Oracle® TimesTen In-Memory Database

TTClasses Guide

Release 11.2.1

E13074-02

August 2009

Oracle TimesTen In-Memory Database TTClasses Guide, Release 11.2.1

E13074-02

Copyright © 2006, 2009, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

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

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

Contents

Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, Oracle TimesTen In-Memory Database operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

TimesTen supports a variety of programming interfaces, including ODBC (Open DataBase Connectivity), OCI (Oracle Call Interface), Oracle PL/SQL, and Oracle Pro*C/C++ (precompiler for embedded SQL and PL/SQL instructions in C or C++ code).

The TimesTen C++ Interface Classes (TTClasses) library was written to provide an easy-to-use, high-performance interface to Oracle TimesTen In-Memory Database. This C++ class library provides wrappers around the most common ODBC functionality.

Audience

This guide is for application developers who use and administer TimesTen ODBC.

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

Related documents

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

`http://www.oracle.com/technology/documentation/timesten_doc.html`

Conventions

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

This document uses the following text conventions:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

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

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

Convention	Meaning
`install_dir`	The path that represents the directory where the current release of TimesTen is installed.
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique alphanumeric instance name. This name appears in the install path.
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
`release` or *`rr`*	Three numbers that represent the first three numbers of the TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.
`jdk_version`	Two digits that represent the version number of the major JDK release. Specifically, 14 represent JDK 1.4; 5 represents JDK 5.
`DSN`	The data source name.

Documentation Accessibility

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Technical support

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

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

1 Introduction to TTClasses

This chapter includes the following topics:

- Overview of TTClasses
- Scope of TTClasses

Overview of TTClasses

The Oracle TimesTen In-Memory Database provides high performance through a standard ODBC and SQL interface. Unlike other RDBMS implementations of ODBC, access to TimesTen row ODBC is extremely fast. The TimesTen C++ Interface Classes (TTClasses) was developed to meet the demand for an API that is easier to use than ODBC, but does not sacrifice performance. This C++ class library provides wrappers around the most common ODBC functionality. Using this library allows easier interaction with TimesTen data stores.

In addition, the TTClasses library is 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 when compared to hand-coded ODBC.

Scope of TTClasses

TTClasses is a wrapper around all major ODBC functionality.

In addition to providing a C++ interface to TimesTen's ODBC interface, TTClasses also provides an interface to the TimesTen's Transaction Log API (XLA). XLA allows an application to monitor one or more tables in a TimesTen data store. When other applications change that table, the changes are reported through XLA to the monitoring application. TTClasses provides an easy-to-use interface to the most commonly used aspects of XLA functionality. For more information about XLA, see the chapter "XLA and TimesTen Event Management" in the *Oracle TimesTen In-Memory Database C Developer's Guide*.

Note: TTClasses has been integrated with TimesTen since release 6.0. Previous versions of TTClasses were distributed separately from TimesTen, so earlier versions of TTClasses were compatible with multiple TimesTen versions. Starting with TimesTen 6.0, TTClasses is no longer tested or supported in combination with any other TimesTen release besides the release that it ships with.

2 Compiling TTClasses

TTClasses comes compiled and preconfigured during TimesTen installation. To recompile TTClasses, use the `make` (UNIX) or `nmake` (Windows) command. This chapter describes the compilation process and introduces the Quick Start demo applications for TTClasses.

Compiling TTClasses on UNIX

To use TTClasses, ensure that your shell environment variables are set correctly. You can optionally run one of the following scripts or add a call to one of these scripts in your login initialization script (`.profile` or `.cshrc`), where *`install_dir`* is the installation directory for your TimesTen instance:

```
install_dir/bin/ttenv.sh    (sh/ksh/bash)
install_dir/bin/ttenv.csh   (csh/tcsh)
```

After your PATH and shared library load path are configured properly, you can change to the TTClasses directory and recompile TTClasses:

```
$ cd install_dir/ttclasses
$ make
```

Compiling TTClasses on Windows

On Windows, you can either set environment variables during installation or run:

```
install_dir\bin\ttenv.bat
```

If you want to recompile TTClasses, change to the `ttclasses` directory, where *`install_dir`* is the installation directory for your TimesTen instance:

```
install_dir\ttclasses
```

Before recompiling, ensure that the PATH, INCLUDE, and LIB environment variables point to the correct Visual Studio directories. There is a batch file you can run, named `VCVARS32.BAT` (Visual C++ 6.0) or `VSVARS32.BAT` (Visual Studio .NET) in the Visual Studio directory tree, that will set up your PATH, INCLUDE, and LIB environment variables correctly.

Then, to recompile TTClasses if you are using Visual Studio .NET:

```
install_dir\ttclasses> nmake /f Makefile.vsdotnet
```

Or if you are using VC++ 6.0:

```
install_dir\ttclasses> nmake
```

Compilation options

The following "make target" options are available when you compile TTClasses:

- `all`: Build a shared optimized library
- `shared_opt`: Build a shared optimized library
- `shared_debug`: Build a shared debug library
- `static_opt`: Build a static optimized library
- `static_debug`: Build a static debug library
- `opt`: Build the optimized libraries (shared and static)
- `debug`: Build the debug libraries (shared and static)
- `clean`: Delete the TTClasses libraries and object files

Note: If you do not specify an option when you compile, the default is `all`.

To specify a make target, use the name of the make target on the command line.

To build a shared, debug version of TTClasses:

(Unix)

```
$ make clean shared_debug
```

(Windows)

```
install_dir\ttclasses> nmake clean shared_debug
```

Compiling TTClasses for client/server mode

To compile TTClasses for client/server mode, use the `MakefileCS` Makefile.

Example 2–1 shows how to build a client/server version of TTClasses.

Example 2–1 TTClasses compilation for client/server mode

(Unix)

```
$ make -f MakefileCS clean all
```

(Windows)

```
install_dir\ttclasses> nmake /f MakefileCS clean all
```

Installing TTClasses after compilation (UNIX only)

After compilation, install the library so all users of the TimesTen instance can use TTClasses. This step is not part of compilation because different privileges are required for installing TTClasses than for compiling TTClasses.

Example 2–2 shows the steps to install the TTClasses library on a UNIX or Linux system. Note that installation occurs automatically after compilation on Microsoft Windows.

Example 2–2 Installation of TTClasses library

```
$ cd install_dir/ttclasses
$ make install
```

TTClasses compiler macros

Most users do not need to manipulate the TTClasses Makefile. If you need to modify the TTClasses Makefile manually, you can add flags for the TTClasses compiler macros to the Makefile. For Unix, add `-D`*`flagname`*; for Windows, add `/D`*`flagname`*.

This section includes information about the following compiler macros:

- TTEXCEPT: Throw C++ exceptions
- Flags for C++ I/O streams: TTC_USE_STRINGSTREAM, USE_OLD_CPP_STREAMS
- TTDEBUG:Generate additional debugging and error checking logic
- TT_64BIT: Use TTClasses with 64-bit TimesTen

Also see "Platform-specific compiler macros" on page 2-4.

TTEXCEPT: Throw C++ exceptions

Compile TTClasses with the `-DTTEXCEPT` flag to make TTClasses throw C++ exceptions. Put try/catch blocks around all TTClasses function calls and catch exceptions of type `TTStatus`. See "TTStatus" on page 4-1.

Flags for C++ I/O streams: TTC_USE_STRINGSTREAM, USE_OLD_CPP_STREAMS

There are multiple types of C++ streams, and they are not compatible with each other. TimesTen provides two related flags. The types of streams you use in your application determines which flag to use, or whether you should use neither, as follows:

- For relatively new (but not the newest) types of streams, where you are including `<iostream>` and using the `ostrstream` class, use neither flag. This is the default for most platforms and compilers.
- For the newest types of streams, where you are including `<iostream>` and using the `ostringstream` class, use the TTC_USE_STRINGSTREAM flag.
- For old types of streams, where you are including `<iostream.h>` and using the `ostrstream` class, use the USE_OLD_CPP_STREAMS flag. This is the default for some older platforms and compilers.

Check your TTClasses Makefile to see if either flag is set. If the flags are not set properly, then update the Makefile as appropriate, recompile TTClasses, and replace the previous TTClasses library file with the recompiled one.

Further information is provided in the subsections that follow.

Neither flag: for C++ I/O stream code with ostrstream

As noted above, if you are using C++ I/O stream code where you are including `<iostream>` and using the `ostrstream` class, neither the TTC_USE_STRINGSTREAM flag nor the USE_OLD_CPP_STREAMS flag should be set. If either is set in the TTClasses Makefile, remove the line and recompile TTClasses.

TTC_USE_STRINGSTREAM: for newest C++ I/O stream code with ostringstream

This compiler flag is for use with C++ compilers that reliably support the newest C++ stream types and the `ostringstream` class. If your program uses the newest stream types, then TTClasses must be compiled with the `-DTTC_USE_STRINGSTREAM` setting. This allows TimesTen to use `ostringstream` and `<iostream>` for compatibility with your program code.

Also note that if your program uses the newest C++ streams, the USE_OLD_CPP_ STREAMS flag must *not* be set. If the TTClasses Makefile has the wrong settings, correct them and recompile TTClasses.

Note: With `gcc` version 3.2 or higher, the TTC_USE_ STRINGSTREAM flag is set by default in the file *`install_dir`*`/include/ttclasses/TTIostream.h`.

USE_OLD_CPP_STREAMS: for old C++ I/O stream code

This compiler flag is for older C++ compilers that do not support `<iostream>`. If your program uses old C++ streams, where you include `<iostream.h>` and use `ostrstream`, then TTClasses must be compiled with the `-DUSE_OLD_CPP_STREAMS` setting to be compatible with your program code.

Also note that if your program uses old C++ streams, the TTC_USE_STRINGSTREAM flag must *not* be set. If the TTClasses Makefile has the wrong settings, correct them and recompile TTClasses.

TTDEBUG:Generate additional debugging and error checking logic

Compile TTClasses with `-DTTDEBUG` to generate extra debugging information. This extra information reduces performance slightly, so use this flag only in development (not production) systems.

TT_64BIT: Use TTClasses with 64-bit TimesTen

Compile TTClasses with `-DTT_64BIT` if you are writing a 64-bit TimesTen application.

Note that 64-bit TTClasses has been tested on AIX, HP-UX, Solaris, Red Hat Linux, and Tru64.

Platform-specific compiler macros

The following compiler macros are specific to a particular platform or compiler combination. You should not have to specify these compiler macros manually. Their use is determined by the Makefile chosen by the `configure` program.

GCC

Compile TTClasses with the `-DGCC` flag when using `gcc` on any platform.

HPUX

Compile TTClasses with the `-DHPUX` flag when compiling on HP-UX.

About the TimesTen TTClasses demos

After you have configured your C++ environment, you can confirm that everything is set up correctly by compiling and running the TimesTen Quick Start demo applications. Refer to the Quick Start welcome page at `*install_dir*/quickstart.html`, especially the links under Sample Programs, for information about the following:

- Demo schema and setup: The `build_sampledb` script creates a sample database and demo schema. You must run this before you start using the demos.
- Demo environment and setup: The `ttquickstartenv` script, a superset of the `ttenv` script generally used for TimesTen setup, sets up the demo environment. You must run this each time you enter a session where you want to compile and run any of the demos.
- Demos and setup: TimesTen provides demos for TTClasses and XLA in subdirectories under the `*install_dir*/quickstart/sample_code` directory. For instructions on compiling and running the demos, see the README files in the subdirectories.
- What the demos do: A synopsis of each demo is provided.

3 Using TTClasses

This chapter contains brief descriptions of the recommended way to use TTClasses. It includes the following topics:

- Using TTCmd, TTConnection, and TTStatus
- TTClasses logging
- TTClasses XLA classes

Using TTCmd, TTConnection, and TTStatus

While TTClasses can be used in a number of ways, the following general approach has been 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 that will be used by an application are prepared when the application begins, using separate TTCmd objects for each statement. In ODBC (and thus in the C++ classes), statements are bound to a particular connection, so a full set of all statements used by the application will often be associated with every connection to the TimesTen database.

An easy way to accomplish this is to develop an application-specific class that is derived from TTConnection. For an application called `XYZ`, you can create a class called `XYZConnection`, derived from TTConnection. The `XYZConnection` class contains private TTCmd members representing the prepared SQL statements that can be used in the application. In addition, the `XYZConnection` class provides new public methods to implement the application-specific database functionality, which can be implemented using the private TTCmd members.

Example 3–1 Definition of a Connection Class

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

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

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

In the preceding 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 application (storing account balances, for example). The implementation of these three methods can use `updateData()`, `insertData()`, and `queryData()` provided in TTCmd to implement the specific functionality of the application.

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

Example 3–2   Definition of a Connect() method

This example shows the `XYZConnection::Connect()` method.

```
void
XYZConnection::Connect(const char* connStr, TTStatus&
                       stat)
{
  TTStatus stat2;

  try {
    TTConnection::Connect(connStr, stat);
    updateData.Prepare(this,
                       "update mydata v
                       "set foo = ? where bar = ?",
                       stat);
    insertData.Prepare(this,
                       "insert into mydata "
                       "values(?,0)", stat);
    queryData.Prepare(this,
                     "select i from mydata where name "
                     " = ?", stat);
    Commit(stat);
  }
  catch (TTStatus st) {
    cerr << "Error in XYZConnection::Connect: " << st
         << endl;
    Rollback(stat2);
  }
  return;
}
```

This `Connect()` method causes the `XYZConnection` to be made fully operational. The application-specific methods are fully functional after `Connect()` has been called.

This approach to application design works well with the design of the TTConnectionPool class. The application can create numerous objects of type `XYZConnection` and can add them to TTConnectionPool. By calling TTConnectionPool`::ConnectAll()`, the application can cause all connections in the pool to be connected to the database, as well as causing all SQL statements to be prepared, in a single line of code.

This approach to application design allows the database components of an application to be separated from the remainder of the application; only the `XYZConnection` class contains database-specific code.

An example of this type of design can be found in several of the sample programs that are included with TTClasses. The simplest example is `sample.cpp`.

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.

TTClasses logging

TTClasses has a logging facility that allows applications to capture useful debugging information about running TTClasses programs. TTClasses logging is done on at the process level. You can enable logging for a specific process and produce a single output log stream for the process. TTClasses logging is disabled by default.

TTClasses supports different levels of logging information. See Example 3–4 on page 3-4 for more information about what is printed at each log level.

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

Note that at the more verbose log levels (INFO and DEBUG), so much log data is generated that application performance is adversely affected. We strongly discourage using 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. Example 3–3 demonstrates how to print TTClasses log information at log level WARN into the `/tmp/ttclasses.log` output file.

Example 3–3   Printing TTClasses log information

```
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. You can change the `sample.cpp` program to use different log levels so you can see how errors are printed at log level ERROR and how huge amounts of logs are generated at log levels INFO and DEBUG.

See "TTGlobal" on page 4-25 for more information about using the `TTGlobal` class for logging.

TTClasses XLA classes

The Transaction Log API (XLA) is a set of functions that enable you to implement applications that:

- Monitor TimesTen for changes to specified tables in a local data store
- Receive real-time notification of these changes

One of the purposes of XLA is to provide a high-performance, asynchronous alternative to triggers.

For additional information about XLA, see the chapter "XLA and TimesTen Event Management" in the *Oracle TimesTen In-Memory Database C Developer's Guide*.

TTClasses XLA demos

TimesTen provides TTClasses XLA demos in the following location:

- `install_dir/quickstart/sample_code/ttclasses/xla`

Refer to "About the TimesTen TTClasses demos" on page 2-5.

The README file in the `ttclasses` directory contains instructions for building and running the TTClasses XLA demos (as well as other TTClasses demos).

Acknowledging XLA updates at transaction boundaries

XLA returns notification of changes to specific tables in the database, as well as 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.

Example 3–4 TTClasses XLA program

This example shows a typical main loop of a TTClasses XLA program.

```
TTXlaPersistConnection conn; // XLA connection
TTXlaTableList list(&conn); // tables being monitored
ttXlaUpdateDesc_t ** arry; // ptr to returned XLA recs
TTStatus stat;
int records_fetched;
// ...

loop {
  // fetch the updates
  conn.fetchUpdatesWait(&arry, MAX_RECS_TO_FETCH,
                             &records_fetched, ...);

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

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

Inside the `HandleChange()` method, depending on whether the record is an insert, update, or delete, the appropriate method of 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 the example loop 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()`.

Under typical circumstances of only a few records per transaction, this is not an issue. When you ask XLA to return at most 1000 records with the `fetchUpdatesWait()` method, usually only a few records are returned. 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. When you pull the XLA records out a few at a time, XLA usually makes sure that the last record returned is on a transaction boundary.

In summary: if you ask for 1000 records from XLA, and XLA returns only 15, it is highly probable that the 15th record is at the end of a transaction.

XLA guarantees one of the following:

- *Either* a batch of records will end with a completed transaction (perhaps multiple transactions in a single batch of XLA records)
- *Or* a batch of records will contain a partial transaction, with no completed transactions in the same batch, and that subsequent batches of XLA records will be returned for that single transaction until its transaction boundary has been reached.

Careful XLA applications need to verify whether the last record in a batch of XLA records has a transaction boundary and call `ackUpdates()` only on this transaction boundary. This is especially important when operations involve a large number of rows. If a bulk insert/update/delete operation has been performed on the database and the XLA application asks for 1000 records, it might receive all 1000 records (or fewer than 1000). The last record returned through XLA will probably *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, because `ackUpdates()` is a relatively expensive operation. Careful XLA applications should make sure to call this method only on a transaction boundary, so that when the application or system or database fails, the XLA bookmark is at the start of a transaction after the system recovers.

The `HandleChange()` method has a second parameter to allow passing information between `HandleChange()` and the main XLA loop. Compare Example 3–4 with Example 3–5 (the `do_acknowledge` setting, and the `&do_acknowledge` parameter of the `HandleChange()` call).

Example 3–5 TTClasses XLA program using transaction boundaries

In this example, `ackUpdates()` is called only when the `do_acknowledge` flag indicates that this batch of XLA records is at a transaction boundary.

```
TTXlaPersistConnection conn; // XLA connection
TTXlaTableList list(&conn); // tables being monitored
ttXlaUpdateDesc_t ** arry; // ptr to returned XLA recs
TTStatus stat;
int records_fetched;
int do_acknowledge;
// ...
loop {
  // fetch the updates
  conn.fetchUpdatesWait(&arry, MAX_RECS_TO_FETCH,
                              &records_fetched, ...);

  do_acknowledge = FALSE;

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

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

In addition to this change to the XLA main loop, the `HandleChange()` method needs to be overwritten to use `ttXlaUpdateDesc_t`.

4 Class Descriptions

This chapter contains descriptions of all classes in the external interface to TTClasses and brief descriptions of some of the internal TTClasses. It is divided into the following sections:

- Commonly used TTClasses
- System catalog classes
- XLA classes
- Internal classes

Commonly used TTClasses

This section includes the following classes:

- TTStatus
- TTConnection
- TTCmd
- TTConnectionPool
- TTGlobal

TTStatus

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

Subclasses

`TTStatus` has the following subclasses:

- TTError
- TTWarning

TTError `TTError` is a subclass of `TTStatus` and is used to encapsulate ODBC errors (return codes: SQL_ERROR, SQL_INVALID_HANDLE).

TTWarning `TTWarning` is a subclass of `TTStatus` and is used to encapsulate ODBC warnings (return code: SQL_SUCCESS_WITH_INFO).

ODBC warnings are usually not as serious as ODBC errors and should be handled with different logic. Logging ODBC warnings to an application's log is usually appropriate, but ODBC errors usually need to be programmatically handled.

Public members

Member	Description
`rc`	Return code from the failing ODBC call. Typical values for this field are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_NO_DATA_FOUND, and SQL_INVALID_HANDLE.
`native_error`	TimesTen native error number (if any) for the failing ODBC call.
`odbc_error`	ODBC error code for the failing ODBC call.
`err_msg`	ASCII printable error message for the failing ODBC call.

Public methods

Method	Description
ostream()	Prints errors to a stream.

ostream()

```
friend ostream& operator<<(ostream&, TTStatus&)
```

This method can be used to print the error to a stream.

Usage

You must build the library with the TTEXCEPT preprocessor variable defined. `TTStatus` objects are thrown as exceptions whenever an error occurs. This allows C++ applications to use `{try/catch}` to detect and recover from failure.

Example 4–1 shows the use of `TTStatus` with TTEXCEPT defined.

Example 4–1 Using TTStatus with TTEXCEPT defined

```
try {
  cmd1.Prepare(&conn, "select * from foo", stat);
  cmd2.Prepare(&conn, "insert into foo values(?,?,?)",
               stat);
  cmd3.Prepare(&conn, "update foo set x = ? where y=?",
               stat);
  conn.Commit(stat);
}
catch (TTStatus st) {
  cerr << "Error preparing statements: " << st << endl;
  // Rollback, exit(), throw -- whatever is appropriate
}
```

Example 4–2 Error reporting with TTError and TTWarning versus TTStatus

`TTError` objects are thrown for ODBC errors, and `TTWarning` objects are thrown for ODBC warnings. This example shows `TTError` and `TTWarning` for error handling.

```
// catching TTError & TTWarning exceptions
try {
  // some TTClasses method calls
}
catch (TTWarning warn) {
   cerr << "Warning encountered: " << warn << endl;
}
catch (TTError err) {
    cerr << "Error encountered: " << err << endl;
}
```

TTConnection

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

Usage

All applications that use TimesTen must create at least one `TTConnection` object.

Multithreaded applications that wish to use TimesTen from multiple threads simultaneously must create more than one `TTConnection` object. 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 the "TTConnectionPool" on page 4-23 for additional information about this option.

A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating a child process, the child cannot use the same connection.

It is not desirable for an application to be constantly connecting to and disconnecting from TimesTen because it degrades performance. Instead, establish database connections at the beginning of the application process and reuse them for the life of the process.

Public members

None.

Public methods

Method	Description
Connect()	Opens a new connection to a TimesTen data store.
Disconnect()	Closes a connection to a TimesTen data store.
Rollback()	Rolls back changes made to the database through this connection since the last call to `Commit()` or `Rollback()` methods.
isConnected()	Returns TRUE if the object is connected to TimesTen.
getHdbc()	Returns the ODBC level "HDBC" associated with this connection.
SetIsoReadCommitted()	Sets the transaction isolation level of the connection to be TXN_READ_COMMITTED.
SetIsoSerializable()	Sets the transaction isolation level of the connection to be TXN_SERIALIZABLE.

Method	Description
CheckpointBlocking()	Performs a blocking checkpoint operation on the data store by calling the `ttCkptBlocking` TimesTen built-in procedure.
CheckpointNonBlocking()	Performs a "true fuzzy" checkpoint operation on the data store by calling the `ttCkpt` TimesTen built-in procedure.
DurableCommit()	Performs a durable commit operation on the data store.
SetLockWait()	Sets the lock timeout interval for the connection by calling the `ttLockWait` TimesTen built-in procedure.
SetPrefetchCloseOn()	Turns on the TT_PREFETCH_CLOSE connection option. This is useful for optimizing SELECT query performance for client/server connections to TimesTen.
SetPrefetchCloseOff()	Turns off the TT_PREFETCH_CLOSE connection option.
SetPrefetchCount()	Allows a user application to tune the number of rows that the TimesTen ODBC driver SQLFetch call will prefetch for a SELECT statement.
SetAutocommitOff()	Sets AUTOCOMMIT off for the connection.
SetAutoCommitOn()	Sets AUTOCOMMIT on for the connection.
GetTTContext()	Returns the connection's context value.

Connect() `void Connect(const char* connStr, TTStatus&)`

Opens a new connection to a TimesTen data store. The connection string specified in the *`connStr`* parameter is used to create the connection.

Example 4–3 Using the Connect() method and checking for errors

A `TTStatus` object is thrown as an exception if an error occurs. When calling this method, any warnings returned can often be safely ignored. The following logic is preferred when using the `Connect()` method.

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

```
TTConnection conn;
TTStatus stat;
...

try {
  conn.Connect("DSN=mydsn", stat);
}
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(TTStatus&)`

Closes a connection to a TimesTen data store. A `TTStatus` object is thrown as an exception if an error occurs.

Commit()

```
void Commit(TTStatus&)
```

Commits a transaction to the TimesTen database. All other operations performed on this connection since the last call to the `Commit()` or `Rollback()` methods will be committed. A `TTStatus` object is thrown as an exception if an error occurs.

Rollback()

```
void Rollback(TTStatus&)
```

Abandons a transaction. Any changes made to the database through this connection since the last call to `Commit()` or `Rollback()` methods will be undone. A `TTStatus` object is thrown as an exception if an error occurs.

isConnected()

```
bool isConnected()
```

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

getHdbc()

```
HDBC getHdbc()
```

Returns the ODBC level "HDBC" associated with this connection.

SetIsoReadCommitted()

```
void SetIsoReadCommitted(TTStatus &)
```

Sets the transaction isolation level of the connection to be TXN_READ_COMMITTED. Read-committed isolation offers the best combination of single-transaction performance and good multiconnection concurrency.

SetIsoSerializable()

```
void SetIsoSerializable(TTStatus &)
```

Sets the transaction isolation level of the connection to be TXN_SERIALIZABLE. In general, serializable isolation offers fair individual transaction performance but extremely poor concurrency. READ_COMMITTED isolation level (see "SetIsoReadCommitted()" on page 4-5) should be preferred over SERIALIZABLE isolation level in almost all situations.

CheckpointBlocking()

```
void CheckpointBlocking(int timeout, int retries, TTStatus &)
```

Performs a blocking checkpoint operation on the data store by calling the `ttCkptBlocking` TimesTen built-in procedure with the *`timeout`* and *`retries`* parameters.

See the *Oracle TimesTen In-Memory Database Reference* for more information about `ttCkptBlocking`.

CheckpointNonBlocking()

```
void CheckpointNonBlocking(TTStatus &)
```

Note: This is the preferred type of checkpoint.

Performs a "true fuzzy" checkpoint operation on the data store by calling the `ttCkpt` TimesTen built-in procedure.

See the *Oracle TimesTen In-Memory Database Reference* for more information about `ttCkpt`.

DurableCommit()

```
void DurableCommit(TTStatus &)
```

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

See the *Oracle TimesTen In-Memory Database Reference* for more information about `ttDurableCommit`.

SetLockWait()

```
void SetLockWait(int secs, TTStatus &)
```

Sets the lock timeout interval for the connection by calling the `ttLockWait` TimesTen built-in procedure with the *`secs`* parameter. In general, a 2 or 3 second lock timeout is sufficient for most applications. The default lock timeout interval is 10 seconds.

See the *Oracle TimesTen In-Memory Database Reference* for more information about `ttLockWait`.

SetPrefetchCloseOn()

```
void SetPrefetchCloseOn(TTStatus &)
```

Turns on the TT_PREFETCH_CLOSE connection option, which is useful for optimizing SELECT query performance for client/server connections to TimesTen. Note that this method provides no benefit for directly connected TimesTen applications (for example, non-client/server programs).

See "Bulk fetch rows of TimesTen data" in the *Oracle TimesTen In-Memory Database C Developer's Guide* for more information about TT_PREFETCH_CLOSE.

SetPrefetchCloseOff()

```
void SetPrefetchCloseOff(TTStatus &)
```

Turns off the TT_PREFETCH_CLOSE connection option.

SetPrefetchCount()

```
void SetPrefetchCount(int numrows, TTStatus &)
```

Allows a user 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: 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 "Bulk fetch rows of TimesTen data" in the *Oracle TimesTen In-Memory Database C Developer's Guide* for more information about TT_PREFETCH_COUNT.

SetAutocommitOff()

```
void SetAutoCommitOff(TTStatus &)
```

Sets AUTOCOMMIT off for the connection.

Note that this method is automatically called by `TTConnection::Connect()` because TimesTen runs with optimal performance only with AUTOCOMMIT turned off.

SetAutoCommitOn()

```
void SetAutoCommitOn(TTStatus &)
```

Sets AUTOCOMMIT on for the connection, which means that every SQL statement now occurs in its own transaction.

Note that TimesTen generally runs much faster with AUTOCOMMIT turned off.

When AUTOCOMMIT is off, committing SELECT statements requires explicit calls to `TTCmd::Close()`.

GetTTContext()

```
void GetTTContext(char * output, TTStatus &)
```

Returns the connection's context value, which is unique to each connection to a TimesTen data store. The context of a connection can be used to correlate TimesTen connections with PIDs using the `ttStatus` TimesTen utility, for example.

The context value is returned through the `output` parameter. This method must be called with an array of CHAR[17] or larger for the *`output`* parameter.

This method calls the `ttContext` TimesTen built-in procedure. See the *Oracle TimesTen In-Memory Database C Developer's Guide* for more information about `ttContext`.

TTCmd

Encapsulates a single SQL statement that will be used multiple times in an application program. You can think of `TTCmd` as a value-added C++ wrapper around the ODBC HSTMT handle.

Public members

None.

`TTCmd` has three kinds of public methods:

- Public methods
- Public methods for obtaining properties of a TTCmd object
- Public methods for batch operations

Public methods

Method	Description
Prepare()	Associates a SQL statement with `TTCmd`.
RePrepare()	Allows a statement to be reprepared.
Execute()	Invokes a SQL statement that has been prepared for execution.

Method	Description
ExecuteImmediate()	Invoke a SQL statement that has not been previously prepared.
FetchNext()	Fetches rows from the answer set, one at a time. It returns 0 when a row was successfully fetched, and 1 when no more rows are available.
Close()	Closes the answer set when the application has finished fetching rows.
Drop()	Frees a prepared SQL statement and all resources associated with it.
setQueryTimeout()	Sets a timeout value for a query.
setQueryThreshold()	Sets a threshold for writing a warning to the support log and throwing an SNMP trap when the execution time of a SQL statement exceeds the specified value.
setMaxRows()	Sets a limit on the number of rows returned by a SELECT statement.
getMaxRows()	Returns the current limit on the number of rows returned by a SELECT statement.
getRowCount()	Returns the number of rows that were affected by the recently executed SQL operation.
setParam()	Sets the value of parameters before executing a prepared SQL statement.
setParamNull()	Sets the value of a parameter to NULL before executing a prepared SQL statement.
getParamPrecision()	Returns the precision of the specified parameter in a prepared statement.
getParamScale()	Returns the scale of the specified parameter in a prepared statement.
getParamNull()	Indicates whether the specified parameter can be NULL.
getColumn()	Returns the values associated with a particular column.
isColumnNull()	Indicates whether the specified column's value is NULL.
getColumnLength()	Returns the length of the specified column, in bytes.
getColumnNullable()	Returns the values for columns of the current row of the answer set and indicates whether a column's value is NULL.
getNextColumn()	Returns the values for columns of the next row of the answer set.
getNextColumnNullable()	Returns the values for columns of the next row of the answer set and indicates whether a column's value is NULL.
getQueryThreshold()	Retrieves the query threshold value.
printColumn()	Prints a specified column to an output stream.

Public methods for obtaining properties of a TTCmd object

Method	Description
getNParameters()	Returns the number of input parameters.
getNColumns()	Returns the number of output columns.
getParamType()	Returns the ODBC data type of the specified parameter.

Method	Description
getColumnName()	Returns the name of the specified column.
getColumnType()	Returns the ODBC data type of the specified column.
getColumnPrecision()	Returns the precision of the specified column.
getColumnScale()	Returns the scale of the specified column.

Public methods for batch operations

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

Prepare()

`void Prepare(TTConnection*, const char* sqlP, TTStatus&)`

This method associates a SQL statement with `TTCmd`. It takes three parameters:

- A pointer to a `TTConnection` object, which should be already connected to the database by a call to `TTConnection::Connect()`
- A `const char *` parameter, which is the SQL string being prepared
- A `TTStatus` object

Before `TTCmd` 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 that it will be executed in an efficient manner. The `Prepare()` method 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 and parameters at prepare time.

Note that prepare is a relatively expensive operation. When an application establishes a connection to TimesTen, using `TTConnection::Connect()`, the application should prepare all `TTCmd` objects associated with the connection. A `TTStatus` object is thrown as an exception if an error occurs.

RePrepare()
```
void RePrepare(TTConnection *cP, TTStatus & stat)
```

This method allows a statement to be reprepared. It is useful only when a statement handle in a prepared statement has been invalidated.

Execute()
```
void Execute(TTStatus&)
```

This method invokes a SQL statement that has been prepared for execution.

Use `Execute` to invoke a SQL statement previously prepared with the `Prepare()` method, after any necessary parameter values are defined using `setParam()` methods.

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 the `Close()` method does not need to be called. A `TTStatus` object is thrown as an exception if an error occurs.

ExecuteImmediate()
```
int ExecuteImmediate(TTConnection*, const char * sqlP, TTStatus& stat)
```

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 only executed a few 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.

FetchNext()
```
int FetchNext(TTStatus& stat)
```

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

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

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

After executing a SELECT using the `Execute()` method, the answer 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 answer set. A `TTStatus` object is thrown as an exception if an error occurs.

Close()

```
void Close(TTStatus&)
```

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

Failure to close the answer set may result in locks being held on rows for too long, causing concurrency problems as well as memory leaks and other errors. A `TTStatus` object is thrown as an exception if an error occurs.

Drop()

```
void Drop(TTStatus&)
```

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

It is more efficient to use multiple `TTCmd` objects to execute multiple SQL statements. Use the `Drop()` method only if it is certain that a particular SQL statement will not be used again. A `TTStatus` object is thrown as an exception if an error occurs.

setQueryTimeout()

```
void setQueryTimeout(const int nSecs, TTStatus&)
```

This method allows applications to stop long running queries as needed by setting a timeout value on the query. The *`nSecs`* value specifies the timeout in seconds.

Note that there is no default query timeout value.

setQueryThreshold()

```
void TTCmd::setQueryThreshold(const int seconds, TTStatus &stat)
```

This method sets a threshold for writing a warning to the support log and throwing an SNMP trap when the execution time of a SQL statement exceeds the specified value.

setMaxRows()

```
void setMaxRows(const int nRows, TTStatus &stat)
```

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, fetching beyond the max number of rows set will cause the statement to return SQL_NO_DATA_FOUND. The `TTCmd` object will TRUE if the `eof()` method is called. The default is to return all rows. To reset a limit to again return all rows, call `setMaxRows()` with *`nRows`* set to 0. The limit is only meaningful for SELECT statements.

getMaxRows()

```
int getMaxRows(TTStatus &stat)
```

This method returns the current limit of number of rows returned by a SELECT statement from this `TTCmd` object. A return value of 0 means all rows are returned.

getRowCount()

```
int getRowCount()
```

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

setParam()

```
void setParam(int pno, ...)
```

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

The various `setParam()` versions are used to set the value of parameters 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 legal), 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 the *Oracle TimesTen In-Memory Database SQL Reference*.

The first argument passed to `setParam()` is the position of the parameter to be set. The first (left-most) 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 TTClasses 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.

Table 4–1 shows the supported SQL data types and the appropriate versions of `setParam()` to use for each parameter type. Note that SQL data types not mentioned are not supported in this version of TTClasses.

Note: In this table, the length *`len`* is in bytes.

Table 4–1 TTCmd::setParam() variants for supported data types

Data type	setParam variants supported
TT_TINYINT	`setParam(int, unsigned char)`
TT_SMALLINT	`setParam(int, SQLSMALLINT)`
TT_INTEGER	`setParam(int, SQLINTEGER)`
TT_BIGINT	`setParam(int, SQLBIGINT)`
BINARY_FLOAT	`setParam(int, float)`
BINARY_DOUBLE	`setParam(int, double)`

Table 4–1 (Cont.) TTCmd::setParam() variants for supported data types

Data type	setParam variants supported

NUMBER
TT_DECIMAL
	`setParam(int, char*)`
	`setParam(int, const char*)`
	`setParam(int, SQLTINYINT)`
	`setParam(int, SQLSMALLINT)`
	`setParam(int, SQLINTEGER)`
	`setParam(int, SQLBIGINT)`
	Note: The integer type methods are appropriate only for columns declared with the scale parameter set to zero, such as NUMBER(8) or NUMBER(8,0).

TT_CHAR
CHAR
TT_VARCHAR
VARCHAR2
	`setParam(int, char*)`
	`setParam(int, const char*)`

TT_NCHAR NCHAR NVARCHAR2	`setParam(int, SQLWCHAR*, int` *`len`*`)`
BINARY VARBINARY	`setParam(int, const void*, int` *`len`*`)`
DATE TT_TIMESTAMP TIMESTAMP	`setParam(int, TIMESTAMP_STRUCT*)`
TT_DATE	`setParam(int, DATE_STRUCT*)`
TT_TIME	`setParam(int, TIME_STRUCT*)`

setParamNull()

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

This method can be used to indicate that the value for parameter number *pno* should be the SQL NULL value.

See "setParam()" on page 4-12.

getParamPrecision()

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

This method returns the precision of parameter number *pno*.

getParamScale()

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

This method returns the scale of parameter number *pno*.

getParamNull()

```
bool getParamNull(int pno);
```

This method indicates whether parameter number *pno* can be NULL.

getColumn()

```
void getColumn (int cno, ...)
```

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

The `getColumn()` method returns 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 is the ordinal number of the column whose value is to be fetched. The second argument passed to the `getColumn()` method is a pointer to a variable which is to receive the value of the specified column. The type of this argument varies depending on the type of the column being returned.

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

Integer type methods include one of the following functions: SQLTINYINT, SQLSMALLINT, SQLINTEGER, or SQLBIGINT. They are appropriate only for columns 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
SQLTINYINT	0<=*p*<=2
SQLSMALLINT	0<=*p*<=4
SQLINTEGER	0<=*p*<=9
SQLBIGINT	0<=*p*<=18

To ensure that all values in the column will 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 <= *p* <= 18. For example:

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

Table 4–2 shows the supported SQL data types and the appropriate versions of getColumn() and getColumnNullable() to use for each parameter type.

Note: In this table, length *lenP* is in bytes.

Table 4–2 TTCmd::getColumn()/getColumnNullable() variants for supported data types

Data type	getColumn variants supported

TT_TINYINT	`getColumn(int cno, unsigned char*)`
	`getColumnNullable(int cno, SQLTINYINT*)`

TT_SMALLINT	`getColumn(int cno, SQLSMALLINT*)`
	`getColumnNullable(int cno, SQLSMALLINT*)`

Table 4–2 (Cont.) TTCmd::getColumn()/getColumnNullable() variants for supported data

Data type	getColumn variants supported

TT_INTEGER	`getColumn(int cno, SQLINTEGER*)`
	`getColumnNullable(int cno, SQLINTEGER*)`

TT_BIGINT	`getColumn(int cno, SQLBIGINT*)`
	`getColumnNullable(int cno, SQLBIGINT*)`

BINARY_FLOAT	`getColumn(int cno, float*)`
	`getColumnNullable(int cno, float*)`

BINARY_DOUBLE	`getColumn(int cno, double*)`
	`getColumnNullable(int cno, double*)`

NUMBER
TT_DECIMAL
	`getColumn(int cno, char**)`
	`getColumn(int cno, SQLTINYINT*)`
	`getColumn(int cno, SQLSMALLINT*)`
	`getColumn(int cno, SQLINTEGER*)`
	`getColumn(int cno, SQLBIGINT*)`
	`getColumnNullable(int cno, char**)`
	`getColumnNullable(int cno, SQLTINYINT*)`
	`getColumnNullable(int cno, SQLSMALLINT*)`
	`getColumnNullable(int cno, SQLINTEGER*)`
	`getColumnNullable(int cno, SQLBIGINT*)`
	Note: The integer type methods are appropriate only for columns declared with the scale parameter set to zero.

TT_CHAR
CHAR
TT_VARCHAR
VARCHAR2
	`getColumn(int cno, char**)`
	`getColumnNullable(int cno, char**)`

TT_NCHAR
NCHAR
TT_NVARCHAR
NVARCHAR2
	`getColumn(int cno, SQLWCHAR*)`
	`getColumnNullable(int cno, SQLWCHAR*)`

BINARY
VARBINARY
	`void getColumn(int cno, void** binPP, int* lenP)`
	`void getColumnNullable(int cno, void** binPP, int* lenP)`

DATE
TT_TIMESTAMP
TIMESTAMP
	`getColumn(int cno, TIMESTAMP_STRUCT*)`
	`getColumnNullable(int cno, TIMESTAMP_STRUCT*)`

TT_DATE	`getColumn(int cno, DATE_STRUCT*)`
	`getColumnNullable(int cno, DATE_STRUCT*)`

TT_TIME	`getColumn(int cno, TIME_STRUCT*)`
	`getColumnNullable(int cno, TIME_STRUCT*)`

Other SQL data types are not supported in this release of the TTClasses library.

isColumnNull()

`bool isColumnNull(int cno)`

This method provides another way to determine whether the value of column number *cno* is NULL.

See information about the `getColumnNullable()` method, following shortly.

getColumnLength()

`int getColumnLength(int cno)`

This method returns the length, in bytes, of column number *cno*, not counting the NULL terminator; or, it returns SQL_NULL_DATA if the value is NULL. This method is generally useful only when accessing columns of type VARBINARY or NVARCHAR2. The value returned is between 0 and the column precision, inclusive. See "getColumnPrecision()" on page 4-18.

getColumnNullable()

`bool getColumnNullable(int cno, ...)`

The `getColumnNullable()` method is similar to the `getColumn()` method, described previously. However, in addition to the behavior of `getColumn()`, the `getColumnNullable()` method also returns an indication of 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 in the variable pointed to by the second parameter and the `getColumnNullable()` method returns FALSE.

See "getColumn()" on page 4-14.

getNextColumn()

`void getNextColumn(int iP, ...)`

The `getNextColumn()` method, as well as the `getNextColumnNullable()` method (described shortly), can be used to fetch the values for columns of the next row of the result set. Before the `getNextColumn()` and `getNextColumnNullable()` methods can be used, the `FetchNext()` method must be used to fetch the first row from the result set of a SELECT statement. The *iP* parameter represents a pointer to an internal column number. When you use `getNextColumn()`, the columns are fetched in order. You cannot change the fetch order.

See Table 4–3 on page 4-19 for the supported SQL data types and the appropriate versions of `getColumn()` (described previously) to use for each parameter type. This information can be used for `getNextColumn()`.

getNextColumnNullable()

`bool getColumnNullable(int iP, ...)`

The `getNextColumnNullable()` method is similar to the `getNextColumn()` method (described previously). However, in addition to the behavior of `getColumn()`, the `getNextColumnNullable()` method returns an indication of 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 in the variable pointed to by the second parameter and the `getColumnNullable()` method returns FALSE.

The *iP* parameter represents a pointer to an internal column number. When you use `getNextColumn()`, the columns are fetched in order. You cannot change the fetch order.

See Table 4–3 on page 4-19 for the supported SQL data types and the appropriate versions of getColumn() to use for each parameter type. This information can be used for `getNextColumnNullable()`.

getQueryThreshold()

```
int TTCmd::getQueryThreshold(TTStatus &stat)
```

Retrieves the query threshold value. See "setQueryThreshold()" on page 4-11.

printColumn()

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

This method prints column number *cno* to an output stream. Use this method for debugging or for demo programs.

Usage

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

Only database operations that need to be executed only *once* 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).

Methods for obtaining properties of a prepared TTCmd object

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.

getNParameters()

```
int getNParameters()
```

This method returns the number of input parameters.

getNColumns()

```
int getNColumns()
```

This method returns the number of output columns.

getParamType()

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

This method 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`.

getColumnName()

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

This method returns the name of column number *cno*.

getColumnType()

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

This method returns the data type of column number *cno*. The value returned is the parameter's 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`.

getColumnPrecision() `int getColumnPrecision(int cno)`

This method returns the precision of column number *cno*. This value is generally interesting only when generating output from table columns of type CHAR, VARCHAR2, BINARY, VARBINARY, NCHAR and NVARCHAR2.

getColumnScale()

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

This method returns the scale of column number *cno*.

Batch operations

TimesTen supports the ODBC function `SQLBindParams` for batch insert, update and delete operations. TTClasses provides an interface to `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()`. Note the similarity to normal TTClasses (non-batch) operations, where a statement is compiled using `Prepare`, which also performs the binding of all parameters automatically, and then executed using `Execute()`.

See the TTClasses `bulktest` sample program for an example of using a batch operation.

This section describes the `TTCmd` methods that expose the batch INSERT/UPDATE/DELETE functionality to TTClasses users.

PrepareBatch()

```
void PrepareBatch(TTConnection*, const char * sqlP,
     TTCmd::TTCMD_USER_BIND_LEVEL level,
     unsigned short batchSize, TTStatus&)
```

`PrepareBatch()` is the analog of the `Prepare()` method for batch INSERT, UPDATE, or DELETE statements. The `TTConnection*` and *`sqlP`* and `TTStatus&` parameters are used the in the same as for the `Prepare()` method.

There is only one valid value for the *`level`* parameter, as follows:

```
TTCmd::TTCMD_USER_BIND_PARAMS
```

The `batchSize` parameter specifies the maximum number of insert/update/delete operations that will be performed using subsequent calls to `ExecuteBatch()`.

BindParameter()

```
void BindParameter(int pno, unsigned short batchSize,
      TYPE*, [SQLLEN*], TTStatus&)
```

The overloaded `BindParameter()` method is used to bind an array of values (one for each parameter) for a statement compiled using `PrepareBatch()`. The `batchSize` value of this call must match the `batchSize` value specified in `PrepareBatch()`. Similarly, the bound arrays should contain at least as many values as the bound arrays in `PrepareBatch()`. You must determine the correct data type for each parameter. Note that if an inappropriate type is specified, a runtime error will be written to the TTClasses global logging facility at the `TTLog::TTLOG_ERR` logging level.

Before each invocation of `ExecuteBatch()`, the application should fill these arrays with valid parameter values.

For the SQL types SQL_BINARY, SQL_VARBINARY, SQL_WCHAR, and SQL_ WVARCHAR), an additional SQLLEN* parameter, for an array of parameter lengths, in bytes, is required. This additional array must be at least `batchSize` in length and filled with valid length values before `ExecuteBatch()` is called.

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

Table 4–3 TTCmd::BindParameter() variants for supported data types

SQL data type	BindParameter variants supported
TT_TINYINT	`BindParameter(... SQLTINYINT*...)`
TT_SMALLINT	`BindParameter(...SQLSMALLINT*...)`
TT_INTEGER	`BindParameter(...SQLINTEGER*...)`
TT_BIGINT	`BindParameter(...SQLBIGINT*...)`
BINARY_FLOAT	`BindParameter(...float*...)`
BINARY_DOUBLE	`BindParameter(...double*...)`
NUMBER TT_DECIMAL	`BindParameter(...char**...)`
TT_CHAR CHAR TT_VARCHAR VARCHAR2	`BindParameter(...char**...)`
TT_NCHAR NCHAR TT_NVARCHAR NVARCHAR2	`BindParameter(...SQLWCHAR**, SQLLEN*...)`
BINARY VARBINARY	`BindParameter(...const void**, SQLLEN*...)`

Table 4–3 (Cont.) TTCmd::BindParameter() variants for supported data types

SQL data type	BindParameter variants supported
DATE TT_TIMESTAMP TIMESTAMP	`BindParameter(... TIMESTAMP_STRUCT*...)`
TT_DATE	`BindParameter(...DATE_STRUCT*...)`
TT_TIME	`BindParameter (...TIME_STRUCT*...)`

setParamLength()

`setParamLength(int` *`pno`*`, unsigned short` *`rowno`*`, int` *`len`*`)`

This method sets the length of one of the bound parameter values before a call to `ExecuteBatch()`. The *`pno`* parameter specifies which parameter in the statement will be set. The *`rowno`* parameter specifies for which row the length will be set. The *`len`* parameter specifies the length being set, in bytes, not counting the NULL terminator; or, it can be set to SQL_NTS for a null-terminated string. (It could also be set to SQL_NULL_DATA, but TimesTen provides the `setParamNull()` method for this case.)

For types apart from SQL_BINARY, SQL_VARBINARY, SQL_WCHAR, and SQL_WVARCHAR, this is the only method available to explicitly set the length of a parameter before an `ExecuteBatch()` call. For these types, the length can also be explicitly set through manipulation of the `SQLLEN*` array, which is the fourth parameter of the `BindParameter()` call.

setParamNull()

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

This method sets one of the bound parameter values to NULL before a call to `ExecuteBatch()`. The first parameter, *`pno`*, specifies which parameter in the statement will be set. The second parameter, *`rowno`*, specifies for which row the length will be set.

For types apart from SQL_BINARY, SQL_VARBINARY, SQL_WCHAR, and SQL_WVARCHAR, this is the only method available to explicitly set the NULL value of a parameter before an `ExecuteBatch()` call. For these types, nullability can also be explicitly set through manipulation of the `SQLLEN*` array, which is the fourth parameter to the `BindParameter()` call.

ExecuteBatch()

`void ExecuteBatch(unsigned short` *`numRows`*`, TTStatus&)`

This method returns the number of rows in a batch that have been updated. The number represents *`*pirow`* from the ODBC `SQLSetParams` call.

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 *`batchSize`* specified in the `PrepareBatch()` and `BindParameter()` calls. The *`numRows`* parameter can be less than the *`batchSize`* value, as required by the application logic.

Before calling `ExecuteBatch()`, the application should fill the arrays of parameters bound using `BindParameter()` with valid values. Null values can be specified as necessary using by using `setParamNull()`.

Example 4–4 also shows how to use the `ExecuteBatch()` method. (The `bulktest` demo also shows usage of this method.)

Example 4–4 Using the ExecuteBatch() method

Create a table with two columns:

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

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;
TTStatus stat;

// (assume a connection has already been established)

try {

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

// Commit the prepared statement

connection.Commit(stat);

// Bind the arrays of parameters

insert.BindParameter(1, BATCH_SIZE, int_array, stat);
insert.BindParameter(2, BATCH_SIZE, char_array, stat);

// Execute 5 batches, inserting 5 * BATCH_SIZE rows into
// the database

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);
      }

       // Execute the batch insert statement,
       // which inserts the entire contents of the
       // integer and char arrays in one operation.

       int num_ins = insert.ExecuteBatch(BATCH_SIZE, stat);

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

       connection.Commit(stat);

   } // for iter
```

The number of rows (*num_ins* in the example) can be less than `BATCH_SIZE` if, for example, there is a uniqueness constraint on one of the columns. You can use code similar to that in Example 4–5 to check for this situation and roll back the transaction as necessary.

Example 4–5 Using ExecuteBatch() and checking against BATCH_SIZE

```
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

       int num_ins = insert.ExecuteBatch(BATCH_SIZE, stat);

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

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

   } // for iter
```

The integer output of `TTCmd::ExecuteBatch()` is *`*pirow`*, equivalent to the third parameter of an ODBC `SQLParamOptions` call. See the *ODBC Programmer's Manual* for information about `SQLParamOptions`.

TTConnectionPool

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

In general, multithreaded applications can be written using one of two basic 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 which are used for the duration of the application. When a thread needs to perform a database transaction, it checks out an idle connection from the pool, performs its transaction, and then returns the connection to the pool. This is the scenario that the `TTConnectionPool` class assists with.

Note: For best overall performance, TimesTen recommends having one or two concurrent direct-memory database connections for each CPU of the database server. For no reason should your number of concurrent direct-memory database connections (the size of your connection pool) be more than twice as many CPUs on the database server. In client/server mode, 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 a number of `TTConnection` objects, but does not call their `Connect()` methods, which would actually connect them to TimesTen. The application uses the `TTConnectionPool::AddConnectionToPool()` method to place the connection objects into the pool. Then, the application calls `TTConnectionPool::ConnectAll()` to connect all the connections to TimesTen. Threads wanting to use TimesTen use the `getConnection()` method and `freeConnection()` methods to get and return idle connections.

Public members

None.

Public methods

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

AddConnectionToPool()

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

This method is used to add a `TTConnection` object, or an object of a class derived from `TTConnection`, to the connection pool.

ConnectAll()

```
void ConnectAll(const char* connStr, TTStatus&)
```

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

getConnection()

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

Checks out an idle connection from the connection pool for 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()` will block 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`* milliseconds. If no connection is available in that time then `getConnection()` returns NULL to the caller.

freeConnection()

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

Returns a connection to the pool for reassignment to another thread. Applications should not free connections that are in the midst of a transaction. `TTConnection::Commit()` or `TTConnection::Rollback()` should be called immediately prior to calling `freeConnection()`.

DisconnectAll()

```
void DisconnectAll(TTStatus&)
```

Disconnects all connections in the connection pool from TimesTen.

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

getStats()

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

Queries the `TTConnectionPool` for status information. Data returned is:

- *`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-water mark for the most number of connections in use at one time.
- *`nForcedCommits`*: Number of times that `freeConnection()` had to call `Commit()` on a connection before checking it into the pool. If this counter is non-zero then the user application is not calling `TTConnection::Commit()` or `Rollback()` before returning a connection to the pool.

TTGlobal

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

Public members

None.

Public methods

Method	Description
setLogStream()	Specifies where TTClasses logging information should be sent.
setLogLevel()	Specifies the verbosity level of TTClasses logging.
disableLogging()	Disables TTClasses logging.

setLogStream()

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

Specifies where TTClasses logging information should be sent. By default, if TTClasses logging is enabled, logging is to `stderr`. Using this method, a user 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);
```

setLogLevel()

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

This method specifies the verbosity level of TTClasses logging. Table 4–4 describes TTClasses logging levels. The levels are cumulative.

Table 4–4 TTClasses logging levels

Logging level	Description
`TTLog::TTLOG_NIL`	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 `TTCmd::Prepare()`, 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 `TTCmd` and `TTConnection` objects, including the SQL string where appropriate.

Table 4–4 (Cont.) TTClasses logging levels

Logging level	Description
`TTLog::TTLOG_DEBUG`	Also logs debugging information, such as all bound parameter values for each call to `TTCmd::Execute()`.

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

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

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);
```

Using TTGlobal

The `TTGlobal` 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 instead of during production.

When logging from a multithreaded program, you may encounter a problem where log output from different program threads is intermingled when written to disk. 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 "TTClasses logging" on page 3-3 for more information about using `TTGlobal`.

System catalog classes

`TTCatalog` is included in the TimesTen C++ Interface Classes to facilitate reading metadata from the database system catalog.

The `TTCatalog` class is different from using the other classes in the TimesTen C++ Interface Classes. After connecting to the database and reading its system catalogs, the `TTCatalog` constructor disconnects from the database, and no further direct database interaction is done. The resulting object contains data structures that contains all of the information that was read from the database catalog, and which is easily accessible to a user program.

Each `TTCatalog` internally contains an array of `TTCatalogTable` objects. Each `TTCatalogTable` contains an array of `TTCatalogColumn` objects and an array of `TTCatalogIndex` objects. When accessing by index, access to these arrays is zero-based.

The following ODBC functions are used inside `TTCatalog`:

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

This section includes the following classes:

- TTCatalog
- TTCatalogTable
- TTCatalogColumn
- TTCatalogIndex

TTCatalog

The `TTCatalog` class is the top-level class used for programmatically accessing metadata information about tables in a database. A `TTCatalog` object has an internal array of `TTCatalogTable` objects inside it. Apart from the constructor, all public methods of `TTCatalog` are used to gain read-only access to that `TTCatalogTable` array.

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

```
TTCatalog (TTConnection*)
```

Then to use the `TTCatalog` object, call its `fetchCatalogData()` method, described shortly.

Public members

None.

Public methods

Method	Description
fetchCatalogData()	Reads the catalogs in the data store for information about tables and indexes as it constructs itself and stores this information into its internal data structures
getNumTables()	Returns the total number of tables in the database, both user and system tables.
getNumUserTables()	Returns the number of user tables in the database.
getNumSysTables()	Returns the number of system tables in the database.
getTable()	Returns a constant reference to the `TTCatalogTable` object corresponding to the specified database table.
getTable()	Returns a constant reference to the `TTCatalogTable` corresponding to the *n*th table in the system (where *n* is specified).
getUserTable()	Returns a constant reference to the `TTCatalogTable` corresponding to the *n*th user table in the system (where *n* is specified).
getTableIndex()	Returns the index in the `TTCatalog` object for the specified table.

fetchCatalogData()

```
fetchCatalogData(TTStatus &)
```

This method is the only one that interacts with the data store. The connection to the data store was cached by the constructor, so the only parameter is a `TTStatus` object. This method reads the catalogs in the database for information about tables, and indexes as it constructs itself, storing the information into its internal data structures.

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

You must call this method before you use any other `TTCatalog` accessor method.

This example demonstrates the use of `TTCatalog`. It does not check `stat.rc` after the two database calls.

Example 4–6   Fetching catalog data

```
TTConnection conn;
TTStatus stat;
conn.Connect(DSN=TptbmData37, stat);
TTCatalog cat (&conn);
cat.fetchCatalogData(stat);
// 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;
```

getNumTables()

```
int getNumTables()
```

Returns the total number of tables in the database, both user and system tables.

getNumUserTables()

```
int getNumUserTables()
```

Returns the number of user tables in the database.

getNumSysTables()

```
int getNumSysTables()
```

Returns the number of system tables in the database.

There is no corresponding `getSysTable()` method.

getTable()

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

Returns a constant reference to the `TTCatalogTable` object corresponding to the database table named *`tblname`* owned by *`owner`*. See "TTCatalogTable" on page 4-29.

getTable()

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

Returns a constant reference to the `TTCatalogTable` corresponding to table number *`tno`* in the system. This method is intended to facilitate iteration through all of the tables in the system; the order of the tables in this array is arbitrary.

Note that the following relationship is asserted to hold:

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

getUserTable()

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

Returns a constant reference to the `TTCatalogTable` 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.

Note that the following relationship is asserted to hold:

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

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 the *`owner.tblname`* does not exist. It returns -1 if the catalog did not call `fetchCatalogData()` before calling `TTCatalog::getTableIndex()`.

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

Example 4–7 Retrieving table information from a catalog

```
TTConnection conn;
TTStatus stat;
conn.Connect(...);
TTCatalog cat (&conn);
cat.fetchCatalogData(stat);

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

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

TTCatalogTable

Used to store all metadata information about a table's columns and indexes.

Public members

None.

Public methods

Method	Description
getTableOwner()	Returns the owner of the table.
getTableName()	Returns the name of the table.
getNumColumns()	Returns the number of columns in the table.

Method	Description
getNumIndexes()	Returns the number of indexes on the table.
getColumn()	Returns a constant reference to the `TTCatalogColumn` corresponding to the ith column in the table.
getIndex()	Returns a constant reference to the `TTCatalogIndex` corresponding to the ith index in the table.
isSystemTable()	Returns TRUE if the table is a system table.
isUserTable()	Returns TRUE if the table is a user table.

getTableOwner()

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

Returns the owner of the table.

getTableName()

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

Returns the name of the table.

getNumColumns()

```
int getNumColumns()
```

Returns the number of columns in the table.

getNumIndexes()

```
int getNumIndexes()
```

Returns the number of indexes on the table.

getColumn()

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

Returns a constant reference to the `TTCatalogColumn` corresponding to column number *`i`* in the table. This method is intended to facilitate iteration through all of the user tables in the system.

Note that the following relationship is asserted to hold:

0 <= *`i`* <= `getNumColumns()`

getIndex()

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

Returns a constant reference to the `TTCatalogIndex` corresponding to index number *`i`* in the table. This method is intended to facilitate iteration through all of the user tables in the system; the order of a table's indexes in this array is arbitrary.

Note that the following relationship is asserted to hold:

0 <= *`i`* <= `getNumColumns()`

isSystemTable()

```
bool isSystemTable()
```

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

isUserTable()

```
bool isUserTable()
```

Returns TRUE if this is a user table. It returns FALSE otherwise. Note that the definition of a user table is one that is not a system table. Thus `isUserTable()` returns the opposite of `isSystemTable()` for any table.

The `isSystemTable()` and `isUserTable()` methods 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. See the TTClasses demo program `catalog` for an example of how this can be done.

TTCatalogColumn

The `TTCatalogColumn` class is used to store all metadata information about a single table column of the `TTCatalogTable` object it is associated with.

Public members

None.

Public methods

Method	Description
getColumnName()	Return the name of the column.
getDataType()	Returns an integer representing the ODBC SQL data type of the column.
getTypeName()	Returns the database-dependent name that corresponds to the type returned by `getDataType()`.
getNullable()	Returns SQL_NO_NULLS, SQL_NULLABLE, or SQL_NULLABLE_UNKNOWN.
getPrecision()	Returns the precision of the column.
getLength()	Returns the length of the column, in bytes.
getScale()	Returns the scale of the column.
getRadix()	Returns the radix of the column.

getColumnName()

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

Return 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.

getTypeName()

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

Returns the database-dependent name that corresponds to the type returned by `getdataType()`.

getNullable()

```
int getNullable()
```

Returns SQL_NO_NULLS, SQL_NULLABLE, or SQL_NULLABLE_UNKNOWN.

getPrecision()

```
int getPrecision()
```

Returns the precision of the column.

getLength()

```
int getLength()
```

Returns the length of the column, in bytes.

getScale()

```
int getScale()
```

Returns the scale of the column.

getRadix()

```
int getRadix()
```

Returns the radix of the column.

TTCatalogIndex

Used to store all information about an index of the `TTCatalogTable` it is associated with.

Public members

None.

Public methods

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

getIndexName()

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

Returns the name of the index.

getIndexOwner()

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

Returns the owner of the index.

getTableName()

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

Returns the name of the table for which the index was created.

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 TTREE_INDEX. For other databases, allowable values are SQL_INDEX_HASHED and SQL_INDEX_ CLUSTERED.

isUnique()

```
bool isUnique()
```

Returns whether the index is a unique index. TRUE means it is unique; FALSE means it is not unique.

getNumColumns()

```
int getNumColumns()
```

Returns the number of columns in the index.

getColumnName()

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

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

getCollation()

```
char getCollation (int i)
```

Returns the collation of column number *i* in the index. Values returned are 'A' for ascending and 'D' for descending index order.

XLA classes

TTClasses provides a set of C++ classes that make it easy to write applications that use the TimesTen Transaction Log API (XLA).

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

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

Table 4–5 lists the TTClasses XLA classes and their descriptions.

Table 4–5 TTClasses XLA classes

Class	Description
TTXlaPersistConnection	Defines a persistent connection to a TimesTen data store.
TTXlaRowViewer	Fetches column values from a particular update record.
TTXlaTableHandler	Provides methods that enable and disable change tracking for a table. Methods are also provided to handle update notification records from XLA.
TTXlaTableList	Provides a list of TTXlaTableHandler objects. This class is used to route a particular change to the appropriate method for processing. Incoming update notification records are routed to the appropriate method of the appropriate TTXlaTableHandler object for processing.

TTXlaPersistConnection

`TTXlaPersistConnection` defines a persistent connection to a TimesTen data store.

Public members

None

Public methods

Method	Description
Connect()	Connects with the specified bookmark.
Connect()	Connects with the specified bookmark. It creates a bookmark if one does not exist.
DeleteBookmarkAndDisconnect()	Deletes the bookmark and disconnects from the data store.
Disconnect()	Closes an XLA connection to a TimesTen data store.
ackUpdates()	Advances the bookmark to the next set of updates.
getBookmarkIndex()	Stores the current place in the transaction log.
setBookmarkIndex()	Returns to the saved transaction log index.
fetchUpdatesWait()	Fetches updates to the transaction log within the specified wait period.

Connect()

```
virtual void Connect(const char* connStr, const char * bookmark,
                     bool createBookmark, TTStatus&);
```

Each persistent XLA connection has a name (or *`bookmark`*) associated with it, so that upon disconnect and reconnect, the same place in the transaction log can be found. The name for a connection's bookmark is specified in the *`bookmark`* parameter.

> **Note:** Only one XLA connection can connect with a given bookmark name. An error will be returned if multiple connections try to connect to the same bookmark.

Whether this is a new bookmark, or a previously created bookmark, is specified by the *`createBookmark`* boolean parameter. If you specify that a bookmark is new (`createBookmark==true`) and it already exists, an error will be returned. Similarly, if you specify that a bookmark already exists (`createBookmark==false`) and it does not already exist, an error will be returned.

Connect()

```
virtual void Connect(const char* connStr, const char * bookmark, TTStatus&);
```

This second connect method first tries to connect using the supplied bookmark, reusing it (implicit value of `createBookmark==false`). If that bookmark does not exist, the method then tries to connect and create a new bookmark with the name *`bookmark`* (implicit value of `createBookmark==true`).

This method is provided as a convenience, to simplify XLA connection logic in case the developer does not wish to worry about whether the XLA bookmark exists.

DeleteBookmarkAndDisconnect()

```
void DeleteBookmarkAndDisconnect(TTStatus&)
```

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

Disconnect()

```
virtual void Disconnect(TTStatus&)
```

This method closes an XLA connection to a TimesTen data store. The XLA bookmark persists after you call this method. If you want to delete the bookmark and disconnect from the data store, then use `DeleteBookmarkAndDisconnect()` instead (described immediately above).

ackUpdates()

```
void ackUpdates(TTStatus &)
```

This method is used to advance the bookmark to the next set of updates. After you have acknowledged a set of updates, the updates cannot be viewed again. See the descriptions of `getBookMarkIndex()` and `setBookMarkIndex()`, following shortly, 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 transaction log files do not fill up the disk where they are stored. 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, the records can be retrieved when the application reconnects using XLA.

getBookmarkIndex()

```
void getBookmarkIndex(TTStatus &)
```

This method acquires the current bookmark location.

setBookmarkIndex()

```
void setBookmarkIndex(TTStatus &)
```

This method restores the bookmark to the previously acquired bookmark location. Use this method to replay a batch of records multiple times.

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.

fetchUpdatesWait()

```
void fetchUpdatesWait(ttXlaUpdateDesc_t*** arry, int maxrecs,
                      int* recsP, int seconds, TTStatus&)
```

This method is used by an XLA application to fetch a set of records describing changes to a data store. 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.

The caller specifies the maximum number of records it is willing to receive. When the method returns, the caller receives the number of records actually returned, as well as an array of pointers which 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.

Usage

A persistent XLA application can create multiple TTXlaPersistConnection objects. Each `TTXlaPersistConnection` object must be associated with its own bookmark, which is specified at connect time and must be maintained through the `ackUpdates()` and `deleteBookmark()` methods.

After a persistent XLA connection is made, 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 prevent the transaction log from overfilling disk. After processing a batch of updates, the application should call `ackUpdates()` in order 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 persistent XLA application disconnects after `fetchUpdatesWait()`, but before `ackUpdates()`, the next connection (with the same bookmark name) which calls `fetchUpdatesWait()` will see that same batch of updates.

Updates that occur while a `TTXlaPersistConnection` object is disconnected to the data store are not lost, but are stored in the transaction log until another `TTXlaPersistConnection` object connects with the same bookmark name.

TTXlaRowViewer

The `TTXlaRowViewer` class allows application developers to examine XLA change notification record structures to fetch old and new column values.

Before a row can be examined, the `TTXlaRowViewer` object must be associated with a table (using the `setTable()` method) and a row (using the `setTuple()` method). The table is a `TTXlaTable` object previously defined. The row is part of a `ttXlaUpdateDesc_t` structure as returned by XLA using the TTXlaPersistConnection`::fetchUpdateWait()` method.

Public members

None

Public methods

Method	Description
setTable()	Associates `TTXlaRowViewer` with the specified table.
setTuple()	Associates the `TTXlaRowViewer` object with the specified row image.
isNull()	Indicates whether the specified column in a row image is NULL.
"Get()"	Fetches the value of the specified column in a row image.

setTable()

```
void setTable(TTXlaTable*)
```

This associates this `TTXlaRowViewer` with a particular table.

setTuple()

```
void setTuple(ttXlaUpdateDesc_t*, int whichTuple)
```

This method associates this `TTXlaRowViewer` object with a particular row image.

The `ttXlaUpdateDesc_t` structures that are returned by TTXlaPersistConnection`::fetchUpdatesWait()` contain either zero, one, or two rows.

- 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 data store 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 of row images in the update structure should be examined. Values for this parameter are:
  - INSERTED_TUP: Examine the newly 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

After the `setTable()` and `setTuple()` methods are called, the following methods can be used to fetch information about row images in the update records.

isNull()

```
bool isNull(int whichCol)
```

Indicates whether a particular column in a row image is NULL (returns TRUE) or not (returns FALSE).

The *`whichCol`* parameter is the column number for the column to be interrogated.

Get()

```
void Get(int col, ...)
```

Fetches the value of a particular column in a row image.

These methods are very similar to the TTCmd`::getColumn()` methods.

The *`col`* parameter is the column number for the column to be interrogated.

Table 4–6 shows the supported SQL data types and the appropriate versions of `Get` to use for each parameter type. There are six variants for the NUMBER data type and two variants for the FLOAT data type. Design the application according to the kind of data that is stored. For example, data of type NUMBER(9,0) can be accessed by the `Get(int, int*)` method without loss of data.

Note: In this table, length *`len`* is in bytes.

Table 4–6 TTXlaRowViewer::Get() variants for supported data types

XLA data type	Database data type	Get variant
TTXLA_CHAR_TT	TT_CHAR	`Get(int, char**)`
TTXLA_NCHAR_TT	TT_NCHAR	`Get(int, SQLWCHAR**, int* len)`
TTXLA_VARCHAR_TT	TT_VARCHAR	`Get(int, char**)`
TTXLA_NVARCHAR_TT	TT_NVARCHAR	`Get(int, SQLWCHAR**, int* len)`
TTXLA_TINYINT	TT_TINYINT	`Get(int, unsigned char*)`
TTXLA_SMALLINT	TT_SMALLINT	`Get(int, short*)`
TTXLA_INTEGER	TT_INTEGER	`Get(int, int*)`
TTXLA_BIGINT	TT_BIGINT	`Get(int, SQLBIGINT*)`
TTXLA_BINARY_FLOAT	BINARY_FLOAT	`Get(int, float*)`
TTXLA_BINARY_DOUBLE	BINARY_DOUBLE	`Get(int, double*)`
TTXLA_DECIMAL_TT	TT_DECIMAL	`Get(int, char**)`
TTXLA_TIME	TT_TIME	`Get(int, TIME_STRUCT*)`
TTXLA_DATE_TT	TT_DATE	`Get(int, DATE_STRUCT*)`
TTXLA_TIMESTAMP_TT	TT_TIMESTAMP	`Get(int, TIMESTAMP_STRUCT*)`
TTXLA_BINARY	BINARY	`Get(int, const void**, int* len)`
TTXLA_VARBINARY	VARBINARY	`Get(int, const void**, int* len)`

TTXLA_NUMBER	NUMBER	`Get(int, double*)`
		`Get(int, char**)`
		`Get(int, char*)`
		`Get(int, short*)`
		`Get(int, int*)`
		`Get(int, SQLBIGINT*)`

Table 4–6 (Cont.) TTXlaRowViewer::Get() variants for supported data types

XLA data type	Database data type	Get variant
TTXLA_DATE	DATE	`Get(int, TIMESTAMP_STRUCT*)`
TTXLA_TIMESTAMP	TIMESTAMP	`Get(int, TIMESTAMP_STRUCT*)`
TTXLA_CHAR	CHAR	`Get(int, char**)`
TTXLA_NCHAR	NCHAR	`Get(int, SQLWCHAR**, int*` *`len`*`)`
TTXLA_VARCHAR	VARCHAR2	`Get(int, char**)`
TTXLA_NVARCHAR	NVARCHAR2	`Get(int, SQLWCHAR**, int*` *`len`*`)`

TTXLA_FLOAT	FLOAT	`Get(int, double*)`
		`Get(int, char**)`

Usage

It is used to fetch column values from row images contained in change notification records.

TTXlaTableHandler

The `TTXlaTableHandler` class 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 with a particular table and initializes the `TTXlaTable` object contained within this object:

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

(See "Internal classes" on page 4-42 regarding `TTXlaTable`.)

Public members

None

Protected Members

Member	Description
`TTXlaTable tbl;`	The object associated with the table being handled.
`TTXlaRowViewer row;`	Used to view the row being inserted or deleted, or the old image of the row being changed.
`TTXlaRowViewer row2;`	Used to view the new image of the row being updated.

Public methods

Method	Description
EnableTracking()	Enables XLA update tracking for the underlying table.
DisableTracking()	Disables XLA update tracking for the underlying table.
HandleChange()	Dispatches a record from `ttXlaUpdateDesc_t` to the appropriate handling routine for processing.

Method	Description
HandleDelete()	Invoked when the `HandleChange()` method is called to process a delete operation.
HandleInsert()	Invoked when the `HandleChange()` method is called to process an insert operation.
HandleUpdate()	Invoked when the `HandleChange()` method is called to process an update operation.
generateSQL()	Returns the SQL associated with a given XLA record.

EnableTracking()

```
virtual void EnableTracking(TTStatus&);
```

Enables XLA update tracking for the underlying table. Until this method is called, XLA will not return information about changes to this table.

DisableTracking()

```
virtual void DisableTracking(TTStatus&);
```

Disables XLA update tracking for the underlying table. After this method is called, XLA will not return information about changes to this table.

HandleChange()

```
virtual void HandleChange(ttXlaUpdateDesc_t*, void* pData = 0);
```

Dispatches a `ttXlaUpdateDesc_t` to the appropriate handling routine for processing. The update description is analyzed to determine if it is a delete, insert or update. The appropriate virtual method—`HandleDelete()`, `HandleInsert()`, or `HandleUpdate()`—is then called.

See "Acknowledging XLA updates at transaction boundaries" on page 3-4 for an example that uses the *`pData`* parameter.

HandleDelete()

```
virtual void HandleDelete(ttXlaUpdateDesc_t*) = 0;
```

This method will be invoked whenever the `HandleChange()` method is called to process a delete operation.

This method is not implemented in the `TTXlaTableHandler` base class, but must be provided by any classes derived from it. Application developers should put their logic to handle deleted rows in this method.

The row that was deleted from the table is available through the `RowViewer` named `row`.

HandleInsert()

```
virtual void HandleInsert(ttXlaUpdateDesc_t*) = 0;
```

This method will be invoked whenever the `HandleChange()` method is called to process a insert operation.

This method is not implemented in the `TTXlaTableHandler` base class, but must be provided by any classes derived from it. Application developers should put their logic to handle inserted rows in this method.

The row that was inserted from the table is available through the `RowViewer` named `row`.

HandleUpdate()

```
virtual void HandleUpdate(ttXlaUpdateDesc_t*) = 0;
```

This method will be invoked whenever the `HandleChange()` method is called to process an update operation.

This method is not implemented in the `TTXlaTableHandler` base class, but must be provided by any classes derived from it. Application developers should put their logic to handle updated rows in this method.

The previous version of the row that was updated from the table is available through the `RowViewer` object named `row`; the new version of the row is available through the `RowViewer` object named `row2`.

generateSQL()

```
void generateSQL (ttXlaUpdateDesc_t*, char * buffer,
                  SQLINTEGER maxLen, SQLINTEGER *actualLen, TTStatus &);
```

This method can be used to print out the SQL associated with a given XLA record. The SQL string is returned through the *`buffer`* parameter, which the caller of this method has allocated space for and specified its length in the *`maxLen`* parameter. The *`actualLen`* parameter returns information about the actual length of the SQL string returned. Lengths are in bytes.

If *`maxLen`* is less than the generated SQL string, a `TTStatus` error will be returned, and the contents of buffer and *`actualLen`* will be unmodified.

Usage

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

One strategy is to derive multiple classes from `TTXlaTableHandler`, one for each table. Business logic to handle changes to customers might be implemented in a `CustomerTableHandler` class, while business logic to handle changes to orders might be implemented in a `OrderTableHandler` class.

Another strategy 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.

TTXlaTableList

The `TTXlaTableList` class is used to dispatch update notification events to the appropriate `TTXlaTableHandler`. A list of `TableHandler` objects is maintained in the class. As update notifications are received from XLA, the appropriate `Handle` methods of the appropriate `TableHandler` is called to process each 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 include both of these objects in a `TTXlaTableList`. As XLA update notification records are fetched from XLA, they can be dispatched to the correct handler by simply calling `TTXlaTableList::HandleChange()`.

The constructor has two forms:

```
TTXlaTableList(TTXlaConnection *, unsigned int i)
```

Where *i* is the number of database objects to monitor.

Or:

```
TTXlaTableList(TTXlaConnection* 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.

Public members

None

Public methods

Method	Description
add()	Adds a `TableHandler` to the list.
del()	Deletes a `TableHandler` from the list.
HandleChange()	Processes a record obtained from `ttXlaUpdateDesc_t`.

add()

```
void add(TTXlaTableHandler* h);
```

Adds a `TableHandler` to the list.

del()

```
void del(TTXlaTableHandler* h);
```

Deletes a `TableHandler` from the list.

HandleChange()

```
void HandleChange(ttXlaUpdateDesc_t* p, TTStatus&);
```

When a `ttXlaUpdateDesc_t` is received from XLA, it can be processed by calling this method. This method determines which table the record references and calls the `HandleChange()` method of the appropriate `TableHandler`.

Usage

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

Internal classes

These classes are provided in the C++ class library and are used internally in other classes. Their implementation may change.

- `TTCommand`: The base class of TTCmd, the `TTCommand` class provides a low level C++ mapping for ODBC statements (SQLHSTMT) and ODBC function calls.
- `TTParameter`: TTCmd implements self-defining parameters through the `TTParameter` class.
- `TTColumn`: TTCmd implements self-defining columns through the `TTColumn` class.

- `TTXlaTable`: `TTXlaTable` objects define information about tables which are being monitored for changes.
- `TTXlaColumn`: TTXlaRowViewer uses this function to define a single column in a table.

Index