Table of Contents

Overview

libyottadb is a library for for accessing the YottaDB engine from C using its Simple API. A process can both call the Simple API as well as call functions written in M, the scripting language embedded in YottaDB, 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 document for a valid technical reason. Also, this document itself is incomplete and still evolving.

Quick Start

The Quick Start section needs to be fleshed out.

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

Keys, Values, Nodes, Variables, and Subscripts

The fundamental core data structure provided by YottaDB is *key-value tuples*. For example, the following is a set of key value tuples:

```
["Capital","Belgium","Brussels"]
["Capital","Thailand","Bangkok"]
["Capital","USA","Washington, DC"]
```

Note that data in YottaDB is *always* ordered. [1] Even if you input data out of order, YottaDB always stores them in order. In the discussion below, data is therefore always shown in order. For example, in the example below, data may well be loaded by country.

Each of the above tuples is called a *node*. In an *n*-tuple, the first *n*-1 items can be thought of as the *keys*, and the last item is the *value* associated with the keys.

While YottaDB itself assigns no meaning to the data in each node, by convention, application maintainability is improved by using meaningful keys, for example:

```
["Capital","Belgium","Brussels"]
["Capital","Thailand","Bangkok"]
["Capital","USA","Washington, DC"]
["Population","Belgium",1367000]
["Population","Thailand",8414000]
["Population","USA",325737000]
```

As YottaDB assigns no inherent meaning to the keys or values, its key value structure lends itself to implementing *Variety*. [2] For example, if an application wishes to add historical census results under "Population", the following is a perfectly valid set of tuples (source: United States Census):

```
["Capital","Belgium","Brussels"]
["Capital","Thailand","Bangkok"]
["Capital","USA","Washington, DC"]
["Population","Belgium",1367000]
["Population","Thailand",8414000]
["Population","USA",325737000]
["Population","USA",17900802,3929326]
["Population","USA",18000804,5308483]
…
["Population","USA",20100401,308745538]
```

In the above, 17900802 represents August 2, 1790, and an application would determine from the number of keys whether a node represents the current population or historical census data.

In YottaDB, the first key is called a *variable*, and the remaining keys are called *subscripts* allowing for a representation both compact and familiar to a programmer, e.g., `Capital("Belgium")="Brussels"`. The set of all nodes under a variable is called a *tree* (so in the example, there are two trees, one under `Capital` and the other under `Population`). The set of all nodes under a variable and a leading set of its subscripts is called a *subtree* (e.g., `Population("USA")` is a subtree of the `Population` tree). [3]

With this notation, the `Population` tree can be represented as follows:

```
Population("Belgium")=1367000
Population("Thailand")=8414000
Population("USA")=325737000
Population("USA",17900802)=3929326
Population("USA",18000804)=5308483
…
Population("USA",20100401)=308745538
```

Note that the trees are displayed in breadth-first order. YottaDB has functions for applications to traverse trees in both breadth-first and depth-first order.

If the application designers now wish to enhance the application to add historical dates for capitals, the `Capital("Thailand")` subtree might look like this (source: The Four Capitals of Thailand).

```
Capital("Thailand")="Bangkok"
Capital("Thailand",1238,1378)="Sukhothai"
Capital("Thailand",1350,1767)="Ayutthaya"
Capital("Thailand",1767,1782)="Thonburi"
Capital("Thailand",1782)="Bangkok"
```

Variables vs. Subscripts vs. Values

When viewed as `[“Capital”,”Belgium”,”Brussels”]` each component is a string, and in an abstract sense they are all conceptually the same. When viewed as `Capital(“Belgium”)=”Brussels”` differences become apparent:

- Variables are ASCII strings from 1 to 31 characters, the first of which is “%”, or a letter from “A” through “Z” and “a” through “z”. Subsequent characters are alphanumeric (“A” through “Z”, “a” through “z”, and “0” through “9”). Variable names are case-sensitive, and are always in ASCII order (i.e., “Capital” always precedes “Population”).
- Subscripts are sequences of bytes from 0 (the null or empty string, “”) to 1048576 bytes (1MiB). When a subscript is a canonical number, YottaDB internally converts it to, and stores it as, a number. When ordering subscripts:
    - Empty string subscripts precede all numeric subscripts. *Note: YottaDB strongly recommends against applications that use null subscripts.*
    - Numeric subscripts precede string subscripts. Numeric subscripts are in numeric order.
    - String subscripts follow numeric subscripts and collate in byte order. [4]
- Like subscripts, values are sequences of bytes, except that ordering is not meaningful. YottaDB automatically converts between numbers and strings, depending on the type of operand required by an operator or argument required by a function (see Numeric Considerations).

This means that if an application were to store the current capital of Thailand as `Capital(“Thailand”,”current”)=”Bangkok”` instead of `Capital(“Thailand”)=”Bangkok”`, the above subtree would have the following order:

```
Capital("Thailand",1238,1378)="Sukhothai"
Capital("Thailand",1350,1767)="Ayutthaya"
Capital("Thailand",1767,1782)="Thonburi"
Capital("Thailand",1782)="Bangkok"
Capital("Thailand","current")="Bangkok"
```

Local and Global Variables

YottaDB is a database, and data in a database must *persist* and *be shared*. The variables discussed above are specific to an application process (i.e., are not shared).

- *Local* variables reside in process memory, are specific to an application process, are not shared between processes, and do not persist beyond the lifetime of a process. [5]
- *Global* variables reside in databases, are shared between processes, and persist beyond the lifetime of any individual process.

Syntactically, local and global variables look alike, with global variable names having a caret (“^”) preceding their names. Unlike the local variables above, the global variables below are shared between processes and are persistent.

```
^Population("Belgium")=1367000
^Population("Thailand")=8414000
^Population("USA")=325737000
```

Even though they may appear superficially similar, a local variable is distinct from a global variable of the same name. Thus `^X` can have the value 1 and `X` can at the same time have the value `“The quick brown fox jumps over the lazy dog.` For maintainability **YottaDB strongly recommends that applications use different names for local and global variables, except in the special case where a local variable is an in-process cached copy of a corresponding global variable.**

Global Directories

To application software, files in a file system provide persistence. This means that global variables must be stored in files for persistence. A *global directory file* provides a process with a mapping from the name of every possible global variable name to a *database file*. A *database* is a set of database files to which global variables are mapped by a global directory. Global directories are created and maintaind by a utility program called the Global Directory Editor, which is discussed at length in the GT.M Administration and Operations Guide and is outside the purview of this document.

The name of the global directory file required to access a global variable such as `^Capital`, is provided to the process at startup by the environment variable `ydb_gbldir`.

In addition to the implicit global directory an application may wish to use alternate global directory names. For example, consider an application that wishes to provide an option to display names in other languages while defaulting to English. This can be accomplished by having different versions of the global variable `^Capital` for different languages, and having a global directory for each language. A global variable such as `^Population` would be mapped to the same database file for all languages, but a global variable such as `^Capital` would be mapped to a database file with language-specific entries. So a default global directory `Default.gld` mapping a `^Capital` to a database file with English names can be specified in the environment variable `ydb_gbldir` but a different global directory file, e.g., `ThaiNames.gld` can have the same mapping for a global variable such as `^Population` but a different database file for `^Capital`.

Thus, we can have:

```
^|"ThaiNames.gld"|Capital("Thailand")="□□□□□□□□□"
^|"ThaiNames.gld"|Capital("Thailand",1238,1378)="□□□□□□□"
^|"ThaiNames.gld"|Capital("Thailand",1350,1767)="□□□□□□"
^|"ThaiNames.gld"|Capital("Thailand",1767,1782)="□□□□□□"
^|"ThaiNames.gld"|Capital("Thailand",1782)="□□□□□□□□□"
```

The global directory name can itself be a variable name. So if the variable `CurrLangGld` is set to `“ThaiNames.gld”`, the capital of Thailand can be referred to in the current language, e.g., `^|CurrLangGld|Capital(“Thailand”)=”□□□□□□□□□”`

A global variable reference that explictly specifies a global directory is called an *extended reference*.

Intrinsic Special Variables

In addition to local and global variables, YottaDB also has a set of *Intrinsic Special Variables*. Just as global variables are distinguised by a “^” prefix, intrinsic special variables are distinguished by a “$” prefix. Instead of using an extended reference, an application can set an intrinsic special variable `$zgbldir=”ThaiNames.gld”` to use the `ThaiNames.gld` mapping.

Unlike local and global variable names, intrinsic special variable names are case-insensitive and so `^zgbldir` and `$ZGblDir` refer to the same intrinsic special variable. Also intrinsic special variables have no subscripts.

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

Symbolic constants all fit within the range of a C `int`.

Function Return Codes

Return codes from calls to libyottadb are of type `int`. 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 have `YDB_` prefixes other than `YDB_ERR_`

`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_` and are all less than zero. [6] The symbolic constants below are not a complete list of all error messages that Simple API functions can return — error return codes can indicate system errors and database errors, not just application errors. The `ydb_message()` function provides a way to get more detailed information about any error code returned by a Simple API function, including error codes for return values without symbolic constants.

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

`YDB_ERR_INSUFFSUBS` — A call to `ydb_node_next_s()` or `ydb_node_previous_s()` did not provide enough parameters for the return values. [7]

`YDB_ERR_INVSTRLEN` — A buffer provided by the caller is not long enough for a 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 a `ydb_string_t` value whose `xyz->alloc` indicates insufficient 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 a `ydb_string_t` structure is greater than the `alloc` field.

`YDB_ERR_INVSVN` — A special variable name provided by the caller is invalid.

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

`YDB_ERR_LVUNDEF` — No value exists at a requested local variable node. [8]

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

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

`YDB_ERR_VARNAMEINVALID` — A variable name is too long. [9]

Limits

Symbolic constants for limits are prefixed with `YDB_MAX_`.

`YDB_MAX_IDENT` —The maximum space in bytes required to store a complete variable name, not including the preceding caret for a global variable. Therefore, when allocating space for a string to hold a global variable name, add 1 for the caret, and when allocating space for a string to hold an extended global reference, add 3 (the caret and two "|" characters) as well as the maximum path for a global directory file.

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

Data Structures

`ydb_string_t` is a descriptor for a string [10] value, and consists of the following fields:

- `alloc` and `used` — fields of type `unsigned int` where `alloc` ≥ `used` except when a YDB_ERR_INVSTRLEN occurs.
- `address` — pointer to an `unsigned char`, the starting address of a string.

Macros

`YDB_ALLOC_STRING(string[,actalloc])` — Allocate a `ydb_string_t` structure and set its `address` field to point to `string`, and its `used` field to the length of string excluding the terminating null character. Set its `alloc` field to `actalloc` if specified, otherwise to `used`. Return the address of the structure. Note that if string is a `const` any code that attempts to change the value of the string pointed to by this `ydb_string_t` structure will almost certainly result in a segmentation violation (SIGSEGV). [11]

`YDB_COPY_STRING(dest,src)` — Confirm that `dest->alloc` ≥ `src->used`, and if so copy `src->used` bytes from memory pointed to by `src->address` to the memory pointed to by `dest->address`, returning `YDB_STATUS_OK`. If `dest->alloc` < `src-used`, return `YDB_ERR_INVSTRLEN`.

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

`YDB_FREE_STRING_DEEP(x)` — Free the memory referenced by `x->address` and free the `ydb_string_t` structure pointed to by `x`.

`YDB_NEW_STRING(string[,minalloc])` — Allocate memory sufficient to hold `string` (excluding the trailing null character) and copy `string` to that memory. If `minalloc` is specified, allocate at least `minalloc` bytes. At the implementer's option, the allocation may be further rounded up to a preferred size. Copy `string` to the newly allocated memory. Allocate a `ydb_string_t` structure and set its `address` field to point to the newly allocated memory, its `alloc` field to point to the size of allocated memory, and its `used` field to the length of `string`. Return the address of the new ydb_string_t structure. Use an empty string as the value of string to preallocate structures for use, e.g., YDB_NEW_STRING("",YDB_MAX_IDENT) to create space for a local variable name to be returned by a function such as ydb_subscript_next_s().

YDB_SET_STRING(x, string) — Check whether the x->alloc has sufficient space for string and if so, copy string excluding the terminating null character to the memory pointed to by x->address and set x->used to the length of string.

Simple API

As all subscripts and node data passed to libyottadb using the Simple API are strings, use the printf() and scanf() family of functions to convert between numeric values and strings which are canonical numbers.

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:

- int count and int *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. [12]

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

ydb_data_s()

```
int ydb_data_s(unsigned int *value,
        int 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 *varname and the *subscript list.

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

ydb_get_s()

```
int ydb_get_s(ydb_string_t *value,
        int count,
```

```
        ydb_string_t *varname[,
        ydb_string_t *subscript, ... ]);
```

If `value->alloc` is large enough to accommodate the result, to the location pointed to by `value->address`, `ydb_get_s()` copies the value of the value of the data at the specified node or intrinsic special variable, setting `value->used`, and returning `YDB_STATUS_OK`; and `YDB_ERR_INVSTRLEN` otherwise.

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.

Note: 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 subsequent call to `ydb_get()` to get that data. If a caller cannot ensure from the application design 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()

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

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

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

ydb_kill_excl_s()

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

`*varnamelist->address` points to 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()

```
int ydb_length_s(unsigned int *value,
        int 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_message()

```
int ydb_message(ydb_string_t *msgtext, int status)
```

Set `msgtext->address` to a location that has the text for the condition corresponding to `status`, and both `msgtext->alloc` and `msgtext->used` to its length (with no trailing null character). Note: as `msgtext->address` points to an address in a read-only region of memory, any attempt to modify the message will result in a segmentation violation (SIGSEGV). `ydb_message()` returns `YDB_STATUS_OK` for a valid `status` and `YDB_ERR_UNKNOWN` if `status` does not map to a known error.

ydb_node_next_s()

```
int ydb_node_next_s(int *count,
        ydb_string_t *varname,
        ydb_string_t *subscript[, ... ]);
```

`ydb_node_next_s()` facilitates depth-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 `*count` 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 `int` value returned by `ydb_node_next_s()` returns an error code, `*count` 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 `*count` 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. In this case `*count` reports the actual number of subscripts in the node, and the parameters report as many subscripts as can be reported.
- 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 `*count` do not contain meaningful values.

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

ydb_node_previous_s()

```
int ydb_node_previous_s(int *count,
        ydb_string_t *varname,
        [ ydb_string_t *subscript, ... ]);
```

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()

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

Copies the `value->used` bytes at `value->address` as the value of the specified node or intrinsic special variable specified, returning `YDB_STATUS_OK` or an error code such as `YDB_ERR_INVSVN`.

ydb_subscript_next_s()

```
int ydb_subscript_next_s(int *count,
        ydb_string_t *varname[, ydb_string_t *subscript, ... ]);
```

`ydb_subscript_next_s()` returns the next subscript at the deepest level specified by `*count`, by copying that next subscript to the memory referenced by that `subscript->address`, and setting the corresponding `subscript->used` with its length. If there is no next subscript at that level, it decrements `*count`. [13]

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()

```
int ydb_subscript_previous_s(int *count,
        ydb_string_t *varname[, ydb_string_t *subscript, ... ]);
```

`ydb_subscript_previous_s()` returns the preceding subscript at the deepest level specified by `*count`, by copying that previous subscript to the memory referenced by that `subscript->address`, and setting the corresponding `subscript->used` to its length. If there is no previous subscript, it decrements `*count`. [14]

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

ydb_withdraw_s()

```
int ydb_withdraw_s(int 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 subtrees specified, leaving the subtrees intact.

Programming Notes

Numeric Considerations

To ensure the accuracy of financial calculations, [15] YottaDB internally stores numbers 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.

When passed a string that is a canonical number for use as a subscript, libyottadb automatically converts it to a number. This automatic internal conversion is immaterial for applications:

- that simply store and retrieve data associated with subscripts, potentially testing for the existence of nodes; or
- whose subscripts are all numeric, and should be collated in numeric order.

This automatic internal conversion is material to applications that use:

- numeric subscripts and expect the subscripts to be sorted in lexical order rather than numeric order; or
- mixed numeric and non-numeric subscripts, including subscripts that are not canonical numbers.

Applications that are affected by automatic internal conversion should prefix their subscripts with a character such as “x” which ensures that subscripts 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”, “999999999999999999”, “9999999999999999990”. 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), “9999999999999999999” (19 significant digits), “-0” (the canonical representation of 0 is “0”).

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”, “.999999999999999999”.
   - The following are not canonical numbers “+.1” (starts with “+”), “0.3” (first digit is “0”), “.9999999999999999990” (last digit is “0”), “.9999999999999999999” (more than 18 significant digits).

3. Any of the above two forms followed by “E” (upper case only) followed by a canonical integer in the range -43 to 47 such that the magnitude of the resulting number is between 1E-43 through.1E47.

To do

Transaction Processing

Universal NoSQL

---

1 The terms “collate”, “order”, and “sort” are equivalent.

2 Variety is one of the *three* “V”s of “big data” - Velocity, Volume, and Variety. YottaDB handles all three very well.

3 Of course, the ability to represent the data this way does not in any way detract from the ability to represent the same data another way, such as XML or JSON, with which you are comfortable. However, note while any data that can be represented in JSON can be stored in a YottaDB tree not all trees that YottaDB is capable of storing can be represented in JSON, or at least, may require some encoding in order to be represented in JSON.

4 Where the natural byte order does not result in linguistically and culturally correct ordering of strings, YottaDB has a framework for an application to create and use custom collation routines.

5 In other words, what YottaDB calls a local variable, the C programming language calls a global variable. There is no C counterpart to a YottaDB global variable.

6 Note for implementers: the actual values are negated ZMESSAGE error codes.

7 Note for implementers: this is a new error, not currently in the code base.

8 Note for implementers: under the covers, this is `UNDEF` but renamed to be more meaningful.

9 Note for implementers: While correctly issuing GVINVALID for too-long global variable names, YottaDB silently truncates local variable names that are too long. The implementation should catch this. `YDB_ERR_VARNAMEINVALID` can map to the existing GVINVALID, and change the message returned by `ydb_message()` appropriately.

10 Strings in YottaDB are arbitrary sequences of bytes that are not null-terminated. Other languages may refer to them as binary data or blobs.

11 Note for implementers: under the covers, `YDB_ALLOC_*()`, `YDB_FREE_*()`, and `YDB_NEW_*()` macros 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). Also, for efficiency reasons, we may want to have two macros, `YDB_ALLOC_STRING()` and `YDB_ALLOC_STRLIT()`.

12 Note for implementers: the implementation should attempt to limit the damage by not looking for more subscripts than are permitted by `YDB_MAX_SUB`.

13 This behavior provides symmetry with ydb_subscript_previous_s().

14 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 discriminate between the two cases.

15 For example, since a number such as .01 is not exactly representable as a binary or hexadecimal floating point number adding a list of currency values using floating point arithmetic does not guarantee that the result will be correct to the penny, which is a requirement for financial calculations.