data type, 1-3
  description, 1-21
  explicit conversion, 1-22
  implicit conversion, 1-22
  in expressions, 1-21

in INSERT SELECT statement, 1-22
ROWID pseudocolumn
in expressions, 3-3
ROWNUM pseudocolumn, 3-1
ROWNUM reserved word, 7-2
rows
inserting, 5-147
retrieving, 5-166
selecting, 5-166
ROWS reserved word, 7-2
RPAD function, 3-61
RTRIM function, 3-63

S

search condition
compatible predicates, 4-3
general syntax, 4-2
logical operators in, 4-2
SQL syntax, 4-2
type conversion, 4-3
value extensions, 4-3
SELECT
select list, 5-167
SELECT ANY SEQUENCE system privilege
definition, 6-3
SELECT ANY TABLE system privilege
definition, 6-3
select list
defined, 5-167
SQL syntax, 5-176
SELECT object privilege
definition, 6-4
SELECT reserved word, 7-2
SELECT statement, 5-166
GROUP BY clause, 5-167
HAVING clause, 5-168
maximum tables per query, 5-167
ORDER BY clause, 5-168
unique rows, 5-167
WHERE clause, 5-167
SELF reserved word, 7-2
SESSION_USER function, 3-91
SESSION_USER reserved word, 7-2
SET clause
in ALTER ACTIVE STANDBY PAIR statement, 5-3
in ALTER CACHE GROUP statement, 5-6
in ALTER REPLICATION statement, 5-14
SET PAGES, 5-30
SET reserved word, 7-2
SIGN function, 3-65
simple names, 2-2
SMALLINT data type, 1-7, 1-34
SMALLINT reserved word, 7-2
SOME reserved word, 7-2
sorting of NULL values, 1-29
SQL naming rules, 2-1
SQL statements
ALTER REPLICATION, 5-14
ALTER SESSION, 5-23
ALTER TABLE, 5-29
CREATE CACHE GROUP, 5-53
CREATE FUNCTION, 5-66
CREATE INDEX, 5-69
CREATE MATERIALIZED VIEW, 5-73
CREATE PACKAGE, 5-81
CREATE PACKAGE BODY, 5-83
CREATE PROCEDURE, 5-84
CREATE SEQUENCE, 5-101
CREATE TABLE, 5-107
CREATE VIEW, 5-124
DELETE, 5-126
DROP CACHE GROUP, 5-130
DROP FUNCTION, 5-131
DROP PROCEDURE, 5-137
DROP REPLICATION, 5-138
DROP SEQUENCE, 5-139
DROP TABLE, 5-141
FLUSH CACHE GROUP, 5-143
INSERT, 5-147
INSERT SELECT, 5-150
LOAD CACHE GROUP, 5-151
SELECT, 5-166
UNLOAD CACHE GROUP, 5-186
UPDATE, 5-188
SQL syntax
CREATE PACKAGE, 5-81
CREATE PACKAGE BODY, 5-83
CREATE PROCEDURE, 5-84
SQLBindCol ODBC function
and NULL values, 1-29
SQLBindParameter ODBC function
and NULL values, 1-29
SQRT function, 3-67
storage requirements, 1-26
string functions, 3-68
strings
in constants, 3-11
truncated in UPDATE statement, 5-189
subqueries, 3-6
subquery
in EXISTS predicates, 4-16
SUBSCRIBER clause
in ALTER REPLICATION statement, 5-19
in CREATE REPLICATION statement, 5-92
SUBSTR function, 3-69
SUBSTR4 function, 3-69
SUBSTRB function, 3-69
subtraction operator
in expressions, 3-3
SUM function, 3-8
SYS_CONTEXT function, 3-72
SYSDATE, 7-2
SYSDATE function, 3-74
system managed cache group, 5-53
system privilege, 6-1
system tables, 2-1
SYSTEM_USER function, 3-92
SYSTEM_USER reserved word, 7-2

T

table names
  in ALTER TABLE statement, 5-31
  in CREATE INDEX statement, 5-69
  in CREATE TABLE statement, 5-108
  in DROP INDEX statement, 5-132
  in INSERT SELECT statement, 5-150
table owner
  not specified, 5-132
tables
  creating, 5-107
  derived, 5-180
  dropping, 5-141
  inserting rows, 5-147
  maximum cardinality, 5-113
  maximum per query, 5-167
  owner not specified, 5-132
  specifying in SELECT statement, 5-179
  unique constraints, 5-189
temporary table, 5-107, 5-108
  object privilege, 5-108
TIME
  ODBC-time-literal syntax, 3-13
  values in constants, 3-13
TIME data type, 1-3, 1-22, 1-23, 1-34
time literal
  defined, 3-13
  in constants, 3-13
time string
  in constants, 3-13
TIMEOUT clause
  in ALTER REPLICATION statement, 5-20
  in CREATE REPLICATION, 5-92
TIMESTAMP
  in CHECK CONFLICTS clause, 5-94
  ODBC literal, 3-14
  replication, 5-94, 5-95
  values in constants, 3-14
TIMESTAMP data type, 1-4, 1-23, 1-27, 1-34
timestamp literal
  defined, 3-14
  in constants, 3-14
timestamp ODBC literal, 3-14
timestamp string
  in constants, 3-14
TimesTen data type mapping, 1-9
TimesTen interval, 1-23
TimesTen type mode, 1-32
TIMEZONE data type
  conversions, 1-24
TINYINT data type, 1-34
TO reserved word, 7-2
TO_CHAR function, 3-76
  SQL syntax, 3-76, 3-79
TO_DATE function, 3-78
  SQL syntax, 3-78
TO_NUMBER function, 3-79
TRANSMIT clause
  replication, 5-92
TRANSMIT DURABLE/NONDURABLE clause
  in CREATE REPLICATION statement, 5-92
TRIM function, 3-80
TRUNC (date), 3-83
TRUNC (date) function, 3-83
TRUNC (expression) function, 3-84
TRUNCATE TABLE statement, 5-184
truncation
  and numeric data, 1-32
  in character data, 1-32
  of data, 1-31
TT_BIGINT data type, 1-17, 1-27
TT_CHAR data type, 1-8
TT_DATE data type, 1-4, 1-23, 1-27
TT_DECIMAL data type, 1-8, 1-27
TT_HASH function, 3-85
TT_INTEGER data type, 1-4, 1-18, 1-27, 1-34
TT_NCHAR data type, 1-9
TT_NVARCHAR data type, 1-9, 1-35
TT_SMALLINT data type, 1-19, 1-27
TT_SYSDATE reserved word, 7-2
TT_TIME data type, 1-27
TT_TIMESTAMP data type, 1-5, 1-27
TT_TINYINT data type, 1-19, 1-27
TT_VARCHAR data type, 1-9, 1-35
ttRepSyncSet built-in procedure, 5-49
type conversion
  and overflow, 1-32
type mode, 1-1
TypeMode connection attribute, 1-1

U

UID reserved word, 7-3
UID SQL function, 3-86
unary minus
  in expressions, 3-3
unary plus
  in expressions, 3-3
underflow
  defined, 1-32
Unicode characters
  example, 4-24
  pattern matching, 4-24
UNION reserved word, 7-3
unique constraints
  on tables, 5-189
UNIQUE INDEX clause
  defined, 5-69
UNIQUE reserved word, 7-3
unique rows, 5-167
UNISTR, 3-87
UNLOAD ANY CACHE GROUP system privilege
  definition, 6-3
UNLOAD CACHE GROUP statement
  defined, 5-186
UNLOAD object privilege
  definition, 6-4
UPDATE ANY TABLE system privilege
  definition, 6-3
UPDATE object privilege

definition, 6-4
UPDATE reserved word, 7-3
UPDATE SET clause
in MERGE statement, 5-155
UPDATE statement, 5-188
FIRST N clause, 5-188
string truncation, 5-189
WHERE clause omitted, 5-189
UPPER function, 3-45
USER function, 3-90
USER functions, 3-88
user ID in names, 2-1
user managed cache group, 5-53
USER reserved word, 7-3
USING reserved word, 7-3
UTF-8 Unicode characters, 3-12

V

VARBINARY data type, 1-5, 1-20, 1-27, 1-35
VARCHAR data type, 1-5
VARCHAR reserved word, 7-3
VARCHAR2 data type, 1-5, 1-13, 1-27
variables in SQL statements, 2-2
views
CREATE MATERIALIZED VIEW statement, 5-73
CREATE VIEW statement, 5-124
restrictions on detail tables, 5-74
restrictions on queries, 5-75, 5-124
restrictions on views, 5-74

W

WHEN reserved word, 7-3
WHERE clause
in SELECT statement, 5-167
WHERE reserved word, 7-3

X

XLA system privilege
definition, 6-3