DataHarvester - an abstraction layer for accessing scientific data from various sources

W. Waltenberger, G. Richter and H. V. Riedel
Austrian Academy of Sciences, Institute of High Energy Physics, A-1050 Vienna, Austria, EU
E-mail: walten@hephy.oeaw.ac.at

Abstract. A tool is presented that is capable of reading from, writing to and converting between various sources. Currently supported file formats are ROOT, HBOOK, HDF, XML, SQLITE and a few text file formats. A plugin mechanism decouples the file-format specific "backends" from the main library. All data are internally represented as "heterogeneous hierarchic tuples"; no other data structure exists in the DataHarvester.

1. Motivation
One frequent task in many scientific activities is the collection of scientific data in tuples. The ROOT file format [1] is the major player in the high energy physics community, having replaced the older HBOOK [2] file format. Still, the usage of other general-purpose persistency solutions such as HDF [3] or AIDA [4] is being contemplated in some HEP experiments.

The DataHarvester project started as a very simple frontend to ROOT persistency, that was meant to hide from the user some of ROOT’s idiosyncracies. Later it was realized by one of the authors that ROOT could indeed be replaced by another persistency solution, like HDF. Indeed, the technology that is used “behind the scenes” could be made pluggable. In fact, persistency solutions are all about reading/writing tuple-like data from/to a storage medium. Later still more “backends” like XML and SQLITE were added, and Python was supported via the SWIG [5] interface generator. What remains from the early days, though, is the committment to provide a package that is as easy as possible to use, with performance issues being almost completely neglected within the code. It should thus explicitly not be used for large-scale data production and analysis; rather it is envisaged to be a simple and user-friendly tool for debugging and small-scale analyses which interfaces nicely with many file formats and analysis tools.

2. Design goals
This is a list of design goals that has been defined in the early stages of the project:

- Simple tasks must be simple: only a few simple lines of code should have to be written for storing e.g. two floating point variables in a tuple.
- Complicated tasks must be achievable without too much effort.
- User code should be “agnostic” with respect to the persistency technology: changing from storing data into ROOT files to storing into AIDA files should not impose non-trivial changes in the user code.
• The user should have to handle as few objects as possible: it is unnecessary for the user to be exposed to e.g. file handles. The same argument holds for tuple objects.
• The user should not have to supply redundant information: supplying a “float” variable for a column of a tuple implies that the type of that column be “float”. It should not be necessary to state this explicitly.
• Implicit specification of the data must be sufficient. In the case that the user “breaks” his/her own implicit specifications, warnings or error messages must be issued.
• Small changes in the information content of a tuple, such as adding a column should trigger changes in only one place in the code.
• It should be impossible or as difficult as possible for the user to introduce inconsistencies.
• It is desirable to be able to document what content is stored in what column in what tuple (meta information). The meta information should be stored together with the information.
• Filling a tuple should be possible anywhere in the user code and not be limited to one single source file.
• Producing tuples should be non-intrusive. Debugging an algorithm with the DataHarvester should not imply major changes in the algorithmic code.

3. Sample user code
In the following we show two examples of how to use the DataHarvester.

The following listing shows Python code that reads from an XML file:

```python
import DataHarvester
r = DataHarvester.Reader_file("bla.xml")
for tuple in r:
    for row in tuple:
        for data in row:
            print data, "=", row[data]
```

Here is a sample C++ program that writes into two files at once:

```cpp
#include <DataHarvester/Writer.h>

int main()
{
    using namespace DataHarvester;

    Tuple t("event");
    t["id"] = 1; // event id = 1

    t["track:name"] = "track"; // add first track
    t["track:E"] = 3.14;
    t.fill("track");

    t["track:name"] = "track"; // add second track
    t["track:E"] = 1.41;
    t.fill("track");

    t.fill(); // fill event

    // stream into my.txt, as well as my.hdf
    Writer::file("my.txt:my.hdf") << t;

    Writer::close(); // close all open handles
}
```
4. Behind the scenes
What happens internally when the user passes information to the DataHarvester? Fig. 1 shows a sequence diagram that illustrates the dynamic loading of the technology-specific backend. It can also be seen how files are created implicitly, not explicitly, in the DataHarvester.

![Sequence diagram of the DataHarvester in action.](image)

**Figure 1.** Sequence diagram of the DataHarvester in action.

**C++ and weak typing**
When using the DataHarvester’s C++ interface, the data type is never defined explicitly. The DataHarvester “knows” what kind of data he is given, and passes the data along with the type information to the backends.

This is achieved via massive operator overloading of a class called MultiType, see Fig. 2.

```
MultiType
| MultiType(double/string/int/bool) |
| operator=(double/string/int/bool); MultiType |
| operator=(double/string/int/bool); bool const |
| operator=(const char *); bool const |
| operator=(const char *); bool const |
| isADouble/isAString/isAInt/isABool(); bool const |
| operator double(); double |
| operator string(); string |
| operator int(); int |
| operator bool(); bool |
| isADouble(); double |
| isAString(); string |
| isAInt(); int |
| isABool(); bool |
| isAScalar(); string const |
```

**Figure 2.** A MultiType represents a basic “scalar” which knows of its type (int, float, string, etc.)
5. Command line interface
The DataHarvester package contains two command line utilities: one (DataHarvester-converter) for conversion between different file formats, the other (DataHarvester-systemwriter) for writing “system information” into a file of any type that is supported by the DataHarvester.

6. Availability
The DataHarvester is hosted at HepForge [6]:
http://projects.hepforge.org/dataharvester
This site contains the source code, finalised packages and documentation.
So far the code has been tested on Linux only. Porting it to other Unix platforms will be done on demand only; Windows is currently not a target platform.

7. Summary
A tool has been presented that intends to bridge the gaps between different file formats, analysis tools, and programming languages. Ease of use has been the predominant design goal; since it adds a layer to existing persistency technology, optimal performance will not and cannot be achieved; rather it could be used as a tool for analysing which persistency solution performs well for a user's specific needs.

8. Outlook
Apart from consolidation of the existing code (particularly the technology-specific backends), here is a list of desirable new features:

- a backend for OpenOffice spreadsheets;
- a backend for various types of SQL/ODBC databases;
- support for more programming languages (e.g. Java, Ruby);
- automatic mapping of objects onto hierarchic tuples (e.g. via gccxml);
- support for generic data filters.

Acknowledgements
Thanks are due to Winni Mitaroff for useful discussions and careful reading of the manuscript.

References
[1] ROOT. http://root.cern.ch.
[2] HBOOK. http://http://wwwinfo.cern.ch/asdoc/hbook_html3/hboomain.html.
[3] HDF5. http://www.hdfgroup.org.
[4] Academic Software Organization. AIDA, Abstract Interfaces for Data Analysis. http://http://aida.freehep.org/.
[5] SWIG – Simplified Wrapper and Interface Generator. http://www.swig.org.
[6] CEDAR HepForge. http://www.hepforge.org.