>>EBF<<

An efficient and easy to use binary file format

Sanjib Sharma
Sydney Institute for Astronomy
University Of Sydney

www.ebfformat.sourceforge.net
pip install ebfpy (python2.7)
https://github.com/sanjibs/ebfpy (python2+python3)

Python, C, C++, Fortran, Java, Matlab, IDL

(coming soon in node.js)

Documentation for EBF file format and Sotware Utilities

>>EBF<<

Table Of Contents

Next topic

Introduction

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Contents:

Motivation

- **Why do we need a format?**
  - Otherwise only the program that wrote the data can read it. Or custom reading routine for each data
  - Difficult to share data with others.

- **Should be binary and not ascii?**
  - 100 times slower
  - Write 700 (6) MB/s, Read 1800 (18) MB/s

Problems with binary data.

- Data type information needs to be specified
  - 010111001110
  - Binary data without specified data type is just 0 and 1.

- Not portable due to Endianness (little vs big)
  - In a multi byte word the most significant byte is to the left (big) or right (little). Intel vs IBM processors.
  - 0x 12345678

BE: 12	34	56	78
LE: 78	56	34	12

- **Should write multiple items in same file and have support for random read access.**
  - Helps to organize data in one place
  - If not random access then the exact sequence in which the data was written need to be known.
  - New features cannot be easily introduced.

Code

```
int64 glon[100];
int64 glat[100];
int64 pselect[100,100];
N=1000;
float64 ra[N], dec[N], feh[N], age[N];
```

ra=read(“check”,”/table/ra”)

File (“check”)

“/pselect”
“/glon”
“/glat”
“/table/ra”
“/table/dec”
“/table/age”
“/table/feh”

Why not use HDF5 or FITS?

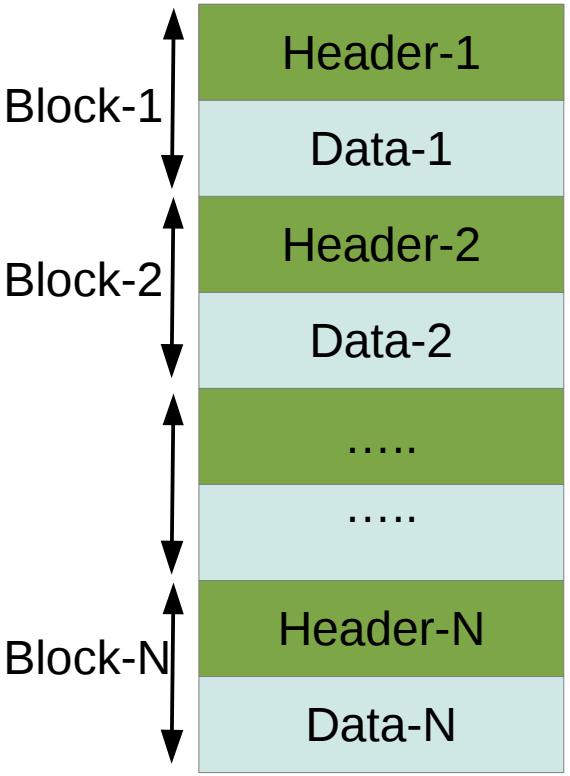
- FITS
  - does not support multiple items tagged by names.
  - Sequential access too slow for large number of items
- HDF5
  - a complicated format. (460 functions)
  - API not user friendly. Steep learning curve.
  - Main API only C. In other languages one has to rely on foreign language interface to call the C routines.
  - Not fully type safe. Errors not detected at compile time
  - (…,...,...,...,HDF5_NATIVE_INT,x)
- Writing lot of small items requires too much memory.
  - Per item: 4KB for FITS and 2KB for HDF

EBF design goals

- Binary format for speed
- Multiple items with random access
- Automatic type and endian conversion.
- Support for attributes and data units.

- Ease of use
- Design APIs such that it is harder to make mistakes, and when you do it will give a compilation error.
- Support for multiple programming languages.
  - No use of foreign language interface.
  - Pure code in all languages.

The Format



Header

char	Signature[8]
char	version[4]
int32	endian_test =1684234849 (abcd)
int32	header_size
int32	name_size
int32	data_type
int32	data_size
int32	rank
int32	unit_size
int32	sdef_size
int64	dim[rank]
char	name[name_size]
char	unit[unit_size]
char	sdef[sdef_size]
char	extra[64]

44 bytes

Defining structures

Nested (recursive structures allowed)

Field name:
“sdef”

Field:
“struct
{
    float32 density;
    float64 mass;
    int32   metals 3 2;
    struct   {
    float32 pos 3;
    float32 vel 3;
    }point 1;
}”

- Only idl and python
- Byte alignment issues make it less portable for static languages like C/C++
- Preferably split and write each field as separate arrays.

Supported Data types

Data Type		Integer Code
undefined		0
char		1
int32	(int)	2
int64	(long)	3
float32	(float)	4
float64	(double)	5
int16	(short)	6
structure		8
int8	(unsigned char)	9
uint8	(signed char)	10
uint16	(unsigned short)	11
uint32	(unsigned int)	12
uint64	(unsigned long)	13

Note in **Java** no unsigned int, ebf routines do auto conversion.

API for Dynamic languages: Python,IDL,Matlab

- `import ebf`
- `x=np.arange(10)`
- `ebf.write("check.ebf","/x",x,"w")`
- `x=ebf.read("check.ebf","/x")`

- `d={'x1':x,'x2':x}`
- `ebf.write("check.ebf","/mydata/",d,"a")  # "/mydata/x1", "/mydata/x2"`
- `data=ebf.read("check.ebf","/mydata/")`
  - *Only objects in current path*
  - `data["x1"], data["x2"]`

- `data=ebf.read("check.ebf","/mydata1/",recon=1)`
  - *All objects recursively in current path*
  - `data["x1"]`
  - `data["x1_attributes"]["mass"]`

Python: iterate in chunks over big data

```
import ebf
x=np.arange(10000000)
d={'x1':x,'x2':x,'x3':x}
ebf.write("check.ebf","/",d,"a")


temp=0
for x3 in ebf.iterate("check.ebf","/x3",1000):
     temp=temp+np.sum(x3)
print(temp)


temp=0
for data in ebf.iterate("check.ebf","/x1+",1000):
     temp=temp+np.sum(data['x2'])+np.sum(data['x3'])
print(temp)
```

API C++

- `double x[100];`
- `ebf::Write("check.ebf","/x1",&x[0],"w","100 km/s",100);`
  - Auto type inference
- `ebf::WriteAs<int>("check.ebf","/x2",&x[0],"a","100,km/s",10,10);`
  - Auto type conversion
- `vector<float> y;`
- `ebf::Read("check.ebf","/x",y);`
  - Auto resize and type conversion

API C

- `double x[100];`
- **`Ebf_WriteFloat64`**`("check.ebf","/x1",&x[0],"w","100 km/s",100);`
  - Type Safe , will give compilation error for wrong types

- `Float *y;`
- `EbfDataInfo dinfo=Ebf_GetDataInfo("check.ebf","/x1");`
- `y=(float *)malloc(dinfo.elements*8)`
- **`Ebf_ReadFloat32`**`("check.ebf","/x",&y[0],dinfo.elements);`
  - Auto type conversion and Type Safe.

API C++/C/Java/Fortran

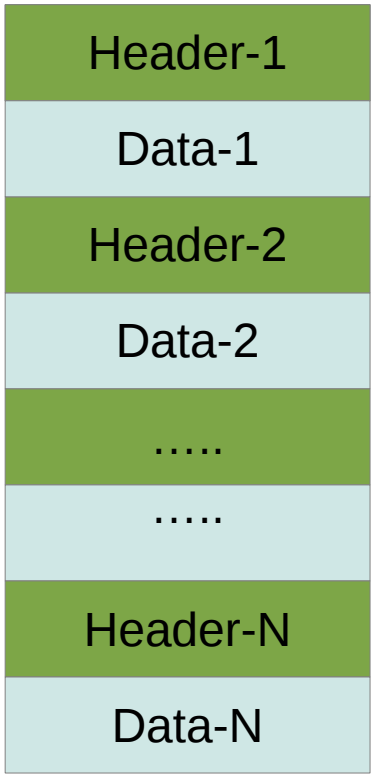
```
Efile efile;
efile.Open("check.ebf","/x","w",Ebf_type("int32"),"km/s");
efile.Write(&x1[0]);     // float
efile.Write(&x2[1],10);  // double
Efile.Close();

efile.Open("check.ebf","/x");
Efile.Seek(20)          // move to 20th elements
efile.Read(&y1[0]);     // float
efile.Read(&y2[1],10); // double
Efile.Close();
```

Iterating without loading the full data (C++ only)

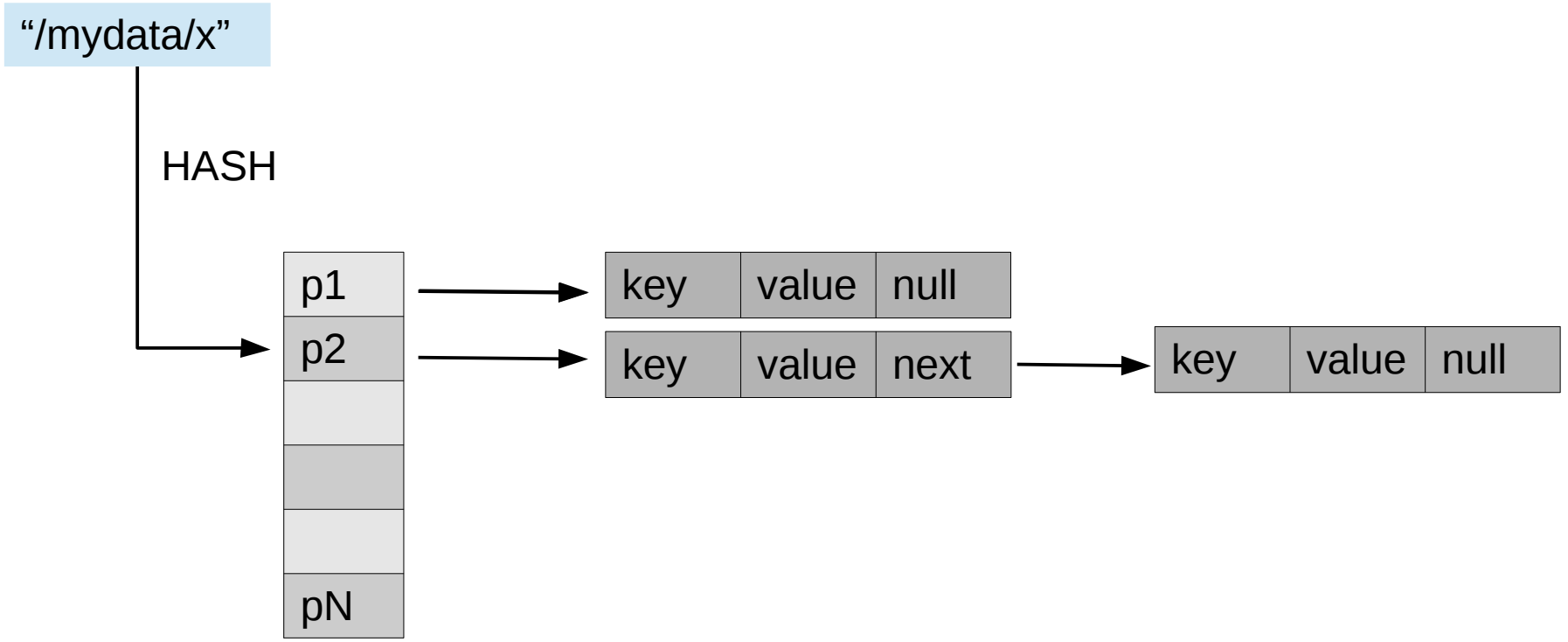
- `ebf::EbfVector<float> x("check.ebf","/x");`
- `x[i];`
- `x(i,j);  // multidimensional index`
- `x(i,j,k);  // multidimensional index`
- Only 1000 items loaded at a time, full data never loaded.
- Useful for traversing Big Data sets with a small amount of RAM.

Ebf Internals



- Sequential access is *O(N)*
- **Hash map** *O(1)*
  - *hashmap[dataname]=location*
- e.g., large number of data objects
  - 100x100 grid (in age and metallicity) of isochrone tables
  - “feh0.1_age2.0”, “feh0.1_age2.0”

Hashmap “/.ebf/htable”



Where to put the hashtable?

- Putting in beginning: not a good idea.
  - Have to choose some size, which better be small.
  - Cannot change the size if it is full.

- Solution: adaptive
  - **Put** in the beginning location of hastable './ebf/info' (int64 x[5])
  - **Put** hashtable of fixed size after that './ebf/htable'
  - **If** (the hashtable is full)
    - **Rename** the old table
    - **Add a new table** at the end, of *double* the previous size.
    - **Update** './ebf/info' to reflect the new location

Reserved paths

- /.ebf/info
- /.ebf/htable
- /.tr/

```
sharma@tau:~/ws/wscpp/ebfpy3$ ebftk check_table.ebf
check_table.ebf 5039 bytes
-------------------------------------------------------------------
name                            dtype    endian  unit        dim
-------------------------------------------------------------------
/.ebf/info                      int64    little              [5]
/.ebf/htable                    int8     little              [1256]
/x5                             int64    little              [128]
/.tr/x2.0                       int64    little              [128]
/x3                             int64    little              [128]
sharma@tau:~/ws/wscpp/ebfpy3$
```

Remove or restore items, '/.tr/'

- ebf.rename('check.ebf','/x','')

  '/x' → '/.tr/x.0'

- Write and remove again

  '/x' → '/.tr/x.1'

The ebf toolkit *ebftk*

```
sharma@tau:~/ws/wscpp/ebfpy3$ ebftk check_table.ebf
check_table.ebf 5039 bytes
-------------------------------------------------------------------
name                            dtype    endian  unit       dim
-------------------------------------------------------------------
/.ebf/info                      int64    little             [5]
/.ebf/htable                    int8     little             [1256]
/x5                             int64    little             [128]
/.tr/x2.0                       int64    little             [128]
/x3                             int64    little             [128]
sharma@tau:~/ws/wscpp/ebfpy3$
```

The ebf toolkit *ebftk*

```
sharma@tau:~/ws/wscpp/ebfpy3$ ebftk -stat check_table.ebf "/x5 /x3"
name                 items           min           max          mean        stddev
/x5                    128  -65763.0000  -65636.0000  -65699.5000       36.9493
/x3                    128  -65763.0000  -65636.0000  -65699.5000       36.9493
sharma@tau:~/ws/wscpp/ebfpy3$
```

The ebf toolkit *ebftk*

$ebftk –help

NAME:

ebftk - a toolkit for EBF files (version 0.0.20)

USAGE:

```
ebftk    -diff  file1 file2
ebftk    -list filename
ebftk    -stat filename "TagName1 TagName2 .."
ebftk    -copy src_file dest_file
ebftk    -copy src_file dest_file TagName
ebftk    -cat filename "TagName1 TagName2 .."
ebftk    -csv filename "TagName1 TagName2 .."
```

Attributes and data units

- Unlike HDF or FITS, no special interface for attributes, just write like other data items.
  - “/data”,
  - “/data_attributes/attr1”, “/data_attributes/attr2”

- Units are not attributes they are part of definition of data.
  - Attributes can also have units
  - “/density” (100,100), kg/m^2
  - “/density_attributes/x” (100,), m
  - “/density_attributes/y” (100,), m

Performance

(1000 data items of size 4 bytes, array of 10^7 float)

Language	Item write	Item read	Data write	Data read
	KOP/s	KOP/s	MB/s	MB/s
C/C++ EBF	9	23	775	1800
C/C++ HDF5	1.5	1.5	775	1800
C/C++ FITS	0.2	0.5	344	502
C/C++ ASCII			5.6	18
Fortran90/2003	6.0	8.3	950	1120
Java	2.3	7.4	270	727
Python EBF	1.72	1.07	466	620
Python HDF5	0.95	1.0	659	1030
Python FITS	0.74	0.0012	427	1047
IDL EBF	2.7	2.6	113	772
IDL HDF5	5.0	7.4	110	94
IDL FITS	2.7	0.007	80	360
Matlab EBF	0.26	0.26	680	1175
Matlab HDF5	0.26	0.86	1000	1030
Matlab FITS		0.0004		78

Summary

- Easier to use and at the same time performance at par with HDF.
  - Type safe, auto data type conversion
- More finer control of reading and writing.
  - Skip, tree walks.
- Available at
  - www.ebfformat.sourceforge.net
  - https://github.com/sanjibs/ebfpy (python2 + python3)
    - pip install ebfpy (for python 2.7)
  - sanjib.sharma@gmail.com

Conclusions

- Size of items cannot be expaned. Could be supported in future.
- No support for hyperslab selection
  - HDF5 can do both of above, as it uses B-trees
- Easier to use and at the same time performance at par with HDF.
- Available at
  - https://github.com/sanjibs/ebfpy
  - sanjib.sharma@gmail.com

>>EBF<<

An efficient and easy to use binary file format

- Why do we need a format?
- Why binary and not ascii?
  - Write 700 (6) MB/s, Read 1800 (18) MB/s
- Why do we need to write multiple items?
- What are the problems with binary data?
- Why not HDF or FITS?
- Ebf Design goals
- Ebf Internals: blocks
  - Header, Structures, Supported Datatypes
- EBF API C,C++,Java,Fortran (statically typed languages)
  - EbfVector
- EBF API Python,IDL,Matlab (dynamically typed languages)
  - iterator
- Ebf Internals:
  - Hashtable, reserved paths,checksum
- Ebf tool kit
- Attributes and units
- Performance
- Conclusions

Checksum

- What if file is modified external to the program?
  - e.g. running interactive session in IDL, while data being generated externally
- Time
  - Time is not reliable
  - Time is not accurate (resolution)
  - Diff wont work
- Checksum
  - A unique 64 bit number created out of
    - (dataname, location)
  - Added as the first item “`/.ebf/info`”
  - Updated each time a new item is written

/.ebf/info

- Int64 x[5]
- x[0]- checksum
  - Hash created out of string
  - s='(key_0, location_0)...(key_N, location_N')
- x[1]- location of item hash-table
- x[2]- location of hash-table skipping header
- x[3]- 1, unused
- x[4]- unused

/.ebf/htable

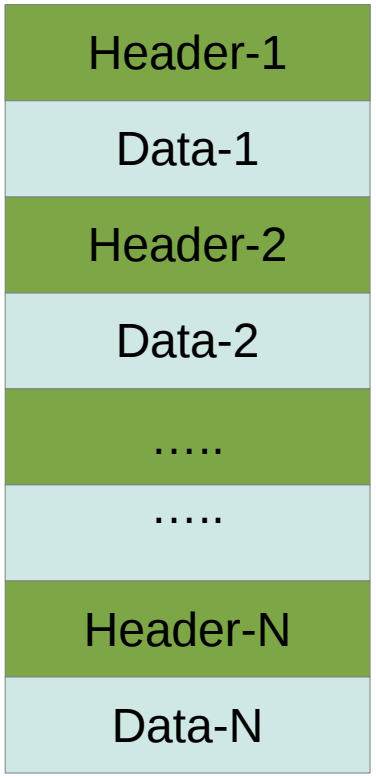
- Hash Table
  - struct ht_header;
  - int64 ht_table[htcapacity];
  - struct items[itemcapacity]
    - struct{int64 keyloc; int64 keysize; int64 value; int64 next; int64 tnext; }
  - char keys[keycapacity];
    - key_i = keys[items['keyloc'][i]: items['keyloc'][i] +items['keysize'][i]]

Ebf Internals

EbfMap *a hash map*
getLocation(filename,dataname)

EbfHeader
read
write

Efile
open
read
write
close



- Sequential access is $O(N^2)$
- Hash map *O(1)*
  - *hashmap[dataname]=location*
- e.g., large number of data objects, 100x100 grid (in age and metallicity) of isochrone tables

Motivation

- **Why do we need a format?**
  - Otherwise only the program that wrote the data can read it. Or custom reading routine for each data
  - Difficult to share data with others.

- **Should be binary and not ascii?**
  - 100 times slower
  - Write 700 (6) MB/s, Read 1800 (18) MB/s

- **Should write multiple items in same file and have random access support.**
  - Helps to organize data in one place
  - If not random access then the exact sequence in which the data was written need to be known.
    - New features cannot be easily introduced.
  - 100x100 grid (in age and metallicity) of isochrone tables