Oracle TimesTen In-Memory Database TTClasses Guide

Release 6.0

B25263-03

ORACLE
TIMESTEN

For last-minute updates, see the TimesTen release notes.

March 2006

Printed in the United States of America

Contents

Introduction

At the core of TimesTen is an in-memory database (IMDB) that provides extremely high performance through a standard ODBC and SQL interface. Unlike other RDBMS implementations of ODBC, access to TimesTen row ODBC is extremely fast. But despite its wide acceptance as an industry standard, ODBC can sometimes be challenging to use. Demand for an API that is easier to use than ODBC, but does not sacrifice performance, led to the development of TTClasses.

Introducing the TimesTen C++ Interface Classes (“TTClasses”)

The TimesTen C++ Interface Classes library (hereafter “TTClasses”) was written to provide an easy-to-use, high-performance interface to TimesTen. This C++ class library provides “wrappers” around the most common ODBC functionality. Using this library allows for 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.   Where possible, the C++ library uses TimesTen in an optimal manner (e.g., autocommit is disabled by default), and parameterized SQL is strongly encouraged, and its use greatly simplified, when compared to hand-coded ODBC.

Scope of TTClasses

TTClasses is a wrapper around all *major* ODBC functionality – all that's necessary to write a high performance real-time application.

In addition to providing a C++ interface to TimesTen's ODBC interface, TTClasses also provides an interface to 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 will be reported through XLA to the monitoring application. TTClasses provides an easy-to-use interface to the most commonly-used aspects of XLA functionality.

TTClasses is delivered in source code form, and can be compiled for use on various platforms. See the file LICENSE.txt for the TTClasses license agreement.

Note: This version of TTClasses has been completely integrated with TimesTen 6.0. Previous versions of TTClasses were distributed separately from TimesTen, so these earlier versions of TTClasses were compatible with multiple TimesTen versions. Starting with TimesTen 6.0, TTClasses is no longer a separate product and as such no longer is tested or supported in combination with any other TimesTen release besides the one that it ships with.

1

Compiling TTClasses

Past versions of TTClasses used the “configure” script for configuring a TTClasses makefile.

TTClasses in TimesTen 6.0 comes pre-configured during TimesTen installation, so all you need to do to build TTClasses is type “make” (UNIX) or “nmake” (Windows).

Example 1.1 Compiling TTClasses on Unix

To build TTClasses and run the TTClasses demo programs, you need to make sure your shell environment variables are set correctly. Assuming your TimesTen 6.0 instance is named “tt60” and is installed at

```
/opt/TimesTen/tt60
```

simply run one of the following scripts (better still, add a call to one of these scripts in your login initialization script, e.g., `.profile` or `.cshrc`):

```
/opt/TimesTen/tt60/bin/ttenv.sh    (sh/ksh/bash)
/opt/TimesTen/tt60/bin/ttenv.csh  (csh/tcsh)
```

If you chose an instance name other than `tt60`, use that name in place of `tt60` in the above directory paths.

Once your PATH and shared-library load path are configured properly, chdir to the TTClasses directory and type:

```
$ cd /opt/TimesTen/tt60/ttclasses
$ make
```

Compiling TTClasses on Windows is also straight-forward.

Example 1.2 Compiling TTClasses on Windows

Change to the TTClasses installation directory (by default, this is):

```
C:\TimesTen\tt60\ttclasses
```

To compile this Makefile, you will need to have your PATH, INCLUDE, and LIB environment variables point to the correct Visual Studio directories. There is a batch file 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. Run this batch file.

If you are using Visual Studio .NET:

```
C:\TimesTen\tt60\ttclasses> nmake /f Makefile.vsdotnet
```

If you are using VC++ 6.0:

```
C:\TimesTen\tt60\ttclasses> nmake
```

Compilation Options

When compiling TTClasses, there are multiple “make targets” available.

These are

- **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 & static)
- **debug** -- build the debug libraries (shared & static)
- **clean** -- deletes the TTClasses libraries and object files

Note: Most users will want to use the shared, optimized version of TTClasses, which is the default (“all”) when you do “make” (or “nmake”) with no options.

To choose another make target, use the name of the make target on the command line to make.

Example 1.3 Compiling a debug, shared library

To build a shared, debug version of TTClasses:

(Unix)

```
$ make clean shared_debug
```

(Windows)

```
C:\TimesTen\tt60\ttclasses> nmake clean shared_debug
```

Compiling TTClasses for client/server

To compile TTClasses for client/server mode, use the “MakefileCS” makefile.

Example 1.4 Compiling TTClasses in client/server mode

To build a client/server version of TTClasses:

(Unix)

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

(Windows)

```
C:\TimesTen\tt60\ttclasses> nmake /f MakefileCS clean all
```

Installing TTClasses (Unix only)

After compilation, you will probably want to 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 with compiling TTClasses.

Note that installation occurs automatically after compilation on Windows.

Example 1.5 Installing TTClasses on Unix

```
$ cd /opt/TimesTen/tt60/ttclasses
$ make install
```

TTClasses compiler macros

In general, you will not need to manipulate the TTClasses Makefile. The following section discusses the TTClasses compiler macros used in these Makefiles, in case you ever need to modify the TTClasses Makefile by hand.

There are a few different categories of compiler macros used within the TTClasses source code.

For UNIX Makefiles, these flags can be added to your Makefile with “–D<flagname>”; for Windows, you use “/D<flagname>”.

TTEXCEPT -- “throw C++ exceptions”

Compile TTClasses with “-DTTEXCEPT” to make TTClasses throw C++ exceptions. All of the TTClasses demo programs assume that exceptions are turned on, and all TTClasses testing is done with exceptions turned on.

Note that if you use exceptions, you will have to put try/catch blocks around all TTClasses function calls (and catch exceptions of type “TTStatus” — see the section on TTStatus in a later chapter).

And note that if you *do not* use exceptions, you will have to check the “TTStatus::rc” value after every single TTClasses function call (checking for != SQL_SUCCESS). See the description of TTStatus for more information.

USE_OLD_CPP_STREAMS -- “use old C++ iostream code”

There are at least two major types of C++ streams, and they are not generally compatible with each other. If you’re trying to write robust code (and who isn’t?), you should not use both stream implementations inside a program.

If your program uses old C++ streams (e.g., if your code has “#include <iostream.h>”), then *you must* compile TTClasses with the macro “-DUSE_OLD_CPP_STREAMS” in order to be compatible with the rest of your program code.

If your program uses new C++ streams (e.g., your code has “#include <iostream>”), then *you must not* use this compiler macro.

TTDEBUG -- “generate additional debugging & error checking logic”

Compile TTClasses with “-DTTDEBUG” to generate extra debugging information. Of course, this extra information reduces performance (a little), so this flag should in general be used 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.

Additional (platform-specific) compiler macros

The following compiler macros are all specific to a particular platform and/or compiler combination. You should (almost) never have to specify these compiler macros by hand (their use is determined by the Makefile chosen by the “configure” program).

GCC

Compile TTClasses with “-DGCC” when using `gcc` on any platform.

HPUX

Compile TTClasses with “-DHPUX” when compiling on HP-UX.

MERANT

Compile TTClasses with “-DMERANT” when using the Merant ODBC Driver Manager.

1

Class Descriptions

This chapter contains descriptions of all classes in the external interface to TTClasses, plus brief descriptions of some of the other (internal) TTClasses.

Classes most commonly used by applications

TTStatus, TTError, TTWarning

The TTStatus class is used by other classes in the TTClasses collection to report errors and warnings. You can think of TTStatus as a value-added C++ wrapper around the ODBC function SQLError.

TTError is a subclass of TTStatus, and is used to encapsulate ODBC errors (return codes: SQL_ERROR, SQL_INVALID_HANDLE). 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 (e.g., logging ODBC warnings to an application's log is usually appropriate; but ODBC errors usually need to be programmatically handled).

Public Members

`rc`

This is the return code from the failing ODBC call. Typical values for this field are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR and SQL_NO_DATA_FOUND.

`native_error`

This is the TimesTen native error number (if any) for the failing ODBC call.

`odbc_error`

This is the ODBC error code for the failing ODBC call.

`err_msg`

This is the ASCII printable error message for the failing ODBC call.

Public Methods

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

This method can be used to emit a “pretty-printed” version of the error to a stream.

Example 1.1 Using TTStatus with ostreams

```
TTStatus stat;
// ...
cerr << "Error fetching data: " << stat << endl;
```

Usage

Depending on how the TTClasses library was built, TTStatus objects are used in one of two different ways.

If the library was built with the `TTEXCEPT` preprocessor variable defined (this is the default and recommended use of TTClasses), TTStatus objects are thrown as exceptions whenever an error occurs. This allows C++ applications to use {try/catch} in C++ to detect and recover from failure, resulting in very readable source code.

Example 1.2 Using TTStatus with exceptions

```
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
}
```

If you build TTClasses without the TTEXCEPT preprocessor variable defined, TTStatus objects are returned (by reference) from most method calls. The caller must explicitly check for errors after each method call, as demonstrated in the next example.

Example 1.3 Using TTStatus without exceptions

```
TTStatus stat;
[...]
cmd1.Prepare(&conn, "select * from foo", stat);
if (stat.rc) {
  cerr << "Error preparing statement: " << stat << endl;
  // Rollback, exit(), throw -- whatever is appropriate
}
cmd2.Prepare(&conn, "insert into foo values(?,?,?)",
             stat);
```

```
if (stat.rc) {
  cerr << "Error preparing statement: " << stat << endl;
  // Rollback, exit(), throw -- whatever is appropriate
}
cmd3.Prepare(&conn, "update foo set x = ? where y = ?",
             stat);
if (stat.rc) {
  cerr << "Error preparing statement: " << stat << endl;
  // Rollback, exit(), throw -- whatever is appropriate
}
conn.Commit(stat) ;
if (stat.rc) {
  cerr << "Error in commit: " << stat << endl;
  // Rollback, exit(), throw -- whatever is appropriate
}
```

Note that with exceptions enabled, TTError objects are thrown for ODBC errors and TTWarnings are thrown for ODBC warnings.

The following example illustrates how TTError and TTWarning correlate to TTStatus: the two code fragments shown have identical behavior.

Example 1.4 Correlation between TTStatus, TTError and TTWarning

```
// first code fragment: using TTStatus
try {
  // some TTClasses method calls
}
catch (TTStatus st) {
  if (st.rc == SQL_SUCCESS_WITH_INFO) {
    cerr << ‘’Warning encountered: ‘’ << st << endl;
  }
  else {
    cerr << ‘’Error encountered: ‘’ << st << endl;
  }
}

// second code fragment: using TTError & TTWarning
try {
  // some TTClasses method calls
}
catch (TTWarning warn) {
   cerr << ‘’Warning encountered: ‘’ << warn << endl;
}
catch (TTError err) {
    cerr << ‘’Error encountered: ‘’ << st << 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.

Public Members

None.

Public Methods

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

The Connect method is used to open a new connection to a TimesTen data store. The connection string specified in the connStr parameter is used to create the connection. For example,

Example 1.5

```
TTConnection conn;
TTStatus stat;
conn.Connect("DSN=mydsn", stat);
// Now this connection can be used to interact with
// TimesTen
```

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

Warnings can sometimes be generated when calling this method, which can often safely be ignored. The following logic is preferred when using ::Connect()

```
try {
  conn.Connect(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
}
```

`void Disconnect (TTStatus&)`

The Disconnect method is used to close a connection to a TimesTen data store.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`void Commit (TTStatus&)`

The Commit method is used to commit 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.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`void Rollback (TTStatus&)`

The Rollback method is used to abandon a transaction. Any changes made to the database through this connection since the last call to Commit() or Rollback() methods will be undone.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`bool isConnected()`

Returns true if the object is connected to TimesTen (using the Connect method) or false if not.

`HDBC getHdbc()`

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

`void SetIsoReadCommitted (TTStatus &)`

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

`void SetIsoSerializable (TTStatus &)`

Sets the connection's transaction isolation level to be TXN_SERIALIZABLE. In general, serializable isolation offers fair individual transaction performance but extremely poor concurrency. READ_COMMITTED isolation level (see method SetIsoReadCommitted() above) should be preferred over SERIALIZABLE isolation level in (almost) all situations.

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

Performs a blocking checkpoint operation on the data store, by calling the TimesTen built-in procedure ttCkptBlocking with the parameters (timeout, retries).

See the *TimesTen Reference Guide* for more information about ttCkptBlocking.

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

Performs a blocking checkpoint operation on the data store, by calling the TimesTen built-in procedure ttCkptBlocking with the parameters (timeout, retries).

This method is the newer, preferred version of **CheckpointFuzzy**.

See the *TimesTen Reference Guide* for more information about ttCkptBlocking.

`void CheckpointNonBlocking (TTStatus &)`

Note: This is the preferred type of checkpoint.

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

See the *TimesTen Reference Guide* for more information about `ttCkpt`.

`void DurableCommit (TTStatus &)`

Performs a durable commit operation on the data store (that is, a commit operation which flushes the in-memory log buffer to disk), by calling the TimesTen built-in procedure `ttDurableCommit`. See the *TimesTen Reference Guide* for more information about `ttDurableCommit`.

`void CompactDataStore (int blocks, TTStatus &)`

Performs a data store compaction operation.

If *blocks* > 0, this method calls the TimesTen built-in procedure `ttCompactTS` with the parameter *blocks*; if blocks <= 0, this method calls the TimesTen built-in procedure `ttCompact`. Frequent data store compaction is necessary to prevent fragmentation.

Note: TimesTen 6.0 performs automatic data store compaction, so the TTConnection::CompactDataStore() method is deprecated and will be removed in the next version of TTClasses.

`void SetLockWait (int secs, TTStatus &)`

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

See the *TimesTen Reference Guide* for more information about ttLockWait.

`void SetPrefetchCloseOn (TTStatus &)`

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

See the *TimesTen Developer's Guide* (chapter titled "Performing Tuning") for more information about TT_PREFETCH_CLOSE.

`void SetPrefetchCloseOff (TTStatus &)`

Turns off the TT_PREFETCH_CLOSE connection option.

`void SetPrefetchCount (int numrows, TTStatus &)`

This method allows a user application to tune the number of rows that the TimesTen ODBC driver will internally fetch at a time for a SELECT statement. The parameter “numrows” can be between 1 and 128.

Note: This method is not equivalent to executing TTCmd::FetchNext() multiple times; instead, proper use of this parameter will reduce the amount of time that each call to TTCmd::FetchNext() takes.

See the *TimesTen Developer’s Guid*e for more information on TT_PREFETCH_COUNT.

`void SetAutoCommitOff (TTStatus &)`

Sets AUTOCOMMIT off for the connection.

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

`void SetAutoCommitOn (TTStatus &)`

Sets AUTOCOMMIT on for the connection, which means that every SQL statement will now occur 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().

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

Returns the value of 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 TimesTen utility ttStatus, for example.

The context value is returned through the first parameter, “output”; this method must be called with an array of char[17] (or larger) for the “output” parameter.

This method calls the TimesTen Built-in Procedure `ttContext`. See the *TimesTen Reference Manual* for more information on `ttContext`.

Usage

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

Multi-threaded applications that wish to use TimesTen from multiple threads simultaneously must create more than one TTConnection object. Typically one of two strategies will be used:

- One TTConnection object will be created for each thread when the thread is created, or

- A “pool” of TTConnection objects will be created when the application process starts and they will be “shared” by the threads in the process. See the TTConnectionPool class for additional information about this option.

Connections are relatively heavyweight. It is not desirable for an application to constantly be connecting to and disconnecting from TimesTen, and will cause performance problems. Instead, establish database connections at the beginning of the application process and reuse them for the life of the process.

TTCmd

The TTCmd object 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.

Public Methods

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

Before a TTCmd can be used, a SQL statement (such as SELECT, INSERT, UPDATE or DELETE) must be associated with it. This association is done using the Prepare method. The Prepare method also performs all the necessary plan generation and optimization required to insure that the SQL statement will be executed in an efficient manner.

The Prepare method does not execute the statement; rather, it simply associates the statement with the TTCmd object and determines the plan to be used to execute the statement.

With TimesTen, statements are typically *parameterized* for performance. For example, rather than issuing the following two SQL statements:

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

…it is vastly more efficient to Prepare a single parameterized statement and execute it multiple times:

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

The value for “?” will be specified at runtime using the setParam methods described below.

There is no need to explicitly “bind” columns or parameters to a SQL statement, as is the case with ODBC directly. TTCmd will automatically define and bind all necessary columns and parameters at Prepare time.

Note that Prepare is a relatively expensive operation. Typically at the beginning of an application when a connection is established to TimesTen (using TTConnection::Connect), the application will Prepare all TTCmd objects associated with the connection.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`void Execute (TTStatus&)`

The Execute method is used to invoke 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, 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. The Close method must be invoked 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 need not be called.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

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

The ExecuteImmediate method is used to invoke a SQL statement that has not been previously prepared.

ExecuteImmediate is a convenient alternative to Prepare/Execute when a SQL statement is only executed a few times, for example, DDL statements (e.g., CREATE TABLE, DROP TABLE, …) and for infrequently used DML statements that do not return a result set (e.g., DELETE FROM <tablename>, …).

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.

`int FetchNext (TTStatus & stat)`

After executing a prepared SQL SELECT statement using Execute, the FetchNext method is used to fetch rows from the answer set, one at a time.

After fetching a row of the answer set, use the various overloaded versions of the getColumn method (described below) 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, it is imperative that the answer set be closed using the Close method. Once the Close method is called, the FetchNext method can not be used to fetch additional rows from the answer set.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`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 deleted 10 rows, getRowCount returns 10.

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

Calling this function for a statement will limit the number of rows subsequently fetched 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, i.e. 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.

`int getMaxRows (TTStatus &stat)`

This function will return the current limit of number of rows returned by a SELECT from this TTCmd. A return value of 0 means all rows are returned.

`void setParamNull (int i)`

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

The various overloaded setParam methods are all described together in this section.

The setParam methods are used to set the value of parameters prior to 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 prior to executing the statement.

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. Various overloaded versions of setParam take different data types for the second argument.

The setParamNull method can be used by an application to indicate that the value for a particular parameter should be the SQL “NULL” pseudo-value.

This version of the TTClasses library does not support a robust set of 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. As such, 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.

The following table shows the supported SQL data types and the appropriate version(s) of setParam to use for each parameter type.

Note that SQL data types not mentioned are not supported in this version of TTClasses.

Table 1: *TTCmd::setParam* variants for supported SQL data types

SQL Data Type	**setParam variants supported**
SQL_TINYINT (1-byte int)	setParam (int, SQLTINYINT)
SQL_SMALLINT (2-byte int)	setParam (int, SQLSMALLINT)
SQL_INTEGER (4-byte int)	setParam (int, SQLINTEGER)
SQL_BIGINT (8-byte int)	setParam (int, SQLBIGINT)
SQL_FLOAT SQL_REAL	setParam (int, float)
SQL_DOUBLE	setParam (int, double)
SQL_DECIMAL	setParam (int, char*) setParam (int, const char*)
SQL_NUMERIC	setParam (int, char*) setParam (int, const char*)

Table 1: *TTCmd::setParam* variants for supported SQL data types

SQL_CHAR	setParam (int, char*) setParam (int, const char*)
SQL_VARCHAR	setParam (int, char*) setParam (int, const char*)
SQL_WCHAR	setParam (int, SQLWCHAR*, int len)
SQL_WVARCHAR	setParam (int, SQLWCHAR*, int len)
SQL_BINARY	setParam (int, const void*, int len)
SQL_VARBINARY	setParam (int, const void*, int len)
SQL_DATETIME	setParam (int, TIMESTAMP_STRUCT*)
SQL_TIMESTAMP	setParam (int, TIMESTAMP_STRUCT*)
SQL_DATE	setParam (int, DATE_STRUCT*)
SQL_TIME	setParam (int, TIME_STRUCT*)

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

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

`bool isColumnNull (int i)`

All the overloaded varieties of the getColumn and getColumnNullable methods are described in this section.

The getColumn and getColumnNullable methods can be used to fetch the values for columns of the current row of the answer set. Before the getColumn and/or 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 isColumnNull method provides another way to ask whether a column's value is NULL.

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.

The getColumnNullable method is similar to the getColumn method. However, in addition to the behavior of getColumn, the getColumnNullable method also returns an indication of whether the column's value is the SQL "NULL" pseudo-value. If the column's value is NULL the second parameter's value is set to an "eye catcher" value (e.g., -9999), and the return value from the method is true. If the column's value is not NULL it is returned in the variable pointed to by the second parameter and the getColumnNullable method returns false.

This version of the TTClasses library does not support a robust set of 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.

The following table shows the supported SQL data types and the appropriate version(s) of getColumn to use for each parameter type.

Table 2: *TTCmd::getColumn[Nullable]* variants for supported SQL data types

SQL Data Type	**getColumn variant(s) supported**
SQL_BIT	getColumn (int i, int*) getColumnNullable (int i, int*)
SQL_TINYINT (1-byte int)	getColumn (int i, SQLTINYINT*) getColumnNullable (int i, SQLTINYINT*)
SQL_SMALLINT (2-byte int)	getColumn (int i, SQLSMALLINT*) getColumnNullable (int i, SQLSMALLINT*)
SQL_INTEGER (4-byte int)	getColumn (int i, SQLINTEGER*) getColumnNullable (int i, SQLINTEGER*)
SQL_BIGINT (8-byte int)	getColumn (int i, SQLBIGINT*) getColumnNullable (int i, SQLBIGINT*)
SQL_FLOAT SQL_REAL	getColumn (int i, float*) getColumnNullable (int i, float*)
SQL_DOUBLE	getColumn (int i double*) getColumnNullable (int i, double*)
SQL_DECIMAL	getColumn (int i, char**) getColumnNullable (int i, char**)

Table 2: *TTCmd::getColumn[Nullable]* variants for supported SQL data types

SQL_NUMERIC	getColumn (int i, char**) getColumnNullable (int i, char**)
SQL_CHAR	getColumn (int i, char**) getColumnNullable (int i, char**)
SQL_VARCHAR	getColumn (int i, char**) getColumnNullable (int i, char**)
SQL_WCHAR	getColumn (int i, SQLWCHAR*) getColumnNullable (int i, SQLWCHAR*)
SQL_WVARCHAR	getColumn (int i, SQLWCHAR*) getColumnNullable (int i, SQLWCHAR*)
SQL_BINARY	void getColumn (int i, void** binPP, int* lenP) void getColumnNullable (int i, void** binPP, int* lenP)
SQL_VARBINARY	void getColumn (int i, void** binPP, int* lenP) void getColumnNullable (int i, void** binPP, int* lenP)
SQL_DATETIME	getColumn (int i, TIMESTAMP_STRUCT*) getColumnNullable (int i, TIMESTAMP_STRUCT*)
SQL_TIMESTAMP	getColumn (int i, DATE_STRUCT*) getColumnNullable (int i, DATE_STRUCT*)
SQL_DATE	getColumn (int i, DATE_STRUCT*) getColumnNullable (int i, DATE_STRUCT*)
SQL_TIME	getColumn (int i, TIME_STRUCT*) getColumnNullable (int i, TIME_STRUCT*)

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

`int getColumnLength (int cno)`

Returns the length of column # *cno*. This is generally only useful when accessing columns of type VARBINARY or NVARCHAR. The value returned is between 0 and the column's precision (see getColumnPrecision method below), inclusive.

`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 done 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 for too long on rows, causing concurrency problems, as well as memory leaks and other errors.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`void Drop (TTStatus&)`

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

It is much 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.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error, upon return from the method.

`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.

Note that there is no default query timeout value.

Usage

Each SQL statement executed multiple times in a program should have its own TTCmd object. During program initialization these TTCmd objects should be Prepare()'d, once each, and then Execute()'d 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() + Execute(), instead), and is also incompatible with insert/update/delete statements which are subsequently queried using getRowcount() to see how many rows were inserted/updated/deleted. These limitations have been placed on ExecuteImmediate() to discourage its use except in a few situations (e.g., creating or dropping a table).

DDL queries

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 only provide meaningful results when a statement has previously been prepared (see the Prepare method above).

`int getNParameters ()`

Returns the number of input parameters.

`int getNColumns ()`

Returns the number of output columns.

`int getParamType (int pno)`

Returns the data type of parameter # *pno*. The value returned is the parameter's ODBC type (e.g., SQL_INTEGER, SQL_REAL, SQL_BINARY, SQL_CHAR, etc.) as found in <sql.h>; additional TimesTen types (SQL_WCHAR, SQL_WVARCHAR) can be found in the TimesTen header file <timesten.h>.

`int getColumnType (int cno)`

Returns the data type of column # *cno*. The value returned is the parameter's ODBC type (e.g., SQL_INTEGER, SQL_REAL, SQL_BINARY, SQL_CHAR, etc.) as found in <sql.h>; additional TimesTen types (SQL_WCHAR, SQL_WVARCHAR) can be found in the TimesTen header file <timesten.h>.

`const char * getColumnName (int cno)`

Returns the name of the column # *cno*.

`int getColumnPrecision (int cno)`

Returns the precision of column # *cno*. This value is generally only interesting when generating output from table columns of type CHAR, VARCHAR, BINARY, VARBINARY, NCHAR and NVARCHAR.

Batch Operations

TimesTen supports the ODBC function SQLBindParams for batch insert, update and delete operations, and 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 example program “bulktest.cpp” for an example of using the batch operation functionality.

This section describes the TTCmd methods which expose the batch insert/update/delete functionality to TTClasses users.

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

PrepareBatch() is the analog of Prepare() for batch insert/update/delete statements. This function’s TTConnection* and const char* sqlP and TTStatus& parameters are used the same as in Prepare().

There is only one valid value for the “level” parameter currently (in the future, additional values will be added). This value is

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().

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

The various overloaded BindParameter methods are all described together in this section.

The BindParameter methods are used to bind an array of values (one for each parameter) for a statement compiled using PrepareBatch(). The “batchSize” parameter of this call must match the value of “batchSize” specified in PrepareBatch(); similarly, the bound arrays should be at least this large. It is left up to the user to determine the correct type to bind to each parameter; note that if the wrong type is bound, a runtime error will be written to the TTClasses global logging facility at the TTLog::TTLOG_ERR logging level.

Prior to each invocation of ExecuteBatch(), the user application should fill these arrays with valid parameter values.

For four SQL types (SQL)_[VAR]BINARY and SQL_W[VAR]CHAR), an additional SQLLEN* parameter, an array of SQLLEN’s, is required, to hold the length of parameter values. This additional array must be at least “batchSize” in length, and filled with valid length values prior to calling ExecuteBatch().

The following table shows the supported SQL data types and the appropriate version(s) of BindParameter to use for each parameter type.

Table 3: *TTCmd::BindParameter* variants for supported SQL data types

SQL Data Type	BindParameter variants supported
SQL_TINYINT (1-byte int)	BindParameter (... SQLTINYINT*...)
SQL_SMALLINT (2-byte int)	BindParameter (...SQLSMALLINT*...)
SQL_INTEGER (4-byte int)	BindParameter (...SQLINTEGER*...)
SQL_BIGINT (8-byte int)	BindParameter (...SQLBIGINT*...)
SQL_FLOAT SQL_REAL	BindParameter (...float*...)
SQL_DOUBLE	BindParameter (...double*...)
SQL_DECIMAL	BindParameter (...char**...)
SQL_NUMERIC	BindParameter (...char**...)
SQL_CHAR	BindParameter (...char**...)
SQL_VARCHAR	BindParameter (...char**...)
SQL_WCHAR	BindParameter (...SQLWCHAR**, SQLLEN*...)
SQL_WVARCHAR	BindParameter (...SQLWCHAR**, SQLLEN*...)
SQL_BINARY	BindParameter (...const void**, SQLLEN*...)
SQL_VARBINARY	BindParameter (...const void**, SQLLEN*...)
SQL_DATETIME	BindParameter (... TIMESTAMP_STRUCT*...)
SQL_TIMESTAMP	BindParameter (... TIMESTAMP_STRUCT*...)

SQL_DATE	BindParameter (...DATE_STRUCT*...)
SQL_TIME	BindParameter (...TIME_STRUCT*...)

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

`setParamNull(int pno, unsigned short rowno)`

These two methods are used to set the length, or null-ness, of one of the bound parameter values, prior to 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; the third parameter of setParamLength, *len*, specifies the length being set.

For types apart from SQL_[VAR]BINARY and SQL_W[VAR]CHAR, these are the only methods available to explicitly set the length/null-ness of a value prior to a ExecuteBatch() call. For these four types, the length and nullability can also be explicitly set through manipulation of the SQLLEN* array, which was the 4th parameter to the BindParameter() call for these types.

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

After preparing a SQL statement with PrepareBatch(), and then calling BindParameter() for each parameter ("?") in that SQL statement, ExecuteBatch() is used to execute that statement "numRows" times. The value of "numRows" must be no more than the "batchSize" specified in the PrepareBatch() and BindParameter() calls; however, "numRows" can be less than "batchSize", as required by the application logic.

Prior to calling ExecuteBatch(), the application should fill the arrays of parameters bound using BindParameter() with valid values. Null values can be specified as necessary using setParamNull(int pno, int rowno) (see above).

TTConnectionPool

The TTConnectionPool class is used by multi-threaded applications to manage a pool of connections.

In general, multi-threaded applications can be written using one of two basic strategies:

- If there are a relatively small number of threads, and the threads are long-lived, each thread could be assigned to a different connection, which are used for the duration of the application. In this scenario, the TTConnectionPool class is not necessary.
- If there are a large number of threads in the process, and/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 1-2 concurrent direct-memory database connections per CPU of the database server. For no reason should your number of concurrent direct-memory database connections (i.e., the size of your connection pool) be more than twice as many CPUs on the database server. In client/server mode, however, TimesTen will support 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 method (which would actually connect them to TimesTen). The application then uses the TTConnectionPool::AddConnectionToPool method to put the connection objects into the pool. It then calls TTConnectionPool::ConnectAll to connect all the connections to TimesTen. Threads wanting to use TimesTen then use getConnection and freeConnection methods to get and return idle connections.

Public Members

None.

Public Methods

`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.

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

Once a number of 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.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

`TTConnection* getConnection (int timeout_millis=0)`

When a thread wishes to make use of TimesTen it uses the getConnection method to "check out" an idle connection from the connection pool. 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 specified, getConnection will wait for a free connection for no more than *timeout* milliseconds; if no connection is available in that time then getConnection will return NULL to the caller.

`void freeConnection (TTConnection*)`

This method is used to return 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.

`void DisconnectAll (TTStatus&)`

All connections in the connection pool will be disconnected from TimesTen.

Applications must call DisconnectAll prior to termination in order to avoid overhead associated with process failure analysis and recovery.

If exceptions are enabled, a TTStatus object will be thrown as an exception if an error occurs. If exceptions are disabled, the TTStatus& object passed as the last parameter to the method will contain information about any error upon return from the method.

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

This method allows the user application to query the TTConnectionPool for status information. Data returned is:

- nGets: # of calls to getConnection()
- nFrees: # of calls to freeConnection()
- nWaits: # of times a call to getConnection() had to wait before returning a connection
- nTimeouts: # of calls to getConnection() that timed out
- maxInUse: High water mark for the most number of connections in use at one time
- nForcedCommits: The 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 (includes the TTClasses logging facility)

The TTGlobal class provides a logging facility within TTClasses, to aid program development and application logging.

Public Members

None.

Public Methods

`static void setLogStream (ostream & str)`

This method specifies where TTClasses logging information should be sent. By default, if logging is turned on, TTClasses logs to standard error; using this method, a user application can log to a file (or any other ostream &); for example, to log to the text file “app_log.txt”:

Example 1.6

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

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

This method specifies the verbosity level of TTClasses logging. There are six permissible logging levels:

- TTLog::TTLOG_NIL : no logging of any kind
- TTLog::TTLOG_FATAL_ERR : a very minimal logging level; only TTClasses fatal errors (e.g., serious misuse of TTClasses methods) will be logged
- TTLog::TTLOG_ERR : any errors (such as SQL_ERROR return codes) will be logged; also, everything from TTLog::TTLOG_FATAL_ERR.
- TTLog::TTLOG_WARN : warnings will be logged, in addition to everything in TTLog::TTLOG_ERR
- TTLog::TTLOG_INFO : informational messages will be logged, in addition to everything in TTLog::TTLOG_WARN
- TTLog::TTLOG_DEBUG : the most verbose logging level, all sorts of debugging information gets logged in this log level.

By default, TTClasses logging starts out as TTLog::TTLOG_INFO. At this level, things such as prepared query plans are logged, as are the values of bound parameters prior to execution of a TTCmd.

To set the different logging level to TTLog::TTLOG_ERR, for example, you simply need to add the following line to your program:

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

`static void disableLogging()`

This method disables all logging; note that the following two statements are identical:

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

Miscellaneous Comments about TTGlobal

TTGlobal's logging facility can be extremely useful for debugging all sorts of problems inside a TTClasses program. Note, however, that the most verbose logging levels (TTLog::TTLOG_INFO and TTLog::TTLOG_DEBUG) are really quite verbose, and can generate an extremely large amount of output. As a result, it's probably a good idea to only use these logging levels during development, or when trying to diagnose a bug.

When logging from a multi-threaded program, you may encounter the problem where log output different program threads get mixed up when being written to disk. To alleviate this problem, disable ostream-buffering with the ios_base::unitbuf iostream manipulator.

Here's how to send TTClasses logging to the file "app_log.txt" at logging level TTLog::TTLOG_ERR, and make sure that logging to this file is not buffered (i.e., so that different threads' outputs aren't mixed together):

Example 1.7

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

System catalog classes (TTCatalog)

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

Use of the TTCatalog class is quite different in flavor 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

The following ODBC functions will be incorporated into later versions of TTCatalog

- SQLForeignKeys()
- SQLPrimaryKeys()[1]

TTCatalog

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

Public Members

None.

Public Methods

`(constructor) TTCatalog (TTConnection*)`

The TTCatalog constructor simply caches the TTConnection* parameter and initializes all the internal data structures appropriately. In order to actually use the TTCatalog object, you must first call fetchCatalogData() (described below).

`fetchCatalogData (TTStatus &)`

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

Subsequent use of the constructed TTCatalog object is completely "off-line" after it is constructed, it is no longer attached to the database.

You must call this method before you use any of the other TTCatalog accessor methods, or else they will not return useful information[2].

Here is a simple piece of code demonstrating the use of TTCatalog (minus the usual checking of stat.rc after the two database calls):

1. Note: In the TimesTen ODBC driver, SQLStatistics returns a superset of the information of SQLPrimaryKeys().
2. Note that this is a change in TTCatalog behavior that first originated in TTClasses version 1.6. Previously, the TTCatalog constructor performed the functionality described by the fetchCatalogData() method. This change was made to address concerns about inadvertent fetching of all catalog data by a naive user of the TTCatalog class.

Example 1.8

```
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;
```

`int getNumTables()`

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

`int getNumUserTables()`

Returns the number of user tables in the database.

`int getNumSysTables()`

Returns the number of system tables in the database.

`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 the next section for usage of TTCatalogTable objects.

`const TTCatalogTable & getTable (int i)`

Returns a constant reference to the TTCatalogTable corresponding to the i^{th} table 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 <= i <= getNumTables().

`const TTCatalogTable & getUserTable (int i)`

Returns a constant reference to the TTCatalogTable corresponding to the i^{th} user table 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 <= i <= getNumUserTables().

TTCatalogTable

The TTCatalogTable class is used to store all metadata information about a table's columns and indexes.

Public Members

None.

Public Methods

`const char * getTableOwner()`

Returns the table's owner.

`const char * getTableName()`

Returns the table's name.

`int getNumColumns()`

Returns the number of columns in the table.

`int getNumIndexes()`

Returns the number of indexes on the table.

`const TTCatalogColumn & getColumn (int i)`

Returns a constant reference to the TTCatalogColumn corresponding to the ith column 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().

`const TTCatalogIndex & getIndex (int i)`

Returns a constant reference to the TTCatalogIndex corresponding to the ith index 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().

TTCatalogColumn

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

Public Members

None.

Public Methods

`const char * getColumnName()`

Return the column's name.

`int getDataType()`

Returns an integer representing the data type of the column. This is the standard ODBC SQL Type.

`const char * getTypeName()`

Returns the (database-dependent) name which corresponds to the type returned by getdata type().

`int getNullable()`

Returns SQL_NO_NULLS, SQL_NULLABLE, or SQL_NULLABLE_UNKNOWN.

`int getPrecision()`

Returns the column's precision.

`int getLength()`

Returns the column's length.

`int getScale()`

Returns the column's scale.

`int getRadix()`

Returns the column's radix.

TTCatalogIndex

The TTCatalogIndex class is used to store all information about an index (of the TTCatalogTable it is associated with).

Public Members

None.

Public Methods

`const char * getIndexName()`

Returns the index's name.

`const char * getIndexOwner()`

Returns the index's owner.

`const char * getTableName()`

Returns the name of the table on which this index lives.

`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.

`bool isUnique()`

Returns whether the index is a unique index; true means it is unique, false means it is not unique.

`int getNumColumns()`

Returns the number of columns in the index.

`const char * getColumnName (int i)`

Returns the column name of the ith column in the index.

`char getCollation (int i)`

Returns the collation of the ith column in the index. Values returned are ‘A’ for ascending and ‘D’ for descending index order.

Internal classes

These classes are provided in the C++ class library and are used “under the covers” of the classes documented above.

TTCommand

The base class of TTCmd, the TTCommand class provides a low level C++ mapping for ODBC statements (SQLHSTMTs) and ODBC function calls.

The details of TTCommand are intentionally not provided here, since TTCommand is not part of the public interface to TTClasses. The user of TTClasses is strongly discouraged from depending on any part of TTCommand, as this class is considered an internal implementation detail which can change at any time.

TTParameter

TTCmd implements self-defining parameters through the TTParameter class.

The details of TTParameter are intentionally not provided here, since TTParameter is not part of the public interface to TTClasses. The user of TTClasses is strongly discouraged from depending on any part of TTParameter, as this class is considered an internal implementation detail which can change at any time.

TTColumn

TTCmd implements self-defining columns through the TTColumn class.

The details of TTColumn are intentionally not provided here, since TTColumn is not part of the public interface to TTClasses. The user of TTClasses is strongly discouraged from depending on any part of TTColumn, as this class's is considered an internal implementation detail which can change at any time.

Change Monitoring (XLA) Classes

TTClasses provides a set of C++ classes which make it easy to write applications that use TimesTen's *Transaction Log API*, also called 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 the monitored table(s), the application using XLA is informed of the changes.

TTClasses provides a set of classes to make it simpler to write XLA applications. The provided classes are:

- **TTXlaConnection**. This class defines a specialized non-persistent connection to a TimesTen data store. A non-persistent XLA application uses a connection of this type rather than a generic TTConnection.
- **TTXlaPersistConnection**. This class, derived from TTXlaConnection, defines a persistent XLA connection to a TimesTen data store. A persistent XLA applications uses a connection of this type, rather than a (non-persistent) TTXlaConnection. This class is used in much the same way as TTXlaConnection, with a few additional methods for handling the new persistent XLA bookmarks. The mechanism behind persistent XLA is simply keeping an LSN (log sequence number) placeholder in the TimesTen transaction log, then walking through the transaction log at leisure. This is dramatically different from non-persistent XLA, which has a limited buffer size.
- **TTXlaTable**. This class encapsulates logic and catalog information associated with a particular table to be monitored.
- **TTXlaColumn**. This class encapsulates logic and catalog information associated with a particular column in a table that is being monitored.
- **TTXlaRowViewer**. This class provides an easy way to fetch column values from a particular update record.
- **TTXlaTableHandler**. This class provides methods that enable and disable change tracking for a table. Additionally, methods are provided to handle update notification records from XLA
- **TTXlaTableList**. Provides for 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.

These classes are described in more detail in the subsequent sections.

TTXlaConnection

TTXlaConnection is a wrapper around a TTConnection that can be used with XLA. It has a few basic methods in common with standard TTConnections (e.g., Connect, Disconnect), but cannot in general be used in the place of a TTConnection. TTXlaConnection is a connection to TimesTen dedicated to performing XLA operations.

Only one TTXlaConnection object can be connected to a TimesTen data store simultaneously.

Public Members

None

Public Methods

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

The Connect method is used to open a new XLA connection to a TimesTen data store. The connection string specified in the connStr parameter is used to create the connection.

`virtual void Disconnect (TTStatus&)`

The Disconnect method is used to close an XLA connection to a TimesTen data store.

`void setXlaBufferSize (SQLUBIGINT newsz, SQLUBIGINT* oldszP, TTStatus&)`

When an application makes a change to a TimesTen database, information describing the change is stored in a shared memory buffer until an XLA-enabled application can fetch the data. The setXlaBufferSize method can be used to set the size of this buffer for a particular data store. This method can only be used after the TTXlaConnection object has been connected to the database using the Connect method.

When a data store is created, the size of the XLA buffer defaults to zero. This has the effect of disabling XLA. Until a buffer is defined by this method applications may not make use of XLA.

The size of this buffer, once set, is stored persistently in the database. Even if a power failure or other system failure causes the database to be reloaded into memory, when reloaded the buffer size will be set as defined by the last call to the setXlaBufferSize method.

The caller to this method specifies the new size of the XLA buffer (in bytes). The previous buffer size is returned to the caller through the oldP argument. Errors, if any, are reported through the TTStatus argument.

void fetchUpdates (ttXlaUpdateDesc_t* arry, int maxrecs, int* recsP, TTStatus&)**

The fetchUpdates() 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. 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 which are returned by this method are those defined in the XLA specification. No C++ object-oriented encapsulation of these methods is provided. See also, however, the other classes in this chapter, as they are helpful in decoding the ttXlaUpdateDesc_t structures returned by this method.

ttXlaHandle_t getXlaHandle()

If an application wishes to use XLA functionality that does not have C++ encapsulations, the underlying ttXlaHandle_t value for this connection may be returned through this method. This value is used by the low level XLA functions.

Usage

An XLA application must create one and only one TTXlaConnection object, and use the Connect method to connect it to a data store. Once the object is connected, the setXlaBufferSize method is used to enable XLA.

Once XLA is enabled, the application should enter a loop in which the fetchUpdates() method is called repeatedly until application termination. This loop should fetch updates from XLA as rapidly as possible, to prevent the XLA buffer from filling. If the XLA buffer fills, subsequent changes to the database by all other applications will be rejected. Expensive processing should be deferred to other applications or other threads.

Finally, at application termination, it is recommended that the setXlaBufferSize method be used to set the XLA buffer size to its previous value. This must be performed before calling the Disconnect method of the object.

TTXlaPersistConnection

TTXlaPersistConnection is a wrapper around TTXlaConnection for use with persistent XLA. It has all the functionality of a standard TTXlaConnection (except for setXlaBufferSize(), which is meaningless for persistent XLA), and also has additional functionality associated with persistent XLA.

Note that one major difference between persistent and non-persistent XLA is that multiple TTXlaPersistConnection objects can be connected to a TimesTen data store simultaneously. Thus, persistent XLA is also *multi-user* XLA.

Public Members

None

Public Methods

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

The TTXlaPersistConnection::Connect() method has a different signature than TTConnection::Connect (and TTXlaConnection::Connect). This is because, in order to support persistent XLA, additional information needs to be supplied at data store connect time.

The extra parameters are bookmark (const char*) and createBookmark (bool).

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, and 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.

`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 yet or not.

Bookmark maintenance is the subject of the next four methods.

`void deleteBookmark (TTStatus&)`

This method deletes the bookmark that is currently attached to, so that the data store no longer keeps records relevant to that bookmark.

Note that without a bookmark a TTXlaPersistConnection cannot do anything useful, so this method also makes an implicit call to TTXlaPersistConnection::Disconnect().

Thus, if you simply want to disconnect from the data store, use TTXlaPersistConnection::Disconnect(); if you want to also delete the bookmark that you're attached to (and avoid having the data store keep track of update records relevant for your bookmark), then use TTXlaPersistConnection::deleteBookmark().

`void ackUpdates (TTStatus &)`

This method is used to advance the bookmark to the next set of updates. Once you have ACKnowledged a set of UPDATES ("ackUpdates") for a given bookmark, they can never be looked at again, on that bookmark. See the next two methods which are used for replaying a set of updates potentially multiple times.

`void getBookmarkIndex (TTStatus &)`

`void setBookmarkIndex (TTStatus &)`

Each ttXlaPersistConnection maintains a bookmark-within-a-bookmark, for replaying a batch of records potentially multiple times. You use the getBookmarkIndex() method to store the current place in the transaction log, and then you use setBookmarkIndex() to go back to the saved transaction log index.

Note that ackUpdates() invalidates the stored transaction log placeholder – after ackUpdates(), a call to setBookmarkIndex() returns an error, since it is no longer possible to go back to that earlier transaction log index.

Note that while some applications may require the ability to replay updates multiple times, most applications will probably not need this functionality.

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

This is the blocking version of the TTXlaConnection::fetchUpdates() method. It is used almost identically to fetchUpdate(), except that a fifth parameter (seconds: how long to wait for updates) is supplied.

Using this method allows the user application to push the "polling for updates" loop inside the TimesTen engine, instead of writing explicit nanosleep() calls. In truth, the net effect is pretty much the same.

See TTXlaConnection::fetchUpdates() for more information about the input, output and usage of this method.

Usage

TTXlaPersistConnection() is used almost identically as TTXlaConnection, except for a few minor differences:

- A persistent XLA application may 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.
- 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.
- The setXlaBufferSize() method is meaningless for TTXlaPersistConnection, and should not be used (it returns an error).
- There are several bookmark maintenance methods specific to TTXlaPersistConnection.

Once a persistent XLA connection is made, the application should enter a loop in which the fetchUpdates[Wait]() 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. OK, there's not quite the same danger here as with non-persistent XLA, but it's probably a good idea to work through the updates rapidly, so that your XLA application does not fall behind the overall transaction rate. After processing a batch of updates, your app should call ackUpdates() in order to acknowledge those updates and get ready for the next call to fetchUpdates(). A batch of updates can be replayed using the setBookmarkIndex() and getBookmarkIndex() methods. Also, if your persistent XLA application disconnects after fetchUpdates[Wait](), but before ackUpdates(), the next connection (with the same bookmark name – assuming it hasn't been deleted using deleteBookmark()) which calls fetchUpdates[Wait] will see that same batch of updates.

TTXlaTable

TTXlaTable objects define information about tables which are being monitored for changes.

Public Members

`TTXlaColumn col[TT_XLA_MAX_COLUMNS]`

The 'col' member defines an array of TTXlaColumn objects, one for each column in the table. These objects, in turn, have information about the name and type of each column in this table.

Public Methods

`void Define (TTXlaConnection* const char* owner, const char* name, TTStatus&)`

This method associates this TTXlaTable object with a particular named table in the data store. Input parameters to this method specify the connection to the data store, the name of the table's owner and the name of the table itself. Owner name must be specified, and does not default.

This method causes catalog information associated with this table to be fetched from the data store and populated into the TTXlaTable object.

`int getNCols()`

This method returns the number of columns in the table.

`int getColNumber (const char* colName)`

This method can be used to map a particular named column to a column number. It is particularly useful in combination with the various versions of TTXlaRowViewer, when it can be used to fetch column values by column name.

`void enableTracking (bool* oldValue, TTStatus&)`

Calling this method enables the tracking of changes to this table using XLA. Until this method is called, changes to this table are not available through XLA.

This method enables change tracking for this table, and returns the prior state of change tracking.

Please note that the "tracking" state of tables in the TimesTen database is persistent. Even if a power failure or other system calamity causes the data store to be reloaded into RAM from disk, the XLA-enabled state of a table will be retained.

`void disableTracking (bool* oldValue, TTStatus&)`

Calling this method disables the tracking of changes to this table using XLA. After this method is called, changes to this table are not available through XLA.

This method disables change tracking for this table, and returns the prior state of change tracking.

Please note that the "tracking" state of tables in the TimesTen database is persistent. Even if a power failure or other system calamity causes the data store to be reloaded into RAM from disk, the XLA-enabled state of a table will be retained.

`void setTracking (bool newValue, TTStatus&)`

Calling this method enables or disables the tracking of changes to this table using XLA. If "newValue" is "true", tracking is enabled; if "false" it is disabled.

This method is convenient to use when “enableTracking” or “disableTracking” has previously been used, and it is desired to return the tracking state of a table to its previous value.

Please note that the “tracking” state of tables in the TimesTen database is persistent. Even if a power failure or other system calamity causes the data store to be reloaded into RAM from disk, the XLA-enabled state of a table will be retained.

Usage

An XLA application that is tracking changes to one or more tables should create a TTXlaTable object for each table to be monitored, and should call the Define method to define each table. As XLA records describing changes to a table are processed these TTXlaTable objects are valuable in interpreting what has changed, as will be seen in upcoming examples.

TTXlaColumn

This object defines a single column in a table.

Public Members

None

Public Methods

`int getType()`

Returns the ODBC type of the column (such as SQL_INTEGER, SQL_CHAR, etc.)

`const char* getColName()`

Returns the name of the column.

`SQLULEN getPrecision()`

Returns the numeric precision of the column. Only meaningful for DECIMAL columns.

`int getScale()`

Returns the numeric scale of the column. Only meaningful for DECIMAL columns.

`SQLUINTEGER getSysColNum()`

Returns the “system column number” of the column. (How is this used?)

`SQLUINTEGER getUserColNum()`

Returns the “user column number” of the column as described by XLA. This number is not frequently used in XLA applications.

`SQLUINTEGER getSize()`

`SQLUINTEGER getOffset()`

`SQLUINTEGER getNullOffset()`

`SQLUINTEGER getFlags()`

These 4 methods return information about the column which is interesting when parsing XLA update notification records, but which will not normally be used by an XLA application. Using the TTXlaRowViewer::Get() methods provides a much simpler way to examine update notification records.

`void Define (ttXlaColDesc_t* colP, TTStatus& stat)`

Causes the TTXlaColumn object to be populated with catalog information about a particular column from a particular table.

Applications will normally not call this method; it is automatically invoked using TTXlaTable::Define on each column in the table.

Usage

TTXlaColumn objects and methods can be useful to applications which need to understand the schema of tables being monitored. Applications can easily access this information through the "col" array of TTXlaColumn objects maintained in the TTXlaTable object.

TTXlaRowViewer

The TTXlaRowViewer class is a powerful class that 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 TTXlaConnection::fetchUpdates method.

Public Members

None

Public Methods

`void setTable (TTXlaTable*)`

This associates this TTXlaRowViewer with a particular table.

`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 TTXlaConnection::fetchUpdates contain either zero, one or two rows.

- Structures that define a row that was inserted into a table contain one row image … the row that was actually inserted.
- Structures that define a row that was deleted from a table contain one row image … the row that was deleted.
- Structures that define a row that was updated in a table contain two row images … the "old" row and the "new" row.
- 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.

`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. An easy way to get this column number is to use the TTXlaTable::getColNumber method, which will give you the column number for a particular named column.

`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. An easy way to get this column number is to use the TTXlaTable::getColNumber method, which returns the column number for the specified column name.

The following table shows the supported SQL data types and the appropriate version(s) of Get to use for each parameter type.

Table 4: *TTXlaRowViewer::Get* variants for supported SQL data types

SQL Data Type	**Get variants supported**
SQL_TINYINT	Get (int, SQLTINYINT*)
SQL_SMALLINT	Get (int, short*)
SQL_INTEGER	Get (int, int*)
SQL_LONG	*<not currently supported>*
SQL_BIGINT	Get (int, SQLBIGINT*)
SQL_FLOAT	Get (int, float*)
SQL_DOUBLE	Get (int, double*)
SQL_DECIMAL	Get (int, char**)
SQL_NUMERIC	Get (int, char**)
SQL_CHAR	Get (int, char**)
SQL_VARCHAR	Get (int, char**)
SQL_WCHAR	Get (int, SQLWCHAR**, int * len)
SQL_WVARCHAR	Get (int, SQLWCHAR**, int * len)
SQL_BINARY	Get (int, const void**, int * len)
SQL_VARBINARY	Get (int, const void**, int * len)
SQL_DATETIME	Get (int, TIMESTAMP_STRUCT*)
SQL_TIMESTAMP	Get (int, TIMESTAMP_STRUCT*)
SQL_DATE	Get (int, DATE_STRUCT*)
SQL_TIME	Get (int, TIME_STRUCT*)

Usage

This class is the “workhorse” of the XLA classes. 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.

Public Members

None

Protected Members

`TTXlaTable tbl;`

The TTXlaTable object associated with the table being handled.

`TTXlaRowViewer row;`

Within the HandleChange, HandleDelete, HandleInsert and HandleUpdate methods (described below), this RowViewer object can be used to view the row being inserted or deleted, or the “old” image of the row being changed.

`TTXlaRowViewer row2;`

Within the HandleChange, HandleDelete, HandleInsert and HandleUpdate methods (described below), this RowViewer object can be used to view the “new” image of the row being updated.

Public Methods

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

This associates this TTXlaRowViewer with a particular table. It initializes the TTXlaTable object contained within this object.

`virtual void EnableTracking (TTStatus&);`

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

`virtual void DisableTracking (TTStatus&);`

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

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

This method can be used to “dispatch” 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 the section “Acknowledging XLA updates at transaction boundaries” on page 55 for information about how to use the pData parameter.

`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”.

`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”.

`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 “old” version of the row that was updated from the table is available through the RowViewer named “row”; the “new” version of the row is available row “row2”.

`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 parameter “actualLen” returns information about the actual length of the SQL string returned.

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.

The TTClasses demo program < `demo/ttclasses/xlademo.cpp` > shows how to use this method.

Usage

It is expected that application developers will derive one or more classes from TTXlaTableHandler, and will put most if not all of their application's logic in the HandleInsert, HandleDelete, and HandleUpdate methods of that class.

One strategy would be to derive multiple classes from this one, one per 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 would be to derive one or more "generic" classes from this one to handle various scenarios. For example, a generic class derived from this one 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.

Public Members

None

Public Methods

`(constructor) TTXlaTableList (TTXlaConnection* cP);`

Used to create a TableList. The cP parameter references the database connection to be used for XLA operations.

`void add (TTXlaTableHandler* h);`

Used to add a TableHandler to the list.

`void del (TTXlaTableHandler* h);`

Used to delete a TableHandler from the list.

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

When a ttXlaUpdateDesc_t is received from XLA, it can be processed by calling this method. This method will determine which table the record references, and will in turn call 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:

1

Suggested Usage

This chapter contains brief descriptions of the recommended way to use TTClasses; however, programmers will likely find the best way to learn how to use TTClasses is to examine the sample programs that are included with TTClasses.

Core TTClasses (TTCmd, TTConnection, TTStatus) Suggested Usage

While TTClasses can be used in a number of ways, the following general approach has been successful in numerous projects, and can easily be adapted to a variety of applications.

In order to achieve optimal performance, real time applications should use prepared SQL statements exclusively. Ideally all SQL statements that will be used by an application are prepared when the application begins, using separate `TTCmd` objects for each statement. Since in ODBC (and thus in the C++ classes) statements are bound to a particular connection, it follows that a full set of all statements that will be 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 might create a class called “XYZConnection”, derived from `TTConnection`. The XYZConnection class would contain private `TTCmd` members representing the prepared SQL statements that can be used in the application. Additionally, the XYZConnection class would provide new public methods to implement the application-specific database functionality, which would be implemented using the private `TTCmd` members.

As an example, consider the following class.

Example 1.1 Example class that inherits from TTConnection

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

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

In this example, an XYZConnection object is a connection to TimesTen that can be used to perform three application-specific operations: addUser, updateUser and queryUser. These operations are specific to the application (storing account balances, for example). The implementation of these three methods would presumably use the updateData, insertData and queryData `TTCmd` objects provided in the class to implement the specific functionality of the application.

In order 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 actually establish the database connection), but will also call the Prepare method for each `TTCmd` object to cause the SQL statements to be prepared for later use.

Example:

Example 1.2 class 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 then add them to a `TTConnectionPool`. By calling `TTConnectionPool::ConnectAll()`, the application can then 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 (or your application's version, of course!) 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 best and simplest example is probably < `demo/ttclasses/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.

Change Notification (XLA) Suggested Usage

See the < `demo/ttclasses/changemon.cpp`> demo program, shipped with TimesTen, as an example of how to monitor changes to a table. The slightly more complicated < `demo/ttclasses/changemon_multi.cpp`> demonstrates how to monitor multiple tables.

Another useful XLA demo program worth looking at is < `demo/ttclasses/xlademo.cpp` >.

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 describes how to acknowledge updates only at transaction boundaries, which is a common requirement for XLA applications.

The typical "main loop" of an TTClasses/XLA program looks like this:

Example 1.3 Typical TTClasses/XLA "main loop"

```
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
  // could also be conn.fetchUpdates(...);
  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, stat);
  } // 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/delete, one of < `HandleInsert(), HandleUpdate(), HandleDelete()` > is called.

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 above 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 will not be an issue. This is because, when you ask XLA to return "at most 1000 records", in the `fetchUpdates()` or `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 pretty quickly, just a few at a time. And when you are pulling these XLA records out a few at a time, TimesTen XLA usually makes sure that the last record returned is on a transaction boundary.

In short: 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. However, this is not 100% guaranteed.

XLA guarantees that *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 is reached.

Careful XLA applications need to verify whether the last record in a batch of XLA records has a transaction boundary; and only call ackUpdates() on this transaction boundary.

The case where this is always going to be an issue is handling large-row operations. If a bulk insert/update/delete operation has been performed on the database, when the XLA application asks for 1000 records, it might receive all 1000 records (or fewer than 1000), and 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. To repeat: careful XLA applications need to check for the transaction boundary, and only call `ackUpdates()` when it has been reached.

Of course, calling `ackUpdates()` for every single transaction boundary is a bad idea, because `ackUpdates()` is a relatively expensive (slow) operation. But careful XLA applications should make sure to only call this method on a transaction boundary, so that in the case of application or system or database failure, when the system recovers, the XLA bookmark is at the start of a transaction -- and not in the middle of a transaction

The `HandleChange()` method for this reason has a third parameter, to allow passing information between `HandleChange()` and the main XLA loop. Compare the previous loop with this one:

Example 1.4 Improved XLA "main loop", where the transaction boundary is monitored

```
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
  // could also be conn.fetchUpdates(...);
  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, stat, &do_acknowledge);
  } // end for each record fetched

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

As you can see in this code, `ackUpdates()` is only called when the do_acknowledge flag indicates that this batch of XLA records is at a transaction boundary.

In addition to this change to the XLA "main loop", the `HandleChange()` method needs to be (over-)written to make use of the function `isXLACommitRecord(ttXlaUpdateDesc_t*).`

The TTClasses demo < `demo/ttclasses/xlademo.cpp` > demonstrates rewriting the `HandleChange()` method. Please take a look at this demo program and run it to see how the XLA end-of-transaction flag can be monitored and acted upon.

Index