Hierarchical Data Format query language (HDFql)

Reference Manual

Version 1.4.0

March 2017

Disclaimer

Every effort has been made to ensure that this document is as accurate as possible. The information contained in this document is provided without any express, statutory or implied warranties. The founders of HDFql shall have neither liability nor responsibility to any person or entity with respect to any loss or damage arising from the information in this document or the usage of HDFql.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

HDFql stands for "Hierarchical Data Format query language" and is the first tool that enables users to manage HDF[1] files through a high-level language. This language was designed to be simple to use and similar to SQL thus dramatically reducing the learning effort. HDFql can be seen as an alternative to the C API (which contains more than 400 low-level functions that are far from easy to use!) and to existing wrappers for C++, Java, Python, C# and Fortran for manipulating HDF files. Whenever possible, it automatically uses parallelism to speed-up operations hiding its inherent complexity from the user.

As an example, imagine that one needs to create an HDF file named "myFile.h5" and, inside it, a group named "myGroup" containing an attribute named "myAttribute" of type float with a value of 12.4. Using the C API, it could be implemented like this:

```
hid_t file;
hid_t group;
hid_t dataspace;
hid_t attribute;
hsize_t dimension;
float value;
H5Fcreate("myFile.h5", H5F_ACC_EXCL, H5P_DEFAULT, H5P_DEFAULT);
file = H5Fopen("myFile.h5", H5F_ACC_RDWR, H5P_DEFAULT);
group = H5Gcreate(file, "myGroup", H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
dimension = 1;
dataspace = H5Screate_simple(1, &dimension, NULL);
attribute = H5Acreate(group, "myAttribute", H5T_NATIVE_FLOAT, dataspace, H5P_DEFAULT,
H5P_DEFAULT);
value = 12.4;
H5Awrite(attribute, H5T_NATIVE_FLOAT, &value);
```

[1] Hierarchical Data Format is the name of a set of file formats and libraries designed to store and organize large amounts of numerical data. It is currently supported by the non-profit HDF Group, whose mission is to ensure continued development of HDF technologies and the continued accessibility of data currently stored in HDF. Please refer to the website http://www.hdfgroup.org for additional information.

In HDFql, the same example can easily be implemented just by doing this:

```
create file myFile.h5
use file myFile.h5
create group myGroup
create attribute myGroup/myAttribute as float default 12.4
```

2. INSTALLATION

The official website of the Hierarchical Data Format query language (HDFql) is http://www.hdfql.com. Here, the most recent documentation and examples that illustrate how to solve disparate use-cases using HDFql can be found. In addition, in the download area (http://www.hdfql.com/download) all versions of HDFql ever publicly released are available. These versions are packaged as ZIP files, each one of them meant for a particular platform (i.e. Windows, Linux or Mac OS X), architecture (i.e. 32 bit or 64 bit) and compiler[1]. When decompressed, such ZIP files typically have the following organization in terms of directories and files contained within:

```
HDFql-x.y.z
     |
     + example (directory that contains examples in C, C++, Java, Python, C# and Fortran)
     |
     + include (directory that contains HDFql C and C++ header files)
     |
     + lib (directory that contains HDFql C release/debug static and shared libraries)
     |
     + bin (directory that contains HDFql command-line interface and a proper launcher)
     |
     + wrapper (directory that contains wrappers for C++, Java, Python, C# and Fortran)
     |
     + doc (directory that contains HDFql reference manual)
     |
     - LICENSE.txt (file that contains information about HDFql license)
     |
     - RELEASE.txt (file that contains information about HDFql releases)
     |
     - README.txt (file that contains succinct information about HDFql)
```

[1] At the time of writing, HDFql only supports Microsoft Visual Studio and Gnu Compiler Collection (GCC) compilers. Additional compilers will be supported in the near future, namely MinGW (http://www.mingw.org) and Clang (http://clang.llvm.org).

The following sections provide concise instructions on how to install HDFql in the different platforms that it currently supports – namely Windows, Linux and Mac OS X.

2.1 WINDOWS

- Download the appropriate ZIP file according to the HDFql version, architecture and compiler of interest from http://www.hdfql.com/download. For instance, if the HDFql version of interest is 1.0.0 and it is to be used in a machine running Windows 32 bit and, eventually, be linked against C or C++ code using the Microsoft Visual Studio 2010 compiler then the file to download is "HDFql-1.0.0_Windows32_VS-2010.zip".
- Unzip the downloaded file using Windows Explorer in-build capabilities or a free tool such as 7-Zip (http://www.7-zip.org).

2.2 LINUX

- Download the appropriate ZIP file according to the HDFql version, architecture and compiler of interest from http://www.hdfql.com/download. For instance, if the HDFql version of interest is 1.1.0 and it is to be used in a machine running Linux 64 bit and, eventually, be linked against C or C++ code using the GCC 4.8.x compiler then the file to download is "HDFql-1.1.0_Linux64_GCC-4.8.zip".
- Unzip the downloaded file using the Archive Manager or the KArchive (if in GNOME or KDE respectively), or by opening a terminal and executing "*unzip <downloaded_zip_file>*". If the unzip utility is not installed in the machine, it can be done by executing from a terminal:
    - In a Red Hat-based distribution, "*sudo yum install unzip*".
    - In a Debian-based distribution, "*sudo apt-get install unzip*".

2.3 MAC OS X

- Download the appropriate ZIP file according to the HDFql version, architecture and compiler of interest from http://www.hdfql.com/download. For instance, if the HDFql version of interest is 1.3.0 and it is to be used in a machine running Mac OS X 64 bit and, eventually, be linked against C or C++ code using the GCC 4.9.x compiler then the file to download is "HDFql-1.3.0_Darwin64_GCC-4.9.zip".

- Unzip the downloaded file using the Archive Utility or by opening a terminal and executing "*unzip <downloaded_zip_file>*". If the unzip utility is not installed in the machine, it can be done by executing "*sudo port install unzip*" from a terminal.

3. USAGE

After following the instructions provided in chapter INSTALLATION, HDFql is ready for usage. It can be used in C through static and shared libraries; in C++, Java, Python, C# and Fortran through wrappers; and finally, through a command-line interface named “HDFqlCLI”. The subsequent sections provide guidance on usage and basic troubleshooting information to solve issues that may arise.

3.1 C

HDFql can be used in the C programming language through static and shared libraries. These libraries are stored in the directory “lib”. The following short program illustrates how HDFql can be used in such language.

```
// include HDFql C header file (make sure it can be found by the C compiler)
#include "HDFql.h"

int main(int argc, char *argv[])
{
    // display HDFql version in use
    printf("HDFql version: %s\n", HDFQL_VERSION);

    // create an HDF file named "my_file.h5"
    hdfql_execute("CREATE FILE my_file.h5");

    // use (i.e. open) HDF file "my_file.h5"
    hdfql_execute("USE FILE my_file.h5");

    // create a dataset named "my_dataset" of type int
    hdfql_execute("CREATE DATASET my_dataset AS INT");

    return 0;
}
```

Assuming that the program is stored in a file named "example.c", it must first be compiled before it can be launched from a terminal. To compile the program against the HDFql C static library:

- In Microsoft Visual Studio, by executing "*cl.exe example.c /I<hdfql_include_directory> <hdfql_lib_directory>\HDFql.lib /link /LTCG /NODEFAULTLIB:libcmt.lib*" from a terminal.
- In Gnu Compiler Collection (GCC), by executing "*gcc example.c -I<hdfql_include_directory> <hdfql_lib_directory>/libHDFql.a -fopenmp -lm -ldl*" from a terminal.

To compile the same program against the HDFql C shared library:

- In Microsoft Visual Studio, by executing "*cl.exe example.c /I<hdfql_include_directory> <hdfql_lib_directory>\HDFql_dll.lib*" from a terminal.
- In Gnu Compiler Collection (GCC), by executing "*gcc example.c -I<hdfql_include_directory> -L<hdfql_lib_directory> -lHDFql -lm -ldl*" from a terminal.

Of note, debug versions of the HDFql C static and shared libraries are also available. These are stored in the directory "debug" found under the directory "lib". To compile C programs using debug libraries, the instructions described in the above bullet points should be followed with two modifications: (1) the directory storing the libraries should be updated ("<hdfql_lib_directory>\debug" in Microsoft Visual Studio; "<hdfql_lib_directory>/debug" in GCC); (2) the suffix "D" should be added to the name of the libraries ("HDFqlD.lib" and "HDFql_dllD.lib" in Microsoft Visual Studio; "libHDFqlD.a" and "libHDFqlD.so" in GCC).

In case the program does not compile, likely a C compiler is not installed in the machine. If a C compiler is missing, the solution is:

- In Windows, download and install a free version of Microsoft Visual Studio from the website https://www.visualstudio.com/downloads.
- In Linux, install the GCC C compiler by executing from a terminal:

    - In a Red Hat-based distribution, "*sudo yum install gcc*".
    - In a Debian-based distribution, "*sudo apt-get install gcc*".
- In Mac OS X, install the GCC C compiler by executing "*xcode-select --install*" from a terminal. If xcode-select does not support the parameter "--install" (due to being outdated), download and install the Command-Line Tools package from http://developer.apple.com/downloads which includes GCC.

In case the compiled program does not launch, most likely the HDFql C shared library (which is needed to launch the program) was not found. The solution is:

- In Windows, copy the file "HDFql_dll.dll" (stored in "<hdfql_lib_directory>") into the directory where the program is located. Alternatively, add the directory where the file "HDFql_dll.dll" is located to the environment variable "PATH" by executing "*set PATH=<hdfql_lib_directory>;%PATH%*" from a terminal.
- In Linux, add the directory where the file "libHDFql.so" is located to the environment variable "LD_LIBRARY_PATH" by executing from a terminal:
    - In Bash shell, "*export LD_LIBRARY_PATH=<hdfql_lib_directory>:$LD_LIBRARY_PATH*".
    - In C shell, "*setenv LD_LIBRARY_PATH <hdfql_lib_directory>:$LD_LIBRARY_PATH*".
- In Mac OS X, add the directory where the file "libHDFql.dylib" is located to the environment variable "DYLD_LIBRARY_PATH" by executing from a terminal:
    - In Bash shell, "*export DYLD_LIBRARY_PATH=<hdfql_lib_directory>:$DYLD_LIBRARY_PATH*".
    - In C shell, "*setenv DYLD_LIBRARY_PATH <hdfql_lib_directory>:$DYLD_LIBRARY_PATH*".

3.2 C++

HDFql can be used in the C++ programming language through static and shared libraries. These libraries are stored in the directory "cpp" found under the directory "wrapper". The following short program illustrates how HDFql can be used in such language.

```
// include HDFql C++ header file (make sure it can be found by the C++ compiler)
#include <iostream>
#include "HDFql.hpp"

int main(int argc, char *argv[])
{
    // display HDFql version in use
    std::cout << "HDFql version: " << HDFql::Version << std::endl;

    // create an HDF file named "my_file.h5"
    HDFql::execute("CREATE FILE my_file.h5");

    // use (i.e. open) HDF file "my_file.h5"
    HDFql::execute("USE FILE my_file.h5");

    // create a dataset named "my_dataset" of type int
    HDFql::execute("CREATE DATASET my_dataset AS INT");

    return 0;
}
```

Assuming that the program is stored in a file named “example.cpp”, it must first be compiled before it can be launched from a terminal. To compile the program against the HDFql C++ static library:

- In Microsoft Visual Studio, by executing “*cl.exe example.cpp /EHsc /I<hdfql_include_directory> <hdfql_cpp_wrapper_directory>\HDFql.lib /link /LTCG /NODEFAULTLIB:libcmt.lib*” from a terminal.
- In Gnu Compiler Collection (GCC), by executing “*g++ example.cpp -I<hdfql_include_directory> <hdfql_cpp_wrapper_directory>/libHDFql.a -fopenmp -ldl*” from a terminal.

To compile the same program against the HDFql C++ shared library:

- In Microsoft Visual Studio, by executing “*cl.exe example.cpp /EHsc /I<hdfql_include_directory> <hdfql_cpp_wrapper_directory>\HDFql_dll.lib*” from a terminal.
- In Gnu Compiler Collection (GCC), by executing “*g++ example.cpp -I<hdfql_include_directory> -L<hdfql_cpp_wrapper_directory> -lHDFql -ldl*” from a terminal.

In case the program does not compile, likely a C++ compiler is not installed in the machine. If a C++ compiler is missing, the solution is:

- In Windows, download and install a free version of Microsoft Visual Studio from the website https://www.visualstudio.com/downloads.
- In Linux, install the GCC C++ compiler by executing from a terminal:
    - In a Red Hat-based distribution, "*sudo yum install gcc-c++*".
    - In a Debian-based distribution, "*sudo apt-get install g++*".
- In Mac OS X, install the GCC C++ compiler by executing "*xcode-select --install*" from a terminal. If xcode-select does not support the parameter "--install" (due to being outdated), download and install the Command-Line Tools package from http://developer.apple.com/downloads which includes GCC.

In case the compiled program does not launch, most likely the HDFql C++ shared library (which is needed to launch the program) was not found. The solution is:

- In Windows, copy the file "HDFql_dll.dll" (stored in "<hdfql_cpp_wrapper_directory>") into the directory where the program is located. Alternatively, add the directory where the file "HDFql_dll.dll" is located to the environment variable "PATH" by executing "*set PATH=<hdfql_cpp_wrapper_directory>;%PATH%*" from a terminal.
- In Linux, add the directory where the file "libHDFql.so" is located to the environment variable "LD_LIBRARY_PATH" by executing from a terminal:
    - In Bash shell, "*export LD_LIBRARY_PATH=<hdfql_cpp_wrapper_directory>:$LD_LIBRARY_PATH*".
    - In C shell, "*setenv LD_LIBRARY_PATH <hdfql_cpp_wrapper_directory>:$LD_LIBRARY_PATH*".
- In Mac OS X, add the directory where the file "libHDFql.dylib" is located to the environment variable "DYLD_LIBRARY_PATH" by executing from a terminal:

- In Bash shell, "*export DYLD_LIBRARY_PATH=<hdfql_cpp_wrapper_directory>:$DYLD_LIBRARY_PATH*".
- In C shell, "*setenv DYLD_LIBRARY_PATH <hdfql_cpp_wrapper_directory>:$DYLD_LIBRARY_PATH*".

3.3 JAVA

HDFql can be used in the Java programming language through a wrapper named "HDFql.java". This wrapper is stored in the directory "java" found under the directory "wrapper". The following short program illustrates how HDFql can be used in such language.

```
public class Example
{
    public static void main(String args[])
    {
        // load HDFql shared library (make sure it can be found by the JVM)
        System.loadLibrary("HDFql");

        // display HDFql version in use
        System.out.println("HDFql version: " + HDFql.VERSION);

        // create an HDF file named "my_file.h5"
        HDFql.execute("CREATE FILE my_file.h5");

        // use (i.e. open) HDF file "my_file.h5"
        HDFql.execute("USE FILE my_file.h5");

        // create a dataset named "my_dataset" of type int
        HDFql.execute("CREATE DATASET my_dataset AS INT");
    }
}
```

Assuming that the program is stored in a file named "Example.java", it must first be compiled by executing "*javac Example.java*" before it can be launched by executing "*java Example*" from a terminal. In case the program does not compile or launch, likely the Java Development Kit (JDK) is not installed in the machine or the HDFql Java wrapper was not found. For the former, install the JDK by following the instructions available at

http://www.oracle.com/technetwork/java/javase/downloads. For the latter, add the directory where the file "HDFql.java" (i.e. the wrapper) is located to the environment variables "CLASSPATH" and, depending on the platform, "PATH", "LD_LIBRARY_PATH" or "DYLD_LIBRARY_PATH":

- In Windows, by executing "*set CLASSPATH=<hdfql_java_wrapper_directory>;.;%CLASSPATH%*" and "*set PATH=<hdfql_java_wrapper_directory>;%PATH%*" from a terminal.
- In Linux, by executing from a terminal:
    - In Bash shell, "*export CLASSPATH=<hdfql_java_wrapper_directory>:.:$CLASSPATH*" and "*export LD_LIBRARY_PATH=<hdfql_java_wrapper_directory>:$LD_LIBRARY_PATH*".
    - In C shell, "*setenv CLASSPATH <hdfql_java_wrapper_directory>:.:$CLASSPATH*" and "*setenv LD_LIBRARY_PATH <hdfql_java_wrapper_directory>:$LD_LIBRARY_PATH*".
- In Mac OS X, by executing from a terminal:
    - In Bash shell, "*export CLASSPATH=<hdfql_java_wrapper_directory>:.:$CLASSPATH*" and "*export DYLD_LIBRARY_PATH=<hdfql_java_wrapper_directory>:$DYLD_LIBRARY_PATH*".
    - In C shell, "*setenv CLASSPATH <hdfql_java_wrapper_directory>:.:$CLASSPATH*" and "*setenv DYLD_LIBRARY_PATH <hdfql_java_wrapper_directory>:$DYLD_LIBRARY_PATH*".

3.4 PYTHON

HDFql can be used in the Python programming language through a wrapper named "HDFql.py". This wrapper is stored in the directory "python" found under the directory "wrapper". The following short script illustrates how HDFql can be used in such language.

```
# import HDFql module (make sure it can be found by the Python interpreter)
import HDFql

# display HDFql version in use
print("HDFql version: %s" % HDFql.VERSION)

# create an HDF file named "my_file.h5"
```

```
HDFql.execute("CREATE FILE my_file.h5")

# use (i.e. open) HDF file "my_file.h5"
HDFql.execute("USE FILE my_file.h5")

# create a dataset named "my_dataset" of type int
HDFql.execute("CREATE DATASET my_dataset AS INT")
```

Assuming that the script is stored in a file named “example.py” it can be launched by executing “*python example.py*” from a terminal. In case the script does not launch, likely the Python interpreter is not installed in the machine or the HDFql Python wrapper was not found. For the former, install the Python interpreter by following the instructions available at http://www.python.org/download. For the latter, add the directory where the file “HDFql.py” (i.e. the wrapper) is located to the environment variable “PYTHONPATH”:

- In Windows, by executing “*set PYTHONPATH=<hdfql_python_wrapper_directory>;%PYTHONPATH%*” from a terminal.
- In Linux/Mac OS X, by executing from a terminal:
    - In Bash shell, “*export PYTHONPATH=<hdfql_python_wrapper_directory>:$PYTHONPATH*”.
    - In C shell, “*setenv PYTHONPATH <hdfql_python_wrapper_directory>:$PYTHONPATH*”.

Besides these steps, a scientific computing package named NumPy must be installed when working with user-defined variables (please refer to the function hdfql_variable_register for additional information). NumPy can be found at http://www.scipy.org/scipylib/download.html along with instructions on how to install it.

3.5 C#

HDFql can be used in the C# programming language through a wrapper named “HDFql.cs”. This wrapper is stored in the directory “csharp” found under the directory “wrapper”. The following short program illustrates how HDFql can be used in such language.

```
public class Example
{
    public static void Main(string []args)
    {
        // display HDFql version in use
        System.Console.WriteLine("HDFql version: {0}", HDFql.Version);

        // create an HDF file named "my_file.h5"
        HDFql.Execute("CREATE FILE my_file.h5");

        // use (i.e. open) HDF file "my_file.h5"
        HDFql.Execute("USE FILE my_file.h5");

        // create a dataset named "my_dataset" of type int
        HDFql.Execute("CREATE DATASET my_dataset AS INT");
    }
}
```

Assuming that the program is stored in a file named “Example.cs”, it must first be compiled before it can be launched from a terminal. In Windows, the program can be compiled as follows:

- In Microsoft .NET Framework, by executing “*csc.exe <hdfql_csharp_wrapper_directory>*.cs Example.cs*” from a terminal.
- In Mono, by executing “*mcs.bat <hdfql_csharp_wrapper_directory>*.cs Example.cs*” from a terminal.

In Linux and Mac OS X, the program can be compiled in Mono by executing “*mcs <hdfql_csharp_wrapper_directory>/*.cs Example.cs*” from a terminal (of note, Microsoft .NET Framework does not support these platforms).

In case the program does not compile, likely a C# compiler is not installed in the machine. If a C# compiler is missing, the solution is:

- In Windows, download and install either Microsoft .NET Framework or Mono from the websites https://www.microsoft.com/net/download/framework or http://www.mono-project.com/download, respectively.
- In Linux and Mac OS X, download and install Mono from the website http://www.mono-project.com/download.

Depending on the platform, the compiled program may be launched as follows:

- In Windows by:
    - Executing "*Example.exe*" from a terminal if it was compiled in Microsoft .NET Framework.
    - Executing "*mono.exe Example.exe*" from a terminal if it was compiled in Mono.
- In Linux and Mac OS X by executing "*mono Example.exe*" from a terminal.

In case the compiled program does not launch, most likely the HDFql C# wrapper (which is needed to launch the program) was not found. The solution is to add the directory where the file "HDFql.cs" (i.e. the wrapper) is located to the environment variable "PATH", "LD_LIBRARY_PATH" or "DYLD_LIBRARY_PATH" (depending on the platform):

- In Windows, by executing "*set PATH=<hdfql_csharp_wrapper_directory>;%PATH%*" from a terminal.
- In Linux, by executing from a terminal:
    - In Bash shell, "*export LD_LIBRARY_PATH=<hdfql_csharp_wrapper_directory>:$LD_LIBRARY_PATH*".
    - In C shell, "*setenv LD_LIBRARY_PATH <hdfql_csharp_wrapper_directory>:$LD_LIBRARY_PATH*".
- In Mac OS X, by executing from a terminal:
    - In Bash shell, "*export DYLD_LIBRARY_PATH=<hdfql_csharp_wrapper_directory>:$DYLD_LIBRARY_PATH*".

- In C shell, "*setenv DYLD_LIBRARY_PATH <hdfql_csharp_wrapper_directory>:$DYLD_LIBRARY_PATH*".

3.6 FORTRAN

HDFql can be used in the Fortran programming language through static and shared libraries. These libraries are stored in the directory "fortran" found under the directory "wrapper". The following short program illustrates how HDFql can be used in such language.

```
PROGRAM Example
    ! use HDFql module (make sure it can be found by the Fortran compiler)
    USE HDFql

    ! declare variable
    INTEGER :: state

    ! display HDFql version in use
    WRITE(*, *) "HDFql version: ", HDFQL_VERSION

    ! create an HDF file named "my_file.h5"
    state = hdfql_execute("CREATE FILE my_file.h5" // CHAR(0))

    ! use (i.e. open) HDF file "my_file.h5"
    state = hdfql_execute("USE FILE my_file.h5" // CHAR(0))

    ! create a dataset named "my_dataset" of type int
    state = hdfql_execute("CREATE DATASET my_dataset AS INT" // CHAR(0))
END PROGRAM
```

Assuming that the program is stored in a file named "example.f90", it must first be compiled before it can be launched from a terminal. To compile the program against the HDFql Fortran static library:

- In Gnu Compiler Collection (GCC), by executing "*gfortran example.f90 -I<hdfql_fortran_wrapper_directory> <hdfql_fortran_wrapper_directory>/libHDFql.a -fopenmp -ldl*" from a terminal.

To compile the same program against the HDFql Fortran shared library:

- In Gnu Compiler Collection (GCC), by executing "*gfortran example.f90 -I<hdfql_fortran_wrapper_directory> -L<hdfql_fortran_wrapper_directory> -lHDFql -ldl*" from a terminal.

In case the program does not compile, likely a Fortran compiler is not installed in the machine. If a Fortran compiler is missing, the solution is:

- In Linux, install the GCC Fortran compiler by executing from a terminal:
    - In a Red Hat-based distribution, "*sudo yum install gcc-gfortran*".
    - In a Debian-based distribution, "*sudo apt-get install gfortran*".
- In Mac OS X, install the GCC Fortran compiler by executing "*xcode-select --install*" from a terminal. If xcode-select does not support the parameter "--install" (due to being outdated), download and install the Command-Line Tools package from http://developer.apple.com/downloads which includes GCC.

Of note, an incorrect warning is raised by the GCC Fortran compiler when using the HDFql module ("Warning: Only array FINAL procedures declared for derived type 'hdfql_cursor' defined at (1), suggest also scalar one"). This warning does not interfere with the final compilation result, though, and it has been solved in the GCC Fortran compiler version 7.0.0 (please refer to https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58175 for additional information).

In case the compiled program does not launch, most likely the HDFql Fortran shared library (which is needed to launch the program) was not found. The solution is:

- In Linux, add the directory where the file "libHDFql.so" is located to the environment variable "LD_LIBRARY_PATH" by executing from a terminal:
    - In Bash shell, "*export LD_LIBRARY_PATH=<hdfql_fortran_wrapper_directory>:$LD_LIBRARY_PATH*".
    - In C shell, "*setenv LD_LIBRARY_PATH <hdfql_fortran_wrapper_directory>:$LD_LIBRARY_PATH*".

- In Mac OS X, add the directory where the file “libHDFql.dylib” is located to the environment variable “DYLD_LIBRARY_PATH” by executing from a terminal:
    - In Bash shell, “*export DYLD_LIBRARY_PATH=<hdfql_fortran_wrapper_directory>:$DYLD_LIBRARY_PATH*”.
    - In C shell, “*setenv DYLD_LIBRARY_PATH <hdfql_fortran_wrapper_directory>:$DYLD_LIBRARY_PATH*”.

3.7 COMMAND-LINE INTERFACE

A command-line interface named “HDFqlCLI” is available and can be used for manipulating HDF files. It is stored in the directory “bin”. To launch the command-line interface, open a terminal (“cmd” if in Windows, “xterm” if in Linux, or “Terminal” if in Mac OS X), go to the directory “bin”, and type “*HDFqlCLI*” (if in Windows) or “*./HDFqlCLI*” (if in Linux/Mac OS X). The list of parameters accepted by the command-line interface can be viewed by launching it with the parameter “--help”. At the time of writing, this list includes the following parameters:

- --help (show the list of parameters accepted by HDFqlCLI)
- --version (show the version of HDFqlCLI)
- --mac-address (show the MAC address(es) of the machine)
- --debug (show debug information when executing HDFql operations)
- --no-path (do not show group path currently in use in HDFqlCLI prompt)
- --execute=X (execute HDFql operation(s) “X" and exit)
- --execute-file=X (execute HDFql operation(s) stored in file “X” and exit)
- --save-file=X (save executed HDFql operation(s) to file “X”)

In case the command-line interface does not launch, most likely the HDFql shared library (which is needed to launch the interface) was not found. Depending on the platform, the solution is:

- In Windows, to either:
    - Copy the file "HDFql_dll.dll" (stored in "<hdfql_lib_directory>") into the directory where the command-line interface is located.
    - Add the directory where the file "HDFql_dll.dll" is located to the environment variable "PATH" by executing "*set PATH=<hdfql_lib_directory>;%PATH%*" from a terminal.
    - Execute the batch file named "launch.bat" which properly sets up the environment variable "PATH" and launches the command-line interface from a terminal.
- In Linux, to either:
    - Add the directory where the file "libHDFql.so" is located to the environment variable "LD_LIBRARY_PATH" by executing from a terminal:
        - In Bash shell, "*export LD_LIBRARY_PATH=<hdfql_lib_directory>:$LD_LIBRARY_PATH*".
        - In C shell, "*setenv LD_LIBRARY_PATH <hdfql_lib_directory>:$LD_LIBRARY_PATH*".
    - Execute the bash script file named "launch.sh" which properly sets up the environment variable "LD_LIBRARY_PATH" and launches the command-line interface from a terminal.
- In Mac OS X, to either:
    - Add the directory where the file "libHDFql.dylib" is located to the environment variable "DYLD_LIBRARY_PATH" by executing from a terminal:
        - In Bash shell, "*export DYLD_LIBRARY_PATH=<hdfql_lib_directory>:$DYLD_LIBRARY_PATH*".
        - In C shell, "*setenv DYLD_LIBRARY_PATH <hdfql_lib_directory>:$DYLD_LIBRARY_PATH*".
    - Execute the bash script file named "launch.sh" which properly sets up the environment variable "DYLD_LIBRARY_PATH" and launches the command-line interface from a terminal.

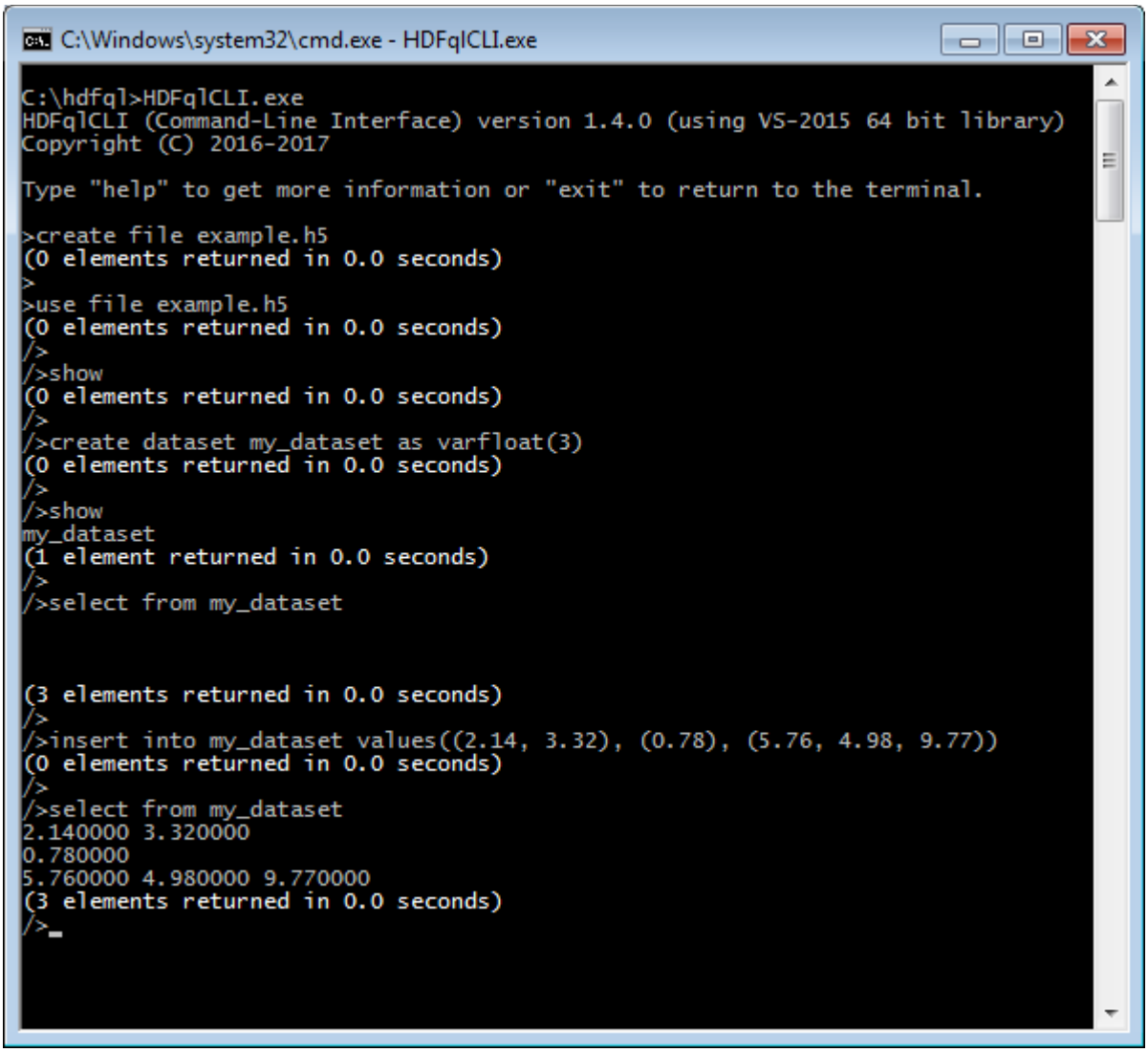

Figure 3.1 – Illustration of the command-line interface “HDFqlCLI”

4. CURSOR

Generally speaking, a cursor is a control structure that is used to iterate through the results returned by a query (that was previously executed). It can be seen as an effective means to abstract the programmer from low-level implementation details of accessing data stored in specific structures. This chapter provides a description of cursors and subcursors in HDFql, as well as examples and illustrations to demonstrate these two concepts in practice.

4.1 DESCRIPTION

HDFql provides cursors which offer several ways to traverse result sets according to specific needs. The following list enumerates these functionalities (please refer to their links for further information):

- First (moves cursor to the first position within the result set – hdfql_cursor_first)
- Last (moves cursor to the last position within the result set – hdfql_cursor_last)
- Next (moves cursor to the next position within the result set – hdfql_cursor_next)
- Previous (moves cursor to the previous position within the result set – hdfql_cursor_previous)
- Absolute (moves cursor to an absolute position within the result set – hdfql_cursor_absolute)
- Relative (moves cursor to a relative position within the result set – hdfql_cursor_relative)

Besides their traversal functionalities, a particular feature of cursors in HDFql is that they store result sets returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations. To retrieve values from result sets, the functions starting with “hdfql_cursor_get” can be used. These and remaining functions offered by cursors can be found in Table 5.7 (each of these begins with the prefix “hdfql_cursor”).

When a certain operation is executed, HDFql stores the result set returned by this operation in its default cursor. This cursor is available to the programmer and is automatically created and initialized upon loading the HDFql library by a program. If additional cursors are needed, they can be created like this:

```
// create a cursor named "my_cursor"
HDFQL_CURSOR my_cursor;
```

Before a cursor can be used to store and eventually traverse a result set, it must be properly initialized (refer to the function hdfql_cursor_initialize for further information). Initializing a cursor can be done like this:

```
// initialize a cursor named "my_cursor"
hdfql_cursor_initialize(&my_cursor);
```

To switch between different cursors (to be used for separate needs), the function hdfql_cursor_use may be employed:

```
// use a cursor named "my_cursor"
hdfql_cursor_use(&my_cursor);
```

The following C snippet illustrates usage of the HDFql default cursor and a user-defined cursor, as well as some typical operations performed on/by these.

```
// create a cursor named "my_cursor"
HDFQL_CURSOR my_cursor;

// create datasets named "my_dataset0" and "my_dataset1" of type float
hdfql_execute("CREATE DATASET my_dataset0 AS FLOAT");
hdfql_execute("CREATE DATASET my_dataset1 AS FLOAT(4, 2)");

// select (i.e. read) dataset "my_dataset0" and populate HDFql default cursor with it
hdfql_execute("SELECT FROM my_dataset0");

// initialize cursor "my_cursor" and use it
hdfql_cursor_initialize(&my_cursor);
hdfql_cursor_use(&my_cursor);
```

```
// select (i.e. read) dataset "my_dataset1" and populate cursor "my_cursor" with it
hdfql_execute("SELECT FROM my_dataset1");

// use HDFql default cursor and display its number of elements (should be 1)
hdfql_cursor_use(NULL);
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));

// use cursor "my_cursor" and display its number of elements (should be 8 - i.e. 4x2)
hdfql_cursor_use(&my_cursor);
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));

// display elements of cursor "my_cursor" (should display 8 elements)
while(hdfql_cursor_next(NULL) == HDFQL_SUCCESS)
{
    printf("Current element of cursor is %f\n", *hdfql_cursor_get_float(NULL));
}
```

When populating a cursor with data from a dataset or attribute with two or more dimensions, the data is always linearized into a single dimension. The linearization process is depicted in Figure 4.1. Subsequently, if need be, it is up to the programmer to access the data (stored in the cursor) according to its original dimensions. In this case, the SHOW [DATASET | ATTRIBUTE] DIMENSION operation – which returns the original dimensions of a dataset or attribute – may be useful to help in the task of going from one dimension to the original dimensions.

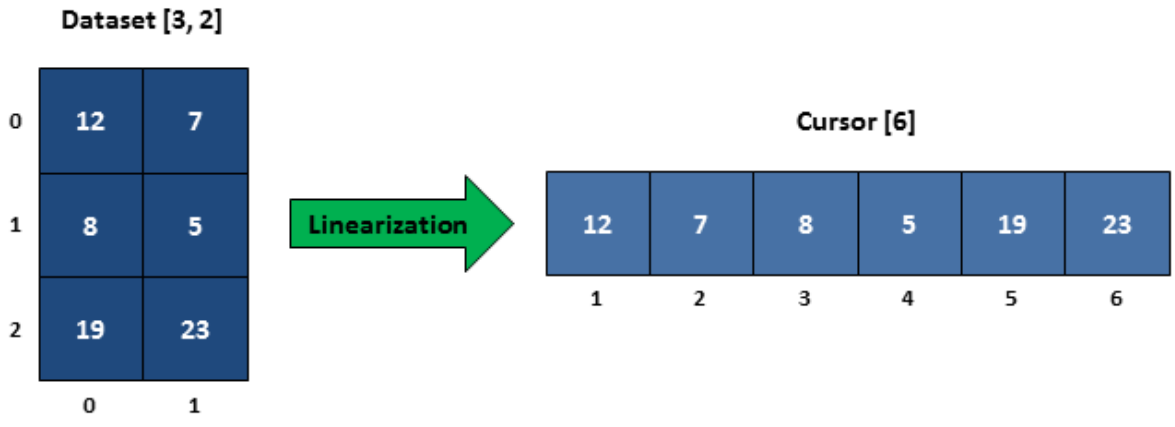

Figure 4.1 – Linearization of a two dimensional dataset into a (one dimensional) cursor

4.2 SUBCURSOR

HDFql also provides subcursors – they are meant to complement (i.e. help) cursors in the task of storing data of type variable-length (i.e. VARTINYINT, UNSIGNED VARTINYINT, VARSMALLINT, UNSIGNED VARSMALLINT, VARINT, UNSIGNED VARINT, VARBIGINT, UNSIGNED VARBIGINT, VARFLOAT, VARDOUBLE and VARCHAR). In practice, when a dataset or attribute of type variable-length is read through a DATA QUERY LANGUAGE (DQL) operation, only the first value of the variable data is stored in the cursor (as expected), while all values of the variable data are stored in the subcursor. In other words, each position of the cursor stores the first value of the variable data and also points to a subcursor that in turn stores all the values of the variable data. The values stored in a subcursor (which are also known as a result subset) can be accessed with the functions starting with "hdfql_subcursor_get" (enumerated in Table 5.7). Similar to cursors, HDFql subcursors offer several ways to traverse result subsets, namely:

- First (moves subcursor to the first position within the result subset – hdfql_subcursor_first)
- Last (moves subcursor to the last position within the result subset – hdfql_subcursor_last)
- Next (moves subcursor to the next position within the result subset – hdfql_subcursor_next)
- Previous (moves subcursor to the previous position within the result subset – hdfql_subcursor_previous)
- Absolute (moves subcursor to an absolute position within the result subset – hdfql_subcursor_absolute)
- Relative (moves subcursor to a relative position within the result subset – hdfql_subcursor_relative)

The following C snippet illustrates usage of the HDFql subcursors, as well as some typical operations performed on/by these.

```
// create a dataset named "my_dataset" of type variable-length int of one dimension (size
4)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(4)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((7, 8, 5, 3), (9), (6, 1, 2), (4, 0))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
```

```
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the next position within the result set (stored)
while(hdfql_cursor_next(NULL) == HDFQL_SUCCESS)
{
    // display elements of the cursor in use
    printf("Current element of cursor is %d\n", *hdfql_cursor_get_int(NULL));

    // move the subcursor in use to the next position within the result subset
    while(hdfql_subcursor_next(NULL) == HDFQL_SUCCESS)
    {
        // display elements of the subcursor in use
        printf("   Current element of subcursor is %d\n", *hdfql_subcursor_get_int(NULL));
    }
}
```

The output of executing the snippet would be similar to this:

```
Current element of cursor is 7
   Current element of subcursor is 7
   Current element of subcursor is 8
   Current element of subcursor is 5
   Current element of subcursor is 3
Current element of cursor is 9
   Current element of subcursor is 9
Current element of cursor is 6
   Current element of subcursor is 6
   Current element of subcursor is 1
   Current element of subcursor is 2
Current element of cursor is 4
   Current element of subcursor is 4
   Current element of subcursor is 0
```

4.3 EXAMPLES

The following C snippets demonstrate how HDFql cursors and subcursors are populated with (variable) data stored in datasets or attributes, along with illustrations to facilitate understanding of the populating process and its final result.

```
// create a dataset named "my_dataset0" of type short
hdfql_execute("CREATE DATASET my_dataset0 AS SMALLINT");

// insert (i.e. write) a value into dataset "my_dataset0"
hdfql_execute("INSERT INTO my_dataset0 VALUES(7)");

// select (i.e. read) dataset "my_dataset0" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset0");
```

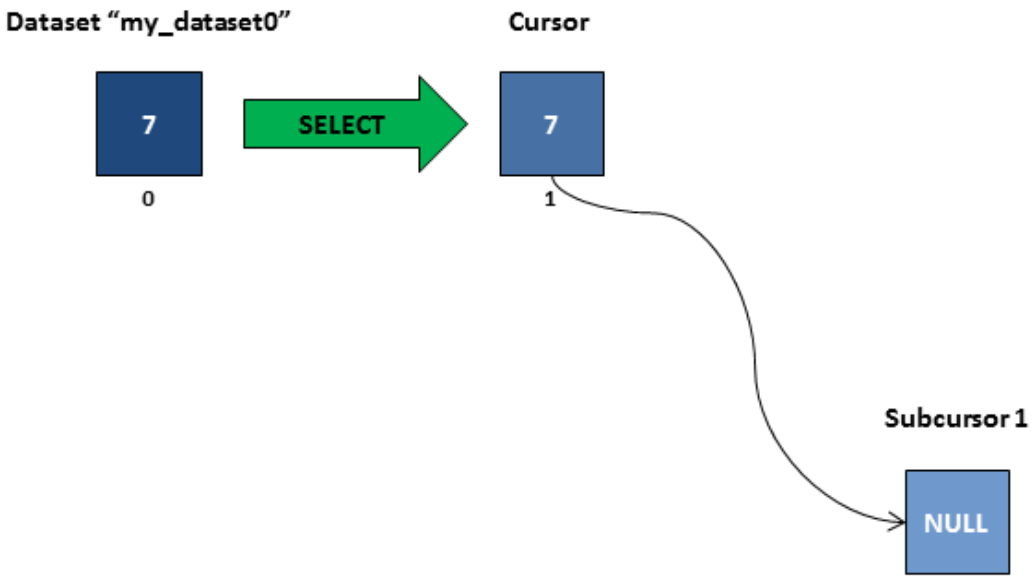

Figure 4.2 – Cursor populated with data from dataset “my_dataset0”

```
// create a dataset named "my_dataset1" of type float of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset1 AS FLOAT(3)");

// insert (i.e. write) values into dataset "my_dataset1"
hdfql_execute("INSERT INTO my_dataset1 VALUES(5.5, 8.1, 4.9)");

// select (i.e. read) dataset "my_dataset1" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset1");
```

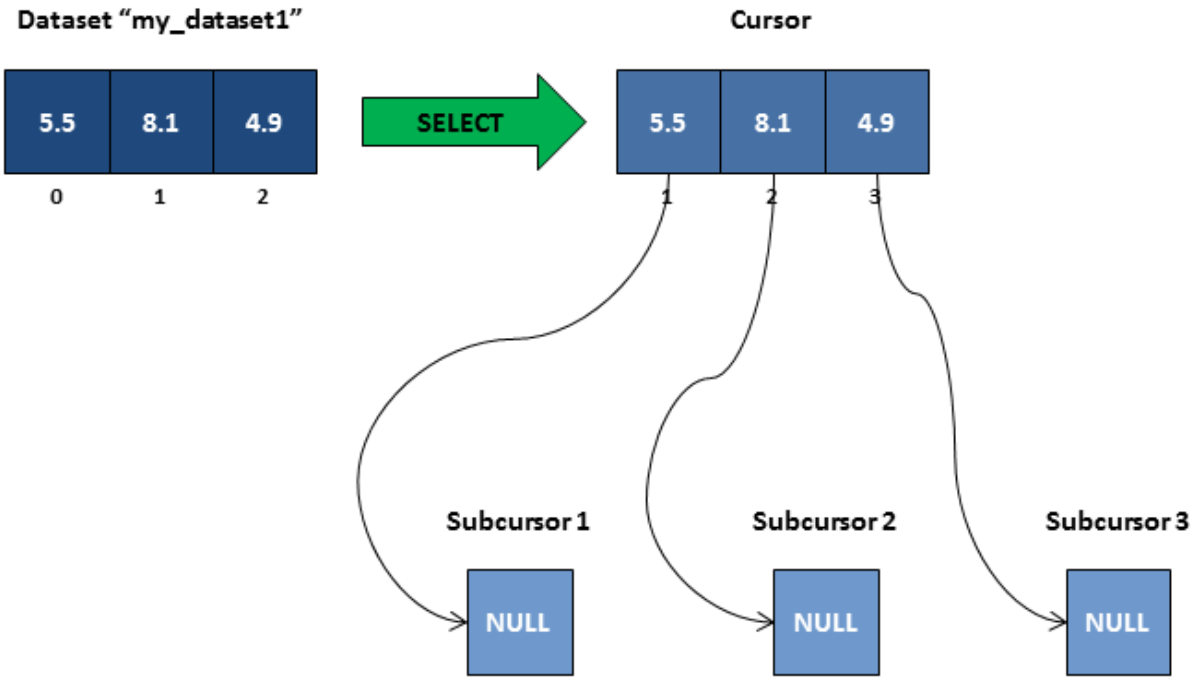

Figure 4.3 – Cursor populated with data from dataset "my_dataset1"

```
// create a dataset named "my_dataset2" of type double of two dimensions (size 3x2)
hdfql_execute("CREATE DATASET my_dataset2 AS DOUBLE(3, 2)");

// insert (i.e. write) values into dataset "my_dataset2"
hdfql_execute("INSERT INTO my_dataset2 VALUES((3.2, 1.3), (0, 0.2), (9.1, 6.5))");

// select (i.e. read) dataset "my_dataset2" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset2");
```

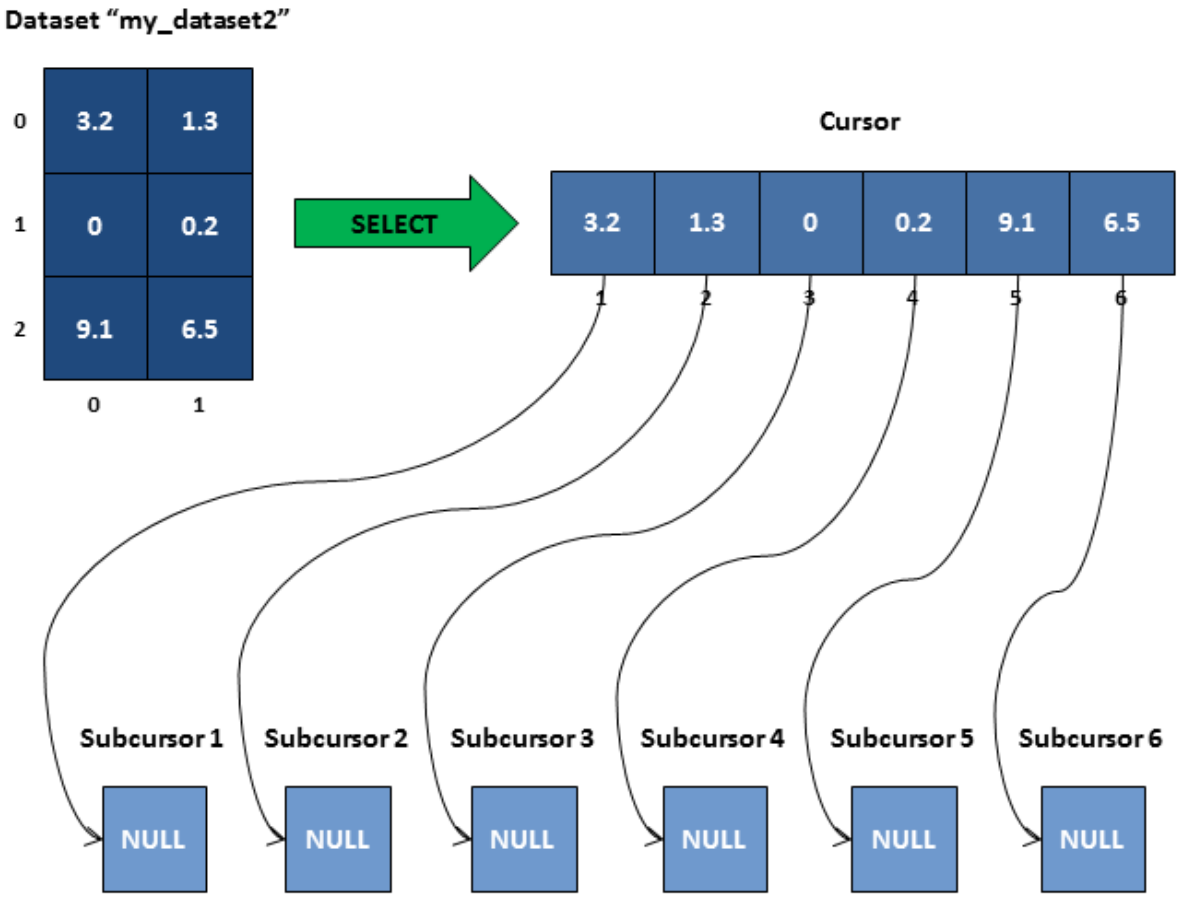

Figure 4.4 – Cursor populated with data from dataset "my_dataset2"

```
// create a dataset named "my_dataset3" of type variable-length short
hdfql_execute("CREATE DATASET my_dataset3 AS VARSMALLINT");

// insert (i.e. write) values into dataset "my_dataset3"
hdfql_execute("INSERT INTO my_dataset3 VALUES(7, 9, 3)");

// select (i.e. read) dataset "my_dataset3" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset3");
```

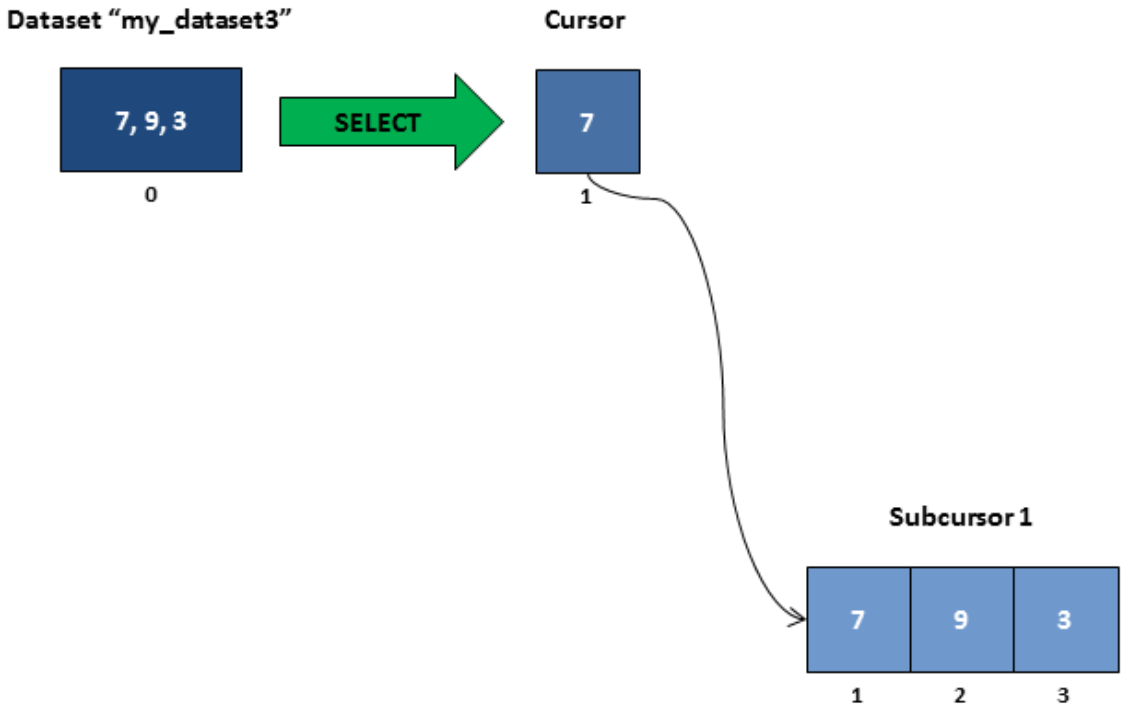

Figure 4.5 – Cursor and its subcursor populated with data from dataset “my_dataset3”

```
// create a dataset named "my_dataset4" of type variable-length float of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset4 AS VARFLOAT(3)");

// insert (i.e. write) values into dataset "my_dataset4"
hdfql_execute("INSERT INTO my_dataset4 VALUES((5.5), (8.1, 2.2), (4.9, 3.4, 5.6))");

// select (i.e. read) dataset "my_dataset4" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset4");
```

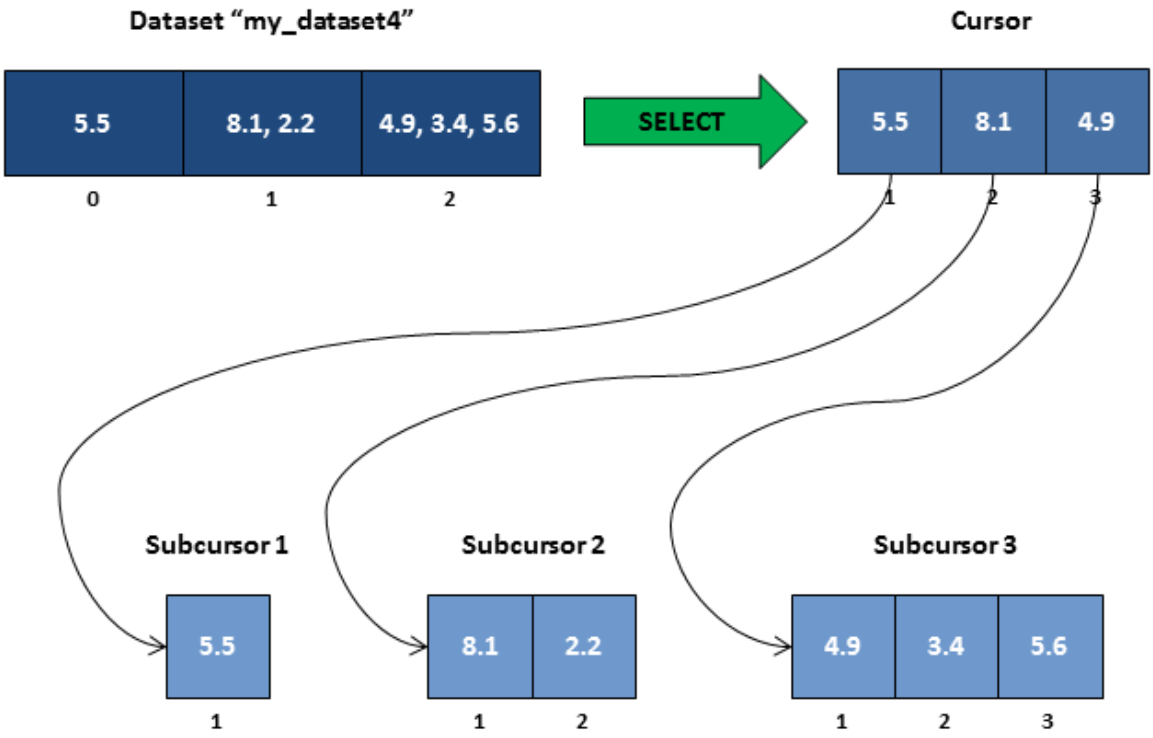

Figure 4.6 – Cursor and its subcursors populated with data from dataset "my_dataset4"

```
// create a dataset named "my_dataset5" of type variable-length double of two dimensions (size 3x2)
hdfql_execute("CREATE DATASET my_dataset5 AS VARDOUBLE(3, 2)");

// insert (i.e. write) values into dataset "my_dataset5"
hdfql_execute("INSERT INTO my_dataset5 VALUES(((3.2, 8, 6.7), (1.3, 0.2)), ((0), (0.2, 1.5)), ((9.1, 2, 4, 7), (6.5)))");

// select (i.e. read) dataset "my_dataset5" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset5");
```

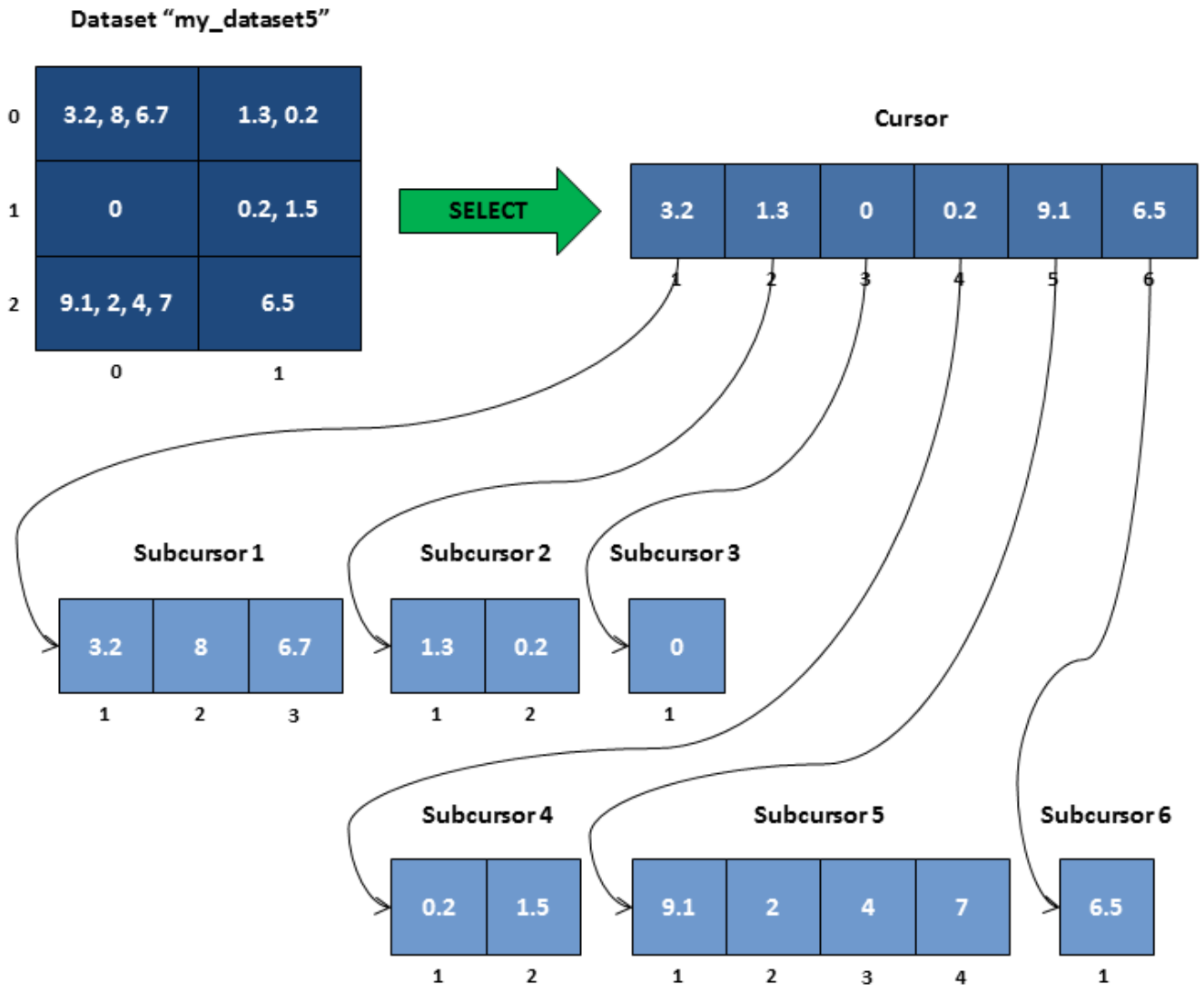

Figure 4.7 – Cursor and its subcursors populated with data from dataset "my_dataset5"

5. APPLICATION PROGRAMMING INTERFACE

An application programming interface (API) specifies how software components should interact with each other. In practice, an API comes in the form of a library that includes specifications for functions, data structures, object classes, constants and variables. A good API makes it easier to develop a program by providing all the building blocks. This chapter is devoted to describing HDFql API and how to use it through practical examples in C, C++, Java, Python, C# and Fortran.

5.1 CONSTANTS

A constant is an identifier whose associated value cannot typically be altered by the program during its execution. Using a constant instead of specifying a value multiple times in the program not only simplifies code maintenance, but can also supply a meaningful name for it. Constants in the C programming languages follow a naming convention of writing all words in uppercase and separating each word with an underscore (_). The following table summarizes all existing HDFql constants in C.

HDFql Constant in C	Description	Datatype	Value
HDFQL_VERSION	Represents the HDFql version in use	char *	1.4.0
HDFQL_YES	Represents the concept “Yes”	int	0
HDFQL_NO	Represents the concept “No”	int	-1
HDFQL_ENABLED	Represents the concept “Enabled”	int	0
HDFQL_DISABLED	Represents the concept “Disabled”	int	-1
HDFQL_UNDEFINED	Represents the concept “Undefined”	int	-1
HDFQL_TRACKED	Represents the HDF tracked creation order strategy	int	1
HDFQL_INDEXED	Represents the HDF indexed creation order strategy	int	2

HDFQL_DIRECTORY	Represents a directory	int	1
HDFQL_FILE	Represents a file	int	2
HDFQL_GROUP	Represents the HDF object type group	int	4
HDFQL_DATASET	Represents the HDF object type dataset	int	8
HDFQL_ATTRIBUTE	Represents the HDF object type attribute	int	16
HDFQL_SOFT_LINK	Represents the HDF soft link type	int	32
HDFQL_HARD_LINK	Represents the HDF hard link type	int	64
HDFQL_EXTERNAL_LINK	Represents the HDF external link type	int	128
HDFQL_CONTIGUOUS	Represents the HDF contiguous layout/strategy	int	1
HDFQL_COMPACT	Represents the HDF compact layout/strategy	int	2
HDFQL_CHUNKED	Represents the HDF chunked layout/strategy	int	4
HDFQL_TINYINT	Represents the tiny integer datatype (TINYINT)	int	1
HDFQL_UNSIGNED_TINYINT	Represents the unsigned tiny integer datatype (UNSIGNED TINYINT)	int	2
HDFQL_SMALLINT	Represents the small integer datatype (SMALLINT)	int	4
HDFQL_UNSIGNED_SMALLINT	Represents the unsigned small integer datatype (UNSIGNED SMALLINT)	int	8
HDFQL_INT	Represents the integer datatype (INT)	int	16
HDFQL_UNSIGNED_INT	Represents the unsigned integer datatype (UNSIGNED INT)	int	32
HDFQL_BIGINT	Represents the big integer datatype (BIGINT)	int	64
HDFQL_UNSIGNED_BIGINT	Represents the unsigned big integer datatype (UNSIGNED BIGINT)	int	128
HDFQL_FLOAT	Represents the float datatype (FLOAT)	int	256
HDFQL_DOUBLE	Represents the double datatype (DOUBLE)	int	512
HDFQL_CHAR	Represents the char datatype (CHAR)	int	1024
HDFQL_VARTINYINT	Represents the variable-length tiny integer datatype (VARTINYINT)	int	2048

HDFQL_UNSIGNED_VARTINYINT	Represents the unsigned variable-length tiny integer datatype (UNSIGNED VARTINYINT)	int	4096
HDFQL_VARSMALLINT	Represents the variable-length small integer datatype (VARSMALLINT)	int	8192
HDFQL_UNSIGNED_VARSMALLINT	Represents the unsigned variable-length small integer datatype (UNSIGNED VARSMALLINT)	int	16384
HDFQL_VARINT	Represents the variable-length integer datatype (VARINT)	int	32768
HDFQL_UNSIGNED_VARINT	Represents the unsigned variable-length integer datatype (UNSIGNED VARINT)	int	65536
HDFQL_VARBIGINT	Represents the variable-length big integer datatype (VARBIGINT)	int	131072
HDFQL_UNSIGNED_VARBIGINT	Represents the unsigned variable-length big integer datatype (UNSIGNED VARBIGINT)	int	262144
HDFQL_VARFLOAT	Represents the variable-length float datatype (VARFLOAT)	int	524288
HDFQL_VARDOUBLE	Represents the variable-length double datatype (VARDOUBLE)	int	1048576
HDFQL_VARCHAR	Represents the variable-length char datatype (VARCHAR)	int	2097152
HDFQL_OPAQUE	Represents the opaque datatype (OPAQUE)	int	4194304
HDFQL_NATIVE_ENDIAN	Represents the native architecture byte ordering	int	1
HDFQL_LITTLE_ENDIAN	Represents the little endian byte ordering	int	2
HDFQL_BIG_ENDIAN	Represents the big endian byte ordering	int	4
HDFQL_ASCII	Represents the ASCII character encoding	int	1
HDFQL_UTF8	Represents the UTF8 character encoding	int	2
HDFQL_SUCCESS	Represents an operation that succeeded	int	0
HDFQL_ERROR_PARSE	Represents an operation that failed due to a parsing error	int	-1
HDFQL_ERROR_NOT_FOUND	Represents an operation that failed due to an object (e.g. directory, file, group, dataset) not being found	int	-2
HDFQL_ERROR_NO_ACCESS	Represents an operation that failed due to an object	int	-3

	(e.g. directory, file, group, dataset) not being accessible		
HDFQL_ERROR_ALREADY_EXISTS	Represents an operation that failed due to an object (e.g. directory, file, group, dataset) already existing	int	-4
HDFQL_ERROR_EMPTY	Represents an operation that failed due to its internal structure being empty (cannot be processed further)	int	-5
HDFQL_ERROR_FULL	Represents an operation that failed due to its internal structure being full (cannot be processed further)	int	-6
HDFQL_ERROR_BEFORE_FIRST	Represents an operation that failed due to trying to position/access an element before the first one	int	-7
HDFQL_ERROR_AFTER_LAST	Represents an operation that failed due to trying to position/access an element after the last one	int	-8
HDFQL_ERROR_NO_ADDRESS	Represents an operation that failed due to a user-defined variable having no address (i.e. is NULL)	int	-9
HDFQL_ERROR_NOT_REGISTERED	Represents an operation that failed due to a user-defined variable not being registered	int	-10
HDFQL_ERROR_OUTSIDE_LIMIT	Represents an operation that failed due to being outside the limit (cannot be processed further)	int	-11
HDFQL_ERROR_UNKNOWN	Represents an operation that failed due to an unknown/unexpected error	int	-99

Table 5.1 – HDFql constants in C

HDFql also supports other programming languages namely C++, Java, Python, C# and Fortran through wrappers. The below tables provide examples on how HDFql constants are defined in these programming languages.

In C++, the prefix “HDFQL_” of the name of constants (defined in C) is replaced by the namespace “HDFql” and its underscores (_) are discarded. The remainder of the name of constants follows the upper camel-case convention. The following table lists a subset of HDFql constants as defined in C and details how these are defined/can be used in C++.

HDFql Constant in C	Corresponding Definition in C++
HDFQL_VERSION	HDFql::Version
HDFQL_SUCCESS	HDFql::Success
HDFQL_ERROR_PARSE	HDFql::ErrorParse
HDFQL_TINYINT	HDFql::TinyInt
HDFQL_UNSIGNED_BIGINT	HDFql::UnsignedBigInt
HDFQL_UTF8	HDFql::Utf8

Table 5.2 – HDFql constants in C and their corresponding definitions in C++

In Java, the prefix “HDFQL_” of the name of constants (defined in C) is replaced by the class “HDFql”. The remainder of the name of constants remains exactly the same. The following table lists a subset of HDFql constants as defined in C and details how these are defined/can be used in Java.

HDFql Constant in C	Corresponding Definition in Java
HDFQL_VERSION	HDFql.VERSION
HDFQL_SUCCESS	HDFql.SUCCESS
HDFQL_ERROR_PARSE	HDFql.ERROR_PARSE
HDFQL_TINYINT	HDFql.TINYINT
HDFQL_UNSIGNED_BIGINT	HDFql.UNSIGNED_BIGINT
HDFQL_UTF8	HDFql.UTF8

Table 5.3 – HDFql constants in C and their corresponding definitions in Java

In Python, the prefix “HDFQL_” of the name of constants (defined in C) is replaced by the class “HDFql”. The remainder of the name of constants remains exactly the same. The following table lists a subset of HDFql constants as defined in C and details how these are defined/can be used in Python.

HDFql Constant in C	Corresponding Definition in Python
HDFQL_VERSION	HDFql.VERSION
HDFQL_SUCCESS	HDFql.SUCCESS
HDFQL_ERROR_PARSE	HDFql.ERROR_PARSE
HDFQL_TINYINT	HDFql.TINYINT
HDFQL_UNSIGNED_BIGINT	HDFql.UNSIGNED_BIGINT
HDFQL_UTF8	HDFql.UTF8

Table 5.4 – HDFql constants in C and their corresponding definitions in Python

In C#, the prefix “HDFQL_” of the name of constants (defined in C) is replaced by the class “HDFql” and its underscores (_) are discarded. The remainder of the name of constants follows the upper camel-case convention. The following table lists a subset of HDFql constants as defined in C and details how these are defined/can be used in C#.

HDFql Constant in C	Corresponding Definition in C#
HDFQL_VERSION	HDFql.Version
HDFQL_SUCCESS	HDFql.Success
HDFQL_ERROR_PARSE	HDFql.ErrorParse
HDFQL_TINYINT	HDFql.TinyInt
HDFQL_UNSIGNED_BIGINT	HDFql.UnsignedBigInt
HDFQL_UTF8	HDFql.Utf8

Table 5.5 – HDFql constants in C and their corresponding definitions in C#

In Fortran, the name of constants is the same as in C and can be written in any case. The following table lists a subset of HDFql constants as defined in C and details how these are defined/can be used in Fortran.

HDFql Constant in C	Corresponding Definition in Fortran
HDFQL_VERSION	HDFQL_VERSION
HDFQL_SUCCESS	HDFQL_SUCCESS
HDFQL_ERROR_PARSE	HDFQL_ERROR_PARSE
HDFQL_TINYINT	HDFQL_TINYINT
HDFQL_UNSIGNED_BIGINT	HDFQL_UNSIGNED_BIGINT
HDFQL_UTF8	HDFQL_UTF8

Table 5.6 – HDFql constants in C and their corresponding definitions in Fortran

5.2 FUNCTIONS

A function is a group of instructions that together perform a specific task, requiring direction back to the caller on completion of the task. Any given function might be called at any point during a program's execution, including by other functions or itself. It provides better modularity of a program and a high degree of code reusing. The following table summarizes all existing HDFql functions in C.

HDFql Function in C	Description
hdfql_execute	Execute a script (composed of one or more operations)
hdfql_execute_get_status	Get status of the last executed operation
hdfql_error_get_line	Get error line of the last executed operation
hdfql_error_get_position	Get error position of the last executed operation
hdfql_error_get_message	Get error message of the last executed operation
hdfql_cursor_initialize	Initialize a cursor for subsequent use
hdfql_cursor_use	Set the cursor to be used for storing the result of operations
hdfql_cursor_use_default	Set HDFql default cursor as the one to be used for storing the result of operations
hdfql_cursor_clear	Clear (i.e. empty) the cursor in use

hdfql_cursor_clone	Clone (i.e. duplicate) a cursor into another one
hdfql_cursor_get_datatype	Get datatype of the cursor in use
hdfql_cursor_get_count	Get number of elements (i.e. result set size) stored in the cursor in use
hdfql_subcursor_get_count	Get number of elements (i.e. result subset size) stored in the subcursor in use
hdfql_cursor_get_position	Get current position of cursor in use within result set
hdfql_subcursor_get_position	Get current position of subcursor in use within result subset
hdfql_cursor_first	Move the cursor in use to the first position within result set
hdfql_subcursor_first	Move the subcursor in use to the first position within result subset
hdfql_cursor_last	Move the cursor in use to the last position within result set
hdfql_subcursor_last	Move the subcursor in use to the last position within result subset
hdfql_cursor_next	Move the cursor in use one position forward from its current position
hdfql_subcursor_next	Move the subcursor in use one position forward from its current position
hdfql_cursor_previous	Move the cursor in use one position backward from its current position
hdfql_subcursor_previous	Move the subcursor in use one position backward from its current position
hdfql_cursor_absolute	Move the cursor in use to an absolute position within the result set
hdfql_subcursor_absolute	Move the subcursor in use to an absolute position within the result subset
hdfql_cursor_relative	Move the cursor in use to a relative position within result set
hdfql_subcursor_relative	Move the subcursor in use to a relative position within result subset
hdfql_cursor_get_size	Get current element size (in bytes) of the cursor in use
hdfql_subcursor_get_size	Get current element size (in bytes) of the subcursor in use
hdfql_cursor_get	Get current element of the cursor in use as a generic (typeless) pointer
hdfql_subcursor_get	Get current element of the subcursor in use as a generic (typeless) pointer
hdfql_cursor_get_tinyint	Get current element of the cursor in use as a TINYINT
hdfql_subcursor_get_tinyint	Get current element of the subcursor in use as a TINYINT
hdfql_cursor_get_unsigned_tinyint	Get current element of the cursor in use as an UNSIGNED TINYINT

hdfql_subcursor_get_unsigned_tinyint	Get current element of the subcursor in use as an UNSIGNED TINYINT
hdfql_cursor_get_smallint	Get current element of the cursor in use as a SMALLINT
hdfql_subcursor_get_smallint	Get current element of the subcursor in use as a SMALLINT
hdfql_cursor_get_unsigned_smallint	Get current element of the cursor in use as an UNSIGNED SMALLINT
hdfql_subcursor_get_unsigned_smallint	Get current element of the subcursor in use as an UNSIGNED SMALLINT
hdfql_cursor_get_int	Get current element of the cursor in use as an INT
hdfql_subcursor_get_int	Get current element of the subcursor in use as an INT
hdfql_cursor_get_unsigned_int	Get current element of the cursor in use as an UNSIGNED INT
hdfql_subcursor_get_unsigned_int	Get current element of the subcursor in use as an UNSIGNED INT
hdfql_cursor_get_bigint	Get current element of the cursor in use as a BIGINT
hdfql_subcursor_get_bigint	Get current element of the subcursor in use as a BIGINT
hdfql_cursor_get_unsigned_bigint	Get current element of the cursor in use as an UNSIGNED BIGINT
hdfql_subcursor_get_unsigned_bigint	Get current element of the subcursor in use as an UNSIGNED BIGINT
hdfql_cursor_get_float	Get current element of the cursor in use as a FLOAT
hdfql_subcursor_get_float	Get current element of the subcursor in use as a FLOAT
hdfql_cursor_get_double	Get current element of the cursor in use as a DOUBLE
hdfql_subcursor_get_double	Get current element of the subcursor in use as a DOUBLE
hdfql_cursor_get_char	Get current element of the cursor in use as a CHAR
hdfql_subcursor_get_char	Get current element of the subcursor in use as a CHAR
hdfql_variable_register	Register a variable for subsequent use
hdfql_variable_unregister	Unregister a variable
hdfql_variable_get_number	Get number of a variable
hdfql_variable_get_datatype	Get datatype of a variable
hdfql_variable_get_count	Get number of elements (i.e. result set size) stored in a variable
hdfql_variable_get_size	Get size (in bytes) of a variable

hdfql_variable_get_dimension_count	Get number of dimensions of a variable
hdfql_variable_get_dimension	Get size of a certain dimension of a variable

Table 5.7 – HDFql functions in C

In C++, the prefix "hdfql_" of the name of functions (defined in C) is replaced by the namespace "HDFql" and its underscores (_) are discarded. The remainder of the name of functions follows the lower camel-case convention. The following table lists a subset of HDFql functions as defined in C and details how these are defined/can be used in C++.

HDFql Function in C	Corresponding Definition in C++
hdfql_execute	HDFql::execute
hdfql_cursor_next	HDFql::cursorNext
hdfql_cursor_get_tinyint	HDFql::cursorGetTinyInt
hdfql_cursor_get_unsigned_int	HDFql::cursorGetUnsignedInt
hdfql_subcursor_get_big_int	HDFql::subcursorGetBigInt
hdfql_variable_get_number	HDFql::variableGetNumber

Table 5.8 – HDFql functions in C and their corresponding definitions in C++

In Java, the prefix "hdfql_" of the name of functions (defined in C) is replaced by the class "HDFql" and its underscores (_) are discarded. The remainder of the name of functions follows the lower camel-case convention. The following table lists a subset of HDFql functions as defined in C and details how these are defined/can be used in Java.

HDFql Function in C	Corresponding Definition in Java
hdfql_execute	HDFql.execute
hdfql_cursor_next	HDFql.cursorNext

hdfql_cursor_get_tinyint	HDFql.cursorGetTinyInt
hdfql_cursor_get_unsigned_int	HDFql.cursorGetUnsignedInt
hdfql_subcursor_get_big_int	HDFql.subcursorGetBigInt
hdfql_variable_get_number	HDFql.variableGetNumber

Table 5.9 – HDFql functions in C and their corresponding definitions in Java

In Python, the prefix “hdfql_” of the name of functions (defined in C) is replaced by the class “HDFql”. The remainder of the name of functions remains exactly the same. The following table lists a subset of HDFql functions as defined in C and details how these are defined/can be used in Python.

HDFql Function in C	Corresponding Definition in Python
hdfql_execute	HDFql.execute
hdfql_cursor_next	HDFql.cursor_next
hdfql_cursor_get_tinyint	HDFql.cursor_get_tinyint
hdfql_cursor_get_unsigned_int	HDFql.cursor_get_unsigned_int
hdfql_subcursor_get_big_int	HDFql.subcursor_get_big_int
hdfql_variable_get_number	HDFql.variable_get_number

Table 5.10 – HDFql functions in C and their corresponding definitions in Python

In C#, the prefix “hdfql_” of the name of functions (defined in C) is replaced by the class “HDFql” and its underscores (_) are discarded. The remainder of the name of functions follows the upper camel-case convention. The following table lists a subset of HDFql functions as defined in C and details how these are defined/can be used in C#.

HDFql Function in C	Corresponding Definition in C#
hdfql_execute	HDFql.Execute

hdfql_cursor_next	HDFql.CursorNext
hdfql_cursor_get_tinyint	HDFql.CursorGetTinyInt
hdfql_cursor_get_unsigned_int	HDFql.CursorGetUnsignedInt
hdfql_subcursor_get_big_int	HDFql.SubcursorGetBigInt
hdfql_variable_get_number	HDFql.VariableGetNumber

Table 5.11 – HDFql functions in C and their corresponding definitions in C#

In Fortran, the name of functions is the same as in C and can be written using any case. The following table lists a subset of HDFql functions as defined in C and details how these are defined/can be used in Fortran.

HDFql Function in C	Corresponding Definition in Fortran
hdfql_execute	hdfql_execute
hdfql_cursor_next	hdfql_cursor_next
hdfql_cursor_get_tinyint	hdfql_cursor_get_tinyint
hdfql_cursor_get_unsigned_int	hdfql_cursor_get_unsigned_int
hdfql_subcursor_get_big_int	hdfql_subcursor_get_big_int
hdfql_variable_get_number	hdfql_variable_get_number

Table 5.12 – HDFql functions in C and their corresponding definitions in Fortran

5.2.1 HDFQL_EXECUTE

Syntax

int hdfql_execute(const char **script*)

Description

Execute a script named *script*. A script can be composed of one or more operations – in case of multiple operations these can either be separated with a semicolon (;) or an end of line (EOL) terminator. In HDFql, operations are case insensitive meaning that, for example, operation "SHOW DATASET" is equivalent to "show dataset" or any other case variation. If a certain operation raises an error, any subsequent operations within *script* are not executed. Please refer to Table 6.2 for a complete enumeration of HDFql operations.

Parameter(s)

script – string containing one or more operations to execute. Multiple operations are either separated with a semicolon (;) or an end of line (EOL) terminator.

Return

int – depending on the success in executing *script*, it can either be HDFQL_SUCCESS, HDFQL_ERROR_PARSE, HDFQL_ERROR_NOT_FOUND, HDFQL_ERROR_NO_ACCESS, HDFQL_ERROR_ALREADY_EXISTS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_FULL, HDFQL_ERROR_BEFORE_FIRST, HDFQL_ERROR_AFTER_LAST, HDFQL_ERROR_NO_ADDRESS, HDFQL_ERROR_NOT_REGISTERED, HDFQL_ERROR_OUTSIDE_LIMIT or HDFQL_ERROR_UNKNOWN.

Example(s)

```
// declare variable
int status;

// execute script (composed of only one operation - i.e. SHOW USE FILE)
status = hdfql_execute("SHOW USE FILE");

// display message about the status of executed script (i.e. successful or not)
if (status == HDFQL_SUCCESS)
   printf("Execution was successful\n");
else
   printf("Execution was not successful and returned status is %d\n", status);
```

```
// execute script (composed of two operations - i.e. USE FILE my_file.h5 and SHOW)
```

```
hdfql_execute("USE FILE my_file.h5 ; SHOW");
```

5.2.2 HDFQL_EXECUTE_GET_STATUS

Syntax

int hdfql_execute_get_status(void)

Description

Get status of the last executed operation. In other words, this function returns the status of the last call of hdfql_execute.

Parameter(s)

None

Return

int – depending on the success of the last executed operation, it can either be HDFQL_SUCCESS, HDFQL_ERROR_PARSE, HDFQL_ERROR_NOT_FOUND, HDFQL_ERROR_NO_ACCESS, HDFQL_ERROR_ALREADY_EXISTS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_FULL, HDFQL_ERROR_BEFORE_FIRST, HDFQL_ERROR_AFTER_LAST, HDFQL_ERROR_NO_ADDRESS, HDFQL_ERROR_NOT_REGISTERED, HDFQL_ERROR_OUTSIDE_LIMIT or HDFQL_ERROR_UNKNOWN.

Example(s)

```
// declare variable
int status;

// execute script (composed of only one operation - i.e. SHOW USE DIRECTORY)
hdfql_execute("SHOW USE DIRECTORY");

// get status of last executed script (i.e. SHOW USE DIRECTORY)
status = hdfql_execute_get_status();

// display message about the status of last executed script (i.e. successful or not)
```

```
if (status == HDFQL_SUCCESS)
    printf("Execution was successful\n");
else
    printf("Execution was not successful and returned status is %d\n", status);
```

5.2.3 HDFQL_ERROR_GET_LINE

Syntax

int hdfql_error_get_line(void)

Description

Get error line of the last executed operation. In other words, this function returns the number of the line (in the script) where an error was raised during the last call of hdfql_execute. The first line in the script is designated as number one (1).

Parameter(s)

None

Return

int – number of the line (in the script) where an error has occurred during the last executed operation. If the last executed operation was sucessful, the number of the line will be HDFQL_UNDEFINED.

Example(s)

```
// execute script (composed of only one operation - i.e. CREATE FILE my_file.h5 - which
is syntactically correct)
hdfql_execute("CREATE FILE my_file.h5");

// display number of the line where an error occurred during the last executed operation
(should be -1 - i.e. HDFQL_UNDEFINED)
printf("Error line number is %d\n", hdfql_error_get_line());

// execute script (composed of only one operation - i.e. CREATE FILEX my_file.h5 - which
is syntactically incorrect due to a typo in "FILEX")
```

```
hdfql_execute("CREATE FILEX my_file.h5");

// display number of the line where an error occurred during the last executed operation (should be 1)
printf("Error line number is %d\n", hdfql_error_get_line());
```

5.2.4 HDFQL_ERROR_GET_POSITION

Syntax

int hdfql_error_get_position(void)

Description

Get error position of the last executed operation. In other words, this function returns the position in the line where an error was raised during the last call of hdfql_execute. The first position in the line is designated as number one (1).

Parameter(s)

None

Return

int – position in the line where an error has occurred during the last executed operation. If the last executed operation was sucessful, the position in the line will be HDFQL_UNDEFINED.

Example(s)

```
// execute script (composed of only one operation - i.e. CREATE FILE my_file.h5 - which is syntactically correct)
hdfql_execute("CREATE FILE my_file.h5");

// display position in the line where an error occurred during the last executed operation (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Error position is %d\n", hdfql_error_get_position());

// execute script (composed of only one operation - i.e. CREATE FILEX my_file.h5 - which
```

```
is syntactically incorrect due to a typo in "FILEX")
hdfql_execute("CREATE FILEX my_file.h5");

// display position in the line where an error occurred during the last executed
operation (should be 8)
printf("Error position is %d\n", hdfql_error_get_position());
```

5.2.5 HDFQL_ERROR_GET_MESSAGE

Syntax

char *hdfql_error_get_message(void)

Description

Get error message of the last executed operation. In other words, this function returns the message of the error that was raised during the last call of hdfql_execute.

Parameter(s)

None

Return

char – pointer to the message of an error that has occurred during the last executed operation. If the last executed operation was sucessful, the pointer will be NULL.

Example(s)

```
// execute script (composed of only one operation - i.e. CREATE FILE my_file.h5 - which
is syntactically correct)
hdfql_execute("CREATE FILE my_file.h5");

// display message of an error that occurred during the last executed operation (should
be "NULL")
printf("%s\n", hdfql_error_get_message());

// execute script (composed of only one operation - i.e. CREATE FILEX my_file.h5 - which
```

```
is syntactically incorrect due to a typo in "FILEX")
hdfql_execute("CREATE FILEX my_file.h5");

// display message of an error that occurred during the last executed operation (should
be "Unknown token “FILEX”")
printf("%s\n", hdfql_error_get_message());
```

5.2.6 HDFQL_CURSOR_INITIALIZE

Syntax

int hdfql_cursor_initialize(HDFQL_CURSOR **cursor*)

Description

Initialize a cursor named *cursor* for subsequent use. Before a new cursor is used for the first time, it should always be initialized (otherwise unexpected errors may arise). The initialization of a cursor sets its datatype attribute to undefined (HDFQL_UNDEFINED), its current element to NULL, and resets its count and position attributes to zero making it ready for usage. Of note, the process of initializing a cursor is only required in C and performed once, while in other programming languages supported by HDFql – namely, C++, Java, Python, C# and Fortran – such initialization is redundant as it is done automatically when declaring a cursor.

Parameter(s)

cursor – pointer to a cursor (previously declared) to initialize with default values. If the pointer is NULL (in C), the cursor in use is initialized instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is initialized instead).

Return

int – depending on the success in initializing *cursor*, it can either be HDFQL_SUCCESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// create a cursor named "my_cursor"
HDFQL_CURSOR my_cursor;

// initialize cursor "my_cursor"
hdfql_cursor_initialize(&my_cursor);

// use cursor "my_cursor"
hdfql_cursor_use(&my_cursor);

// display number of elements in cursor "my_cursor" (should be 0)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));
```

5.2.7 HDFQL_CURSOR_USE

Syntax

int hdfql_cursor_use(HDFQL_CURSOR **cursor*)

Description

Set the cursor named *cursor* as the one to be used for storing results of operations.

Parameter(s)

cursor – pointer to a cursor to use for storing the result of operations. If the pointer is NULL (in C), the HDFql default cursor is used instead (i.e. equivalent of calling the function hdfql_cursor_use_default). The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is used instead).

Return

int – depending on the success in using *cursor*, it can either be HDFQL_SUCCESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// create a cursor named "my_cursor"
HDFQL_CURSOR my_cursor;

// use cursor "my_cursor"
hdfql_cursor_use(&my_cursor);

// initialize cursor "my_cursor"
hdfql_cursor_initialize(NULL);

// display datatype of cursor "my_cursor" (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));

// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// display (again) datatype of cursor "my_cursor" (should be 1024 - i.e. HDFQL_CHAR)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));

// use HDFql default cursor
hdfql_cursor_use(NULL);

// display datatype of HDFql default cursor (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));
```

5.2.8 HDFQL_CURSOR_USE_DEFAULT

Syntax

int hdfql_cursor_use_default(void)

Description

Set HDFql default cursor as the one to be used for storing results of operations.

Parameter(s)

None

Return

int – depending on the success in using HDFql default cursor, it can either be HDFQL_SUCCESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// create a cursor named "my_cursor"
HDFQL_CURSOR my_cursor;

// initialize cursor "my_cursor"
hdfql_cursor_initialize(&my_cursor);

// use cursor "my_cursor"
hdfql_cursor_use(&my_cursor);

// display datatype of cursor "my_cursor" (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));

// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// display (again) datatype of cursor "my_cursor" (should be 1024 - i.e. HDFQL_CHAR)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));

// use HDFql default cursor
hdfql_cursor_use_default();

// display datatype of HDFql default cursor (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));
```

5.2.9 HDFQL_CURSOR_CLEAR

Syntax

int hdfql_cursor_clear(HDFQL_CURSOR **cursor*)

Description

Clear (i.e. empty) a cursor named *cursor*. Specifically, this function removes all elements (i.e. result set) stored in the cursor, specifies its datatype attribute to undefined (HDFQL_UNDEFINED), changes its current element to NULL, and resets its count and position attributes to zero.

Parameter(s)

cursor – pointer to a cursor to clear (i.e. empty). If the pointer is NULL (in C), the cursor in use is cleared instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is cleared instead).

Return

int – depending on the success in clearing *cursor*, it can either be HDFQL_SUCCESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// display number of elements in the cursor in use (should be 1)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));

// clear the cursor in use
hdfql_cursor_clear(NULL);

// display (again) number of elements in the cursor in use (should be 0)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));
```

5.2.10 HDFQL_CURSOR_CLONE

Syntax

int hdfql_cursor_clone(HDFQL_CURSOR *cursor_original, HDFQL_CURSOR *cursor_clone)

Description

Clone (i.e. duplicate) a cursor named *cursor_original* into another one named *cursor_clone*. In other words, *cursor_clone* will be an exact copy of *cursor_original*, meaning that it will have the same datatype, count and position values, store the same result set, and have the same current element as the original cursor.

Parameter(s)

cursor_original – pointer to a cursor to clone. If the pointer is NULL (in C), the cursor in use is the one to be cloned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is the one to be cloned instead).

cursor_clone – pointer to the cursor that will be a clone (i.e. duplicate) of the original cursor.

Return

int – depending on the success in cloning *cursor_original* into *cursor_clone*, it can either be HDFQL_SUCCESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// create a cursor named "my_cursor"
HDFQL_CURSOR my_cursor;

// initialize cursor "my_cursor"
hdfql_cursor_initialize(&my_cursor);

// get current working directory (it will be stored in HDFql default cursor)
hdfql_execute("SHOW USE DIRECTORY");

// clone the cursor in use (i.e. HDFql default cursor) into the cursor "my_cursor"
hdfql_cursor_clone(NULL, &my_cursor, HDFQL_NO);

// use cursor "my_cursor"
hdfql_cursor_use(&my_cursor);

// display number of elements in the cursor in use (should be 1)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));
```

5.2.11 HDFQL_CURSOR_GET_DATATYPE

Syntax

int hdfql_cursor_get_datatype(HDFQL_CURSOR **cursor*)

Description

Get the datatype of a cursor named *cursor*. If the cursor has never been populated or has been initialized or cleared, the returned datatype is undefined (HDFQL_UNDEFINED). Please refer to Table 6.3 for a complete enumeration of HDFql datatypes.

Parameter(s)

cursor – pointer to a cursor to get its datatype. If the pointer is NULL (in C), the datatype of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the datatype of the cursor in use is returned instead).

Return

int – depending on the datatype of the cursor or its state (i.e. whether it has never been populated or has been initialized or cleared), it can either be HDFQL_TINYINT, HDFQL_UNSIGNED_TINYINT, HDFQL_SMALLINT, HDFQL_UNSIGNED_SMALLINT, HDFQL_INT, HDFQL_UNSIGNED_INT, HDFQL_BIGINT, HDFQL_UNSIGNED_BIGINT, HDFQL_FLOAT, HDFQL_DOUBLE, HDFQL_CHAR, HDFQL_VARTINYINT, HDFQL_UNSIGNED_VARTINYINT, HDFQL_VARSMALLINT, HDFQL_UNSIGNED_VARSMALLINT, HDFQL_VARINT, HDFQL_UNSIGNED_VARINT, HDFQL_VARBIGINT, HDFQL_UNSIGNED_VARBIGINT, HDFQL_VARFLOAT, HDFQL_VARDOUBLE, HDFQL_VARCHAR, HDFQL_OPAQUE or HDFQL_UNDEFINED.

Example(s)

```
// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// display datatype of the cursor in use (should be 1024 - i.e. HDFQL_CHAR)
```

```
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));

// clear the cursor in use
hdfql_cursor_clear(NULL);

// display (again) datatype of the cursor in use (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Datatype of cursor is %d\n", hdfql_cursor_get_type(NULL));
```

5.2.12 HDFQL_CURSOR_GET_COUNT

Syntax

int hdfql_cursor_get_count(HDFQL_CURSOR **cursor*)

Description

Get the number of elements (i.e. result set size) stored in a cursor named *cursor*. If the result set stores data from a dataset or attribute that does not have a dimension (i.e. if it is scalar), the returned number of elements is one. Otherwise, if the result set stores data from a dataset or attribute that has dimensions, the returned number of elements equals the multiplication of all its dimensions' sizes (e.g. if a cursor stores a result set of two dimensions of size 10x3, the number of elements is 30). If the cursor has never been populated or has been initialized or cleared, the returned number of elements is zero.

Parameter(s)

cursor – pointer to a cursor to get its number of elements (i.e. result set size). If the pointer is NULL (in C), the number of elements of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the number of elements of the cursor in use is returned instead).

Return

int – number of elements (i.e. result set size) stored in the cursor.

Example(s)

```
// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// display number of elements in the cursor in use (should be 1)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));

// clear the cursor in use
hdfql_cursor_clear(NULL);

// display (again) number of elements in the cursor in use (should be 0)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));
```

5.2.13 HDFQL_SUBCURSOR_GET_COUNT

Syntax

int hdfql_subcursor_get_count(HDFQL_CURSOR **cursor*)

Description

Get the number of elements (i.e. result subset size) stored in the subcursor in use. If the cursor that the subcursor belongs to has never been populated or has been initialized or cleared, the returned number of elements is zero.

Parameter(s)

cursor – pointer to a cursor to get the number of elements (i.e. result subset size) stored in the subcursor in use. If the pointer is NULL (in C), the number of elements of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the number of elements of the subcursor of the cursor in use is returned instead).

Return

int – number of elements (i.e. result subset size) stored in the subcursor.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// display number of elements in the cursor in use (should be 4 - i.e. 2x2)
printf("Number of elements in cursor is %d\n", hdfql_cursor_get_count(NULL));

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display number of elements in the subcursor in use (should be 3)
printf("Number of elements in subcursor is %d\n", hdfql_subcursor_get_count(NULL));

// move the cursor in use to next position within the result set (i.e. second position)
hdfql_cursor_next(NULL);

// display number of elements in the subcursor in use (should be 1)
printf("Number of elements in subcursor is %d\n", hdfql_subcursor_get_count(NULL));
```

5.2.14 HDFQL_CURSOR_GET_POSITION

Syntax

int hdfql_cursor_get_position(HDFQL_CURSOR **cursor*)

Description

Get current position of a cursor named *cursor* within the result set. The first element of the result set is at position one (1), while the last element is located at the position returned by hdfql_cursor_get_count. If the cursor has never been populated or has been initialized or cleared, or in case the result set is empty, the

returned current position is zero. If the cursor was moved before the first element or after the last element, the returned current position is zero or the number of elements in the result set plus one (1), respectively.

Parameter(s)

cursor – pointer to a cursor to get its current position within the result set. If the pointer is NULL (in C), the current position of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current position of the cursor in use is returned instead).

Return

int – current position of the cursor within the result set.

Example(s)

```
// clear the cursor in use
hdfql_cursor_clear(NULL);

// display position of the cursor in use within the result set (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));

// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// display (again) position of the cursor in use within the result set (should be 1)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));
```

5.2.15 HDFQL_SUBCURSOR_GET_POSITION

Syntax

int hdfql_subcursor_get_position(HDFQL_CURSOR **cursor*)

Description

Get current position of the subcursor in use within the result subset. The first element of the result subset is at position one (1), while the last element is located at the position returned by hdfql_subcursor_get_count. If the cursor that the subcursor belongs to has never been populated or has been initialized or cleared, or in case the result subset is empty, the returned current position is zero. If the subcursor was moved before the first element or after the last element, the returned current position is zero or the number of elements in the result subset plus one (1), respectively.

Parameter(s)

cursor – pointer to a cursor to get the current position of the subcursor in use within the result subset. If the pointer is NULL (in C), the current position of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current position of the subcursor of the cursor in use is returned instead).

Return

int – current position of the subcursor within the result subset.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// display position of the subcursor in use within the result subset (should be -1 - i.e.
HDFQL_UNDEFINED)
```

```
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));

// move the subcursor in use to the next position within the result subset (two times)
hdfql_subcursor_next(NULL);
hdfql_subcursor_next(NULL);

// display (again) position of the subcursor in use within the result subset (should be 2)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));
```

5.2.16 HDFQL_CURSOR_FIRST

Syntax

int hdfql_cursor_first(HDFQL_CURSOR **cursor*)

Description

Move a cursor named *cursor* to the first position within the result set. In other words, the cursor will point to the first element of the result set and its position is set to one (1). If the result set is empty, an error is returned and its position remains unchanged (i.e. remains zero).

Parameter(s)

cursor – pointer to a cursor to move to the first position within the result set. If the pointer is NULL (in C), the cursor in use is moved to the first position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is moved to the first position instead).

Return

int – depending on the success in moving the cursor to the first position within the result set, it can either be HDFQL_SUCCESS or HDFQL_ERROR_EMPTY.

Example(s)

```
// get current working directory
```

```
hdfql_execute("SHOW USE DIRECTORY");

// display position of the cursor in use within the result subset (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// display (again) position of the cursor in use within the result set (should be 1)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));
```

5.2.17 HDFQL_SUBCURSOR_FIRST

Syntax

int hdfql_subcursor_first(HDFQL_CURSOR **cursor*)

Description

Move the subcursor in use to the first position within the result subset. In other words, the subcursor will point to the first element of the result subset and its position is set to one (1). If the result subset is empty, an error is returned and its position remains unchanged (i.e. remains zero).

Parameter(s)

cursor – pointer to a cursor to move the subcursor in use to the first position within the result subset. If the pointer is NULL (in C), the subcursor of the cursor in use is moved to the first position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the subcursor of the cursor in use is moved to the first position instead).

Return

int – depending on the success in moving the subcursor to the first position within the result subset, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions (size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// display position of the subcursor in use within the result subset (should be -1 - i.e. HDFQL_UNDEFINED)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));

// move the subcursor in use to the first position within the result subset
hdfql_subcursor_first(NULL);

// display (again) position of the subcursor in use within the result subset (should be 1)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));
```

5.2.18 HDFQL_CURSOR_LAST

Syntax

int hdfql_cursor_last(HDFQL_CURSOR **cursor*)

Description

Move a cursor named *cursor* to the last position within the result set. In other words, the cursor will point to the last element of the result set and its position is set to the value returned by hdfql_cursor_get_count. If the result set is empty, an error is returned and its position remains unchanged (i.e. remains zero).

Parameter(s)

cursor – pointer to a cursor to move to the last position within the result set. If the pointer is NULL (in C), the cursor in use is moved to the last position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is moved to the last position instead).

Return

int – depending on the success in moving the cursor to the last position within the result set, it can either be HDFQL_SUCCESS or HDFQL_ERROR_EMPTY.

Example(s)

```
// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// move the cursor in use to the last position within the result set
hdfql_cursor_last(NULL);

// display position of the cursor in use within the result set (should be 1)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));
```

5.2.19 HDFQL_SUBCURSOR_LAST

Syntax

int hdfql_subcursor_last(HDFQL_CURSOR **cursor*)

Description

Move the subcursor in use to the last position within the result subset. In other words, the subcursor will point to the last element of the result subset and its position is set to the value returned by hdfql_subcursor_get_count. If the result subset is empty, an error is returned and its position remains unchanged (i.e. remains zero).

Parameter(s)

cursor – pointer to a cursor to move the subcursor in use to the last position within the result subset. If the pointer is NULL (in C), the subcursor of the cursor in use is moved to the last position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the subcursor of the cursor in use is moved to the last position instead).

Return

int – depending on the success in moving the subcursor to the last position within the result subset, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// display position of subcursor in use within the result subset (should be -1 - i.e.
HDFQL_UNDEFINED)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));

// move the subcursor in use to the last position within the result set
hdfql_subcursor_last(NULL);

// display (again) position of subcursor in use within the result subset (should be 3)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));
```

5.2.20 HDFQL_CURSOR_NEXT

Syntax

int hdfql_cursor_next(HDFQL_CURSOR **cursor*)

Description

Move a cursor named *cursor* one position forward from its current position. In other words, the cursor will point to the next element of the result set and its position is incremented by one. If the result set is empty or the cursor is in the last position, an error is returned and its position remains unchanged (i.e. remains zero) or is set to the value returned by hdfql_cursor_get_count plus one (1), respectively.

Parameter(s)

cursor – pointer to a cursor to move one position forward from its current position. If the pointer is NULL (in C), the cursor in use is moved one position forward from its current position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is moved one position forward from its current position instead).

Return

int – depending on the success in moving the cursor one position forward from its current position, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// get current working directory
hdfql_execute("SHOW USE DIRECTORY");

// move the cursor in use to the next position within the result set
hdfql_cursor_next(NULL);

// display position of cursor within the result set (should be 1)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));
```

5.2.21 HDFQL_SUBCURSOR_NEXT

Syntax

int hdfql_subcursor_next(HDFQL_CURSOR *_cursor_)

Description

Move the subcursor in use one position forward from its current position. In other words, the subcursor will point to the next element of the result subset and its position is incremented by one. If the result subset is empty or the subcursor is in the last position, an error is returned and its position remains unchanged (i.e. remains zero) or is set to the value returned by hdfql_subcursor_get_count plus one (1), respectively

Parameter(s)

cursor – pointer to a cursor to move the subcursor in use one position forward from its current position. If the pointer is NULL (in C), the subcursor of the cursor in use is moved one position forward from its current position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the subcursor of the cursor in use is moved one position forward from its current position instead).

Return

int – depending on the success in moving the subcursor one position forward from its current position, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");
```

```
// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// display position of subcursor in use within the result set (should be -1 - i.e.
HDFQL_UNDEFINED)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));

// move the subcursor in use to the next position within the result subset (two times)
hdfql_subcursor_next(NULL);
hdfql_subcursor_next(NULL);

// display (again) position of subcursor in use within the result subset (should be 2)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));
```

5.2.22 HDFQL_CURSOR_PREVIOUS

Syntax

int hdfql_cursor_previous(HDFQL_CURSOR **cursor*)

Description

Move a cursor named *cursor* one position backward from its current position. In other words, the cursor will point to the previous element of the result set and its position is decremented by one. If the result set is empty or the cursor is in the first position, an error is returned and its position remains unchanged (i.e. remains zero) or is set to zero, respectively.

Parameter(s)

cursor – pointer to a cursor to move one position backward from its current position. If the pointer is NULL (in C), the cursor in use is moved one position backward from its current position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is moved one position backward from its current position instead).

Return

int – depending on the success in moving the cursor one position backward from its current position, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY or HDFQL_ERROR_BEFORE_FIRST.

Example(s)

```
// create a dataset named "my_dataset" of type float of two dimensions (size 2x10)
hdfql_execute("CREATE DATASET my_dataset AS FLOAT(2, 10)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the last position within the result set
hdfql_cursor_last(NULL);

// move the cursor in use to the previous position within the result set
hdfql_cursor_previous(NULL);

// display position of cursor in use within the result set (should be 19 - i.e. 2x10-1)
printf("Position of cursor is %d\n", hdfql_cursor_get_position(NULL));
```

5.2.23 HDFQL_SUBCURSOR_PREVIOUS

Syntax

int hdfql_subcursor_previous(HDFQL_CURSOR **cursor*)

Description

Move the subcursor in use one position backward from its current position. In other words, the subcursor will point to the previous element of the result subset and its position is decremented by one. If the result subset is empty or the subcursor is in the first position, an error is returned and its position remains unchanged (i.e. remains zero) or is set to zero, respectively.

Parameter(s)

cursor – pointer to a cursor to move the subcursor in use one position backward from its current position. If the pointer is NULL (in C), the subcursor of the cursor in use is moved one position backward from its current position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the subcursor of the cursor in use is moved one position backward from its current position instead).

Return

int – depending on the success in moving the subcursor one position backward from its current position, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// move the subcursor in use to the last position within the result subset
hdfql_subcursor_last(NULL);

// move the subcursor in use to the previous position within the result subset (two
times)
hdfql_subcursor_previous(NULL);
hdfql_subcursor_previous(NULL);

// display position of the subcursor within the result subset (should be 1 - i.e. 3-1-1)
printf("Position of subcursor is %d\n", hdfql_subcursor_get_position(NULL));
```

5.2.24 HDFQL_CURSOR_ABSOLUTE

Syntax

int hdfql_cursor_absolute(HDFQL_CURSOR **cursor*, int *position*)

Description

Move a cursor named *cursor* to an absolute position *position* within the result set. If *position* is positive, the cursor will position itself with reference to the beginning of the result set. If *position* is negative, the cursor will position itself with reference to the end of the result set. The first element of the result set is at position one (1), while the last element is located at the position returned by hdfql_cursor_get_count. An attempt to move the cursor before the first element will return an error and set the position of the cursor to zero, while an attempt to move the cursor after the last element will return an error and set the position of the cursor to number of elements in the result set plus one (1).

Parameter(s)

cursor – pointer to a cursor to move to an absolute position within the result set. If the pointer is NULL (in C), the cursor in use is moved to an absolute position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is moved to an absolute position instead).

position – absolute position to which to move the cursor.

Return

int – depending on the success in moving the cursor to an absolute position within the result set, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create six HDF groups named "g1", "g2", "g3", "g4" and "g5"
hdfql_execute("CREATE GROUP g1, g2, g3, g4, g5");
```

```
// populate cursor in use with all existing groups (should be g1, g2, g3, g4, g5)
hdfql_execute("SHOW GROUP");

// move the cursor in use to absolute position 3 within the result set
hdfql_cursor_absolute(NULL, 3);

// display current element of the cursor in use within the result set (should be g3)
printf("Current element of cursor is %s", hdfql_cursor_get_char(NULL));

// move the cursor in use to absolute position -2 within the result set
hdfql_cursor_absolute(NULL, -2);

// display current element of the cursor in use within the result set (should be g4)
printf("Current element of cursor is %s", hdfql_cursor_get_char(NULL));
```

5.2.25 HDFQL_SUBCURSOR_ABSOLUTE

Syntax

int hdfql_subcursor_absolute(HDFQL_CURSOR **cursor*, int *position*)

Description

Move the subcursor in use to an absolute position *position* within the result subset. If *position* is positive, the subcursor will position itself with reference to the beginning of the result subset. If *position* is negative, the subcursor will position itself with reference to the end of the result subset. The first element of the result subset is at position one (1), while the last element is located at the position returned by hdfql_subcursor_get_count. An attempt to move the subcursor before the first element will return an error and set the position of the subcursor to zero, while an attempt to move the subcursor after the last element will return an error and set the position of the subcursor to number of elements in the result subset plus one (1).

Parameter(s)

cursor – pointer to a cursor to move the subcursor in use to an absolute position within the result subset. If the pointer is NULL (in C), the subcursor of the cursor in use is moved to an absolute position instead. The

equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the subcursor of the cursor in use is moved to an absolute position instead).

position – absolute position to which to move the subcursor.

Return

int – depending on the success in moving the subcursor to an absolute position within the result subset, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// move the subcursor in use to absolute position 3 within the result subset
hdfql_subcursor_absolute(NULL, 3);

// display current element of the subcursor in use within the result subset (should be 5)
printf("Current element of subcursor is %d", hdfql_cursor_get_int(NULL));

// move the subcursor in use to absolute position -2 within the result subset
hdfql_subcursor_absolute(NULL, -2);

// display current element of the subcursor in use within the result subset (should be 8)
printf("Current element of subcursor is %d", hdfql_cursor_get_int(NULL));
```

5.2.26 HDFQL_CURSOR_RELATIVE

Syntax

int hdfql_cursor_relative(HDFQL_CURSOR *_cursor_, int _position_)

Description

Move a cursor named *cursor* to a relative position *position* with respect to its current position. If *position* is positive, the cursor will go forward in the result set relative to its current position. If *position* is negative, the cursor will go backward in the result set relative to its current position. The first element of the result set is at position one (1), while the last element is located at the position returned by hdfql_cursor_get_count. An attempt to move the cursor before the first element will return an error and set the position of the cursor to zero, while an attempt to move the cursor after the last element will return an error and set the position of the cursor to number of elements in the result set plus one (1).

Parameter(s)

cursor – pointer to a cursor to move to a relative position with respect to its current position. If the pointer is NULL (in C), the cursor in use is moved to a relative position instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the cursor in use is moved to a relative position instead).

position – relative position to which to move the cursor.

Return

int – depending on the success in moving the cursor to a relative position with respect to its current position, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create six HDF groups named "g1", "g2", "g3", "g4" and "g5"
hdfql_execute("CREATE GROUP g1, g2, g3, g4, g5");
```

```
// populate cursor in use with all existing groups (should be g1, g2, g3, g4, g5)
hdfql_execute("SHOW GROUP");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// move the cursor in use to relative position 2 within the result set
hdfql_cursor_relative(NULL, 2);

// display current element of the cursor within the result set (should be g3)
printf("Current element of cursor is %s", hdfql_cursor_get_char(NULL));

// move the cursor in use to relative position -2 within the result set
hdfql_cursor_relative(NULL, -2);

// display current element of the cursor within the result set (should be g1)
printf("Current element of cursor is %s", hdfql_cursor_get_char(NULL));
```

5.2.27 HDFQL_SUBCURSOR_RELATIVE

Syntax

int hdfql_subcursor_relative(HDFQL_CURSOR **cursor*, int *position*)

Description

Move the subcursor in use to a relative position *position* with respect to its current position. If *position* is positive, the subcursor will go forward in the result set relative to its current position. If *position* is negative, the subcursor will go backward in the result set relative to its current position. The first element of the result subset is at position one (1), while the last element is located at the position returned by hdfql_subcursor_get_count. An attempt to move the subcursor before the first element will return an error and set the position of the subcursor to zero, while an attempt to move the subcursor after the last element will return an error and set the position of the subcursor to number of elements in the result set plus one (1).

Parameter(s)

cursor – pointer to a cursor to move the subcursor in use to a relative position with respect to its current position. If the pointer is NULL (in C), the subcursor of the cursor in use is moved to a relative position instead.

The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the subcursor of the cursor in use is moved to a relative position instead).

position – relative position to which to move the subcursor.

Return

int – depending on the success in moving the subcursor to a relative position with respect to its current position, it can either be HDFQL_SUCCESS, HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of two dimensions
(size 2x2)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(2, 2)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(((7, 8, 5), (9)), ((6, 1, 2, 3), (4, 0)))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// move the subcursor in use to the first position within the result subset
hdfql_subcursor_first(NULL);

// move the subcursor in use to relative position 2 within the result subset
hdfql_subcursor_relative(NULL, 2);

// display current element of the subcursor in use within the result subset (should be 5)
printf("Current element of subcursor is %d", hdfql_cursor_get_int(NULL));

// move the subcursor in use to relative position -1 within the result subset
hdfql_subcursor_relative(NULL, -1);

// display current element of the subcursor in use within the result subset (should be 8)
```

```
printf("Current element of subcursor is %d", hdfql_cursor_get_int(NULL));
```

5.2.28 HDFQL_CURSOR_GET_SIZE

Syntax

int hdfql_cursor_get_size(HDFQL_CURSOR *_cursor_)

Description

Get the current element size (in bytes) of a cursor named *cursor*. If the result set it empty or the cursor is located before or after the first or last element of the result set, an error is returned instead.

Parameter(s)

cursor – pointer to a cursor to get the current element size (in bytes). If the pointer is NULL (in C), the current element size of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element size of the cursor in use is returned instead).

Return

int – depending on the success in getting the current element size (in bytes) of the cursor, it can either be ≥ 0 (i.e. the size itself), HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create an HDF group named "my_group"
hdfql_execute("CREATE GROUP my_group");

// populate cursor in use with all existing groups (should be my_group)
hdfql_execute("SHOW GROUP");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);
```

```
// display current element size (in bytes) of the cursor in use within the result set
(should be 8 - i.e. 8x1)
printf("Current element size (in bytes) of cursor is %d\n", hdfql_cursor_get_size(NULL));
```

5.2.29 HDFQL_SUBCURSOR_GET_SIZE

Syntax

int hdfql_subcursor_get_size(HDFQL_CURSOR **cursor*)

Description

Get the current element size (in bytes) of the subcursor in use. If the result subset it empty or the subcursor is located before or after the first or last element of the result subset, an error is returned instead.

Parameter(s)

cursor – pointer to a cursor to get the current element size (in bytes) of the subcursor in use. If the pointer is NULL (in C), the current element size of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element size of the subcursor of the cursor in use is returned instead).

Return

int – depending on the success in getting the current element size (in bytes) of the subcursor, it can either be ≥ 0 (i.e. the size itself), HDFQL_ERROR_EMPTY, HDFQL_ERROR_BEFORE_FIRST or HDFQL_ERROR_AFTER_LAST.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length char of one dimension
(size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARCHAR(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(Red, Green, Blue)");
```

```
// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to the first position within the result set
hdfql_cursor_first(NULL);

// move the subcursor in use to the first position within the result subset
hdfql_subcursor_first(NULL);

// display current element size (in bytes) of the subcursor within the result subset
(should be 3 - i.e. 3x1)
printf("Current element size (in bytes) of subcursor is %d\n",
hdfql_subcursor_get_size(NULL));
```

5.2.30 HDFQL_CURSOR_GET

Syntax

void *hdfql_cursor_get(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a generic (typeless) pointer. It is up to the programmer to interpret the returned pointer according to their needs. If the result set it empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a generic (typeless) pointer. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

void – generic (typeless) pointer to the current element of the cursor. If there is no current element, the pointer is NULL.

Example(s)

```
// create a dataset named "my_dataset" of type float of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS FLOAT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(5.5, 8.1, 4.9)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a float (should be 5.5)
printf("Current element of cursor is %f\n", (float *) hdfql_cursor_get(NULL));
```

5.2.31 HDFQL_SUBCURSOR_GET

Syntax

void *hdfql_subcursor_get(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as a generic (typeless) pointer. It is up to the programmer to interpret the returned pointer according to their needs. If the result subset it empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a generic (typeless) pointer. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

void – generic (typeless) pointer to the current element of the subcursor. If there is no current element, the pointer is NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length float of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARFLOAT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5.5, 2.2), (8.1), (4.9, 3.4, 5.6))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// move the subcursor in use to next position within the result subset (i.e. first position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a float (should be 5.5)
printf("Current element of subcursor is %f\n", (float *) hdfql_subcursor_get(NULL));

// move the subcursor in use to next position within the result subset (i.e. second position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a float (should be 2.2)
printf("Current element of subcursor is %f\n", (float *) hdfql_subcursor_get(NULL));
```

5.2.32 HDFQL_CURSOR_GET_TINYINT

Syntax

char *hdfql_cursor_get_tinyint(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a TINYINT. In other words, the current element is interpreted as a “char” C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a TINYINT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

char – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type char of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS TINYINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a char (should be 12)
printf("Current element of cursor is %d\n", *hdfql_cursor_get_tinyint(NULL));
```

5.2.33 HDFQL_SUBCURSOR_GET_TINYINT

Syntax

char *hdfql_subcursor_get_tinyint(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as a TINYINT. In other words, the current element is interpreted as a "char" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a TINYINT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

char – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length char of one dimension
(size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARTINYINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);
```

```
// display current element of the cursor in use as a char (should be 5)
printf("Current element of cursor is %d\n", *hdfql_cursor_get_tinyint(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a char (should be 5)
printf("Current element of subcursor is %d\n", *hdfql_subcursor_get_tinyint(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a char (should be 2)
printf("Current element of subcursor is %d\n", *hdfql_subcursor_get_tinyint(NULL));
```

5.2.34 HDFQL_CURSOR_GET_UNSIGNED_TINYINT

Syntax

unsigned char *hdfql_cursor_get_unsigned_tinyint(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as an UNSIGNED TINYINT. In other words, the current element is interpreted as an "unsigned char" C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a UNSIGNED TINYINT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

unsigned char – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type unsigned char of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED TINYINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned char (should be 12)
printf("Current element of cursor is %u\n", *hdfql_cursor_get_unsigned_tinyint(NULL));
```

5.2.35 HDFQL_SUBCURSOR_GET_UNSIGNED_TINYINT

Syntax

unsigned char *hdfql_subcursor_get_unsigned_tinyint(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as an UNSIGNED TINYINT. In other words, the current element is interpreted as an “unsigned char” C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as an UNSIGNED TINYINT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

unsigned char – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length unsigned char of one
dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED VARTINYINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned char (should be 5)
printf("Current element of cursor is %u\n", *hdfql_cursor_get_unsigned_tinyint(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned char (should be 5)
printf("Current element of subcursor is %u\n",
*hdfql_subcursor_get_unsigned_tinyint(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
```

```
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned char (should be 2)
printf("Current element of subcursor is %u\n",
*hdfql_subcursor_get_unsigned_tinyint(NULL));
```

5.2.36 HDFQL_CURSOR_GET_SMALLINT

Syntax

short *hdfql_cursor_get_smallint(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a SMALLINT. In other words, the current element is interpreted as a "short" C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a SMALLINT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

short – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type short of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS SMALLINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");
```

```
// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a short (should be 12)
printf("Current element of cursor is %d\n", *hdfql_cursor_get_smallint(NULL));
```

5.2.37 HDFQL_SUBCURSOR_GET_SMALLINT

Syntax

short *hdfql_subcursor_get_smallint(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as a SMALLINT. In other words, the current element is interpreted as a "short" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a SMALLINT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

short – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length short of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARSMALLINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a short (should be 5)
printf("Current element of cursor is %d\n", *hdfql_cursor_get_smallint(NULL));

// move the subcursor in use to next position within the result subset (i.e. first position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a short (should be 5)
printf("Current element of subcursor is %d\n", *hdfql_subcursor_get_smallint(NULL));

// move the subcursor in use to next position within the result subset (i.e. second position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a short (should be 2)
printf("Current element of subcursor is %d\n", *hdfql_subcursor_get_smallint(NULL));
```

5.2.38 HDFQL_CURSOR_GET_UNSIGNED_SMALLINT

Syntax

unsigned short *hdfql_cursor_get_unsigned_smallint(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as an UNSIGNED SMALLINT. In other words, the current element is interpreted as an “unsigned short” C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as an UNSIGNED SMALLINT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

unsigned short – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type unsigned short of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED SMALLINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned short (should be 12)
printf("Current element of cursor is %u\n", *hdfql_cursor_get_unsigned_smallint(NULL));
```

5.2.39 HDFQL_SUBCURSOR_GET_UNSIGNED_SMALLINT

Syntax

unsigned short *hdfql_subcursor_get_unsigned_smallint(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as an UNSIGNED SMALLINT. In other words, the current element is interpreted as an "unsigned short" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as an UNSIGNED SMALLINT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

unsigned short – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length unsigned short of one
dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED VARSMALLINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");
```

```
// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned short (should be 5)
printf("Current element of cursor is %u\n", *hdfql_cursor_get_unsigned_smallint(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned short (should be 5)
printf("Current element of subcursor is %u\n",
*hdfql_subcursor_get_unsigned_smallint(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned short (should be 2)
printf("Current element of subcursor is %u\n",
*hdfql_subcursor_get_unsigned_smallint(NULL));
```

5.2.40 HDFQL_CURSOR_GET_INT

Syntax

int *hdfql_cursor_get_int(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as an INT. In other words, the current element is interpreted as an “int” C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as an INT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and

Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

int – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type int of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS INT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned short (should be 12)
printf("Current element of cursor is %d\n", *hdfql_cursor_get_int(NULL));
```

5.2.41 HDFQL_SUBCURSOR_GET_INT

Syntax

int *hdfql_subcursor_get_int(HDFQL_CURSOR *cursor*)

Description

Get the current element of the subcursor in use as an INT. In other words, the current element is interpreted as an “int” C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as an INT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

int – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length int of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an int (should be 5)
printf("Current element of cursor is %d\n", *hdfql_cursor_get_int(NULL));

// move the subcursor in use to next position within the result subset (i.e. first position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an int (should be 5)
printf("Current element of subcursor is %d\n", *hdfql_subcursor_get_int(NULL));

// move the subcursor in use to next position within the result subset (i.e. second position)
hdfql_subcursor_next(NULL);
```

```
// display current element of the subcursor in use as an int (should be 2)
printf("Current element of subcursor is %d\n", *hdfql_subcursor_get_int(NULL));
```

5.2.42 HDFQL_CURSOR_GET_UNSIGNED_INT

Syntax

unsigned int *hdfql_cursor_get_unsigned_int(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as an UNSIGNED INT. In other words, the current element is interpreted as an "unsigned int" C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as an UNSIGNED INT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

unsigned int – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type unsigned int of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED INT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");
```

```
// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned int(should be 12)
printf("Current element of cursor is %u\n", *hdfql_cursor_get_unsigned_int(NULL));
```

5.2.43 HDFQL_SUBCURSOR_GET_UNSIGNED_INT

Syntax

unsigned int *hdfql_subcursor_get_unsigned_int(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as an UNSIGNED INT. In other words, the current element is interpreted as an "unsigned int" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as an UNSIGNED INT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

unsigned int – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length unsigned int of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED VARINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned int (should be 5)
printf("Current element of cursor is %u\n", *hdfql_cursor_get_unsigned_int(NULL));

// move the subcursor in use to next position within the result subset (i.e. first position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned int (should be 5)
printf("Current element of subcursor is %u\n", *hdfql_subcursor_get_unsigned_int(NULL));

// move the subcursor in use to next position within the result subset (i.e. second position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned int (should be 2)
printf("Current element of subcursor is %u\n", *hdfql_subcursor_get_unsigned_int(NULL));
```

5.2.44 HDFQL_CURSOR_GET_BIGINT

Syntax

long long *hdfql_cursor_get_bigint(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a BIGINT. In other words, the current element is interpreted as a "long long" C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a BIGINT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

long long – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type long long of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS BIGINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a long long (should be 12)
printf("Current element of cursor is %lld\n", *hdfql_cursor_get_bigint(NULL));
```

5.2.45 HDFQL_SUBCURSOR_GET_BIGINT

Syntax

long long *hdfql_subcursor_get_bigint(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as a BIGINT. In other words, the current element is interpreted as a "long long" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a BIGINT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

long long – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length long long of one dimension
(size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARBIGINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);
```

```
// display current element of the cursor in use as a long long (should be 5)
printf("Current element of cursor is %lld\n", *hdfql_cursor_get_bigint(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a long long (should be 5)
printf("Current element of subcursor is %lld\n", *hdfql_subcursor_get_bigint(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a long long (should be 2)
printf("Current element of subcursor is %lld\n", *hdfql_subcursor_get_bigint(NULL));
```

5.2.46 HDFQL_CURSOR_GET_UNSIGNED_BIGINT

Syntax

unsigned long long *hdfql_cursor_get_unsigned_bigint(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as an UNSIGNED BIGINT. In other words, the current element is interpreted as an “unsigned long long” C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as an UNSIGNED BIGINT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

unsigned long long – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type unsigned long long of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED BIGINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(12, 34, 23)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned long long (should be 12)
printf("Current element of cursor is %llu\n", *hdfql_cursor_get_unsigned_bigint(NULL));
```

5.2.47 HDFQL_SUBCURSOR_GET_UNSIGNED_BIGINT

Syntax

unsigned long long *hdfql_subcursor_get_unsigned_bigint(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as an UNSIGNED BIGINT. In other words, the current element is interpreted as an "unsigned long long" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as an UNSIGNED BIGINT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

unsigned long long – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length unsigned long long of one
dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS UNSIGNED VARBIGINT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((5, 2), (8), (4, 3, 9))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as an unsigned long long (should be 5)
printf("Current element of cursor is %llu\n", *hdfql_cursor_get_unsigned_bigint(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned long long (should be 5)
printf("Current element of subcursor is %llu\n",
*hdfql_subcursor_get_unsigned_bigint(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
```

```
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as an unsigned long long (should be 2)
printf("Current element of subcursor is %llu\n",
*hdfql_subcursor_get_unsigned_bigint(NULL));
```

5.2.48 HDFQL_CURSOR_GET_FLOAT

Syntax

float *hdfql_cursor_get_float(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a FLOAT. In other words, the current element is interpreted as a "float" C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a FLOAT. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

float – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type float of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS FLOAT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(5.5, 8.1, 4.9)");
```

```
// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a float (should be 5.5)
printf("Current element of cursor is %f\n", *hdfql_cursor_get_float(NULL));
```

5.2.49 HDFQL_SUBCURSOR_GET_FLOAT

Syntax

float *hdfql_subcursor_get_float(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as a FLOAT. In other words, the current element is interpreted as a “float” C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a FLOAT. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

float – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length float of one dimension
```

```
(size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARFLOAT(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((7.5, 3.1), (4.5), (4.9, 3.2, 9.7, 8.8))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a float (should be 7.5)
printf("Current element of cursor is %f\n", *hdfql_cursor_get_float(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a float (should be 7.5)
printf("Current element of subcursor is %f\n", *hdfql_subcursor_get_float(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a float (should be 3.1)
printf("Current element of subcursor is %f\n", *hdfql_subcursor_get_float(NULL));
```

5.2.50 HDFQL_CURSOR_GET_DOUBLE

Syntax

double *hdfql_cursor_get_double(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a DOUBLE. In other words, the current element is interpreted as a "double" C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a DOUBLE. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

double – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type double of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS DOUBLE(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(5.5, 8.1, 4.9)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a double (should be 5.5)
printf("Current element of cursor is %f\n", *hdfql_cursor_get_double(NULL));
```

5.2.51 HDFQL_SUBCURSOR_GET_DOUBLE

Syntax

double *hdfql_subcursor_get_double(HDFQL_CURSOR *_cursor_)

Description

Get the current element of the subcursor in use as a DOUBLE. In other words, the current element is interpreted as a "double" C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a DOUBLE. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

double – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length double of one dimension
(size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARDOUBLE(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES((7.5, 3.1), (4.5), (4.9, 3.2, 9.7, 8.8))");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);
```

```
// display current element of the cursor in use as a double (should be 7.5)
printf("Current element of cursor is %f\n", *hdfql_cursor_get_double(NULL));

// move the subcursor in use to next position within the result subset (i.e. first
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a double (should be 7.5)
printf("Current element of subcursor is %f\n", *hdfql_subcursor_get_double(NULL));

// move the subcursor in use to next position within the result subset (i.e. second
position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a double (should be 3.1)
printf("Current element of subcursor is %f\n", *hdfql_subcursor_get_double(NULL));
```

5.2.52 HDFQL_CURSOR_GET_CHAR

Syntax

char *hdfql_cursor_get_char(HDFQL_CURSOR **cursor*)

Description

Get the current element of a cursor named *cursor* as a CHAR. In other words, the current element is interpreted as a “char” C datatype and returned as a pointer of such type. If the result set is empty or the cursor is located before or after the first or last element of the result set, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element as a CHAR. If the pointer is NULL (in C), the current element of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the cursor in use is returned instead).

Return

char – pointer to the current element of the cursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type char of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS CHAR(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(Red)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a char (should be Red)
printf("Current element of cursor is %s\n", hdfql_cursor_get_char(NULL));
```

5.2.53 HDFQL_SUBCURSOR_GET_CHAR

Syntax

char *hdfql_subcursor_get_char(HDFQL_CURSOR **cursor*)

Description

Get the current element of the subcursor in use as a CHAR. In other words, the current element is interpreted as a “char” C datatype and returned as a pointer of such type. If the result subset is empty or the subcursor is located before or after the first or last element of the result subset, the returned element is NULL.

Parameter(s)

cursor – pointer to a cursor to get the current element of the subcursor in use as a CHAR. If the pointer is NULL (in C), the current element of the subcursor of the cursor in use is returned instead. The equivalent of a NULL pointer in C++, Java, Python, C# and Fortran is NULL, null, None, null and 0, respectively. While in C *cursor* is

mandatory, in C++, Java, Python, C# and Fortran it is optional (when not provided, the current element of the subcursor of the cursor in use is returned instead).

Return

char – pointer to the current element of the subcursor. If there is no current element, the pointer will be NULL.

Example(s)

```
// create a dataset named "my_dataset" of type variable-length char of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS VARCHAR(3)");

// insert (i.e. write) values into dataset "my_dataset"
hdfql_execute("INSERT INTO my_dataset VALUES(Red, Green, Blue)");

// select (i.e. read) dataset "my_dataset" and populate cursor in use with it
hdfql_execute("SELECT FROM my_dataset");

// move the cursor in use to next position within the result set (i.e. first position)
hdfql_cursor_next(NULL);

// display current element of the cursor in use as a char (should be Red)
printf("Current element of cursor is %s\n", hdfql_cursor_get_char(NULL));

// move the subcursor in use to next position within the result subset (i.e. first position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a char (should be Red)
printf("Current element of subcursor is %s\n", hdfql_subcursor_get_char(NULL));

// move the subcursor in use to next position within the result subset (i.e. second position)
hdfql_subcursor_next(NULL);

// display current element of the subcursor in use as a char (should be Green)
printf("Current element of subcursor is %s\n", hdfql_subcursor_get_char(NULL));
```

5.2.54 HDFQL_VARIABLE_REGISTER

Syntax

int hdfql_variable_register(const void *_variable_)

Description

Register a variable named *variable* for subsequent use. In other words, for HDFql to be able to read or write from/to a user-defined variable it must first be registered. If the operation was successful, *variable* is registered and a number is assigned to it. This number – calculated by HDFql – starts with zero and is incremented by one every time a new variable is registered. If *variable* is registered more than once, only one number is assigned to it (namely the number assigned upon the first registering). Of note, currently up to 16 variables can be registered at any given time. While in C, C++ and Fortran any variable may be registered (as long HDFql can properly read and write values from/to it), the following restrictions apply for other programming languages (supported by HDFql):

- In Java, only a variable that is an array of “byte”, “short”, “int”, “long”, “float”, “double" or “String” datatype or its corresponding wrapper class “Byte”, “Short”, “Integer”, “Long”, “Float” or “Double” may be registered. Any attempt to register a variable that is not an array or of the datatype/wrapper class previously enumerated will return an error.

- In Python, only a variable that is a NumPy array of “int8”, “uint8”, “int16”, “uint16”, “int32”, “uint32”, “int64”, “uint64”, “float32”, “float64” or “S*size*” datatype may be registered. Any attempt to register a variable that is not a NumPy array or of the datatype previously enumerated will return an error. Please refer to http://www.numpy.org for additional information.

- In C#, only a variable that is an array of datatype “SByte”, “Byte”, “Int16”, “UInt16”, “Int32”, “UInt32”, “Int64”, “UInt64”, “Single”, “Double” or “String” datatype or its alias “sbyte”, “byte”, “short”, “ushort”, “int”, “uint”, “long”, “ulong”, “float”, “double” or “string” may be registered. Any attempt to register a variable that is not an array or of the datatype/alias previously enumerated will return an error.

In general, it is advisable to register a variable just before executing the HDFql operation which employs it, and to unregister it as soon as it is no longer used (this is especially relevant in C# where variables are pinned when registered and thus cannot be moved by the Garbage Collector). This can be done via the function hdfql_variable_unregister.

Parameter(s)

variable – variable to register for subsequent use.

Return

int – depending on the success in registering the variable for subsequent use, it can either be ≥ 0 (i.e. the number assigned to the variable when successfully registered), HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_FULL.

Example(s)

```
// declare variables
char script[1024];
short data[3];

// create a dataset named "my_dataset" of type short of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS SMALLINT(3)");

// assign values to variable "data"
data[0] = 21;
data[1] = 18;
data[2] = 75;

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to insert (i.e. write) values from variable "data" into dataset
"my_dataset"
sprintf(script, "INSERT INTO my_dataset VALUES FROM MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);
```

5.2.55 HDFQL_VARIABLE_UNREGISTER

Syntax

int hdfql_variable_unregister(const void **variable*)

Description

Unregister a variable named *variable*. In other words, HDFql will free up any memory that may have been allocated to manage the variable as well as the number assigned to it (the number may then be assigned to a new variable registered subsequently). In general, it is advisable to unregister a variable as soon as it is no longer used by HDFql (this is especially relevant in C# as variables are unpinned when unregistered and thus may again be moved by the Garbage Collector). If *variable* has never been registered or has already been unregistered, an error is returned.

Parameter(s)

variable – variable to unregister.

Return

int – depending on the success in unregistering the variable, it can either be HDFQL_SUCCESS, HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// declare variables
char script[1024];
short data[3];

// create a dataset named "my_dataset" of type short of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset AS SMALLINT(3)");

// assign values to variable "data"
data[0] = 21;
data[1] = 18;
data[2] = 75;

// register variable "data" for subsequent use (by HDFql)
```

```
hdfql_variable_register(&data);

// prepare script to insert (i.e. write) values from variable "data" into dataset
"my_dataset"
sprintf(script, "INSERT INTO my_dataset VALUES FROM MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// unregister variable "data" as it is no longer used/needed (by HDFql)
hdfql_variable_unregister(&data);
```

5.2.56 HDFQL_VARIABLE_GET_NUMBER

Syntax

int hdfql_variable_get_number(const void *_variable_)

Description

Get the number of a variable named *variable*. This refers to the number that was calculated by HDFql and assigned to the variable upon registering it with the function hdfql_variable_register. If *variable* has never been registered or has been unregistered, an error is returned.

Parameter(s)

variable – variable to get the number (calculated by HDFql) assigned to it.

Return

int – depending on the success in getting the number assigned to the variable, it can either be ≥ 0, HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// declare variables
short data0[3];
```

```
float data1[5];

// register variable "data0" for subsequent use (by HDFql)
hdfql_variable_register(&data0);

// register variable "data1" for subsequent use (by HDFql)
hdfql_variable_register(&data1);

// display number of variable "data0" (should be 0)
printf("Number of variable is %d\n", hdfql_variable_get_number(&data0));

// display number of variable "data1" (should be 1)
printf("Number of variable is %d\n", hdfql_variable_get_number(&data1));
```

5.2.57 HDFQL_VARIABLE_GET_DATATYPE

Syntax

int hdfql_variable_get_datatype(const void **variable*)

Description

Get the datatype of a variable named *variable*. This function should help the programmer to better handle the content stored in *variable*. The datatype refers to the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation redirected into memory – and not the datatype of *variable* declared in the program. If *variable* has never been registered, populated (through the redirection of the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation into memory), or in case it has been unregistered, the returned datatype is undefined (HDFQL_UNDEFINED). Please refer to Table 6.3 for a complete enumeration of HDFql datatypes.

Parameter(s)

variable – variable to get its datatype.

Return

int – depending on the success in getting the datatype of the variable, it can either be HDFQL_TINYINT, HDFQL_UNSIGNED_TINYINT, HDFQL_SMALLINT, HDFQL_UNSIGNED_SMALLINT, HDFQL_INT,

HDFQL_UNSIGNED_INT, HDFQL_BIGINT, HDFQL_UNSIGNED_BIGINT, HDFQL_FLOAT, HDFQL_DOUBLE, HDFQL_CHAR, HDFQL_VARTINYINT, HDFQL_UNSIGNED_VARTINYINT, HDFQL_VARSMALLINT, HDFQL_UNSIGNED_VARSMALLINT, HDFQL_VARINT, HDFQL_UNSIGNED_VARINT, HDFQL_VARBIGINT, HDFQL_UNSIGNED_VARBIGINT, HDFQL_VARFLOAT, HDFQL_VARDOUBLE, HDFQL_VARCHAR , HDFQL_OPAQUE, HDFQL_UNDEFINED, HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// declare variables
char script[1024];
char data[1024];

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to get current working directory and populate variable "data" with it
sprintf(script, "SHOW USE DIRECTORY INTO MEMORY %u", hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// display datatype of variable "data" (should be 1024 - i.e. HDFQL_CHAR)
printf("Datatype of variable is %d\n", hdfql_variable_get_datatype(&data));
```

5.2.58 HDFQL_VARIABLE_GET_COUNT

Syntax

int hdfql_variable_get_count(const void **variable*)

Description

Get the number of elements (i.e. result set size) stored in a variable named *variable*. This function should help the programmer to better handle the content stored in *variable*. If the result set stores data from a dataset or attribute that does not have a dimension (i.e. if it is scalar), the returned number of elements is one. Otherwise, if the result set stores data from a dataset or attribute that has dimensions, the returned number of elements equals the multiplication of all its dimensions' sizes (e.g. if a variable stores a result set of two

dimensions of size 10x3, the number of elements is 30). If *variable* has never been populated (through the redirection of the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation into memory), the returned number of elements is zero.

Parameter(s)

variable – variable to get its number of elements (i.e. resut set size).

Return

int – depending on the success in getting the number of elements of the variable, it can either be ≥ 0, HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// declare variables
char script[1024];
int data[5][3];

// create a dataset named "my_dataset" of type int of two dimensions (size 5x3)
hdfql_execute("CREATE DATASET my_dataset AS INT(5, 3)");

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to select (i.e. read) dataset "my_dataset" and populate variable "data"
with it
sprintf(script, "SELECT FROM my_dataset INTO MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// display number of elements in variable "data" (should be 15 - i.e. 5x3)
printf("Number of elements in variable is %d\n", hdfql_variable_get_count(&data));
```

5.2.59 HDFQL_VARIABLE_GET_SIZE

Syntax

int hdfql_variable_get_size(const void *_variable_)

Description

Get the size (in bytes) of a variable named *variable*. This function should help the programmer to better handle the content stored in *variable*. The size (in bytes) refers to the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation redirected into memory – and not the size (in bytes) that *variable* has in the program. If *variable* has never been registered or has been unregistered, an error is returned. If *variable* has never been populated (through the redirection of the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation into memory), the returned size is zero. Please refer to Table 6.3 for a complete enumeration of HDFql datatypes and their corresponding sizes (in bytes).

Parameter(s)

variable – variable to get its size (in bytes).

Return

int – depending on the success in getting the size (in bytes) of the variable, it can either be ≥ 0, HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// declare variables
char script[1024];
int data[5][3];

// create a dataset named "my_dataset" of type int of two dimensions (size 5x3)
hdfql_execute("CREATE DATASET my_dataset AS INT(5, 3)");

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);
```

```
// prepare script to select (i.e. read) dataset "my_dataset" and populate variable "data"
with it
sprintf(script, "SELECT FROM my_dataset INTO MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// display size (in bytes) of variable "data" (should be 60 - i.e. 5x3x4)
printf("Size (in bytes) of variable is %d\n", hdfql_variable_get_size(&data));
```

5.2.60 HDFQL_VARIABLE_GET_DIMENSION_COUNT

Syntax

int hdfql_variable_get_dimension_count(const void **variable*)

Description

Get the number of dimensions of a variable named *variable*. This function should help the programmer to better handle the content stored in *variable*. The number of dimensions refers to the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation redirected into memory – and not the number of dimensions that *variable* has in the program. If *variable* has never been registered or has been unregistered, an error is returned. If *variable* has never been populated (through the redirection of the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation into memory), the returned number of dimensions is zero.

Parameter(s)

variable – variable to get its number of dimensions.

Return

int – depending on the success in getting the number of dimensions of the variable, it can either be ≥ 0, HDFQL_ERROR_NO_ADDRESS or HDFQL_ERROR_NOT_REGISTERED.

Example(s)

```
// declare variables
char script[1024];
int data[5][3];

// create a dataset named "my_dataset" of type int of two dimensions (size 5x3)
hdfql_execute("CREATE DATASET my_dataset AS INT(5, 3)");

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to select (i.e. read) dataset "my_dataset" and populate variable "data"
with it
sprintf(script, "SELECT FROM my_dataset INTO MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// display number of dimensions of variable "data" (should be 2)
printf("Number of dimensions in variable is %d\n",
hdfql_variable_get_dimension_count(&data));
```

5.2.61 HDFQL_VARIABLE_GET_DIMENSION

Syntax

int hdfql_variable_get_dimension(const void *_variable_, int _index_)

Description

Get the size of a certain dimension specified in *index* of a variable named *variable*. This function should help the programmer to better handle the content stored in *variable*. The size of a certain dimension refers to the result of a DATA QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation redirected into memory – and not the size of a certain dimension that *variable* has in the program. The index of the first dimension is zero (*index* must be between 0 and the value returned by hdfql_variable_get_dimension_count – 1 inclusive). If *variable* has never been registered, populated (through the redirection of the result of a DATA

QUERY LANGUAGE (DQL) or DATA INTROSPECTION LANGUAGE (DIL) operation into memory), or in case it has been unregistered, an error is returned.

Parameter(s)

variable – variable to get the size of one of its dimensions.

index – index of the dimension to get its size.

Return

int – depending on the success in getting the size of a certain dimension of the variable, it can either be ≥ 0, HDFQL_ERROR_NO_ADDRESS, HDFQL_ERROR_NOT_REGISTERED or HDFQL_ERROR_OUTSIDE_LIMIT.

Example(s)

```
// declare variables
char script[1024];
int data[5][3];

// create a dataset named "my_dataset" of type int of two dimensions (size 5x3)
hdfql_execute("CREATE DATASET my_dataset AS INT(5, 3)");

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to select (i.e. read) dataset "my_dataset" and populate variable "data"
with it
sprintf(script, "SELECT FROM my_dataset INTO MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// display size of the first dimension of variable "data" (should be 5)
printf("Size of first dimension of variable is %d\n", hdfql_variable_get_dimension(0));

// display size of the second dimension of variable "data" (should be 3)
printf("Size of second dimension of variable is %d\n", hdfql_variable_get_dimension(1));
```

5.3 EXAMPLES

The following subsections present practical examples on how to use (some of) the HDFql functions previously described in the C, C++, Java, Python, C# and Fortran programming languages. The output of executing these examples can be seen in subsection OUTPUT.

5.3.1 C

```
// include HDFql C header file (make sure it can be found by the C compiler)
#include "HDFql.h"

int main(int argc, char *argv[])
{

    // declare variables
    HDFQL_CURSOR my_cursor;
    char script[1024];
    int values[3][2];
    int x;
    int y;

    // display HDFql version in use
    printf("HDFql version: %s\n", HDFQL_VERSION);

    // create an HDF file named "example_c.h5" and use (i.e. open) it
    hdfql_execute("CREATE FILE example_c.h5");
    hdfql_execute("USE FILE example_c.h5");

    // populate HDFql default cursor with name of the HDF file in use and display it
    hdfql_execute("SHOW USE FILE");
    hdfql_cursor_first(NULL);
    printf("File in use: %s\n", hdfql_cursor_get_char(NULL));

    // create an attribute named "example_attribute" of type float with a value of 12.4
    hdfql_execute("CREATE ATTRIBUTE example_attribute AS FLOAT DEFAULT 12.4");

    // select (i.e. read) attribute "example_attribute" and display its value
    hdfql_execute("SELECT FROM example_attribute");
```

```
    hdfql_cursor_first(NULL);
    printf("Attribute value: %f\n", *hdfql_cursor_get_float(NULL));

    // create a dataset named "example_dataset" of type int of two dimensions (size 3x2)
    hdfql_execute("CREATE DATASET example_dataset AS INT(3, 2)");

    // populate variable "values" with certain values
    for(x = 0; x < 3; x++)
    {
        for(y = 0; y < 2; y++)
        {
            values[x][y] = x * 2 + y + 1;
        }
    }

    // register variable "values" for subsequent use (by HDFql)
    hdfql_variable_register(&values);

    // insert (i.e. write) content of variable "values" into dataset "example_dataset"
    sprintf(script, "INSERT INTO example_dataset VALUES FROM MEMORY %u",
hdfql_variable_get_number(&values));
    hdfql_execute(script);

    // populate variable "values" with zeros (i.e. reset variable)
    for(x = 0; x < 3; x++)
    {
        for(y = 0; y < 2; y++)
        {
            values[x][y] = 0;
        }
    }

    // select (i.e. read) dataset "example_dataset" into variable "values"
    sprintf(script, "SELECT FROM example_dataset INTO MEMORY %u",
hdfql_variable_get_number(&values));
    hdfql_execute(script);

    // unregister variable "values" as it is no longer used/needed (by HDFql)
    hdfql_variable_unregister(&values);

    // display content of variable "values"
```

```
    printf("Variable:\n");
    for(x = 0; x < 3; x++)
    {
        for(y = 0; y < 2; y++)
        {
            printf("%d\n", values[x][y]);
        }
    }

    // another way to select (i.e. read) dataset "example_dataset" using HDFql default
cursor
    hdfql_execute("SELECT FROM example_dataset");

    // display content of HDFql default cursor
    printf("Cursor:\n");
    while(hdfql_cursor_next(NULL) == HDFQL_SUCCESS)
    {
        printf("%d\n", *hdfql_cursor_get_int(NULL));
    }

    // initialize cursor "my_cursor" and use it
    hdfql_cursor_initialize(&my_cursor);
    hdfql_cursor_use(&my_cursor);

    // populate cursor "my_cursor" with size of dataset "example_dataset" and display it
    hdfql_execute("SHOW SIZE example_dataset");
    hdfql_cursor_first(NULL);
    printf("Dataset size: %d\n", *hdfql_cursor_get_int(NULL));

    return 0;

}
```

5.3.2 C++

```
// include HDFql C++ header file (make sure it can be found by the C++ compiler)
#include <iostream>
#include "HDFql.hpp"
```

```
int main(int argc, char *argv[])
{

    // declare variables
    HDFql::Cursor myCursor;
    char script[1024];
    int values[3][2];
    int x;
    int y;

    // display HDFql version in use
    std::cout << "HDFql version: " << HDFql::Version << std::endl;

    // create an HDF file named "example_cpp.h5" and use (i.e. open) it
    HDFql::execute("CREATE FILE example_cpp.h5");
    HDFql::execute("USE FILE example_cpp.h5");

    // populate HDFql default cursor with name of the HDF file in use and display it
    HDFql::execute("SHOW USE FILE");
    HDFql::cursorFirst();
    std::cout << "File in use: " << HDFql::cursorGetChar() << std::endl;

    // create an attribute named "example_attribute" of type float with a value of 12.4
    HDFql::execute("CREATE ATTRIBUTE example_attribute AS FLOAT DEFAULT 12.4");

    // select (i.e. read) attribute "example_attribute" and display its value
    HDFql::execute("SELECT FROM example_attribute");
    HDFql::cursorFirst();
    std::cout << "Attribute value: " << *HDFql::cursorGetFloat() << std::endl;

    // create a dataset named "example_dataset" of type int of two dimensions (size 3x2)
    HDFql::execute("CREATE DATASET example_dataset AS INT(3, 2)");

    // populate variable "values" with certain values
    for(x = 0; x < 3; x++)
    {
        for(y = 0; y < 2; y++)
        {
            values[x][y] = x * 2 + y + 1;
        }
    }
```

```
    // register variable "values" for subsequent use (by HDFql)
    HDFql::variableRegister(&values);

    // insert (i.e. write) content of variable "values" into dataset "example_dataset"
    sprintf(script, "INSERT INTO example_dataset VALUES FROM MEMORY %u",
HDFql::variableGetNumber(&values));
    HDFql::execute(script);

    // populate variable "values" with zeros (i.e. reset variable)
    for(x = 0; x < 3; x++)
    {
        for(y = 0; y < 2; y++)
        {
            values[x][y] = 0;
        }
    }

    // select (i.e. read) dataset "example_dataset" into variable "values"
    sprintf(script, "SELECT FROM example_dataset INTO MEMORY %u",
HDFql::variableGetNumber(&values));
    HDFql::execute(script);

    // unregister variable "values" as it is no longer used/needed (by HDFql)
    HDFql::variableUnregister(&values);

    // display content of variable "values"
    std::cout << "Variable:" << std::endl;
    for(x = 0; x < 3; x++)
    {
        for(y = 0; y < 2; y++)
        {
            std::cout << values[x][y] << std::endl;
        }
    }

    // another way to select (i.e. read) dataset "example_dataset" using HDFql default
cursor
    HDFql::execute("SELECT FROM example_dataset");

    // display content of HDFql default cursor
```

```
    std::cout << "Cursor:" << std::endl;
    while(HDFql::cursorNext() == HDFql::Success)
    {
        std::cout << *HDFql::cursorGetInt() << std::endl;
    }

    // use cursor "myCursor"
    HDFql::cursorUse(&myCursor);

    // populate cursor "myCursor" with size of dataset "example_dataset" and display it
    HDFql::execute("SHOW SIZE example_dataset");
    HDFql::cursorFirst();
    std::cout << "Dataset size: " << *HDFql::cursorGetInt() << std::endl;

    return 0;

}
```

5.3.3 JAVA

```
public class HDFqlExample
{
    public static void main(String args[])
    {
        // declare variables
        HDFqlCursor myCursor;
        int values[][];
        int x;
        int y;

        // load HDFql shared library (make sure it can be found by the JVM)
        System.loadLibrary("HDFql");

        // display HDFql version in use
        System.out.println("HDFql version: " + HDFql.VERSION);

        // create an HDF file named "example_java.h5" and use (i.e. open) it
        HDFql.execute("CREATE FILE example_java.h5");
        HDFql.execute("USE FILE example_java.h5");
```

```
        // populate HDFql default cursor with name of the HDF file in use and display it
        HDFql.execute("SHOW USE FILE");
        HDFql.cursorFirst();
        System.out.println("File in use: " + HDFql.cursorGetChar());

        // create an attribute named "example_attribute" of type float with a value of 12.4
        HDFql.execute("CREATE ATTRIBUTE example_attribute AS FLOAT DEFAULT 12.4");

        // select (i.e. read) attribute "example_attribute" and display its value
        HDFql.execute("SELECT FROM example_attribute");
        HDFql.cursorFirst();
        System.out.println("Attribute value: " + HDFql.cursorGetFloat());

        // create a dataset named "example_dataset" of type int of two dimensions (size
3x2)
        HDFql.execute("CREATE DATASET example_dataset AS INT(3, 2)");

        // create variable "values" and populate it with certain values
        values = new int[3][2];
        for(x = 0; x < 3; x++)
        {
            for(y = 0; y < 2; y++)
            {
                values[x][y] = x * 2 + y + 1;
            }
        }

        // register variable "values" for subsequent use (by HDFql)
        HDFql.variableRegister(values);

        // insert (i.e. write) content of variable "values" into dataset "example_dataset"
        HDFql.execute("INSERT INTO example_dataset VALUES FROM MEMORY " +
HDFql.variableGetNumber(values));

        // populate variable "values" with zeros (i.e. reset variable)
        for(x = 0; x < 3; x++)
        {
            for(y = 0; y < 2; y++)
            {
                values[x][y] = 0;
```

```
            }
        }

        // select (i.e. read) dataset "example_dataset" into variable "values"
        HDFql.execute("SELECT FROM example_dataset INTO MEMORY " +
HDFql.variableGetNumber(values));

        // unregister variable "values" as it is no longer used/needed (by HDFql)
        HDFql.variableUnregister(values);

        // display content of variable "values"
        System.out.println("Variable:");
        for(x = 0; x < 3; x++)
        {
            for(y = 0; y < 2; y++)
            {
                System.out.println(values[x][y]);
            }
        }

        // another way to select (i.e. read) dataset "example_dataset" using HDFql default
cursor
        HDFql.execute("SELECT FROM example_dataset");

        // display content of HDFql default cursor
        System.out.println("Cursor:");
        while(HDFql.cursorNext() == HDFql.SUCCESS)
        {
            System.out.println(HDFql.cursorGetInt());
        }

        // create cursor "myCursor" and use it
        myCursor = new HDFqlCursor();
        HDFql.cursorUse(myCursor);

        // populate cursor "myCursor" with size of dataset "example_dataset" and display it
        HDFql.execute("SHOW SIZE example_dataset");
        HDFql.cursorFirst();
        System.out.println("Dataset size: " + HDFql.cursorGetInt());
    }
}
```

5.3.4 PYTHON

```
# import HDFql module (make sure it can be found by the Python interpreter)
import HDFql
import numpy

# display HDFql version in use
print("HDFql version: %s" % HDFql.VERSION)

# create an HDF file named "example_python.h5" and use (i.e. open) it
HDFql.execute("CREATE FILE example_python.h5")
HDFql.execute("USE FILE example_python.h5")

# populate HDFql default cursor with name of the HDF file in use and display it
HDFql.execute("SHOW USE FILE")
HDFql.cursor_first()
print("File in use: %s" % HDFql.cursor_get_char())

# create an attribute named "example_attribute" of type float with a value of 12.4
HDFql.execute("CREATE ATTRIBUTE example_attribute AS FLOAT DEFAULT 12.4")

# select (i.e. read) attribute "example_attribute" and display its value
HDFql.execute("SELECT FROM example_attribute")
HDFql.cursor_first()
print("Attribute value: %f" % HDFql.cursor_get_float())

# create a dataset named "example_dataset" of type int of two dimensions (size 3x2)
HDFql.execute("CREATE DATASET example_dataset AS INT(3, 2)")

# create variable "values" and populate it with certain values
values = numpy.zeros((3, 2), dtype = numpy.int32)
for x in range(3):
    for y in range(2):
        values[x][y] = x * 2 + y + 1

# register variable "values" for subsequent use (by HDFql)
HDFql.variable_register(values)

# insert (i.e. write) content of variable "values" into dataset "example_dataset"
```

```
HDFql.execute("INSERT INTO example_dataset VALUES FROM MEMORY %d" %
HDFql.variable_get_number(values))

# populate variable "values" with zeros (i.e. reset variable)
for x in range(3):
    for y in range(2):
        values[x][y] = 0

# select (i.e. read) dataset "example_dataset" into variable "values"
HDFql.execute("SELECT FROM example_dataset INTO MEMORY %d" %
HDFql.variable_get_number(values))

# unregister variable "values" as it is no longer used/needed (by HDFql)
HDFql.variable_unregister(values)

# display content of variable "values"
print("Variable:")
for x in range(3):
    for y in range(2):
        print(values[x][y])

# another way to select (i.e. read) dataset "example_dataset" using HDFql default cursor
HDFql.execute("SELECT FROM example_dataset")

# display content of HDFql default cursor
print("Cursor:")
while HDFql.cursor_next() == HDFql.SUCCESS:
    print(HDFql.cursor_get_int())

# create cursor "my_cursor" and use it
my_cursor = HDFql.Cursor()
HDFql.cursor_use(my_cursor)

# populate cursor "my_cursor" with size of dataset "example_dataset" and display it
HDFql.execute("SHOW SIZE example_dataset")
HDFql.cursor_first()
print("Dataset size: %d" % HDFql.cursor_get_int())
```

5.3.5 C#

```
public class HDFqlExample
{
    public static void Main(string []args)
    {
        // declare variables
        HDFqlCursor myCursor;
        int [,]values;
        int x;
        int y;

        // display HDFql version in use
        System.Console.WriteLine("HDFql version: {0}", HDFql.Version);

        // create an HDF file named "example_csharp.h5" and use (i.e. open) it
        HDFql.Execute("CREATE FILE example_csharp.h5");
        HDFql.Execute("USE FILE example_csharp.h5");

        // populate HDFql default cursor with name of the HDF file in use and display it
        HDFql.Execute("SHOW USE FILE");
        HDFql.CursorFirst();
        System.Console.WriteLine("File in use: {0}", HDFql.CursorGetChar());

        // create an attribute named "example_attribute" of type float with a value of 12.4
        HDFql.Execute("CREATE ATTRIBUTE example_attribute AS FLOAT DEFAULT 12.4");

        // select (i.e. read) attribute "example_attribute" and display its value
        HDFql.Execute("SELECT FROM example_attribute");
        HDFql.CursorFirst();
        System.Console.WriteLine("Attribute value: {0}", HDFql.CursorGetFloat());

        // create a dataset named "example_dataset" of type int of two dimensions (size
3x2)
        HDFql.Execute("CREATE DATASET example_dataset AS INT(3, 2)");

        // create variable "values" and populate it with certain values
        values = new int[3, 2];
        for(x = 0; x < 3; x++)
        {
            for(y = 0; y < 2; y++)
```

```
            {
                values[x, y] = x * 2 + y + 1;
            }
        }

        // register variable "values" for subsequent use (by HDFql)
        HDFql.VariableRegister(values);

        // insert (i.e. write) content of variable "values" into dataset "example_dataset"
        HDFql.Execute("INSERT INTO example_dataset VALUES FROM MEMORY " +
HDFql.VariableGetNumber(values));

        // populate variable "values" with zeros (i.e. reset variable)
        for(x = 0; x < 3; x++)
        {
            for(y = 0; y < 2; y++)
            {
                values[x, y] = 0;
            }
        }

        // select (i.e. read) dataset "example_dataset" into variable "values"
        HDFql.Execute("SELECT FROM example_dataset INTO MEMORY " +
HDFql.VariableGetNumber(values));

        // unregister variable "values" as it is no longer used/needed (by HDFql)
        HDFql.VariableUnregister(values);

        // display content of variable "values"
        System.Console.WriteLine("Variable:");
        for(x = 0; x < 3; x++)
        {
            for(y = 0; y < 2; y++)
            {
                System.Console.WriteLine(values[x, y]);
            }
        }

        // another way to select (i.e. read) dataset "example_dataset" using HDFql default
cursor
        HDFql.Execute("SELECT FROM example_dataset");
```

```
        // display content of HDFql default cursor
        System.Console.WriteLine("Cursor:");
        while(HDFql.CursorNext() == HDFql.Success)
        {
            System.Console.WriteLine(HDFql.CursorGetInt());
        }

        // create cursor "myCursor" and use it
        myCursor = new HDFqlCursor();
        HDFql.CursorUse(myCursor);

        // populate cursor "myCursor" with size of dataset "example_dataset" and display it
        HDFql.Execute("SHOW SIZE example_dataset");
        HDFql.CursorFirst();
        System.Console.WriteLine("Dataset size: {0}", HDFql.CursorGetInt());
    }
}
```

5.3.6 FORTRAN

```
PROGRAM HDFqlExample

    ! use HDFql module (make sure it can be found by the Fortran compiler)
    USE HDFql

    ! declare variables
    TYPE(HDFQL_CURSOR) :: my_cursor
    CHARACTER :: string_number
    INTEGER, DIMENSION(3, 2) :: values
    INTEGER :: state
    INTEGER :: x
    INTEGER :: y

    ! display HDFql version in use
    WRITE(*, *) "HDFql version: ", HDFQL_VERSION

    ! create an HDF file named "example_fortran.h5" and use (i.e. open) it
    state = hdfql_execute("CREATE FILE example_fortran.h5" // CHAR(0))
```

```
    state = hdfql_execute("USE FILE example_fortran.h5" // CHAR(0))

    ! populate HDFql default cursor with name of the HDF file in use and display it
    state = hdfql_execute("SHOW USE FILE" // CHAR(0))
    state = hdfql_cursor_first()
    WRITE(*, *) "File in use: ", hdfql_cursor_get_char()

    ! create an attribute named "example_attribute" of type float with a value of 12.4
    state = hdfql_execute("CREATE ATTRIBUTE example_attribute AS FLOAT DEFAULT 12.4" //
CHAR(0))

    ! select (i.e. read) attribute "example_attribute" and display its value
    state = hdfql_execute("SELECT FROM example_attribute" // CHAR(0))
    state = hdfql_cursor_first()
    WRITE(*, *) "Attribute value: ", hdfql_cursor_get_float()

    ! create a dataset named "example_dataset" of type int of two dimensions (size 3x2)
    state = hdfql_execute("CREATE DATASET example_dataset AS INT(3, 2)" // CHAR(0))

    ! populate variable "values" with certain values
    DO x = 1, 2
        DO y = 1, 3
            values(y, x) = x * 3 + y - 3
        END DO
    END DO

    ! register variable "values" for subsequent use (by HDFql)
    state = hdfql_variable_register(LOC(values))
    WRITE(string_number, "(I0)") state

    ! insert (i.e. write) content of variable "values" into dataset "example_dataset"
    state = hdfql_execute("INSERT INTO example_dataset VALUES FROM MEMORY " //
string_number // CHAR(0))

    ! populate variable "values" with zeros (i.e. reset variable)
    DO x = 1, 2
        DO y = 1, 3
            values(y, x) = 0
        END DO
    END DO
```

```
    ! select (i.e. read) dataset "example_dataset" into variable "values"
    state = hdfql_execute("SELECT FROM example_dataset INTO MEMORY " // string_number //
CHAR(0))

    ! unregister variable "values" as it is no longer used/needed (by HDFql)
    state = hdfql_variable_unregister(LOC(values))

    ! display content of variable "values"
    WRITE(*, *) "Variable:"
    DO x = 1, 2
        DO y = 1, 3
            WRITE(*, *) values(y, x)
        END DO
    END DO

    ! another way to select (i.e. read) dataset "example_dataset" using HDFql default
cursor
    state = hdfql_execute("SELECT FROM example_dataset" // CHAR(0))

    ! display content of HDFql default cursor
    WRITE(*, *) "Cursor:"
    DO WHILE(hdfql_cursor_next() .EQ. HDFQL_SUCCESS)
        WRITE(*, *) hdfql_cursor_get_int()
    END DO

    ! use cursor "my_cursor"
    state = hdfql_cursor_use(my_cursor)

    ! populate cursor "my_cursor" with size of dataset "example_dataset" and display it
    state = hdfql_execute("SHOW SIZE example_dataset" // CHAR(0))
    state = hdfql_cursor_first()
    WRITE(*, *) "Dataset size: ", hdfql_cursor_get_int()

END PROGRAM
```

5.3.7 OUTPUT

```
HDFql version: 1.4.0
File in use: example_c.h5
```

```
Attribute value: 12.400000
Variable:
1
2
3
4
5
6
Cursor:
1
2
3
4
5
6
Dataset size: 24
```

6. LANGUAGE

HDFql is a high-level language to manage HDF files in a simple and natural way. It was designed to be similar to SQL (wherever possible) so that its learning effort is kept at minimum while still providing great power and flexibility to the programmer. This chapter describes datatypes, post-processing options to further process result sets, and operations (i.e. the language itself) available in HDFql. It also introduces text formatting conventions used throughout this chapter to describe HDFql operations (Table 6.1), and a summary of existing operations (Table 6.2). Before continuing, it is highly recommended to first read the HDF User's Guide available at http://www.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide.pdf to facilitate the understanding of the current chapter.

Convention	Description	Example
Bold	Keyword that must be typed exactly as shown	**CREATE**
Italic	Value that the programmer must supply	*dataset_name*
Between brackets ([])	Optional keyword/value	[**DATASET**]
Between braces ({})	Logical grouping of keywords/values	{[**TRUNCATE**] **BINARY FILE** *file_name*}
Separated by pipe (|)	Set of keywords/values from which one must be chosen	**GROUP** | **DATASET** | **ATTRIBUTE**
Ellipsis (…)	Keyword/value that can be repeated/supplied several times	*dim1, …, dimX*

Table 6.1 – HDFql operations text formatting conventions

Operation	Description
CREATE DIRECTORY	Create a directory
CREATE FILE	Create an HDF file
CREATE GROUP	Create an HDF group

CREATE DATASET	Create an HDF dataset
CREATE ATTRIBUTE	Create an HDF attribute
CREATE [SOFT | HARD] LINK	Create an HDF soft or hard link
CREATE EXTERNAL LINK	Create an HDF external link
ALTER DIMENSION	Alter (i.e. change) dimensions of an existing HDF dataset
RENAME DIRECTORY	Rename (or move) an existing directory
RENAME FILE	Rename (or move) an existing file
RENAME [GROUP | DATASET | ATTRIBUTE]	Rename (or move) an existing HDF group, dataset or attribute
COPY FILE	Copy an existing file
COPY [GROUP | DATASET | ATTRIBUTE]	Copy an existing HDF group, dataset or attribute
DROP DIRECTORY	Drop (i.e. delete) an existing directory
DROP FILE	Drop (i.e. delete) an existing file
DROP [GROUP | DATASET | ATTRIBUTE]	Drop (i.e. delete) an existing HDF group, dataset or attribute
INSERT	Insert (i.e. write) data into an HDF dataset or attribute
SELECT	Select (i.e. read) data from an HDF dataset or attribute
SHOW FILE VALIDITY	Get validity of a file (i.e. whether it is a valid HDF file or not)
SHOW USE DIRECTORY	Get working directory currently in use
SHOW USE FILE	Get HDF file currently in use
SHOW ALL USE FILE	Get all HDF files in use (i.e. open)
SHOW USE GROUP	Get HDF group currently in use
SHOW [GROUP | DATASET | ATTRIBUTE]	Get HDF objects (i.e. groups, datasets or attributes) or check existence of an object
SHOW TYPE	Get type of an HDF object (i.e. group, dataset or attribute)
SHOW STORAGE TYPE	Get storage type of an HDF dataset
SHOW [DATASET | ATTRIBUTE] DATATYPE	Get datatype of an HDF dataset or attribute

SHOW [DATASET | ATTRIBUTE] ENDIANNESS	Get endianness of an HDF dataset or attribute
SHOW [DATASET | ATTRIBUTE] CHARSET	Get charset of an HDF dataset or attribute
SHOW STORAGE DIMENSION	Get storage dimensions of an HDF dataset
SHOW [DATASET | ATTRIBUTE] DIMENSION	Get dimensions of an HDF dataset or attribute
SHOW [DATASET | ATTRIBUTE] MAX DIMENSION	Get maximum dimensions of an HDF dataset or attribute
SHOW [ATTRIBUTE] ORDER	Get (creation) order strategy of an HDF group or dataset
SHOW [DATASET | ATTRIBUTE] TAG	Get tag of an HDF dataset or attribute named *object_name*
SHOW FILE SIZE	Get size (in bytes) of a file
SHOW [DATASET | ATTRIBUTE] SIZE	Get size (in bytes) of an HDF dataset or attribute
SHOW RELEASE DATE	Get release date of HDFql library
SHOW HDFQL VERSION	Get version of HDFql library
SHOW HDF VERSION	Get version of HDF library used by HDFql
SHOW PCRE VERSION	Get version of PCRE library used by HDFql
SHOW ZLIB VERSION	Get version of ZLIB library used by HDFql
SHOW DIRECTORY	Get directory names within a directory
SHOW FILE	Get file names within a directory or check existence of a file
SHOW MAC ADDRESS	Get MAC address(es) of the machine where HDFql is executed
SHOW EXECUTE STATUS	Get execution status of the last operation
SHOW [[USE] FILE | DATASET] CACHE	Get cache parameters for accessing HDF files or datasets
SHOW FLUSH	Get status of the automatic flushing
SHOW DEBUG	Get status of the debug mechanism
USE DIRECTORY	Use a directory for subsequent operations
USE FILE	Use (i.e. open) an HDF file for subsequent operations
USE GROUP	Use (i.e. open) an HDF group for subsequent operations
FLUSH [GLOBAL | LOCAL]	Flush the entire virtual HDF file (global) or only the HDF file (local) currently in

	use
CLOSE FILE	Close HDF file currently in use
CLOSE ALL FILE	Close all HDF files in use
CLOSE GROUP	Close HDF group currently in use
SET [FILE | DATASET] CACHE	Set cache for accessing HDF files or datasets
ENABLE FLUSH [GLOBAL | LOCAL]	Enable automatic flushing of the entire virtual HDF file or only the HDF file
ENABLE DEBUG	Enable debug mechanism
DISABLE FLUSH	Disable automatic flushing of the entire virtual HDF file or only the HDF file
DISABLE DEBUG	Disable debug mechanism
RUN	Run (i.e. execute) an external command

Table 6.2 – HDFql operations

6.1 DATATYPES

A datatype is a classification identifying one of various types of data such as integer, real or string, which determines the possible values for that type, the operations that can be done on values of that type, the meaning of the data, and the way values of that type can be stored. In other words, a datatype is a classification of data that tells HDFql how the user intends to use it. The following table summarizes all existing HDFql datatypes and how these map with the HDF5 datatypes[1].

HDFql	HDF5	Range of Values
TINYINT	H5T_NATIVE_CHAR	-128 to 127 (1 byte)
UNSIGNED TINYINT	H5T_NATIVE_UCHAR	0 to 255 (1 byte)

[1] For a detailed explanation of HDF5 datatypes please refer to https://support.hdfgroup.org/HDF5/doc1.8/UG/HDF5_Users_Guide-Responsive HTML5/index.html#t=HDF5_Users_Guide/Datatypes/HDF5_Datatypes.htm.

SMALLINT	H5T_NATIVE_SHORT	-32,768 to 32,767 (2 bytes)
UNSIGNED SMALLINT	H5T_NATIVE_USHORT	0 to 65,535 (2 bytes)
INT	H5T_NATIVE_INT	-2,147,483,648 to 2,147,483,647 (4 bytes)
UNSIGNED INT	H5T_NATIVE_UINT	0 to 4,294,967,295 (4 bytes)
BIGINT	H5T_NATIVE_LLONG	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (8 bytes)
UNSIGNED BIGINT	H5T_NATIVE_ULLONG	0 to 18,446,744,073,709,551,615 (8 bytes)
FLOAT	H5T_NATIVE_FLOAT	-3.4E + 38 to 3.4E + 38 (4 bytes)
DOUBLE	H5T_NATIVE_DOUBLE	-1.79E + 308 to 1.79E + 308 (8 bytes)
CHAR	H5T_C_S1	0 to 255 (*size* * 1 byte)
VARTINYINT	H5T_NATIVE_CHAR	-128 to 127 (*size* * 1 byte)
UNSIGNED VARTINYINT	H5T_NATIVE_UCHAR	0 to 255 (*size* * 1 byte)
VARSMALLINT	H5T_NATIVE_SHORT	-32,768 to 32,767 (*size* * 2 bytes)
UNSIGNED VARSMALLINT	H5T_NATIVE_USHORT	0 to 65,535 (*size* * 2 bytes)
VARINT	H5T_NATIVE_INT	-2,147,483,648 to 2,147,483,647 (*size* * 4 bytes)
UNSIGNED VARINT	H5T_NATIVE_UINT	0 to 4,294,967,295 (*size* * 4 bytes)
VARBIGINT	H5T_NATIVE_LLONG	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (*size* * 8 bytes)
UNSIGNED VARBIGINT	H5T_NATIVE_ULLONG	0 to 18,446,744,073,709,551,615 (*size* * 8 bytes)

VARFLOAT	H5T_NATIVE_FLOAT	-3.4E + 38 to 3.4E + 38 (*size* * 4 bytes)
VARDOUBLE	H5T_NATIVE_DOUBLE	-1.79E + 308 to 1.79E + 308 (*size* * 8 bytes)
VARCHAR	H5T_C_S1	0 to 255 (*size* * 1 byte)
OPAQUE	H5T_OPAQUE	0 to 255 (*size* * 1 byte)

Table 6.3 – HDFql datatypes and their corresponding definitions in HDF5

6.1.1 TINYINT

The TINYINT HDFql datatype corresponds to the H5T_NATIVE_CHAR HDF5 datatype. It may store a value between -128 and 127, and occupies 1 byte in memory. Depending on the programming language supported by HDFql, the TINYINT datatype is represented by:

- In C, the “char” datatype.
- In C++, the “char” datatype.
- In Java, the “byte” datatype or its corresponding wrapper class “Byte”.
- In Python, the “int8” NumPy datatype.
- In C#, the “SByte” datatype or its alias “sbyte”.
- In Fortran, the “INTEGER(KIND = 1)” datatype.

6.1.2 UNSIGNED TINYINT

The UNSIGNED TINYINT HDFql datatype corresponds to the H5T_NATIVE_UCHAR HDF5 datatype. It may store a value between 0 and 255, and occupies 1 byte in memory. Depending on the programming language supported by HDFql, the UNSIGNED TINYINT datatype is represented by:

- In C, the “unsigned char” datatype.
- In C++, the “unsigned char” datatype.
- In Java[2], the “byte” datatype or its corresponding wrapper class “Byte”.
- In Python, the “uint8” NumPy datatype.
- In C#, the “Byte” datatype or its alias “byte”.
- In Fortran[3], the “INTEGER(KIND = 1)” datatype.

6.1.3 SMALLINT

The SMALLINT HDFql datatype corresponds to the H5T_NATIVE_SHORT HDF5 datatype. It may store a value between -32,768 and 32,767, and occupies 2 bytes in memory. Depending on the programming language supported by HDFql, the SMALLINT datatype is represented by:

- In C, the “short” datatype.
- In C++, the “short” datatype.
- In Java, the “short” datatype or its corresponding wrapper class “Short”.
- In Python, the “int16” NumPy datatype.
- In C#, the “Int16” datatype or its alias “short”.
- In Fortran, the “INTEGER(KIND = 2)” datatype.

[2] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[3] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

6.1.4 UNSIGNED SMALLINT

The UNSIGNED SMALLINT HDFql datatype corresponds to the H5T_NATIVE_USHORT HDF5 datatype. It may store a value between 0 and 65,535, and occupies 2 bytes in memory. Depending on the programming language supported by HDFql, the UNSIGNED SMALLINT datatype is represented by:

- In C, the “unsigned short” datatype.
- In C++, the “unsigned short” datatype.
- In Java[4], the “short” datatype or its corresponding wrapper class “Short”.
- In Python, the “uint16” NumPy datatype.
- In C#, the “UInt16” datatype or its alias “ushort”.
- In Fortran[5], the “INTEGER(KIND = 2)” datatype.

6.1.5 INT

The INT HDFql datatype corresponds to the H5T_NATIVE_INT HDF5 datatype. It may store a value between -2,147,483,648 and 2,147,483,647, and occupies 4 bytes in memory. Depending on the programming language supported by HDFql, the INT datatype is represented by:

- In C, the “int” datatype.
- In C++, the “int” datatype.
- In Java, the “int” datatype or its corresponding wrapper class “Integer”.
- In Python, the “int32” NumPy datatype.

[4] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[5] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

- In C#, the “Int32” datatype or its alias “int”.
- In Fortran, the “INTEGER(KIND = 4)” or “INTEGER” datatypes.

6.1.6 UNSIGNED INT

The UNSIGNED INT HDFql datatype corresponds to the H5T_NATIVE_UINT HDF5 datatype. It may store a value between 0 and 4,294,967,295, and occupies 4 bytes in memory. Depending on the programming language supported by HDFql, the UNSIGNED INT datatype is represented by:

- In C, the “unsigned int” datatype.
- In C++, the “unsigned int” datatype.
- In Java[6], the “int” datatype or its corresponding wrapper class “Integer”.
- In Python, the “uint32” NumPy datatype.
- In C#, the “UInt32” datatype or its alias “uint”.
- In Fortran[7], the “INTEGER(KIND = 4)” or “INTEGER” datatypes.

6.1.7 BIGINT

The BIGINT HDFql datatype corresponds to the H5T_NATIVE_LLONG HDF5 datatype. It may store a value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807, and occupies 8 bytes in memory. Depending on the programming language supported by HDFql, the BIGINT datatype is represented by:

- In C, the “long long” datatype.

[6] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[7] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

- In C++, the “long long” datatype.
- In Java, the “long” datatype or its corresponding wrapper class “Long”.
- In Python, the “int64” NumPy datatype.
- In C#, the “Int64” datatype or its alias “long”.
- In Fortran, the “INTEGER(KIND = 8)” datatype.

6.1.8 UNSIGNED BIGINT

The UNSIGNED BIGINT HDFql datatype corresponds to the H5T_NATIVE_ULLONG HDF5 datatype. It may store a value between 0 and 18,446,744,073,709,551,615, and occupies 8 bytes in memory. Depending on the programming language supported by HDFql, the UNSIGNED BIGINT datatype is represented by:

- In C, the “unsigned long long” datatype.
- In C++, the “unsigned long long” datatype.
- In Java[8], the “long” datatype or its corresponding wrapper class “Long”.
- In Python, the “uint64” NumPy datatype.
- In C#, the “UInt64” datatype or its alias “ulong”.
- In Fortran[9], the “INTEGER(KIND = 8)” datatype.

[8] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[9] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

6.1.9 FLOAT

The FLOAT HDFql datatype corresponds to the H5T_NATIVE_FLOAT HDF5 datatype. It may store a value between -3.4E + 38 and 3.4E + 38, and occupies 4 bytes in memory. Depending on the programming language supported by HDFql, the FLOAT datatype is represented by:

- In C, the “float” datatype.
- In C++, the “float” datatype.
- In Java, the “float” datatype or its corresponding wrapper class “Float”.
- In Python, the “float32” NumPy datatype.
- In C#, the “Single” datatype or its alias “float”.
- In Fortran, the “REAL(KIND = 4)” or “REAL” datatypes.

6.1.10 DOUBLE

The DOUBLE HDFql datatype corresponds to the H5T_NATIVE_DOUBLE HDF5 datatype. It may store a value between -1.79E + 308 and 1.79E + 308, and occupies 8 bytes in memory. Depending on the programming language supported by HDFql, the DOUBLE datatype is represented by:

- In C, the “double” datatype.
- In C++, the “double” datatype.
- In Java, the “double” datatype or its corresponding wrapper class “Double”.
- In Python, the “float64” NumPy datatype.
- In C#, the “Double” datatype or its alias “double”.
- In Fortran, the “REAL(KIND = 8)” or “DOUBLE PRECISION” datatypes.

6.1.11 CHAR

The CHAR HDFql datatype corresponds to the H5T_C_S1 HDF5 datatype. It may store a value between 0 and 255, and occupies *size* * 1 byte in memory (*size* being the length of the string). The CHAR datatype is useful for storing fixed-length strings. Depending on the programming language supported by HDFql, the CHAR datatype is represented by:

- In C, the “char [*size*]” datatype.
- In C++, the “char [*size*]” datatype.
- In Java, the “String” object.
- In Python, the “S*size*” NumPy datatype.
- In C#, the “String” datatype or its alias “string”.
- In Fortran, the “CHARACTER(LEN = *size*)” datatype.

6.1.12 VARTINYINT

The VARTINYINT HDFql datatype corresponds to the H5T_NATIVE_CHAR HDF5 datatype. It may store a value between -128 and 127, and occupies *size* * 1 byte in memory (*size* being the number of elements composing the VARTINYINT datatype). Depending on the programming language supported by HDFql, the VARTINYINT datatype is represented by:

- In C, the “char” datatype.
- In C++, the “char” datatype.
- In Java, the “byte” datatype or its corresponding wrapper class “Byte”.
- In Python, the “int8” NumPy datatype.
- In C#, the “SByte” datatype or its alias “sbyte”.
- In Fortran, the “INTEGER(KIND = 1)” datatype.

6.1.13 UNSIGNED VARTINYINT

The UNSIGNED VARTINYINT HDFql datatype corresponds to the H5T_NATIVE_UCHAR HDF5 datatype. It may store a value between 0 and 255, and occupies *size* * 1 byte in memory (*size* being the number of elements composing the VARTINYINT datatype). Depending on the programming language supported by HDFql, the UNSIGNED VARTINYINT datatype is represented by:

- In C, the “unsigned char” datatype.
- In C++, the “unsigned char” datatype.
- In Java[10], the “byte” datatype or its corresponding wrapper class “Byte”.
- In Python, the “uint8” NumPy datatype.
- In C#, the “Byte” datatype or its alias “byte”.
- In Fortran[11], the “INTEGER(KIND = 1)” datatype.

6.1.14 VARSMALLINT

The VARSMALLINT HDFql datatype corresponds to the H5T_NATIVE_SHORT HDF5 datatype. It may store a value between -32,768 and 32,767, and occupies *size* * 2 bytes in memory (*size* being the number of elements composing the VARSMALLINT datatype). Depending on the programming language supported by HDFql, the VARSMALLINT datatype is represented by:

- In C, the “short” datatype.
- In C++, the “short” datatype.

[10] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[11] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

- In Java, the “short” datatype or its corresponding wrapper class “Short”.
- In Python, the “int16” NumPy datatype.
- In C#, the “Int16” datatype or its alias “short”.
- In Fortran, the “INTEGER(KIND = 2)” datatype.

6.1.15 UNSIGNED VARSMALLINT

The UNSIGNED VARSMALLINT HDFql datatype corresponds to the H5T_NATIVE_USHORT HDF5 datatype. It may store a value between 0 and 65,535, and occupies *size* * 2 bytes in memory (*size* being the number of elements composing the VARSMALLINT datatype). Depending on the programming language supported by HDFql, the UNSIGNED VARSMALLINT datatype is represented by:

- In C, the “unsigned short” datatype.
- In C++, the “unsigned short” datatype.
- In Java[12], the “short” datatype or its corresponding wrapper class “Short”.
- In Python, the “uint16” NumPy datatype.
- In C#, the “UInt16” datatype or its alias “ushort”.
- In Fortran[13], the “INTEGER(KIND = 2)” datatype.

[12] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[13] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

6.1.16 VARINT

The VARINT HDFql datatype corresponds to the H5T_NATIVE_INT HDF5 datatype. It may store a value between -2,147,483,648 and 2,147,483,647, and occupies *size* * 4 bytes in memory (*size* being the number of elements composing the VARINT datatype). Depending on the programming language supported by HDFql, the VARINT datatype is represented by:

- In C, the “int” datatype.
- In C++, the “int” datatype.
- In Java, the “int” datatype or its corresponding wrapper class “Integer”.
- In Python, the “int32” NumPy datatype.
- In C#, the “Int32” datatype or its alias “int”.
- In Fortran, the “INTEGER(KIND = 4)” datatype.

6.1.17 UNSIGNED VARINT

The UNSIGNED VARINT HDFql datatype corresponds to the H5T_NATIVE_UINT HDF5 datatype. It may store a value between 0 and 4,294,967,295, and occupies *size* * 4 bytes in memory (*size* being the number of elements composing the UNSIGNED VARINT datatype). Depending on the programming language supported by HDFql, the UNSIGNED VARINT datatype is represented by:

- In C, the “unsigned int” datatype.
- In C++, the “unsigned int” datatype.
- In Java[14], the “int” datatype or its corresponding wrapper class “Integer”.
- In Python, the “uint32” NumPy datatype.

[14] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

- In C#, the “UInt32” datatype or its alias “uint”.
- In Fortran[15], the “INTEGER(KIND = 4)” datatype.

6.1.18 VARBIGINT

The VARBIGINT HDFql datatype corresponds to the H5T_NATIVE_LLONG HDF5 datatype. It may store a value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807, and occupies *size* * 8 bytes in memory (*size* being the number of elements composing the VARBIGINT datatype). Depending on the programming language supported by HDFql, the VARBIGINT datatype is represented by:

- In C, the “long long” datatype.
- In C++, the “long long” datatype.
- In Java, the “long” datatype or its corresponding wrapper class “Long”.
- In Python, the “int64” NumPy datatype.
- In C#, the “Int64” datatype or its alias “long”.
- In Fortran, the “INTEGER(KIND = 8)” datatype.

6.1.19 UNSIGNED VARBIGINT

The UNSIGNED VARBIGINT HDFql datatype corresponds to the H5T_NATIVE_ULLONG HDF5 datatype. It may store a value between 0 and 18,446,744,073,709,551,615, and occupies *size* * 8 bytes in memory (*size* being the number of elements composing the UNSIGNED VARBIGINT datatype). Depending on the programming language supported by HDFql, the UNSIGNED VARBIGINT datatype is represented by:

- In C, the “unsigned long long” datatype.

[15] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

- In C++, the “unsigned long long” datatype.
- In Java[16], the “long” datatype or its corresponding wrapper class “Long”.
- In Python, the “uint64” NumPy datatype.
- In C#, the “UInt64” datatype or its alias “ulong”.
- In Fortran[17], the “INTEGER(KIND = 8)” datatype.

6.1.20 VARFLOAT

The VARFLOAT HDFql datatype corresponds to the H5T_NATIVE_FLOAT HDF5 datatype. It may store a value between -3.4E + 38 and 3.4E + 38, and occupies *size* * 4 bytes in memory (*size* being the number of elements composing the VARFLOAT datatype). Depending on the programming language supported by HDFql, the VARFLOAT datatype is represented by:

- In C, the “float” datatype.
- In C++, the “float” datatype.
- In Java, the “float” datatype or its corresponding wrapper class “Float”.
- In Python, the “float32” NumPy datatype.
- In C#, the “Single” datatype or its alias “float”.
- In Fortran, the “REAL(KIND = 4)” datatype.

[16] By design, Java does not support unsigned datatypes. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Java.

[17] Although there has been some effort to specify unsigned datatypes in Fortran, nothing concrete is available. Therefore, the programmer is responsible for making the conversion from a signed number to its equivalent unsigned in Fortran.

6.1.21 VARDOUBLE

The VARDOUBLE HDFql datatype corresponds to the H5T_NATIVE_DOUBLE HDF5 datatype. It may store a value between -1.79E + 308 and 1.79E + 308, and occupies *size* * 8 bytes in memory (*size* being the number of elements composing the VARDOUBLE datatype). Depending on the programming language supported by HDFql, the VARDOUBLE datatype is represented by:

- In C, the “double” datatype.
- In C++, the “double” datatype.
- In Java, the “double” datatype or its corresponding wrapper class “Double”.
- In Python, the “float64” NumPy datatype.
- In C#, the “Double” datatype or its alias “double”.
- In Fortran, the “REAL(KIND = 8)” or “DOUBLE PRECISION” datatypes.

6.1.22 VARCHAR

The VARCHAR HDFql datatype corresponds to the H5T_C_S1 HDF5 datatype. It may store a value between 0 and 255, and occupies *size* * 1 byte in memory (*size* being the length of the string). The VARCHAR datatype is useful for storing variable-length strings. Depending on the programming language supported by HDFql, the VARCHAR datatype is represented by:

- In C, the “char [*size*]” datatype.
- In C++, the “char [*size*]” datatype.
- In Java, the “String” object.
- In Python, the “S*size*” NumPy datatype.
- In C#, the “String” datatype or its alias “string”.
- In Fortran, the “CHARACTER(LEN = *size*)” datatype.

6.1.23 OPAQUE

The OPAQUE HDFql datatype corresponds to the H5T_C_S1 HDF5 datatype. It may store a value between 0 and 255, and occupies *size* * 1 byte in memory (*size* being the number of elements composing the OPAQUE datatype). Depending on the programming language supported by HDFql, the VARCHAR datatype is represented by:

- In C, the “char [*size*]” datatype.
- In C++, the “char [*size*]” datatype.
- In Java, the “byte [*size*]” datatype or its corresponding wrapper class “Byte [*size*]”.
- In Python, the “S*size*” NumPy datatype.
- In C#, the “SByte [*size*]” datatype or its alias “sbyte [*size*]”.
- In Fortran, the “CHARACTER(LEN = *size*)” datatype.

6.2 POST-PROCESSING

Post-processing options enable processing (i.e. transformation) results of a query according to the programmer’s needs such as ordering or redirecting. These options are optional and may be used to create a (linear) pipeline to further process result sets returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations. In case a pipeline is composed of two or more options, the order in which they are used is important and should always follow this sequence: ORDER, TOP, BOTTOM, STEP and INTO (e.g. usage of TOP followed by INTO is permitted, while the inverse – i.e. usage of INTO followed by TOP – is not permitted). The next subsections describe the post-processing options provided by HDFql.

Post-processing Option	Description
ORDER	Order (i.e. sort) a result set in an ascending, descending or reverse way
TOP	Truncate a result set after a certain given position in a topmost way

BOTTOM	Truncate a result set after a certain given position in a bottommost way
STEP	Step (i.e. jump) the result set at every given position
INTO	Redirect (i.e. write) result sets returned into a file or memory

Table 6.4 – HDFql post-processing options

6.2.1 ORDER

Syntax

ORDER {**ASC** | **DESC** | {**REV**, …, **REV**} | **CREATION**}

Description

Order (i.e. sort) a result set in an ascending, descending or reverse way using either the keyword ASC, DESC or REV respectively. When in an ascending or descending order, HDFql automatically uses all available CPU cores to speed-up the task completion. Additionally, when performing this type of ordering on a result set coming from a dataset or attribute with two or more dimensions, the ordering is done only on the last dimension. When reverse ordering a result set coming from a dataset or attribute with two or more dimensions, multiple REV keywords may be specified to enable the ordering of specific dimensions (e.g. if "ORDER REV, , REV" is specified, reverse ordering is done both on the first and third dimensions while the second remains unchanged). Finally, a special type of ordering can be performed on a SHOW [GROUP | DATASET | ATTRIBUTE] operation using the keyword CREATION allowing HDF objects (i.e. groups, datasets and attributes) to be returned according to their time of creation – in contrast to the default behaviour which returns objects in an ascending order.

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use (which stores the result set) is ordered in function of the keyword used, namely ASC, DESC, REV or CREATION. If the INTO post-processing option is

specified (besides the ORDER post-processing option), the cursor in use remains unchanged. Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// create a dataset named "my_dataset0" of type float of three dimensions (size 5x8x4)
hdfql_execute("CREATE DATASET my_dataset0 AS FLOAT(5, 8, 4)");

// populate cursor in use with the dimensions of dataset "my_dataset0" (should be 5, 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset0");

// populate cursor in use with the dimensions of dataset "my_dataset0" in ascending order (should be 4, 5, 8)
hdfql_execute("SHOW DIMENSION my_dataset0 ORDER ASC");

// populate cursor in use with the dimensions of dataset "my_dataset0" in descending order (should be 8, 5, 4)
hdfql_execute("SHOW DIMENSION my_dataset0 ORDER DESC");

// populate cursor in use with the dimensions of dataset "my_dataset0" in reversed order (should be 4, 8, 5)
hdfql_execute("SHOW DIMENSION my_dataset0 ORDER REV");
```

```
// create a dataset named "my_dataset1" of type double of two dimensions (size 3x2)
hdfql_execute("CREATE DATASET my_dataset1 AS DOUBLE(3, 2)");

// insert (i.e. write) values into dataset "my_dataset1"
hdfql_execute("INSERT INTO my_dataset1 VALUES((3.2, 1.3), (0, 0.2), (9.1, 6.5))");

// populate cursor in use with data from dataset "my_dataset1" (should be 3.2, 1.3, 0, 0.2, 9.1, 6.5)
hdfql_execute("SELECT FROM my_dataset1");

// populate cursor in use with data from dataset "my_dataset1" in ascending order (should be 1.3, 3.2, 0, 0.2, 6.5, 9.1)
hdfql_execute("SELECT FROM my_dataset1 ORDER ASC");

// populate cursor in use with data from dataset "my_dataset1" in descending order
```

```
(should be 3.2, 1.3, 0.2, 0, 9.1, 6.5)
hdfql_execute("SELECT FROM my_dataset1 ORDER DESC");

// populate cursor in use with data from dataset "my_dataset1" in reversed order on the
first dimension only (should be 9.1, 6.5, 0, 0.2, 3.2, 1.3)
hdfql_execute("SELECT FROM my_dataset1 ORDER REV");

// populate cursor in use with data from dataset "my_dataset1" in reversed order on the
second dimension only (should be 1.3, 3.2, 0.2, 0, 6.5, 9.1)
hdfql_execute("SELECT FROM my_dataset1 ORDER , REV");

// populate cursor in use with data from dataset "my_dataset1" in reversed order on both
the first and second dimensions (should be 6.5, 9.1, 0.2, 0, 1.3, 3.2)
hdfql_execute("SELECT FROM my_dataset1 ORDER REV, REV");
```

6.2.2 TOP

Syntax

TOP *top_value*

Description

Truncate a result set after position *top_value* in a topmost way. In other words, all elements after position *top_value* are discarded from the result set. If *top_value* is negative, the TOP option will behave as the BOTTOM option with a positive *top_value*. Of note, the TOP option is not available in a DATA QUERY LANGUAGE (DQL) operation as the hyperslab functionalities found in such operation make this option redundant.

Parameter(s)

top_value – to be defined.

Return

If the INTO post-processing option is not specified, the cursor in use (which stores the result set) is truncated in a topmost way in function of the position provided. If the INTO post-processing option is specified (besides the

TOP post-processing option), the cursor in use remains unchanged. Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// create a dataset named "my_dataset" of type float of three dimensions (size 5x8x4)
hdfql_execute("CREATE DATASET my_dataset AS FLOAT(5, 8, 4)");

// populate cursor in use with the dimensions of dataset "my_dataset" (should be 5, 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset");

// populate cursor in use with the topmost (i.e. first) dimension of dataset "my_dataset"
(should be 5)
hdfql_execute("SHOW DIMENSION my_dataset TOP 1");

// populate cursor in use with the two topmost dimensions of dataset "my_dataset" (should
be 5, 8)
hdfql_execute("SHOW DIMENSION my_dataset TOP 2");

// populate cursor in use with the two bottommost dimensions of dataset "my_dataset"
(should be 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset TOP -2");
```

6.2.3 BOTTOM

Syntax

BOTTOM *bottom_value*

Description

Truncate a result set after position *bottom_value* in a bottommost way. In other words, all elements before position *bottom_value* are discarded from the result set. If *bottom_value* is negative, the BOTTOM option will behave as the TOP option with a positive *bottom_value*. Of note, the BOTTOM option is not available in a DATA QUERY LANGUAGE (DQL) operation as the hyperslab functionalities found in such operation make this option redundant.

Parameter(s)

bottom_value – to be defined.

Return

If the INTO post-processing option is not specified, the cursor in use (which stores the result set) is truncated in a bottommost way in function of the position provided. If the INTO post-processing option is specified (besides the BOTTOM post-processing option), the cursor in use remains unchanged. Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// create a dataset named "my_dataset" of type float of three dimensions (size 5x8x4)
hdfql_execute("CREATE DATASET my_dataset AS FLOAT(5, 8, 4)");

// populate cursor in use with the dimensions of dataset "my_dataset" (should be 5, 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset");

// populate cursor in use with the bottommost (i.e. last) dimension of dataset
"my_dataset" (should be 4)
hdfql_execute("SHOW DIMENSION my_dataset BOTTOM 1");

// populate cursor in use with the two bottommost dimensions of dataset "my_dataset"
(should be 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset BOTTOM 2");

// populate cursor in use with the two topmost dimensions of dataset "my_dataset" (should
be 5, 8)
hdfql_execute("SHOW DIMENSION my_dataset BOTTOM -2");
```

6.2.4 STEP

Syntax

STEP *step_value*

Description

Step (i.e. jump) the result set at every *step_value* position. In other words, all elements between steps are discarded from the result set. Of note, the STEP option is not available in a DATA QUERY LANGUAGE (DQL) operation as the hyperslab functionalities found in such operation make this option redundant.

Parameter(s)

step_value – to be defined.

Return

If the INTO post-processing option is not specified, the cursor in use (which stores the result set) is stepped in function of the position provided. If the INTO post-processing option is specified (besides the STEP post-processing option), the cursor in use remains unchanged. Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// create a dataset named "my_dataset" of type float of three dimensions (size 5x8x4)
hdfql_execute("CREATE DATASET my_dataset AS FLOAT(5, 8, 4)");

// populate cursor in use with the dimensions of dataset "my_dataset" (should be 5, 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset");

// populate cursor in use with the dimensions of dataset "my_dataset" (should be 5, 8, 4)
hdfql_execute("SHOW DIMENSION my_dataset STEP 1");

// populate cursor in use with every second dimension of dataset "my_dataset" (should be 5, 4)
hdfql_execute("SHOW DIMENSION my_dataset STEP 2");

// populate cursor in use with every third dimension of dataset "my_dataset" (should be 5)
hdfql_execute("SHOW DIMENSION my_dataset STEP 3");
```

6.2.5 INTO

Syntax

INTO {[**TRUNCATE**] [**DOS** | **UNIX**] [**TEXT**] **FILE** *file_name* [**SEPARATOR** *separator_value*] [**SPLIT** *split_value*]} | {[**TRUNCATE**] **BINARY FILE** *file_name*} | {**MEMORY** *variable_number* [**SIZE** *variable_size*]}]

Description

Redirect (i.e. write) result sets returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations into a file or memory (by default – i.e. when the INTO post-processing option is not specified – a result set is stored in the cursor in use at the moment of executing the operation). More specifically, the redirection can be done into:

- A text file using optional parameters such as which terminator to use – DOS (CR+LF) or UNIX (LF) – for the end of line (EOL), which separator to use between elements (of the result set), or the number of elements to write per line before starting writing remaining elements in a new line.
- A binary file.
- A variable that was previously registered through the function hdfql_variable_register.

When redirecting a result set into a file that already exists, the result set is appended to it. To overwrite an existing file, specify the keyword TRUNCATE (ALL DATA STORED IN THE FILE WILL BE PERMANENTLY LOST).

Parameter(s)

file_name – to be defined.

separator_value – to be defined.

split_value – to be defined.

variable_number – number of the variable that will store the result set (i.e. data) returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations. The number is returned by the function hdfql_variable_register upon registering the variable or, subsequently, returned by the function hdfql_variable_get_number.

variable_size – to be defined.

Return

The cursor in use remains unchanged when using the INTO post-processing option. Please refer to the chapter CURSOR for additional information.

Example(s)

```
// create a dataset named "my_dataset0" of type short of one dimension (size 3)
hdfql_execute("CREATE DATASET my_dataset0 AS SMALLINT(3)");

// insert (i.e. write) values into dataset "my_dataset0"
hdfql_execute("INSERT INTO my_dataset0 VALUES(65, 66, 67)");

// populate cursor in use with data from dataset "my_dataset0" (should be 65, 66, 67)
hdfql_execute("SELECT FROM my_dataset0");

// select (i.e. read) data from dataset "my_dataset0" and write it into a text file named
"my_file.txt" using default separator "," (should be "65,66,67" in one single line)
hdfql_execute("SELECT FROM my_dataset0 INTO FILE my_file.txt");

// select (i.e. read) data from dataset "my_dataset0" and write it into a text file named
"my_file.txt" using separator "**" (should be "65**66**67" in one single line)
hdfql_execute("SELECT FROM my_dataset0 INTO TEXT FILE my_file.txt SEPARATOR **");

// select (i.e. read) data from dataset "my_dataset0" and write it into a text file named
"my_file.txt" splitting every two values in a new line using a UNIX-based EOL terminator
(should be "65,65" in the first line and "67" in the second line)
hdfql_execute("SELECT FROM my_dataset0 INTO UNIX TEXT FILE my_file.txt SPLIT 2");

// select (i.e. read) data from dataset "my_dataset0" and write it into a binary file
(truncate it if it already exists) named "my_file.bin" (should be "ABC")
hdfql_execute("SELECT FROM my_dataset0 INTO TRUNCATE BINARY FILE my_file.bin");
```

```
// declare variables
char script[1024];
double data[3][2];
int x;
```

```
int y;

// create a dataset named "my_dataset1" of type double of two dimensions (size 3x2)
hdfql_execute("CREATE DATASET my_dataset1 AS DOUBLE(3, 2)");

// insert (i.e. write) values into dataset "my_dataset1"
hdfql_execute("INSERT INTO my_dataset1 VALUES((3.2, 1.3), (0, 0.2), (9.1, 6.5))");

// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to select (i.e. read) dataset "my_dataset1" and populate variable
"data" with it
sprintf(script, "SELECT FROM my_dataset1 INTO MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// unregister variable "data" as it is no longer used/needed (by HDFql)
hdfql_variable_unregister(&data);

// display content of variable "data" (should be 3.2, 1.3, 0, 0.2, 9.1, 6.5)
for(x = 0; x < 3; x++)
{
    for(y = 0; y < 2; y++)
    {
        printf("%d\n", data[x][y]);
    }
}
```

6.3 DATA DEFINITION LANGUAGE (DDL)

Data Definition Language (DDL) is, generally speaking, syntax for defining and modifying structures that store data. In HDFql, the DDL assembles the operations that enable the creation, alteration, renaming, copying and deletion of HDF files, groups, datasets, attributes and links. These operations begin either with the keyword CREATE, ALTER, RENAME, COPY or DROP.

6.3.1 CREATE DIRECTORY

Syntax

CREATE DIRECTORY *directory_name1, ..., directory_nameX*

Description

Create a directory named *directory_name*. Multiple directories can be created at once by separating these with a comma (,). If *directory_name* already exists, it will not be overwritten, no subsequent directories are created, and an error is raised. In case *directory_name* has intermediate directories that do not exist, besides *directory_name* being created, all these intermediate directories will be created on the fly (e.g. when creating the directory "my_directory/my_subdirectory/my_subsubdirectory", besides "my_subsubdirectory" being created, "my_directory" and "my_subdirectory" will be created in case they do not exist).

Parameter(s)

directory_name – name of the directory to create. Multiple directories are separated with a comma (,).

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create a directory named "my_directory0" (the directory will not be overwritten if it already exists)
CREATE DIRECTORY my_directory0

# create a directory named "my_directory1" in a root directory named "data" (neither directory will be overwritten if they already exist; directory "data" will be created on the fly if it does not exist)
CREATE DIRECTORY /data/my_directory1

# create two directories named "my_directory2" and "my_directory3" (neither directory will be overwritten if they already exist)
CREATE DIRECTORY my_directory2, my_directory3
```

6.3.2 CREATE FILE

Syntax

CREATE [**TRUNCATE**] **FILE** *file_name1, ..., file_nameX*

Description

Create an HDF file named *file_name*. Multiple files can be created at once by separating these with a comma (,). If *file_name* already exists, it will not be overwritten, no subsequent files are created, and an error is raised. To overwrite an existing file, specify the keyword TRUNCATE (ALL DATA STORED IN THE FILE WILL BE PERMANENTLY LOST).

Parameter(s)

file_name – name of the HDF file to create. Multiple files are separated with a comma (,).

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create an HDF file named "my_file0.h5" (the file will not be overwritten if it already exists)
CREATE FILE my_file0.h5

# create an HDF file named "my_file1.h5" in a root directory named "data" (the file will not be overwritten if it already exists)
CREATE FILE /data/my_file1.h5

# create two HDF files named "my_file2.h5" and "my_file3.h5" (both files will be overwritten if they already exist)
CREATE TRUNCATE FILE my_file2.h5, my_file3.h5
```

6.3.3 CREATE GROUP

Syntax

CREATE [**TRUNCATE**] **GROUP** *group_name1, ..., group_nameX*

[**ORDER** {**TRACKED** | **INDEXED**}]

[**STORAGE COMPACT** *object_max_compact* **DENSE** *object_min_dense*]

[**ATTRIBUTE** [**ORDER** {**TRACKED** | **INDEXED**}] [**STORAGE COMPACT** *attribute_max_compact* **DENSE** *attribute_min_dense*]]

Description

Create an HDF group named *group_name*. Multiple groups can be created at once by separating these with a comma (,). If *group_name* already exists, it will not be overwritten, no subsequent groups are created, and an error is raised. To overwrite an existing group, specify the keyword TRUNCATE (ALL DATA STORED IN THE GROUP WILL BE PERMANENTLY LOST). In case *group_name* has intermediate groups that do not exist, besides *group_name* being created, all these intermediate groups will be created on the fly (e.g. when creating the group “my_group/my_subgroup/my_subsubgroup”, besides “my_subsubgroup” being created, “my_group” and “my_subgroup” will be created in case they do not exist).

Parameter(s)

group_name – name of the HDF group to create. Multiple groups are separated with a comma (,).

object_max_compact – to be defined.

object_min_dense – to be defined.

attribute_max_compact – to be defined.

attribute_min_dense – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create an HDF group named "my_group0" (the group will not be overwritten if it already exists)
CREATE GROUP my_group0

# create an HDF group named "my_group1" in a root group named "data" (neither group will be overwritten if they already exist; group "data" will be created on the fly if it does not exist)
CREATE GROUP /data/my_group1

# create two HDF groups named "my_group2" and "my_group3" (both groups will be overwritten if they already exist)
CREATE TRUNCATE GROUP my_group2, my_group3
```

```
# create an HDF group named "my_group4" that tracks the objects' (i.e. groups and datasets) creation order within the group and using compact storage
CREATE GROUP my_group4 ORDER TRACKED STORAGE COMPACT 10 DENSE 7

# create an HDF group named "my_group5" that indexes the attributes' creation order
CREATE GROUP my_group5 ATTRIBUTE ORDER INDEXED
```

6.3.4 CREATE DATASET

Syntax

CREATE [**TRUNCATE**] [**CONTIGUOUS** | **COMPACT** | {**CHUNKED** [**(***chunked_dim1, ..., chunked_dimX***)**]}] **DATASET** *dataset_name1, ..., dataset_nameX* **AS** [**NATIVE** | **LITTLE ENDIAN** | **BIG ENDIAN** | **ASCII** | **UTF8**] *datatype* [**(UNLIMITED** | {*dataset_dim1* [**TO** {*dataset_max_dim1* | **UNLIMITED**}]}*, ...,* **UNLIMITED** | {*dataset_dimX* [**TO** {*dataset_max_dimX* | **UNLIMITED**}]}**)**]

[**TAG** *tag_value*]

[**DEFAULT** *default_value*]

[**ATTRIBUTE** [**ORDER** {**TRACKED** | **INDEXED**}] [**STORAGE COMPACT** *attribute_max_compact* **DENSE** *attribute_min_dense*]]

[**ENABLE** [**SHUFFLE**] [**SCALEOFFSET** [*scaleoffset_value*]] [**NBIT PRECISION** *precision_value* **OFFSET** *offset_value*] [**ZLIB** [**LEVEL** *level_value*]] [**FLETCHER32**]]

Description

Create an HDF dataset named *dataset_name*. Multiple datasets can be created at once by separating these with a comma (,). If *dataset_name* already exists, it will not be overwritten, no subsequent datasets are created, and an error is raised. To overwrite an existing dataset, specify the keyword TRUNCATE (ALL DATA STORED IN THE DATASET WILL BE PERMANENTLY LOST).

Parameter(s)

chunked_dim – to be defined.

dataset_name – name of the HDF dataset to create. Multiple datasets are separated with a comma (,).

datatype – to be defined.

dataset_dim – to be defined.

dataset_max_dim – to be defined.

tag_value – to be defined.

default_value – to be defined.

attribute_max_compact – to be defined.

attribute_min_dense – to be defined.

scaleoffset_value – to be defined.

precision_value – to be defined.

offset_value – to be defined.

level_value – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type int (the dataset will not be overwritten if it already exists)
CREATE DATASET my_dataset0 AS INT

# create an HDF dataset named "my_dataset1" of type char in a root group named "data" (the dataset will not be overwritten if it already exists)
CREATE DATASET /data/my_dataset1 AS CHAR

# create two HDF datasets named "my_dataset2" and "my_dataset3" of type short (both datasets will be overwritten if they already exist)
CREATE TRUNCATE DATASET my_dataset2, my_dataset3 AS SMALLINT
```

```
# create an HDF dataset named "my_dataset4" of type unsigned long long using a big endian representation
CREATE DATASET my_dataset4 AS BIG ENDIAN UNSIGNED BIGINT

# create an HDF dataset named "my_dataset5" of type int using a little endian representation with a default value 80178
CREATE DATASET my_dataset5 AS LITTLE ENDIAN INT DEFAULT 80178

# create an HDF dataset named "my_dataset6" of type char using an ASCII representation
CREATE DATASET my_dataset6 AS ASCII CHAR
```

```
# create an HDF dataset named "my_dataset7" of type float of one dimension (size 1024)
CREATE DATASET my_dataset7 AS FLOAT(1024)

# create a compact HDF dataset named "my_dataset8" of type double of three dimensions (size 2x5x10)
CREATE COMPACT DATASET my_dataset8 AS DOUBLE(2, 5, 10)
```

```
# create a chunked (20x100) HDF dataset named "my_dataset9" of type unsigned char of two dimensions (size 500x1000)
CREATE CHUNKED(20, 100) DATASET my_dataset9 AS UNSIGNED TINYINT(500, 1000)
```

```
# create an HDF dataset named "my_dataset10" of type int of two dimensions (size 20x400) using the N-bit data compression filter
CREATE DATASET my_dataset10 AS INT(20, 400) ENABLE NBIT PRECISION 16 OFFSET 4

# create an HDF dataset named "my_dataset11" of type float of one dimension (size 500000) using both the ZLIB data compression and Fletcher32 checksum error detection filters
CREATE DATASET my_dataset11 AS FLOAT(500000) ENABLE ZLIB LEVEL 5 FLETCHER32
```

```
# create an HDF dataset named "my_dataset12" of type variable-length float
CREATE DATASET my_dataset12 AS VARFLOAT

# create an HDF dataset named "my_dataset13" of type variable-length short of one dimension (size 5) with a default value 876
CREATE DATASET my_dataset13 AS VARSMALLINT(5) DEFAULT 876

# create an HDF dataset named "my_dataset14" of type variable-length char with a default value "Hierarchical Data Format"
CREATE DATASET my_dataset14 AS VARCHAR DEFAULT "Hierarchical Data Format"
```

```
# create an HDF dataset named "my_dataset15" of type opaque
CREATE DATASET my_dataset15 AS OPAQUE

# create an HDF dataset named "my_dataset16" of type opaque of one dimension (size 6) with the default ASCII values 72, 68, 70, 0, 113 and 108 (i.e. "HDF\0ql")
CREATE DATASET my_dataset16 AS OPAQUE(6) DEFAULT 72, 68, 70, 0, 113, 108

# create an HDF dataset named "my_dataset17" of type opaque of two dimensions (size 10x1024) with a tag value "Raw data"
CREATE DATASET my_dataset17 AS OPAQUE(10, 1024) TAG "Raw data"
```

```
# create an HDF dataset named "my_dataset18" of type float of one dimension (size 5 and extendible up to 10)
CREATE CHUNKED DATASET my_dataset18 AS FLOAT(5 TO 10)

# create an HDF dataset named "my_dataset19" of type variable-length int of one dimension (size 1 and extendible to an unlimited size)
CREATE CHUNKED DATASET my_dataset19 AS VARINT(UNLIMITED)

# create an HDF dataset named "my_dataset20" of type double of three dimensions (first dimension with size 3 and extendible up to 5; second dimension with size 7; third dimension with size 20 and extendible to an unlimited size)
CREATE CHUNKED DATASET my_dataset20 AS DOUBLE(3 TO 5, 7, 20 TO UNLIMITED)
```

6.3.5 CREATE ATTRIBUTE

Syntax

CREATE [**TRUNCATE**] **ATTRIBUTE** *attribute_name1, ..., attribute_nameX* **AS** [**NATIVE** | **LITTLE ENDIAN** | **BIG ENDIAN** | **ASCII** | **UTF8**] *datatype* [**(***attribute_dim1, ...,* attribute_*dimX***)**]

[**TAG** *tag_value*]

[**DEFAULT** *default_value*]

Description

Create an HDF attribute named *attribute_name*. Multiple attributes can be created at once by separating these with a comma (,). If *attribute_name* already exists, it will not be overwritten, no subsequent attributes are created, and an error is raised. To overwrite an existing attribute, specify the keyword TRUNCATE (ALL DATA STORED IN THE ATTRIBUTE WILL BE PERMANENTLY LOST).

Parameter(s)

attribute_name – name of the HDF attribute to create. Multiple attributes are separated with a comma (,).

datatype – to be defined.

attribute_dim – to be defined.

tag_value – to be defined.

default_value – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create an HDF attribute named "my_attribute0" of type int (the attribute will not be overwritten if it already exists)
CREATE ATTRIBUTE my_attribute0 AS INT

# create an HDF attribute named "my_attribute1" of type char in a root group named "data" (the attribute will not be overwritten if it already exists)
CREATE ATTRIBUTE /data/my_attribute1 AS CHAR

# create two HDF attributes named "my_attribute2" and "my_attribute3" of type short (both attributes will be overwritten if they already exist)
CREATE TRUNCATE ATTRIBUTE my_attribute2, my_attribute3 AS SMALLINT
```

```
# create an HDF attribute named "my_attribute4" of type unsigned long long using a big endian representation
CREATE ATTRIBUTE my_attribute4 AS BIG ENDIAN UNSIGNED BIGINT

# create an HDF attribute named "my_attribute5" of type int using a little endian representation with a default value 80178
CREATE ATTRIBUTE my_attribute5 AS LITTLE ENDIAN INT DEFAULT 80178

# create an HDF attribute named "my_attribute6" of type char using an UTF8 representation
CREATE ATTRIBUTE my_attribute6 AS UTF8 CHAR
```

```
# create an HDF attribute named "my_attribute7" of type float of one dimension (size 512)
CREATE ATTRIBUTE my_attribute7 AS FLOAT(512)
```

```
# create an HDF attribute named "my_attribute8" of type unsigned char of two dimensions (size 500x1000)
CREATE ATTRIBUTE my_attribute8 AS UNSIGNED TINYINT(500, 1000)
```

```
# create an HDF attribute named "my_attribute9" of type variable-length float
CREATE ATTRIBUTE my_attribute9 AS VARFLOAT

# create an HDF attribute named "my_attribute10" of type variable-length short of one dimension (size 5) with a default value 876
CREATE ATTRIBUTE my_attribute10 AS VARSMALLINT(5) DEFAULT 876

# create an HDF attribute named "my_attribute11" of type variable-length char with a default value "Hierarchical Data Format"
CREATE ATTRIBUTE my_attribute11 AS VARCHAR DEFAULT "Hierarchical Data Format"
```

```
# create an HDF attribute named "my_attribute12" of type opaque
CREATE ATTRIBUTE my_attribute12 AS OPAQUE

# create an HDF attribute named "my_attribute13" of type opaque of one dimension (size 6) with the default ASCII values 72, 68, 70, 0, 113 and 108 (i.e. "HDF\0ql")
CREATE ATTRIBUTE my_attribute13 AS OPAQUE(6) DEFAULT 72, 68, 70, 0, 113, 108

# create an HDF attribute named "my_attribute14" of type opaque of two dimensions (size 10x1024) with a tag value "Raw data"
CREATE ATTRIBUTE my_attribute14 AS OPAQUE(10, 1024) TAG "Raw data"
```

6.3.6 CREATE [SOFT | HARD] LINK

Syntax

CREATE [**TRUNCATE**] [**SOFT** | **HARD**] **LINK** *link_name1, ..., link_nameX* **TO** *object_name1, ..., object_nameX*

Description

Create an HDF soft or hard link named *link_name* to a group or dataset named *object_name*. Multiple links can be created at once by separating these with a comma (,). If *link_name* already exists, it will not be overwritten, no subsequent links are created, and an error is raised. To overwrite an existing link, specify the keyword TRUNCATE. If neither the keyword SOFT nor HARD is specified, a soft link is created by default. To create a hard link, the keyword HARD must be specified.

Parameter(s)

link_name – name of the HDF soft or hard link to create. Multiple links are separated with a comma (,).

object_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create an HDF group named "my_group0"
CREATE GROUP my_group0

# create an HDF dataset named "my_dataset0" of type variable-length unsigned int
CREATE DATASET my_dataset0 AS UNSIGNED VARINT

# create an HDF soft link named "my_link0" to group "my_group0" (the soft link will not be overwritten if it already exists)
CREATE LINK my_link0 TO my_group0

# create an HDF soft link named "my_link1" to dataset "my_dataset0" (the soft link will not be overwritten if it already exists)
CREATE SOFT LINK my_link1 TO my_dataset0

# create two HDF soft links named "my_link2" and "my_link3" to dataset "my_dataset0" and group "my_group0" respectively (both soft links will be overwritten if they already exist)
CREATE TRUNCATE SOFT LINK my_link2, my_link3 TO my_dataset0, my_group0
```

```
# create an HDF group named "my_group1"
CREATE GROUP my_group1

# create an HDF dataset named "my_dataset1" of type variable-length unsigned int
CREATE DATASET my_dataset1 AS UNSIGNED VARINT

# create an HDF hard link named "my_link4" to group "my_group1" (the hard link will not be overwritten if it already exists)
CREATE HARD LINK my_link4 TO my_group1

# create an HDF hard link named "my_link5" to dataset "my_dataset1" (the hard link will not be overwritten if it already exists)
CREATE HARD LINK my_link5 TO my_dataset1

# create two HDF hard links named "my_link6" and "my_link7" to dataset "my_dataset1" and group "my_group1" respectively (both hard links will be overwritten if they already exist)
CREATE TRUNCATE HARD LINK my_link6, my_link7 TO my_dataset1, my_group1
```

6.3.7 CREATE EXTERNAL LINK

Syntax

CREATE [**TRUNCATE**] **EXTERNAL LINK** *link_name1, ..., link_nameX* **TO** *file_name1 object_name1, ..., file_nameX object_nameX*

Description

Create an HDF external link named *link_name* to a group or dataset named *object_name* belonging to another HDF file named *file_name*. Multiple external links can be created at once by separating these with a comma (,). If *link_name* already exists, it will not be overwritten, no subsequent external links are created, and an error is raised. To overwrite an existing external link, specify the keyword TRUNCATE.

Parameter(s)

link_name – name of the HDF external link to create. Multiple external links are separated with a comma (,).

file_name – to be defined.

object_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# use (i.e. open) an HDF file named "my_file0.h5"
USE FILE my_file0.h5

# create an HDF group named "my_group"
CREATE GROUP my_group

# create an HDF dataset named "my_dataset" of type variable-length unsigned int
CREATE DATASET my_dataset AS UNSIGNED VARINT

# use (i.e. open) an HDF file named "my_file.h5"
USE FILE my_file1.h5

# create an HDF external link named "my_link0" to group "my_group" in file "my_file0.h5"
(the external link will not be overwritten if it already exists)
CREATE EXTERNAL LINK my_link0 TO my_file0.h5 my_group

# create an HDF external link named "my_link1" to dataset "my_dataset" in file
"my_file0.h5" (the external link will be overwritten if it already exists)
CREATE TRUNCATE EXTERNAL LINK my_link1 TO my_file0.h5 my_dataset

# create two HDF external links named "my_link2" and "my_link3" to dataset "my_dataset"
and group "my_group" in file "my_file0.h5" (neither external links will be overwritten if
they already exist)
CREATE EXTERNAL LINK my_link2, my_link3 TO my_file0.h5 my_dataset, my_file0.h5 my_group
```

6.3.8 ALTER DIMENSION

Syntax

ALTER DIMENSION *dataset_name1, ..., dataset_nameX* **TO (***dim1, ..., dimX***)**

Description

Alter (i.e. change) the dimensions of an existing dataset named *dataset_name*. Multiple datasets can have their dimensions altered at once by separating these with a comma (,). If *dataset_name* was not found or its dimensions could not be altered (due to unknown/unexpected reasons), no subsequent datasets are altered, and an error is raised. Depending on the prefix of the value specified (in *dim1, …, dimX*), one of the following behaviors applies:

- If its prefix is "+", the dimension will have its size increased by this value.
- If its prefix is "-", the dimension will have its size decreased by this value.
- In case its prefix is neither "+" nor "-", the dimension will carry the size of this value.

To preserve the value of a certain dimension (i.e. for its size not to be altered), it should be skipped with a comma (,).

Parameter(s)

dataset_name – name of the HDF dataset whose dimensions are to be altered (i.e. changed). Multiple datasets are separated with a comma (,).

dim – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset" of type double of three dimensions (first
```

```
dimension with size 2 and extendible up to 10; second dimension with size 7; third
dimension with size 20 and extendible to an unlimited size)
CREATE CHUNKED DATASET my_dataset AS DOUBLE(2 TO 10, 7, 20 TO UNLIMITED)

# show (i.e. get) current dimensions of dataset "my_dataset" (should be 2, 7, 20)
SHOW DIMENSION my_dataset

# alter (i.e. change) dimensions of dataset "my_dataset" to set its first dimension size
to 6, and increase the third dimension size by 10 (the second dimension size remains
intact)
ALTER DIMENSION my_dataset TO (6, , +10)

# show (i.e. get) current dimensions of dataset "my_dataset" (should be 6, 7, 30)
SHOW DIMENSION my_dataset

# alter (i.e. change) dimensions of dataset "my_dataset" to increase its first dimension
size by 2, to set the second dimension size to 3, and to decrease the third dimension
size by 5
ALTER DIMENSION my_dataset TO (+2, 3, -5)

# show (i.e. get) current dimensions of dataset "my_dataset" (should be 8, 3, 25)
SHOW DIMENSION my_dataset
```

6.3.9 RENAME DIRECTORY

Syntax

RENAME DIRECTORY *directory_name1, ..., directory_nameX* **AS** *new_directory_name1, ..., new_directory_nameX*

Description

Rename (or move) an existing directory named *directory_name* as *new_directory_name*. Multiple directories can be renamed (or moved) at once by separating these with a comma (,). If *new_directory_name* already exists, it will not be overwritten, no subsequent directories are renamed (or moved), and an error is raised.

Parameter(s)

directory_name – name of the directory to rename (or move). Multiple directories are separated with a comma (,).

new_directory_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# rename a directory named "my_directory0" as "my_directory1" (the directory
"my_directory1" will not be overwritten if it already exists)
RENAME DIRECTORY my_directory0 AS my_directory1

# rename two directories named "my_directory2" and "my_directory3" as "my_directory4" and
"my_directory5" respectively (neither directory will be overwritten if it already exists)
RENAME DIRECTORY my_directory2, my_directory3 AS my_directory4, my_directory5

# move a directory named "my_directory6" into a root directory named "data" and rename it
as "my_directory7" (the directory "my_directory7" will not be overwritten if it already
exists)
RENAME DIRECTORY my_directory6 AS /data/my_directory7

# move a directory named "my_directory8" into a relative directory named "backup" (the
directory "my_directory8" will not be overwritten if it already exists)
RENAME DIRECTORY my_directory8 AS backup/
```

6.3.10 RENAME FILE

Syntax

RENAME [**TRUNCATE**] **FILE** *file_name1, ..., file_nameX* **AS** *new_file_name1, ..., new_file_nameX*

Description

Rename (or move) an existing file named *file_name* as *new_file_name*. Multiple files can be renamed (or moved) at once by separating these with a comma (,). If *new_file_name* already exists, it will not be overwritten, no subsequent files are renamed (or moved), and an error is raised. To overwrite an existing file, specify the keyword TRUNCATE (ALL DATA STORED IN THE FILE WILL BE PERMANENTLY LOST).

Parameter(s)

file_name – name of the file to rename (or move). Multiple files are separated with a comma (,).

new_file_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# rename a file named "my_file0.h5" as "my_file1.h5" (the file "my_file1.h5" will not be overwritten if it already exists)
RENAME FILE my_file0.h5 AS my_file1.h5

# rename a file named "my_file2.h5" as "my_file3.h5" in file "my_file0.h5" (the external link will not be overwritten if it already exists)
RENAME TRUNCATE FILE my_file2.h5 AS my_file3.h5

# rename two files named "my_file4.h5" and "my_file5.h5" as "my_file6.h5" and "my_file7.h5" respectively (both files will be overwritten if they already exist)
RENAME TRUNCATE FILE my_file4.h5, my_file5.h5 AS my_file6.h5, my_file7.h5

# move a file named "my_file8.h5" into a root directory named "data" and rename it as "my_file9.h5" (the file "my_file9.h5" will not be overwritten if it already exists)
RENAME FILE my_file8.h5 AS /data/my_file9.h5

# move a file named "my_file10.h5" into a relative directory named "backup" (the file "my_file10.h5" will not be overwritten if it already exists)
RENAME FILE my_file10.h5 AS backup/
```

6.3.11 RENAME [GROUP | DATASET | ATTRIBUTE]

Syntax

RENAME [**TRUNCATE**] [**GROUP** | **DATASET** | **ATTRIBUTE**] *object_name1, ..., object_nameX* **AS** *new_object_name1, ..., new_object_nameX*

Description

Rename (or move) an existing HDF group, dataset or attribute named *object_name* as *new_object_name*. Multiple groups, datasets or attributes can be renamed (or moved) at once by separating these with a comma (,). If *new_object_name* already exists, it will not be overwritten, no subsequent objects are renamed (or moved), and an error is raised. To overwrite an existing object, specify the keyword TRUNCATE (ALL DATA STORED IN THE OBJECT WILL BE PERMANENTLY LOST). In case (1) a group and an attribute or (2) a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword GROUP, DATASET nor ATTRIBUTE is specified, the object to be renamed is the group or dataset (the attribute will not be renamed – to rename it, the operation must be executed again). To explicitly rename an object according to its type, the keyword GROUP, DATASET or ATTRIBUTE must be specified. While the renaming (or moving) of groups and datasets to a different location is supported by the HDF library, this is not the case for attributes; HDFql overcomes this limitation by (1) creating a new attribute with the same characteristics as the existing one (e.g. datatype, number of dimensions) using the new specified location and name, (2) writing the data from the existing attribute to the newly created attribute, and (3) deleting the existing attribute.

Parameter(s)

object_name – name of the object to rename (or move). Multiple objects are separated with a comma (,).

new_object_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.3.12 COPY FILE

Syntax

COPY [**TRUNCATE**] **FILE** *file_name1, ..., file_nameX* **TO** *new_file_name1, ..., new_file_nameX*

Description

Copy an existing file named *file_name* to *new_file_name*. Multiple files can be copied at once by separating these with a comma (,). If *new_file_name* already exists, it will not be overwritten, no subsequent files are copied, and an error is raised. To overwrite an existing file, specify the keyword TRUNCATE (ALL DATA STORED IN THE FILE WILL BE PERMANENTLY LOST).

Parameter(s)

file_name – name of the file to copy. Multiple files are separated with a comma (,).

new_file_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.3.13 COPY [GROUP | DATASET | ATTRIBUTE]

Syntax

COPY [**TRUNCATE**] [**GROUP** | **DATASET** | **ATTRIBUTE**] *object_name1, ..., object_nameX* **TO** *new_object_name1, ..., new_object_nameX*

Description

Copy an existing HDF group, dataset or attribute named *object_name* to *new_object_name*. Multiple groups, datasets or attributes can be copied at once by separating these with a comma (,). If *new_object_name* already exists, it will not be overwritten, no subsequent objects are copied, and an error is raised. To overwrite an existing object, specify the keyword TRUNCATE (ALL DATA STORED IN THE OBJECT WILL BE PERMANENTLY LOST). In case (1) a group and an attribute or (2) a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword GROUP, DATASET nor ATTRIBUTE is specified, the object to be copied is the group or dataset. To explicitly copy an object according to its type, the keyword GROUP, DATASET or ATTRIBUTE must be specified.

Parameter(s)

object_name – name of the object to copy. Multiple objects are separated with a comma (,).

new_object_name – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.3.14 DROP DIRECTORY

Syntax

DROP DIRECTORY *directory_name1, ..., directory_nameX*

Description

Drop (i.e. delete) an existing directory named *directory_name*. Multiple directories can be dropped at once by separating these with a comma (,). If *directory_name* contains directories or files (i.e. if it is not empty), it will not be dropped, no subsequent directories are dropped, and an error is raised.

Parameter(s)

directory_name – name of the directory to drop (i.e. delete). Multiple directories are separated with a comma (,).

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.3.15 DROP FILE

Syntax

DROP FILE *file_name1, ..., file_nameX*

Description

Drop (i.e. delete) an existing file named *file_name*. Multiple files can be dropped at once by separating these with a comma (,). If *file_name* was not found or could not be dropped (due to unknown/unexpected reasons), no subsequent files are dropped, and an error is raised.

Parameter(s)

file_name – name of the file to drop (i.e. delete). Multiple files are separated with a comma (,).

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.3.16 DROP [GROUP | DATASET | ATTRIBUTE]

Syntax

DROP {**GROUP** | **DATASET** | **ATTRIBUTE**} | {[**GROUP** | **DATASET** | **ATTRIBUTE**] [{*object_name1, ..., object_nameX*} | {[*object_name*] **LIKE** *regular_expression* [**DEEP** *deep_value*]}]}

Description

Drop (i.e. delete) an existing HDF group, dataset or attribute named *object_name*. Multiple groups, datasets or attributes can be dropped at once by separating these with a comma (,). If *object_name* was not found or could not be dropped (due to unknown/unexpected reasons), no subsequent objects are dropped, and an error is raised. In case (1) a group and an attribute or (2) a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword GROUP, DATASET nor ATTRIBUTE is specified, the object to be dropped is the group or dataset (the attribute will not be dropped – to drop it, the operation must be executed again). To explicitly drop an object according to its type, the keyword GROUP, DATASET or ATTRIBUTE must be specified.

Parameter(s)

object_name – name of the object to drop (i.e. delete). Multiple objects are separated with a comma (,).

regular_expression – to be defined.

deep_value – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.4 DATA MANIPULATION LANGUAGE (DML)

Data Manipulation Language (DML) is, generally speaking, syntax for defining and modifying data stored in structures. In HDFql, the DML is composed of only one operation (INSERT), which enables the insertion (i.e. writing) of data into HDF datasets or attributes.

6.4.1 INSERT

Syntax

INSERT INTO [**DATASET** | **ATTRIBUTE**] *object_name1, ..., object_nameX* [**(***start1***:***stride1***:***count1***:***block1*, ..., *startX***:***strideX***:***countX***:***blockX***)**]

[**VALUES** {**(***val1, ..., valX***)** | **FROM** {{[**DOS** | **UNIX**] [**TEXT**] **FILE** *file_name* [**SEPARATOR** *separator_value*]} | {**BINARY FILE** *file_name*} | {**MEMORY** *variable_number* [**SIZE** *variable_size*]}}]

Description

Insert (i.e. write) data into an HDF dataset or attribute named *object_name*. Multiple datasets or attributes can be written at once by separating these with a comma (,). If *object_name* was not found or could not be written (due to unknown/unexpected reasons), no subsequent objects are written, and an error is raised. HDFql provides several ways of inserting data into a dataset or attribute from disparate input sources, namely:

- A cursor (default input source when nothing is explicity specified). Example: "*INSERT INTO my_dataset*".
- Direct values. Example: "*INSERT INTO my_dataset VALUES(0, 2, 4, 6, 8)*".
- A text file using optional parameters such as which terminator to use – DOS (CR+LF) or UNIX (LF) – for the end of line (EOL) or which separator to use between elements (of the result set). Example: "*INSERT INTO my_dataset FROM TEXT FILE my_file.txt*".
- A binary file. Example: "*INSERT INTO my_dataset FROM BINARY FILE my_file.bin*".
- A variable that was previously registered through the function hdfql_variable_register. Example: "*INSERT INTO my_dataset FROM MEMORY 0*".

In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the object that will have data inserted into it is the dataset. To explicitly insert data into an object according to its type, the keyword DATASET or ATTRIBUTE must be specified.

By default, the entire *object_name* is written when performing an insert operation. To write only a subset (i.e. portion) of *object_name*, hyperslab[18] functionalities can be used (these are only available for datasets; i.e. not for attributes[19]). To enable hyperslabs, the *start*, *stride*, *count* and *block* parameters may be specified and separated by a colon (:). For each dimension of *object_name*, a set of such parameters may be specified and each set should be separated by a comma (,). In case *start* is not specified, its default value is 0 (i.e. the first position of the dimension in question); in case *start* is negative, its value will be the last position of the dimension in question minus the value of *start*. In case *stride* is not specified, its default value is equal to the value of *block*. In case *count* is not specified, its default value is 1. In case *block* is not specified, its default value is the size of the dimension in question minus the value of *start*. Since hyperslabs can be complicated to set up, the operation ENABLE DEBUG may be helpful to obtain info/debug information in case of errors.

[18] At the time of writing, only regular hyperslabs are supported by HDFql. Additional hyperslabs will be supported in the near future, namely irregular hyperslabs and per element hyperslabs.

[19] By design, hyperslabs for attributes are not supported by the HDF5 API. To overcome this limitation, HDFql will implement (pseudo) hyperslabs to enable writing a subset of an attribute in the near future.

Parameter(s)

object_name – name of the HDF dataset or attribute to insert (i.e. write) data into. Multiple datasets or attributes are separated with a comma (,).

start – to be defined.

stride – to be defined.

count – to be defined.

block – to be defined.

val – to be defined.

file_name – to be defined.

separator_value – to be defined.

variable_number – number of the variable whose data will be inserted (i.e. written) into the HDF dataset or attribute. The number is returned by the function hdfql_variable_register upon registering the variable or, subsequently, returned by the function hdfql_variable_get_number.

variable_size – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# create dataset named "my_dataset0" of type short of one dimension (size 3)
CREATE DATASET my_dataset0 AS SMALLINT(3)

# create dataset named "my_dataset1" of type int of one dimension (size 5)
CREATE DATASET my_dataset1 AS INT(5)

# insert (i.e. write) values into dataset "my_dataset0"
INSERT INTO my_dataset0 VALUES(65, 66, 67)
```

```
# populate cursor in use with data from dataset "my_dataset0" (should be 65, 66, 67)
SELECT FROM my_dataset0

# insert (i.e. write) values into dataset "my_dataset1" from cursor in use (should be 65, 66, 67, 0, 0)
INSERT INTO my_dataset1
```

```
# create dataset named "my_dataset2" of type float of one dimension (size 512)
CREATE DATASET my_dataset2 AS FLOAT(512)

# insert (i.e. write) values into dataset "my_dataset2" from a text file named "my_file0.txt" that has values separated with "," (i.e. default separator)
INSERT INTO my_dataset2 VALUES FROM FILE my_file0.txt

# insert (i.e. write) values into dataset "my_dataset2" from a text file named "my_file1.txt" that has a DOS-based end of line (EOL) terminator and values separated with "**"
INSERT INTO my_dataset2 VALUES FROM DOS TEXT FILE my_file1.txt SEPARATOR **

// insert (i.e. write) values into dataset "my_dataset2" from a binary file named "my_file.bin"
INSERT INTO my_dataset2 VALUES FROM BINARY FILE my_file.bin
```

```
# create dataset named "my_dataset3" of type short of one dimension (size 5)
CREATE DATASET my_dataset3 AS SMALLINT(5)

# insert (i.e. write) value 9 into position #3 of dataset "my_dataset3" using hyperslabs
INSERT INTO my_dataset3(3) VALUES(9)

# populate cursor in use with data from dataset "my_dataset3" (should be 0, 0, 0, 9, 0)
SELECT FROM my_dataset3

# insert (i.e. write) value 9 into position #4 of dataset "my_dataset3" using hyperslabs
INSERT INTO my_dataset3(-1) VALUES(7)

# populate cursor in use with data from dataset "my_dataset3" (should be 0, 0, 0, 9, 7)
SELECT FROM my_dataset3
```

```
# insert (i.e. write) values 5 and 3 into position #0 and #1 of dataset "my_dataset3" using hyperslabs
INSERT INTO my_dataset3(:::2) VALUES(5, 3)

# populate cursor in use with data from dataset "my_dataset3" (should be 5, 3, 0, 9, 7)
SELECT FROM my_dataset3

# create dataset named "my_dataset4" of type int of two dimensions (size 3x3)
CREATE DATASET my_dataset4 AS INT(3, 3)

# insert (i.e. write) value 8 into position #2 of the first dimension and position #1 of the second dimension of dataset "my_dataset4" using hyperslabs
INSERT INTO my_dataset4(2, 1) VALUES(8)

# populate cursor in use with data from dataset "my_dataset4" (should be 0, 0, 0, 0, 0, 0, 0, 8, 0)
SELECT FROM my_dataset4

# insert (i.e. write) values 4 and 6 into position #2 of the first dimension and position #1 of the second dimension of dataset "my_dataset4" using hyperslabs
INSERT INTO my_dataset4(1, 1:) VALUES(4, 6)

# populate cursor in use with data from dataset "my_dataset4" (should be 0, 0, 0, 0, 4, 6, 0, 8, 0)
SELECT FROM my_dataset4
```

```
// declare variables
char script[1024];
double data[2][2];

// create a dataset named "my_dataset3" of type double of two dimensions (size 2x2)
hdfql_execute("CREATE DATASET my_dataset3 AS DOUBLE(2, 2)");

// assign values to variable "data"
data[0][0] = 21.1;
data[0][1] = 18.8;
data[1][0] = 75.6;
data[1][1] = 56.3;
```

```
// register variable "data" for subsequent use (by HDFql)
hdfql_variable_register(&data);

// prepare script to insert (i.e. write) values from variable "data" into dataset
"my_dataset3"
sprintf(script, "INSERT INTO my_dataset3 VALUES FROM MEMORY %u",
hdfql_variable_get_number(&data));

// execute script
hdfql_execute(script);

// unregister variable "data" as it is no longer used/needed (by HDFql)
hdfql_variable_unregister(&data);
```

6.5 DATA QUERY LANGUAGE (DQL)

Data Query Language (DQL) is, generally speaking, syntax for retrieving data stored in structures. In HDFql, the DQL is composed of only one operation (SELECT). It enables retrieval (i.e. reading) of data stored in HDF datasets or attributes according to certain criteria. Moreover, it supports POST-PROCESSING options to further process/transform results of the operation according to the programmer's needs.

6.5.1 SELECT

Syntax

SELECT FROM [**DATASET** | **ATTRIBUTE**] *object_name* [**(***start1***:***stride1***:***count1***:***block1*, ..., *startX***:***strideX***:***countX***:***blockX***)**]

[**CACHE** [**SLOTS** {*slots_value* | **DEFAULT** | **FILE**}] [**SIZE** {*size_value* | **DEFAULT** | **FILE**}] [**PREEMPTION** {*preemption_value* | **DEFAULT** | **FILE**}]]

[*post_processing_option1* ... *post_processing_optionX*]

Description

Select (i.e. read) data from an HDF dataset or attribute named *object_name*. In case the keyword CACHE is specified, the dataset is read using cache parametrized with the values *slots_value*, *size_value* and *preemption_value* (this will overwrite any dataset cache parameters that may have been set through the operation SET [FILE | DATASET] CACHE). HDFql provides several ways of writing data that was read from a dataset or attribute into disparate output sources, namely:

- A cursor (default output source when nothing is explicity specified). Example: "*SELECT FROM my_dataset*".
- A text file using optional parameters such as which terminator to use – DOS (CR+LF) or UNIX (LF) – for the end of line (EOL) or which separator to use between elements (of the result set). Example: "*SELECT FROM my_dataset INTO TEXT FILE my_file.txt*".
- A binary file. Example: "*SELECT FROM my_dataset INTO BINARY FILE my_file.bin*".
- A variable that was previously registered through the function hdfql_variable_register. Example: "*SELECT FROM my_dataset INTO MEMORY 0*".

In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the object for which data will be read is the dataset. To explicitly read data from an object according to its type, the keyword DATASET or ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

By default, the entire *object_name* is read when performing a select operation. To read only a subset (i.e. portion) of *object_name*, hyperslab[20] functionalities can be used (these are only available for datasets; i.e. not for attributes[21]). To enable hyperslabs, the *start*, *stride*, *count* and *block* parameters may be specified and separated by a colon (:). For each dimension of *object_name*, a set of such parameters may be specified and each set should be separated by a comma (,). In case *start* is not specified, its default value is 0 (i.e. the first position of the dimension in question); in case *start* is negative, its value will be the last position of the

[20] At the time of writing, only regular hyperslabs are supported by HDFql. Additional hyperslabs will be supported in the near future, namely irregular hyperslabs and per element hyperslabs.

[21] By design, hyperslabs for attributes are not supported by the HDF5 API. To overcome this limitation, HDFql will implement (pseudo) hyperslabs to enable reading a subset of an attribute in the near future.

dimension in question minus the value of *start*. In case *stride* is not specified, its default value is equal to the value of *block*. In case *count* is not specified, its default value is 1. In case *block* is not specified, its default value is the size of the dimension in question minus the value of *start*. Since hyperslabs can be complicated to set up, the operation ENABLE DEBUG may be helpful to obtain info/debug information in case of errors.

Parameter(s)

object_name – name of the HDF dataset or attribute to select (i.e. read) data from.

start – to be defined.

stride – to be defined.

count – to be defined.

block – to be defined.

slots_value – to be defined.

size_value – to be defined.

preemption_value – to be defined.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with data of the dataset or attribute in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6 DATA INTROSPECTION LANGUAGE (DIL)

HDFql has certain operations that retrieve information about the internals of HDF files but also about HDFql itself and the runtime environment. These operations are part of the Data Introspection Language (DIL) and they all begin with the keyword SHOW. Moreover, these operations support POST-PROCESSING options to further process/transform the result of operations according to the programmer's needs. Typically, a DIL operation has the following syntactical form:

SHOW *operation_name* [*post_processing_option1* ... *post_processing_optionX*]

6.6.1 SHOW FILE VALIDITY

Syntax

SHOW FILE VALIDITY *file_name1, ..., file_nameX*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get the validity of a file named *file_name*. Multiple files' validities can be checked at once by separating these with a comma (,). If *file_name* was not found or its validity could not be checked (due to unknown/unexpected reasons), no subsequent files are checked, and an error is raised. The result of the operation can either be HDFQL_YES or HDFQL_NO depending on whether *file_name* is a valid HDF file or not. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

file_name – name of the file whose validity is to be obtained. Multiple files are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO

post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF file named "my_file0.h5"
CREATE FILE my_file0.h5

# show (i.e. get) validity of file "my_file0.h5" (should be 0 - i.e. HDFQL_YES)
SHOW FILE VALIDITY my_file0.h5

# run touch command to create an empty file named "not_an_hdf_file"
RUN "touch not_an_hdf_file"

# show (i.e. get) validity of file "not_an_hdf_file" (should be -1 - i.e. HDFQL_NO)
SHOW FILE VALIDITY not_an_hdf_file

# show (i.e. get) validity of both files "my_file.h5" and "not_an_hdf_file" at once
(should be 0, -1)
SHOW FILE VALIDITY my_file0.h5, not_an_hdf_file
```

6.6.2 SHOW USE DIRECTORY

Syntax

SHOW USE DIRECTORY

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get the working directory currently in use. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# set working directory currently in use to "/"
USE DIRECTORY /

# show (i.e. get) current working directory (should be /)
SHOW USE DIRECTORY

# create a directory named "my_directory"
CREATE DIRECTORY my_directory

# set working directory currently in use to "my_directory" (more precisely
"/my_directory")
USE DIRECTORY my_directory

# show (i.e. get) current working directory (should be /my_directory)
SHOW USE DIRECTORY

# create two directories named "my_subdirectory0" and "my_subdirectory1" (both
directories will be created in directory "/my_directory")
CREATE DIRECTORY my_subdirectory0, my_subdirectory1

# set directory currently in use to "my_subdirectory0" (more precisely
"/my_directory/my_subdirectory0")
USE DIRECTORY my_subdirectory0

# set directory currently in use to "my_subdirectory1" located one level up (more
precisely "/my_directory/my_subdirectory1")
```

```
USE DIRECTORY ../my_subdirectory1

# set directory currently in use two levels up (should be /)
USE DIRECTORY ../..
```

6.6.3 SHOW USE FILE

Syntax

SHOW USE FILE

[*post_processing_option1 ... post_processing_optionX*]

Description

Get the HDF file currently in use. If no file is in use, the result of the operation is empty. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.4 SHOW ALL USE FILE

Syntax

SHOW ALL USE FILE

[*post_processing_option1 ... post_processing_optionX*]

Description

Get all HDF files in use (i.e. open). If no files are in use, the result of the operation is empty. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.5 SHOW USE GROUP

Syntax

SHOW USE GROUP

[*post_processing_option1 ... post_processing_optionX*]

Description

Get the HDF group currently in use. If no file is in use, the result of the operation is empty. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# use (i.e. open) an HDF file named "my_file.h5"
USE FILE my_file.h5

# show (i.e. get) current working group (should be /)
SHOW USE GROUP

# create an HDF group named "my_group"
CREATE GROUP my_group

# set group currently in use to "my_group" (more precisely "/my_group")
USE GROUP my_group

# show (i.e. get) current working group (should be /my_group)
SHOW USE GROUP

# create two HDF groups named "my_subgroup0" and "my_subgroup1" (both groups will be
created in group "/my_group")
CREATE GROUP my_subgroup0, my_subgroup1
```

```
# set group currently in use to "my_subgroup0" (more precisely "/my_group/my_subgroup0")
USE GROUP my_subgroup0

# show (i.e. get) current working group (should be /my_group/my_subgroup0)
SHOW USE GROUP

# set group currently in use to "." (the group currently in use will not change as "."
refers to the current working group itself)
USE GROUP .

# show (i.e. get) current working group (should be /my_group/my_subgroup0)
SHOW USE GROUP

# set group currently in use to "my_subgroup1" located one level up (more precisely
"/my_group/my_subgroup1")
USE GROUP ../my_subgroup1

# set group currently in use two levels up (should be /)
USE GROUP ../..
```

6.6.6 SHOW [GROUP | DATASET | ATTRIBUTE]

Syntax

SHOW [**GROUP** | **DATASET** | **ATTRIBUTE**] [*object_name*] [**LIKE** *regular_expression* [**DEEP** *deep_value*]]

[**WHERE** *condition*]

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get HDF objects (i.e. groups, datasets or attributes) within an HDF group or dataset named *object_name* or check the existence of an object named *object_name*. If *object_name* is not specified, all objects are returned – to return only objects of type group, dataset or attribute, specify the keyword GROUP, DATASET or ATTRIBUTE respectively. If *object_name* is specified, one of the following behaviors applies:

- If it ends with "/", *object_name* will be treated as a group or dataset, and all groups, datasets or attributes stored in *object_name* are returned.
- If it does not end with "/", *object_name* will be checked for its existence. If it does exist, *object_name* is returned; otherwise, if it does not exist, an error is raised.

If the keyword LIKE is specified, only objects with names complying with a regular expression named *regular_expression* will be returned (in HDFql, regular expressions are the ones specified by PCRE which closely follow PERL5 syntax – please refer to http://www.pcre.org and http://perldoc.perl.org/perlre.html for additional information). If *regular_expression* includes "**", recursive search is performed (i.e. HDFql will search in all existing groups and subgroups). To limit the recursiveness, the keyword DEEP may be specified along with a value *deep_value* representing the maximum recursiveness limit. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – to be defined.

regular_expression – to be defined.

deep_value – to be defined.

condition – to be defined.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# set group currently in use to "/" (i.e. the root of the HDF file)
USE GROUP /
```

```
# create two HDF groups named "my_group0" and "my_group1" (both groups will be created in group "/")
CREATE GROUP my_group0, my_group1

# create one HDF dataset named "my_dataset0" of type unsigned short (it will be created in group "/")
CREATE DATASET my_dataset0 AS UNSIGNED SMALLINT

# create one HDF dataset named "my_dataset1" of type short (it will be created in group "/my_group0")
CREATE DATASET my_group0/my_dataset1 AS SMALLINT

# create two HDF attributes named "my_attribute0" and "my_attribute1" of type long long (both attributes will be created in group "/")
CREATE ATTRIBUTE my_attribute0, my_attribute1 AS BIGINT

# create one HDF attribute named "my_attribute2" of type char (it will be created in group "/my_group0")
CREATE ATTRIBUTE my_group0/my_attribute2 AS TINYINT

# create one HDF attribute named "my_attribute3" of type unsigned char (it will be created in dataset "/my_dataset0")
CREATE ATTRIBUTE my_dataset0/my_attribute3 AS UNSIGNED TINYINT

# show (i.e. get) all HDF objects existing in group "/" (should be my_group0, my_group1, my_dataset0, my_attribute0, my_attribute1)
SHOW

# show (i.e. get) all HDF groups existing in group "/" (should be my_group0, my_group1)
SHOW GROUP

# show (i.e. get) all HDF datasets existing in group "/" (should be my_dataset0)
SHOW DATASET

# check if HDF object "my_groupX" exists (should raise an error)
SHOW my_groupX

# check if HDF object "my_group0" exists (should be my_group0)
SHOW my_group0
```

```
# show (i.e. get) all HDF objects existing within group "my_group0" (should be
my_dataset1 and my_attribute2)
SHOW my_group0/

# show (i.e. get) all HDF attributes existing within group "my_group0" (should be
my_attribute2)
SHOW ATTRIBUTE my_group0/

# show (i.e. get) all HDF objects existing within dataset "my_dataset0" (should be
my_attribute3)
SHOW my_dataset0/
```

```
# create an HDF group named "my_group1" that tracks the objects' (i.e. groups and
datasets) creation order within the group
CREATE GROUP my_group1 ORDER TRACKED

# create two HDF groups named "subgroup1" and "subgroup0" (both groups will be created in
group "/my_group1")
CREATE GROUP my_group1/subgroup1, my_group1/subgroup0

# create two HDF datasets named "dataset1" and "dataset0" of type float (both datasets
will be created in group "/my_group1")
CREATE DATASET my_group1/dataset1, my_group1/dataset0 AS FLOAT

# show (i.e. get) all HDF objects existing within group "my_group1" (should be dataset0,
dataset1, subgroup0 and subgroup1)
SHOW my_group1/

# show (i.e. get) all HDF objects existing within group "my_group1" ordered by their time
of creation (should be subgroup1, subgroup0, dataset1 and dataset2)
SHOW my_group1/ ORDER CREATION

# create an HDF dataset named "my_dataset1" of type double that tracks the attributes'
creation order within the dataset
CREATE DATASET my_dataset1 AS DOUBLE ATTRIBUTE ORDER TRACKED

# create two HDF attributes named "attribute2" and "attribute0" of type int (both
attributes will be created in dataset "/my_dataset1")
CREATE ATTRIBUTE my_dataset1/attribute2, my_dataset1/attribute0 AS INT
```

```
# create an HDF attribute named "attribute1" of type short (it will be created in dataset "/my_dataset1")
CREATE ATTRIBUTE my_dataset1/attribute1 AS SMALLINT

# show (i.e. get) all HDF objects existing within dataset "my_dataset1" (should be attribute0, attribute1 and attribute2)
SHOW my_dataset1/

# show (i.e. get) all HDF objects existing within dataset "my_dataset1" ordered by their time of creation (should be attribute2, attribute0 and attribute1)
SHOW my_dataset1/ ORDER CREATION
```

6.6.7 SHOW TYPE

Syntax

SHOW TYPE *object_name1, ..., object_nameX*

[*post_processing_option1 ... post_processing_optionX*]

Description

Get type of an object named *object_name*. Multiple objects' types can be obtained at once by separating these with a comma (,). If *object_name* was not found or its type could not be checked (due to unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. The result of the operation can either be HDFQL_GROUP, HDFQL_DATASET or HDFQL_ATTRIBUTE depending on whether *object_name* is a group, dataset or attribute respectively. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the object whose type is to be obtained. Multiple objects are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF group named "my_object0"
CREATE GROUP my_object0

# create an HDF dataset named "my_object1" of type double
CREATE DATASET my_object1 AS DOUBLE

# create an HDF attribute named "my_object2" of type float
CREATE ATTRIBUTE my_object2 AS FLOAT

# show (i.e. get) type of object "my_object0" (should be 4 - i.e. HDFQL_GROUP)
SHOW TYPE my_object0

# show (i.e. get) type of object "my_object1" (should be 8 - i.e. HDFQL_DATASET)
SHOW TYPE my_object1

# show (i.e. get) type of object "my_object2" (should be 16 - i.e. HDFQL_ATTRIBUTE)
SHOW TYPE my_object2

# show (i.e. get) type of both objects "my_object0" and "my_object2" at once (should be
4, 16)
SHOW TYPE my_object0, my_object2
```

6.6.8 SHOW STORAGE TYPE

Syntax

SHOW STORAGE TYPE *dataset_name1, ..., dataset_nameX*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get storage type of an HDF dataset named *dataset_name*. Multiple datasets' storage types can be obtained at once by separating these with a comma (,). If *dataset_name* was not found or its storage type could not be checked (due to unknown/unexpected reasons), no subsequent datasets are checked, and an error is raised. The result of the operation can either be HDFQL_CONTIGUOUS, HDFQL_COMPACT or HDFQL_CHUNKED depending on whether the storage type is contiguous, compact or chunked respectively. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

dataset_name – name of the HDF dataset whose storage type is to be obtained. Multiple datasets are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.9 SHOW [DATASET | ATTRIBUTE] DATATYPE

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **DATATYPE** *object_name1, ..., object_nameX*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get datatype of an HDF dataset or attribute named *object_name*. Multiple objects' datatypes can be obtained at once by separating these with a comma (,). If *object_name* was not found or its datatype could not be checked (due to unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. The result of the operation can either be HDFQL_TINYINT, HDFQL_UNSIGNED_TINYINT, HDFQL_SMALLINT, HDFQL_UNSIGNED_SMALLINT, HDFQL_INT, HDFQL_UNSIGNED_INT, HDFQL_BIGINT, HDFQL_UNSIGNED_BIGINT, HDFQL_FLOAT, HDFQL_DOUBLE, HDFQL_CHAR, HDFQL_VARTINYINT, HDFQL_UNSIGNED_VARTINYINT, HDFQL_VARSMALLINT, HDFQL_UNSIGNED_VARSMALLINT, HDFQL_VARINT, HDFQL_UNSIGNED_VARINT, HDFQL_VARBIGINT, HDFQL_UNSIGNED_VARBIGINT, HDFQL_VARFLOAT, HDFQL_VARDOUBLE, HDFQL_VARCHAR or HDFQL_OPAQUE (please refer to Table 6.3 for additional information about datatypes). In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the datatype returned belongs to the dataset. To explicitly get the datatype of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose datatype is to be obtained. Multiple datasets or attributes are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type double
CREATE DATASET my_dataset0 AS DOUBLE
```

```
# show (i.e. get) datatype of dataset "my_dataset0" (should be 512 - i.e. HDFQL_DOUBLE)
SHOW DATATYPE my_dataset0

# create an HDF dataset named "my_dataset1" of type float
CREATE DATASET my_dataset1 AS FLOAT

# show (i.e. get) datatype of dataset "my_dataset1" (should be 256 - i.e. HDFQL_FLOAT)
SHOW DATATYPE my_dataset1

# create an HDF dataset named "my_common" of type short
CREATE DATASET my_common AS SMALLINT

# create an HDF attribute named "my_common" of type int
CREATE ATTRIBUTE my_common AS INT

# show (i.e. get) datatype of dataset "my_common" (should be 4 - i.e. HDFQL_SMALLINT)
SHOW DATATYPE my_common

# show (i.e. get) datatype of dataset "my_common" (should be 4 - i.e. HDFQL_SMALLINT)
SHOW DATASET DATATYPE my_common

# show (i.e. get) datatype of attribute "my_common" (should be 16 - i.e. HDFQL_INT)
SHOW ATTRIBUTE DATATYPE my_common
```

6.6.10 SHOW [DATASET | ATTRIBUTE] ENDIANNESS

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **ENDIANNESS** *object_name1, ..., object_nameX*

[*post_processing_option1 ... post_processing_optionX*]

Description

Get endianness of an HDF dataset or attribute named *object_name*. Multiple objects' endiannesses can be obtained at once by separating these with a comma (,). If *object_name* was not found or its endianness could not be checked (due to unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. The result of the operation can either be HDFQL_LITTLE_ENDIAN, HDFQL_BIG_ENDIAN or

HDFQL_UNDEFINED depending on whether the endianness is little, big or undefined (i.e. endianness is not applicable to *object_name*) respectively. In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the endianness returned belongs to the dataset. To explicitly get the endianness of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose endiannessis to be obtained. Multiple datasets or attributes are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.11 SHOW [DATASET | ATTRIBUTE] CHARSET

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **CHARSET** *object_name1, ..., object_nameX*

[*post_processing_option1 ... post_processing_optionX*]

Description

Get charset of an HDF dataset or attribute named *object_name*. Multiple objects' charsets can be obtained at once by separating these with a comma (,). If *object_name* was not found or its charset could not be checked (due to unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. The result of the operation can either be HDFQL_ASCII, HDFQL_UTF8 or HDFQL_UNDEFINED depending on whether the charset is ASCII, UTF8 or undefined (i.e. *object_name* is neither of datatype HDFQL_CHAR nor HDFQL_VARCHAR) respectively. In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the charset returned belongs to the dataset. To explicitly get the charset of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose charset is to be obtained. Multiple datasets or attributes are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type char
CREATE DATASET my_dataset0 AS CHAR

# show (i.e. get) charset of dataset "my_dataset0" (should be 1 - i.e. HDFQL_ASCII)
SHOW CHARSET my_dataset0

# create an HDF dataset named "my_dataset1" of type char of one dimension (size 20)
```

```
CREATE DATASET my_dataset1 AS UTF8 CHAR(20)

# show (i.e. get) charset of dataset "my_dataset1" (should be 2 - i.e. HDFQL_UTF8)
SHOW CHARSET my_dataset1

# create an HDF dataset named "my_common" of type short
CREATE DATASET my_common AS UTF8 CHAR

# create an HDF attribute named "my_common" of type variable-length char
CREATE ATTRIBUTE my_common AS ASCII VARCHAR

# show (i.e. get) charset of dataset "my_common" (should be 2 - i.e. HDFQL_UTF8)
SHOW CHARSET my_common

# show (i.e. get) datatype of dataset "my_common" (should be 2 - i.e. HDFQL_UTF8)
SHOW DATASET CHARSET my_common

# show (i.e. get) charset of attribute "my_common" (should be 1 - i.e. HDFQL_ASCII)
SHOW ATTRIBUTE CHARSET my_common
```

6.6.12 SHOW STORAGE DIMENSION

Syntax

SHOW STORAGE DIMENSION *dataset_name*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get storage dimensions of an HDF dataset named *dataset_name*. If *dataset_name* is chunked (i.e. its storage type is HDFQL_CHUNKED), it returns the chunk layout dimensions; otherwise, if it is not chunked, no result is returned. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

dataset_name – name of the HDF dataset whose storage dimensions are to be obtained.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type unsigned int
CREATE DATASET my_dataset0 AS UNSIGNED INT

# show (i.e. get) storage dimensions of dataset "my_dataset0" (should be empty)
SHOW STORAGE DIMENSION my_dataset0

# create an HDF dataset named "my_dataset1" of type double of one dimension (size 15)
CREATE CHUNKED DATASET my_dataset1 AS DOUBLE(15)

# show (i.e. get) storage dimensions of dataset "my_dataset1" (should be 15)
SHOW STORAGE DIMENSION my_dataset1

# create an HDF dataset named "my_dataset2" of type float of three dimensions (size
3x5x20)
CREATE CHUNKED(1, 2, 10) DATASET my_dataset2 AS FLOAT(3, 5, 20)

# show (i.e. get) storage dimensions of dataset "my_dataset2" (should be 1, 2, 10)
SHOW STORAGE DIMENSION my_dataset2
```

6.6.13 SHOW [DATASET | ATTRIBUTE] DIMENSION

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **DIMENSION** *object_name*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get dimensions of an HDF dataset or attribute named *object_name*. In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the dimensions returned belong to the dataset. To explicitly get the dimensions of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. If *object_name* does not have a dimension (i.e. if it is scalar), the returned value is one. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose dimensions are to be obtained.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type unsigned int
CREATE DATASET my_dataset0 AS UNSIGNED INT

# show (i.e. get) dimensions of dataset "my_dataset0" (should be 1)
SHOW DIMENSION my_dataset0

# create an HDF dataset named "my_dataset1" of type double of one dimension (size 15)
CREATE DATASET my_dataset1 AS DOUBLE(15)

# show (i.e. get) dimensions of dataset "my_dataset1" (should be 15)
SHOW DIMENSION my_dataset1

# create an HDF attribute named "my_attribute0" of type int of one dimension (size 1)
CREATE ATTRIBUTE my_attribute0 AS INT(1)
```

```
# show (i.e. get) dimensions of attribute "my_attribute0" (should be 1)
SHOW DIMENSION my_attribute0

# create an HDF attribute named "my_attribute1" of type short of two dimensions (size
2x3)
CREATE ATTRIBUTE my_attribute1 AS SMALLINT(2, 3)

# show (i.e. get) dimensions of attribute "my_attribute1" (should be 2, 3)
SHOW DIMENSION my_attribute1

# create an HDF dataset named "my_dataset2" of type float of three dimensions (first
dimension with size 2 and extendible up to 10; second dimension with size 5; third
dimension with size 20 and extendible to an unlimited size)
CREATE CHUNKED DATASET my_dataset2 AS FLOAT(3 TO 10, 5, 20 TO UNLIMITED)

# show (i.e. get) dimensions of dataset "my_dataset2" (should be 3, 5, 20)
SHOW DIMENSION my_dataset2
```

6.6.14 SHOW [DATASET | ATTRIBUTE] MAX DIMENSION

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **MAX DIMENSION** *object_name*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get maximum dimensions of an HDF dataset or attribute named *object_name*. In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the dimensions returned belong to the dataset. To explicitly get the maximum dimensions of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. If *object_name* does not have a dimension (i.e. if it is scalar), the returned value is one. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose maximum dimensions are to be obtained.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type unsigned int
CREATE DATASET my_dataset0 AS UNSIGNED INT

# show (i.e. get) maximum dimensions of dataset "my_dataset0" (should be 1)
SHOW MAX DIMENSION my_dataset0

# create an HDF dataset named "my_dataset1" of type double of one dimension (size 15)
CREATE DATASET my_dataset1 AS DOUBLE(15)

# show (i.e. get) maximum dimensions of dataset "my_dataset1" (should be 15)
SHOW MAX DIMENSION my_dataset1

# create an HDF attribute named "my_attribute0" of type int of one dimension (size 1)
CREATE ATTRIBUTE my_attribute0 AS INT(1)

# show (i.e. get) dimensions of attribute "my_attribute0" (should be 1)
SHOW DIMENSION my_attribute0

# create an HDF attribute named "my_attribute1" of type short of two dimensions (size
2x3)
CREATE ATTRIBUTE my_attribute1 AS SMALLINT(2, 3)

# show (i.e. get) maximum dimensions of attribute "my_attribute1" (should be 2, 3)
SHOW MAX DIMENSION my_attribute1

# create an HDF dataset named "my_dataset2" of type float of three dimensions (first
```

```
dimension with size 2 and extendible up to 10; second dimension with size 5; third
dimension with size 20 and extendible to an unlimited size)
CREATE CHUNKED DATASET my_dataset2 AS FLOAT(3 TO 10, 5, 20 TO UNLIMITED)

# show (i.e. get) maximum dimensions of dataset "my_dataset2" (should be 10, 5, -1)
SHOW MAX DIMENSION my_dataset2
```

6.6.15 SHOW [ATTRIBUTE] ORDER

Syntax

SHOW [**ATTRIBUTE**] **ORDER** *object_name1, ..., object_nameX*

[*post_processing_option1 ... post_processing_optionX*]

Description

Get (creation) order strategy of an HDF group or dataset named *object_name*. Multiple objects' (creation) order strategies can be obtained at once by separating these with a comma (,). If *object_name* was not found or its (creation) order strategy could not be checked (due to unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. The result of the operation can either be HDFQL_TRACKED, HDFQL_INDEXED or HDFQL_UNDEFINED depending on whether the (creation) order strategy is tracked, indexed or undefined (i.e. *object_name* was created without any (creation) order strategy) respectively. By default, the returned (creation) order strategy refers to objects (i.e. groups and datasets) within a group; to return the (creation) order strategy of attributes within a group or dataset, the keyword ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF group or dataset whose (creation) order strategy is to be obtained. Multiple groups or datasets are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF group named "my_group0"
CREATE GROUP my_group0

# show (i.e. get) (creation) order strategy of objects within group "my_group0" (should
be -1 - i.e. HDFQL_UNDEFINED)
SHOW ORDER my_group0

# show (i.e. get) (creation) order strategy of attributes within group "my_group0"
(should be -1 - i.e. HDFQL_UNDEFINED)
SHOW ATTRIBUTE ORDER my_group0

# create an HDF group named "my_group1" that tracks both the objects' (i.e. groups and
datasets) and the attributes' creation order within the group
CREATE GROUP my_group1 ORDER TRACKED ATTRIBUTE ORDER INDEXED

# show (i.e. get) (creation) order strategy of objects within group "my_group1" (should
be 1 - i.e. HDFQL_TRACKED)
SHOW ORDER my_group1

# show (i.e. get) (creation) order strategy of attributes within group "my_group1"
(should be 2 - i.e. HDFQL_INDEXED)
SHOW ATTRIBUTE ORDER my_group1

# create an HDF dataset named "my_dataset0" of type int that tracks the attributes'
creation order within the dataset
CREATE DATASET my_dataset0 AS INT ATTRIBUTE ORDER TRACKED

# show (i.e. get) (creation) order strategy of attributes within dataset "my_dataset0"
(should be 1 - i.e. HDFQL_TRACKED)
SHOW ATTRIBUTE ORDER my_dataset0
```

```
# show (i.e. get) (creation) order strategy of attributes within both group "my_group1"
and dataset "my_dataset0" at once (should be 2, 1)
SHOW ATTRIBUTE ORDER my_group1, my_dataset0
```

6.6.16 SHOW [DATASET | ATTRIBUTE] TAG

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **TAG** *object_name1, ..., object_nameX*

[*post_processing_option1 ... post_processing_optionX*]

Description

Get tag of an HDF dataset or attribute named *object_name*. Multiple objects' tags can be obtained at once by separating these with a comma (,). If *object_name* was not found or its tag could not be checked (due to its datatype not being HDFQL_OPAQUE or for unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the tag returned belongs to the dataset. To explicitly get the tag of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose tag is to be obtained. Multiple datasets or attributes are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# create an HDF dataset named "my_dataset0" of type opaque
CREATE DATASET my_dataset0 AS OPAQUE

# show (i.e. get) tag of dataset "my_dataset0" (should be empty)
SHOW TAG my_dataset0

# create an HDF dataset named "my_dataset1" of type opaque of one dimension (size 15)
with a tag value "my_tag1"
CREATE DATASET my_dataset1 AS OPAQUE(15) TAG my_tag1

# show (i.e. get) tag of dataset "my_dataset1" (should be my_tag1)
SHOW TAG my_dataset1

# create an HDF attribute named "my_attribute0" of type opaque of two dimensions (size
3x5) with a tag value "Hierarchical Data Format"
CREATE ATTRIBUTE my_attribute0 AS OPAQUE(3, 5) TAG "Hierarchical Data Format"

# show (i.e. get) tag of attribute "my_attribute0" (should be Hierarchical Data Format)
SHOW TAG my_attribute0
```

6.6.17 SHOW FILE SIZE

Syntax

SHOW FILE SIZE [*file_name1, ..., file_nameX*]

[*post_processing_option1 ... post_processing_optionX*]

Description

Get size (in bytes) of a file named *file_name*. Multiple files' sizes can be obtained at once by separating several file names with a comma (,). If *file_name* was not found or its size could not be checked (due to unknown/unexpected reasons), no subsequent files are checked, and an error is raised. If no file is specified then the size (in bytes) of the file currently in use will be returned instead. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

file_name – name of the file whose size (in bytes) is to be obtained. Multiple files are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.18 SHOW [DATASET | ATTRIBUTE] SIZE

Syntax

SHOW [**DATASET** | **ATTRIBUTE**] **SIZE** *object_name1, ..., object_nameX*

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get size (in bytes) of an HDF dataset or attribute named *object_name*. Multiple objects' sizes can be obtained at once by separating these with a comma (,). If *object_name* was not found or its size could not be checked (due to unknown/unexpected reasons), no subsequent objects are checked, and an error is raised. In case a dataset and an attribute with identical names (*object_name*) are stored in the same location (i.e. group) and neither the keyword DATASET nor ATTRIBUTE is specified, the size returned belongs to the dataset. To explicitly get the size of *object_name* according to its type, the keyword DATASET or ATTRIBUTE must be specified. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

object_name – name of the HDF dataset or attribute whose size is to be obtained. Multiple datasets or attributes are separated with a comma (,).

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.19 SHOW RELEASE DATE

Syntax

SHOW RELEASE DATE

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get release date of HDFql. The format of the date returned is YYYY/MM/DD. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# show (i.e. get) release date of HDFql (should be something similar to 2017/03/21)
SHOW RELEASE DATE
```

6.6.20 SHOW HDFQL VERSION

Syntax

SHOW HDFQL VERSION

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get version of HDFql library. The format of the version returned is MAJOR.MINOR.REVISION. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and

independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# show (i.e. get) version of HDFql library (should be something similar to 1.4.0)
SHOW HDFQL VERSION
```

6.6.21 SHOW HDF VERSION

Syntax

SHOW HDF VERSION

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get version of the HDF library used by HDFql. The format of the version returned is MAJOR.MINOR.REVISION. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# show (i.e. get) version of the HDF library used by HDFql (should be something similar to 1.8.16)
SHOW HDF VERSION
```

6.6.22 SHOW PCRE VERSION

Syntax

SHOW PCRE VERSION

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get version of the PCRE library used by HDFql. The format of the version returned is MAJOR.MINOR. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# show (i.e. get) version of the PCRE library used by HDFql (should be something similar to 8.39)
```

```
SHOW PCRE VERSION
```

6.6.23 SHOW ZLIB VERSION

Syntax

SHOW ZLIB VERSION

[*post_processing_option1 ... post_processing_optionX*]

Description

Get version of the ZLIB library used by HDFql. The format of the version returned is MAJOR.MINOR.REVISION. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# show (i.e. get) version of the ZLIB library used by HDFql (should be something similar
to 1.2.11)
SHOW ZLIB VERSION
```

6.6.24 SHOW DIRECTORY

Syntax

SHOW DIRECTORY [*directory_name*]

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get directory names within a directory named *directory_name*. If *directory_name* is not specified, all directory names within the current working directory are returned. Otherwise, if *directory_name* is specified, all directory names within this directory are returned. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

directory_name – name of the directory whose directory names are to be obtained.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.25 SHOW FILE

Syntax

SHOW FILE [*directory_name* | *file_name*]

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get file names within a directory named *directory_name* or check existence of a file named *file_name*. If neither *directory_name* nor *file_name* are specified, all file names within the current working directory are returned. If *directory_name* is specified, all file names within this directory are returned. Alternatively, if *file_name* is specified, its existence is checked: if the file exists, its name is returned; otherwise (if it does not exist), an error is raised. Multiple files can be checked for their existence at once by separating these with a comma (,). Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

directory_name – to be defined.

file_name – to be defined.

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.26 SHOW MAC ADDRESS

Syntax

SHOW MAC ADDRESS

[*post_processing_option1 ... post_processing_optionX*]

Description

Get MAC address(es) of the machine where HDFql is executed. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
# show (i.e. get) MAC address(es) of the machine where HDFql is executed (should be
something similar to E7-2A-E9-8B-CA-4E)
SHOW MAC ADDRESS
```

6.6.27 SHOW EXECUTE STATUS

Syntax

SHOW EXECUTE STATUS

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get execution status of the last operation. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.28 SHOW [[USE] FILE | DATASET] CACHE

Syntax

SHOW [[**USE**] **FILE** | **DATASET**] **CACHE** [**SLOTS** | **SIZE** | **PREEMPTION**]

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get cache parameter values for accessing HDF files or datasets. In case neither the keyword SLOT, SIZE nor PREEMPTION is specified, all cache parameter values (i.e. for slots, size and preemption) are returned. To return a specific cache parameter value, the keyword SLOT, SIZE or PREEMPTION must be specified. In case

neither the keyword FILE, USE FILE nor DATASET is specified, the cache parameters returned refers to files by default (optionally, the keyword FILE may be specified to make the purpose of this operation clearer). To explicitly return cache parameters of datasets or the file currently in use, the keyword DATASET or USE FILE must be specified.

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.29 SHOW FLUSH

Syntax

SHOW FLUSH

[*post_processing_option1* ... *post_processing_optionX*]

Description

Get status of the automatic flushing. The status can either be HDFQL_ENABLED or HDFQL_DISABLED. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.6.30 SHOW DEBUG

Syntax

SHOW DEBUG

[*post_processing_option1 ... post_processing_optionX*]

Description

Get status of the debug mechanism. The status can either be HDFQL_ENABLED or HDFQL_DISABLED. Post-processing options may be applied to the result of the operation such as ordering and redirecting (please refer to the section POST-PROCESSING for additional information).

Parameter(s)

None

Return

If the INTO post-processing option is not specified, the cursor in use is populated with the result of the operation in case the operation succeeded; in case the operation failed, the cursor in use is cleared. If the INTO post-processing option is specified, the cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR and subsection INTO for additional information.

Example(s)

```
// TO BE DEFINED
```

6.7 MISCELLANEOUS

This section assembles all remaining HDFql operations that – due to their heterogeneous nature and functionality – do not fit in the previous sections about the language for data definition, manipulation, querying and introspection.

6.7.1 USE DIRECTORY

Syntax

USE DIRECTORY *directory_name*

Description

Use a directory named *directory_name* for subsequent operations. This will change the current working directory to *directory_name* thus avoiding the need to explicitly provide the full path of this directory when working within it (i.e. subsequent operations are done relatively to this directory, unless otherwise specified). If *directory_name* was not found or could not be opened (due to unknown/unexpected reasons), an error is raised.

Parameter(s)

directory_name – name of the directory to use for subsequent operations.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.7.2 USE FILE

Syntax

USE [**READONLY**] **FILE** *file_name1, ..., file_nameX*

[**CACHE** [**SLOTS** {*slots_value* | **DEFAULT**}] [**SIZE** {*size_value* | **DEFAULT**}] [**PREEMPTION** {*preemption_value* | **DEFAULT**}]]

Description

Use (i.e. open) an HDF file named *file_name* for subsequent operations. Multiple files can be opened at once by separating these with a comma (,). If *file_name* was not found or could not be opened (due to unknown/unexpected reasons), no subsequent files are opened, and an error is raised. By default, the file is opened with read/write permissions. To open a file with read only permission, the keyword READONLY should be specified (any subsequent attempt to write into this file will return an error). HDFql tracks opened files in a stack fashion (i.e. LIFO) meaning that the most recently opened file is the one currently in use. In case the keyword CACHE is specified, HDFql opens the file using cache parametrized with the *slots_value*, *size_value* and *preemption_value* values (this will overwrite any file cache that may have been set through the operation SET [FILE | DATASET] CACHE).

Parameter(s)

file_name – name of the HDF file to use (i.e. open) for subsequent operations. Multiple files are separated with a comma (,).

slots_value – to be defined.

size_value – to be defined.

preemption_value – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# use (i.e. open) an HDF file named "my_file0.h5" located in the current working
directory
USE FILE my_file0.h5

# use (i.e. open) an HDF file named "my_file1.h5" located in a root directory named
"data"
USE FILE /data/my_file1.h5

# use (i.e. open) two HDF files named "my_file2.h5" and "my_file3.h5" with read only
permissions (both files are located in the current working directory)
USE READONLY FILE my_file2.h5, my_file3.h5

# use (i.e. open) an HDF file named "my_file4.h5" located in the current working
directory with cache slots, size and preemption values of 1523, 262144 and 0.6
respectively
USE FILE my_file4.h5 CACHE SLOTS 1523 SIZE 262144 PREEMPTION 0.6
```

6.7.3 USE GROUP

Syntax

USE GROUP *group_name*

Description

Use (i.e. open) an HDF group named *group_name* for subsequent operations. This will change the current working group to *group_name* thus avoiding the need to explicitly provide the full path of this group when working within it (i.e. subsequent operations are done relatively to this group, unless otherwise specified). If *group_name* was not found or could not be opened (due to unknown/unexpected reasons), an error is raised. Upon using (i.e. opening) an HDF file, the group currently in use is "/" (i.e. the root of the HDF file). Besides the name of the group to be used for subsequent operations, *group_name* may be composed of special tokens (that are not part of the name of the group itself). These are:

- "/" to separate multiple groups. Example: "*USE GROUP my_group/my_subgroup/my_subsubgroup*".
- "." to refer to the group currently in use. Example: "*USE GROUP .*".
- ".." to go up one level from the group currently in use. Example: "*USE GROUP ..*".

Parameter(s)

group_name – name of the HDF group to use (i.e. open) for subsequent operations.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# set group currently in use to "/" (i.e. the root of the HDF file)
USE GROUP /

# create two HDF groups named "my_group0" and "my_group1" (both groups will be created in group "/")
```

```
CREATE GROUP my_group0, my_group1

# create an HDF dataset named "my_dataset0" of type double (it will be created in group "/")
CREATE DATASET my_dataset0 AS DOUBLE

# set group currently in use to "my_group0" (more precisely "/my_group0")
USE GROUP my_group0

# create an HDF dataset named "my_dataset1" of type double (it will be created in group "/my_group0")
CREATE DATASET my_dataset1 AS DOUBLE

# create an HDF group named "my_subgroup0" (it will be created in group "/my_group0")
CREATE GROUP my_subgroup0

# create an HDF dataset named "my_dataset2" of type variable-length double (it will be created in group "/my_group0/my_subgroup0")
CREATE DATASET my_subgroup0/my_dataset2 AS VARDOUBLE

# create an HDF attribute named "my_attribute0" of type float (it will be created in group "/")
CREATE ATTRIBUTE ../my_attribute0 AS FLOAT

# set group currently in use to "my_subgroup0" (more precisely "/my_group0/my_subgroup0")
USE GROUP my_subgroup0

# create an HDF attribute named "my_attribute1" of type char (it will be created in group "/my_group1")
CREATE ATTRIBUTE ../../my_group1/my_attribute1 AS CHAR

# create an HDF attribute named "my_attribute2" of type variable-length char (it will be created in group "/")
CREATE ATTRIBUTE /my_attribute2 AS VARCHAR

# set group currently in use to "." (the group currently in use will not change as "." refers to the current working group itself)
USE GROUP .

# create an HDF attribute named "my_attribute3" of type int (it will be created in group "/my_group0/my_subgroup0")
```

```
CREATE ATTRIBUTE my_attribute3 AS INT

# set group currently in use one level up (should be /)
USE GROUP ..

# create an HDF attribute named "my_attribute4" of type short (it will be created in
group "/my_group0")
CREATE ATTRIBUTE my_attribute4 AS SMALLINT
```

6.7.4 FLUSH [GLOBAL | LOCAL]

Syntax

FLUSH [**GLOBAL** | **LOCAL**]

Description

Flush the entire virtual HDF file (global) or the specific HDF file (local) currently in use. All buffered data will be written into the disk. If neither the keyword GLOBAL nor LOCAL is specified, a global flush is performed by default (optionally, the keyword GLOBAL may be specified to make the purpose of this operation clearer). To perform a local flush, the keyword LOCAL must be specified. If no file is currently used, no flush is performed.

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.7.5 CLOSE FILE

Syntax

CLOSE FILE [*file_name*]

Description

Close the HDF file currently in use. Multiple files can be closed at once by separating these with a comma (,). If *file_name* is not in use (i.e. open) or it is not possible to close it (due to unknown/unexpected reasons, no subsequent files are closed, and an error is raised. Before closing a file, all buffered data will be written into it. After closing a file, the file in use will be the one most recently used before the closed file. If *file_name* is specified, it will be closed regardless of whether it is the file currently in use or not. The *file_name* must match exactly the name of the file when it was opened (otherwise no file will be closed).

Parameter(s)

file_name – name of the HDF file to close. Multiple files are separated with a comma (,).

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.7.6 CLOSE ALL FILE

Syntax

CLOSE ALL FILE

Description

Close all HDF files in use. All buffered data will be written into the respective files before closing them. If it is not possible to close a file (due to unknown/unexpected reasons), no subsequent files are closed, and an error is raised.

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.7.7 CLOSE GROUP

Syntax

CLOSE GROUP

Description

Close the HDF group currently in use. After closing it, the group currently in use will be "/" (i.e. the root of the HDF file). If no file is currently used, no group is closed.

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
// TO BE DEFINED
```

6.7.8 SET [FILE | DATASET] CACHE

Syntax

SET [**FILE** | **DATASET**] **CACHE** [**SLOTS** {*slots_value* | **DEFAULT** | **FILE**}] [**SIZE** {*size_value* | **DEFAULT** | **FILE**}] [**PREEMPTION** {*preemption_value* | **DEFAULT** | **FILE**}]

Description

Set cache parameters to default or specific values for accessing HDF files or datasets. All files or datasets that are subsequently opened or accessed (through the operations USE FILE or SELECT respectively) will use the default values defined by the HDF5 API or user-defined cache parameter values. These cache parameters are:

- Slots – number of chunk slots in the raw data chunk cache of the file or dataset. Due to the hashing strategy, its value should ideally be a prime number. When the keyword DEFAULT is specified, its value is 521 (i.e. default value defined by the HDF5 API). When the keyword FILE is specified, its value will be as defined in the file cache slots parameter.

- Size – total size of the raw data chunk cache in bytes for the file or dataset. When the keyword DEFAULT is specified, its value is 1048576 (i.e. 1 MB – default value defined by the HDF5 API). When the keyword FILE is specified, its value will be as defined in the file cache size parameter.

- Preemption – chunk preemption policy. Its value must be between 0 and 1 inclusive. It indicates the weighting according to which chunks which have been fully read or written are penalized when determining which chunks to flush from cache. When the keyword DEFAULT is specified, its value is 0.75

(i.e. default value defined by the HDF5 API). When the keyword FILE is specified, its value will be as defined in the file cache preemption parameter.

In case neither the keyword FILE nor DATASET is specified, the setting of the cache parameters refers to files by default (optionally, the keyword FILE may be specified to make the purpose of this operation clearer). To explicitly set the cache parameters to datasets, the keyword DATASET must be specified.

Parameter(s)

slots_value – to be defined.

size_value – to be defined.

preemption_value – to be defined.

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# use (i.e. open) an HDF file named "my_file0.h5" with cache slots, size and preemption values of 521, 1048576 and 0.75 respectively (these are the default values defined by the HDF5 API)
USE FILE my_file0.h5

# set cache slots and preemption values to 2297 and 0.9 respectively (the cache size value remains intact) for subsequent usage (i.e. opening) of HDF files
SET CACHE SLOTS 2297 PREEMPTION 0.9

# use (i.e. open) an HDF file named "my_file1.h5" with cache slots, size and preemption values of 2297, 1048576 and 0.9 respectively
USE FILE my_file1.h5

# set cache slots, size and preemption values to 1523, 262144 and 0.6 respectively for subsequent usage (i.e. opening) of HDF files
SET FILE CACHE SLOTS 1523 SIZE 262144 PREEMPTION 0.6

# use (i.e. open) an HDF file named "my_file2.h5" with cache slots, size and preemption
```

```
values of 1523, 262144 and 0.6 respectively
USE FILE my_file2.h5

# set cache size value to 1048576 (default value defined by the HDF5 API) and preemption
value to 0.4 (the cache slots value remains intact) for subsequent usage (i.e. opening)
of HDF files
SET FILE CACHE SIZE DEFAULT PREEMPTION 0.4

# use (i.e. open) an HDF file named "my_file3.h5" with cache slots, size and preemption
values of 1523, 1048576 and 0.4 respectively
USE FILE my_file3.h5
```

```
# select (i.e. read) an HDF dataset named "my_dataset0" with cache slots, size and
preemption values of 521, 1048576 and 0.75 respectively (these are the default values
defined by the HDF5 API)
SELECT FROM my_dataset0

# set cache slots and preemption values to 2297 and 0.9 respectively (the cache size
value remains intact) for subsequent selection (i.e. reading) of HDF datasets
SET DATASET CACHE SLOTS 2297 PREEMPTION 0.9

# select (i.e. read) an HDF dataset named "my_dataset1" with cache slots, size and
preemption values of 2297, 1048576 and 0.9 respectively
SELECT FROM my_dataset1

# set cache slots, size and preemption values to 1523, 262144 and 0.6 respectively for
subsequent selection (i.e. reading) of HDF datasets
SET DATASET CACHE SLOTS 1523 SIZE 262144 PREEMPTION 0.6

# select (i.e. read) an HDF dataset named "my_dataset2" with cache slots, size and
preemption values of 1523, 262144 and 0.6 respectively
SELECT FROM my_dataset2

# set cache size value to 1048576 (default value defined by the HDF5 API) and preemption
value to 0.4 (the cache slots value remains intact) for subsequent selection (i.e.
reading) of HDF datasets
SET DATASET CACHE SIZE DEFAULT PREEMPTION 0.4

# select (i.e. read) an HDF dataset named "my_dataset3" with cache slots, size and
```

```
preemption values of 1523, 1048576 and 0.4 respectively
SELECT FROM my_dataset3

# set cache slots, size and preemption values to 3089, 2048 and 0.85 respectively for
subsequent usage (i.e. opening) of HDF files
SET FILE CACHE SLOTS 3089 SIZE 2048 PREEMPTION 0.85

# set cache slots value to 521 (default value defined by the HDF5 API), size value to
1024, and preemption value to 0.85 (defined by the cache preemption value for HDF files)
for subsequent selection (i.e. reading) of HDF datasets
SET DATASET CACHE SLOTS DEFAULT SIZE 1024 PREEMPTION FILE

# select (i.e. read) an HDF dataset named "my_dataset4" with cache slots, size and
preemption values of 521, 1024 and 0.85 respectively
SELECT FROM my_dataset4
```

6.7.9 ENABLE FLUSH [GLOBAL | LOCAL]

Syntax

ENABLE FLUSH [**GLOBAL** | **LOCAL**]

Description

Enable automatic flushing of the entire virtual HDF file (global) or only the HDF file (local) currently in use. Automatic flushing (i.e. all buffered data is written into the disk) will subsequently occur whenever an operation modifying the file is executed. If neither the keyword GLOBAL nor LOCAL is specified, automatic global flushing is set by default (optionally, the keyword GLOBAL may be specified to make the purpose of this operation clearer). To set automatic local flushing, the keyword LOCAL must be specified.

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# enable automatic flushing of the entire virtual HDF file (global) currently in use
ENABLE FLUSH

# enable automatic flushing of the entire virtual HDF file (global) currently in use
ENABLE FLUSH GLOBAL

# enable automatic flushing of only the HDF file (local) currently in use
ENABLE FLUSH LOCAL
```

6.7.10 ENABLE DEBUG

Syntax

ENABLE DEBUG

Description

Enable debug mechanism (i.e. info/debug messages will be displayed when executing operations). This operation should help the programmer have a better understanding of the parameters HDFql is receiving, the operation performed, and the return value of this operation. Additionally, info/debug messages of the HDF5 API itself are displayed in case of an error.

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# enable debug mechanism (i.e. info/debug messages will be displayed when executing
operations)
```

```
ENABLE DEBUG
```

6.7.11 DISABLE FLUSH

Syntax

DISABLE FLUSH

Description

Disable automatic flushing of the entire virtual HDF file (global) or only the HDF file (local) currently in use.

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# disable automatic flushing of the entire virtual HDF file (global) or only the HDF file
(local) currently in use
DISABLE FLUSH
```

6.7.12 DISABLE DEBUG

Syntax

DISABLE DEBUG

Description

Disable debug mechanism (i.e. info/debug messages will not be displayed when executing operations).

Parameter(s)

None

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# disable debug mechanism (i.e. info/debug messages will not be displayed when executing operations)
DISABLE DEBUG
```

6.7.13 RUN

Syntax

RUN *command1, ..., commandX*

Description

Run (i.e. execute) an external command named *command*. Multiple commands can be run at once by separating these with a comma (,). If *command* was not found or it was not possible to run (due to unknown/unexpected reasons), no subsequent commands are run, and an error is raised. If *command* has parameters, both the *command* and parameters should be surrounded by double-quotes (“).

Parameter(s)

command – name of an external command to run (i.e. execute). Multiple external commands are separated with a comma (,).

Return

The cursor in use remains unchanged after executing the operation (and independently of the success or failure of this operation). Please refer to the chapter CURSOR for additional information.

Example(s)

```
# run notepad text editor (if "notepad.exe" was not found, an error is raised)
RUN notepad.exe

# run firefox and open HDFql website (if "firefox" was not found, an error is raised)
RUN "firefox http://www.hdfql.com"

# run notepad text editor to edit file "my_file.html" and afterwards (i.e. after closing notepad) open "my_file.html" by running chrome
RUN "notepad.exe my_file.html", "chrome my_file.html"
```

GLOSSARY

Application programming interface (API)

An application programming interface (API) specifies how software components should interact with each other. In practice, an API comes in the form of a library that includes specifications for functions, data structures, object classes, constants and variables. A good API makes it easier to develop a program by providing all the building blocks.

Attribute

An (HDF) attribute is a metadata object describing the nature and/or intended usage of a primary data object. A primary data object may be a group, dataset or committed datatype. Attributes are assumed to be very small as data objects go, so storing them as standard (HDF) datasets would be inefficient.

Cursor

A cursor is a control structure that is used to iterate through the results returned by a query (that was previously executed). It can be seen as an effective means to abstract the programmer from low-level implementation details of accessing data stored in specific structures. In HDFql, cursors offer several ways to traverse result sets according to specific needs and they also store result sets returned returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations.

Dataset

A (HDF) dataset is an object composed of a collection of data elements and metadata that stores a description of the data elements, data layout and all other information necessary to write and read the data. A dataset may be a multidimensional array of data elements and it may have zero or more attributes.

Datatype

A datatype is a classification identifying one of various types of data such as integer, real or string, which determines the possible values for that type, the operations that can be done on values of that type, the meaning of the data, and the way values of that type can be stored. In other words, a datatype is a classification of data that tells HDFql how the user intends to use it.

Group

A (HDF) group is a container structure which can hold zero or more objects (i.e. datasets and other groups). Every object must be a member of at least one group, except the root group, which (as the sole exception) may not belong to any group.

Post-processing

Post-processing options enable processing (i.e. transformation) results of a query according to the programmer's needs such as ordering or redirecting. These options are optional and may be used to create a (linear) pipeline to further process result sets returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations.

Result set

A result set stores the results returned by DATA QUERY LANGUAGE (DQL) and DATA INTROSPECTION LANGUAGE (DIL) operations.

Result subset

A result subset stores the results returned by a DATA INTROSPECTION LANGUAGE (DIL) operation that was performed on a dataset or attribute of type variable-length.

Subcursor

A subcursor is meant to complement (i.e. help) cursors in the task of storing data of type variable-length (i.e. VARTINYINT, UNSIGNED VARTINYINT, VARSMALLINT, UNSIGNED VARSMALLINT, VARINT, UNSIGNED VARINT, VARBIGINT, UNSIGNED VARBIGINT, VARFLOAT, VARDOUBLE and VARCHAR). In practice, when a dataset or attribute of type variable-length is read through a DATA QUERY LANGUAGE (DQL) operation, only the first value of the variable data is stored in the cursor (as expected), while all values of the variable data are stored in the subcursor. In other words, each position of the cursor stores the first value of the variable data and also points to a subcursor that in turn stores all the values of the variable data. Similar to cursors, HDFql subcursors offer several ways to traverse result subsets.