CLI/ODBC

CLI is a software development standard defined in ISO/IEC 9075-3: 2003.

The CLI defines functions and specifications for how to pass SQL to the database and how to receive and analyze the results. This CLI was developed in the early 1990s and was developed exclusively for C and COBOL languages, and its specifications have been maintained to date.

The most widely known standard interface to date is ODBC (Open Database Connectivity), which provides a way for a client program to access a database regardless of the type of database. The current ODBC API version is 3.52 and is defined in ISO and X/Open standards.

Standard CLI Functions

See the following links for usage of the standard functions.

- Wikipedia
- Open Group Document

You can refer to the following function.

SQLAllocConnect	SQLDisconnect	SQLGetDescField	SQLPrepare
SQLAllocEnv	SQLDriverConnect	SQLGetDescRec	SQLPrimaryKeys
SQLAllocHandle	SQLExecDirect	SQLGetDiagRec	SQLStatistics
SQLAllocStmt	SQLExecute	SQLGetEnvAttr	SQLRowCount
SQLBindCol	SQLFetch	SQLGetFunctions	SQLSetConnectAttr
SQLBindParameter+	SQLFreeConnect	SQLGetInfo	SQLSetDescField
SQLColAttribute	SQLFreeEnv	SQLGetStmtAttr	SQLSetDescRec
SQLColumns	SQLFreeHandle	SQLGetTypeInfo	SQLSetEnvAttr
SQLConnect	SQLFreeStmt	SQLNativeSQL	SQLSetStmtAttr
SQLCopyDesc	SQLGetConnectAttr	SQLNumParams	SQLStatistics
SQLDescribeCol	SQLGetData	SQLNumResultCols	SQLTables

Connection String for Connecting

To connect through the CLI, you need to create a connection string. The contents of each are as follows.

Connection String Item Name	Item Description
DSN	Specifies the data source name. ODBC specifies the section name of the file containing the resource, and CLI specifies the server name or IP address.
DBNAME	Describes the DB name of Machbase.
SERVER	Indicates the host name or IP address of the server where Machbase is located.
NLS_USE	Sets the language type to use with each other (currently unused, kept for future expansion).
UID	User ID
PWD	User password
PORT_NO	Port number to connect to
PORT_DIR	Specifies the file path to use when connecting to a Unix domain from Unix. (It is specified when modified from the server, and it works even if it is not specified by default.)
CONNTYPE	Specifies the connection method between the client and the server. 1: Connection with TCP / IP INET 2: Connect to Unix Domain

COMPRESS	Indicates whether to compress the Append protocol. If this value is 0, it is transmitted without compression. If this value is any value greater than 0, it is compressed only if the Append record is larger than its value. Ex) COMPRESS = 512 Only when the record size is larger than 512, it is compressed and operates. For remote connection, compression improves transmission performance.
SHOW_HIDDEN_COLS	Decides whether to show the hidden column (_arrival_time) when executing it with select *. If it is 0, it is not shown. If it is 1, information of the corresponding column is output.
CONNECTION_TIMEOUT	Sets how long to wait on the first connection. The default setting is 30 seconds. This value is set higher if the server response on the first connection is slower than 30 seconds.

An example of CLI connection is as follows.

```
sprintf(connStr,"SERVER=127.0.0.1;COMPRESS=512;UID=SYS;PWD=MANAGER;CONNTYPE=1;PORT_NO=%d", MACHBASE_PORT_NO);

if (SQL_ERROR == SQLDriverConnect( gCon, NULL, (SQLCHAR *)connStr, SQL_NTS, NULL, 0, NULL,
SQL_DRIVER_NOPROMPT )) {
   ...
}
```

Extension CLI Function (APPEND)

The CLI extension function is a function for implementing the Append protocol provided to input data to the Machbase server at high speed.

This function consists of four functions: channel open, channel data input, channel flush, and channel closing.

Understanding Append Protocol

The Append protocol provided by Machbase works asynchronously. The term asynchronous means that the response to a specific job requested by the client to the server does not completely synchronize with each other but occurs at the moment when an arbitrary event occurs. That is, even if a client has performed an append, you can not immediately get or verify the results of that execution, and you can check it at any time when the server is ready. For this reason, developers who develop applications using the Append protocol should have an understanding of the following internal behaviors. The following discussion is about how and when a client detects asynchronous errors that occur in the server.

Append Data Transfer

In a typical call such as SQLExecute or SQLExecDirect (), Machbase uses a synchronous scheme that returns the results back to the client immediately. However, SQLAppendDataV2 () does not send a request immediately after user data is entered. Instead, it waits until all of the client communication buffers are full, and then it sends the data to the client all at once. The reason for this design is that the input data of the client using Append assumes tens to hundreds of thousands of records per second, so it utilizes the buffering method for high-speed data transmission. For this reason, if the user wants to transmit the contents of the buffer at will, the user can input data explicitly by calling SQLAppendFlush () function.

Append Data Error Check

As mentioned earlier, the Append protocol is buffered and operates asynchronously. In particular, it is very important to understand when and how an error is detected because it takes a method to detect an error only when an error occurs, without receiving any response when an error does not occur in the server. In addition, since the cost of detecting an error is relatively large, it is very inefficient to check each time a record is input, and currently Machbase is designed to detect an error only in the following cases explicitly. When an error is detected, the error callback function set by the user is called every time.

1. Checks after all the transmit buffers are full and the data has been explicitly sent to the server,
2. Checks after explicitly sending data to the server from within SQLAppendFlush ()
3. Checks just before shut down from within SQLAppendClose ()

In other words, it is basically designed to detect errors only in the above three cases, and is designed to minimize the occurrence of I/O.

Additional Options for Checking Server Errors

In order to achieve the maximum performance, the default error detection technique can be more frequently checked and utilized by the user if desired. This can be done by adjusting the last argument to the SQLAppendOpen () function, aErrorCheckCount. When this value is 0, it does not perform any checking operation and operates basically. However, if this value is greater than 0, SQLAppendData () is explicitly checked for errors every time it is called. In other words, if this value is 10, you pay the cost of checking for errors every 10 appends. Therefore, when this value is small, system resources for error detection are used much, so it should be adjusted to an appropriate number.

Leaving Trace Log When Server Error Occurs

If you want to leave a trace log for the append data where an error occurs, set the prepared property DUMP_APPEND_ERROR to 1 on the server. Wit h this setting, the specification of the record that generated the error in the mach.trc file is written to the file. However, if the number of errors is excessive, the amount of system resources used will increase drastically, which may degrade the overall performance of Machbase.

APPEND Function Description

SQLAppendOpen

```
SQLRETURN SQLAppendOpen(SQLHSTMT   aStatementHandle,
                        SQLCHAR   *aTableName,
                        SQLINTEGER aErrorCheckCount );
```

This function opens a channel for the target table. If this channel is not closed afterwards, it is kept open continuously.
A maximum of 1024 statements can be set for one connection. You can use SQLAppendOpen for each statement.

1. aStatementHandle: Represents the handle of the Statement to be appended.
2. aTableName: Indicates the name of the table to which Append will be performed.
3. aErrorCheckCount: Decides whether to check the server for errors whenever several data are input. If this value is 0, no error is checked arbitrarily.

SQLAppendData (deprecated)

```
SQLRETURN  SQLAppendData(SQLHSTMT StatementHandle, void *aData[]);
```

This function is a function that inputs data for the channel.

- aData is an array containing pointers to the data to be input. The number of arrays must match the number of columns held by the table specified at Open.
- The return value can be SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, or SQL_ERROR.
  In particular, if SQL_SUCCESS_WITH_INFO is returned, there may be errors such as a lengthy input column being truncated, so check the result again.

Configuration According to Data Type

Numeric and character types

- Types such as float, double, short, int, long long, and char * work well with pointers to their values.

Address type

- In the case of ipv4, it is passed as an array of 5-byte unsigned char.
- The first byte is set to 4, the next 4 bytes are set to consecutive address values.
- For example, in the case of 127.0.0.1, five byte arrays **0x04, 0x7f, 0x00, 0x00, and 0x01** are entered in order.

```
// For tables with four column information (short (16), int (32), long (64), varchar)

testAppendIPFunc()
{
   short val1 = 0;
   int   val2 = 1;
   long long  val3 = 2;
   char *val4 = "my string";
   void *valueArray[4];

   valueArray[0] = (void *)&val1;
   valueArray[1] = (void *)&val2;
   valueArray[2] = (void *)&val3;
   valueArray[3] = (void *)val4;

   SQLAppendData(aStmt, valueArray);
}
```

Configuration According to Data Type

datetime type

- Since Machbase internally has a nano-unit time resolution value, it must be converted when setting the time on the client, and it is expressed as a 64-bit unsigned integer value.
  Therefore, for proper conversion, you need to add nano values after converting to seconds using the UNIX library mktime.
  Machbase time = (total time (seconds) since January 1, 1970) * 1,000,000,000 + milli-second * 1,000,000 + micro-second * 1000 + nano-second;

```
// Code if Date String is entered as "Year - Month - Date: Minute: Second Millis: Micro: Nano"

testAppendDateStrFunc(char *aDateString)
{
    int yy, int mm, int dd, int hh, int mi, int ss;
    unsigned long t1;
    void *valueArray[5];
    sscanf(aDateString, "%d-%d-%d %d:%d:%d %d:%d:%d",
        &yy, &mm, &dd, &hh, &mi, &ss, &mmm, &uuu, &nnn);
    sTm.tm_year = yy - 1900;
    sTm.tm_mon = mm - 1;
    sTm.tm_mday = dd;
    sTm.tm_hour = hh;
    sTm.tm_min = mi;
    sTm.tm_sec = ss;
    t1 = mktime(&sTm);
    t1 = t1 * 1000000000L;
    t1 = t1 + (mmm*1000000L) + (uuu*1000) + nnn;

    valueArray[4] = &t1;
    SQLAppendData(aStmt, valueArray);
}
```

SQLAppendDataByTime(deprecated)

```
SQLRETURN  SQLAppendDataByTime(SQLHSTMT StatementHandle, SQLBIGINT aTime, void *aData[]);
```

This function is a function to input data for the corresponding channel, and the value of _arrival_time stored in the DB can be set to a specific time value instead of the current time.
For example, you want to enter the date in the log file a month ago as the date.

- aTime is a time value set to _arrival_time.
- aData is an array containing pointers to the data to be input.
- The number of arrays must match the number of columns held by the table specified at Open.

For the rest, refer to the SQLAppendData () function.

```
// For tables with four column information (short (16), int (32), long (64), varchar)

testAppendFuncWithTime()
{
   long long sTime = 1;
   short val1 = 0;
   int   val2 = 1;
   long long  val3 = 2;
   char *val4 = "my string";
   void *valueArray[4];

   valueArray[0] = (void *)&val1;
   valueArray[1] = (void *)&val2;
   valueArray[2] = (void *)&val3;
   valueArray[3] = (void *)val4;

   SQLAppendDataByTime(aStmt, sTime, valueArray);
}
```

SQLAppendDataV2

```
SQLRETURN  SQLAppendDataV2(SQLHSTMT StatementHandle, SQL_APPEND_PARAM *aData);
```

This function is a newly introduced Append function since Machbase 2.0. It is a convenient function that improves the input method inconvenient in existing functions.
In the case of TEXT and BINARY type introduced in 2.0 especially, input is possible only in SQLAppendDataV2 () function.

- Can input NULL for each type
- Can input string length when inputting VARCHAR
- Can input binary and string data when inputting IPv4 or IPv6
- Can specify data length for TEXT, BINARY type

The function arguments are structured as follows.

- aData is a pointer to an array of arguments called SQL_APPEND_PARAM. The number of this array must match the number of columns held by the table specified at Open.
- The return value can be SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, or SQL_ERROR. In particular, if SQL_SUCCESS_WITH_INFO is returned, there may be errors such as a lengthy input column being truncated, so check the result again.

Below is the definition of `SQL_APPEND_PARAM` that will actually be used in V2 , which is included in `machbase_sqlcli.h`.

```
typedef struct machAppendVarStruct
{
    unsigned int mLength;
    void *mData;
} machAppendVarStruct;

/* for IPv4, IPv6 as bin or string representation */
typedef struct machAppendIPSttypedef struct machbaseAppendVarStruct
{
    unsigned int mLength;
    void        *mData;
} machbaseAppendVarStruct;

/* for IPv4, IPv6 as bin or string representation */
typedef struct machbaseAppendIPStruct
{
    unsigned char   mLength; /* 0:null, 4:ipv4, 6:ipv6, 255:string representation */
    unsigned char   mAddr[16];
    char           *mAddrString;
} machbaseAppendIPStruct;

/* Date time*/
typedef struct machbaseAppendDateTimeStruct
{
    long long       mTime;
#if defined(SUPPORT_STRUCT_TM)
    struct tm       mTM;
#endif
    char           *mDateStr;
    char           *mFormatStr;
} machbaseAppendDateTimeStruct;

typedef union machbaseAppendParam
{
    short                       mShort;
    unsigned short              mUShort;
    int                         mInteger;
    unsigned int                mUInteger;
    long long                   mLong;
    unsigned long long          mULong;
    float                       mFloat;
    double                      mDouble;
    machbaseAppendIPStruct      mIP;
    machbaseAppendVarStruct     mVar;     /* for all varying type */
    machbaseAppendVarStruct     mVarchar; /* alias */
    machbaseAppendVarStruct     mText;    /* alias */
    machbaseAppendVarStruct     mBinary;  /* binary */
    machbaseAppendVarStruct     mBlob;    /* reserved alias */
    machbaseAppendVarStruct     mClob;    /* reserved alias */
    machbaseAppendDateTimeStruct mDateTime;
} machbaseAppendParam;

#define SQL_APPEND_PARAM machbaseAppendParam
```

As you can see from the above, there is a structure in which a shared structure `machbaseAppendParam` which internally contains one argument. The length and value for the data and string can be explicitly entered for each data type. Examples of actual use are as follows.

Fixed-Length Numeric Type Input

Fixed-length numeric types are short, ushort, integer, uinteger, long, ulong, float, and double. This type can be entered by directly assigning a value to the structure member of SQL_APPEND_PARAM.

Database Type	NULL Macro	SQL_APPEND_PARAM Member
SHORT	SQL_APPEND_SHORT_NULL	mShort
USHORT	SQL_APPEND_USHORT_NULL	mUShort

INTEGER	SQL_APPEND_INTEGER_NULL	mInteger
UINTEGER	SQL_APPEND_UINTEGER_NULL	mUInteger
LONG	SQL_APPEND_LONG_NULL	mLong
ULONG	SQL_APPEND_ULONG_NULL	mULong
FLOAT	SQL_APPEND_FLOAT_NULL	mFloat
DOUBLE	SQL_APPEND_DOUBLE_NULL	mDouble

The following is an example of entering actual values.

```
// Assume that the Table Schema consists of eight columns, SHORT, USHORT, INTEGER, UINTEGER, LONG, ULONG,
FLOAT, and DOUBLE, respectively.

void testAppendExampleFunc()
{
    SQL_APPEND_PARAM sParam[8];

    /* fixed column */
    sParam[0].mShort = SQL_APPEND_SHORT_NULL;
    sParam[1].mUShort = SQL_APPEND_USHORT_NULL;
    sParam[2].mInteger = SQL_APPEND_INTEGER_NULL;
    sParam[3].mUInteger = SQL_APPEND_UINTEGER_NULL;
    sParam[4].mLong = SQL_APPEND_LONG_NULL;
    sParam[5].mULong = SQL_APPEND_ULONG_NULL;
    sParam[6].mFloat = SQL_APPEND_FLOAT_NULL;
    sParam[7].mDouble = SQL_APPEND_DOUBLE_NULL;

    SQLAppendDataV2(Stmt, sParam);

    /* FIXED COLUMN Value */
    sParam[0].mShort = 2;
    sParam[1].mUShort = 3;
    sParam[2].mInteger = 4;
    sParam[3].mUInteger = 5;
    sParam[4].mLong = 6;
    sParam[5].mULong = 7;
    sParam[6].mFloat = 8.4;
    sParam[7].mDouble = 10.9;

    SQLAppendDataV2(Stmt, sParam);
}
```

Date Type Input

Below is an example of inputting data of DATETIME type. Several macros are available for convenience.

Performs operations on the mDateTime member in SQL_APPEND_PARAM. The following macro can specify a date by setting a 64-bit integer value called mTime in the mDateTime structure.

```
typedef struct machbaseAppendDateTimeStruct
{
    long long       mTime;
#if defined(SUPPORT_STRUCT_TM)
    struct tm       mTM;
#endif
    char           *mDateStr;
    char           *mFormatStr;
} machbaseAppendDateTimeStruct;
```

Macro	Description

SQL_APPEND_DATETIME_NOW	Enters the current client time.
SQL_APPEND_DATETIME_STRUCT_TM	Sets a value to mTM, the struct tm structure of mDateTime, and inputs the value to the database.
SQL_APPEND_DATETIME_STRING	Sets a value for the string type of mDateTime and enters it into the database. mDateStr: real date string value assigned mFormatStr: format string assignment for date string
SQL_APPEND_DATETIME_NULL	Enters the value of the date column as NULL.
Any 64-bit Value	This value is entered as the actual datetime. This value represents an integer value in nanoseconds since January 1, 1970. For example, if this value is 1 billion (1,000,000,000), it represents 0: 1: 1 on January 1, 1970. (GMT)

```
// Assume that the table schema consists of eight columns, SHORT, USHORT, INTEGER, UINTEGER, LONG, ULONG,
FLOAT, and DOUBLE, respectively.

void testAppendDateTimeFunc()
{
    SQL_mach_PARAM sParam[1];
    /* NULL Insert */
    sParam[0].mDateTime.mTime   = SQL_APPEND_DATETIME_NULL;
    SQLAppendDataV2(Stmt, sParam);

    /* Current Time  */
    sParam[0].mDateTime.mTime      = SQL_APPEND_DATETIME_NOW;
    SQLAppendDataV2(Stmt, sParam);

    /* nano second since 1970/01/01 */
    sParam[0].mDateTime.mTime      = 1234;
    SQLAppendDataV2(Stmt, sParam);

    /*  String format time  */
    sParam[0].mDateTime.mTime      = SQL_APPEND_DATETIME_STRING;
    sParam[0].mDateTime.mDateStr   = "23/May/2014:17:41:28";
    sParam[0].mDateTime.mFormatStr = "DD/MON/YYYY:HH24:MI:SS";
    SQLAppendDataV2(Stmt, sParam);

    /*  struct tm based time  */
    sParam[0].mDateTime.mTime      = SQL_APPEND_DATETIME_STRUCT_TM;
    sParam[0].mDateTime.mTM.tm_year = 2000 - 1900;
    sParam[0].mDateTime.mTM.tm_mon  =  11;
    sParam[0].mDateTime.mTM.tm_mday  = 31;
    SQLAppendDataV2(Stmt, sParam);
}
```

Internet Address Type Input

The following is an example of inputting IPv4 and IPv6 type data. There are also several macros available for your convenience. Performs operations on the mLength member in SQL_APPEND_PARAM.

```
/* for IPv4, IPv6 as bin or string representation */
typedef struct machbaseAppendIPStruct
{
    unsigned char   mLength; /* 0:null, 4:ipv4, 6:ipv6, 255:string representation */
    unsigned char   mAddr[16];
    char           *mAddrString;
} machbaseAppendIPStruct;
```

Macro (Set to mLength)	Description
SQL_APPEND_IP_NULL	Enters a NULL value in the corresponding column

SQL_APPEND_IP_IPV4	mAddr has IPv4
SQL_APPEND_IP_IPV6	mAddr has IPv6
SQL_APPEND_IP_STRING	mAddrString has an address string.

The following is an example of entering actual values for each case.

```
void testAppendIPFunc()
{
    SQL_APPEND_PARAM sParam[1];
    /* NULL */
    sParam[0].mIP.mLength  = SQL_APPEND_IP_NULL;
    SQLAppendDataV2(Stmt, sParam);

    /* Direct array access */
    sParam[0].mIP.mLength  = SQL_APPEND_IP_IPV4;
    sParam[0].mIP.mAddr[0] = 127;
    sParam[0].mIP.mAddr[1] = 0;
    sParam[0].mIP.mAddr[2] = 0;
    sParam[0].mIP.mAddr[3] = 1;
    SQLAppendDataV2(Stmt, sParam);

    /* IPv4 from binary */
    sParam[0].mIP.mLength  = SQL_APPEND_IP_IPV4;
    *(in_addr_t *)(sParam[0].mIP.mAddr) = inet_addr("192.168.0.1");
    SQLAppendDataV2(Stmt, sParam);

    /* IPv4 : ipv4 from string */
    sParam[0].mIP.mLength     = SQL_APPEND_IP_STRING;
    sParam[0].mIP.mAddrString = "203.212.222.111";
    SQLAppendDataV2(Stmt, sParam);

    /* IPv4 : ipv4 from invalid string */
    sParam[0].mIP.mLength     = SQL_APPEND_IP_STRING;
    sParam[0].mIP.mAddrString = "ip address is not valid";
    SQLAppendDataV2(Stmt, sParam);                          // invalid IP value

    /* IPv6 : ipv6 from binary bytes */
    sParam[0].mIP.mLength  = SQL_APPEND_IP_IPV6;
    sParam[0].mIP.mAddr[0]  = 127;
    sParam[0].mIP.mAddr[1]  = 127;
    sParam[0].mIP.mAddr[2]  = 127;
    sParam[0].mIP.mAddr[3]  = 127;
    sParam[0].mIP.mAddr[4]  = 127;
    sParam[0].mIP.mAddr[5]  = 127;
    sParam[0].mIP.mAddr[6]  = 127;
    sParam[0].mIP.mAddr[7]  = 127;
    sParam[0].mIP.mAddr[8]  = 127;
    sParam[0].mIP.mAddr[9]  = 127;
    sParam[0].mIP.mAddr[10] = 127;
    sParam[0].mIP.mAddr[11] = 127;
    sParam[0].mIP.mAddr[12] = 127;
    sParam[0].mIP.mAddr[13] = 127;
    sParam[0].mIP.mAddr[14] = 127;
    sParam[0].mIP.mAddr[15] = 127;
    SQLAppendDataV2(Stmt, sParam);

    sParam[0].mIP.mLength     = SQL_APPEND_IP_STRING;
    sParam[0].mIP.mAddrString = "::127.0.0.1";
    SQLAppendDataV2(Stmt, sParam);

    sParam[0].mIP.mLength     = SQL_APPEND_IP_STRING;
    sParam[0].mIP.mAddrString = "FFFF:FFFF:1111:2222:3333:4444:7733:2123";
    SQLAppendDataV2(Stmt, sParam);
}
```

Variable Data Types (Character and Binary Data) Input

Variable data types include VARCHAR and TEXT, and BLOB and CLOB. In existing functions, only VARCHAR was supported, and there was no way for the user to enter the length of the string. For that reason, we had to get the length through the strlen () function each time, but from function V2, the user can directly specify the length for the variable data type. Thus, if the user knows the length in advance, data can be input more quickly. Internally, the variable data type is a structure. However, for convenience of development, members are created separately for each data type.

```
typedef struct machAppendVarStruct
{
    unsigned int mLength;
    void *mData;
} machAppendVarStruct;
```

When inputting a variable data type, set the length of the data to mLength and set the primitive data pointer to mData. If mLength is greater than the defined schema, it is automatically truncated. At this time, SQLAppendDataV2 () returns **SQL_SUCCESS_WITH_INFO** and also fills the internal structure with a related warning message. To see this warning message, use SQLError () function.

Database Type	NULL Macro	SQL_APPEND_PARAM Member (mVar is acceptable)
VARCHAR	SQL_APPEND_VARCHAR_NULL	mVarchar
TEXT	SQL_APPEND_TEXT_NULL	mText
BINARY	SQL_APPEND_BINARY_NULL	mBinary
BLOB	SQL_APPEND_BLOB_NULL	mBlob
CLOB	SQL_APPEND_CLOB_NULL	mClob

The following is an example of entering actual values for each environment. Assumes that there is one VARCHAR column.

```
CREATE TABLE ttt (name VARCHAR(10));
```

```
void testAppendVarcharFunc()
{
    SQL_mach_PARAM sParam[1];

    /*  VARCHAR : NULL */
    sParam[0].mVarchar.mLength = SQL_APPEND_VARCHAR_NULL
    SQLAppendDataV2(Stmt, sParam); /* OK */

    /*  VARCHAR : string */
    strcpy(sVarchar, "MY VARCHAR");
    sParam[0].mVarchar.mLength = strlen(sVarchar);
    sParam[0].mVarchar.mData   = sVarchar;
    SQLAppendDataV2(Stmt, sParam); /* OK */

    /*  VARCHAR : Truncation! */
    strcpy(sVarchar, "MY VARCHAR9"); /* Truncation! */
    sParam[0].mVarchar.mLength = strlen(sVarchar);
    sParam[0].mVarchar.mData   = sVarchar;
    SQLAppendDataV2(Stmt, sParam);  /* SQL_SUCCESS_WITH_INFO */
}
```

The following is an example of input for the Text type.

```
CREATE TABLE ttt (doc TEXT);
```

```
void testAppendFunc()
{
    SQL_mach_PARAM sParam[1];

    /*  VARCHAR : NULL */
    sParam[0].mText.mLength = SQL_APPEND_TEXT_NULL
    SQLAppendDataV2(Stmt, sParam); /* OK */

    /*  VARCHAR : string */
    strcpy(sText, "This is the sample document for tutorial.");
    sParam[0].mVar.mLength = strlen(sText);
    sParam[0].mVar.mData   = sText;
    SQLAppendDataV2(Stmt, sParam); /* OK */
}
```

SQLAppendDataByTimeV2

```
SQLRETURN  SQLAppendDataByTimeV2(SQLHSTMT StatementHandle, SQLBIGINT aTime, SQL_APPEND_PARAM  *aData);
```

This function is a function to input data for the corresponding channel, and the value of _arrival_time stored in the DB can be set to a specific time value instead of the current time. For example, you want to enter the date in the log file a month ago as the date.

- aTime is the time value to be set to _arrival_time. *You must enter the nano second value from January 1, 1970 to the present. Also, input values must be sorted in order from the past to the present.*
- aData is an array containing pointers to the data to be input. The number of arrays must match the number of columns held by the table specified at Open.

For the rest, refer to the SQLAppendDataV2 () function.

SQLAppendFlush

```
SQLRETURN SQLAppendFlush(SQLHSTMT StatementHandle);
```

This function immediately sends the data accumulated in the current channel buffer to the Machbase server.

SQLAppendClose

```
SQLRETURN SQLAppendClose(SQLHSTMT   aStmtHandle,
                         SQLBIGINT* aSuccessCount,
                         SQLBIGINT* aFailureCount);
```

This function closes the currently open channel. If an unopened channel exists, an error occurs.

- aSuccessCount: The number of successful Append records.
- aFailureCount: The number of failed Append records.

SQLAppendSetErrorCallback

```
SQLRETURN SQLAppendSetErrorCallback(SQLHSTMT aStmtHandle, SQLAppendErrorCallback aFunc);
```

This function sets the callback function that is called when an error occurs during append. If you do not set this function, the client will ignore any errors that occur in the server.

- aStmtHandle: Specifies a Statement to check for errors.
- aFunc: Specifies the function pointer to call on Append failure.

The prototype for SQLAppendErrorCallback is:

```
typedef void (*SQLAppendErrorCallback)(SQLHSTMT aStmtHandle,
                                       SQLINTEGER aErrorCode,
                                       SQLPOINTER aErrorMessage,
                                           SQLLEN aErrorBufLen,
                                       SQLPOINTER aRowBuf,
                                           SQLLEN aRowBufLen);
```

- aStatementHandle: the statement handle that generated the error
- aErrorCode: 32-bit error code that caused the error
- aErrorMessage: string for the error code
- aErrorBufLen: the length of aErrorMessage
- aRowBuf: a string containing the detailed description of the record that caused the error
- aRowBufLen: length of aRowBuf

Example of Using Error Callback (dumpError)

```
void dumpError(SQLHSTMT    aStmtHandle,
               SQLINTEGER  aErrorCode,
               SQLPOINTER  aErrorMessage,
               SQLLEN      aErrorBufLen,
               SQLPOINTER  aRowBuf,
               SQLLEN      aRowBufLen)
{
    char       sErrMsg[1024] = {0, };
    char       sRowMsg[32 * 1024] = {0, };

    if (aErrorMessage != NULL)
    {
        strncpy(sErrMsg, (char *)aErrorMessage, aErrorBufLen);
    }

    if (aRowBuf != NULL)
    {
        strncpy(sRowMsg, (char *)aRowBuf, aRowBufLen);
    }

    fprintf(stdout, "Append Error : [%d][%s]\n[%s]\n\n", aErrorCode, sErrMsg, sRowMsg);
}


......

    if( SQLAppendOpen(m_IStmt, TableName, aErrorCheckCount) != SQL_SUCCESS )
    {
        fprintf(stdout, "SQLAppendOpen error\n");
        exit(-1);
    }
    // Set callback.
    assert(SQLAppendSetErrorCallback(m_IStmt, dumpError) == SQL_SUCCESS);

    doAppend(sMaxAppend);

    if( SQLAppendClose(m_IStmt, &sSuccessCount, &sFailureCount) != SQL_SUCCESS )
    {
        fprintf(stdout, "SQLAppendClose error\n");
        exit(-1);
    }
}
```

SQLSetConnectAppendFlush

```
SQLRETURN SQL_API SQLSetConnectAppendFlush(SQLHDBC hdbc, SQLINTEGER option)
```

The data input by Append is written to the communication buffer and is sent to the server when the user calls the SQLAppendFlush function in the waiting state or the communication buffer becomes full. You can use this function if you want the user to send data by append to the server at regular intervals even if the buffer is not full. This function computes the difference between the last transmitted time and the current time every 100ms, and transfers the contents of the communication buffer to the server when the specified time (1 second if not set) has passed.

The parameters are:

- hdbc: DB connection handle.
- If option: 0, auto flush is off; otherwise, auto flush is on.

Executing on an unconnected hdbc will result in an error.

SQLSetStmtAppendInterval

```
SQLRETURN SQL_API SQLSetStmtAppendInterval(SQLHSTMT hstmt, SQLINTEGER fValue)
```

Uses SQLSetConnectAppendFlush to turn off automatic flushing or flushing for a particular statement when you turn on flushing on a time unit.

The parameters are:

- hstmt: This is the statement handle that you want to adjust the flush interval.
- fValue: The value to which you want to adjust the flush interval. **If 0, flush is not performed and the unit is ms**. Set to a multiple of 100 since the thread that determines whether to flush every 100ms is executed. It does not automatically flush at exactly the right time. **1000 is the default value** .

Execution of this function will succeed even if time-based flush is not running.

Error Check and Description

This is a description of the code and how to check for errors when using the Append related functions. If the return value in the CLI function is not SQL_SUCCESS, you can check the error message using the following code.

```
    SQLINTEGER errNo;
    int msgLength;
    char sqlState[6];
    char errMsg[1024];

    if (SQL_SUCCESS == SQLError ( env, con, stmt, (SQLCHAR *)sqlState, &errNo,
                                  (SQLCHAR *)errMsg, 1024, &msgLength ))
    {
        // (Specify error code value as 5-digit number.)
        printf("ERROR-%05d: %s\n", errNo, errMsg);
    }
```

The error message returned from the Append related function is as follows.

Item	Message	Description
SQLAppendOpen	statement is already opened.	Occurs when SQLAppendOpen is executed in duplicate.
	Failed to close stream protocol.	Stream protocol termination failed.
	Failed to read protocol.	A network read error occurred.
	cannot read column meta.	Invalid column meta information structure
	cannot allocate memory.	An internal buffer memory allocation error occurred.
	cannot allocate compress memory.	Compressed buffer memory allocation error occurred.
	invalid return after reading column meta.	Return value has an error.
SQLAppendData	statement is not opened.	Called AppendData without AppendOpen.
	column() truncated :	Occurs when you enter data that is larger than the size specified in the varchar type column.
	Failed to add binary.	Write error in communication buffer occurred.
SQLAppendClose	statement is not opened.	Not in AppendOpen state.

	Failed to close stream protocol.	Stream protocol termination failed.
	Failed to close buffer protocol.	Buffer protocol termination failed.
	cannot read column meta.	The column meta information structure is incorrect.
	invalid return after reading column meta.	Return value has an error.
SQLAppendFlush	statement is not opened.	Not in AppendOpen state
	Failed to close stream protocol.	A network write error occurred.
SQLSetErrorCallback	statement is not opened.	Not in AppendOpen state.
	Protocol Error (not APPEND_DATA_PROTOCOL)	Communication buffer read result is not APPEND_DATA_PROTOCOL value.
SQLAppendDataV2	Invalid date format or date string.	Occurs when the datetime type is wrong.
	statement is not opened.	Not in AppendOpen state
	column() truncated :	This occurs when you enter data that is larger than the size specified in the binary type column.
	column() truncated :	Occurs when you enter data that is larger than the size specified in the varchar and text type column.
	Failed to add stream.	Write error in communication buffer occurred.
	IP address length is invalid.	The mLength value of the IPv4, IPv6 type structure is specified incorrectly.
	IP string is invalid.	Not in IPv4 or IPv6 format.
	Unknown data type has been specified.	Not the data type used by Machbase.

Column wise parameter binding

The SQLAppend function, which is used to enter a large amount of data into Machbase quickly, can be used only when entering a log / tag table, and the SQLAppend function cannot be used to perform a bulk update on a lookup or volatile table.

For this purpose, Machbase 5.5 and later versions support column wise parameter binding. (Row wise format parameter binding is not yet supported.)

Set SQL_ATTR_PARAM_BIND_TYPE in the argument Attribute of the function SQLSetStmtAttr () and SQL_PARAM_BIND_BY_COLUMN in the parameter param.

For each column to bind, set the parameter to an array and the indicator variable to an array. Then call SQLBindParameter () with this parameter.

The figure below shows how columnar binding works for each parameter array.

Column A (parameter A)		**Column B (parameter B)**		**Column C (parameter C)**	
Value_Array	Indicator/ length array	Value_Array	Indicator/ length array	Value_Array	Indicator/ length array

The following example inserts a large amount of data using column wise parameter binding.

```
#define DESC_LEN 51
#define ARRAY_SIZE 10
SQLCHAR * Statement = "INSERT INTO Parts (PartID, Description, Price) VALUES (?, ?, ?)";

SQLUINTEGER PartIDArray[ARRAY_SIZE];
SQLCHAR DescArray[ARRAY_SIZE][DESC_LEN];
SQLREAL PriceArray[ARRAY_SIZE];
/* Bind variable array  */
SQLINTEGER PartIDIndArray[ARRAY_SIZE], DescLenOrIndArray[ARRAY_SIZE], PriceIndArray[ARRAY_SIZE];
SQLUSMALLINT i, ParamStatusArray[ARRAY_SIZE];
SQLUINTEGER ParamsProcessed;

// Set the SQL_ATTR_PARAM_BIND_TYPE statement attribute to use
// column-wise binding.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_BIND_TYPE, SQL_PARAM_BIND_BY_COLUMN, 0);
// Specify the number of elements in each parameter array.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, ARRAY_SIZE, 0);
// Specify an array in which to return the status of each set of
// parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR, ParamStatusArray, 0);
// Specify an SQLUINTEGER value in which to return the number of sets of
// parameters processed.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &ParamsProcessed, 0);
// Bind the parameters in column-wise fashion.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    PartIDArray, 0, PartIDIndArray);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1, 0,
    DescArray, DESC_LEN, DescLenOrIndArray);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    PriceArray, 0, PriceIndArray);
```