

VSI OpenVMS PHP updates

Brett Cameron (brett.cameron@vmssoftware.com), November 2020

Introduction

This paper provides a brief overview of new PHP extensions provided by VMS Software Inc. (VSI) for the Oracle RDB and Mimer relational databases, for the VSI DATATRIEVE data query and reporting product, and for the Redis in-memory cache. These extensions are included in the PHP 5.6.10L kit for OpenVMS, which may be installed on OpenVMS V8.4-1H1 or higher. The Oracle RDB, Mimer, and DATATRIEVE extensions have been developed specifically by VSI, while Redis support is facilitated via the PhpRedis open source project (see <https://github.com/phpredis/phpredis>).

The following sections provide a brief overview of the four extensions, outlining the functions that are available, and providing simple examples to illustrate their usage. It should be noted that the RDB, Mimer, and DATATRIEVE extensions have been kept deliberately simple. It is likely that additional functionality will be added to these extensions over time with new releases of PHP for VSI OpenVMS; however the functionality that is currently provided should be more than sufficient for most cases.

DATATRIEVE API

DATATRIEVE is a database query and report writing product, originally developed in the late 1970's for the PDP-11 and subsequently for VAX/VMS and OpenVMS¹. DATATRIEVE can be used with flat files, indexed RMS files, as well as with a variety of databases, including Oracle RDB and the Oracle CODASYL DBMS. The product uses a SQL-like language to interact with data sources and to generate reports, and Oracle CDD or plain text files can be used to create dictionaries and define object definitions for the various data sources. Despite being such an old product, DATATRIEVE remains to this day very popular, due in no small part because of its ability to interact with indexed RMS files, which are often used form of data storage for OpenVMS applications.

Most commonly, DATATRIEVE is used via the command line interface; however, it also provides a powerful programmable API, which can be used to include DATATRIEVE functionality directly into application programs. To further enhance this capability, VSI have developed API bindings for the DATATRIEVE API for several popular scripting languages, including PHP.

Being able to use the DATATRIEVE API directly from PHP allows software developers to implement powerful web applications that can interact with indexed RMS files and other supported data sources. The ability to efficiently update and retrieve data stored in indexed RMS files is especially useful, as many older OpenVMS applications make extensive use of such files for data storage, as noted above. The PHP interface for the most part maps directly to the native DATATRIEVE API and is straightforward to use. Once record structures have been defined for the data in question, the general sequence of API calls to interact with the data will be largely the same from one application to the next. There are also a number of ancillary functions provided by the PHP interface that do not map directly to DATATRIEVE API functions. These functions facilitate access to API data structures

¹ DATATRIEVE remains popular to this day and is widely used by OpenVMS customers, primarily for generating reports from data stored in RMS files.

and provide direct access to the interactive data management capabilities of the DATATRIEVE product. Error codes and other constants provided by the DATATRIEVE API map directly to equivalent constant values in the PHP interface, as will be described below.

The following tables list the various functions provided by the PHP DATATRIEVE interface. Note that the interface comprises two sets of functions, namely functions that interact directly with the DATATRIEVE API and another set of functions that can be used to define record structures and get and set values stored in those structures. It should be noted that the information presented in these tables assumes that the reader has a good knowledge of the DATATRIEVE API and of OpenVMS data types. Additional information regarding the DATATRIEVE API can be found in the DATATRIEVE Guide to Programming and Customizing.

| Function | Description |
|----------------------------------|--|
| <code>dtr_aux_buf()</code> | Returns the contents of the auxiliary message buffer. The content of the auxiliary message buffer is an ASCII string of maximum length 255 characters. |
| <code>dtr_command()</code> | Passes a command string to DATATRIEVE. Note that it is not possible to specify any substitution directives in the command string (this functionality may be added in future). |
| <code>dtr_condition()</code> | Returns a numeric value identifying the status of the last executed DATATRIEVE command or statement. |
| <code>dtr_cont()</code> | Instructs DATATRIEVE to continue processing when at a “continue”, “message”, or “print-line” stall-point. |
| <code>dtr_create_udk()</code> | Can be used to add a new user-defined keyword (UDK) to DATATRIEVE. |
| <code>dtr_dtr()</code> | Invokes the DATATRIEVE terminal server, giving the program access to all DATATRIEVE interactive data management capabilities. |
| <code>dtr_end_udk()</code> | Ends processing of a user-defined keyword (previously defined via a call to <code>dtr_create_udk()</code>). |
| <code>dtr_finish()</code> | Releases all memory and other resources associated with the API reference in question and ends the DATATRIEVE session. The API reference handle cannot be used after performing this call. |
| <code>dtr_get_port()</code> | Gets a record from DATATRIEVE and loads it into the supplied program buffer. |
| <code>dtr_get_rec_len()</code> | Returns the length of the current (last read) record. |
| <code>dtr_get_string()</code> | Gets a string from DATATRIEVE and copies it into the supplied program buffer. This call can be used to interpret the arguments to a user-defined keyword. |
| <code>dtr_get_udk_index()</code> | Returns the index number of the current user-defined keyword. |
| <code>dtr_info()</code> | Returns information about a specified DATATRIEVE object, where the specified object identifier has been determined via a previous call to <code>dtr_lookup()</code> . |
| <code>dtr_init()</code> | Initializes the DATATRIEVE API, returning a unique API reference that can be used with subsequent API calls. |
| <code>dtr_lookup()</code> | Returns a number that identifies a DATATRIEVE object such as a dictionary object or the identifier for a DATATRIEVE keyword. The |

| | |
|------------------------------|---|
| | object number can then be used in a call to <code>dtr_info()</code> to get information about the object in question. Dictionary objects include entities such as domains, collections, record definitions, fields, sub-schemas, sets, and plots. |
| <code>dtr_msg_buf()</code> | Returns the contents of the message buffer. The content of the message buffer is an ASCII string of maximum length 255 characters. |
| <code>dtr_port_eof()</code> | Terminates the passing of records from your program to DATATRIEVE. |
| <code>dtr_put_port()</code> | Passes a record from your application into DATATRIEVE. The structure of the record must be previously defined and known to DATATRIEVE. |
| <code>dtr_put_value()</code> | Passes a value to DATATRIEVE. The specified value must be a valid ASCII string. |
| <code>dtr_row()</code> | Returns an array of string values representing the individual fields of the last (current) read record. The string values are read from the current record message buffer based on a supplied C-like format specification. In general, it is recommended to use variants of the <code>%c</code> format specifier to precisely extract field values, as illustrated in the second example program below. |
| <code>dtr_skip()</code> | Skips the specified number of output records (the records are read and discarded). This call can be useful when programmatically accessing pre-existing DATATRIEVE reports that return some number of header records that are of no particular interest to the program. |
| <code>dtr_state()</code> | Returns the value of the current stall-point code. This value can be used to determine what action to take next or to determine that DATATRIEVE is in the expected state following the completion of a particular operation or sequence of operations. |
| <code>dtr_unwind()</code> | Stops execution of a command or statement passed to DATATRIEVE (most commonly used in an interactive context). |

Table 1. Functions provided by the DATATRIEVE PHP interface. In most cases the methods, their arguments, and return values map to those of the corresponding DATATRIEVE API functions. The interface also includes several additional functions to get and set values in the DATATRIEVE access block data structure.

Details of function arguments and return values are not described here; however for the most part the functions and their arguments and return values closely mirror those of the associated DTR API function, as is illustrated by the PHP code examples that are included below. One perhaps more notable difference is that constants defined for the PHP interface are prefixed by `DTR_` instead of `DTR$`. Again, this is illustrated in the example code.

As commented previously, in addition to the above functions that map to equivalent functions in the native DATATRIEVE API, there are another set of PHP functions that can be used to define record structures and to get and set values stored in those structures. These record structures are used by the DATATRIEVE PHP interface when retrieving data records as well as when inserting and updating data. After allocating a new data structure object, the data structure is defined via a series of "add" calls to specify the location (offset), size, and type of individual record fields. It should be noted that for some data types such as integers and floats the size is implicit and does not need to be specified, while for other types such as strings and numeric data types such as packed decimal it is necessary to explicitly specify the size of the field, and in the case of numeric types other attributes such as scale and the number of digits also need to be given. It should be noted that for numeric fields the

field type is specified using standard OpenVMS descriptor types, for which PHP constants are defined in the `dscdef` module (see Table 3 below).

| Function | Description |
|-----------------------------|--|
| <code>rec_new()</code> | Allocates a new record structure and returns a reference to the new structure for use with other functions. |
| <code>rec_delete()</code> | Deletes a record structure and its contents. The reference to the record structure cannot be used after the record has been deleted. |
| <code>rec_getstr()</code> | Returns the string value stored in the specified field index of the record structure. Note that the string is trimmed to remove any trailing spaces. Field indexes start from 0. |
| <code>rec_getint()</code> | Returns the integer value stored in the specified field index of the record structure. Note that integer values are always returned as signed 64-bit integers, irrespective of how they are stored. |
| <code>rec_addstr()</code> | Adds a string to the specified record structure. The size and offset of the string must be specified. The string value may be <code>null</code> . |
| <code>rec_addint()</code> | Adds an integer of the specified type at the specified offset within the record. Valid integer types are signed and unsigned byte, word (2 bytes), longword (4 bytes), and quadword (8 bytes), as indicated by the constants defined in Table 3 below. |
| <code>rec_load()</code> | Can be used to load binary data into a previously allocated and defined record structure. The length of the supplied data must be at least equal to the sum of the individual record field sizes. Note that this function would not typically be used in conjunction with <code>DATATRIEVE</code> and would more typically be used to load the contents of raw RMS records. |
| <code>rec_addfloat()</code> | Adds a floating-point value of the specified type at the specified offset within the record. Valid floating point types are IEEE and F-floating single precision (4 bytes), and IEEE and G-floating double precision (8 bytes) types, as indicated by the constants defined in Table 3 below. |
| <code>rec_getfloat()</code> | Returns the floating-point value stored in the specified field index of the record structure. Note that floating point values are always returned as double precision values, irrespective of whether they are stored in the record structure and single or double precision. |
| <code>rec_addnum()</code> | Adds a numeric field to the record structure. In addition to specifying the type for the field (as determined by one of the numeric type constants listed below in Table 3), it is also necessary to specify the scale, number of digits, and overall length of the numeric value, and its offset (starting location) within the record. Numeric values are supplied as double precision values and are converted internally to the desired type (if such conversion is possible). |
| <code>rec_gethex()</code> | Returns as a hexadecimal string the integer value stored at the specified field index of the record structure. |
| <code>rec_getnum()</code> | Returns as a double precision value the numeric value stored in the specified field index of the record structure. |
| <code>rec_setstr()</code> | Sets the value of the string stored at the specified field index. Note that strings will be silently truncated if they are longer than the space |

| | |
|-----------------------------|--|
| | previously allocated by <code>rec_addstr()</code> and will be space-padded if they are shorter than the space previously allocated. Once the record structure has been defined it cannot be modified. |
| <code>rec_setint()</code> | Sets the value of the integer stored at the specified field index. The supplied value must be able to be stored in the memory previously allocated by <code>rec_addint()</code> . |
| <code>rec_setnum()</code> | Sets the value of the numeric field at the specified field index. The value to be set is supplied as double precision value, which is converted internally to the required numeric type (if such conversion is possible). The supplied value must be able to be stored in the memory previously allocated by <code>rec_addnum()</code> . |
| <code>rec_setfloat()</code> | Sets the value of the floating-point field stored at the specified index. Note that the supplied value must be able to be stored in the memory previously allocated by <code>rec_addfloat()</code> (attempting to store a double precision value in a 32-bit field will result in data corruption). |

Table 2. Functions for the creation and manipulation of record structures to be used in conjunction with the DATATRIEVE interface for reading and updating data records.

As noted above, when numeric data are added to a record the type of the numeric value is indicated using the appropriate constant from the `dscdef` module, as illustrated in the following example, with the list of supported numeric types as listed in Table 3. In general for the DATATRIEVE interface it is recommended to work with strings where possible, as this significantly simplifies the definition and use of record structures; however this will not always be possible, as ultimately the record structure is determined by the structure of the underlying data.

```
// Define the pieces of the UAF record that we are interested in (the definition does not need
// to be sequential).
//
$r = rec_new();
rec_addstr($r, null, 32, 4); // 0) Username
rec_addint($r, DSC_K_DTYPE_WU, 0, 36); // 1) UIC member
rec_addint($r, DSC_K_DTYPE_WU, 0, 38); // 2) UIC group
rec_addstr($r, null, 31, 52); // 3) Account name
rec_addint($r, DSC_K_DTYPE_B, 0, 84); // 4) Length of owners name
rec_addstr($r, null, 31, 85); // 5) Owner's name
rec_addint($r, DSC_K_DTYPE_B, 0, 116); // 6) Length of default device
rec_addstr($r, null, 31, 117); // 7) Default device
rec_addint($r, DSC_K_DTYPE_B, 0, 148); // 8) Length of default directory
rec_addstr($r, null, 31, 149); // 9) Default directory
rec_addint($r, DSC_K_DTYPE_LU, 0, 468); // 10) Flags
rec_addint($r, DSC_K_DTYPE_B, 0, 516); // 11) Base priority
rec_addint($r, DSC_K_DTYPE_Q, 0, 364); // 12) Expiration date
rec_addint($r, DSC_K_DTYPE_QU, 0, 412); // 13) Privileges vector
```

The above example illustrates the definition of a record structure that could hypothetically be used in conjunction with DATATRIEVE to read user records from `SYSUAF.DAT`. The definition includes a number of data types, including strings and integer fields of various size and signedness.

The table below lists the supported data types for numeric, integer, and floating-point field and identifies the function that must be used in each case to add a field of the given type to a record structure. As noted previously, once the record structure has been defined, it cannot be changed, and it is not possible to set field values that do not conform to the defined type. The various constants are defined in the `dscdef` extension.

| Constant | Type | Function |
|----------------------------|-------------|-------------------------|
| <code>DSC_K_DTYPE_B</code> | Signed byte | <code>rec_addint</code> |

| | | |
|-----------------|--|--------------|
| DSC_K_DTYPE_BU | Unsigned byte | rec_addint |
| DSC_K_DTYPE_F | Single-precision floating point (F-floating) | rec_addfloat |
| DSC_K_DTYPE_FS | IEEE float, single-precision | rec_addfloat |
| DSC_K_DTYPE_FT | IEEE float, double-precision | rec_addfloat |
| DSC_K_DTYPE_G | Double-precision floating point (G-floating) | rec_addfloat |
| DSC_K_DTYPE_L | Signed longword | rec_addint |
| DSC_K_DTYPE_LU | Unsigned longword | rec_addint |
| DSC_K_DTYPE_NL | Numeric string, left separate sign | rec_addnum |
| DSC_K_DTYPE_NLO | Numeric string, left over-punched sign | rec_addnum |
| DSC_K_DTYPE_NR | Numeric string, right separate sign | rec_addnum |
| DSC_K_DTYPE_NRO | Numeric string, right over-punched sign | rec_addnum |
| DSC_K_DTYPE_NU | Numeric string, unsigned | rec_addnum |
| DSC_K_DTYPE_NZ | Numeric string, zoned sign | rec_addnum |
| DSC_K_DTYPE_P | Packed decimal string | rec_addnum |
| DSC_K_DTYPE_Q | Signed quadword | rec_addint |
| DSC_K_DTYPE_QU | Unsigned quadword | rec_addint |
| DSC_K_DTYPE_W | Signed word | rec_addint |
| DSC_K_DTYPE_WU | Unsigned word | rec_addint |

Table 3. Constants for supported data types for integer, numeric, and floating point fields to be used when adding fields to record structures using the `rec_addint()`, `rec_addfloat()`, or `rec_addnum()` methods.

The following PHP script illustrates use of the DATATRIEVE PHP interface to query and read records from the example file `YACHTS.DAT` provided with the DATATRIEVE product. It is assumed that this file has been copied into the same directory as the PHP script and that the record structure of the file has been previously defined in the DATATRIEVE dictionary. Assuming that the environment is set up correctly, the script will read and display data records for yachts of length greater than 30 feet. The example illustrates basic usage of the DATATRIEVE PHP interface and the definition of record structures and retrieval of individual record fields using the functions defined in Table 2.

```
<?php
    if (! extension_loaded('rec')) {
        if (! dl('rec.exe')) {
            exit;
        }
    }
    if (! extension_loaded('dtr')) {
        if (! dl('dtr.exe')) {
            exit;
        }
    }

    $r = rec_new();
    rec_addstr($r, null, 10, 0);
    rec_addstr($r, null, 10, 10);
    rec_addstr($r, null, 6, 20);
    rec_addstr($r, null, 3, 26);
    rec_addstr($r, null, 5, 29);
    rec_addstr($r, null, 2, 34);
    rec_addstr($r, null, 5, 36);

    list($sts, $dab) = dtr_init(100, 0);
```

```

if ($sts != 1) {
    printf("Initialization error\n");
    exit();
}

list($sts, $cond, $state) = dtr_command($dab, 'set dictionary cdd$stop.dtr$lib.demo;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

if ($state == DTR_K_STL_MSG) {
    printf("%s\n", dtr_msg_buf($dab));
    dtr_cont($dab);
}

list($sts, $cond, $state) = dtr_command($dab, 'declare port yport using yacht;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

while ($state == DTR_K_STL_MSG) {
    printf("%s\n", dtr_msg_buf($dab));
    list($sts, $cond, $state) = dtr_cont($dab);
}

list($sts, $cond, $state) = dtr_command($dab, 'ready yachts; ready yport write;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

while ($state == DTR_K_STL_MSG) {
    printf("%s\n", dtr_msg_buf($dab));
    list($sts, $cond, $state) = dtr_cont($dab);
}

list($sts, $cond, $state) = dtr_command($dab, 'yport = yachts with loa > 30;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

while ($state == DTR_K_STL_MSG) {
    printf("%s\n", dtr_msg_buf($dab));
    list($sts, $cond, $state) = dtr_cont($dab);
}

while ($state == DTR_K_STL_PGET) {
    $sts = dtr_get_port($dab, $r);

    if ($sts != 1) {
        printf("Error fetching data\n");
        exit();
    }

    printf("%-10s\t%-10s\t%-6s\t%-3s\t%-5s\t%-2s\t%-5s\n", rec_getstr($r, 0),
        rec_getstr($r, 1), rec_getstr($r, 2), rec_getstr($r, 3),
        rec_getstr($r, 4), rec_getstr($r, 5), rec_getstr($r, 6));
    $state = dtr_state($dab);
}

$sts = dtr_finish($dab);

if ($sts != 1) {
    printf("Problem tidying up\n");
    exit();
}

rec_delete($r);
?>

```

The following example script illustrates the use of the DATATRIEVE PHP interface to query an RDB database. Note that in this case the result set is processed as if the user were performing an interactive DATATRIEVE session. After issuing the `print` command to display the results of the proceeding query operation, the `dtr_skip()` function is used to skip the header lines of what would be the interactive output, and `dtr_row()` is then used within a loop to read all returned records in accordance with the given format specification. Note that this "interactive" approach must be used when working with RDB; it is not possible to define and use record structures as done in the previous example, which reads records from an RMS file as opposed to an RDB database.

```
<?php
    if (! extension_loaded('dtr')) {
        if (! dl('dtr.exe')) {
            exit;
        }
    }

list($sts, $dab) = dtr_init(100, 0);

if ($sts != 1) {
    printf("Initialization error\n");
    exit();
}

list($sts, $cond, $state) = dtr_command($dab, 'set dictionary cdd$stop.dtr$lib.demo.rdb;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

if ($state == DTR_K_STL_MSG) {
    printf("%s\n", dtr_msg_buf($dab));
    dtr_cont($dab);
}

list($sts, $cond, $state) = dtr_command($dab, 'ready jobs shared read;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

while ($state == DTR_K_STL_MSG) {
    printf("%s\n", dtr_msg_buf($dab));
    list($sts, $cond, $state) = dtr_cont($dab);
}

list($sts, $cond, $state) = dtr_command($dab, 'print jobs;');

if ($sts != 1) {
    printf("Command error\n");
    exit();
}

$sts = dtr_skip($dab, 4);

$fmt = "%4c  %1c  %20c %11c  %11c";

while (dtr_state($dab) == DTR_K_STL_LINE) {
    $row = dtr_row($dab, $fmt);
    print_r($row);
}

$sts = dtr_finish($dab);

if ($sts != 1) {
    printf("Problem tidying up\n");
    exit();
}

?>
```


RDB API

Oracle RDB is arguably the most commonly used relational database platform for OpenVMS. Over the years, various custom solutions have been developed to serve data stored in RDB databases via some form of web-based interface; however it is relatively straightforward to implement a generic RDB interface by creating a simple PHP extension that will allow developers to include queries and other database operations directly into PHP web pages.

With this philosophy in mind, VSI have developed a simple PHP extension for Oracle RDB that implements the set of functions listed in the table below.

| Function | Description |
|-----------------------------------|--|
| <code>rdb_fetch_row()</code> | Fetches a row of data using the specified cursor. The fetched data is returned as an array of strings. |
| <code>rdb_ncol()</code> | Returns the number of columns (values) that would be returned by a fetch for the specified cursor. |
| <code>rdb_attach()</code> | Attaches to the specified database. |
| <code>rdb_close_cursor()</code> | Closes the specified cursor. |
| <code>rdb_commit()</code> | Commits the current database transaction. |
| <code>rdb_data()</code> | Returns the data value for the specified column for the last fetch operation. |
| <code>rdb_declare_cursor()</code> | Declares a cursor. |
| <code>rdb_detach()</code> | Disconnects from the database. |
| <code>rdb_exec()</code> | Executes a previously prepared SQL statement. |
| <code>rdb_execi()</code> | Executes the supplied SQL statement immediately. |
| <code>rdb_fetch()</code> | Fetches a row of data for the specified cursor but does not explicitly return the fetched data. |
| <code>rdb_free()</code> | Frees resources associated with the supplied cursor handle. |
| <code>rdb_open_cursor()</code> | Opens a cursor using the supplied (previously declared) cursor handle. |
| <code>rdb_prepare()</code> | Prepares an SQL statement and returns a handle for the prepared statement. |
| <code>rdb_rollback()</code> | Rolls back the current database transaction. |
| <code>rdb_set_readonly()</code> | Starts a read-only transaction. |
| <code>rdb_error()</code> | Returns a description of the last error. |
| <code>rdb_sqlcode()</code> | Returns the <code>SQLCODE</code> for the last database operation. |

The interface has been kept deliberately simple, providing a small set of easy-to-use functions that provide sufficient functionality to address most requirements. Usage of the interface is for the most part self-explanatory and a detailed description of each individual function is therefore not required. Instead, usage is perhaps best illustrated by way of example.

The following example script illustrates the use of the extension to declare and use a database cursor to query the `employees` table in the sample database provided with Oracle RDB installations.

The code begins by attaching to the database using the `rdb_attach()` function, which takes as input a single argument specifying the name of the database to attach to (in this case via the logical name `sql$database`). Note that the `rdb_attach()` function returns a completion status; it does not return any form of connection handle. For any real-world web application, the database attach would generally be performed only once as part of some initialization routine, as repeatedly connecting and disconnecting to and from the database when processing each PHP web page request would incur significant overhead.

After connecting to the database, the script declares a cursor using the `rdb_declare_cursor()` function. The inputs to this function are the name of the cursor and the SQL statement for the cursor. Upon successful completion, the function returns a cursor handle that can then be used in subsequent calls to open and close the cursor and to fetch records. Note that it is up to the caller to specify a unique name for each cursor. Also note that cursor handles can be reused until they are explicitly freed via a call to the `rdb_free()` function, which frees allocated memory and cleans up any database resources associated with the handle that are no longer required.

Before opening the cursor, the script starts a new transaction using the `rdb_set_readonly()` function, which explicitly starts a read-only transaction on the database. It should be noted that it is not necessary to explicitly start a transaction; however it is generally good practice to do so, and if it is known that the transaction will be only reading data then it is generally beneficial to declare the transaction as read-only. This reduces resource usage and helps to avoid contention issues. After starting the transaction, the script opens the cursor (using the previously allocated handle) and fetches rows until the end of the record stream is reached. Data values are displayed for each fetched record, and the cursor is closed once all records have been processed. Finally, the read-only transaction is rolled back, resources associated with the cursor handle are freed, and the script detaches from the database by calling `rdb_detach()`. Note that the transaction could equally be committed via a call to `rdb_commit()`, however strictly speaking there is no actual work to commit.

```
<?php
if (! extension_loaded('rdb')) {
    if (! dl('rdb.exe')) {
        exit;
    }
}

if (rdb_attach('sql$database') == -1) {
    printf("%s\n", rdb_error());
    exit;
}

$cursor = 'C0001';
$ch = rdb_declare_cursor($cursor, 'select employee_id,last_name,first_name from employees');

if ($ch == NULL) {
    printf("%s\n", rdb_error());
    exit;
}

rdb_set_readonly();

if (rdb_open_cursor($ch) == -1) {
    printf("%s\n", rdb_error());
    exit;
}

while (($row = rdb_fetch_row($ch)) != NULL) {
    print_r(array_values($row));
}

if (rdb_sqlcode() != 100) {
    printf("%s\n", rdb_error());
    exit;
}
```

```

if (rdb_close_cursor($ch) == -1) {
    printf("%s\n", rdb_error());
    exit;
}

rdb_rollback();

rdb_free($ch);
rdb_detach();
?>

```

Note that all functions return either an integer completion code, a cursor or statement handle, or a result set. In the event of an error, functions that return an integer completion code will return a value of -1 to indicate that an error has occurred, and other functions will return a NULL value in the event of an error. Whenever an error is returned, the function `rdb_error()` can be used to get a textual description of the last error and the function `rdb_sqlcode()` can be used to get the integer SQL error code. It should also be noted that the error handling in the above example could be improved by ensuring that any outstanding transaction is committed or rolled back, and resources are properly cleaned up before existing following an error. Similar comments apply to other example code included in this document.

The above example uses the `rdb_fetch_row()` function to retrieve a row of data at a time, loading values for each column into an array. This is arguably the most convenient method of fetching data; however it is also possible to retrieve individual column values by using the `rdb_fetch()` and `rdb_data()` functions as illustrated below, where the call to `rdb_fetch_row()` in the above example has been replaced by a call to `rdb_fetch()` and a loop that uses `rdb_data()` to retrieve the values for each column for each row that is fetched. Using `rdb_fetch_row()` is simpler and slightly more efficient; however, there may be situations where this alternative approach may be more appropriate.

```

while (rdb_fetch($ch) == 1) {
    for ($i = 0; $i < 3; $i++) {
        $val = rdb_data($ch, $i);
        echo "\t$val";
    }
    echo "\n";
}

```

The next example illustrates use of the `rdb_execi()` (execute immediate) function to execute a dynamic SQL update statement against the database. Note here that it would generally be good practice to explicitly set the transaction scope before performing the update, reserving only the `employees` table for write operations. By not explicitly setting the scope of the transaction, RDB will start a default read/write transaction reserving all tables in the database for read/write operations, which may cause contention problems if there are multiple database users, and will reserve more resources than necessary for the transaction in question. An explicit scope for the transaction may be established by calling the `rdb_execi()` function with an appropriate SQL “set transaction” statement.

```

<?php
if (! extension_loaded('rdb')) {
    if (! dl('rdb.exe')) {
        exit;
    }
}

rdb_attach('sql$database');

if (rdb_execi('update employees set city=\'Bolton\' where employee_id=\'00249\'') == -1) {
    printf("%s\n", rdb_error());
    exit;
}

rdb_commit();
rdb_detach();
?>

```

Note that it is also possible to prepare SQL statements (using the `rdb_prepare()` function and execute them using `rdb_exec()`); however, the RDB PHP API currently provides no mechanism to specify parameter markers and parameter values. It is intended that this functionality will be included in a subsequent release of the RDB API. From a PHP coding perspective, the lack of such functionality is perhaps of little consequence; however, from a performance perspective it can be advantageous to prepare frequently used SQL statements and it is therefore desirable to have this functionality in the future.

Mimer API

The Mimer database (see <http://www.mimer.com/>) has existed on OpenVMS for almost longer than any other relational database and represents a viable alternative to some of the more commonly known databases for the OpenVMS operating system platform, providing excellent performance and stability with a small footprint and low administrative overhead.

The following table summarizes the functions provided by the Mimer PHP API. As for the RDB API, the set of functions provided is deliberately minimal, providing a basic set of functionalities that should be sufficient for most application needs.

| Function | Description |
|-------------------------------------|--|
| <code>mimerdb_connect()</code> | Connects to the specified database. |
| <code>mimerdb_disconnect()</code> | Disconnects from the database. |
| <code>mimerdb_exec()</code> | Executes the supplied SQL statement. |
| <code>mimerdb_fetch()</code> | Fetches a row of data using the specified cursor. |
| <code>mimerdb_close()</code> | Closes the specified cursor. |
| <code>mimerdb_error()</code> | Returns a description of the last error. |
| <code>mimerdb_ncol()</code> | Returns the number of columns (values) that would be returned by a fetch for the specified cursor. |
| <code>mimerdb_commit()</code> | Commits the current database transaction. |
| <code>mimerdb_rollback()</code> | Rolls back the current database transaction. |
| <code>mimerdb_set_readonly()</code> | Starts a read-only transaction. |
| <code>mimerdb_sqlcode()</code> | Returns the SQLCODE for the last database operation. |

The functions provided by the Mimer API are also very similar in terms of operation to those that are provided by the RDB API; however, there are a few differences that are hopefully illustrated by the following example.

In particular, it should be noted that there are no specific functions to declare, open, and close database cursors. The `mimerdb_exec()` function determines from the provided SQL statement whether the operation will return any rows or not and acts accordingly, returning a value of 0 for success when the statement returns no result set, a positive integer sequence number when there is a result set, and a value of -1 to indicate that an error has occurred. If a positive integer is returned by `mimerdb_exec()` this implies that a cursor has been created and opened, and the result set may be retrieved using `mimerdb_fetch()`, which retrieves data into an array a row at a time. Note that the function `mimerdb_close()` should only be called when `mimerdb_exec()` returns a positive integer value (when there is a result set).

```

<?php
    if (! extension_loaded('mimerdb')) {
        if (! dl('mimerdb.exe')) {
            exit;
        }
    }

    $rv = mimerdb_connect('mydb', 'SYSADM', 'password');
    if ($rv != 1) {
        printf("%s\n", mimerdb_error());
        exit;
    }

    mimerdb_set_readonly();

    $ch = mimerdb_exec('select * from metric_groups');
    if ($ch == -1) {
        printf("%s\n", mimerdb_error());
        exit;
    }

    while (($row = mimerdb_fetch($ch)) != NULL) {
        print_r(array_values($row));
    }

    if (mimerdb_sqlcode() != 100) {
        printf("%s\n", mimerdb_error());
        exit;
    }

    mimerdb_close($ch);
    mimerdb_rollback();
    mimerdb_disconnect();
?>

```

With the exception of the `mimerdb_fetch()` and `mimerdb_error()` functions, all functions return an integer value that can be used to determine whether the operation in question was successful or not. A return value of -1 indicates that an error occurred, whereupon the function `mimerdb_error()` may be used to retrieve a textual description of the error and the function `mimerdb_sqlcode()` can be used to determine the integer SQL status code associated with the error in question. The function `mimerdb_fetch()` returns an array upon successful completion or `NULL` if an error occurred.

As with the Oracle RDB PHP interface, the Mimer PHP interface can only be connected to one database at a time.

Redis API

Redis (<https://redis.io/>) is a powerful, efficient, and functionally rich open source in-memory data structure store that can be used as used as an in-memory database, cache, and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, and geospatial indexes with radius queries. Redis is commonly used to improve the performance of websites by caching in memory frequently accessed static data; however, it is also applicable to a wide range of other use-cases. Redis has recently been ported to OpenVMS, and the inclusion of Redis client functionality in the PHP distribution for OpenVMS provided by VSI makes it readily possible to leverage Redis functionality from OpenVMS-based PHP applications.

As noted above, the PHP Redis extension is provided by the PhpRedis open source project (<https://github.com/phpredis/phpredis>) and the reader should refer to the PhpRedis documentation for full details regarding the use of the API; however it is perhaps useful to consider a few simple examples here.

The first example below illustrates the basics of connecting to the Redis cache and verifying that the cache is functioning by sending it a “PING” command, to which the cache should respond with the text “+PONG”. The example also calls the `info()` method to retrieve data about the cache, including

software version details and various operational metrics (for a detailed description of these data the reader should refer to the Redis documentation).

```
<?php
if (! extension_loaded('redis')) {
    if (! dl('redis.exe')) {
        exit;
    }
}

$redis = new Redis();

$redis->connect('127.0.0.1', 6379);
echo print_r($redis->info(), true);
echo $redis->ping();
$redis->close();
?>
```

The next example illustrates adding a simple key/value pair to the cache and the retrieval and deletion of the key (and associated data) from the cache. The code also illustrates the use of exception handling that is provided by the API to catch and manage unexpected error conditions.

```
<?php
if (! extension_loaded('redis')) {
    if (! dl('redis.exe')) {
        exit;
    }
}

$redis = new Redis();
try {
    $redis->connect('127.0.0.1', 6379);
    $redis->set('greeting', 'Hello!');
    $reply = $redis->get('greeting');

    if ($reply){
        echo "Reply: '{$reply}'\n";
        if ($redis->delete('greeting')){
            echo "Key deleted\n";
        }
    }else{
        echo "Key not found\n";
    }
} catch (RedisException $e){
    $exceptionMsg = $e->getMessage();
    echo "Houston we have a problem: {$exceptionMsg}\n";
}
?>
```

The following final example illustrates some basic Redis list operations, such as populating a list and retrieving list elements. Once again, refer to <https://github.com/phpredis/phpredis> for additional information.

```
<?php
if (! extension_loaded('redis')) {
    if (! dl('redis.exe')) {
        exit;
    }
}

// Connect to Redis on localhost
$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
echo "Connection to server successful";

// Store list data
$redis->lpush("language-list", "COBOL");
$redis->lpush("language-list", "FORTRAN");
$redis->lpush("language-list", "Pascal");

// Get the stored data and print it
$arList = $redis->lrange("language-list", 0 ,5);
print_r($arList);
```

```
$redis->delete("language-list");  
?>
```