Contents

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

General overview here, comparison between the different data types (Raw, CTF, ESDs, AOD) produced by the experiment and why they are needed.

2.1 AOD Production

How AODs are generated in the pipeline. Run3.

2.2 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 Grid (Note: Grid or 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. Multiple tasks, by possibly multiple users, are scheduled together in a so-called *Analysis Train*, this groups (sub-)tasks together based on the accessed files thereby reducing file IO and data movement.

Each sub-task runs as it's own separate process. The scheduling of sub-tasks is fully automated and transparent to the task. 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 and nanomsg(TODO: check, CITE)). This can be leveraged to share a single in-memory AOD between several consumers.

2.3 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 it *'HiggsBosonCandidate'*, that would be 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 simulated tracks. (TODO: write better good)

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.

- **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[1]) serialization

[1] One cannot avoid (de)serialization when passing data between machines with different

- **Maintain a similar interface:** The current ALICE software framework is used by X(CITE) users, contains Y(CITE) lines of code, and was written over the course of Z(CITE) years. While it has been deemed possible, even neccesary, 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 rewritting 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.

native data types, such as a big-endian and little-endian machine but that is out-of-scope for the AOD.

Chapter 3

Strengths and Flaws of the Existing implementation

General overview here, ROOT objects, AoS, etc. Note the dfferences in run 3 and 2.

3.1 Strengths

3.2 Weaknesses

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

Chapter 4

The New AOD Format

4.1 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

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

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 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?

4.1.1 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

4.2 Features

4.2.1 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.

4.2.2 Skimming

4.2.3 Growing

4.2.4 Derived Types

4.2.5 Schema Evolution

4.2.6 Interface

4.2.7 Vectorization

4.2.8 On Concurrency

4.3 Usage

4.3.1 Event-building

4.3.2 Writing Analysis Tasks

4.4 Benchmarks

4.5 Future Work

Appendices

Appendix A

Acknowledgements

1 page

Appendix B

Glossary