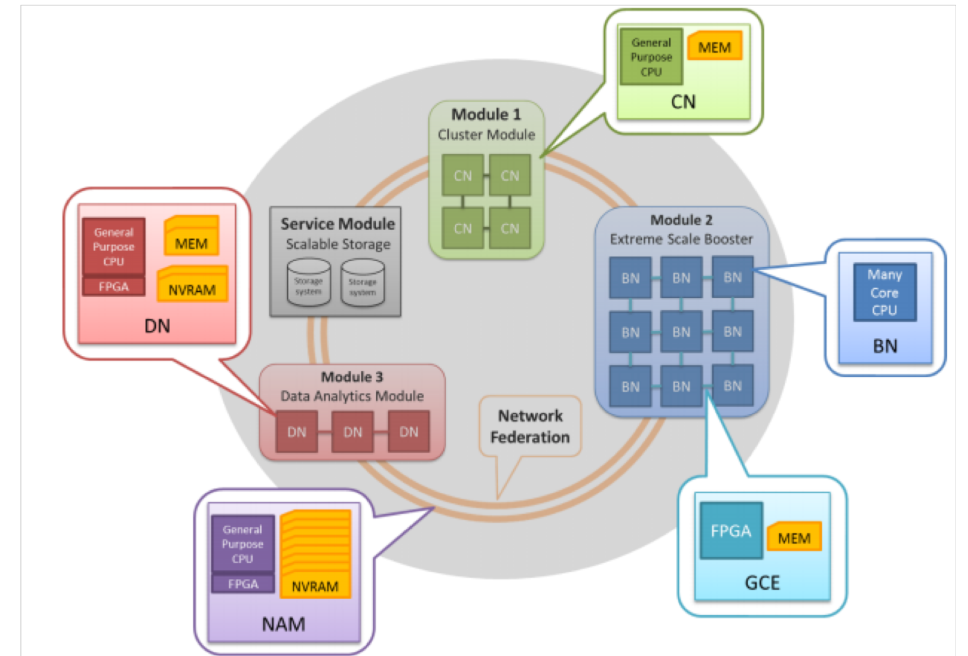
Proposal/Ideas on how to evolve ROOT Data Management Facilities

Viktor Khristenko

**I do apologize, slides are a bit convoluted. They are not meant as "presentation",
But more for technical discussion/log book.**

The DEEP-EST Project



- DEEP– Extreme Scale Technologies
- R&D for Exascale HPC
- CERN / CMS is a collaborating partner
- European Project aiming to build Modular Supercomputing Architecture. Located at Juelich Supercomputing Center (JSC)

Main topics

- ROOT I/O
- ROOT Serialization/Deserialization
- Tree, Forest

Motivation

- Heterogeneous Resources
  - My current activity revolves around porting CMS calorimeter local reco to OpenCL to use with FPGAs
  - The same was already done for GPUs (with cuda)
  - OBSTACLES: ugly to change/pack things for these archs
    - Changing layouts…………
    - A lot of data format changes /
- Forest
  - Google doc, and a question if Forest should apply to Experiments or just end-user analysis
  - In my personal opinion, very tight similarity with Apache Arrow
- Inflexibility of existing ROOT I/O + Serialization/Deser interfaces
  - In fact, they are totally collapsed right now.
  - That goes back to modularity question.
  - You do not need Interpreter to read/write ROOT Records (key + blob)/files,
  - You need Interpreter to serialize/deserialize!

End User or Experiments

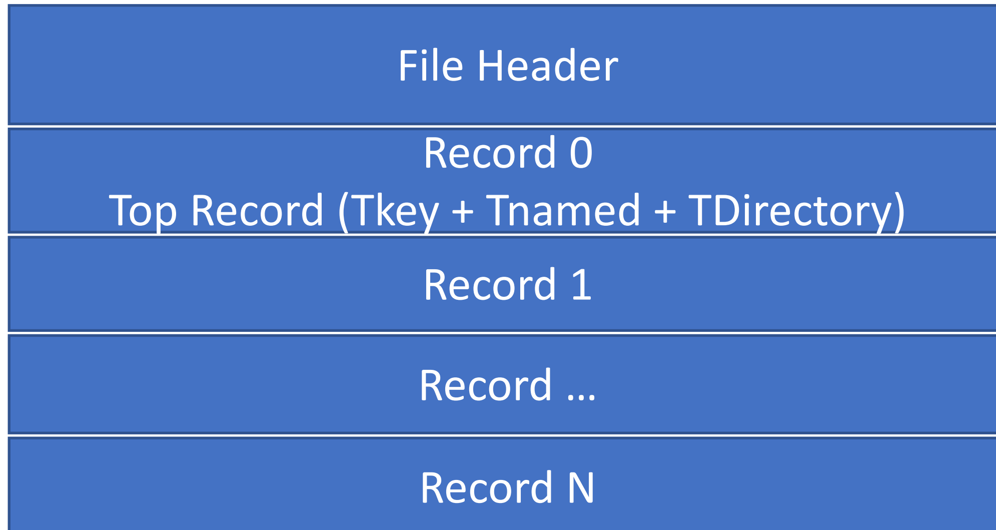
- I aim at Experiments, large scale data processing scenarios, usage of supercomputers, farms, clouds, you name it.
- The important part here is that I'm interested to see how large scale data processing can be improved by improving data handling within ROOT!
- In the end, this is one of the most common/fundamental parts of data intensive workflows – how we handle data.

ROOT I/O

Record

- **Key**
- **Blob**

Tfile rpecification



Typically, but not always at the end of Tfile:
Free Segments Record
Streamers Record
Can be at the end

Streamers Record is not at I/O
Level at all!!! It’s absolutely unnecessary
To have Streamers Record parsed for dealing with
ROOT I/O. IN fact, none of the ROOT I/O primitives are in the Streamers Record themselves!!!

_file0->Map()

```
20180518/152630  At:23301   N=51      TDirectory
20180518/152630  At:23352   N=51      TDirectory
20180518/152630  At:23403   N=716     TDirectory
20180518/152630  At:24119   N=742     KeysList
20180518/152630  At:24861   N=3039    StreamerInfo   CX =  3.04
20180518/152630  At:27900   N=83      FreeSegments
20180518/152630  At:27983   N=1       END
```

My reimplementation of root i/o in c. just example

```
At:   23352 N:    51    TDirectory CX = 1.00
At:   23403 N:   716    TDirectory CX = 1.00
At:   24119 N:   742         TFile CX = 1.00
At:   24861 N:  3039         TList CX = 3.08
At:   27900 N:    83         TFile CX = 1.00
```

ROOT I/O + Serialization / Deserialization

- Basically a single Record is read/written
- A TTree essentially has a track of all of the positions of all Records (for the data stored in TTree)
- I define the ending point of ROOT I/O to be the moment we have put a record into memory of a single node
  - Read from local disk
  - Brought in over the network
  - whatever

20160609/091740	At:607828438	N=130692	TBasket	CX = 6.04
20160609/091740	At:607959130	N=317412	TBasket	CX = 2.49
20160609/091740	At:608276542	N=202800	TBasket	CX = 3.63
20160609/091740	At:608479342	N=562351	TBasket	CX = 1.58
20160609/091740	At:609041693	N=526935	TBasket	CX = 1.69
20160609/091740	At:609568628	N=179014	TBasket	CX = 4.96
20160609/091740	At:609747642	N=272504	TBasket	CX = 3.26
20160609/091740	At:610020146	N=540469	TBasket	CX = 1.64
20160609/091740	At:610560615	N=536728	TBasket	CX = 1.66
20160609/091740	At:611097343	N=556444	TBasket	CX = 1.60
20160609/091740	At:611653787	N=558017	TBasket	CX = 1.59
20160609/091740	At:612211804	N=542522	TBasket	CX = 1.64
20160609/091740	At:612754326	N=564499	TBasket	CX = 1.57
20160609/091740	At:613318825	N=133713	TBasket	CX = 6.64

ROOT Serialization / Deserialization

- In spark-root I basically implemented ROOT serialization/deserialization of baskets without any code generation of the logic.

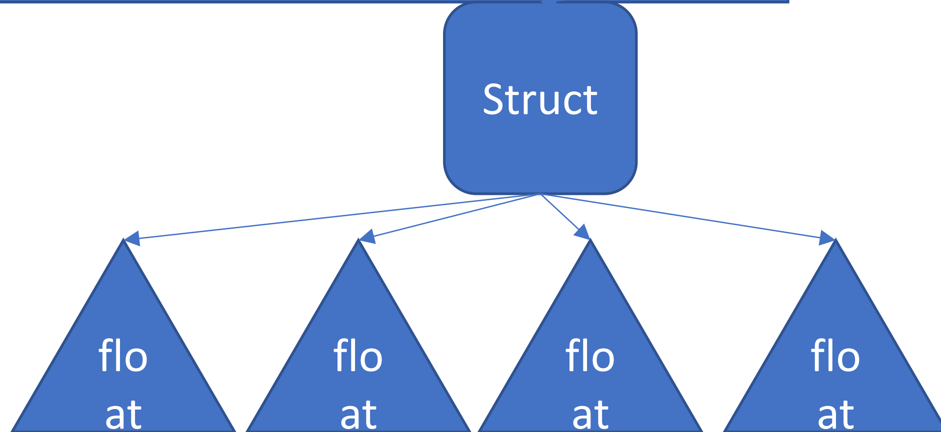
- I define serialization/deserialization to be the procedure to interpret a binary blob in memory as some typed c++ / whatever object.

ROOT I/O + Serialization / Deserialization

- When people say ROOT I/O, they mean both I/O and serialization/interpretation
- ROOT essentially collapses these two completely independent parts.
  - But u can use TX::Streamer method to get just ser/deser
- ROOT has a notion of a type system -> c++ type system. However, most of that can be drilled down even further. Next slides

What I propose

ROOT I/O



A Record will contain

- Key
- Buffer of simple type
    - Corresponds to either buffer of offsets or leaf nodes types

For instance, Buffer of a float leaf will be compressed and a record Created.

In Memory Model

List

Struct

Struct

int

List

flo

flo

flo

int

Layer 1 -> ROOT Records
Layer 2 -> ROOT Memory Model using a common expressive enough type system (not full c++, but preserve c++ type system as well!)
Do not build General Purpose Classes until
It is explicitly requested (configurable… obviously)
Layer 3 -> Representation (can be general purpose classes like what Experiments have now). The point is that you do not build them until requested to.

ROOT I/O

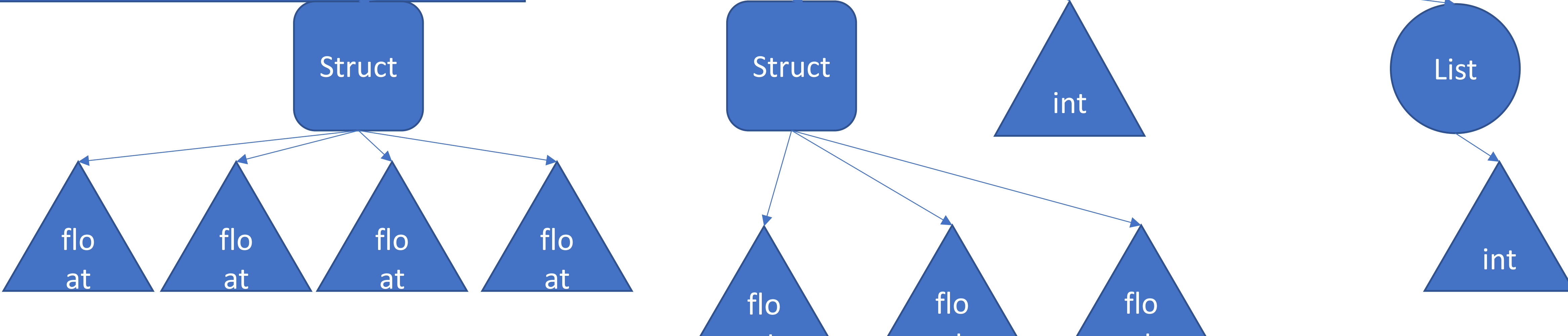
In Memory Model

Essentially, this is what Forest, Apache Arrow do right now. Both provide essentially the Same Memory Model

Why beneficial?! Experiments when porting things to heterogenous archs
Do not need the deserialization step to be done at all on a cpu!!!
You need this vector form!

ROOT I/O

In Memory Model



What I propose

- Provide a clear boundary between I/O and serialization/interpretation
- Establish a common type system and preserve c++ at the same time
- Employ Apache Arrow or something like that for the memory layout layer. See in the next slides
- code generation (may be compile time or may be run time) of interpretation of Memory Model into General Purpose Objects, that are necessary for Experiments…. There is no way to go around this…
- For instance, for GPUs, we do not need ROOT to run deserialization into general purpose std::tuple<int, double, std::vector<int>> etc…. So that I then transfer something to the device. You already have a vectorized form for every leaf of the type tree.

Apache Arrow

- https://arrow.apache.org/docs/memory_layout.html
- It’s a memory layout, it’s not another parquet/data storage format
  - It is a thorough specification of how data to be laid out in memory
  - It is expressive enough to support a form of polymorphism.
  - When doing parsing of CMS RECO/AOD Streamer record, you can see that the number of times we parse pointers of polymorphic things is quite small w.r.t. the cases of things that are really known beforehand.
- Here I just want to show how I view Apache Arrow

- Let’s build a simple example that showcases how
  - given general purpose typical HEP memory layout
  - generate Apache Arrow
  - Get general purpose back

Apache Arrow

```
29  template<typename T>
30  struct particle {
31      std::tuple<T, T, T> position;
32      vector4<T>          momentum;
33      int                 charge;
34      std::vector<int>  some_properties;
35
36      static particle<T> get_random() {
37          return {
38              {static_cast<T>(std::rand()), static_cast<T>(std::rand()), static_cast<T>(std::rand())},
39              vector4<T>::get_random(),
40              static_cast<int>(std::rand() % 5),
41              {1,2,3,4,5,6}
42          };
43      }
44  };
45
46  }
```

Particle class declaration/definition

```
10  using particles = std::vector<hep::particle<float>> ;
11  std::vector<particles> generate(int num_entries, int max_num_particles) {
12      std::vector<particles> batch;
13      for (int i=0; i<num_entries; ++i) {
14          particles ps;
15          int num_particles = std::rand() % max_num_particles;
16          for (int j=0; j<num_particles; ++j) {
17              ps.push_back(hep::particle<float>::get_random());
18          }
19          batch.push_back(ps);
20      }
21
22      return batch;
23  }
```

Generate
Std::vector<std::vector<particle>>
A la events/rows of list of particles

Apache Arrow – Convert Row based to Arrow Arrays

Define all Arrow Array Builders

```
31      // builders for particle struct fields
32      auto *pool = arrow::default_memory_pool();
33      using tuple3_builder = std::array<arrow::FloatBuilder, 3>;
34
35      // momentum field
36      std::vector<std::shared_ptr<arrow::ArrayBuilder>> momentum_values_builders {
37          std::make_shared<arrow::FloatBuilder>(pool),
38          std::make_shared<arrow::FloatBuilder>(pool),
39          std::make_shared<arrow::FloatBuilder>(pool),
40          std::make_shared<arrow::FloatBuilder>(pool)
41      };
42      std::shared_ptr<arrow::StructType> momentum_type {new arrow::StructType{
43          std::vector<std::shared_ptr<arrow::Field>>{
44              std::make_shared<arrow::Field>("x", std::make_shared<arrow::FloatType>(), false),
45              std::make_shared<arrow::Field>("y", std::make_shared<arrow::FloatType>(), false),
46              std::make_shared<arrow::Field>("z", std::make_shared<arrow::FloatType>(), false),
47              std::make_shared<arrow::Field>("t", std::make_shared<arrow::FloatType>(), false)
48          }
49      }};
50      std::shared_ptr<arrow::StructBuilder> momentum_builder { new arrow::StructBuilder{momentum_type, pool,
51          std::move(momentum_values_builders)}};
52
53      // position field
54      std::vector<std::shared_ptr<arrow::ArrayBuilder>> position_values_builders {
55          std::make_shared<arrow::FloatBuilder>(pool),
56          std::make_shared<arrow::FloatBuilder>(pool),
57          std::make_shared<arrow::FloatBuilder>(pool)
58      };
59      std::shared_ptr<arrow::StructType> position_type {new arrow::StructType{
60          std::vector<std::shared_ptr<arrow::Field>>{
61              std::make_shared<arrow::Field>("x", std::make_shared<arrow::FloatType>(), false),
62              std::make_shared<arrow::Field>("y", std::make_shared<arrow::FloatType>(), false),
63              std::make_shared<arrow::Field>("z", std::make_shared<arrow::FloatType>(), false),
64          }
65      }};
66      std::shared_ptr<arrow::StructBuilder> position_builder{ new arrow::StructBuilder{position_type, pool,
67          std::move(position_values_builders)}};
```

Actual Conversion

```
113         // iterate through all the rows and convert row-based to apache arrow data model
114         for (auto const& entry : batch) {
115             // for each particle of particles
116             vector_particles_builder->Append();
117             for (auto const& particle : entry) {
118                 // for each particle
119                 particle_builder->Append();
120
121                 //
122                 // particle fields
123                 //
124                 auto *position_builder =
125                     static_cast<arrow::StructBuilder*>(particle_builder->field_builder(0));
126                 auto *momentum_builder =
127                     static_cast<arrow::StructBuilder*>(particle_builder->field_builder(1));
128                 auto *charge_builder =
129                     static_cast<arrow::Int32Builder*>(particle_builder->field_builder(2));
130                 auto *properties_builder =
131                     static_cast<arrow::ListBuilder*>(particle_builder->field_builder(3));
132
133                 // position field
134                 position_builder->Append();
135                 auto const& [x, y, z] = particle.position;
136                 static_cast<arrow::FloatBuilder*>(position_builder->field_builder(0))
137                     ->Append(x);
138                 static_cast<arrow::FloatBuilder*>(position_builder->field_builder(1))
139                     ->Append(y);
140                 static_cast<arrow::FloatBuilder*>(position_builder->field_builder(2))
141                     ->Append(z);
142
143                 // momentum field
144                 momentum_builder->Append();
145                 auto const& [x1, y1, z1, t1] = particle.momentum;
146                 static_cast<arrow::FloatBuilder*>(momentum_builder->field_builder(0))
147                     ->Append(x1);
148                 static_cast<arrow::FloatBuilder*>(momentum_builder->field_builder(1))
149                     ->Append(y1);
150                 static_cast<arrow::FloatBuilder*>(momentum_builder->field_builder(2))
151                     ->Append(z1);
152                 static_cast<arrow::FloatBuilder*>(momentum_builder->field_builder(3))
153                     ->Append(t1);
```

Apache Arrow – a memory model + type system

std::vector<std::vector<particle>>
A la events/rows of list of particles

List

Array of offsets
Null bit map

Struct

Struct

Struct

int

Array of ints

Array of offsets
Null bit map

List

flo at

Array of floats

flo at

Array of floats

flo at

Array of floats

flo at

Array of floats

flo at

Array of floats

flo at

Array of floats

flo at

Array of floats

int

Array of ints

Apache Arrow – Convert Arrow Arrays to Row based

Get all Arrow Arrays

```cpp
// get at a single entry level
auto const* vector_particles_array = static_cast<arrow::ListArray*>(array.get());
auto const* particles_values_array = static_cast<arrow::StructArray*>(vector_particles_array->values().get());
auto particles_offsets_buffer = vector_particles_array->value_offsets();
auto const *raw_particles_offsets = reinterpret_cast<uint32_t const*>(particles_offsets_buffer->data());

// particle struct fields' arrays
auto const *position_array = static_cast<arrow::StructArray*>(particles_values_array->field(0).get());
auto const *momentum_array = static_cast<arrow::StructArray*>(particles_values_array->field(1).get());
auto const *charge = static_cast<arrow::Int32Array*>(particles_values_array->field(2).get());
auto const *some_properties = static_cast<arrow::ListArray*>(particles_values_array->field(3).get());

// position fields' arrays
auto const *position_x_array = static_cast<arrow::FloatArray*>(position_array->field(0).get());
auto const *position_y_array = static_cast<arrow::FloatArray*>(position_array->field(1).get());
auto const *position_z_array = static_cast<arrow::FloatArray*>(position_array->field(2).get());

// momentum fields' arrays
auto const *momentum_x_array = static_cast<arrow::FloatArray*>(momentum_array->field(0).get());
auto const *momentum_y_array = static_cast<arrow::FloatArray*>(momentum_array->field(1).get());
auto const *momentum_z_array = static_cast<arrow::FloatArray*>(momentum_array->field(2).get());
auto const *momentum_t_array = static_cast<arrow::FloatArray*>(momentum_array->field(3).get());

auto const* raw_properties_offsets = some_properties->raw_value_offsets();
auto const *properties_values_array = static_cast<arrow::Int32Array*>(some_properties->values().get());

// debugging
std::cout << "arrow array size = " << array->length() << std::endl;
std::cout << "arrow ofsets buffer size = " << particles_offsets_buffer->size() << std::endl;
std::cout << "arrow offsets buffer capacity = " << particles_offsets_buffer->capacity() << std::endl;

// for each event (a la entry or row)
```

Build row based representation

```cpp
// debugging
std::cout << "arrow array size = " << array->length() << std::endl;
std::cout << "arrow ofsets buffer size = " << particles_offsets_buffer->size() << std::endl;
std::cout << "arrow offsets buffer capacity = " << particles_offsets_buffer->capacity() << std::endl;

// for each event (a la entry or row)
for (int entry=0; entry < array->length(); ++entry) {
    particles v;
    std::cout << "raw[" << entry << "] = " << raw_particles_offsets[entry] << std::endl;
    std::cout << "Entry = " << entry << " number of particles per entry = " <<
        raw_particles_offsets[entry+1] - raw_particles_offsets[entry] << std::endl;

    // for each particle
    for (int j=raw_particles_offsets[entry]; j<raw_particles_offsets[entry+1]; j++) {
        hep::particle<float> particle {
            // build position
            {
                position_x_array->raw_values()[j], position_y_array->raw_values()[j],
                position_y_array->raw_values()[j]
            },
            // build momentum
            {
                 momentum_x_array->raw_values()[j], momentum_y_array->raw_values()[j],
                 momentum_z_array->raw_values()[j], momentum_t_array->raw_values()[j]
            },
            // charge
            charge->raw_values()[j],
            //std::vector<int>{5, 6, 7}
            // list of some properties
            std::vector<int>{
                &properties_values_array->raw_values()[raw_properties_offsets[j]],
                &properties_values_array->raw_values()[raw_properties_offsets[j+1]]
            }
        };

        v.push_back(particle);
    }

    result.push_back(v);
}
```

What I propose

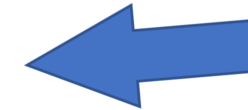
- Provide a clear boundary between I/O and serialization/interpretation
- Establish a common type system and preserve c++ at the same time
  - Apache Arrow defines a good enough to start type system
  - We already have mechanisms to preserve c++ type system (schema) - streamers
- Employ Apache Arrow or something like that for the memory layout layer. Testing in the next slides
- Code generation/use of interpreter for reinterpretating into a representation that a user needs. See next slides
  - At compile time, similar to how ROOT generates code for serialization/deserialization (gen reflex, rootcling…), generate code to interpret Apache Arrow Arrays as row based user defined stuff.
- The idea is to build on top of what rootcling provides but instead of current way of doing deser/ser, employ a memory model + allow this reinterpretation into a user representation to be done on demand
  - For architectures like GPUs, we do not need deserialization step at all!
  - We could help experiments by providing this memory model + on-demand deserialization.

Quick test of Apache Arrow -> root integration

- I want to take arrow::Array and read/write that to ROOT file
- Write functionality sufficient to store the above example :
  - https://github.com/vkhristenko/test-apache-arrow/tree/master/root-arrow

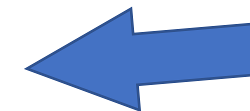
- Similar logic as for TTree
- Just a couple days of evening time

```
vk@viktors-MacBook-Pro:~/software/test-apache-arrow/build$ root -l test.root
root [0]
Attaching file test.root as _file0...
Warning in <TClass::Init>: no dictionary for class ROOT::RArrowInterface is available
Warning in <TClass::Init>: no dictionary for class ROOT::RLink is available
(TFile *) 0x7fa1c81c4010
root [1] _file0->Map()
20181025/235155  At:100    N=114      TFile
20181025/235155  At:214    N=73                    CX =  0.40
20181025/235155  At:287    N=193                   CX =  0.15
20181025/235155  At:480    N=193                   CX =  0.15
20181025/235155  At:673    N=193                   CX =  0.15
20181025/235155  At:866    N=193                   CX =  0.15
20181025/235155  At:1059   N=193                   CX =  0.15
20181025/235155  At:1252   N=193                   CX =  0.15
20181025/235155  At:1445   N=193                   CX =  0.15
20181025/235155  At:1638   N=193                   CX =  0.15
20181025/235155  At:1831   N=197                   CX =  0.15
20181025/235155  At:2028   N=1013                  CX =  0.03
20181025/235155  At:3041   N=318      ROOT::RArrowInterface CX =  1.41
20181025/235155  At:3359   N=771      StreamerInfo   CX =  1.71
20181025/235155  At:4130   N=129      KeysList
20181025/235155  At:4259   N=53       FreeSegments
20181025/235155  At:4312   N=1        END
```



All the data record
A la Baskets

Just to note, that data is there,
But I’m not sure how to trigger
compression

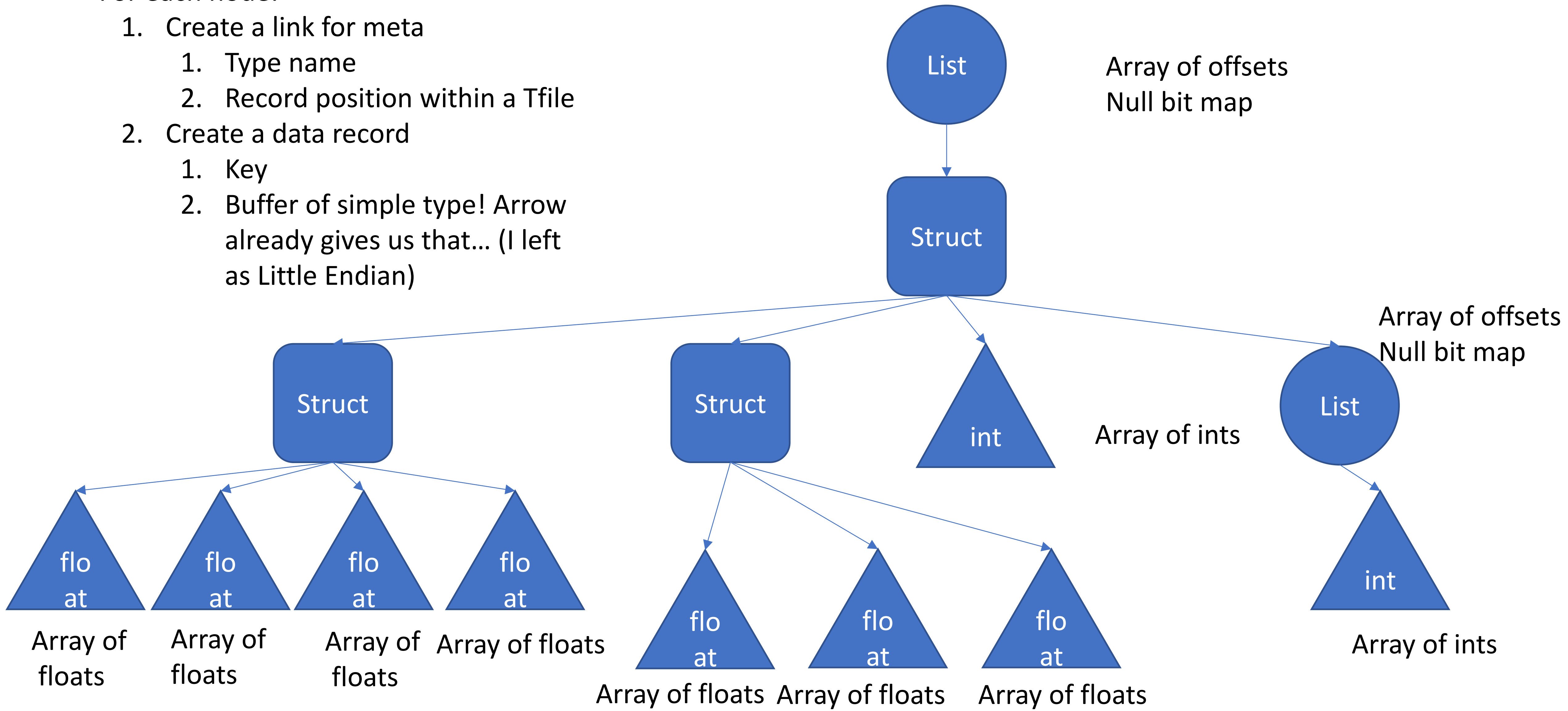


Entry record

Algorithm

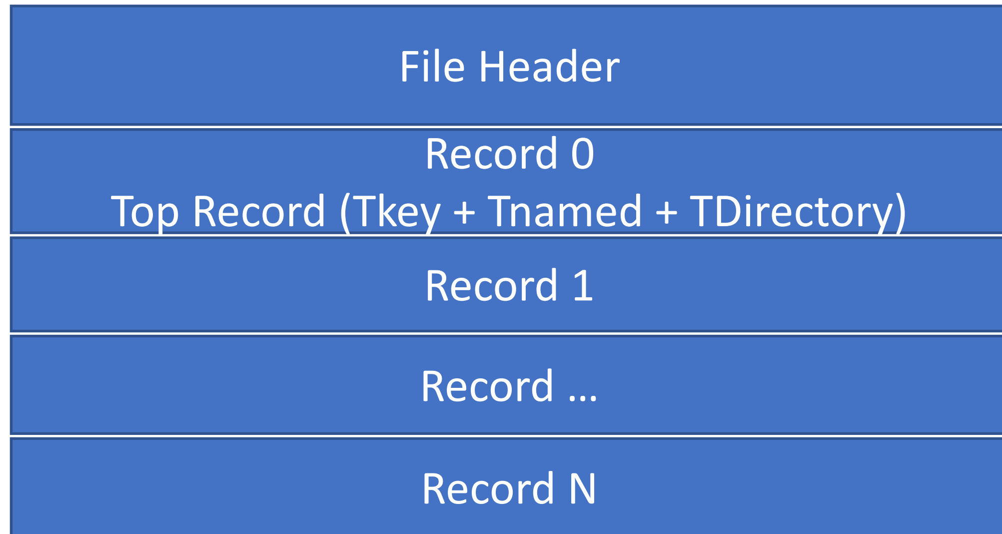
Test impl is here:
https://github.com/vkhristenko/test-apache-arrow/blob/master/root-arrow/src/RVisitors.cpp#L53

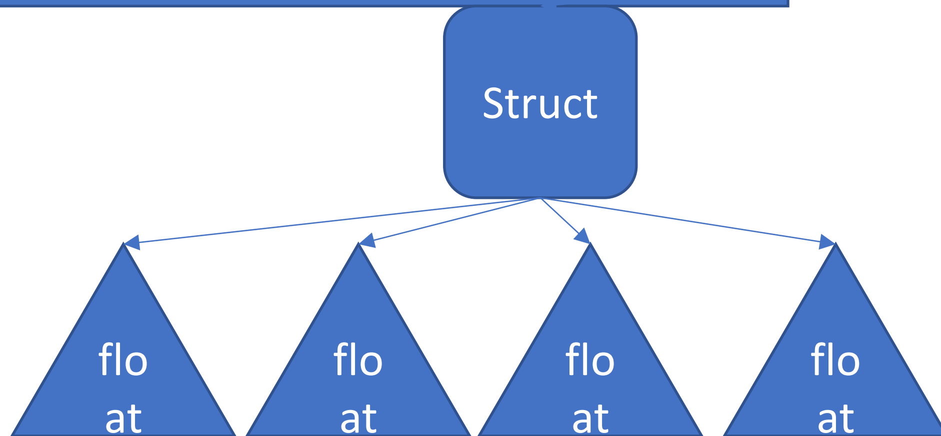
- For each node:
  1. Create a link for meta
     1. Type name
     2. Record position within a Tfile
  2. Create a data record
     1. Key
     2. Buffer of simple type! Arrow already gives us that… (I left as Little Endian)



What I propose

ROOT I/O





A Record will contain

- Key
- Buffer of simple type
  - Corresponds to either buffer of offsets or leaf nodes types

For instance, Buffer of a float leaf will be compressed and a record Created.

List

Struct

Struct

int

List

flo

flo

flo

int

What I propose – memory model + on demand reinterpretation

- Introduce a memory model (like Forest, but we already have Apache Arrow, I can extend my quick prototype to fully read/write -> need a bit of help with Tkey… for now I did a trivial thing like Tbasket does… but something is off…)

- We already do the same but for ROOT’s own layout of binary buffers.

- For archs, other than x86 like ones, vector machines… for instance, we do not need the deserialization step.

- Experiments will further go into this direction, at least we can provide a way to help them
  - in my view this memory model is a substantial help, cause i do not have to go in and try to change the existing data formats………. which is what I have to do now… in order to put stuff on a device.
  - Integration -> if people need general purpose -> on demand deserialize/reinterpret.

What I propose - reinterpretation

- Given experiments want to utilize heterogenous resources, current row based layouts are not good………
- Moreover, most of the time, you do not need to ship the whole object to the e.g. GPU, only some fields

Why this way

- This would all allow us to integrate into current codebase with ROOT I/O
- Provide an interface for general purpose computations as well, since in many cases that is what experiments need
- Skip deserialization step when it's not needed
- Integrate with other open source initiatives, for instance the development of LLVM based compiler to execute analytics directly on Apache Arrow Arrays
- And more…

Why do I want to collaborate

- Genreflex/rootcling
  - Dictionary generation is difficult
  - Source code base is quite large…. I checked
  - I want to build on top of root already provides with dictionary generation
- Goes hand in hand with what Forest is trying to do
  - In my opinion
- See next

Prototype for CMS

- Here is a proposal for a prototype…
- Consider CMS core framework

- First, implement the memory model concept and required code generation/interpretation
- Integrate with the existing cmssw framework
- Provide preliminary tests on dummy workloads
- Integrate with real reco workflows
  - Cpu / gpu

- This could further generalize to the rest of experiments and what ROOT will provide is foundation (memory model + general purpose class assembly + on demand) + different layers of abstractions on top.

Prototyping, or these are just thoughts

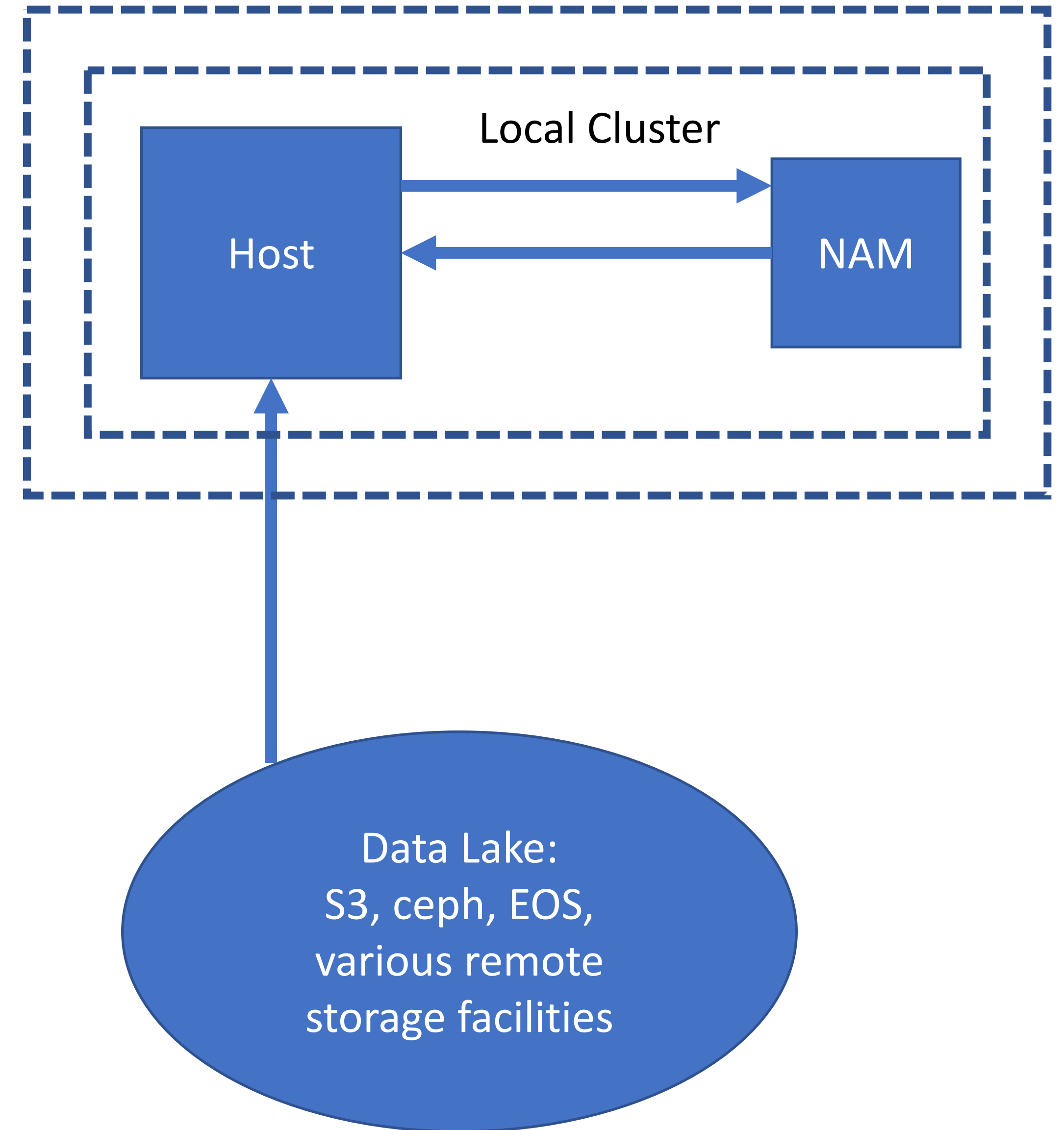
- I'm experimenting with Apache Arrow
  - https://github.com/vkhristenko/test-apache-arrow
  - Complex enough, similar to what experiments have as collection of particles
  - A quick recipe to store apache arrow arrays in TFile

- I've reimplemented ROOT I/O (in my sense) in c
  - https://github.com/vkhristenko/rootio
  - Only with hdfs + local fs (read/write).
    - Want to extend to use mmap + eos integration
  - Can use ROOT to deserialize and write using my implementation
  - For now, record for streamers is empty (key + empty TList)
    - To read back in ROOT need to set a static field of TFile::SetReadStreamerInfo(false)
    - This is experimentation/testbed/playground, not aiming anywhere except testing for now.
  - Bindings for python, rust, cpp { ctype, cffi }
    - read/write of records

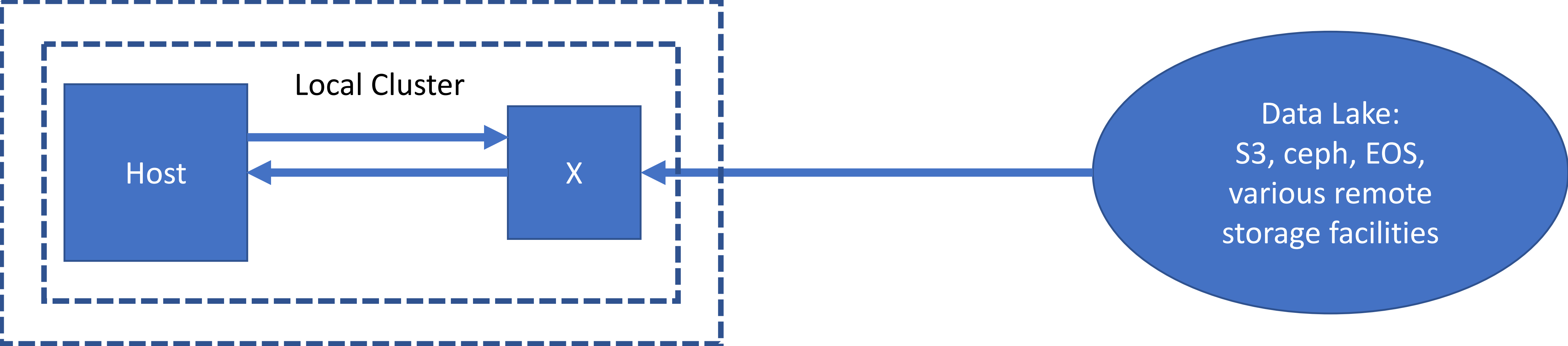
Attract Proposal

- HIOS: Heterogenous I/O for Scale
- Objective: Remove the I/O + Compression/Decompression specific functionality from the host to an FPGA on the cluster.
- Within the DEEP-EST Project, we have NAM
  - Network Attached Memory
    - https://www.deep-projects.eu/hardware/memory-hierarchies/49-nam
  - FPGA with HMC (DDR4) for several TBs per board
  - Right now can malloc/free from the host
- Proposal is to repurpose this board slightly

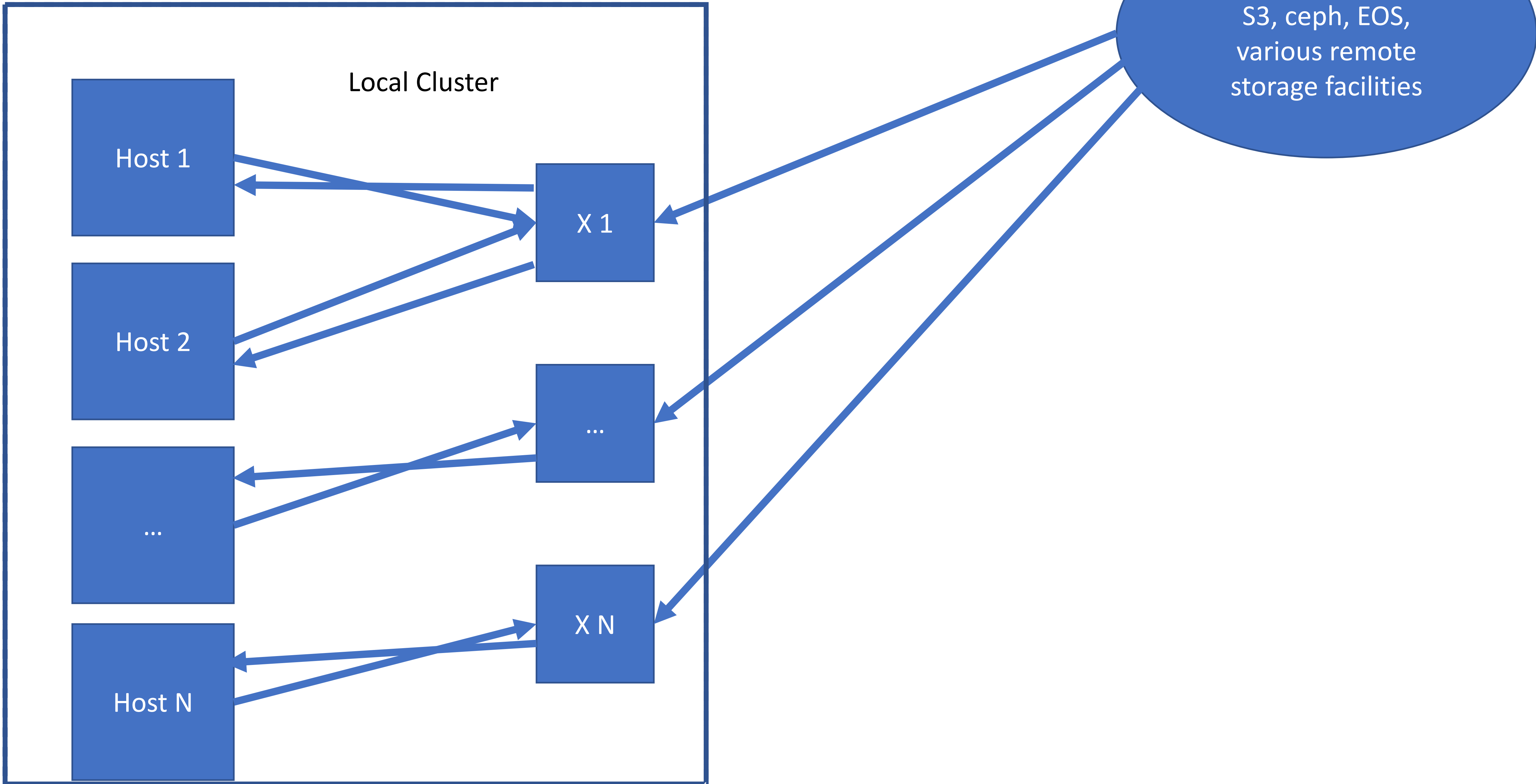
Current usage of NAM



Foreseen usage of X



Foreseen usage of X



Attract Proposal

- HIOS: Heterogenous I/O for Scale

- Why not offload I/O specific logic from the host?
  - Read/Write is quite an important part of all of our workflows.
    - Coding/Decoding of ROOT I/O logic
  - Would allow to use less RAM on the compute host
  - Decompression/Compression of Baskets/Binary Blobs could be also done on the board.

- HPC centers
  - Would allow to use compute units by data intensive applications w/o really changing the host
  - Would allow to utilize the expensive links….

Pure ROOT I/O

- I rewrote the ROOT I/O only part in c so that if ATTRACT is there, I'm ready to put that logic in on that board…

- However, during the ROOT User Workshop there was a question of how to integrate uproot with current root. And this also applies in there
  - If we have a pure well-optimized ROOT I/O for different ways of getting binary blob to a compute unit.
  - Uproot can take advantage of various things python provides for reinterpretation. Other people could potentially build on top of that.
  - Go impl does not need to be standalone, but can utilize the I/O and reinterpret binary buffers whichever they want
  - This is not about the language -> it is about "systematic/architectural/modular approach" which could also enable more people to contribute.
  - Futuristic but, Philippe said back at the workshop "the age of posix i/o comes close to an end". What I'm interested to see is we can also go and influence the future of how I/O is handled. Especially given that I/O is fundamental for data intensive workflows.