

Oracle® TimesTen In-Memory Database TTClasses Guide



Release 22.1

F35399-03

February 2023

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Copyright © 2006, 2023, Oracle and/or its affiliates.

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 is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware 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 that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about 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 unless otherwise set forth in an applicable agreement between you and Oracle. 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, except as set forth in an applicable agreement between you and Oracle.

Contents

What's New

New features in Release 22.1.1.1.0 viii

1 TTClasses Development Environment

Setting TimesTen Environment Variables	1-1
Set Environment Variables on Linux or UNIX	1-1
Set Environment Variables on Windows	1-1
Compiling and Linking Applications	1-2
Compiling and Linking Applications on Linux or UNIX	1-2
Compiling and Linking Applications on Windows	1-3
Considerations when Using an ODBC Driver Manager (Windows)	1-4
About TimesTen Quick Start and Sample Applications	1-4

2 Understanding and Using TTClasses

Overview of TTClasses	2-1
Using TTCmd, TTConnection, and TTConnectionPool	2-1
Managing TimesTen Connections	2-4
Setting Connection Attributes for the Database	2-4
Connecting and Disconnecting	2-5
Connection Methods	2-6
Using TimesTen Features in your Application	2-7
Binding Parameters	2-7
About Parameter Binding	2-7
Binding Input Parameters	2-8
Registering Parameters	2-9
Parameter C Type to SQL Type Mappings	2-10
Binding Output or Input/Output Parameters	2-11
Binding Duplicate Parameters	2-13
Working with REF CURSORS	2-14
Working with ROWIDs	2-17
Working with LOBs	2-18

Differences Between TimesTen LOBs and Oracle Database LOBs	2-18
Using the LOB Simple Data Interface in TTClasses	2-18
Passthrough LOBs in TTClasses	2-21
Setting a Timeout or Threshold for Executing SQL Statements	2-21
Using Automatic Client Failover in a TTClasses Application	2-22
Using TTClasses Logging	2-22
Using XLA in TTClasses	2-23
About TimesTen XLA	2-23
Acknowledging XLA Updates Without Using Transaction Boundaries	2-24
Acknowledging XLA Updates at Transaction Boundaries	2-25
XLA System Privilege	2-26
Using the TTClasses Classes	2-26
How to Use the Commonly Used Classes	2-27
TTGlobal Usage	2-27
TTStatus Usage	2-27
TTConnection Usage	2-28
TTConnectionPool Usage	2-29
TTCmd Usage	2-30
How to Use the System Catalog Classes	2-31
TTCatalog Usage	2-31
TTCatalogTable Usage	2-32
TTCatalogColumn Usage	2-32
TTCatalogIndex Usage	2-32
TTCatalogSpecialColumn Usage	2-32
How to Use the XLA Classes	2-33
About the XLA Classes	2-33
TTXlaPersistConnection Usage	2-33
TTXlaRowViewer Usage	2-34
TTXlaTableHandler Usage	2-35
TTXlaTableList Usage	2-35
TTXlaTable Usage	2-36
TTXlaColumn Usage	2-36

3 Commonly Used Classes Reference

TTGlobal Reference	3-1
Public Members	3-1
Public Methods	3-1
TTStatus Reference	3-3
Subclasses	3-3
Public Members	3-3

Public Methods	3-3
TTConnection Reference	3-4
Public Members	3-5
Public Methods	3-5
TTConnectionPool Reference	3-9
Public Members	3-9
Public Methods	3-9
TTCmd Reference	3-12
Public Members	3-12
Public Methods for General Use and Non-Batch Operations	3-12
Public Methods for Obtaining TTMcmd Object Properties	3-25
Public Methods for Batch Operations	3-27

4 System Catalog Classes Reference

TTCatalog Reference	4-1
Public Members	4-1
Public Methods	4-1
TTCatalogTable Reference	4-4
Public Members	4-4
Public Methods	4-4
TTCatalogColumn Reference	4-7
Public Members	4-7
Public Methods	4-7
TTCatalogIndex Reference	4-8
Public Members	4-8
Public Methods	4-8
TTCatalogSpecialColumn Reference	4-10
Public Members	4-10
Public Methods	4-10

5 XLA Classes Reference

TTXlaPersistConnection Reference	5-1
Public Members	5-1
Public Methods	5-1
TTXlaRowViewer Reference	5-4
Public Members	5-4
Public Methods	5-4
TTXlaTableHandler Reference	5-10
Public Members	5-10

Protected Members	5-10
Public Methods	5-10
TTXlaTableList Reference	5-12
Public Members	5-13
Public Methods	5-13
TTXlaTable Reference	5-13
Public Members	5-14
Public Methods	5-14
TTXlaColumn Reference	5-14
Public Members	5-15
Public Methods	5-15

About This Content

This document provides usage and reference information for TimesTen support for TTClasses.

Audience

This guide is for application developers who administer and access TimesTen through TTClasses.

In addition to familiarity with the particular programming interface you use, you should be familiar with TimesTen, SQL (Structured Query Language), database operations, and ODBC.

Documentation Accessibility

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

Access to Oracle Support

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

Related Resources

Oracle Database documentation is available on the Oracle documentation website. This may be especially useful for Oracle Database features that TimesTen supports but does not attempt to fully document, such as OCI and Pro*C/C++. In particular, these Oracle Database documents may be of interest:

- *Oracle Database Globalization Support Guide*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database SQL Language Reference*

This guide frequently refers to ODBC API reference documentation for further information. This is available from Microsoft or a variety of third parties. For example:

<https://docs.microsoft.com/en-us/sql/odbc/reference/syntax/odbc-api-reference>

Conventions

The following text conventions are used in this document.

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

What's New

This section summarizes new features and functionality of TimesTen Release 22.1 that are documented in this guide, providing links into the guide for more information.

New features in Release 22.1.1.0

- To eliminate conversion warnings seen with the `-Wconversion` compiler flag, there have been data type changes in signatures of these methods:
 - In `TTCmd`: `ExecuteBatch()`, `getMaxRows()`, `getParamPrecision()`, `getColumnPrecision()`, `getColumnLength()`, `getColumn()`, `getColumnNullable()`, `getNextColumn()`, `getNextColumnNullable()`, `registerParam()`, `getParam()`.
 - In `TTXlaRowViewer`: `getColumnPrecision()`, `Get()`.See "[Commonly Used Classes Reference](#)" and "[XLA Classes Reference](#)" for reference information about these classes and methods.
- The TimesTen driver manager is included in this release. For related information, see "[Compiling and Linking Applications on Linux or UNIX](#)".

1

TTClasses Development Environment

TTClasses comes compiled and preconfigured with a TimesTen installation.

The information here includes these topics:

- [Setting TimesTen Environment Variables](#)
- [Compiling and Linking Applications](#)
- [About TimesTen Quick Start and Sample Applications](#)

Setting TimesTen Environment Variables

This section discusses how to set environment variables for TimesTen, on Linux or UNIX or on Windows.

- [Set Environment Variables on Linux or UNIX](#)
- [Set Environment Variables on Windows](#)

Set Environment Variables on Linux or UNIX

To use TTClasses on a Linux or UNIX system, ensure that your shell environment variables are set correctly through the appropriate `ttenv` script in the `timesten_home/bin` directory, where `timesten_home` is the TimesTen instance home directory.

The script is `ttenv.sh` or `ttenv.csh` on Linux or UNIX platforms (which you use depends on your shell).

Instead of doing this directly, you can add a line for the appropriate script to your login initialization script (for example, `.profile` or `.cshrc`).

See Environment Variables in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*.

Set Environment Variables on Windows

Before recompiling your application on a Windows system, ensure that the `PATH`, `INCLUDE`, and `LIB` environment variables point to the correct Visual Studio directories.

Execute the applicable Visual Studio C++ batch file (for example, `VCVARS32.BAT` or `VSVARS32.BAT`) to accomplish this.

Then set environment variables for TimesTen by running the following, where `timesten_home` is the TimesTen instance home directory:

```
timesten_home\bin\ttenv.bat
```

Compiling and Linking Applications

This section discusses how to compile and link your TTClasses applications, including a section on considerations when using an ODBC driver manager on Windows.

- [Compiling and Linking Applications on Linux or UNIX](#)
- [Compiling and Linking Applications on Windows](#)
- [Considerations when Using an ODBC Driver Manager \(Windows\)](#)

**Note:**

Recompile and relink your TTClasses applications after any TimesTen version upgrade.

You can also refer to the following sections in *Oracle TimesTen In-Memory Database C Developer's Guide* for related information:

- [Linking Options](#) for general information about TimesTen linking options, such as using the direct driver versus the client driver or, on Windows, whether to use a driver manager
- [Compiling and Linking Applications](#)

Compiling and Linking Applications on Linux or UNIX

This section documents the TTClasses libraries available to link your applications to, including consideration of the TimesTen driver manager (TTDM).

TTDM is a lightweight ODBC driver manager that is designed and optimized for use with the TimesTen database. An application links directly to the TTDM library, and TTDM dynamically loads the relevant ODBC driver libraries and passes ODBC calls from the application as needed. TTDM allows an application to use TimesTen direct and client/server connections at the same time. See [Introduction to the TimesTen Driver Manager](#) in *Oracle TimesTen In-Memory Database C Developer's Guide*.

For compiling your applications, include the TTClasses header files that are in the `installation_dir/include/ttclasses` directory. You can accomplish this by including `TTInclude.h` from that directory, using the symbolic link from `timesten_home/install` to `installation_dir`, as follows.

Use the following compile command:

```
-Itimesten_home/install/include
```

And use this line in your code:

```
#include <ttclasses/TTInclude.h>
```

TTClasses XLA programs must also include the following:

```
#include <ttclasses/TTXla.h>
```

The following table lists the TTClasses libraries available for linking your applications on Linux or UNIX platforms. As indicated, for each TTClasses library there is a corresponding TimesTen library you must also link your application to.

Usage	TTClasses Library	TimesTen Library
TimesTen direct connections	libttclasses.so	libtten.so
TimesTen client/server connections	libttclassesCS.so	libttenCS.so
TimesTen driver manager	libttclassesTTDM.so	libttdrvMgr.so

For example, adding the following to the link command would result in use of the client driver:

```
-Ltimesten_home/install/lib -lttclassesCS
```

The `-L` option tells the linker to search the TimesTen `lib` directory for library files. The `-lttclassesCS` option links in the driver.

On AIX, when linking applications with the TimesTen ODBC client driver, the C++ runtime library must be included in the link command (because the client driver is written in C++ and AIX does not link it automatically) and must follow the client driver:

```
-Ltimesten_home/install/lib -lttclassesCS -lC_r
```

You can use the Makefile in the TimesTen Classic Quick Start to guide you in creating your own Makefile. (See [About TimesTen Quick Start and Sample Applications](#).)

Compiling and Linking Applications on Windows

For compiling your applications, include the TTClasses header files that are in the `installation_dir\include\ttclasses` directory.

You can accomplish this by including `TTInclude.h` from that directory, using the symbolic link from `timesten_home/install` to `installation_dir`, as follows.

Use the following compile command:

```
/Itimesten_home\install\include
```

And use the following line in your code:

```
#include <ttclasses/TTInclude.h>
```

TTClasses XLA programs must also include the following:

```
#include <ttclasses/TTXla.h>
```

The following table lists the TTClasses libraries available for linking your applications on Windows platforms.

Usage	Library
TimesTen client/server connections	ttclasses221CS.lib
Microsoft ODBC driver manager	ttclasses221DM.lib See the next section, Considerations when Using an ODBC Driver Manager (Windows) .

Add the appropriate library, for example `timesten_home\install\lib\ttclasses221.lib`, to your link command.

You can use the Makefile in the Quick Start `sample_code/ttclasses` directory to guide you in creating your own Makefile. (See [About TimesTen Quick Start and Sample Applications](#).)

Considerations when Using an ODBC Driver Manager (Windows)

Be aware of the following limitations in TTClasses when you use an ODBC driver manager on Windows.

- XLA functionality is not supported.
- REF CURSOR functionality is not supported.

In addition, the driver manager does not support the ODBC C types `SQL_C_BIGINT` and `SQL_C_TINYINT` when used with TimesTen. When using the driver manager, you cannot call methods that use either of these data types in their signatures, such as the applicable overloaded versions of any of the following `TTCmd` methods: `getColumn()`, `getColumnNullable()`, `getNextColumn()`, `getNextColumnNullable()`, `setParam()`, `getParam()`, and `BindParameter()`.

About TimesTen Quick Start and Sample Applications

The TimesTen Classic Quick Start and TimesTen Scaleout sample applications are available from the TimesTen GitHub location. For the TimesTen Classic Quick Start, there is a complete set of tutorials, how-to instructions, and sample applications. For TimesTen Scaleout, there are ODBC and JDBC sample applications. After you have configured your environment, you can confirm that everything is set up correctly by compiling and running the sample applications. For TimesTen Classic, applications are located under the Quick Start `sample_code` directory. For instructions on compiling and running them, see the instructions in the subdirectories. For TimesTen Scaleout, clone the `oracle-timesten-examples` GitHub repository and follow the instructions in the README files.

For TimesTen Classic, the following are included:

- Schema and setup: The `build_sampledb` script (`.sh` on Linux or UNIX or `.bat` on Windows) creates a sample database and schema. Run this script before using the sample applications.
- Environment and setup: The `ttquickstartenv` script (`.sh` or `.csh` on Linux or UNIX, `.bat` on Windows, or as applicable for your system), a superset of the

`ttenv` script typically used for TimesTen setup, sets up the environment. Run this script each time you enter a session where you want to compile or run any of the sample applications.

- Sample applications and setup: The Quick Start provides sample applications and their source code for TTClasses.

2

Understanding and Using TTClasses

This chapter provides some general overview and best practices for TTClasses. It includes the following topics:

- [Overview of TTClasses](#)
- [Using TTCmd, TTConnection, and TTConnectionPool](#)
- [Managing TimesTen Connections](#)
- [Using TimesTen Features in your Application](#)
- [Using TTClasses Logging](#)
- [Using XLA in TTClasses](#)
- [Using the TTClasses Classes](#)

Overview of TTClasses

The TimesTen C++ Interface Classes library (TTClasses) provides wrappers around the most common ODBC functionality to allow database access. It was developed to meet the demand for an API that is easier to use than ODBC but does not sacrifice performance.

TimesTen supports:

- ODBC 2.5, Extension Level 1, as well as some Extension Level 2 features
- ODBC 3.51 core interface conformance

The TTClasses implementation is based on ODBC 2.5.

See TimesTen ODBC Support in *Oracle TimesTen In-Memory Database C Developer's Guide*. Refer to ODBC API reference documentation for general information about ODBC.

In addition to providing a C++ interface to the TimesTen ODBC interface, TTClasses supplies an interface to the TimesTen Transaction Log API (XLA), which is supported by TimesTen Classic. XLA allows an application to monitor one or more tables in a database. When other applications change that table, the changes are reported through XLA to the monitoring application. TTClasses provides a convenient interface to the most commonly used aspects of XLA functionality. For general information about XLA, see XLA and TimesTen Event Management in *Oracle TimesTen In-Memory Database C Developer's Guide*.

TTClasses is also intended to promote best practices when writing application software that uses the TimesTen Data Manager. The library uses TimesTen in an optimal manner. For example, autocommit is disabled by default. Parameterized SQL is strongly encouraged and its use is greatly simplified in TTClasses compared to hand-coded ODBC.

Using TTCmd, TTConnection, and TTConnectionPool

This section describes a general approach to using TTClasses that has been employed successfully and can easily be adapted to a variety of applications.

To achieve optimal performance, real-time applications should use prepared SQL statements. Ideally, all SQL statements used by an application are prepared when the application begins, using a separate `TTCmd` object for each statement. In ODBC, and thus in `TTClasses`, statements are bound to a particular connection, so a full set of the statements used by the application are often associated with every connection to the database.

A convenient way to accomplish this is to develop an application-specific class that is derived from `TTConnection`. For an application named `XYZ`, you can create a class `XYZConnection`, for example. The `XYZConnection` class contains private `TTCmd` members representing the prepared SQL statements that can be used in the application, and provides new public methods to implement the application-specific database functionality through these private `TTCmd` members.

Before a `TTCmd` object can be used, a SQL statement (such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`) must be associated with it. The association is accomplished by using the `Prepare()` method, which also compiles and optimizes the SQL statement to ensure it is executed in an efficient manner. Note that the `Prepare()` method only prepares and does not execute the statement.

With TimesTen, statements are typically parameterized for better performance. Consider the following SQL statements:

```
SELECT col1 FROM table1 WHERE C = 10;
SELECT col1 FROM table1 WHERE C = 11;
```

It is more efficient to prepare a single parameterized statement and execute it multiple times:

```
SELECT col1 FROM table1 WHERE C = ?;
```

The value for "?" is specified at runtime by using the `TTCmd::setParam()` method.

There is no need to explicitly bind columns or parameters to a SQL statement, as is necessary when you use ODBC directly. `TTCmd` automatically defines and binds all necessary columns at prepare time. Parameters are bound at execution time.

Be aware that preparing is a relatively expensive operation. When an application establishes a connection to TimesTen, using `TTConnection::Connect()`, it should prepare all `TTCmd` objects associated with the connection. Prepare all SQL statements prior to the main execution loop of the application.

Anytime a `TTClasses` method encounters an error or warning, it throws a `TTStatus` object as an exception, which the application should catch and handle appropriately. The TimesTen Classic Quick Start sample applications show examples of how this is done. See [About TimesTen Quick Start and Sample Applications](#).

 **Note:**

If `TTConnection` or `TTCmd` lacks the specific get or set method you need, you can access underlying ODBC connection and statement handles directly, through the `TTConnection::getHdbc()` and `TTCmd::getHandle()` methods. Similarly, there is a `TTGlobal::sqlhenv()` method to access the ODBC environment handle.

This is an example of a class that inherits from `TTConnection`.

```
class XYZConnection : public TTConnection {
private:
    TTCmd updateData;
    TTCmd insertData;
    TTCmd queryData;

public:
    XYZConnection();
    ~XYZConnection();
    virtual void Connect (const char* connStr, const char* user, const char*
pwd);
    void updateUser ();
    void addUser (char* nameP);
    void queryUser (const char* nameP, int* valueP);
};
```

In this example, an `XYZConnection` object is a connection to TimesTen that can be used to perform three application-specific operations: `addUser()`, `updateUser()`, and `queryUser()`. These operations are specific to the XYZ application. The implementation of these three methods can use the `updateData`, `insertData`, and `queryData` `TTCmd` objects to implement the database operations of the application.

To prepare the SQL statements of the application, the `XYZConnection` class overloads the `Connect()` method provided by the `TTConnection` base class. The `XYZConnection::Connect()` method calls the `Connect()` method of the base class to establish the database connection and also calls the `Prepare()` method for each `TTCmd` object to prepare the SQL statements for later use.

This next example shows an implementation of the `XYZConnection::Connect()` method.

```
void
XYZConnection::Connect(const char* connStr, const char* user, const char*
pwd)
{
    try {
        TTConnection::Connect(connStr, user, pwd);
        updateData.Prepare(this, "update mydata v set foo = ? where bar = ?");
        insertData.Prepare(this, "insert into mydata values(?,0)");
        queryData.Prepare(this, "select i from mydata where name = ?");
    }
    catch (TTStatus st) {
        cerr << "Error in XYZConnection::Connect: " << st << endl;
    }
}
```



```
    return;  
}
```

This `Connect()` method makes the `XYZConnection` object and its application-specific methods fully operational.

This approach also works well with the design of the `TTConnectionPool` class. The application can create numerous objects of type `XYZConnection` and add them to a `TTConnectionPool` object. By calling `TTConnectionPool::ConnectAll()`, the application connects all connections in the pool to the database and prepares all SQL statements. Use `TTConnectionPool::DisconnectAll()` to disconnect.

This application design allows database access to be easily separated from the application business logic. Only the `XYZConnection` class contains database-specific code.

Examples of this application design can be found in several of the `TTClasses` sample applications provided with TimesTen Classic Quick Start. See [About TimesTen Quick Start and Sample Applications](#).

Note that other configurations are possible. Some customers have extended this scheme further, so that SQL statements to be used in an application are listed in a table in the database, rather than being hard-coded in the application itself. This allows changes to database functionality to be implemented by making database changes rather than application changes.

The following example shows an implementation of the `XYZConnection::Disconnect()` method.

```
void  
XYZConnection::Disconnect()  
{  
    updateData.Drop();  
    insertData.Drop();  
    queryData.Drop();  
  
    TTConnection::Disconnect();  
}
```

Also see [TTCmd Usage](#), [TTConnection Usage](#), and [TTConnectionPool Usage](#).

Managing TimesTen Connections

This section covers how to connect to a database.

- [Setting Connection Attributes for the Database](#)
- [Connecting and Disconnecting](#)
- [Connection Methods](#)

Setting Connection Attributes for the Database

This section refers to information about setting up a TimesTen database, including initial connection attribute settings.

For TimesTen Classic, *Oracle TimesTen In-Memory Database Operations Guide* contains information about creating a DSN (data source name) for a database. A DSN is a logical name that identifies a TimesTen database and the set of connection attributes used for connecting to the database. The type of DSN you create depends on whether your application connects directly to the database or connects by a client. If you intend to connect directly to the database, refer to *Managing TimesTen Databases*. If you intend to create a client connection to the database, refer to *Working with the TimesTen Client and Server*.

For TimesTen Scaleout, refer to *Oracle TimesTen In-Memory Database Scaleout User's Guide* for information about creating a database, setting connection attributes, and connecting to a database, using either a direct connection or a client/server connection. See *Creating a Database and Connecting to a Database*.

 **Note:**

A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating (forking) a child process, the child must not use the connection.

Connecting and Disconnecting

Based on the `XYZConnection` class, you can connect to and disconnect from TimesTen.

See [Using TTCmd](#), [TTConnection](#), and [TTConnectionPool](#).

For example:

```
...

XYZConnection conn;
char connStr[256];
char user[30];
char pwd[30];

...

try {
    conn.Connect(connStr, user, pwd);
}
catch (TTWarning st) {
    cerr << "Warning connecting to TimesTen: " << st << endl;
}
catch (TTError st) {
    cerr << "Error connecting to TimesTen " << st << endl;
    exit(1);
}

// ... Work with the database connection...

try {
    conn.Disconnect();
}
catch (TTStatus st) {
```

```
    cerr << "Error disconnecting from TimesTen: " << st << endl;
    exit(1);
}
```

Connection Methods

There are method signatures for the `TTConnection`, `TTConnectionPool`, and `TTXlaPersistConnection` classes.

```
virtual void
TTConnection::Connect(const char* connStr)

virtual void
TTConnection::Connect(const char* connStr, const char* username,
                      const char* password)

virtual void
TTConnection::Connect(const char* connStr,
                      DRIVER_COMPLETION_ENUM driverCompletion)

void
TTConnectionPool::ConnectAll(const char* connStr)

void
TTConnectionPool::ConnectAll(const char* connStr, const char*
username,
                             const char* password)

virtual void
TTXlaPersistConnection::Connect(const char* connStr, const char*
username,
                                const char* password, const char*
bookmarkStr,
                                bool createBookmarkFlag)

virtual void
TTXlaPersistConnection::Connect(const char* connStr,
                                DRIVER_COMPLETION_ENUM
driverCompletion,
                                const char * bookmarkStr, bool
createBookmarkFlag)

virtual void
TTXlaPersistConnection::Connect(const char* connStr, const char*
username,
                                const char* password, const char*
bookmarkStr)

virtual void
TTXlaPersistConnection::Connect(const char* connStr,
                                DRIVER_COMPLETION_ENUM
driverCompletion,
                                const char * bookmarkStr)
```

 **Note:**

- The connection string (*connStr* value) can specify the user name and password, such as "DSN=testdb;uid=brian;pwd=password". Be aware that for signatures that take connection string, user name, and password arguments, the user name and password arguments take precedence over any user name or password specified in the connection string.
- See [TTConnection Reference](#) for information about DRIVER_COMPLETION_ENUM values.

Using TimesTen Features in your Application

This section covers the following topics for working with data.

- [Binding Parameters](#)
- [Working with REF CURSORS](#)
- [Working with ROWIDs](#)
- [Working with LOBs](#)
- [Setting a Timeout or Threshold for Executing SQL Statements](#)
- [Using Automatic Client Failover in a TTClasses Application](#)

Binding Parameters

This section discusses parameter binding for SQL statements.

- [About Parameter Binding](#)
- [Binding Input Parameters](#)
- [Registering Parameters](#)
- [Parameter C Type to SQL Type Mappings](#)
- [Binding Output or Input/Output Parameters](#)
- [Binding Duplicate Parameters](#)

 **Note:**

The term "bind parameter" as used in TimesTen developer guides (in keeping with ODBC terminology) is equivalent to the term "bind variable" as used in TimesTen PL/SQL documents (in keeping with Oracle Database PL/SQL terminology).

About Parameter Binding

The `TTCmd` class supplies the methods `setParam()` and `BindParameter()` (for batch operations) to bind parameters. It also supplies the method `registerParam()` to support output and input/output parameters or to override default bind types.

The TimesTen binding mechanism (early binding) differs from that of Oracle Database (late binding). TimesTen requires the data types before preparing queries. As a result, there will be an error if the data type of each bind parameter is not specified or cannot be inferred from the SQL statement. This would apply, for example, to the following statement:

```
SELECT 'x' FROM DUAL WHERE ? = ?;
```

You could address the issue as follows, for example:

```
SELECT 'x' from DUAL WHERE CAST(? as VARCHAR2(10)) =  
                           CAST(? as VARCHAR2(10));
```

Binding Input Parameters

For non-batch operations, use the `TTCmd::setParam()` method to bind input parameters for SQL statements, specifying the parameter position and the value to be bound. For batch operations, use the `TTCmd::BindParameter()` method.

See [setParam\(\)](#) and [ExecuteBatch\(\)](#) for an example of batch operations.

For non-batch operations, the example below shows snippets from a class `SampleConnection`, where parameters are bound to insert a row into a table. (This example is from the `basics.cpp` sample application provided with the TimesTen Classic Quick Start. See [About TimesTen Quick Start and Sample Applications.](#))

Assume a table `basics`, defined as follows:

```
create table basics (name char(10) not null primary key, i tt_integer);
```

Implementation of the `Connect()` method is omitted here, but see the example of a `Connect()` implementation in [Using TTCmd, TTConnection, and TTConnectionPool.](#)

```
class SampleConnection : public TTConnection  
{  
    using TTConnection::Connect;  
  
    private:  
        TTCmd          insertData;  
        ...  
  
    protected:  
  
    public:  
        SampleConnection();  
        ~SampleConnection();  
        virtual void Connect(const char* connStr,  
                             DRIVER_COMPLETION_ENUM driverCompletion);  
        void insert(char* nameP);  
        ...  
  
        ...  
        // Assume a Connect() method implemented with the following:
```

```
        // insertData.Prepare(this, "insert into basics values(:name, :value)");
        ...
    }

//-----

void
SampleConnection::insert(char* nameP)
{
    static long i = 0;
    insertData.setParam(1, nameP);
    insertData.setParam(2, i++);
    insertData.Execute();
}

//-----

...

int
main(int argc, char** argv)
{
    ...
    char name[10];
    SampleConnection conn;
    ...

    // Assume conn is an open connection.
    sprintf(name, "Robert");
    try {
        conn.insert(name);
    }
    catch (TTStatus st) {
        cerr << "Error inserting row " << name << ":" << st << endl;
        conn.Rollback();
    }
}
```

Registering Parameters

The `TTCmd` class provides the `registerParam()` method, which enables you to specify the SQL type, precision, and scale of a parameter (as applicable) and whether the parameter is input, output, or input/output.

A `registerParam()` call is required for an output or input/output parameter, which could be a REF CURSOR (output only) or a parameter from a PL/SQL `RETURNING INTO` clause (output only), procedure, or function.

A `registerParam()` call can be either before or after the related `setParam()` or `BindParameter()` call and takes precedence regarding SQL type, precision, and scale (as applicable).

The method signature is as follows:

```
inline void
TTCmd::registerParam(int    pno,
                    int    inputOutputType,
                    short  sqltype,
                    int    precision = 0,
                    short  scale = 0)
```

- *pno* is the parameter position in the statement.
- *inputOutputType* can be `TTCmd::PARAM_IN`, `TTCmd::PARAM_OUT`, or `TTCmd::PARAM_INOUT`.
- *sqltype* is the SQL type of the data (for example, `SQL_INTEGER`).
- *precision* and *scale* (both optional) are used the same way as in an ODBC `SQLBindParameter` call. For primitive types (such as `int`), *precision* and *scale* settings are ignored.



Note:

See [Binding Output or Input/Output Parameters](#), for an example. Also see [registerParam\(\)](#) for additional reference information.

Parameter C Type to SQL Type Mappings

For an input parameter, TTClasses by default derives the SQL type from the bound C type for the `setParam()` or `BindParameter()` call according to the mappings.

It is not typical to need a `registerParam()` call for an input parameter, but you can call it if you must use a particular SQL type or precision or scale.

Table 2-1 TTClasses C Type to SQL Type Mappings

C Type	SQL Type
<code>char*</code>	<code>SQL_CHAR</code> , <code>SQL_VARCHAR</code>
<code>void*</code>	<code>SQL_BINARY</code> , <code>SQL_VARBINARY</code>
<code>double</code>	<code>SQL_FLOAT</code> , <code>SQL_DOUBLE</code>
<code>DATE_STRUCT</code>	<code>SQL_DATE</code>
<code>float</code>	<code>SQL_REAL</code> , <code>SQL_DECIMAL</code>
<code>int</code>	<code>SQL_INTEGER</code>
<code>SQLBIGINT</code>	<code>SQL_BIGINT</code>
<code>SQLCHAR*</code>	<code>SQL_VARCHAR</code>
<code>SQLINTEGER</code>	<code>SQL_INTEGER</code>
<code>SQLSMALLINT</code>	<code>SQL_SMALLINT</code>
<code>SQLTINYINT</code>	<code>SQL_TINYINT</code>

Table 2-1 (Cont.) TTClasses C Type to SQL Type Mappings

C Type	SQL Type
SQLWCHAR*	SQL_WCHAR, SQL_WVARCHAR
TIME_STRUCT	SQL_TIME
TIMESTAMP_STRUCT	SQL_TIMESTAMP
SQLHSTMT	SQL_REFCURSOR

**Note:**

Not all C types shown in the preceding table are supported if you are using a driver manager. Refer to [Considerations when Using an ODBC Driver Manager \(Windows\)](#).

Binding Output or Input/Output Parameters

TTClasses supports output and input/output parameters such as REF CURSORS (output only), parameters from a PL/SQL procedure or function that has OUT or IN OUT parameters, or a parameter from a RETURNING INTO clause (output only).

You must use the `TTCmd::registerParam()` method, described in the preceding section, to inform TTClasses if a parameter in a SQL statement is output or input/output. For the `inputOutputType` setting in the method call, use `TTCmd::PARAM_OUT` or `TTCmd::PARAM_INOUT` as appropriate.

For non-batch operations, after the SQL statement has been executed, use the appropriate `TTCmd::getParam()` method to retrieve the output value, specifying the parameter position and the variable into which the value is placed. There is a signature for each data type.

For batch operations, `TTCmd::bindParam()` is used for output or input/output parameters as well as for input parameters. It is called before the statement is executed. After statement execution, the data for an output value is in the buffer specified in the `bindParam()` call. `bindParam()` has a signature for each data type. For an input/output parameter in batch operations, `bindParam()` is called only once, before statement execution. Before execution the specified buffer contains the input, and after statement execution it contains the output.

The following examples provide code fragments showing the use of output and input/output parameters.

The first example uses input and output parameters. The `setParam()` call binds the value of the input parameter :a. The `getParam()` call retrieves the value of the output parameter :b. The output parameter is also registered as required.

```
...
// t1 has a single TT_INTEGER column
cmd.Prepare(&conn, "insert into t1 values (:a) returning c1 into :b");
cmd.setParam(1, 99);
cmd.registerParam(2, TTMd::PARAM_OUT, SQL_INTEGER);
```



```
cmd.Execute();
SQLINTEGER outval;

if (cmd.getParam(2, &outval))
    cerr << "The output value is null." << endl;
else
    cerr << "The output value is " << outval << endl;
...

```

The next example uses input and output parameters in a batch operation. The first `BindParameter()` call provides the input data for the first parameter :a. The second `BindParameter()` call provides a buffer for output data for the second parameter :b.

```
...
#define BATCH_SIZE 5
int input_int_array[BATCH_SIZE] = { 91, 92, 93, 94, 95 };
int output_int_array[BATCH_SIZE] = { -1, -1, -1, -1, -1 };
SQLULEN numrows;

cmd.PrepareBatch(&conn, "insert into t1 values (:a) returning c1
into :b",
                BATCH_SIZE);
cmd.BindParameter(1, BATCH_SIZE, input_int_array);
cmd.BindParameter(2, BATCH_SIZE, output_int_array);
cmd.registerParam(2, TTCmd::PARAM_OUT, SQL_INTEGER);
numrows = cmd.ExecuteBatch(BATCH_SIZE);
...

```

The following example uses an input/output parameter. It is registered as required. The `setParam()` call binds its input value and the `getParam()` call retrieves its output value.

```
...
cmd.Prepare(&conn, "begin :x := :x + 1; end;");
cmd.registerParam(1, TTCmd::PARAM_INOUT, SQL_INTEGER);
cmd.setParam(1, 99);
cmd.Execute();
SQLINTEGER outval;

if (cmd.getParam(1, &outval))
    cerr << "The output value is null." << endl;
else
    cerr << "The output value is " << outval << endl;
...

```

This final example uses output and input/output parameters. Assume a PL/SQL procedure as follows:

```
create or replace procedure my_proc (
    a in number,
    b in number,
    c out number,
    d in out number ) as

```

```

begin
  c := a + b;
  d := a + b - d;
end my_proc;

```

The input parameters for the procedure are taken as constants in this example rather than as bound parameters, so only the OUT parameter and IN OUT parameter are bound. Both are registered as required. The `setParam()` call provides the input value for the IN OUT parameter `:var1`. The first `getParam()` call retrieves the value for the OUT parameter `:sum`. The second `getParam()` call retrieves the output value for the IN OUT parameter `:var1`.

```

...
cmd.Prepare(&conn, "begin my_proc (10, 5, :sum, :var1); end;");
cmd.registerParam (1, TTCmd::PARAM_OUT, SQL_DECIMAL, 38);
cmd.registerParam (2, TTCmd::PARAM_INOUT, SQL_DECIMAL, 38);
cmd.setParam(2, "99");
cmd.Execute();
SQLINTEGER outval1, outval2;

if (cmd.getParam(1, &outval1))
  cerr << "The first output value is null." << endl;
else
  cerr << "The first output value is " << outval << endl;
if (cmd.getParam(2, &outval2))
  cerr << "The second output value is null." << endl;
else
  cerr << "The second output value is " << outval << endl;
...

```

Binding Duplicate Parameters

In TimesTen, multiple occurrences of the same parameter name in a SQL statement are considered to be distinct parameters. (This is consistent with Oracle Database support for binding duplicate parameters.)

Note:

- "TimesTen mode" for binding duplicate parameters, and the `DuplicateBindMode` connection attribute, are deprecated.
- Refer to *Binding of Duplicate Parameters in SQL Statements in Oracle TimesTen In-Memory Database C Developer's Guide*.

Consider this query:

```

SELECT * FROM employees
  WHERE employee_id < :a AND manager_id > :a AND salary < :b;

```

When parameter position numbers are assigned, a number is given to each parameter occurrence without regard to name duplication. The application must, at a minimum, bind a value for the first occurrence of each parameter name. For any subsequent occurrence of a given parameter name, the application can bind a different value for the occurrence or it can leave the parameter occurrence unbound. In the latter case, the subsequent occurrence takes the same value as the first occurrence. In either case, each occurrence still has a distinct parameter position number.

This example uses a different value for the second occurrence of `a` in the SQL statement above:

```
mycmd.setParam(1, ...); // first occurrence of :a
mycmd.setParam(2, ...); // second occurrence of :a
mycmd.setParam(3, ...); // occurrence of :b
```

To use the same value for both occurrences of `a`:

```
mycmd.setParam(1, ...); // both occurrences of :a
mycmd.setParam(3, ...); // occurrence of :b
```

Parameter `b` is considered to be in position 3 regardless, and the number of parameters is considered to be three.

Working with REF CURSORS

REF CURSOR is a PL/SQL concept, a handle to a cursor over a SQL result set that can be passed between PL/SQL and an application.

In TimesTen, the cursor can be opened in PL/SQL, then the *REF CURSOR* can be passed to the application for processing. This usage is an *OUT REF CURSOR*, an *OUT* parameter with respect to PL/SQL. As with any output parameter, it must be registered using the `TTCmd::registerParam()` method.

See [Registering Parameters](#) and [Binding Output or Input/Output Parameters](#).

In the TimesTen implementation, the *REF CURSOR* is attached to a separate statement handle. The application prepares a SQL statement that has a *REF CURSOR* parameter on one statement handle, then, before executing the statement, binds a second statement handle as the value of the *REF CURSOR*. After the statement is executed, the application can describe, bind, and fetch the results using the same APIs as for any result set.

In *TTCmd* classes, because a *TTCmd* object encapsulates a single SQL statement, two *TTCmd* objects are used to support this *REF CURSOR* model.

See *PL/SQL REF CURSORS in Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for additional information about *REF CURSORS*.

Note:

For passing *REF CURSORS* between PL/SQL and an application, TimesTen supports only *OUT REF CURSORS*, from PL/SQL to the application.

The example below demonstrates the following steps for using a REF CURSOR in TTClasses.

1. Declare a `TTCmd` object for the PL/SQL statement that returns a REF CURSOR (`cmdPLSQL` in the example).
2. Declare a `TTCmd*` pointer to point to a second `TTCmd` object for the REF CURSOR (`cmdRefCursor` in the example).
3. Use the first `TTCmd` object (`cmdPLSQL`) to prepare the PL/SQL statement.
4. Use the `TTCmd::registerParam()` method of the first `TTCmd` object to register the REF CURSOR as an output parameter.
5. Use the first `TTCmd` object to execute the statement.
6. Use the `TTCmd::getParam()` method of the first `TTCmd` object to retrieve the REF CURSOR into the second `TTCmd` object (using `&cmdRefCursor`). There is a `getParam(int paramNo, TTCmd** rcCmd)` signature for REF CURSORS.
7. Fetch the results from the `TTCmd` object for the REF CURSOR and process as desired.
8. Drop the first `TTCmd` object.
9. Drop the pointer to the `TTCmd` object for the REF CURSOR.
10. Issue a `delete` statement to delete the `TTCmd` object for the REF CURSOR.

This example retrieves and processes a REF CURSOR from a PL/SQL anonymous block. See the preceding steps for an explanation.

```
...
TTCmd cmdPLSQL;
TTCmd* cmdRefCur;
TTConnection conn;
...

// c1 is a TT_INTEGER column.
cmdPLSQL.Prepare(&conn, "begin open :rc for select c1 from t; end;")
cmdPLSQL.registerParam(1, TTCmd::PARAM_OUT, SQL_REFCURSOR);
cmdPLSQL.Execute();

if (cmdPLSQL.getParam(1, &cmdRefCur) == false)
{
    SQLINTEGER fetchval;

    while (!cmdRefCursor->FetchNext()) {
        cmdRefCur->getColumn(1, &fetchval);
    }
    cmdRefCursor->Drop();
    delete cmdRefCursor;
}

cmdPLSQL.Drop();
```

Be aware of the following usage notes when using REF CURSORS in TTClasses:

- For passing REF CURSORS between PL/SQL and an application, TimesTen supports only `OUT REF CURSORS`, from PL/SQL to the application, and supports a statement returning only a single REF CURSOR.
- Unlike `TTCmd::getParam()` calls for other data types, a `getParam()` call with a `TTCmd**` parameter for a REF CURSOR can only be called once. Subsequent calls return `NULL`. If you must retrieve a REF CURSOR a second time, you must reexecute the statement.
- If the statement is executed multiple times, the REF CURSOR parameter must be reregistered each time. For example, if you are executing the statement, getting the REF CURSOR parameter, and fetching from the REF CURSOR within a loop, then the parameter registration must also be in the loop, such as follows:

```
cmdPLSQL.Prepare(...);

loop
    cmdPLSQL.registerParam(...);
    cmdPLSQL.Execute();
    cmdPLSQL.getParam(...);
    fetch loop
end loop
```

This is shown the example below.

- Any `TTCmd` object, including one for a REF CURSOR, has an ODBC statement handle allocated for it. The REF CURSOR statement handle is dropped at the time of the `Drop()` statement and the resource is freed after the `delete` statement.

The following example uses a REF CURSOR in a loop. Assume the following declarations and a `TTCConnection` instance `conn`.

```
...
TTCmd query;
TTCmd* ref_cur;
...
```

Here is the loop:

```
...
cerr << "Selecting values using cursor" << endl;
query.Prepare(&conn, "begin open :rc for select c1 from t1;
end;");

for (int round = 0; round < ROUNDS; round++) {
    cerr << "executing ref cursor round# " << (round+1) << endl;
    query.registerParam(1, TTCmd::PARAM_OUT, SQL_REFCURSOR);
    query.Execute();
    query.getParam(1, &ref_cur);

    while(true) {
        fetch_next = ref_cur -> FetchNext();
        if (fetch_next == 1)
            break;
    }
}
```

```
        ref_cur -> getColumn(1, &val);
        cerr << "val = " << val << endl;
    }
    ref_cur->Drop();
    delete ref_cur;
}

conn.Commit();
query.Drop();
...
```

Working with ROWIDs

Each row in a table has a unique identifier known as its *rowid*. An application can retrieve the rowid of a row from the `ROWID` pseudocolumn. Rowids can be represented in either binary or character format.

An application can specify literal rowid values in SQL statements, such as in `WHERE` clauses, as `CHAR` constants enclosed in single quotes.

The ODBC SQL type `SQL_ROWID` corresponds to the SQL type `ROWID`.

For parameters and result set columns, rowids are convertible to and from the C types `SQL_C_BINARY`, `SQL_C_WCHAR`, and `SQL_C_CHAR`. `SQL_C_CHAR` is the default C type for rowids. The size of a rowid is 12 bytes as `SQL_C_BINARY`, 18 bytes as `SQL_C_CHAR`, and 36 bytes as `SQL_C_WCHAR`.

Note that `TTCclasses` has always supported rowids as character strings; however, a `TTCclasses` application can now pass a rowid to a PL/SQL anonymous block as a `ROWID` type instead of as a string. This involves using the `TTCmd::registerParam()` method to register the rowid input parameter as `SQL_ROWID` type, as shown this example:

```
...
TTConnection conn;
TTCmd cmd;
...
cmd.Prepare(&conn, "begin delete from t1 where rowid = :x; end;");
cmd.registerParam(1, TTCmd::PARAM_IN, SQL_ROWID);
cmd.setParam(1, rowid_string);
cmd.Execute();
...
```

Refer to `ROWID Data Type` and `ROWID pseudocolumn` in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about rowids and the `ROWID` data type, including usage and life.



Note:

TimesTen does not support the PL/SQL type `UROWID`.

Working with LOBs

This section discusses the use of LOBs (large objects) in TTClasses. This includes CLOBs (character LOBs), NCLOBs (national character LOBs), and BLOBs (binary LOBs).

The following topics are discussed:

- [Differences Between TimesTen LOBs and Oracle Database LOBs](#)
- [Using the LOB Simple Data Interface in TTClasses](#)
- [Passthrough LOBs in TTClasses](#)

You can also refer to the following.

- Large objects (LOBs) in *Oracle TimesTen In-Memory Database C Developer's Guide* for an overview of LOBs and LOB programming interfaces for C and C++. Only the LOB simple data interface is applicable to TTClasses.
- LOB Data Types in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about LOBs in TimesTen
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for general information about programming with LOBs (but not specific to TimesTen functionality)

Differences Between TimesTen LOBs and Oracle Database LOBs

This section notes differences in LOB support between TimesTen and Oracle Database.

- A key difference between the TimesTen LOB implementation and the Oracle Database implementation is that in TimesTen, a LOB used in an application does not remain valid past the end of the transaction. All such LOBs are invalidated after a commit or rollback, whether explicit or implicit. This includes after any DDL statement.
- TimesTen does not support BFILEs, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs.
- TimesTen does not support binding arrays of LOBs.
- TimesTen does not support batch processing of LOBs.
- Relevant to BLOBs, there are differences in the usage of hexadecimal literals in TimesTen. see the description of *HexadecimalLiteral* in Constants in *Oracle TimesTen In-Memory Database SQL Reference*.

Using the LOB Simple Data Interface in TTClasses

The simple data interface enables applications to access LOB data by binding and defining, just as with other scalar types.

For the simple data interface in TTClasses, use `getParam()` and `setParam()` to bind parameters and use `getColumn()` or `getColumnNullable()` to define result columns. The application can bind or define using a SQL type that is compatible with the corresponding variable type, as follows:

- For BLOB data, use SQL type `SQL_LONGVARBINARY` and C type `SQL_C_BINARY`.
- For CLOB data, use SQL type `SQL_LONGVARCHAR` and C type `SQL_C_CHAR`.
- For NCLOB data, use SQL type `SQL_WLONGVARCHAR` and C type `SQL_C_WCHAR`.

 **Note:**

- TTClasses does not support batch mode for LOBs.
- Binding a CLOB or NCLOB with a C type of `SQL_C_BINARY` is prohibited.

The following example shows use of the LOB simple data interface in TTClasses. Assume a table with NCLOB, BLOB, and CLOB columns has been created and populated. The methods executed on these LOB types are the same as for NCHAR, BINARY, and CHAR, respectively.

```

#ifdef _WIN32
#include <ttcommon.h>
#endif
#include "TTInclude.h"
#define LOB_COL_SIZE 4194304

int main(int argc, char** argv) {

    TTConnection conn;
    TTCmd query;
    char conn_str[100] = "... your connection string ...";
    char tbl_name[20] = "... test table name ...";

    int num_rows = 0;
    char query_stmt[1000];
    int fetch_next;
    int value_is_null = 0;
    int column_type;
    SQLWCHAR * unicode_val;
    u_char * binary_val;
    char * alfanum_val;
    SQLLEN b_len;
    SQLLEN u_len;

    cerr << "Connecting to TimesTen <" << conn_str << ">" << endl;

    try {
        conn.Connect(conn_str);
        sprintf(query_stmt, "select * from %s", tbl_name);
        query.Prepare(&conn, query_stmt);
        query.Execute();
        const int num_result_cols = query.getNColumns();

        while (true) {
            // loop until no rows found
            // fetch a row; if no more rows, break out of loop
            // FetchNext returns 0 for success, 1 for SQL_NO_DATA_FOUND

```



```
fetch_next = query.FetchNext();
if (fetch_next == 1)
    break;

for (int col = 1; col <= num_result_cols; col++) {
    value_is_null = 0;
    column_type = query.getColumnType(col);

    switch (column_type) {

        case SQL_WLONGVARCHAR:

            value_is_null = query.getColumnNullable(col,
                (SQLWCHAR**) & unicode_val, &u_len);
            if (value_is_null) {
                cerr << "NCLOB value is NULL";
            } else {
                cerr << "NCLOB value length = " << u_len << endl;
                // do something with NCLOB value
            }
            break;

        case SQL_LONGVARBINARY:

            value_is_null = query.getColumnNullable(col,
                (void**) & binary_val, &b_len);
            if (value_is_null) {
                cerr << "BLOB value is NULL";
            } else {
                cerr << "BLOB value length = " << b_len << endl;
                // do something with BLOB value
            }
            break;

        case SQL_LONGVARCHAR:

            alfanum_val = (char*) malloc(LOB_COL_SIZE + 1);
            value_is_null = query.getColumnNullable(col,
alfanum_val);
            if (value_is_null) {
                cerr << "CLOB value is NULL";
            } else {
                cerr << "CLOB value length = " <<
strlen(alfanum_val) << endl;
                // do something with BLOB value
            }
            free(alfanum_val);
            break;

        default:
            break;
    }
}

num_rows++;
```

```
        cerr << "row " << num_rows << " fetched" << endl;
    }
    cerr << num_rows << " rows returned" << endl;
} catch (TTErr error) {
    cerr << "\nError" << err << endl;
}
query.Drop();
conn.Disconnect();
return 0;
}
```

Passthrough LOBs in TTClasses

Passthrough LOBs, which are LOBs in Oracle Database accessed through TimesTen, are exposed as TimesTen LOBs and are supported by TimesTen in much the same way that any TimesTen LOB is supported.

However, note the following:

- TimesTen LOB size limitations do not apply to storage of LOBs in the Oracle database through passthrough.
- As with TimesTen local LOBs, a passthrough LOB used in an application does not remain valid past the end of the transaction.

Setting a Timeout or Threshold for Executing SQL Statements

TimesTen offers two ways for you to limit the time for SQL statements or procedure calls to execute, by setting either a timeout value or a threshold value.

When you set a timeout value, if the timeout duration is reached, the statement stops executing and an error is thrown. A value of 0 indicates no timeout. When you set a threshold value, if the threshold is reached, a warning is written to the support log but execution continues. A value of 0 means no warnings.

The query timeout limit has effect only when a SQL statement is actively executing. A timeout does not occur during commit or rollback.

Use the `TTCmd` methods [setQueryTimeout\(\)](#) and [setQueryThreshold\(\)](#) to specify these settings for the `TTCmd` object. Note that these methods override the settings of the TimesTen connection attributes `SQLQueryTimeout` (or `SQLQueryTimeoutMsec`) and `QueryThreshold`, respectively. Each of these connection attributes has a default value of 0, for no timeout or no threshold.

There is also a [getQueryThreshold\(\)](#) method to read the current threshold setting.

In `TTClasses`, these features can be used only at the statement level, not the connection level.

See *Timeouts and Thresholds for Executing SQL Statements* in *Oracle TimesTen In-Memory Database C Developer's Guide*. For information about the relationship between timeout values, see *Choose SQL and PL/SQL Timeout Values* in *Oracle TimesTen In-Memory Database Operations Guide*.

 **Note:**

If both a lock timeout value and a SQL query timeout value are specified, the lesser of the two values causes a timeout first. Regarding lock timeouts, you can refer to `ttLockWait` (built-in procedure) or `LockWait` (general connection attribute) in *Oracle TimesTen In-Memory Database Reference*, or to `Check for Deadlocks and Timeouts` in *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide*.

Using Automatic Client Failover in a TTClasses Application

TTClasses does not have its own functionality for automatic client failover, but a TTClasses application can configure TimesTen automatic client failover in the same way that an ODBC application can.

This is discussed in ODBC Support for Automatic Client Failover in *Oracle TimesTen In-Memory Database C Developer's Guide*. For TimesTen Scaleout, also see Client Connection Failover in *Oracle TimesTen In-Memory Database Scaleout User's Guide*. For TimesTen Classic, see Using Automatic Client Failover in *Oracle TimesTen In-Memory Database Operations Guide*.

Using TTClasses Logging

TTClasses has a logging facility that allows applications to capture debugging information. TTClasses logging is associated with processes. You can enable logging for a specific process and produce a single output log stream for the process.

TTClasses supports different levels of logging information. See the example in [Acknowledging XLA Updates Without Using Transaction Boundaries](#) for more information about what is printed at each log level.

Log level `TTLOG_WARN` is very useful while developing a TTClasses application. It can also be appropriate for production applications because in this log level, database query plans are generated.

At the more verbose log levels (`TTLOG_INFO` and `TTLOG_DEBUG`), so much log data is generated that application performance is adversely affected. Do not use these log levels in a production environment.

Although TTClasses logging can print to either `stdout` or `stderr`, the best approach is to write directly to a TTClasses log file. The example below demonstrates how to print TTClasses log information at log level `TTLOG_WARN` into the `/tmp/ttclasses.log` output file.

**Note:**

TTClasses logging is disabled by default.

```
ofstream output;  
output.open("/tmp/ttclasses.log");  
TTGlobal::setLogStream(output);  
TTGlobal::setLogLevel(TTLog::TTLOG_WARN);
```

First-time users of TTClasses should spend a little time experimenting with TTClasses logging to see how errors are printed at log level `TTLOG_ERROR` and how much information is generated at log levels `TTLOG_INFO` and `TTLOG_DEBUG`.

See [TTGlobal Usage](#) for more information about using the `TTGlobal` class for logging.

Using XLA in TTClasses

This section discusses use of the Transaction Log API (XLA) in TimesTen Classic.

- [About TimesTen XLA](#)
- [Acknowledging XLA Updates Without Using Transaction Boundaries](#)
- [Acknowledging XLA Updates at Transaction Boundaries](#)
- [XLA System Privilege](#)

About TimesTen XLA

XLA is a set of functions that enable you to implement applications that monitor TimesTen for changes to specified database tables and receive real-time notification of these changes.

The primary purpose of XLA is as a high-performance, asynchronous alternative to triggers.

XLA returns notification of changes to specific tables in the database and information about the transaction boundaries for those database changes. This section shows how to acknowledge updates only at transaction boundaries (a common requirement for XLA applications), using one example that does not use and one example that does use transaction boundaries.

Important notes:

- As discussed in [Considerations when Using an ODBC Driver Manager \(Windows\)](#), XLA functionality cannot be used in an application connected to an ODBC driver manager.
- If an XLA bookmark becomes stuck, which can occur if an XLA application terminates unexpectedly or disconnects without first deleting its bookmark or disabling change tracking, there may be an excessive accumulation of transaction log files. This accumulation may result in file system space being filled. See [Monitoring Accumulation of Transaction Log Files](#) in *Oracle TimesTen In-Memory Database Operations Guide*.

Additional notes:

- You can subscribe to tables containing LOB columns, but information about the LOB value itself is unavailable.

- Columns containing LOBs are reported as empty (zero length) or null (if the value is actually `NULL`). In this way, you can tell the difference between a null column and a non-null column.
- An XLA reader cannot subscribe to a table that uses in-memory column-based compression.

See XLA and TimesTen Event Management in *Oracle TimesTen In-Memory Database C Developer's Guide*. In addition, the TTClasses sample applications, provided with TimesTen Classic Quick Start, include XLA applications. See [About TimesTen Quick Start and Sample Applications](#).

Acknowledging XLA Updates Without Using Transaction Boundaries

Inside the `HandleChange()` method, depending on whether the record is an insert, update, or delete, the appropriate method from among the following is called: `HandleInsert()`, `HandleUpdate()`, or `HandleDelete()`.

It is inside `HandleChange()` that you can access the flag that indicates whether the XLA record is the last record in a particular transaction. Thus there is no way in loop in the example in this section for the `HandleChange()` method to pass the information about the transaction boundary to the loop, so that this information can influence when to call `conn.ackUpdates()`.

This is not an issue under typical circumstances of only a few records per transaction. Usually only a few records are returned when you ask XLA to return at most 1000 records with a `fetchUpdatesWait()` call. XLA returns records as quickly as it can, and even if huge numbers of transactions are occurring in the database, you usually can pull the XLA records out quickly, a few at a time, and XLA makes sure that the last record returned is on a transaction boundary. For example, if you ask for 1000 records from XLA but only 15 are returned, it is highly probable that the 15th record is at the end of a transaction.

XLA guarantees one of the following:

- A batch of records ends with a completed transaction (perhaps multiple transactions in a single batch of XLA records).

Or:

- A batch of records contains a partial transaction, with no completed transactions in the same batch, and subsequent batches of XLA records are returned for that single transaction until its transaction boundary has been reached.

This example shows a typical main loop of a TTClasses XLA program. (It also assumes a signal handler is in place.)

```
TTXlaPersistConnection conn; // XLA connection
TTXlaTableList list(&conn); // tables being monitored
ttXlaUpdateDesc_t ** array; // pointer to returned XLA records
int records_fetched;
// ...

while (!signal_received) {
    // fetch the updates
    conn.fetchUpdatesWait(&array, MAX_RECS_TO_FETCH,
        &records_fetched, ...);
```

```

// Interpret the updates
for(j=0;j < records_fetched;j++){
    ttXlaUpdateDesc_t *p;
    p = array[j];
    list.HandleChange(p, NULL);
} // end for each record fetched

// periodically call ackUpdates()
if (/* some condition is reached */) {
    conn.ackUpdates();
}
}

```

Acknowledging XLA Updates at Transaction Boundaries

XLA applications should verify whether the last record in a batch of XLA records is at a transaction boundary, and call `ackUpdates()` only on transaction boundaries. This way, when the application or system or database fails, the XLA bookmark is at the start of a transaction after the system recovers.

This is especially important when operations involve a large number of rows. If a bulk insert, update, or delete operation has been performed on the database and the XLA application asks for 1000 records, it may or may not receive all 1000 records. The last record returned through XLA probably does *not* have the end-of-transaction flag. In fact, if the transaction has made changes to 10,000 records, then clearly a minimum of 10 blocks of 1000 XLA records must be fetched before reaching the transaction boundary.

Calling `ackUpdates()` for every transaction boundary is not recommended, however, because `ackUpdates()` is a relatively expensive operation. Users should balance overall system throughput with recovery time and file system space requirements. (Recall that a TimesTen transaction log file cannot be deleted by a checkpoint operation if XLA has a bookmark that references that log file. See `ttLogHolds` in *Oracle TimesTen In-Memory Database Reference*.) Depending on system throughput, recovery time, and file system space requirements, some applications may find it appropriate to call `ackUpdates()` once or several times per minute, while other applications may need only call it once or several times per hour.

The `HandleChange()` method has a second parameter to allow passing information between `HandleChange()` and the main XLA loop. Compare the example in the preceding section to the example below, specifically the `do_acknowledge` setting and the `&do_acknowledge` parameter of the `HandleChange()` call.

In this example, `ackUpdates()` is called only when the `do_acknowledge` flag indicates that this batch of XLA records is at a transaction boundary. (The example also assumes a signal handler is in place.)

```

TTXlaPersistConnection conn; // XLA connection
TTXlaTableList list(&conn); // tables being monitored
ttXlaUpdateDesc_t ** array; // ptr to returned XLA recs
int records_fetched;
int do_acknowledge;
int j;

// ...
while (!signal_received) {
    // fetch the updates

```

```

    conn.fetchUpdatesWait(&array, MAX_RECS_TO_FETCH,
&records_fetched, ...);

    do_acknowledge = FALSE;

    // Interpret the updates
    for(j=0;j < records_fetched;j++){
        ttXlaUpdateDesc_t *p;
        p = array[j];
        list.HandleChange(p, &do_acknowledge);
    } // end for each record fetched

    // periodically call ackUpdates()
    if (do_acknowledge == TRUE /* and some other conditions ... */ ) {
        conn.ackUpdates();
    }
}

```

In addition to this change to the XLA main loop, the `HandleChange()` method must be overloaded to have two parameters (`ttXlaUpdateDesc_t*`, `void* pData`). See [HandleChange\(\)](#). The TimesTen Classic Quick Start `xlasubscriber1` sample application shows the use of a `pData` parameter. (See [About TimesTen Quick Start and Sample Applications.](#))

XLA System Privilege

The system privilege `XLA` is required for any XLA functionality, such as connecting to TimesTen (which also requires the `CREATE SESSION` privilege) as an XLA reader, executing XLA-related TimesTen C functions, and executing XLA-related TimesTen built-in procedures.

Refer to XLA System Privilege in *Oracle TimesTen In-Memory Database C Developer's Guide*.

Note:

A user with the `XLA` privilege can be notified of any DML statement that executes in the database. As a result, the user with `XLA` privilege can obtain information about database objects that the user has not otherwise been granted access to. In practical terms, the `XLA` privilege is effectively the same as the `SELECT ANY TABLE`, `SELECT ANY VIEW`, and `SELECT ANY SEQUENCE` privileges.

Using the TTClasses Classes

This section provides additional information about how to use each class in the TTClasses library.

- [How to Use the Commonly Used Classes](#)
- [How to Use the System Catalog Classes](#)

- [How to Use the XLA Classes](#)

How to Use the Commonly Used Classes

This section describes how to use each of the commonly used classes in the TTClasses library.

- [TTGlobal Usage](#)
- [TTStatus Usage](#)
- [TTConnection Usage](#)
- [TTConnectionPool Usage](#)
- [TTCmd Usage](#)

TTGlobal Usage

The `TTGlobal` class provides a logging facility within TTClasses.

This logging facility can be very useful for debugging problems inside a TTClasses program. Note, however, that the most verbose logging levels (`TTLog::TTLOG_INFO` and `TTLog::TTLOG_DEBUG`) can generate an extremely large amount of output. Use these logging levels during development or when trying to diagnose a bug. They are not appropriate for most production environments.

When logging from a multithreaded program, you may encounter a problem where log output from different program threads is intermingled when written to the file system. To alleviate this problem, disable `ostream` buffering with the `ios_base::unitbuf` I/O stream manipulator, as in the following example, which sends TTClasses logging to the `app_log.txt` file at logging level `TTLog::TTLOG_ERR`.

```
ofstream log_file("app_log.txt");
log_file << std::ios_base::unitbuf;
TTGlobal::setLogStream(log_file);
TTGlobal::setLogLevel(TTLog::TTLOG_ERR);
```

See [Using TTClasses Logging](#) and [TTGlobal Reference](#) for more information about `TTGlobal`.

TTStatus Usage

The `TTStatus` class is used by other classes in the TTClasses library to catch error and warning exceptions. You can think of `TTStatus` as a value-added C++ wrapper around the `SQLError` ODBC function.

A `TTStatus` object is thrown as an exception whenever an error or warning occurs. This allows C++ applications to use `{try/catch}` blocks to detect and recover from failure.

```
...
TTCmd    myCmd;

try {
    myCmd.ExecuteImmediate(&conn, "create table dummy (c1 int)");
}
```



```

catch (TTStatus st) {
    cerr << "Error creating table: " << st << endl;
    // Rollback, exit(), throw -- whatever is appropriate
}
...

```

`TTStatus` has two subclasses, `TTErrors` and `TTWarning`. ODBC warnings (the Return Receipt warning, for example) are usually not as serious as ODBC errors and should typically be handled with different logic. ODBC errors should be handled programmatically. There may be circumstances where handling ODBC warnings programmatically is warranted, but it is usually sufficient to simply log them.

This example shows the use of `TTErrors` and `TTWarning`. `TTErrors` objects are thrown for ODBC errors. `TTWarning` objects are thrown for ODBC warnings.

```

// catching TTErrors & TTWarning exceptions

try {
    // some TTClasses method calls
}
catch (TTWarning warn) {
    cerr << "Warning encountered: " << warn << endl;
}
catch (TTErrors err) {
    // handle the error; this could be a serious problem
}

```

Note:

TimesTen automatically resolves most transient errors (which is particularly important for TimesTen Scaleout), but if an error detected by your application indicates a SQL state of `TT005` through the `odbc_error` attribute, it is suggested to retry the current transaction. See *Transient Errors (ODBC)* in *Oracle TimesTen In-Memory Database C Developer's Guide*.

Also see [TTStatus Reference](#).

TTConnection Usage

The `TTConnection` class encapsulates the concept of a connection to a database. You can think of `TTConnection` as a value-added C++ wrapper around the ODBC connection handle (`SQLHDBC`). All applications that use TimesTen must create at least one `TTConnection` object.

Multithreaded applications that use TimesTen from multiple threads simultaneously must create multiple `TTConnection` objects. Use one of the following strategies:

- Create one `TTConnection` object for each thread when the thread is created.
- Create a pool of `TTConnection` objects when the application process starts. They are shared by the threads in the process. See [TTConnectionPool Usage](#).

A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating (forking) a child process, the child cannot use the same connection. Any attempt by a child to use a database connection of a parent can cause application failure or a core dump.

Applications should not frequently make and then drop database connections, because connecting and disconnecting are both relatively expensive operations. In addition, short-lived connections eliminate the benefits of prepared statements. Instead, establish database connections at the beginning of the application process and reuse them for the life of the process.

**Tip:**

If you must manipulate the underlying ODBC connection object directly, use the `TTConnection::getHdbc()` method.

Privilege to connect to a database must be granted to users through the `CREATE SESSION` privilege, either directly or through the `PUBLIC` role. See [Connection Methods](#).

Also see [Using TTCmd](#), [TTConnection](#), and [TTConnectionPool](#). and [TTConnection Reference](#).

TTConnectionPool Usage

The `TTConnectionPool` class is used by multithreaded applications to manage a pool of connections.

In general, multithreaded applications can be written using one of the following strategies:

- If there is a relatively small number of threads and the threads are long-lived, each thread can be assigned to a different connection, which is used for the duration of the application. In this scenario, the `TTConnectionPool` class is not necessary.
- If there is a large number of threads in the process, or if the threads are short-lived, a pool of idle connections can be established. These connections are used for the duration of the application. When a thread must perform a database transaction, it checks out an idle connection from the pool, performs its transaction, then returns the connection to the pool. This is the scenario that the `TTConnectionPool` class assists with.

The constructor has two forms:

```
TTConnectionPool()
```

Or:

```
TTConnectionPool(const int size);
```

Where `size` specifies the maximum number of connections in a pool. Without specifying this, the maximum number of connections is 128. Note that if you specify the `size` setting, and you specify a value that is larger than the maximum number of connections according to the setting of the TimesTen `Connections` attribute, you will get an error when the number of connections exceeds the `Connections` value. Also see `Connections` in the *Oracle TimesTen In-Memory Database Reference*.

 **Tip:**

For best overall performance, TimesTen recommends having one or two concurrent direct connections to the database for each CPU of the database server. For no reason should your number of concurrent direct connections (the size of your connection pool) be more than twice the number of CPUs on the database server. For client/server connections, however, TimesTen supports many more connections per CPU efficiently.

To use the `TTConnectionPool` class, an application creates a single instance of the class. It then creates several `TTConnection` objects, instances of either the `TTConnection` class or a user class that extends it, but does not call their `Connect()` methods directly. Instead, the application uses the `TTConnectionPool::AddConnectionToPool()` method to place connection objects into the pool, then calls `TTConnectionPool::ConnectAll()` to establish all the connections to TimesTen. In the background, `ConnectAll()` loops through all the `TTConnection` objects to call their `Connect()` methods.

Threads for TimesTen applications use the `getConnection()` and `freeConnection()` methods to get and return idle connections.

 **Tip:**

If you want to use `TTConnectionPool` and extend `TTConnection`, do not override the `TTConnection::Connect()` method that has *driverCompletion* in the calling sequence, because there is no corresponding `TTConnectionPool::ConnectAll()` method. Instead, override either of the following `Connect()` methods:

```
virtual void Connect(const char* connStr)
virtual void Connect(const char* connStr, const char*
username,
                    const char* password)
```

Then use the appropriate corresponding `ConnectAll()` method.

Privilege to connect to a database must be granted to users through the `CREATE SESSION` privilege, either directly or through the `PUBLIC` role. See [Connection Methods](#).

Also see [Using TTCmd](#), [TTConnection](#), and [TTConnectionPool](#). and [TTConnectionPool Reference](#)

TTCmd Usage

A `TTCmd` object encapsulates a single SQL statement that is used multiple times in an application program. You can think of `TTCmd` as a value-added C++ wrapper around the ODBC statement handle (`SQLHSTMT`). `TTCmd` has three categories of public methods.

Each SQL statement executed multiple times in a program should have its own `TTCmd` object. Each of these `TTCmd` objects should be prepared once during program initialization, then executed with the `Execute()` method multiple times as the program runs.

Only database operations that are to be executed a small number of times should use the `ExecuteImmediate()` method. Note that `ExecuteImmediate()` is not compatible with any type of `SELECT` statement. All queries must use `Prepare()` plus `Execute()` instead.

`ExecuteImmediate()` is also incompatible with `INSERT`, `UPDATE`, or `DELETE` statements that are subsequently polled using `getRowcount()` to see how many rows were inserted, updated or deleted. These limitations have been placed on `ExecuteImmediate()` to discourage its use except in a few particular situations (for example, for creating or dropping a table).

 **Note:**

- Several `TTCmd` methods return an error if used with an ODBC driver manager. See [Considerations when Using an ODBC Driver Manager \(Windows\)](#) for information.
- If you have reason to manipulate the underlying ODBC statement object directly, use the `TTCmd::getHandle()` method.

Also see [Using TTMd, TTConnection, and TTConnectionPool](#). and [TTCmd Reference](#).

How to Use the System Catalog Classes

This section describes how to use each of the TTClasses system catalog classes.

- [TTCatalog Usage](#)
- [TTCatalogTable Usage](#)
- [TTCatalogColumn Usage](#)
- [TTCatalogIndex Usage](#)
- [TTCatalogSpecialColumn Usage](#)

TTCatalog Usage

The `TTCatalog` class is the top-level class used for programmatically accessing metadata information about tables in a database. You can use this class to facilitate reading metadata from the system catalog. A `TTCatalog` object contains data structures with the information that was read.

To use the `TTCatalog` object, call its `fetchCatalogData()` method. The `fetchCatalogData()` method is the only `TTCatalog` method that uses the database connection. All other methods simply return data retrieved by `fetchCatalogData()`.

A `TTCatalog` object contains an internal array of `TTCatalogTable` objects. Aside from the class constructor, all public methods of `TTCatalog` are used to gain read-only access to this `TTCatalogTable` array.

The `TTCatalog` constructor caches the `conn` parameter and initializes all the internal data structures appropriately.

```
TTCatalog (TTCConnection* conn)
```

The following ODBC functions are used inside `TTCatalog`:

- `SQLTables()`
- `SQLColumns()`
- `SQLSpecialColumns()`
- `SQLStatistics()`

Also see [TTCatalog Reference](#).

TTCatalogTable Usage

Each object of the top-level `TTCatalog` class internally contains an array of `TTCatalogTable` objects.

A `TTCatalogTable` object is retrieved through the `TTCatalog::getTable()` method and stores all metadata information about the columns and indexes of a table.

Each `TTCatalogTable` object contains an array of `TTCatalogColumn` objects and an array of `TTCatalogIndex` objects.

Also see [TTCatalogTable Reference](#).

TTCatalogColumn Usage

Each `TTCatalogTable` object contains an array of `TTCatalogColumn` objects and an array of `TTCatalogIndex` objects.

The `TTCatalogColumn` class is used to store all metadata information about a single column of a table. This table is represented by the `TTCatalogTable` object from which the column was retrieved through a `TTCatalogTable::getColumn()` call.

Also see [TTCatalogColumn Reference](#).

TTCatalogIndex Usage

Each `TTCatalogTable` object contains an array of `TTCatalogColumn` objects and an array of `TTCatalogIndex` objects.

The `TTCatalogIndex` class is used to store all metadata information about an index of a table. This table is represented by the `TTCatalogTable` object from which the index was retrieved through a `TTCatalogTable::getIndex()` call.

Also see [TTCatalogIndex Reference](#)

TTCatalogSpecialColumn Usage

Obtain a `TTCatalogSpecialColumn` object by calling the `getSpecialColumn()` method on the relevant `TTCatalogTable` object.

In TimesTen, a rowid pseudocolumn is the only type of special column supported, so a `TTCatalogSpecialColumn` object can only contain information about rowids.

Also see [TTCatalogSpecialColumn Reference](#).

How to Use the XLA Classes

This section introduces the TTClasses XLA classes then describes how to use each of them.

- [About the XLA Classes](#)
- [TTXlaPersistConnection Usage](#)
- [TTXlaRowViewer Usage](#)
- [TTXlaTableHandler Usage](#)
- [TTXlaTableList Usage](#)
- [TTXlaTable Usage](#)
- [TTXlaColumn Usage](#)

About the XLA Classes

TTClasses provides a set of classes for applications to use with the TimesTen Transaction Log API (XLA), which is supported by TimesTen Classic.

XLA is a set of C-callable functions that allow an application to monitor changes made to one or more database tables. Whenever another application changes a monitored table, the application using XLA is informed of the changes. For more information about XLA, see XLA and TimesTen Event Management in *Oracle TimesTen In-Memory Database C Developer's Guide*.

The XLA classes support as many XLA columns as the maximum number of columns supported by TimesTen. For more information, see System Limits in *Oracle TimesTen In-Memory Database Reference*.



Note:

As mentioned in [Considerations when Using an ODBC Driver Manager \(Windows\)](#), XLA functionality is not supported with TTClasses when you use a generic ODBC driver manager.

TTXlaPersistConnection Usage

Use `TTXlaPersistConnection` to create an XLA connection to a database.

An XLA application can create multiple `TTXlaPersistConnection` objects if needed. Each `TTXlaPersistConnection` object must be associated with its own bookmark, which is specified at connect time and must be maintained through the `ackUpdates()` and `deleteBookmarkAndDisconnect()` methods. Most applications require only one or two XLA bookmarks.

After an XLA connection is established, the application should enter a loop in which the `fetchUpdatesWait()` method is called repeatedly until application termination. This loop

should fetch updates from XLA as rapidly as possible to ensure that the transaction log does not fill up available file system space.

 **Note:**

- The transaction log is in a file system location according to the `TimesTen LogDir` attribute setting, if specified, or the `DataStore` attribute setting if `LogDir` is not specified. Refer to *Data Store Attributes in Oracle TimesTen In-Memory Database Reference*.
- Each bookmark establishes its own log hold on the transaction log. (See `ttLogHolds` in *Oracle TimesTen In-Memory Database Reference*.) If any bookmark is not moved forward periodically, transaction logs cannot be purged by checkpoint operations. This can fill up the file system over time.

After processing a batch of updates, the application should call `ackUpdates()` to acknowledge those updates and get ready for the next call to `fetchUpdatesWait()`. A batch of updates can be replayed using the `setBookmarkIndex()` and `getBookmarkIndex()` methods. Also, if the XLA application disconnects after `fetchUpdatesWait()` but before `ackUpdates()`, the next connection (with the same bookmark name) that calls `fetchUpdatesWait()` sees that same batch of updates.

Updates that occur while a `TTXlaPersistConnection` object is disconnected from the database are not lost. They are stored in the transaction log until another `TTXlaPersistConnection` object connects with the same bookmark name.

Privilege to connect to a database must be granted to users through the `CREATE SESSION` privilege, either directly or through the `PUBLIC` role. See [Connection Methods](#). In addition, the `XLA` privilege is required for XLA connections and functionality.

Also see [TTXlaPersistConnection Reference](#).

TTXlaRowViewer Usage

`TTXlaRowViewer`, which represents a row image from change notification records, is used to examine XLA change notification record structures and old and new column values.

Methods of this class are used to examine column values from row images contained in change notification records. Also see related information about the `TTXlaTable` class ([TTXlaTable Usage](#)).

Before a row can be examined, the `TTXlaRowViewer` object must be associated with a row using the `setTuple()` method, which is invoked inside the `TTXlaTableHandler::HandleInsert()`, `HandleUpdate()`, or `HandleDelete()` method, or by a user-written overloaded method. Columns can be checked for null values using the `isNull()` method. Non-null column values can be examined using the appropriate overloaded `Get()` method.

Also see [TTXlaRowViewer Reference](#).

TTXlaTableHandler Usage

The `TTXlaTableHandler` class provides methods that enable and disable change tracking for a table. Methods are also provided to handle update notification records from XLA.

`TTXlaTableHandler` is intended as a base class from which application developers write customized classes to process changes to a particular table.

The constructor associates the `TTXlaTableHandler` object with a particular table and initializes the `TTXlaTable` data member contained within the `TTXlaTableHandler` object:

```
TTXlaTableHandler(TTXlaPersistConnection& conn, const char* ownerP,  
                 const char* nameP)
```

Also see [TTXlaTable Usage](#).

Application developers can derive one or more classes from `TTXlaTableHandler` and can put most of the application logic in the `HandleInsert()`, `HandleDelete()`, and `HandleUpdate()` methods of that class.

One possible design is to derive multiple classes from `TTXlaTableHandler`, one for each table. Business logic to handle changes to customer data might be implemented in a `CustomerTableHandler` class, for example, while business logic to handle changes to order data might be implemented in an `OrderTableHandler` class.

Another possible design is to derive one or more generic classes from `TTXlaTableHandler` to handle various scenarios. For example, a generic class derived from `TTXlaTableHandler` could be used to publish changes using a publish/subscribe system.

See the `xlasubscriber1` and `xlasubscriber2` demos in the TimesTen Classic Quick Start for examples of classes that extend `TTXlaTableHandler`. (Refer to [About TimesTen Quick Start and Sample Applications](#).)

Also see [TTXlaTableHandler Reference](#).

TTXlaTableList Usage

The `TTXlaTableList` class provides a list of `TTXlaTableHandler` objects and is used to dispatch update notification events to the appropriate `TTXlaTableHandler` object.

By registering `TTXlaTableHandler` objects in a `TTXlaTableList` object, the process of fetching update notification records from XLA and dispatching them to the appropriate methods for processing can be accomplished using a loop.

When an update notification is received from XLA, the appropriate `HandleXxx()` method of the appropriate `TTXlaTableHandler` object is called to process the record.

For example, if an object of type `CustomerTableHandler` is handling changes to table `CUSTOMER`, and an object of type `OrderTableHandler` is handling changes to table `ORDERS`, the application should have both of these objects in a `TTXlaTableList` object. As XLA update notification records are fetched from XLA, they can be dispatched to the correct handler by a call to `TTXlaTableList::HandleChange()`.

The constructor has two forms:

```
TTXlaTableList (TTXlaPersistConnection* cP, unsigned int  
num_tbls_to_monitor)
```

Where `num_tbls_to_monitor` is the number of database objects to monitor.

Or:

```
TTXlaTableList (TTXlaPersistConnection* cP);
```

Where `cP` references the database connection to be used for XLA operations. This form of the constructor can monitor up to 150 database objects.

Also see [TTXlaTableList Reference](#).

TTXlaTable Usage

The `TTXlaTable` class encapsulates the metadata for a table being monitored for changes.

`TTXlaTable` acts as a metadata interface for the TimesTen `ttXlaTblDesc_t` C data structure. (See `ttXlaTblDesc_t` in *Oracle TimesTen In-Memory Database C Developer's Guide*.)

When a user application creates a class that extends `TTXlaTableHandler`, it typically calls `TTXlaTable::getColNumber()` to map a column name to its XLA column number. You can then use the column number as input to the `TTXlaRowViewer::Get()` method. This is shown in the `xlasubscriber2` demo in the TimesTen Classic Quick Start. (Refer to [About TimesTen Quick Start and Sample Applications](#).)

This class also provides useful metadata functions to return the name, owner, and number of columns in the table.

Also see [TTXlaTable Reference](#)

TTXlaColumn Usage

A `TTXlaColumn` object contains the metadata for a single column of a table being monitored for changes.

`TTXlaColumn` acts as a metadata interface for the TimesTen `ttXlaColDesc_t` C data structure. (See `ttXlaColDesc_t` in *Oracle TimesTen In-Memory Database C Developer's Guide*.) Information including the column name, type, precision, and scale can be retrieved.

Applications can associate a column with a `TTXlaColumn` object by using the `TTXlaRowViewer::getColumn()` method.

Also see [TTXlaColumn Reference](#).

3

Commonly Used Classes Reference

This reference chapter contains descriptions of commonly used TTClasses and their methods.

- [TTGlobal Reference](#)
- [TTStatus Reference](#)
- [TTConnection Reference](#)
- [TTConnectionPool Reference](#)
- [TTCmd Reference](#)

TTGlobal Reference

The `TTGlobal` class provides a logging facility within TTClasses.

- [Public Members](#)
- [Public Methods](#)

Also see [TTCGlobal Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTGlobal` public methods.

Public Methods Summary

Method	Description
disableLogging()	Disables TTClasses logging.
setLogLevel()	Specifies the verbosity level of TTClasses logging.
setLogStream()	Specifies where TTClasses logging information should be sent.
sqlhenv()	Returns the underlying ODBC environment object (type <code>SQLHENV</code>).

`disableLogging()`

```
static void disableLogging()
```

This method disables all TTClasses logging. Note that the following two statements are identical:

```
TTGlobal::disableLogging();
TTGlobal::setLogLevel(TTLog::TTLOG_NIL);
```

setLogLevel()

```
static void setLogLevel(TTLog::TTLOG_LEVEL level)
```

This method specifies the verbosity level of TTClasses logging. [Table 3-1](#) describes TTClasses logging levels. The levels are cumulative.

Table 3-1 TTClasses Logging Levels

Logging level	Description
TTLog::TTLOG_NIL	There is no logging.
TTLog::TTLOG_FATAL_ERR	Logs fatal errors (serious misuse of TTClasses methods).
TTLog::TTLOG_ERR	Logs all errors, such as SQL_ERROR return codes.
TTLog::TTLOG_WARN	(Default) Also logs warnings and all calls to <code>TTCmd::Prepare()</code> , including the SQL string being prepared. Prints all database optimizer query plans.
TTLog::TTLOG_INFO	Also logs informational messages, such as calls to most methods on <code>TTCmd</code> and <code>TTConnection</code> objects, including the SQL string where appropriate.
TTLog::TTLOG_DEBUG	Also logs debugging information, such as all bound parameter values for each call to <code>TTCmd::Execute()</code> .

To set the logging level to `TTLog::TTLOG_ERR`, for example, add the following line to your program:

```
TTGlobal::setLogLevel(TTLog::TTLOG_ERR);
```

setLogStream()

```
static void setLogStream(ostream& stream)
```

Specifies the file (`ofstream` object) where TTClasses logging information should be sent. By default, if TTClasses logging is enabled, logging is to `stderr`. Using this method, an application can specify logging to a file (or any other `ostream&`), such as in the following example that sets logging to `app_log.txt`:

```
ofstream log_file("app_log.txt");
TTGlobal::setLogStream(log_file);
```

sqlhenv()

```
static SQLHENV sqlhenv()
```

Retrieves the underlying ODBC environment object.

TTStatus Reference

The `TTStatus` class is used by other classes in the `TTClasses` library to catch error and warning exceptions. You can think of `TTStatus` as a value-added C++ wrapper around the `SQLError` ODBC function.

- [Subclasses](#)
- [Public Members](#)
- [Public Methods](#)

Also see [TTCStatus Usage](#).

Subclasses

`TTStatus` has these subclasses:

Subclass	Description
<code>TTErrror</code>	<code>TTErrror</code> is used to encapsulate ODBC errors (return codes <code>SQL_ERROR</code> and <code>SQL_INVALID_HANDLE</code>).
<code>TTWarning</code>	<code>TTWarning</code> is used to encapsulate ODBC warnings (return code <code>SQL_SUCCESS_WITH_INFO</code>).

See [TTCStatus Usage](#) for additional information about `TTErrror` and `TTWarning`.

Public Members

Member	Description
<code>rc</code>	Return code from the failing ODBC call: <code>SQL_SUCCESS</code> , <code>SQL_SUCCESS_WITH_INFO</code> , <code>SQL_ERROR</code> , <code>SQL_NO_DATA_FOUND</code> , or <code>SQL_INVALID_HANDLE</code> (<code>SQL_NO_DATA_FOUND</code> , an ODBC 2.0 return code, is defined in <code>sqlext.h</code> , which is included by <code>timesten.h</code> .)
<code>native_error</code>	TimesTen native error number (if any) for the failing ODBC call
<code>odbc_error</code>	ODBC error state for the failing ODBC call
<code>err_msg</code>	ASCII printable error message for the failing ODBC call

Public Methods

This section summarizes then describes the `TTStatus` public methods.

Public Methods Summary

Method	Description
isConnectionInvalid()	Indicates whether the database connection is invalid.

Method	Description
operator<<	Outputs the error message.
throwError()	Throws an error from the <code>TTStatus</code> object (not typical use).

isConnectionInvalid()

```
bool isConnectionInvalid() const
```

Returns `TRUE` if the database connection is invalid, or `FALSE` if it is valid. Specifically, "invalid" refers to situations when a TimesTen error 846 or 994 is encountered. See Errors 0 - 999 in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

operator<<

The operator (`<<`) writes the error message to an output stream. Following is an example.

```
try {
    // ...
    // something has gone wrong
    throw stat;
}
catch (TTStatus st) {
    cerr << "Caught exception: " << st << endl;
}
```

throwError()

```
void throwError()
```

This is an alternative, but not typical, way to throw an exception. (The more typical usage is shown in the preceding `operator<<` section.)

```
try {
    // ...
    if (/* something has gone wrong */)
        stat.throwError();
}
catch (TTStatus st) {
    cerr << "Caught exception: " << st << endl;
}
```

TTConnection Reference

The `TTConnection` class encapsulates the concept of a connection to a database. You can think of `TTConnection` as a value-added C++ wrapper around the ODBC connection handle (`SQLHDBC`).

- [Public Members](#)

- [Public Methods](#)

Also see [TTConnection Usage](#).

Public Members

Member	Description
<code>DRIVER_COMPLETION_ENUM</code>	<p>Specifies whether there is a prompt for the database to connect to (also depending on whether a database is specified in the connect string).</p> <p>Valid values are <code>TTConnection::DRIVER_NOPROMPT</code>, <code>TTConnection::DRIVER_COMPLETE</code>, <code>TTConnection::DRIVER_PROMPT</code>, and <code>TTConnection::DRIVER_COMPLETE_REQUIRED</code>. These correspond to the values <code>SQL_DRIVER_NOPROMPT</code>, <code>SQL_DRIVER_COMPLETE</code>, <code>SQL_DRIVER_PROMPT</code>, and <code>SQL_DRIVER_COMPLETE_REQUIRED</code> for the standard ODBC <code>SQLDriverConnect</code> function.</p>

Public Methods

This section summarizes then describes the `TTConnection` public methods.

Public Methods Summary

Method	Description
Commit()	Commits a transaction to the database.
Connect()	Opens a new database connection.
Disconnect()	Closes a database connection.
DurableCommit()	Performs a durable commit operation on the database.
getHdbc()	Returns the ODBC connection handle (type <code>SQLHDBC</code>) associated with this connection.
GetTTContext()	Returns the connection context value.
isConnected()	Returns <code>TRUE</code> if the object is connected to TimesTen.
Rollback()	Rolls back changes made to the database through this connection since the last call to <code>Commit()</code> or <code>Rollback()</code> .
SetAutoCommitOff()	Disables autocommit for the connection.
SetAutoCommitOn()	Enables autocommit for the connection.
SetIsoReadCommitted()	Sets the transaction isolation level of the connection to be <code>TXN_READ_COMMITTED</code> .
SetIsoSerializable()	Sets the transaction isolation level of the connection to be <code>TXN_SERIALIZABLE</code> .
SetLockWait()	Sets the lock timeout interval for the connection by calling the <code>ttLockWait</code> TimesTen built-in procedure.
SetPrefetchCloseOff()	Turns off the <code>TT_PREFETCH_CLOSE</code> connection option.

Method	Description
SetPrefetchCloseOn()	Turns on the <code>TT_PREFETCH_CLOSE</code> connection option. This is useful for optimizing <code>SELECT</code> query performance for client/server connections to TimesTen.
SetPrefetchCount()	Allows a user application to tune the number of rows that the TimesTen ODBC driver <code>SQLFetch</code> call prefetches for a <code>SELECT</code> statement.

Commit()

```
void Commit()
```

Commits a transaction to the database. This commits all operations performed on the connection since the last call to the `Commit()` or `Rollback()` method. A `TTStatus` object is thrown as an exception if an error occurs. Also see [Rollback\(\)](#).

Connect()

```
virtual void Connect(const char* connStr)
virtual void Connect(const char* connStr, const char* username,
                    const char* password)
virtual void Connect(const char* connStr, DRIVER_COMPLETION_ENUM
                    driverCompletion)
```

Opens a new database connection. The connection string specified in the `connStr` parameter is used to create the connection. Specify a user and password, either as part of the connect string or as separate parameters, or a `DRIVER_COMPLETION_ENUM` value (refer to [Public Members](#)). Also see the following method, `Disconnect()`.

Privilege to connect to a database must be granted to users through the `CREATE SESSION` privilege, either directly or through the `PUBLIC` role. See [Connection Methods](#).

A `TTStatus` object is thrown as an exception if an error occurs. Any exception warnings are usually informational and can often be safely ignored. The following logic is preferred for use of the `Connect()` method.

`TTWarning` and `TTError` are subclasses of `TTStatus`.

```
TTConnection conn;
...

try {
    conn.Connect("DSN=mydsn", "myuser", "password");
}
catch (TTWarning warn) {
    // warnings from Connect() are usually informational
    cerr << "Warning while connecting to TimesTen: " << warn << endl;
}
catch (TTError err) {
    // handle the error; this could be a serious problem
}
```

Disconnect()

```
void Disconnect()
```

Closes a database connection. A `TTStatus` object is thrown as an exception if an error occurs. Also see the preceding method, `Connect()`.

DurableCommit()

```
void DurableCommit()
```

Performs a durable commit operation on the database. A durable commit operation flushes the in-memory transaction log buffer to the file system. It calls the `ttDurableCommit` TimesTen built-in procedure.

See `ttDurableCommit` in *Oracle TimesTen In-Memory Database Reference*.

getHdbc()

```
SQLHDBC getHdbc()
```

Returns the ODBC connection handle associated with this connection.

GetTTContext()

```
void GetTTContext(char* output)
```

Returns the context value of the connection, a value that is unique for each database connection. The context of a connection can be used to correlate TimesTen connections with PIDs (process IDs) using the `ttStatus` TimesTen utility, for example.

The context value is returned through the `output` parameter, which requires an array of `CHAR[17]` or larger.

This method calls the `ttContext` TimesTen built-in procedure. See `ttContext` in *Oracle TimesTen In-Memory Database Reference*.

isConnected()

```
bool isConnected()
```

Returns `TRUE` if the object is connected to TimesTen using the `Connect()` method or `FALSE` if not.

Rollback()

```
void Rollback()
```

Rolls back (cancels) a transaction. This undoes any changes made to the database through the connection since the last call to `Commit()` or `Rollback()`. A `TTStatus` object is thrown as an exception if an error occurs. Also see [Commit\(\)](#).

SetAutoCommitOff()

```
void SetAutoCommitOff()
```

Disables autocommit for the connection. Also see the following method, `SetAutoCommitOn()`.

This method is automatically called by `TTConnection::Connect()`, because TimesTen runs with optimal performance only with autocommit disabled.

Note that when autocommit is disabled, committing `SELECT` statements requires explicit calls to `TTCmd::Close()`.

SetAutoCommitOn()

```
void SetAutoCommitOn()
```

Enables autocommit for the connection, which means that every SQL statement occurs in its own transaction. Also see the preceding method, `SetAutoCommitOff()`.

`SetAutoCommitOn()` is generally not advisable, because TimesTen runs much faster with autocommit disabled.

SetIsoReadCommitted()

```
void SetIsoReadCommitted()
```

Sets the transaction isolation level of the connection to be `TXN_READ_COMMITTED`. The Read Committed isolation level offers the best combination of single-transaction performance and good multiconnection concurrency. Also see the following method, `SetIsoSerializable()`.

SetIsoSerializable()

```
void SetIsoSerializable()
```

Sets the transaction isolation level of the connection to be `TXN_SERIALIZABLE`. In general, Serializable isolation level offers fair individual transaction performance but extremely poor concurrency. Read Committed isolation level is preferable over Serializable isolation level in almost all situations. Also see the preceding method, `SetIsoReadCommitted()`.

SetLockWait()

```
void SetLockWait(int secs)
```

Sets the lock timeout interval for the connection by calling the `ttLockWait` TimesTen built-in procedure with the `secs` parameter. In general, a two-second or three-second lock timeout is sufficient for most applications. The default lock timeout interval is 10 seconds.

See `ttLockWait` in *Oracle TimesTen In-Memory Database Reference*.

SetPrefetchCloseOff()

```
void SetPrefetchCloseOff()
```

Turns off the `TT_PREFETCH_CLOSE` connection option. Also see the following method, `SetPrefetchCloseOn()`.

SetPrefetchCloseOn()

```
void SetPrefetchCloseOn()
```

Turns on the `TT_PREFETCH_CLOSE` connection option, which is useful for optimizing `SELECT` query performance for serializable transactions in client/server applications. Note that this method provides no benefit for an application using a direct connection to TimesTen. Also see the preceding method, `SetPrefetchCloseOff()`.

See *Optimizing Query Performance in Oracle TimesTen In-Memory Database C Developer's Guide*

SetPrefetchCount()

```
void SetPrefetchCount(int numRows)
```

Allows a client/server application to tune the number of rows that the TimesTen ODBC driver internally fetches at a time for a `SELECT` statement. The value of `numRows` must be between 1 and 128, inclusive.

Note that this method provides no benefit for an application using a direct connection to TimesTen.



Note:

This method is not equivalent to executing `TTCmd::FetchNext()` multiple times. Instead, proper use of this parameter reduces the amount of time for each call to `TTCmd::FetchNext()`.

See *Prefetching Multiple Rows of Data in Oracle TimesTen In-Memory Database C Developer's Guide* for more information about `TT_PREFETCH_COUNT`.

TTConnectionPool Reference

The `TTConnectionPool` class is used by multithreaded applications to manage a pool of connections.

- [Public Members](#)
- [Public Methods](#)

Also see [TTConnectionPool Usage](#)

Public Members

None

Public Methods

This section summarizes then describes the `TTConnectionPool` public methods.

Public Methods Summary

Method	Description
AddConnectionToPool()	Adds a <code>TTConnection</code> object (possibly an object of a class derived from <code>TTConnection</code>) to the connection pool.
ConnectAll()	Connects all the <code>TTConnection</code> objects to TimesTen simultaneously.
DisconnectAll()	Disconnects all connections in the connection pool from TimesTen.
freeConnection()	Returns a connection to the pool for reassignment to another thread.
getConnection()	Checks out an idle connection from the connection pool for a thread.
getStats()	Queries the <code>TTConnectionPool</code> object for connection pool status information.

AddConnectionToPool()

```
int AddConnectionToPool(TTConnection* connP)
```

This method is used to add a `TTConnection` object (possibly an object of a class derived from `TTConnection`) to the connection pool. It returns -1 if there is an error. Also see [freeConnection\(\)](#).

ConnectAll()

```
void ConnectAll(const char* connStr)
void ConnectAll(const char* connStr, const char* username, const char*
password)
```

After all the `TTConnection` objects of an application have been added to the connection pool by `AddConnectionToPool()`, the `ConnectAll()` method can be used to connect all of the `TTConnection` objects to TimesTen simultaneously. The connection string specified in the `connStr` parameter is used to create the connection. Specify a user and password, either as part of the connect string or as separate parameters. Also see the next method, `DisconnectAll()`.

A `TTStatus` object is thrown as an exception if an error occurs.

Privilege to connect to a database must be granted to users through the `CREATE SESSION` privilege, either directly or through the `PUBLIC` role. See [Connection Methods](#).

DisconnectAll()

```
void DisconnectAll()
```

Disconnects all connections in the connection pool from TimesTen. Also see the preceding method, `ConnectAll()`.

Applications must call `DisconnectAll()` before termination to avoid overhead associated with process failure analysis and recovery. A `TTStatus` object is thrown as an exception if an error occurs.

freeConnection()

```
void freeConnection(TTConnection* connP)
```

Returns a connection to the pool for reassignment to another thread. Applications should not free connections that are in the middle of a transaction. `TTConnection::Commit()` or `Rollback()` should be called immediately before the `TTConnection` object is passed to `freeConnection()`. Also see [AddConnectionToPool\(\)](#).

getConnection()

```
TTConnection* getConnection(int timeout_millis=0)
```

Checks out an idle connection from the connection pool for use by a thread. A pointer to an idle `TTConnection` object is returned. The thread should then perform a transaction, ending with either `Commit()` or `Rollback()`, and then should return the connection to the pool using the `freeConnection()` method.

If no idle connections are in the pool, the thread calling `getConnection()` blocks until a connection is returned to the pool by a call to `freeConnection()`. An optional timeout, in milliseconds, can be provided. If this is provided, `getConnection()` waits for a free connection for no more than `timeout_millis` milliseconds. If no connection is available in that time then `getConnection()` returns `NULL` to the caller.

getStats()

```
void getStats(int* nGets, int* nFrees, int* nWaits, int* nTimeouts,  
             int* maxInUse, int* nForcedCommits)
```

Queries the `TTConnectionPool` for status information. The following data are returned:

- `nGets`: Number of calls to `getConnection()`
- `nFrees`: Number of calls to `freeConnection()`
- `nWaits`: Number of times a call to `getConnection()` had to wait before returning a connection
- `nTimeouts`: Number of calls to `getConnection()` that timed out
- `maxInUse`: High point for the most number of connections in use simultaneously
- `nForcedCommits`: Number of times that `freeConnection()` had to call `Commit()` on a connection before checking it into the pool

If this counter is nonzero, the user application is not calling `TTConnection::Commit()` or `Rollback()` before returning a connection to the pool.

TTCmd Reference

A `TTCmd` object encapsulates a single SQL statement that is used multiple times in an application program. You can think of `TTCmd` as a value-added C++ wrapper around the ODBC statement handle (`SQLHSTMT`). `TTCmd` has three categories of public methods.

- [Public Members](#)
- [Public Methods for General Use and Non-Batch Operations](#)
- [Public Methods for Obtaining TTCmd Object Properties](#)
- [Public Methods for Batch Operations](#)

Also see [TTCmd Usage](#).

Public Members

Member	Description
<code>TTCMD_PARAM_INPUTOUTPUT_TYPE</code>	This is used to specify whether a parameter is input, output, or input/output when registering the parameter. Supported values are <code>PARAM_IN</code> , <code>PARAM_INOUT</code> , and <code>PARAM_OUT</code> . See Registering Parameters .

Public Methods for General Use and Non-Batch Operations

This section summarizes then describes the `TTCmd` public methods for general use and non-batch operations.

Public Methods Summary for General Use and Non-Batch Operations

Method	Description
Close()	Closes the result set when the application has finished fetching rows.
Drop()	Frees a prepared SQL statement and all resources associated with it.
Execute()	Invokes a SQL statement that has been prepared for execution.
ExecuteImmediate()	Invokes a SQL statement that has not been previously prepared.
FetchNext()	Fetches rows from the result set, one at a time. It returns 0 when a row was successfully fetched or 1 when no more rows are available.
getColumn()	Retrieves the value in the specified column of the current row of the result set.
getColumnLength()	Returns the length of the specified column, in bytes.
getColumnNullable()	Retrieves the value in the specified column of the current row of the result set and returns a boolean to indicate whether the value is <code>NULL</code> .

Method	Description
getHandle()	Retrieves the underlying ODBC statement handle.
getMaxRows()	Returns the current limit on the number of rows returned by a <code>SELECT</code> statement.
getNextColumn()	Retrieves the value in the next column of the current row of the result set.
getNextColumnNullable()	Retrieves the value in the next column of the current row of the result set and returns a boolean to indicate whether the value is <code>NULL</code> .
getParam()	Each call gets the output value of a specified output or input/output parameter after executing a prepared SQL statement.
getQueryThreshold()	Retrieves the query threshold value.
getRowCount()	Returns the number of rows that were affected by the recently executed SQL operation.
isColumnNull()	Indicates whether the value in the specified column of the current row is <code>NULL</code> .
Prepare()	Associates a SQL statement with the <code>TTCmd</code> object.
printColumn()	Prints the value in the specified column of the current row to an output stream.
registerParam()	Registers a parameter for binding. This is required for output or input/output parameters.
setMaxRows()	Sets a limit on the number of rows returned by a <code>SELECT</code> statement.
setParam()	Each call sets the value of a specified parameter before executing a prepared SQL statement.
setParamLength()	Sets the length, in bytes, of the specified input parameter.
setParamNull()	Sets the value of a parameter to <code>NULL</code> before executing a prepared SQL statement.
setQueryThreshold()	Sets a threshold time limit for execution of each SQL statement. If it is exceeded, a warning is written to the support log.
setQueryTimeout()	Sets a timeout value for SQL statements.

Close()

```
void Close()
```

If a SQL `SELECT` statement is executed using the `Execute()` method, a cursor is opened which may be used to fetch rows from the result set. When the application is finished fetching rows from the result set, it must be closed with the `Close()` method.

Failure to close the result set may result in locks being held on rows for too long, causing concurrency problems, memory leaks, and other errors.

A `TTStatus` object is thrown as an exception if an error occurs.

Drop()

```
void Drop()
```

If a prepared SQL statement will not be used in the future, the statement and resources associated with it can be freed by a call to the `Drop()` method. The `TTCmd` object may be reused for another statement if `Prepare()` is called again.

It is more efficient to use multiple `TTCmd` objects to execute multiple SQL statements. Use the `Drop()` method only if a particular SQL statement will not be used again.

A `TTStatus` object is thrown as an exception if an error occurs.

Execute()

```
void Execute()
```

This method invokes a SQL statement that has been prepared for execution with the `Prepare()` method, after any necessary parameter values are defined using `setParam()` calls. A `TTStatus` object is thrown as an exception if an error occurs.

If the SQL statement is a `SELECT` statement, this method executes the query but does not return any rows from the result set. Use the `FetchNext()` method to fetch rows from the result set one at a time. Use the `Close()` method to close the result set when all appropriate rows have been fetched. For SQL statements other than `SELECT`, no cursor is opened, and a call to the `Close()` method is not necessary.

ExecuteImmediate()

```
int ExecuteImmediate(TTConnection* cP, const char* sqlp)
```

This method invokes a SQL statement that has not been previously prepared.

`ExecuteImmediate()` is a convenient alternative to using `Prepare()` and `Execute()` when a SQL statement is to be executed only a small number of times. Use `ExecuteImmediate()` for DDL statements such as `CREATE TABLE` and `DROP TABLE`, and infrequently used DML statements that do not return a result set (for example, `DELETE FROM table_name`).

`ExecuteImmediate()` is incompatible with SQL statements that return a result set. In addition, statements executed through `ExecuteImmediate()` cannot subsequently be queried by `getRowCount()` to get the number of rows affected by a DML operation. Because of this, `ExecuteImmediate()` calls `getRowCount()` automatically, and its value is the integer return value of this method.

A `TTStatus` object is thrown as an exception if an error occurs.

FetchNext()

```
int FetchNext()
```

After executing a prepared SQL `SELECT` statement using the `Execute()` method, use the `FetchNext()` method to fetch rows from the result set, one at a time.

After fetching a row of the result set, use the appropriate overloaded `getColumn()` method to fetch values from the current row.

If no more rows remain in the result set, `FetchNext()` returns 1. If a row is returned, `FetchNext()` returns 0.

After executing a `SELECT` statement using the `Execute()` method, the result set must be closed using the `Close()` method after all desired rows have been fetched. Note that after the `Close()` method is called, the `FetchNext()` method cannot be used to fetch additional rows from the result set.

A `TTStatus` object is thrown as an exception if an error occurs.

`getColumn()`

```
void getColumn (int cno, TYPE* valueP)
void getColumn (int cno, TYPE* valueP, SQLLEN* byteLenP)
```

The `getColumn()` method, as well as the `getColumnNullable()` method, fetches the values for columns of the current row of the result set. Before `getColumn()` or `getColumnNullable()` can be called, the `FetchNext()` method must be called to fetch the next (or first) row from the result set of a `SELECT` statement. SQL statements are executed using the `Execute()` method.

Table 3-2 below shows the supported TimesTen column types and the appropriate versions of `getColumn()` and `getColumnNullable()` to use for each parameter type. (But note that `getColumnNullable()` also returns a boolean value, which is not indicated in the table. Refer to the `getColumnNullable()` method documentation later in this section.)

Each `getColumn()` call retrieves the value associated with a particular column. Columns are referred to by ordinal number, with "1" indicating the first column specified in the `SELECT` statement. In all cases the first argument passed to the `getColumn()` method, `cno`, is the ordinal number of the column whose value is to be fetched. The second argument, `valueP`, is a pointer to a variable that stores the value of the specified column. The type of this argument varies depending on the type of the column being returned. For `NCHAR`, `NVARCHAR`, and binary types, as shown in the table, the method call also specifies `byteLenP`, a pointer to an integer value for the number of bytes written into the `valueP` buffer.

The `TTClasses` library does not support a large set of data type conversions. The appropriate version of `getColumn()` must be called for each output column in the prepared SQL. Calling the wrong version, such as attempting to fetch an integer column into a `char*` value, results in a thrown exception (`TTStatus` object).

When fetching integer-type data from `NUMBER` columns, `getColumn()` supports the following variants: `SQLTINYINT`, `SQLSMALLINT`, `SQLINTEGER`, and `SQLBIGINT`. They are appropriate only for `NUMBER` fields with the scale parameter set to zero, such as `NUMBER(p)` or `NUMBER(p,0)`. The functions have the following range of precision.

Function	Precision Range
<code>SQLTINYINT</code>	$0 \leq p \leq 2$
<code>SQLSMALLINT</code>	$0 \leq p \leq 4$
<code>SQLINTEGER</code>	$0 \leq p \leq 9$
<code>SQLBIGINT</code>	$0 \leq p \leq 18$

To ensure that all values in the column fit into the variable that the application uses to retrieve information from the database, you can use `SQLBIGINT` for all table columns of data type `NUMBER(p)`, where $0 \leq p \leq 18$. For example:

```
getColumn(int cno, SQLBIGINT* iP)
```

This table shows the supported SQL data types and the appropriate input signatures of `getColumn` and `getColumnNullable` to use for each data type. The data type support also applies to `getNextColumn`, `getNextColumnNullable`, and `getParam`.

Table 3-2 `getColumn()` and `getColumnNullable()` Input Signatures for Supported TimesTen Table Column Types

Data Type	<code>getColumn()</code> and <code>getColumnNullable()</code> Input Signatures Supported
TT_TINYINT	<code>getColumn[Nullable](int cno, SQLTINYINT* iP)</code>
TT_SMALLINT	<code>getColumn[Nullable](int cno, SQLSMALLINT* iP)</code>
TT_INTEGER	<code>getColumn[Nullable](int cno, SQLINTEGER* iP)</code>
TT_BIGINT	<code>getColumn[Nullable](int cno, SQLBIGINT* iP)</code>
BINARY_FLOAT	<code>getColumn[Nullable](int cno, float* fP)</code>
BINARY_DOUBLE	<code>getColumn[Nullable](int cno, double* dP)</code>
NUMBER	<pre>getColumn[Nullable](int cno, char** cPP) getColumn[Nullable](int cno, char* cP) getColumn[Nullable](int cno, SQLTINYINT* iP) getColumn[Nullable](int cno, SQLSMALLINT* iP) getColumn[Nullable](int cno, SQLINTEGER* iP) getColumn[Nullable](int cno, SQLBIGINT* iP)</pre>

Note: The `char*` version allows TTClasses to pass in an array of preallocated storage, and TTClasses copies the `char` output fetched from the database into this array. The integer type methods are appropriate only for columns declared with the scale parameter set to zero.

Table 3-2 (Cont.) getColumn() and getColumnNullable() Input Signatures for Supported TimesTen Table Column Types

Data Type	getColumn() and getColumnNullable() Input Signatures Supported
TT_CHAR CHAR TT_VARCHAR VARCHAR2	getColumn[Nullable](int <i>cno</i> , char** <i>cPP</i>) getColumn[Nullable](int <i>cno</i> , char* <i>cP</i>) Note: The <i>char*</i> version enables you to preallocate the output buffer.
TT_NCHAR NCHAR TT_NVARCHAR NVARCHAR2	getColumn[Nullable](int <i>cno</i> , SQLWCHAR** <i>wcPP</i>) getColumn[Nullable](int <i>cno</i> , SQLWCHAR** <i>wcPP</i> , SQLLEN* <i>byteLenP</i>) Note: Optionally use the <i>byteLenP</i> parameter for the number of bytes in the returned value.
BINARY VARBINARY	getColumn[Nullable](int <i>cno</i> , void** <i>binPP</i> , SQLLEN* <i>byteLenP</i>) getColumn[Nullable](int <i>cno</i> , void* <i>binP</i> , SQLLEN* <i>byteLenP</i>) Note: The <i>void*</i> version enables you to preallocate the output buffer.
DATE TT_TIMESTAMP TIMESTAMP	getColumn[Nullable](int <i>cno</i> , TIMESTAMP_STRUCT* <i>tsP</i>)
TT_DATE	getColumn[Nullable](int <i>cno</i> , DATE_STRUCT* <i>dP</i>)
TT_TIME	getColumn[Nullable](int <i>cno</i> , TIME_STRUCT* <i>tP</i>)

Other TimesTen table column types are not supported in this release of the TTCclasses library.

getColumnLength()

```
SQLULEN getColumnLength(int cno)
```

Returns the length, in bytes, of the value in column number *cno* of the current row, not counting the NULL terminator. Or it returns `SQL_NULL_DATA` if the value is NULL. (For those familiar with ODBC, this is the value stored by ODBC in the last parameter, *pcbValue*, from `SQLBindCol` after a call to `SQLFetch`.) When there is a non-null value, the length returned is between 0 and the column precision, inclusive. See [getColumnPrecision\(\)](#).

For example, assume a `VARCHAR2(25)` column. If the value is null, the length returned is -1. If the value is 'abcde', the length returned is 5.

This method is generally useful only when accessing columns of type CHAR, VARCHAR2, NCHAR, NVARCHAR2, BINARY, and VARBINARY.

getColumnNullable()

```
bool getColumnNullable(int cno, TYPE* valueP)
bool getColumnNullable(int cno, TYPE* valueP, SQLLEN* byteLenP)
```

The `getColumnNullable()` method is similar to the `getColumn()` method and supports the same data types and signatures as documented in [Table 3-2](#) above. However, in addition to the behavior of `getColumn()`, the `getColumnNullable()` method returns a boolean indicating whether the value is the SQL NULL pseudo-value. If the value is NULL, the second parameter is set to a distinctive value (for example, -9999) and the return value from the method is TRUE. If the value is not NULL, it is returned through the variable pointed to by the second parameter and the `getColumnNullable()` method returns FALSE.

getHandle()

```
SQLHSTMT getHandle()
```

If you must manipulate the underlying ODBC statement object, use this method to retrieve the statement handle.

getMaxRows()

```
SQLULEN getMaxRows()
```

This method returns the current limit of the number of rows returned by a SELECT statement from this TTCmd object. A return value of 0 means all rows are returned. Also see [setMaxRows\(\)](#).

getNextColumn()

```
void getNextColumn(TYPE* valueP)
void getNextColumn(TYPE* valueP, SQLLEN* byteLenP)
```

The `getNextColumn()` method, as well as the `getNextColumnNullable()` method, fetches the value of the next column of the current row of the result set. Before `getNextColumn()` or `getNextColumnNullable()` can be called, the `FetchNext()` method must be called to fetch the next (or first) row from the result set of a SELECT statement. When you use `getNextColumn()`, the columns are fetched in order. You cannot change the fetch order.

See [Table 3-2](#) for the supported SQL data types and the appropriate method version to use for each data type. This information can be used for `getNextColumn()`, except there is no `cno` column number parameter for `getNextColumn()`.

getNextColumnNullable()

```
bool getNextColumnNullable(TYPE* valueP)
bool getNextColumnNullable(TYPE* valueP, SQLLEN* byteLenP)
```

The `getNextColumnNullable()` method is similar to the `getNextColumn()` method. However, in addition to the behavior of `getNextColumn()`, the `getNextColumnNullable()` method returns a boolean indicating whether the value is the SQL NULL pseudo-value. If the value is NULL, the second parameter is set to a distinctive value (for example, -9999) and the return value from the method is TRUE. If the value is not NULL, it is returned through the variable pointed to by the second parameter, and the method returns FALSE. When you use `getNextColumnNullable()`, the columns are fetched in order. You cannot change the fetch order.

See [Table 3-2](#) for the supported SQL data types and the appropriate method versions to use for each data type. This information can be used for `getNextColumnNullable()`, except there is no `cno` column number parameter for `getNextColumnNullable()`.

getParam()

```
bool getParam(int pno, TYPE* valueP)
bool getParam(int pno, TYPE* valueP, SQLLEN* byteLenP)
```

Each `getParam()` version is used to retrieve the value of an output or input/output parameter, specified by parameter number, after executing a prepared SQL statement. SQL statements are prepared before use with the `Prepare()` method and are executed with the `Execute()` method. The `getParam()` method is used to provide a variable of appropriate data type for the value for each output parameter after executing the statement.

The first argument passed to `getParam()` is the position of the parameter for the output value. The first parameter in a SQL statement is parameter 1. The second argument passed to `getParam()` is a variable for the output value. Overloaded versions of `getParam()` take different data types for the second argument.

The `getParam()` method supports the same data types documented in [Table 3-2](#). Instead of `cno` for column number, however, `getParam()` takes `pno` for parameter number. For NCHAR, NVARCHAR, and binary types, as shown in that table, the method call specifies `byteLenP`, a pointer to an integer value for the number of bytes in the parameter value.

The `getParam()` return is a boolean that is TRUE if the parameter value is NULL, or FALSE otherwise.

The TTCclasses library does not support a large set of data type conversions. The appropriate overloaded version of `getParam()` must be called for each output parameter in the prepared SQL. Calling the wrong version (attempting to use an integer parameter for a `char*` value, for example) may result in program failure.

See [Binding Output or Input/Output Parameters](#) for examples using `getParam()`.

For REF CURSORS, the following signature is supported to use a `TTCmd` object as a statement handle for the REF CURSOR (data type `SQL_REFCURSOR`). See [Working with REF CURSORS](#) for information and an example.

```
bool getParam(int pno, TTCmd** rcCmd)
```

getQueryThreshold()

```
int getQueryThreshold()
```

Returns the threshold value for the `TTCmd` object, as described in [setQueryThreshold\(\)](#).

If no value has been set with `setQueryThreshold()`, this method returns the value of the ODBC connection option `TT_QUERY_THRESHOLD` (if set) or of the TimesTen general connection attribute `QueryThreshold`.

getRowCount()

```
int getRowCount()
```

This method can be called immediately after `Execute()` to return the number of rows that were affected by the executed SQL operation. For example, after execution of a `DELETE` statement that deletes 10 rows, `getRowCount()` returns 10.

isColumnNull()

```
bool isColumnNull(int cno)
```

This method provides another way to determine whether the value in column number `cno` of the current row is `NULL`, returning `TRUE` if so, or `FALSE` otherwise.

Also see information about the [getColumnNullable\(\)](#) method.

Prepare()

```
void Prepare(TTConnection* cP, const char* sqlp)
```

This method associates a SQL statement with the `TTCmd` object. It takes two parameters:

- A pointer to a `TTConnection` object
The connection object should be connected to the database by a call to `TTConnection::Connect()`.
- A `const char*` parameter for the SQL statement being prepared

 **Note:**

- To avoid unwanted round trips between client and server in client/server connections, the `Prepare()` method performs what is referred to as a "deferred prepare", where the request is not sent to the server until required. See TimesTen Deferred Prepare in *Oracle TimesTen In-Memory Database C Developer's Guide*.
- By default (when connection attribute `PrivateCommands=0`), TimesTen shares prepared SQL statements between connections, so subsequent prepares of the same SQL statement on different connections execute very quickly.

printColumn()

```
void printColumn(int cno, STDOSTREAM& os, const char* nullString) const
```

This method prints the value in column number `cno` of the current row to the output stream `os`. Use this for debugging or for demo programs. Use `nullString` to specify what should be printed if the column value is `NULL` (for example, "NULL" or "?").

registerParam()

```
void registerParam(int pno, TTCMD_PARAM_INPUTOUTPUT_TYPE inputOutputType,
                  short sqltype)
void registerParam(int pno, TTCMD_PARAM_INPUTOUTPUT_TYPE inputOutputType,
                  short sqltype, int precision)
void registerParam(int pno, TTCMD_PARAM_INPUTOUTPUT_TYPE inputOutputType,
                  short sqltype, int precision, short scale)
```

Use this method to register a parameter for binding. This is required for output and input/output parameters and can also be used as appropriate to specify SQL type, precision (maximum number of digits that are used by the data type, where applicable), and scale (maximum number of digits to the right of the decimal point, where applicable). See [Registering Parameters](#).

setMaxRows()

```
void setMaxRows(const int nMaxRows)
```

This method sets a limit on the number of rows returned by a `SELECT` statement. If the number of rows in the result set exceeds the set limit, the `TTCmd::FetchNext()` method returns 1 when it has fetched the last row in the requested set size. Also see [getMaxRows\(\)](#).

The default is to return all rows. To reset a limit to again return all rows, call `setMaxRows()` with `nMaxRows` set to 0. The limit is only meaningful for `SELECT` statements.

setParam()

```
void setParam(int pno, TYPE value)
void setParam(int pno, TYPE* valueP)
void setParam(int pno, TYPE* valueP, int byteLen)
```

All overloaded `setParam()` versions are described in this section.

Each `setParam()` version is used to set the value of a parameter, specified by parameter number, before executing a prepared SQL statement. SQL statements are prepared before use with the `Prepare()` method and are executed with the `Execute()` method. If the SQL statement contains any parameter markers (the "?" character used where a literal constant would be valid), values must be assigned to these parameters before the SQL statement can be executed. The `setParam()` method is used to define a value for each parameter before executing the statement. See *Dynamic Parameters in Oracle TimesTen In-Memory Database SQL Reference*.

[Table 3-3](#) below shows the supported SQL data types and the appropriate versions of `setParam()` to use for each type. SQL data types not mentioned are not supported in this version of TTCmd. For `NCHAR`, `NVARCHAR`, and binary types, as shown in the table, the method call specifies `byteLen`, an integer value for the number of bytes in the parameter value.

The first argument passed to `setParam()` is the position of the parameter to be set. The first parameter in a SQL statement is parameter 1. The second argument passed to `setParam()` is the value of the parameter. Overloaded versions of `setParam()` take different data types for the second argument.

The TTCmd library does not support a large set of data type conversions. The appropriate overloaded version of `setParam()` must be called for each parameter in the prepared SQL. Calling the wrong version (attempting to set an integer parameter to a `char*` value, for example) may result in program failure.

Values passed to `setParam()` are copied into internal buffers maintained by the TTCmd object. These buffers are statically allocated and bound by the `Prepare()` method. The parameter value is the value passed into `setParam()` at the time of the `setParam()` call, not the value at the time of a subsequent `Execute()` method call.

See [Binding Input Parameters](#) and [Binding Output or Input/Output Parameters](#) for examples using `setParam()`. See [Binding Duplicate Parameters](#) regarding duplicate parameters.

Note:

- To set the length of the value for a bound parameter, see [setParamLength\(\)](#).
- To set a value of `NULL` for a bound parameter, see [setParamNull\(\)](#).

Table 3-3 setParam() Signatures for Supported TimesTen Table Column Types

Data Type	setParam() Variants Supported
TT_TINYINT	setParam(int <i>pno</i> , SQLTINYINT <i>value</i>)
TT_SMALLINT	setParam(int <i>pno</i> , SQLSMALLINT <i>value</i>)
TT_INTEGER	setParam(int <i>pno</i> , SQLINTEGER <i>value</i>)
TT_BIGINT	setParam(int <i>pno</i> , SQLBIGINT <i>value</i>)
BINARY_FLOAT REAL	setParam(int <i>pno</i> , float <i>value</i>)
BINARY_DOUBLE DOUBLE	setParam(int <i>pno</i> , double <i>value</i>)
NUMBER	setParam(int <i>pno</i> , char* <i>valueP</i>) setParam(int <i>pno</i> , const char* <i>valueP</i>) setParam(int <i>pno</i> , SQLCHAR* <i>valueP</i>) setParam(int <i>pno</i> , SQLTINYINT <i>value</i>) setParam(int <i>pno</i> , SQLSMALLINT <i>value</i>) setParam(int <i>pno</i> , SQLINTEGER <i>value</i>) setParam(int <i>pno</i> , SQLBIGINT <i>value</i>)
<p>Note: The integer versions are appropriate only for columns declared with the scale parameter set to zero, such as NUMBER(8) or NUMBER(8,0).</p>	
TT_CHAR CHAR TT_VARCHAR VARCHAR2	setParam(int <i>pno</i> , char* <i>valueP</i>) setParam(int <i>pno</i> , const char* <i>valueP</i>) setParam(int <i>pno</i> , SQLCHAR* <i>valueP</i>)
TT_NCHAR NCHAR TT_NVARCHAR NVARCHAR2	setParam(int <i>pno</i> , SQLWCHAR* <i>valueP</i> , int <i>byteLen</i>)
BINARY VARBINARY	setParam(int <i>pno</i> , const void* <i>valueP</i> , int <i>byteLen</i>)

Table 3-3 (Cont.) setParam() Signatures for Supported TimesTen Table Column Types

Data Type	setParam() Variants Supported
DATE	setParam(int <i>pno</i> , TIMESTAMP_STRUCT& <i>valueP</i>)
TT_TIMESTAMP	
TIMESTAMP	
TT_DATE	setParam(int <i>pno</i> , DATE_STRUCT& <i>valueP</i>)
TT_TIME	setParam(int <i>pno</i> , TIME_STRUCT& <i>valueP</i>)

setParamLength()

(Version for non-batch operations)

```
void setParamLength(int pno, int byteLen)
```

Sets the length, in bytes, of the bound value for an input parameter specified by parameter number, before execution of the prepared statement.

**Note:**

There is also a batch version of this method. See [setParamLength\(\)](#).

setParamNull()

(Version for non-batch operations)

```
void setParamNull(int pno)
```

Sets a value of SQL NULL for a bound input parameter specified by parameter number.

**Note:**

There is also a batch version of this method. See [setParamNull\(\)](#).

setQueryThreshold()

```
void setQueryThreshold(const int nSecs)
```

Use this method to specify a threshold time limit, in seconds, for the TTCmd object. (This applies to any SQL statement, not just queries.) If the execution time of a

statement exceeds the threshold, a warning is written to the support log. Execution continues and is not affected by the threshold. Also see [getQueryThreshold\(\)](#).

The `setQueryThreshold()` method has the same effect as using `SQLSetStmtOption` to set `TT_QUERY_THRESHOLD` or setting the TimesTen general connection attribute `QueryThreshold`.

See [Setting a Timeout or Threshold for Executing SQL Statements](#).

setQueryTimeout()

```
void setQueryTimeout(const int nSecs)
```

Use this method to specify how long, in seconds, any SQL statement (not just a query) executes before timing out. By default there is no timeout.

This has the same effect as using `SQLSetStmtOption` to set `SQL_QUERY_TIMEOUT` or setting the TimesTen general connection attribute `SQLQueryTimeout` (or `SQLQueryTimeoutMsec`, to use milliseconds).

See [Setting a Timeout or Threshold for Executing SQL Statements](#).

Public Methods for Obtaining TTCmd Object Properties

There are several useful methods for asking questions about properties of the bound input parameters and output columns of a prepared `TTCmd` object. These methods generally provide meaningful results only when a statement has previously been prepared.

This section summarizes then describes the methods for obtaining `TTCmd` object properties.

Public Methods Summary for Obtaining TTCmd Object Properties

Method	Description
getColumnName()	Returns the name of the specified column.
getColumnNullability()	Indicates whether data in the specified column can have the value <code>NULL</code> .
getColumnPrecision()	Returns the precision of the specified column.
getColumnScale()	Returns the scale of the specified column.
getColumnType()	Returns the ODBC data type of the specified column.
getNColumns()	Returns the number of output columns.
getNParameters()	Returns the number of input parameters.
getParamNullability()	Indicates whether the value of the specified parameter can be <code>NULL</code> .
getParamPrecision()	Returns the precision of the specified parameter in a prepared statement.
getParamScale()	Returns the scale of the specified parameter in a prepared statement.
getParamType()	Returns the ODBC data type of the specified parameter.
isBeingExecuted()	Indicates whether the statement represented by the <code>TTCmd</code> object is being executed.

getColumnName()

```
const char* getColumnName(int cno)
```

Returns the name of column number *cno*.

getColumnNullability()

```
int getColumnNullability(int cno)
```

Indicates whether column number *cno* can NULL data. It returns `SQL_NO_NULLS`, `SQL_NULLABLE`, or `SQLNULLABLE_UNKNOWN`.

getColumnPrecision()

```
SQLULEN getColumnPrecision(int cno)
```

Returns the precision of data in column number *cno*, referring to the size of the column in the database. For example, for a `VARCHAR2(25)` column, the precision returned would be 25.

This value is generally interesting only when generating output from table columns of type `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `BINARY`, and `VARBINARY`.

getColumnScale()

```
int getColumnScale(int cno)
```

Returns the scale of data in column number *cno*, referring to the maximum number of digits to the right of the decimal point.

getColumnType()

```
int getColumnType(int cno)
```

Returns the data type of column number *cno*. The value returned is the ODBC type of the parameter (for example, `SQL_INTEGER`, `SQL_REAL`, `SQL_BINARY`, `SQL_CHAR`) as found in `sql.h`. Additional TimesTen ODBC types (`SQL_WCHAR`, `SQL_WVARCHAR`) can be found in the TimesTen header file `timesten.h`.

getNColumns()

```
int getNColumns()
```

Returns the number of output columns.

getNParameters()

```
int getNParameters()
```

Returns the number of input parameters for the SQL statement.

getParamNullability()

```
int getParamNullability(int pno)
```

Indicates whether parameter number *pno* can have the value `NULL`. It returns `SQL_NO_NULLS`, `SQL_NULLABLE`, or `SQLNULLABLE_UNKNOWN`.



Note:

In earlier releases this method returned `bool` instead of `int`.

getParamPrecision()

```
SQLULEN getParamPrecision(int pno)
```

Returns the precision of parameter number *pno*, referring to the maximum number of digits that are used by the data type. Also see information for [getColumnPrecision\(\)](#), above.

getParamScale()

```
int getParamScale(int pno)
```

Returns the scale of parameter number *pno*, referring to the maximum number of digits to the right of the decimal point.

getParamType()

```
int getParamType(int pno)
```

Returns the data type of parameter number *pno*. The value returned is the ODBC type (for example, `SQL_INTEGER`, `SQL_REAL`, `SQL_BINARY`, `SQL_CHAR`) as found in `sql.h`. Additional TimesTen types (`SQL_WCHAR`, `SQL_WVARCHAR`) can be found in the TimesTen header file `timesten.h`.

isBeingExecuted()

```
bool isBeingExecuted()
```

Indicates whether the statement represented by the `TTCmd` object is being executed.

Public Methods for Batch Operations

TimesTen supports the ODBC function `SQLBindParams` for batch insert, update and delete operations. `TTClasses` provides an interface to the ODBC function `SQLBindParams`.

Performing batch operations with TTClasses is similar to performing non-batch operations. SQL statements are first compiled using `PrepareBatch()`. Then each parameter in that statement is bound to an array of values using `BindParameter()`. Finally, the statement is executed using `ExecuteBatch()`.

See the TTClasses `bulktest` sample program in the TimesTen Classic Quick Start for an example of using a batch operation. Refer to [About TimesTen Quick Start and Sample Applications](#).

This section summarizes then describes the TTCmd methods that expose the batch INSERT, UPDATE, and DELETE functionality to TTClasses users.

Public Methods Summary for Batch Operations

Method	Description
batchSize()	Returns the number of statements in the batch.
BindParameter()	Binds an array of values for one parameter of a statement prepared using <code>PrepareBatch()</code> .
ExecuteBatch()	Invokes a SQL statement that has been prepared for execution by <code>PrepareBatch()</code> . It returns the number of rows in the batch that were updated.
PrepareBatch()	Prepares batch INSERT, UPDATE, and DELETE statements.
setParamLength()	Sets the length, in bytes, of the value of the specified bound parameter before execution of the prepared statement.
setParamNull()	Sets the specified bound parameter to NULL before execution of the prepared statement.

batchSize()

```
u_short batchSize()
```

Returns the number of statements in the batch.

BindParameter()

```
void BindParameter(int pno, unsigned short batSz, TYPE* valueP)
void BindParameter(int pno, unsigned short batSz, TYPE* valueP, size_t
maxByteLen)
void BindParameter(int pno, unsigned short batSz, TYPE* valueP,
SQLLEN* userByteLenP, size_t maxByteLen)
```

The overloaded `BindParameter()` method is used to bind an array of values for a specified parameter in a SQL statement compiled using `PrepareBatch()`. This allows iterating through a batch of repeated executions of the statement with different values. The `pno` parameter indicates the position in the statement of the parameter to be bound, starting from the left, where the first parameter is 1, the next is 2, and so on.

[Table 3-4](#) below shows the supported SQL data types and the appropriate versions of `BindParameter()` to use for each parameter type.

Also see [Binding Duplicate Parameters](#).

The *batSz* (batch size) value of this call must match the *batSz* value specified in `PrepareBatch()`, and the bound arrays should contain at least the *batSz* number of values. You must determine the correct data type for each parameter. If an invalid parameter number is specified, the specified batch size is a mismatch, or the data buffer is null, then a `TTStatus` object is thrown as an exception and a runtime error is written to the `TTClasses` global logging facility at the `TTLog::TTLOG_ERR` logging level.

Before each invocation of `ExecuteBatch()`, the application should fill the bound arrays with valid parameter values. Note that you can use the `setParamNull()` method to set null values, as described in [setParamNull\(\)](#). (Be aware that for batch mode, you must use the two-parameter version of `setParamNull()` that specifies *rowno*. The one-parameter version is for non-batch use only.)

For the SQL types `TT_CHAR`, `CHAR`, `TT_VARCHAR`, and `VARCHAR2`, an additional maximum length parameter is required in the `BindParameter()` call:

- *maxByteLen* of type `size_t` is for the maximum length, in bytes, of any value for this parameter position.

For the SQL types `TT_NCHAR`, `NCHAR`, `TT_NVARCHAR`, `NVARCHAR2`, `BINARY`, and `VARBINARY`, two additional parameters are required in the `BindParameter()` call, an array of parameter lengths and a maximum length:

- *userByteLenP* is an array of `SQLLEN` parameter lengths, in bytes, to specify the length of each value in the batch for this parameter position in the SQL statement. This array must be at least *batSz* in length and filled with valid length values before `ExecuteBatch()` is called. (You can store `SQL_NULL_DATA` in the array of parameter lengths for a null value, which is equivalent to using the `setParamNull()` batch method.)
- *maxByteLen* is as described above. This indicates the maximum length value that can be specified in any element of the *userByteLenP* array.

For data types where *userByteLenP* is not available (or as an alternative where it is available), you can optionally use the `setParamLength()` batch method to set data lengths, as described in [setParamLength\(\)](#), and use the `setParamNull()` batch method to set null values, as described in [setParamNull\(\)](#).

See the example in the next section, [ExecuteBatch\(\)](#), for examples of `BindParameter()` usage.

Table 3-4 BindParameter() Signatures for Supported TimesTen Table Column Types

SQL Data Type	BindParameter() Variants Supported
<code>TT_TINYINT</code>	<code>BindParameter(int pno, short batSz, SQLTINYINT* user_tiP)</code>
<code>TT_SMALLINT</code>	<code>BindParameter(int pno, short batSz, SQLSMALLINT* user_siP)</code>
<code>TT_INTEGER</code>	<code>BindParameter(int pno, short batSz, SQLINTEGER* user_iP)</code>

Table 3-4 (Cont.) BindParameter() Signatures for Supported TimesTen Table Column Types

SQL Data Type	BindParameter() Variants Supported
TT_BIGINT	BindParameter(int <i>pno</i> , short <i>batSz</i> , SQLBIGINT* <i>user_biP</i>)
BINARY_FLOAT	BindParameter(int <i>pno</i> , short <i>batSz</i> , float* <i>user_fP</i>)
BINARY_DOUBLE	BindParameter(int <i>pno</i> , short <i>batSz</i> , double* <i>user_dP</i>)
NUMBER	BindParameter(int <i>pno</i> , short <i>batSz</i> , char** <i>user_cPP</i> , size_t <i>maxByteLen</i>)
TT_CHAR CHAR TT_VARCHAR VARCHAR2	BindParameter(int <i>pno</i> , short <i>batSz</i> , char** <i>user_cPP</i> , size_t <i>maxByteLen</i>)
TT_NCHAR NCHAR TT_NVARCHAR NVARCHAR2	BindParameter(int <i>pno</i> , short <i>batSz</i> , SQLWCHAR** <i>user_wcPP</i> , SQLLEN* <i>userByteLenP</i> , size_t <i>maxByteLen</i>)
BINARY VARBINARY	BindParameter(int <i>pno</i> , short <i>batSz</i> , const void** <i>user_binPP</i> , SQLLEN* <i>userByteLenP</i> , size_t <i>maxByteLen</i>)
DATE TT_TIMESTAMP TIMESTAMP	BindParameter(int <i>pno</i> , short <i>batSz</i> , TIMESTAMP_STRUCT* <i>user_tssP</i>)
TT_DATE	BindParameter(int <i>pno</i> , int <i>batSz</i> , DATE_STRUCT* <i>user_dsP</i>)
TT_TIME	BindParameter(int <i>pno</i> , int <i>batSz</i> , TIME_STRUCT* <i>user_tsP</i>)

ExecuteBatch()SQLULEN ExecuteBatch(unsigned short *numRows*)

After preparing a SQL statement with `PrepareBatch()` and calling `BindParameter()` for each parameter in the SQL statement, use `ExecuteBatch()` to execute the statement `numRows` times. The value of `numRows` must be no more than the `batSz` (batch size) value specified in the `PrepareBatch()` and `BindParameter()` calls, but can be less than `batSz` as required by application logic.

This method returns the number of rows that were updated, with possible values in the range 0 to `batSz`, inclusive. (For those familiar with ODBC, this is the third parameter, `*pirow`, of an ODBC `SQLParamOptions` call. Refer to ODBC API reference documentation for information about `SQLParamOptions`.)

Before calling `ExecuteBatch()`, the application should fill the arrays of parameters previously bound by `BindParameter()` with valid values.

A `TTStatus` object is thrown as an exception if an error occurs (often due to violation of a uniqueness constraint). In this event, the return value is not valid and the batch is incomplete and should generally be rolled back.

The following example shows how to use the `ExecuteBatch()` method. The `bulktest` TimesTen Classic Quick Start demo also shows usage of this method. (See [About TimesTen Quick Start and Sample Applications](#).)

First, create a table with two columns:

```
CREATE TABLE batch_table (a TT_INTEGER, b VARCHAR2(100));
```

Following is the sample code. Populate the rows of the table in batches of 50.

```
#define BATCH_SIZE 50
#define VARCHAR_SIZE 100

int int_array[BATCH_SIZE];
char char_array[BATCH_SIZE][VARCHAR_SIZE];

// Prepare the statement

TTCmd insert;
TTConnection connection;

// (assume a connection has been established)

try {

    insert.PrepareBatch (&connection,
                        (const char*)"insert into batch_table values (?,?)",
                        BATCH_SIZE);

    // Commit the prepared statement
    connection.Commit();

    // Bind the arrays of parameters
    insert.BindParameter(1, BATCH_SIZE, int_array);
    insert.BindParameter(2, BATCH_SIZE, (char **)char_array, VARCHAR_SIZE);

    // Execute 5 batches, inserting a total of 5 * BATCH_SIZE rows into
```



```
// the database
for (int iter = 0; iter < 5; iter++)
{
    // Populate the value arrays with values.
    // (A more meaningful way of putting data into
    // the database is to read values from a file, for example,
    // rather than generating them arbitrarily.)

    for (int i = 0; i < BATCH_SIZE; i++)
    {
        int_array[i] = i * iter + i;
        sprintf(char_array[i], "varchar value # %d", i*iter+ i);
    }

    // Execute the batch insert statement,
    // which inserts the entire contents of the
    // integer and char arrays in one operation.
    SQLLEN num_ins = insert.ExecuteBatch(BATCH_SIZE);

    cerr << "Inserted " << num_ins << " rows." << endl;

    connection.Commit();

} // for iter

} catch (TTErrror er1) {
    cerr << er1 << endl;
}
```

The number of rows updated (*num_ins* in the example) can be less than `BATCH_SIZE` if, for example, there is a violation of a uniqueness constraint on a column. You can use code similar to that in the next example check for this situation and roll back the transaction as necessary.

```
for (int iter = 0; iter < 5; iter++)
{
    // Populate the value arrays with values.
    // (A better way of putting meaningful data into
    // the database is to read values from a file,
    // rather than generating them arbitrarily.)

    for (int i = 0; i < BATCH_SIZE; i++)
    {
        int_array[i] = i * iter + i;
        sprintf(char_array[i], "varchar value # %d", i*iter+i);
    }

    // now we execute the batch insert statement,
    // which does the work of inserting the entire
    // contents of the integer and char arrays in
    // one operation

    SQLLEN num_ins = insert.ExecuteBatch(BATCH_SIZE);
```

```

cerr << "Inserted " << num_ins << " rows (expected "
      << BATCH_SIZE << " rows)." << endl;

if (num_ins == BATCH_SIZE) {
    cerr << "Committing batch" << endl;
    connection.Commit();
}
else {
    cerr << "Some rows were not inserted as expected, rolling back "
          << "transaction." << endl;
    connection.Rollback();
    break; // jump out of batch insert loop
}

} // for loop

```

PrepareBatch()

```
void PrepareBatch(TTConnection* cP, const char* sqlp, unsigned short batSz)
```

`PrepareBatch()` is comparable to the `Prepare()` method but for batch INSERT, UPDATE, or DELETE statements. The `cP` and `sqlp` parameters are used as with `Prepare()`. See [Prepare\(\)](#).

The `batSz` (batch size) parameter specifies the maximum number of insert, update, or delete operations that are performed using subsequent calls to `ExecuteBatch()`.

A `TTStatus` object is thrown as an exception if an error occurs.

Note:

To avoid unwanted round trips between client and server in client/server connections, the `PrepareBatch()` method performs what is referred to as a "deferred prepare", where the request is not sent to the server until required. See TimesTen Deferred Prepare in *Oracle TimesTen In-Memory Database C Developer's Guide*.

setParamLength()

(Version for batch operations)

```
void setParamLength(int pno, unsigned short rowno, int byteLen)
```

This method sets the length of a bound parameter value before a call to `ExecuteBatch()`. The `pno` argument specifies the parameter number in the SQL statement (where the first parameter is number 1). The `rowno` argument specifies the row number in the array of parameters being bound (where the first row is row number 1). The `byteLen` parameter specifies the desired length, in bytes, not counting the NULL terminator. Alternatively, `byteLen` can be set to `SQL_NTS` for a null-terminated string. (It can also be set to `SQL_NULL_DATA`, which is equivalent to using the `setParamNull()` batch method, described next.)

 **Note:**

- For binary and NCHAR types, as shown in [Table 3-4](#), it may be easier to use the `BindParameter() userByteLenP` array to set parameter lengths. Be aware that row numbering in the array of parameters being bound starts with 0 in the `userByteLenP` array but with 1 when you use `setParamLength()`.
- There is also a non-batch version of this method. See [setParamLength\(\)](#). (It is important to use only the two-parameter version for non-batch operations, and only the three-parameter version that specifies `rowno` for batch operations.)

setParamNull()

(Version for batch operations)

```
void setParamNull(int pno, unsigned short rowno)
```

This method sets a bound parameter value to NULL before a call to `ExecuteBatch()`. The `pno` argument specifies the parameter number in the SQL statement (where the first parameter is number 1). The `rowno` argument specifies the row number in the array of parameters being bound (where the first row is row number 1).

 **Note:**

- For binary and NCHAR types, as shown in [Table 3-4](#), there is a `BindParameter() userByteLenP` array. For these types, you can have a null value by specifying `SQL_NULL_DATA` in this array, which is equivalent to using `setParamNull()`. Be aware that row numbering in the bound array of parameters `userByteLenP` starts with 0, but numbering starts with 1 when you use `setParamNull()`.
- There is also a non-batch version of this method. See [setParamNull\(\)](#). (It is important to use only the one-parameter version for non-batch operations, and only the two-parameter version that specifies `rowno` for batch operations.)

4

System Catalog Classes Reference

This reference chapter contains descriptions of TTClasses system catalog classes and their methods. These classes enable you to examine the TimesTen system catalog.

- [TTCatalog Reference](#)
- [TTCatalogTable Reference](#)
- [TTCatalogColumn Reference](#)
- [TTCatalogIndex Reference](#)
- [TTCatalogSpecialColumn Reference](#)

TTCatalog Reference

The `TTCatalog` class is the top-level class used for programmatically accessing metadata information about tables in a database.

- [Public Members](#)
- [Public Methods](#)

Also see [TTCatalog Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTCatalog` public methods.

Public Methods Summary

Method	Description
fetchCatalogData()	Reads the catalogs in the database for information about tables and indexes and stores this information into <code>TTCatalog</code> internal data structures.
getNumSysTables()	Returns the number of system tables in the database.
getNumTables()	Returns the total number of tables (user tables plus system tables) in the database.
getNumUserTables()	Returns the number of user tables in the database.
getTable()	Returns a constant reference to the <code>TTCatalogTable</code> object for the specified table.
getTableIndex()	Returns the index in the <code>TTCatalog</code> object for the specified table.

Method	Description
getUserTable()	Returns a constant reference to the <code>TTCatalogTable</code> object corresponding to the n th user table in the system (where n is specified).

fetchCatalogData()

```
void fetchCatalogData()
```

This is the only `TTCatalog` method that interacts with the database. It reads the catalogs in the database for information about tables and indexes, storing the information into `TTCatalog` internal data structures.

Subsequent use of the constructed `TTCatalog` object is completely offline after it is constructed. It is no longer connected to the database.

You must call this method before you use any of the `TTCatalog` accessor methods.

The following example demonstrates the use of `TTCatalog`.

```
TTCConnection conn;
conn.Connect(DSN=TptbmData37);
TTCatalog cat (&conn);
cat.fetchCatalogData();
// TTCatalog cat is no longer connected to the database;
// you can now query it through its read-only methods.
cerr << "There are " << cat.getNumTables() << " tables in this
database:" << endl;
for (int i=0; i < cat.getNumTables(); i++)
cerr << cat.getTable(i).getTableOwner() << "."
    << cat.getTable(i).getTableName() << endl;
```

getNumSysTables()

```
int getNumSysTables()
```

Returns the number of system tables in the database. Also see the following methods, `getNumTables()` and `getNumUserTables()`.

getNumTables()

```
int getNumTables()
```

Returns the total number of tables in the database (user plus system tables). Also see the preceding method, `getNumSysTables()`, and the following method, `getNumUserTables()`.

getNumUserTables()

```
int getNumUserTables()
```

Returns the number of user tables in the database. Also see the preceding methods, `getNumSysTables()` and `getNumTables()`.

getTable()

```
const TTCatalogTable& getTable(const char* owner, const char* tblname)
const TTCatalogTable& getTable(int tno)
```

Returns a constant reference to the `TTCatalogTable` object for the specified table. Also see [getUserTable\(\)](#).

For the first signature, this is for the table named `tblname` and owned by `owner`.

For the second signature, this is for the table corresponding to table number `tno` in the system. This is intended to facilitate iteration through all the tables in the system. The order of the tables in this array is arbitrary.

The following relationship is true:

```
0 <= tno < getNumTables()
```

Also see [TTCatalogTable Usage](#).

getTableIndex()

```
int getTableIndex(const char* owner, const char* tblname) const
```

This method fetches the index in the `TTCatalog` object for the specified `owner.tblname` object. It returns -2 if `owner.tblname` does not exist. It returns -1 if `fetchCatalogData()` was not called first.

The example retrieves information about the `TTUSER.MYDATA` table from a `TTCatalog` object. You can then call methods of `TTCatalogTable`, described next, to get information about this table.

```
TTCatalogTable cat;
conn.Connect(...);
TTCatalog cat (&conn);
cat.fetchCatalogData();

int idx = cat.getTableIndex("TTUSER", "MYDATA");
if (idx < 0) {
    cerr << "Table TTUSER.MYDATA does not exist." << endl;
    return;
}

TTCatalogTable &table = cat.getTable(idx);
```

getUserTable()

```
const TTCatalogTable& getUserTable(int tno)
```

Returns a constant reference to the `TTCatalogTable` object corresponding to user table number `tno` in the system. This method is intended to facilitate iteration through all of the

user tables in the system. The order of the user tables in this array is arbitrary. Also see [getTable\(\)](#).

The following relationship is true:

```
0 <= tno < getNumUserTables()
```



Note:

There is no equivalent method for system tables.

TTCatalogTable Reference

Each object of the top-level `TTCatalog` class internally contains an array of `TTCatalogTable` objects.

- [Public Members](#)
- [Public Methods](#)

Also see [TTCatalogTable Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTCatalogTable` public methods.

Public Methods Summary

Method	Description
getColumn()	Returns a constant reference to the <code>TTCatalogColumn</code> corresponding to the <i>i</i> th column in the table.
getIndex()	Returns a constant reference to the <code>TTCatalogIndex</code> object corresponding to the <i>n</i> th index in the table, where <i>n</i> is specified.
getNumColumns()	Returns the number of columns in the table.
getNumIndexes()	Returns the number of indexes on the table.
getNumSpecialColumns()	Returns the number of <i>special columns</i> in this table. See TTCatalogSpecialColumn Usage .
getSpecialColumn()	Returns a special column (<code>TTCatalogSpecialColumn</code> object) from this table, according to the specified column number.
getTableName()	Returns the name of the table.
getTableOwner()	Returns the owner of the table.
getTableType()	Returns the table type as returned by the ODBC <code>SQLTables</code> function.

Method	Description
isSystemTable()	Returns <code>TRUE</code> if the table is a system table.
isUserTable()	Returns <code>TRUE</code> if the table is a user table.

getColumn()

```
const TTCatalogColumn& getColumn(int cno)
```

Returns a constant reference to the `TTCatalogColumn` object corresponding to column number `cno` in the table. This method is intended to facilitate iteration through all the columns in the table.

The following relationship is true:

```
0 <= cno < getNumColumns()
```

getIndex()

```
const TTCatalogIndex& getIndex(int num)
```

Returns a constant reference to the `TTCatalogIndex` object corresponding to index number `num` in the table. This method is intended to facilitate iteration through all the indexes of the table. The order of the indexes of a table in this array is arbitrary.

The following relationship is true:

```
0 <= num < getNumIndexes()
```

getNumColumns()

```
int getNumColumns()
```

Returns the number of columns in the table.

getNumIndexes()

```
int getNumIndexes()
```

Returns the number of indexes on the table.

getNumSpecialColumns()

```
int getNumSpecialColumns()
```

Returns the number of *special columns* in this `TTCatalogTable` object. Because `TimesTen` supports only rowid special columns, this always returns 1.

Also see [TTCatalogSpecialColumn Usage](#).

getSpecialColumn()

```
const TTCatalogSpecialColumn& getSpecialColumn(int num) const
```

Returns a *special column* (TTCatalogSpecialColumn object) from this TTCatalogTable object, according to the specified column number. In TimesTen this can be only a rowid pseudocolumn.

Also see [TTCatalogSpecialColumn Usage](#).

getTableName()

```
const char* getTableName()
```

Returns the name of the table.

getTableOwner()

```
const char* getTableOwner()
```

Returns the owner of the table.

getTableType()

```
const char* getTableType() const
```

Returns the table type of this TTCatalogTable object, as from an ODBC SQLTables call. In TimesTen this may be TABLE, SYSTEM TABLE, VIEW, or SYNONYM.

isSystemTable()

```
bool isSystemTable()
```

Returns TRUE if the table is a system table (owned by SYS, TTREP, or GRID), or FALSE otherwise.

The isSystemTable() method and isUserTable() method (described next) are useful for applications that iterate over all tables in a database after a call to TTCatalog::fetchCatalogData(), so that you can filter or annotate tables to differentiate the system and user tables. The TTCatalog demo program catalog provides an example of how this can be done. (See [About TimesTen Quick Start and Sample Applications](#).)

isUserTable()

```
bool isUserTable()
```

Returns TRUE if this is a user table, which is to say it is not a system table, or FALSE otherwise. Note that isUserTable() returns the opposite of isSystemTable() for any table. The description of isSystemTable() discusses the usage and usefulness of these methods.

TTCatalogColumn Reference

Each `TTCatalogTable` object contains an array of `TTCatalogColumn` objects and an array of `TTCatalogIndex` objects.

- [Public Members](#)
- [Public Methods](#)

Also see [TTCatalogColumn Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTCatalogColumn` public methods.

Public Methods Summary

Method	Description
getColumnName()	Return the name of the column.
getDataType()	Returns an integer representing the ODBC SQL data type of the column.
getLength()	Returns the length of the column, in bytes.
getNullable()	Indicates whether the column can contain <code>NULL</code> values. (This is not a boolean value, as noted in the description below.)
getPrecision()	Returns the precision of the column.
getRadix()	Returns the radix of the column.
getScale()	Returns the scale of the column.
getTypeName()	Returns the TimesTen name for the type returned by <code>getDataType()</code> .

getColumnName()

```
const char* getColumnName()
```

Returns the name of the column.

getDataType()

```
int getDataType()
```

Returns an integer representing the data type of the column. This is the standard ODBC SQL type code or a TimesTen extension type code.

getLength()

```
int getLength()
```

Returns the length of data in the column, in bytes.

getNullable()

```
int getNullable()
```

Indicates whether the column can contain `NULL` values. It returns `SQL_NO_NULLS`, `SQL_NULLABLE`, or `SQL_NULLABLE_UNKNOWN`.

getPrecision()

```
int getPrecision()
```

Returns the precision of data in the column, referring to the maximum number of digits that are used by the data type.

getRadix()

```
int getRadix()
```

Returns the radix of the column, according to ODBC `SQLColumns` function output.

getScale()

```
int getScale()
```

Returns the scale of data in the column, which is the maximum number of digits to the right of the decimal point.

getTypeName()

```
const char* getTypeName()
```

Returns the TimesTen name of the type returned by `getDataType()`.

TTCatalogIndex Reference

Each `TTCatalogTable` object contains an array of `TTCatalogColumn` objects and an array of `TTCatalogIndex` objects.

- [Public Members](#)
- [Public Methods](#)

Also see [TTCatalogIndex Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTCatalogIndex` public methods.

Public Methods Summary

Method	Description
getCollation()	Returns the collation of the specified column in the index.
getColumnName()	Returns the name of the specified column in the index.
getIndexName()	Returns the name of the index.
getIndexOwner()	Returns the owner of the index.
getNumColumns()	Returns the number of columns in the index.
getTableName()	Returns the name of the table for which the index was created.
getType()	Returns the type of the index.
isUnique()	Indicates whether the index is a unique index.

getCollation()

```
char getCollation (int num)
```

Returns the collation of column number *num* in the index. Values returned are "A" for ascending order or "D" for descending order.

getColumnName()

```
const char* getColumnName(int num)
```

Returns the name of column number *num* in the index.

getIndexName()

```
const char* getIndexName()
```

Returns the name of the index.

getIndexOwner()

```
const char* getIndexOwner()
```

Returns the owner of the index.

getNumColumns()

```
int getNumColumns()
```

Returns the number of columns in the index.

getTableName()

```
const char* getTableName()
```

Returns the name of the table for which the index was created. This is the table represented by the `TTCatalogTable` object from which the index was retrieved through a `TTCatalogTable::getIndex()` call.

getType()

```
int getType()
```

Returns the type of the index. For TimesTen, the allowable values are `PRIMARY_KEY`, `HASH_INDEX` (the same as `PRIMARY_KEY`), and `RANGE_INDEX`.

isUnique()

```
bool isUnique()
```

Returns `TRUE` if the index is a unique index, or `FALSE` otherwise.

TTCatalogSpecialColumn Reference

This class is a wrapper for results from an ODBC `SQLSpecialColumns` function call on a table represented by a `TTCatalogTable` object.

- [Public Members](#)
- [Public Methods](#)

Also see [TTCatalogSpecialColumn Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTCatalogSpecialColumn` public methods.

Public Methods Summary

Method	Description
getColumnName()	Returns the name of the special column.
getDataType()	Returns the data type of the special column, as an integer.
getLength()	Returns the length of data in the special column, in bytes.
getPrecision()	Returns the precision of the special column.
getScale()	Returns the scale of the special column.
getTypeName()	Returns the data type of the special column, as a character string.

getColumnName()

```
const char* getColumnName()
```

Returns the name of the special column.

getDataType()

```
int getDataType()
```

Returns an integer representing the ODBC SQL data type of the special column. In TimesTen this can be only `SQL_ROWID`.

getLength()

```
int getLength()
```

Returns the length of data in the special column, in bytes.

getPrecision()

```
int getPrecision()
```

Returns the precision for data in the special column, referring to the maximum number of digits used by the data type.

getScale()

```
int getScale()
```

Returns the scale for data in the special column, referring to the maximum number of digits to the right of the decimal point.

getTypeName()

```
const char* getTypeName()
```

Returns the data type name that corresponds to the ODBC SQL data type value returned by `getDataType()`. In TimesTen this can be only `ROWID`.

5

XLA Classes Reference

After a brief introduction, this chapter contains descriptions of TTXClasses XLA classes and their methods.

- [TTXlaPersistConnection Reference](#)
- [TTXlaRowViewer Reference](#)
- [TTXlaTableHandler Reference](#)
- [TTXlaTableList Reference](#)
- [TTXlaTable Reference](#)
- [TTXlaColumn Reference](#)

TTXlaPersistConnection Reference

Use `TTXlaPersistConnection` to create an XLA connection to a database.

- [Public Members](#)
- [Public Methods](#)

Also see [TTXlaPersistConnection Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTXlaPersistConnection` public methods.

Public Methods Summary

Method	Description
ackUpdates()	Advances the bookmark to the next set of updates.
Connect()	Connects with the specified bookmark, or creates one if it does not exist (depending on the method signature).
deleteBookmarkAndDisconnect()	Deletes the bookmark and disconnects from the database.
Disconnect()	Closes an XLA connection to a database, leaving the bookmark in place.
fetchUpdatesWait()	Fetches updates to the transaction log within the specified wait period.
getBookmarkIndex()	Gets the current transaction log position.
setBookmarkIndex()	Returns to the transaction log position that was acquired by a <code>getBookmarkIndex()</code> call.

ackUpdates()

```
void ackUpdates()
```

Use this method to advance the bookmark to the next set of updates. After you have acknowledged a set of updates, the updates cannot be viewed again by this bookmark. Therefore, a `setBookmarkIndex()` call does not allow you to replay XLA records that have been acknowledged by a call to `ackUpdates()`. (See the descriptions of `getBookmarkIndex()` and `setBookmarkIndex()` for information about replaying a set of updates.)

Applications should acknowledge updates when a batch of XLA records have been read and processed, so that the transaction log does not fill up available file system space; however, do not call `ackUpdates()` too frequently, because it is a relatively expensive operation.

If an application uses XLA to read a batch of records and then a failure occurs before `ackUpdates()` is called, the records are retrieved when the application reestablishes its XLA connection.



Note:

The transaction log is in a file system location according to the `TimesTen LogDir` attribute setting, if specified, or the `DataStore` attribute setting if `LogDir` is not specified. Refer to *Data Store Attributes in Oracle TimesTen In-Memory Database Reference*.

Connect()

```
virtual void Connect(const char* connStr, const char* bookmarkStr,
                    bool createBookmarkFlag)
virtual void Connect(const char* connStr, const char* username,
                    const char* password, const char* bookmarkStr,
                    bool createBookmarkFlag)
virtual void Connect(const char* connStr,
                    TTConnection::DRIVER_COMPLETION_ENUM
                    driverCompletion,
                    const char* bookmarkStr, bool createBookmarkFlag)

virtual void Connect(const char* connStr, const char* bookmarkStr)
virtual void Connect(const char* connStr, const char* username,
                    const char* password, const char* bookmarkStr)
virtual void Connect(const char* connStr,
                    TTConnection::DRIVER_COMPLETION_ENUM
                    driverCompletion,
                    const char* bookmarkStr)
```

Each XLA connection has a bookmark name associated with it, so that after disconnecting and reconnecting, the same place in the transaction log can be found. The name for the bookmark of a connection is specified in the `bookmarkStr` parameter.

For the first set of methods listed above, the `createBookmarkFlag` boolean parameter indicates whether the specified bookmark is new or was previously created. If you indicate that a bookmark is new (`createBookmarkFlag==true`) and it already exists, an error is returned. Similarly, if you indicate that a bookmark already exists (`createBookmarkFlag==false`) and it does not exist, an error is returned.

For the second set of methods listed, without `createBookmarkFlag`, TTClasses first tries to connect reusing the supplied bookmark (behavior equivalent to `createBookmarkFlag==false`). If that bookmark does not exist, TTClasses then tries to connect and create a new bookmark with the name `bookmarkStr` (behavior equivalent to `createBookmarkFlag==true`). These methods are provided as a convenience, to simplify XLA connection logic if you would rather not concern yourself with whether the XLA bookmark exists.

In either mode, with or without `createBookmarkFlag`, specify a user name and password either through the connection string or through the separate parameters, or specify a `DRIVER_COMPLETION_ENUM` value. Refer to [TTConnection Reference](#) for information about `DRIVER_COMPLETION_ENUM`.

Privilege to connect to a database must be granted to users through the `CREATE SESSION` privilege, either directly or through the `PUBLIC` role. See [Connection Methods](#). In addition, the XLA privilege is required to create an XLA connection.



Note:

Only one XLA connection can connect with a given bookmark name. An error is returned if multiple connections try to connect to the same bookmark.

deleteBookmarkAndDisconnect()

```
void deleteBookmarkAndDisconnect()
```

This method first deletes the bookmark that is currently associated with the connection, so that the database no longer keeps records relevant to that bookmark, then disconnects from the database.

To disconnect without deleting the bookmark, use the `Disconnect()` method instead.

Disconnect()

```
virtual void Disconnect()
```

This method closes an XLA connection to a database. The XLA bookmark persists after you call this method.

To delete the bookmark and disconnect from the database, use `deleteBookmarkAndDisconnect()` instead.

fetchUpdatesWait()

```
void fetchUpdatesWait(ttXlaUpdateDesc_t*** array, int maxrecs,
                    int* recsP, int seconds)
```

Use this method to fetch a set of records describing changes to a database. A list of `ttXlaUpdateDesc_t` structures is returned. If there are no XLA updates to be fetched, this method waits the specified number of seconds before returning.

Specify the number of seconds to wait, *seconds*, and the maximum number of records to receive, *maxrecs*. The method returns the number of records actually received, *recsP*, and an array of pointers, *arry*, that point to structures defining the changes.

The `ttXlaUpdateDesc_t` structures that are returned by this method are defined in the XLA specification. No C++ object-oriented encapsulation of these methods is provided. Typically, after calling `fetchUpdatesWait()`, an application processes these `ttXlaUpdateDesc_t` structures in a sequence of calls to `TTXlaTableList::HandleChange()`.

See `ttXlaUpdateDesc_t` in *Oracle TimesTen In-Memory Database C Developer's Guide*.

getBookmarkIndex()

```
void getBookmarkIndex()
```

This method gets the current bookmark location, storing it into a class private data member where it is available for use by subsequent `setBookmarkIndex()` calls.

setBookmarkIndex()

```
void setBookmarkIndex()
```

This method returns to the saved transaction log index, restoring the bookmark to the address previously acquired by a `getBookmarkIndex()` call. Use this method to replay a batch of XLA records.

Note that `ackUpdates()` invalidates the stored transaction log placeholder. After `ackUpdates()`, a call to `setBookmarkIndex()` returns an error because it is no longer possible to go back to the previously acquired bookmark location.

TTXlaRowViewer Reference

`TTXlaRowViewer`, which represents a row image from change notification records, is used to examine XLA change notification record structures and old and new column values.

- [Public Members](#)
- [Public Methods](#)

Also see [TTXlaRowViewer Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTXlaRowViewer` public methods.

Public Methods Summary

Method	Description
columnPrec()	Returns the precision of the specified column in the row image.
columnScale()	Returns the scale of the specified column in the row image.
Get()	Fetches the value of the specified column in the row image.
getColumn()	Returns the specified column from the row image.
isColumnTTTimestamp()	Indicates whether the specified column in the row image is a <code>TT_TIMESTAMP</code> column.
isNull()	Indicates whether the specified column in the row image has the value <code>NULL</code> .
numUpdatedCols()	Returns the number of columns in the row image that have been updated.
setTuple()	Associates the <code>TTXlaRowViewer</code> object with the specified row image.
updatedCol()	Returns the column number in the row image of a column that has been updated, typically during iteration through all updated columns.

columnPrec()

```
SQLULEN columnPrec(int cno)
```

Returns the precision of data in column number *cno* of the row image, referring to the maximum number of digits that are used by the data type.

columnScale()

```
int columnScale(int cno)
```

Returns the scale of data in column number *cno* of the row image, referring to the maximum number of digits to the right of the decimal point.

Get()

```
void Get(int cno, TYPE* valueP)
void Get(int cno, TYPE* valueP, SQLULEN* byteLenP)
```

Fetches the value of column number *cno* in the row image. These methods are very similar to the `TTCmd::getColumn()` methods.

Table 5-1 shows the supported SQL data types and the appropriate versions of `Get()` to use for each data type. Design the application according to the types of data that are stored. For example, data of type `NUMBER(9,0)` can be accessed by the `Get(int, int*)` method without loss of information.

Table 5-1 Get() Signatures for Supported Table Column Types

XLA Data Type	Database Data Type	Get() Variant
TTXLA_CHAR_TT	TT_CHAR	Get(int <i>cno</i> , char** <i>cPP</i>)
TTXLA_NCHAR_TT	TT_NCHAR	Get(int <i>cno</i> , SQLWCHAR** <i>wcPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_VARCHAR_TT	TT_VARCHAR	Get(int <i>cno</i> , char** <i>cPP</i>)
TTXLA_NVARCHAR_TT	TT_NVARCHAR	Get(int <i>cno</i> , SQLWCHAR** <i>wcPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_TINYINT	TT_TINYINT	Get(int <i>cno</i> , SQLTINYINT* <i>iP</i>)
TTXLA_SMALLINT	TT_SMALLINT	Get(int <i>cno</i> , short* <i>iP</i>)
TTXLA_INTEGER	TT_INTEGER	Get(int <i>cno</i> , int* <i>iP</i>)
TTXLA_BIGINT	TT_BIGINT	Get(int <i>cno</i> , SQLBIGINT* <i>biP</i>)
TTXLA_BINARY_FLOAT	BINARY_FLOAT	Get(int <i>cno</i> , float* <i>fP</i>)
TTXLA_BINARY_DOUBLE	BINARY_DOUBLE	Get(int <i>cno</i> , double* <i>dP</i>)
TTXLA_TIME	TT_TIME	Get(int <i>cno</i> , TIME_STRUCT* <i>tP</i>)
TTXLA_DATE_TT	TT_DATE	Get(int <i>cno</i> , DATE_STRUCT* <i>dP</i>)
TTXLA_TIMESTAMP_TT	TT_TIMESTAMP	Get(int <i>cno</i> , TIMESTAMP_STRUCT* <i>tsP</i>)

Table 5-1 (Cont.) Get() Signatures for Supported Table Column Types

XLA Data Type	Database Data Type	Get() Variant
TTXLA_BINARY	BINARY	Get(int <i>cno</i> , const void** <i>binPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_VARBINARY	VARBINARY	Get(int <i>cno</i> , const void** <i>binPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_NUMBER	NUMBER	Get(int <i>cno</i> , double* <i>dP</i>) Get(int <i>cno</i> , char** <i>cPP</i>) Get(int <i>cno</i> , short* <i>iP</i>) Get(int <i>cno</i> , int* <i>iP</i>) Get(int <i>cno</i> , SQLBIGINT* <i>biP</i>)
TTXLA_DATE	DATE	Get(int <i>cno</i> , TIMESTAMP_STRUCT* <i>tsP</i>)
TTXLA_TIMESTAMP	TIMESTAMP	Get(int <i>cno</i> , TIMESTAMP_STRUCT* <i>tsP</i>)
TTXLA_CHAR	CHAR	Get(int <i>cno</i> , char** <i>cPP</i>)
TTXLA_NCHAR	NCHAR	Get(int <i>cno</i> , SQLWCHAR** <i>wcPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_VARCHAR	VARCHAR2	Get(int <i>cno</i> , char** <i>cPP</i>)
TTXLA_NVARCHAR	NVARCHAR2	Get(int <i>cno</i> , SQLWCHAR** <i>wcPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_FLOAT	FLOAT	Get(int <i>cno</i> , double* <i>dP</i>) Get(int <i>cno</i> , char** <i>cPP</i>)

Table 5-1 (Cont.) Get() Signatures for Supported Table Column Types

XLA Data Type	Database Data Type	Get() Variant
TTXLA_BLOB	BLOB	Get(int <i>cno</i> , const void** <i>binPP</i> , SQLULEN* <i>byteLenP</i>)
TTXLA_CLOB	CLOB	Get(int <i>cno</i> , char** <i>cPP</i>)
TTXLA_NCLOB	NCLOB	Get(int <i>cno</i> , SQLWCHAR** <i>wcPP</i> , SQLULEN* <i>byteLenP</i>)

getColumn()

```
const TTXlaColumn* getColumn(u_int cno) const
```

Returns a `TTXlaColumn` object with metadata for column number *cno* in the row image.

isColumnTTTimestamp()

```
bool isColumnTTTimestamp(int cno)
```

Returns `TRUE` if column number *cno* in the row image is a `TT_TIMESTAMP` column, or `FALSE` otherwise.

isNull()

```
bool isNull(int cno)
```

Indicates whether the column number *cno* in the row image has the value `NULL`, returning `TRUE` if so or `FALSE` if not.

numUpdatedCols()

```
SQLUSMALLINT numUpdatedCols()
```

Returns the number of columns that have been updated in the row image.

setTuple()

```
void setTuple(ttXlaUpdateDesc_t* updateDescP, int whichTuple)
```

Before a row can be examined, this method must be called to associate the `TTXlaRowViewer` object with a particular row image. It is invoked inside the

TTXlaTableHandler::HandleInsert(), HandleUpdate(), or HandleDelete() method, or by a user-written overloaded method. You would typically call it when overloading the TTXlaTableHandler::HandleChange() method. The TimesTen Classic Quick Start xlasubscriber1 demo provides an example of its usage. (See [About TimesTen Quick Start and Sample Applications](#).)

The ttXlaUpdateDesc_t structures that are returned by TTXlaPersistConnection::fetchUpdatesWait() contain either zero, one, or two rows. Note the following:

- Structures that define a row that was inserted into a table contain the row image of the inserted row.
- Structures that define a row that was deleted from a table contain the row image of the deleted row.
- Structures that define a row that was updated in a table contain the images of the row before and after the update.
- Structures that define other changes to the table or the database contain no row images. For example, structures reporting that an index was dropped contain no row images.

The setTuple() method takes two arguments:

- A pointer to a particular ttXlaUpdateDesc_t structure defining a database change
- An integer specifying which type of row image in the update structure should be examined

The following are valid values:

- INSERTED_TUP: Examine the inserted row.
- DELETED_TUP: Examine the deleted row.
- UPDATE_OLD_TUP: Examine the row before it was updated.
- UPDATE_NEW_TUP: Examine the row after it was updated.

updatedCol()

```
SQLUSMALLINT updatedCol(u_int cno)
```

Returns the column number of a column that has been updated. For the input parameter you can iterate from 1 through *n*, where *n* is the number returned by numUpdatedCols(). The example that follows shows a snippet from the TimesTen Classic Quick Start demo xlasubscriber1, where updatedCol() is used with numUpdatedCols() to retrieve each column that has been updated. (See [About TimesTen Quick Start and Sample Applications](#).)

```
void
SampleHandler::HandleUpdate(ttXlaUpdateDesc_t* )
{
    cerr << row2.numUpdatedCols() << " column(s) updated: ";
    for ( int i = 1; i <= row2.numUpdatedCols(); i++ )
    {
        cerr << row2.updatedCol(i) << "("
            << row2.getColumn(row2.updatedCol(i)-1)->getColName() << ") ";
    }
}
```

```

    cerr << endl;
}

```

TTXlaTableHandler Reference

The `TTXlaTableHandler` class provides methods that enable and disable change tracking for a table. Methods are also provided to handle update notification records from XLA.

- [Public Members](#)
- [Protected Members](#)
- [Public Methods](#)

Also see [TTXlaTableHandler Usage](#).

Public Members

None

Protected Members

Member	Description
<code>TTXlaTable tbl</code>	This is for the metadata associated with the table being handled.
<code>TTXlaRowViewer row</code>	This is used to view the row being inserted or deleted, or the old image of the row being updated, in user-written <code>HandleInsert()</code> , <code>HandleDelete()</code> , and <code>HandleUpdate()</code> methods.
<code>TTXlaRowViewer row2</code>	This is used to view the new image of the row being updated in user-written <code>HandleUpdate()</code> methods.

Public Methods

This section summarizes then describes the `TTXlaTableHandler` public methods.

Public methods summary

Method	Description
DisableTracking()	Disables XLA update tracking for the table.
EnableTracking()	Enables XLA update tracking for the table.
generateSQL()	Returns the SQL associated with a given XLA record.
HandleChange()	Dispatches a record from <code>ttXlaUpdateDesc_t</code> to the appropriate handling routine for processing.
HandleDelete()	This is invoked when the <code>HandleChange()</code> method is called to process a delete operation.
HandleInsert()	This is invoked when the <code>HandleChange()</code> method is called to process an insert operation.

Method	Description
HandleUpdate()	This is invoked when the <code>HandleChange()</code> method is called to process an update operation.

DisableTracking()

```
virtual void DisableTracking()
```

Disables XLA update tracking for the table. After this method is called, the XLA bookmark no longer captures information about changes to the table.

EnableTracking()

```
virtual void EnableTracking()
```

Enables XLA update tracking for the table. Until this method is called, the XLA bookmark does not capture information about changes to the table.

generateSQL()

```
void generateSQL (ttXlaUpdateDesc_t* updateDescP, char* buffer,
                  SQLINTEGER maxByteLen, SQLINTEGER* actualByteLenP)
```

This method prints the SQL associated with a given XLA record. The SQL string is returned through the *buffer* parameter. Allocate space for the buffer and specify its maximum length, *maxByteLen*. The *actualByteLenP* parameter returns information about the actual length of the SQL string returned.

If *maxByteLen* is less than the length of the generated SQL string, a `TTStatus` error is thrown and the contents of *buffer* and *actualByteLenP* are not modified.

HandleChange()

```
virtual void HandleChange(ttXlaUpdateDesc_t* updateDescP)
virtual void HandleChange(ttXlaUpdateDesc_t* updateDescP, void* pData)
```

Dispatches a `ttXlaUpdateDesc_t` object to the appropriate handling routine for processing. The update description is analyzed to determine if it is for a delete, insert or update operation. The appropriate handling method is then called: `HandleDelete()`, `HandleInsert()`, or `HandleUpdate()`.

Classes that inherit from `TTXlaTableHandler` can use the optional *pData* parameter when they overload the `TTXlaTableHandler::HandleChange()` method. This optional parameter is useful for determining whether the batch of XLA records that was just processed ends on a transaction boundary. Knowing this helps an application decide the appropriate time to invoke `TTConnection::ackUpdates()`. See [Acknowledging XLA Updates at Transaction Boundaries](#) for an example that uses the *pData* parameter.

Also see [HandleChange\(\)](#) for `TTXlaTableList` objects.

HandleDelete()

```
virtual void HandleDelete(ttXlaUpdateDesc_t* updateDescP) = 0
```

This method is invoked whenever the `HandleChange()` method is called to process a delete operation.

`HandleDelete()` is not implemented in the `TTXlaTableHandler` base class. It must be implemented by any classes derived from it, with appropriate logic to handle deleted rows.

The row that was deleted from the table is available through the protected member `row` of type `TTXlaRowViewer`.

HandleInsert()

```
virtual void HandleInsert(ttXlaUpdateDesc_t* updateDescP) = 0
```

This method is invoked whenever the `HandleChange()` method is called to process an insert operation.

`HandleInsert()` is not implemented in the `TTXlaTableHandler` base class. It must be implemented by any classes derived from it, with appropriate logic to handle inserted rows.

The row that was inserted into the table is available through the protected member `row` of type `TTXlaRowViewer`.

HandleUpdate()

```
virtual void HandleUpdate(ttXlaUpdateDesc_t* updateDescP) = 0
```

This method is invoked whenever the `HandleChange()` method is called to process an update operation.

`HandleUpdate()` is not implemented in the `TTXlaTableHandler` base class. It must be implemented by any classes derived from it, with appropriate logic to handle updated rows.

The previous version of the row that was updated from the table is available through the protected member `row` of type `TTXlaRowViewer`. The new version of the row is available through the protected member `row2`, also of type `TTXlaRowViewer`.

TTXlaTableList Reference

The `TTXlaTableList` class provides a list of `TTXlaTableHandler` objects and is used to dispatch update notification events to the appropriate `TTXlaTableHandler` object.

- [Public Members](#)
- [Public Methods](#)

Also see [TTXlaTableList Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTXlaTableList` public methods.

Public methods summary

Method	Description
add()	Adds a <code>TTXlaTableHandler</code> object to the list.
del()	Deletes a <code>TTXlaTableHandler</code> object from the list.
HandleChange()	Processes a record obtained from a <code>ttXlaUpdateDesc_t</code> structure.

add()

```
void add(TTXlaTableHandler* tblh)
```

Adds a `TTXlaTableHandler` object to the list.

del()

```
void del(TTXlaTableHandler* tblh)
```

Deletes a `TTXlaTableHandler` object from the list.

HandleChange()

```
void HandleChange(ttXlaUpdateDesc_t* updateDescP)  
void HandleChange(ttXlaUpdateDesc_t* updateDescP, void* pData)
```

When a `ttXlaUpdateDesc_t` object is received from XLA, it can be processed by calling this method, which determines which table the record references and calls the `HandleChange()` method of the appropriate `TTXlaTableHandler` object.

See [HandleChange\(\)](#) for `TTXlaTableHandler` objects, including a discussion of the `pData` parameter.

TTXlaTable Reference

The `TTXlaTable` class encapsulates the metadata for a table being monitored for changes.

- [Public Members](#)
- [Public Methods](#)

Also see [TTXlaTable Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTXlaTable` public methods.

Public methods summary

Method	Description
getColNumber()	Returns the column number of the specified column in the table.
getNCols()	Returns the number of columns in the table.
getOwnerName()	Returns the name of owner of the table.
getTableName()	Returns the name of the table.

getColNumber()

```
int getColNumber(const char* colNameP) const
```

For a specified column name in the table, this method returns its column number, or -1 if there is no column by that name.

getNCols()

```
int getNCols() const
```

Returns the number of columns in the table.

getOwnerName()

```
const char* getOwnerName() const
```

Returns the user name of the owner of the table.

getTableName()

```
const char* getTableName() const
```

Returns the name of the table.

TTXlaColumn Reference

A `TTXlaColumn` object contains the metadata for a single column of a table being monitored for changes.

- [Public Members](#)

- [Public Methods](#)

Also see [TTXlaColumn Usage](#).

Public Members

None

Public Methods

This section summarizes then describes the `TTXlaColumn` public methods.

Public Methods Summary

Method	Description
getColName()	Returns the name of the column.
getPrecision()	Returns the precision of the column.
getScale()	Returns the scale of the column.
getSize()	Returns the size of the column data, in bytes.
getSysColNum()	Returns the system-generated column number of this column as stored in the database.
getType()	Returns the data type of the column, as an integer.
getUserColNum()	Returns a column number optionally specified by the user, or 0.
isNullable()	Indicates whether the column allows <code>NULL</code> values.
isPKColumn()	Indicates whether the column is the primary key for the table.
isTTTimestamp()	Indicates whether the column is a <code>TT_TIMESTAMP</code> column.
isUpdated()	Indicates whether the column was updated.

getColName()

```
const char* getColName() const
```

Returns the name of the column.

getPrecision()

```
SQLULEN getPrecision() const
```

Returns the precision for data in the column, referring to the maximum number of digits that are used by the data type.

getScale()

```
int getScale() const
```

Returns the scale for data in the column, referring to the maximum number of digits to the right of the decimal point.

getSize()

```
SQLINTEGER getSize() const
```

Returns the size of values in the column, in bytes.

getSysColNum()

```
SQLINTEGER getSysColNum() const
```

This is the system-generated column number of the column, numbered from 1. It equals the corresponding `COLNUM` value in `SYS.COLUMNS`. (See `SYS.COLUMNS` in *Oracle TimesTen In-Memory Database System Tables and Views Reference*.)

getType()

```
int getType() const
```

Returns an integer representing the TimesTen XLA data type (`TTXLA_XXX`) of the column. This is a value from the `dataType` field of the TimesTen `ttXlaColDesc_t` data structure. In some cases this corresponds to an ODBC SQL data type (`SQL_XXX`) and the corresponding standard integer value.

Refer to XLA Data Types in *Oracle TimesTen In-Memory Database C Developer's Guide*. The corresponding integer values are defined for use in any TTClasses application that includes the `TTXla.h` header file.

Also refer to `ttXlaColDesc_t` in *Oracle TimesTen In-Memory Database C Developer's Guide*.

getUserColNum()

```
SQLINTEGER getUserColNum() const
```

Returns a column number optionally specified by the user through the `ttSetUserColumnID` TimesTen built-in procedure, or 0.

See `ttSetUserColumnID` in *Oracle TimesTen In-Memory Database Reference*.

isNullable()

```
bool isNullable() const
```

Returns `TRUE` if null values are allowed in the column, or `FALSE` otherwise.

isPKColumn()

```
bool isPKColumn() const
```

Returns `TRUE` if this column is the primary key for the table, or `FALSE` otherwise.

isTTTimestamp()

```
bool isTTTimestamp() const
```

Returns `TRUE` if this column is a `TT_TIMESTAMP` column, or `FALSE` otherwise.

isUpdated()

```
bool isUpdated() const
```

Returns `TRUE` if this column was updated, or `FALSE` otherwise.