Chapter 1

The New AOD Format

1.1 Design

1.1.1 Entity-Component paradigm

In order to meet the previously stated requirements we eventually settled on a specialized Entity-Component paradigm[1] similar to but not to be confused with the Entity-Component-System (ECS) paradigm often seen in game development(CITE).

As the name might suggest, this system is a Composition-over-inheritance style of design wherein a focus is laid not on is-a relationships but has-a type relationships. This type of design has often (CITE) been shown to work well with data-oriented design. In this system an entity is defined as a collection of components. Each component is a simple POD type acting similar to a member variable in c++. Component should, in general, not contain any functions save for operator definitions if needed. Entities in turn should not contain any data but only define functions. By using a meta-template programming and passing components as template arguments to an entity it is in fact possible to selectively mask an entities functions based on the components available to a given entity at any given time.

In a regular ECS system interaction is done via 'Systems', the 'S' in 'ECS', but this does not match well to our criteria of keeping an interface similar to the currently used one, nor does it offer any advantages compared to a simple for-loop for our particular needs.

Implementation

In order to implement our EC system we make heavy use of C++11 meta-template-programming techniques. These allow us to resolve and check many issues at compile time. Furthermore, we try to enforce a strict system of data-ownership. Memory leaks are unfortunately a common occurrence once one deals with over a thousand users. To be specific our EC system consists of three parts to go from AOD-data to an entity, from the bottom up these are

[1] Not to be confused with an Entity-Component-System (ECS) which specifies an interaction via systems as well.

- **Entity**< *Components* ... >**:** a raw entity, containing the components that are to be included.

This gets a bit complicated once we start including entity-keys, 'inner classes', write later when less sleepy. Probably better to use subsubsubsections for each member. add code-examples and pictures to make clear.

1.1.2 Storage Format

A provisional storage format has been developed based on the Entity-Component system described above. An AOD file can contain (in addition to any necessary file-header data), Entity/Component pairs. Meta-data such as identifiers is stored in a header, seperate from the actual component data. This header currently[2] looks like

- Number of Entities
- repeat for *Number of Entities...*
  - Entity Id.
  - Number of Components
  - repeat for *Number of Components...*
    - Component Id
    - Offset in file to start of data
    - size of data
    - number of flags
    - array of flags (such as compression algorithm used, uncompressed size, size and location of any skimming meta-data)

This header will be placed at the start of a file so that when reading from disk one can first only read this header and subsequently only ever read the data that is necessary for the task at hand without having to seek through the file as one would in a list based format.

By applying compression on a per-component-data basis we get two benefits, for the small cost of an increased compression overhead. Firstly, it is possible to leverage whichever form of compression gives the best results for a given data set. This could involve specialized algorithms such as fzip(CITE) for floating point data, which we have benchmarked in (REF). Secondly, we reading back the compressed data we only have to decompress those components which we are interested in.

We envision that in the future support might be added for blocked data and headers, linked together in a list-like fashion. This is because some workflows require the merging of multiple AOD files into a single output file or might want to grow the AOD by adding extra entities. if this happens and the file data is tightly packed, this would involve copying the entirity of the input file(s) in order to make room for the expanded header and data. By chaining data and headers in blocks one only needs to write the added data[3]. In order to optimize access, this resulting 'merged' AOD can then be 'defragmented' should the need arise.

[2] Not really... the flags aren't included yet...

[3] it might even be possible to not even perform this copy if the merge is allowed to be destructive if the underlying storage is using a page based storage system.

1.2 Features

1.2.1 Pruning

Our new Entity-Component system allows us to do very efficient pruning in a very easy-manner. Because one has to specify the components of any entity one wishes to use, the data-handler only ever reads or requests those entity/component pairs which the user has explicitly asked for. In essence, one does not have to do pruning, specifying what can be left out, at all. One has to do anti-pruning: specify what has to be included.

There was initially some concern over the performance impact of this method. Previously, data was stored in a array-of-structs fashion. While this didn't allow for effective pruning it did mean that when iterating over an entity data would be read in a linear, consequetive fashion. Using the new Entity-Component system, data is storen in a struct-of-arrays like fashion and reads will be linear in each component-array but will look scattered when considering entity-wise iteration. To that end, a benchmark was written to see the effective throughput of both methods, the results of which are shown in (REF).

1.2.2 Skimming

Skimming can be done in two fashions, it is either an independent cut or a more complicated formula dependent upon several variables. In the case where we need to skim a single independent variable it is useful to store a minimum and maximum range for different slices of the data. For example for every 32 entries. This can then be repeated storing the minium and maximum of $32^2 = 1024$ entries in such a way we can effectively cut on minimum and maximum values for large sets of data. We will refer to this structure as a *skim-list*.

For the more complicated case where we want to skim on a function of several variables there are two approaches. Either we parameterize the function in such a way that it is a function of the minima and maxima of its depenedent variables or we use the pruning feature to extract those variables needed for our cutting function and use them to generate a list of indices-of-interest. No matter the approach, performing skimming on the server-side will always impose a serialization penalty.

Skimming has not yet been implemented but we envision the skim-list to live side-by-side with the component data inside of the data-handler, computed or loaded on an as-needed basis. The skim-list can be stored in the AOD file, referenced by a flag in the component-data header.

The handler class will also perform the skimming operation, as it is the owner of the data and the part of the code which communicates with other processes.

1.2.3 Growing

Growing, like pruning, is trivial under this Entity-Component system. In order to extend an entity with a given component or add a new entity all together one simply has to define the component types and then they can be added and used just like any other entity/component pair. In section Storage Format(REF) we go into detail as to how one can grow an AOD file.

1.2.4 Derived Types

Derived types have not yet been implemented but have been designed. The principle is simple, if one makes a request to the handler for an Entity/Component pair that is not present the handler will check if it can be constructed from the data available to it, the necessary information for which will be encoded into the deriving type using the meta-template-programming approach used throughout the entity system. If it finds that it can construct the data it will do so and keep it available for future requests, until such a time where it is no longer used *and* the handler needs to free up memory. Care will have to be taken when constructing the data because the datahandler can be shared between multiple clients, therefore it is vital that only one of the clients will start to construct the data. TODO: Psuedo-code here, maybe one of those communication graphs (in essence, check if available, if not, request it, if no response, atomically/locked add the derived data-type with a flag saying it's being build, allocate space, compute it, set flag to done. if available but computing, wait. If someone else requests it but you are computing return that you are computing. if you receive that someone is computing wait on the message that it's done. might duplicate data in worst case?)

1.2.5 Schema Evolution

Discuss, complicated.

1.2.6 Interface

Finish EC part first

1.2.7 Vectorization

raw array access through EntityCollection, easy to vectorize. Try autovectorization using expression templates on entity iterators/Collections. show speedup benchmark.

1.2.8 On Concurrency

per proccess, train based concurrency already, baked in through multiple other options. For local analysis not relevant, ROOT can be problematic with multithreading, but entity data is immutable and thread/fork safe as it is, in general, immutable data. so one can easliy wrap a long calculation in an omp parallel for.

1.3 Usage

show don't tell, add example

1.3.1 Event-building

do tell here, show cost, different methods.

1.3.2 Writing Analysis Tasks

show don't tell

1.4 Benchmarks

Effect of compression, old vs new system, runtime, filesize, effect of vectorization, at least mention distributed things.

1.5 Future Work

expanding the file format, testing everything, connecting up to the analysis train, involving users, autovectorization of entity iteration, auto-gpu targeting?