Table of Contents

Overview

This is user documentation for accessing the YottaDB engine from C using the Simple API. A process can both call the Simple API as well as call functions written in M and exported.

Caveat: This code does not exist yet. The user documentation is being written ahead of the code, and will change in the event the code needs to differ from this documentation.

Using libyottadb

1. Install YottaDB.
2. Include the `yottadb.h` file in your C program and compile it.
3. Perform any database configuration and initialization needed (configuring global directories, creating database files, starting a Source Server process, etc.).
4. Run your program, ensuring either that `libyottadb.so` is in the load path of your program, or that it is preloaded.

Concepts

Key-value

Local and global variables

Subscripts (keys) of variables accessed using Simple API are strings. When a string is a canonical number YottaDB internally converts and stores it as a number. When ordering (collating) subscripts:

- Null (empty string) subscripts precede all numeric subscripts.
  - **YottaDB strongly recommends against applications that use null subscripts.**
- Numeric subscripts precede string subscripts.
  - Numeric subscripts in numeric order.
- String subscripts collate in byte order.

Data Types

Data types are defined by including `yottadb.h` and are one of:

`ydb_int_t` and `ydb_uint_t` — Signed and unsigned integers, that are *at least* 16 bits.

`ydb_long_t` and `ydb_ulong_t` — Signed and unsigned integers, that are *at least* 32 bits.

`ydb_longlong_t` and `ydb_ulonglong_t` - Signed and unsigned integers that are *at least* 64 bits. See Numeric Considerations below.

`ydb_maxsub_t` — A signed integer that is able to store the maximum number of subscripts of a local or global variable, `YDB_MAX_SUB`. It is signed rather than unsigned to permit a negative value for "name level" invocations of `ydb_subscript_next_s()` and `ydb_subscript_previous_t()`.

`ydb_status_t` — A signed integer which is the return value (status) of a call to a libyottadb function.

`ydb_strlen_t` — An unsigned integer type that is able to store the maximum length of a string, `YDB_MAX_STR`.

`ydb_uchar_t` — An unsigned data value that is *exactly* 8-bits (one byte).

Symbolic Constants

The `yottadb.h` file defines several symbolic constants, which are one of the following types:

- Function Return Codes, which in turn are one of:
  - Normal Return Codes
  - Error Return Codes
- Limits
- Other

Function Return Codes

Return codes from calls to libyottadb are of type `ydb_status_t`. Normal return codes are non-negative (greater than or equal to zero); error return codes are negative.

Normal Return Codes

Symbolic constants for normal return codes are prefixed with `YDB_`.

`YDB_STATUS_OK` — Normal return following successful execution.

Error Return Codes

Symbolic constants for error codes returned by calls to libyottadb are prefixed with `YDB_ERR_`. [1] The symbolic constants below are not intended to be a complete list of all error messages that Simple API functions can return - the `ydb_message()` functions provides a way to get detailed information about a error codes for those without symbolic constants.

`YDB_ERR_GVINVALID` — A global variable name is too long. [2]

`YDB_ERR_GVUNDEF` — No value exists at a requested global variable node.

`YDB_ERR_LVUNDEF` — No value exists at a requested local variable node. [3]

`YDB_ERR_INSUFFSUBS` — A call to `ydb_node_next_s()` or `ydb_node_previous_s()` did not provide enough parameters for the return values. [4]

`YDB_ERR_INVSTRLEN` — A buffer provided by the caller is not long enough for the string to be returned, or the length of a string passed as a parameter exceeds `YDB_MAX_STR`. In the event the return code is `YDB_ERR_INVSTRLEN` and if `*xyz` is the `ydb_string_t` value which does not provide sufficient space, then `xyz->used` is set to the size required of a sufficiently large buffer, and `xyz->address` points to the first `xyz->alloc` bytes of the value. In this case the `used` field of the `ydb_string_t` structure is greater than the `alloc` field.

`YDB_ERR_KEY2BIG` — The length of a global variable name and subscripts exceeds the limit configured for a database region.

`YDB_ERR_MAXNRSUBSCRIPTS` — The number of subscripts specified in the call exceeded `YDB_MAX_SUB`.

`YDB_ERR_UNKNOWN` — A call to `ydb_zmessage()` specified an invalid message code.

Limits

Symbolic constants for limits are prefixed with `YDB_MAX_`. Unless otherwise noted, symbolic constants are unsigned integers guaranteed to fit within the range of a `ydb_uint_t` type.

`YDB_MAX_IDENT` —The maximum space in bytes required to store a complete variable name, including the preceding caret for a global variable.

`YDB_MAX_MSG` — The maximum length in bytes of any message string associated with a message code. A buffer of length `YDB_MAX_MSG` bytes ensures that a call to `ydb_zmessage()` will not return a `YDB_ERR_INVSTRLEN` return code.

`YDB_MAX_STR` — The maximum length of a string (or blob) in bytes. A caller to `ydb_get()` that provides a buffer of `YDB_MAX_STR` will never get a `YDB_ERR_INVSTRLEN` error. `YDB_MAX_STR` is guaranteed to fit in a `ydb_ulong_t` type.

`YDB_MAX_SUB` — The maximum number of subscripts for a local or global variable.

Data Structures

`ydb_string_t` is a descriptor for a string [5] value, and consists of the following fields:

- `alloc` and `used` — fields of type `ydb_strlen_t` where `alloc` ≥ `used`
- `address` — pointer to a `ydb_uchar_t`, the starting address of a string

Under normal circumstances `alloc` ≥ `used`; however, this may not be the case when a function returns a `YDB_ERR_INVSTRLEN` error. See YDB_ERR_INVSTRLEN for details.

Macros

`YDB_ALLOC_STRING(x, strlit)` — Allocate a `ydb_string_t` structure and initialize it to `strlit`, returning the address of the structure. Note that while the `used` field is the size of `strlit`, the `alloc` field may be rounded up to a larger value. [6]

`YDB_FREE_STRING(x)` — Free the `ydb_string_t` structure pointed to by `x`.

Simple API

To allow the libyottadb Simple API functions to handle a variable tree whose nodes have varying numbers of subscripts, the actual number of subscripts is itself passed as a parameter.

In the definitions of functions:

- `ydb_maxsub_t count` and `ydb_maxsub_t *count` refer to an actual number subscripts,
- `ydb_string_t *varname` refers to the name of a variable, and
- `[, ydb_string_t *subscript, …]` and `ydb_string_t *subscript[, ydb_string_t *subscript]` refer to placeholders for subscripts whose actual number is defined by `count` or `*count`.

Caveat Specifying a count that exceeds the actual number of parameters passed will almost certainly result in an unpleasant bug that is difficult to troubleshoot. [7]

Function names specific to the libyottadb Simple API end in `_s`. Those common to both Simple API as well as the Comprehensive API do not.

ydb_data_s()

```
ydb_status_t ydb_data_s(ydb_uint_t *value,
        ydb_maxsub_t count,
        ydb_string_t *varname[,
        ydb_string_t *subscript, ...]);
```

In the location pointed to by `value`, `ydb_data_s()` returns the following information about the local or global variable node identified by glvn:

- 0 — There is neither a value nor a sub-tree, i.e., it is undefined.
- 1 — There is a value, but no sub-tree
- 10 — There is no value, but there is a sub-tree.
- 11 — There are both a value and a subtree.

ydb_get_s()

```
ydb_status_t ydb_get_s(ydb_string_t *value,
        ydb_maxsub_t count,
        ydb_string_t *varname[,
        ydb_string_t *subscript, ... ]);
```

In the location pointed to by `value`, `ydb_get_s()` reports the value of the value of the data at the specified node.

If there is no value at the specified global or local variable node, or if the intrinsic special variable does not exist,a non-zero return value of YDB_ERR_GVUNDEF, YDB_ERR_INVSVN, or YDB_ERR_UNDEF indicates the error.

In a database application, a global variable node can potentially be changed by another process between the time that a process calls `ydb_length()` to get the length of the data in a node and a `ydb_get()` call to get that data. If a caller cannot ensure from the application schema that the size of the buffer it provides is large enough for a string returned by `ydb_get()`, it should code in anticipation of a potential `YDB_ERR_INVSTRLEN` return code from `ydb_get()`. See also the discussion at YDB_ERR_INVSTRLEN describing the contents of `*value` when `ydb_get_s()` returns a `YDB_ERR_INVSTRLEN` return code. Similarly, since a node can always be deleted between a call such as `ydb_node_next_s()` and a call to `ydb_get-s()`, a caller of `ydb_get_s()` to access a global variable node should code in anticipation of a potential `YDB_ERR_GVUNDEF`.

ydb_kill_s()

```
ydb_status_t ydb_kill_s([ydb_maxsub_t count,
        ydb_string_t *varname[,
        ydb_string_t *subscript, ...], ...,] NULL);
```

Note: the parameter list **must** be terminated by a NULL pointer.

Kills — deletes all nodes in — each of the local or global variable trees or sub-trees specified. In the special case where the only parameter is a NULL, `ydb_kill_s()` kills all local variables.

ydb_kill_excl_s()

```
ydb_status_t ydb_kill_excl_s(ydb_string_t *varnamelist);
```

`*varnamelist` is a comma separated list of local variable names. `ydb_kill_excl_s()` kills the trees of all local variable names except those on the list.

ydb_length_s()

```
ydb_status_t ydb_length_s(ydb_strlen_t *value,
        ydb_maxsub_t count,
        ydb_string_t *varname[,
        ydb_string_t *subscript, ... ]);
```

In the location pointed to by `*value`, `ydb_length_s()` reports the length of the data in bytes. If the data is numeric, `*value` has the length of the canonical string representation of that value.

If there is no value at the requested global or local variable node, or if the intrinsic special variable does not exist,a non-zero return value of YDB_ERR_GVUNDEF, YDB_ERR_INVSVN, or YDB_ERR_UNDEF indicates the error.

ydb_node_next_s()

```
ydb_status_t ydb_node_next_s(ydb_maxsub_t *value,
        ydb_maxsub_t *count,
        ydb_string_t *varname,
        ydb_string_t *subscript[, ... ]);
```

`ydb_node_next_s()` facilitates breadth-first traversal of a local or global variable tree. Note that the parameters are both inputs to the function as well as outputs from the function, and that the number of subscripts can differ between the input node of the call and the output node reported by the call, which is the reason the number of subscripts is passed by reference.

As an input parameter `*value` specifies the number of subscripts in the input node, which does not need to exist — a value of 0 will return the first node in the tree.

Except when the `ydb_status_t` value returned by `ydb_node_next_s()` returns an error code, `*value` on the return from a call specifies the number of subscripts in the next node, which will be a node with data unless there is no next node (i.e., the input node is the last in the tree), in which case `*value` will be 0 on output.

`ydb_node_next_s()` does not change `*varname`, but does change the `*subscript` parameters.

- A `YDB_ERR_INSUFFSUBS` return code indicates an error if there are insufficient parameters to return the subscript.
- If one of the `subscript->alloc` values indicates insufficient space for an output value, the return code is the error `YDB_ERR_INVSTRLEN`. See also the discussion at YDB_ERR_INVSTRLEN describing the contents of that `*subscript` parameter. In the event of a `YDB_ERR_INVSTRLEN` error, the values in any subscripts beyond that identified by `*value` do not contain meaningful values.

Note that a call to `ydb_node_next_s()` must always have at least one subscript, since it is a *non-sequitur* to call it without subscripts and expect a return without subscripts.

ydb_node_previous_s()

```
ydb_status_t ydb_node_previous_s(ydb_maxsub_t *value,
        ydb_maxsub_t *count,
        ydb_string_t *varname,
        [ ydb_string_t *subscript, ... ]
        NULL);
```

Analogous to `ydb_node_next(s)`, `ydb_node_previous_s()` facilitates breadth-first traversal of a local or global variable tree, except that:

- `ydb_node_previous_s()` reports the predecessor node,
- an input value of 0 for `*value` reports the last node in the tree on output, and
- an output value of 0 for `*value` means there is no previous node.

Other behavior of `ydb_node_previous_s()` is the same as ydb_node_next_s().

ydb_put_s()

```
ydb_status_t ydb_put_s(ydb_string_t *value,
        ydb_maxsub_t count,
        ydb_string_t *varname[,
        ydb_string_t *subscript, ... ]);
```

`ydb_get_s()` sets the value of the data at the specified node to the value referenced by `*value`. If `*value` references a string that is a canonical number, YottaDB converts it to a number and stores the number.

ydb_subscript_next_s()

```
ydb_status_t ydb_subscript_next_s(ydb_string_t *value,
        ydb_maxsub_t *count, ydb_string_t *varname[,
        ydb_string_t *subscript, ... ]);
```

`ydb_subscript_next_s()` returns the next subscript at the lowest level specified by `*count`, by replacing the `subscript->address` at the specified level with the next subscript, and the corresponding `subscript->used` with its length. If there is no next subscript at that level, it decrements `*count`. [8]

If `*count` is zero, `ydb_subscript_next_s()` returns the next local or global variable name, and if `*varname` references the last variable name, `*count` is -1 on the return.

ydb_subscript_previous_s()

```
ydb_status_t ydb_subscript_previous_s(ydb_string_t *value,
        ydb_maxsub_t *count, ydb_string_t *varname[,
        ydb_string_t *subscript, ... ]);
```

`ydb_subscript_previous_s()` returns the preceding subscript at the lowest level specified by `*count`, by replacing the `subscript->address` at the lowest level with the next subscript, and the corresponding `subscript->used` with its length. If there is no previous subscript, it decrements `*count`. [9]

If `*count` is zero, `ydb_subscript_previous_s()` returns the preceding local or global variable name, and if `*varname` references the first global variable name, `*count` is -1 on the return.

ydb_withdraw_s()

```
ydb_status_t ydb_withdraw_s(ydb_maxsub_t count,
        ydb_string_t *varname[,
        ydb_string_t *subscript, ...][, ...] NULL);
```

Note: the parameter list **must** be terminated by a NULL pointer.

Deletes the root node in each of the local or global variable trees or sub-trees specified, leaving the sub-trees unmodified.

Programming Notes

Dynamic typing with automatic conversion

The YottaDB engine internally automatically converts values between numbers and strings as needed. Thus it is legitimate to lexically compare the numbers 2 and 11, with the expected result that 11 precedes 2, and it is equally legitimate to numerically compare the strings “2” and “11”, with the expected result that 11 is greater than 2 — the functions for numeric and lexical comparisons are different.

In the `ydb_value_t` structure, a caller specifies what conversion, if any, it wishes the called libyottadb function to perform on the return value:

- When a value is numeric, and the requested type is a string (the `tag` field is `YDB_STRING_STAR`), libyottadb returns the number as a canonical string in the `ydb_string_t` structure pointed to by `string_star`
- When the value is a string, and the requested type is numeric, libyottadb converts

When returning a string,libyottadb functions *always* check that the `alloc` field of the `ydb_string_t` structure is large enough for the result, returning a `YDB_ERR_STRLEN` error if it is not.

Numeric Considerations

To ensure the accuracy of financial calculations, YottaDB internally stores nnumbers as, and performs arithmetic using, a scaled packed decimal representation with 18 signicant decimal digits, with optimizations for values within a certain subset of its full range. Consequently:

- Any number that is exactly represented in YottaDB can be exactly represented as a string, with reasonably efficient conversion back and forth.
- Any integer value of up to 18 significant digits can be exactly represented by an integer type such as `ydb_longlong-t`, and integers in the inclusive range ±999,999 are handled more efficiently than larger integers.
- In YottaDB there are numbers which can be exactly represented (such as 0.1), but whcih cannot be exactly represented in binary floating point.
- In 64 bit integers and binary floating point formats, there are numbers which can be exactly represented, but which cannot be exactly represented in YottaDB.

This means that for numeric keys which are not guaranteed to be integers:

- In theory, there are edge cases where a value (which would internally be in YottaDB format) returned by a function such as `ydb_subscript_next()` and converted to a `ydb_double_t` when passed back to C application code, and then converted back to YottaDB internal format in a call to `ydb_get()` can result in the node not being found because the double conversion produces a number not identical to the original. Furthermore, there is a cost to the conversion.
- Passing keys back and forth as strings avoids those edge cases, but of course still has a conversion cost.

To preserve accuracy of numeric values that are returned by libyottadb, and which an application code intends to simply pass back to libyottadb as a libyottadb provides a `ydb_numeric_t` type. A value obtained from libyottadb in `ydb_numeric_t` loses no precision when returned to libyottadb, and as a further benefit is very efficient. While the actual value of `ydb_numeric_t` is opaque to application cod, the `ydb_convert()` function is available.

Conversely, when passed a string that is a canonical number for use as a key, libyottadb automatically converts it to a number. This automatic internal conversion is irrelevant for the majority of typical application that:

- simply store and retrieve data associated with keys, potentially testing for the existence of nodes; or
- transfer keys which are numeric values between application code and libyottadb using numeric types and expect numeric ordering.

However, this automatic internal conversion does affect applications that:

- use numeric keys and expect the keys to be sorted in lexical order rather than numeric order; or
- transfer keys which are numeric values between application code and libyottadb as strings that may or may not be canonical numbers.

Applications that are affected by automatic internal conversion should prefix their keys with a character such as “x” which ensures that keys are not canonical numbers.

Canonical Numbers

Conceptually, a canonical number is a string from the Latin character set that represents a decimal number in a standard, concise, form.

1. Any string of decimal digits, optionally preceded by a minus sign ("-"), the first of which is not "0" (except for the number zero itself), that represents an integer of no more than 18 significant digits.
    - The following are canonical numbers: "-1", "0", "3", "10", "99999999999999999", "999999999999999990". Note that the last string has only 18 significant digits even though it is 19 characters long.
    - The following are not canonical numbers: "+1" (starts with "+"), "00" (has an extra leading zero), "999999999999999999" (19 significant digits).
2. Any string of decimal digits, optionally preceded by a minus sign that includes one decimal point ("."), the first and last of which are not "0", that represents a number of no more than 18 significant digits.
    - The following are canonical numbers: "-.1", ".3", ".99999999999999999".
    - The following are not canonical numbers "+.1" (starts with "+"), "0.3" (first digit is "0"), ".999999999999999990" (last digit is "0"), ".9999999999999999999" (more than 18 significant digits).
3. Any of the above two forms followed by "E" followed by a canonical number integer in the range -43 to +47 such that the magnitude of the resulting number is between 1E-43 through.1E47.

Tokens

Since numeric and non-numeric subscripts can be freely intermixed in YottaDB, it requires knowledge of the application schema to know whether an application mixes numeric and string subscripts at the same level for a variable.

Consider whether this can be deferred for an initial implementation.

---

1 Note for implementers: the actual values are negated ZMESSAGE error codes.
2 Note for implementers: YottaDB silently truncates local variable names that are too long. The implementation should catch this and return an error code, e.g., something like `YDB_ERR_LVINVALID`.
3 Note for implementers: under the covers, this is `UNDEF` but renamed to be more meaningful.
4 Note for implementers: this is a new error, not currently in the code base.
5 Strings in YottaDB are arbitrary sequences of bytes that are not null-terminated. Other languages may refer to them as binary data or blobs.
6 Note for implementers: under the covers, `YDB_ALLOC_STRING()` and `YDB_FREE_STRING()` should call the `ydb_malloc()` and `ydb_free()` functions, which are aliases for the `gtm_malloc()` and `gtm_free()` functions (i.e., either prefix calls the same function).
7 Note for implementers: the implementation should attempt to limit the damage by not looking for more subscripts than are permitted by `YDB_MAX_SUB`.

8 This behavior provides symmetry with ydb_subscript_previous_s().

9 Since the empty string is a legal subscript and is the first in YottaDB's natural collation order, simply setting `subscript->used` to zero does not discriminate between the case where the input specifies the first subscript, and the case where there actually is a preceding node with the empty string as a subscript. Decrementing `*count` allows the Simple API to allow for this case.