Contents

1 Introduction **3**
- 1.0.1 CERN . . . 3
- 1.0.2 The ALICE Experiment . . . 3
- 1.1 Run 3 . . . 3
- 1.2 The O2 Work-group . . . 3
  - 1.2.1 Goals . . . 3
  - 1.2.2 Facilities & Framework . . . 3

2 The AOD format **4**
- 2.1 Analysis . . . 4
- 2.2 Requirements . . . 5

3 Strengths and Flaws of the Existing implementation **7**
- 3.1 Strengths . . . 7
- 3.2 Weaknesses . . . 7

4 The New AOD Format **9**
- 4.1 Design . . . 9
  - 4.1.1 Entity-Component paradigm . . . 9
  - 4.1.2 Storage Format . . . 10
- 4.2 Features . . . 11
  - 4.2.1 Pruning . . . 11
  - 4.2.2 Skimming . . . 11
  - 4.2.3 Growing . . . 11
  - 4.2.4 Derived Types . . . 12
  - 4.2.5 Schema Evolution . . . 12
  - 4.2.6 Interface . . . 12
  - 4.2.7 Vectorization . . . 12
  - 4.2.8 On Concurrency . . . 12
- 4.3 Usage . . . 13
  - 4.3.1 Event-building . . . 13
  - 4.3.2 Writing Analysis Tasks . . . 13
- 4.4 Benchmarks . . . 14
- 4.5 Future Work . . . 14

Appendices **15**

A Acknowledgements **16**

B Glossary **17**

Chapter 1

Introduction

1.0.1 CERN

General CERN exerpt.

1.0.2 The ALICE Experiment

Talk about the goals of the ALICE experiment here and and put a nice big picture of the detector explaining how it operates (different detectors, track reconstruction, vertices) so that everyone can understand wth I'm talking about.

1.1 Run 3

Talk about the changes in run 3, continious read-out, challenges posed by this. Estimated increases in data taking.

1.2 The O2 Work-group

1.2.1 Goals

1.2.2 Facilities & Framework

Maybe not needed, Facilities can be high-level overview.

Chapter 2

The AOD format

The AOD (Analysis Object Data) format is intended to be the main file format to be used for the analysis of data produced by the experiment; both physical data and simulated Monte-Carlo data. They are the final output format for both methods of data taking, they are constructed from ESD (Event Summary Data) files containing cleaned cluster-data and reconstructed tracks and vertices. It is estimated that a yearly-median of 21 PB worth of AODs will be stored between 2020 and 2027 (excl. a 3 year shutdown between 2022 and 2025) making efficient usage and storage very important. AODs will be stored at dedicated facilities for subsequent analysis.

2.1 Analysis

Generated AODs will be stored and processed in dedicated *Analysis Facilities* or on local machines. These Analysis Facilities are specialized datacenters, part of the ALICE Computing Grid. Physicists can submit different tasks to the Grid to run against a large (or small) set of data. The Grid software automatically handles the scheduling of tasks and the IO of the AODs. Tasks may be split up over multiple machines or even multiple Grid *sites* at once, the smallest level of granularity the scheduler can handle is a single AOD file.

There are currently two ways of submitting jobs to the Grid. One can submit a single analysis task as a c++ source file, specifying the input files at run-time, or alternatively one can submit a task as part of a so-called Analysis Train. In order to do so, the analysis task has to be checked in to the central source-code repository. It is then possible to tweak the parameters and select the data of interest from a web-interface. Several tasks submitted in such a fashion (called wagons) will be gathered together by the central software reducing the required movement of data. Using analysis-trains has proven to be a much more effective utilization of resources and single-tasks are therefore being phased out in favor of analysis-trains.

Each task (or wagon as they are also called) runs as it's own separate process. Tasks can communicate between each other and other actors using a messaging system build on top of FairMQ(CITE) (which is in build on top of ZeroMQ(CITE) and nanomsg(CITE)). This messaging system can be used to share a single AOD between several tasks using shared memory or fetching data

from the nearest location. for example, the data can be located starting from shared memory, then looking at a different NUMA node, then looking at other connected nodes in memory or on disk, etc., all the way to tape-storage. The exact implementation of this mechanism and topology is independent of the AOD design and Analysis Framework save for two restrictions: the AOD design has to interact (efficiently) with the fairMQ message-passing framework and the framework has to let the messaging system be in charge of memory-allocation in order to be able to use shared memory.

2.2 Requirements

Before the start of the project a list of all necessary features was compiled, these included:

- **Pruning:** The ability to reduce an AOD to a smaller subset by removal of structures and/or components of structures. This is needed in order to allow the physicists to reduce a larger dataset to a much smaller one for subsequent (local) analysis and reduce the required IO- and memory-bandwidth at runtime. For example, the most commonly used object stored in an AOD file is a detector *track*. This is the reconstructed trail left by a particle in the detector. A single track contains many different types of information such as its point-of-origin, it's momentum, the covariance matrix of the reconstruction, references to other associated objects etc. In most cases however, one is not interested in all of these components. Sometimes one might only be interested in a single component of the momentum. Pruning is the act of getting rid of all other data and, in this case, leaving only a single component of the momentum.

- **Skimming:** The ability to infer certain facts about an AOD without having to touch all the data contained therein. For example one might only be interested in tracks which have a positive charge associated with them. Skimming would be the act of reading/copying only those tracks which fit this criteria. (TODO: write psuedo-code for these to use as an example, i.e. $if(charge <= 0)\{continue;\}$)

- **Growing:** The ability to easily extend an AOD with additional data, structures or components. For example, one might like to extend the track structure with a new field to cache the result of a long and complex computation, call our hypothetical field *'bool IsHiggsBosonCandidate'*, that would be an example of growing the track structure. Another option would be the adding of a whole new struct to an existing AOD, for example one might have a set of simulated (Monte-Carlo generated) collisions and wish to extend the AOD file to include the tracks created by this simulation (note that the Monte Carlo generator produces particles which do not interact with the detector so one simulates this interaction starting from the generated particles to create a simulation of raw detector data, after which the normal reconstruction algorithms can be run to create normal AOD data).

This list was later extended to include:

- **Derived Types:** Certain components are rarely used and/or take up a large amount of disk-space when stored and can be recomputed from other stored data if needed. It is therefore favorable if these could be recomputed transparently on a as-needed basis as opposed to being stored in-file. A common example would be so called *V0s*, these are secondary-vertices[1] resulting from the decay of a neutral particle. The V0s can be recomputed from the most common data and it is therefore possible to save on storage space by not including them in the AOD files, at the cost of having to recompute them before being able to use them. By leveraging shared memory and smart disk-caching this compute penalty can be kept to a minimum.

- **Zero-copy message parsing:** Previous work(CITE,Mikolaj?) has shown that a large performance penalty is incurred by data-serialization. In order to avoid this overhead the format should be able to pass data around without any (unecessary[2]) serialization. In practice this means avoiding the use of pointers inside data-structures and keeping any-and-all data of POD-type[3].

- **Maintain a similar interface:** The current ALICE software framework is used by X(CITE) users, contains over Y lines of code, and was written over the course of Z years. While it has been deemed possible, even necessary, to convert parts of the framework to work with the new format a dramatic change in the way physicists interface with the data would involve to much man-hours in terms of rewriting existing software and utilities and forcing every user to adapt to the new framework.

- **Schema Evolution:** On occasion, changes might be made to the way in which data is stored. This might be the conversion from a single float to a 3-dimensional vector or it might be a change in coordinate system to improve accuracy and/or compression. Changes of this kind are called Schema Evolution, and it is important that the framework knows how to handle this so that it can adapt and read both old- and new-formatted data.

In addition to these given requirements the overall guidelines are such that on-disk storage space should be made as small as possible, whereas in-memory data should aim to keep the amount of bandwidth required to a minimum as the current ratio of computing power to bandwidth heavily favors compute.

[1] primary-vertices being vertices generated from a collision at the hearth of the detector and secondary vertices being those formed by the decay of particles

[2] One cannot avoid (de)serialization when passing data between machines with different native data types, such as a big-endian and little-endian machine but that is out-of-scope for the AOD.

[3] POD, or plain-old-data, is a c++ term defined in (CITE). In practice it means that a struct or class cannot contain any virtual types or define data and/or functions in more than one part of its inheritance graph (including itself).

Chapter 3

Strengths and Flaws of the Existing implementation

The current implementation of AOD files is done using *ROOT*(CITE) objects. ROOT is a general-purpose (High-Energy) Physics framework extensively used throughout all of CERNs experiments. Among its features is the ability to define classes as ROOT objects, granting them features such as runtime-introspection, and store collections of ROOT objects in a general-purpose custom data format called a ROOT tree (TTree). The existing AOD format is defined as such a TTree and input and output operations are handled by ROOT.

3.1 Strengths

3.2 Weaknesses

Unfortunately, this approach does come with some downsides. Storing data in a TTree requires that the data is a ROOT object. In order to define a type as a ROOT object in c++ one has to inherit from the class *TObject*, and this causes problems. If a class derives from *TObject* it will inherit virtual functions, and in doing so it will no be a Plain-Old-Data-type (POD-type), that is, it will have a virtual pointer meaning it can't be shared or copied between different nodes without doing some form of serialization to get rid of and restore said pointer. In general, ROOT Trees where never designed to be shared across multiple nodes in a distributed fashion and once loaded from disk does not shy away from using pointers internally.

Another downside is that one is dependent on the features and optimizations found in the generic ROOT Tree. for example ROOT trees are optimized to handle files which are larger than the available memory of a system but there is little control over this feature preventing further optimizations. dependency By building on top of ROOT Trees it also introduces a where we are

finally, using the ROOT approach a collection of structs is defined and stored in a array-of-structs like fashion. As we will show later, this makes it hard to do efficient pruning as different struct members will be interleaved together.

(not flat for one, bottlenecked, full of references, dead data from deep inheritance)

Chapter 4

The New AOD Format

4.1 Design

4.1.1 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* $\ldots >$**:** 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.

4.1.2 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.

4.2 Features

4.2.1 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).

4.2.2 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.

4.2.3 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.

4.2.4 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?)

4.2.5 Schema Evolution

Discuss, complicated.

4.2.6 Interface

Finish EC part first

4.2.7 Vectorization

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

4.2.8 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.

4.3 Usage

show don't tell, add example

4.3.1 Event-building

do tell here, show cost, different methods.

4.3.2 Writing Analysis Tasks

show don't tell

4.4 Benchmarks

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

4.5 Future Work

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

Appendices

Appendix A

Acknowledgements

1 page

Appendix B

Glossary