Chapter 1

The New AOD Format

1.1 Design

In order to meet the previously stated requirements we eventually settled on a specialized Entity-Component paradigm[1] similar to the Entity-Component-System (ECS) paradigm often and first commonly seen in game development(CITE) but with wider application. DESCRIBE ECS PARADIGM

Our specialization differs from the usual implementations in a few ways. First, we do not use Systems for interacting with the Entities as one would in a regular ECS pattern (The 'S' in ECS stands for System after all). This is because of two reasons: The requirement to maintain a similar interface to the old implementation and because systems are often useful in iterating over all Entities which have a certain Component we tend to only iterate over a given entity with certain components which is more suitable to a simple *for* loop.

Secondly, all Components associated with a given Entity are indexed by the same offset into the Component Array. Most existing ECS frameworks will store a different index for each component associated with a given entity. This is because this paradigm has its roots in Game Design where Entities are often created and destroyed many times a second and there are often multiple variations of a given entity existing at once (that is, entities with the same Entity type but different components). By giving each component of an entity it's own index components can be packed together and it is easier to reuse previously allocated slots when creating new entities. For us, AODs are write-once/read-often (in fact, during analysis we try to restrict them to read-only access as much as possible so that data can be shared between processes) which means we do not need to take into account creation and deletion of entities. Furthermore if a component is included for a given entity we include it for all of them. In practice this is done by having a common index for all one-to-one mapped components. That is, for entity 'i' the associated component A is at A[i] and component B is at B[i]. For one-to-many or many-to-many components (for example, 'which clusters were used when reconstructing this track') we access the data using two arrays. First, we have an array of *(offset, size)* pairs for each entity entry followed by and array of the actual component data. The offset/size pair then

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

gives the offset into this data-array where the associated data starts and how many elements from this offset are included. For one-to-many data we only need to store the size part on disk and recompute the offsets dynamically, for many-to-many there is a trade-off between storing both offsets and sizes and duplications in the data-array.

A provisional storage format has been developed (DESCRIBE IT), we imagine in the future that (CHANGES)

Include parts here about what has been attempted but didn't work too, add section on implementation details/difficulties?

1.1.1 Data-flow

Here we will describe how data moves around for analysis tasks and the creation of AODs. how we envision the interaction with MQ

1.2 Features

1.2.1 Pruning

One of the key features that was requested of the new model was the ability to do efficient *pruning* of the data. Pruning is the ability to only read or extract a subset of components for each entity. For example, one might only be interested in the kinematic variables of each track, ignoring for example components relating to the spatial position or the covariances relating to fits. The benefit of this is two-fold: it allows the user to efficiently reduce a dataset before copying it to a local machine for further analysis and when performing analysis it can greatly improve the effective bandwidth.

Previously, data was stored in a array-of-struct like fashion. This method did not allow for efficient pruning for two reasons. Firstly, because data is compressed when stored to disk all components of a given entity where decompressed when any of them where accessed. Secondly, once they had been decompressed into memory all the components where stored in an interleaved fashion. Because memory access happens on a cache-line level of granularity (64 bytes for an x64 based machine) this would mean that in the worst case scenario we need to fetch 64 bytes from memory for every byte of useful data reducing the effective bandwidth (TODO: expand, define) by a factor of 64.

In the new system, due to the use of an Entity-Component system it is possible to do efficient pruning in a natural fashion. Remember that we effectively store *Entity/Component→Data-array* pairs pruning is a matter of only reading those data-arrays which are actually referenced by the user.

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, consequtive fashion. Using the new struct-of-arrays like fashion 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.

1.2.2 Skimming

1.2.3 Growing

1.2.4 Derived Types

1.2.5 Schema Evolution

1.2.6 Interface

1.2.7 Vectorization

1.2.8 On Concurrency

1.3 Usage

1.3.1 Event-building

1.3.2 Writing Analysis Tasks

1.4 Benchmarks

1.5 Future Work