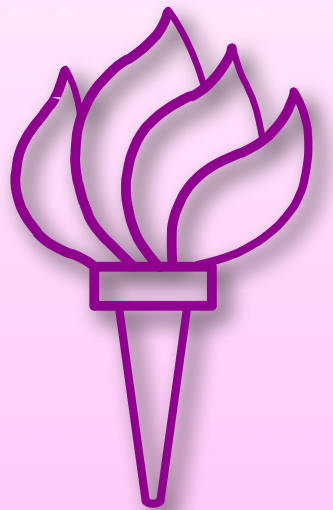
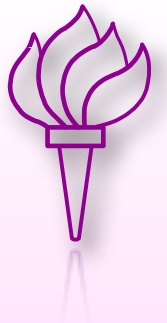
D3PD Status/ Updates



Attila Krasznahorkay

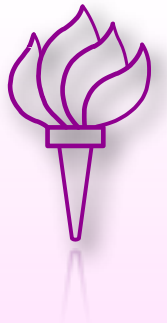
NEW YORK UNIVERSITY

Overview



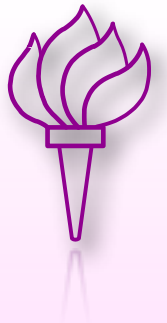
- D3PDMaker software status
  - Analysis cache status
  - Transformation job configuration
  - Tag upload procedure
- D3PDReader software status
  - General idea behind the code
  - Current features
  - Feature requests
- Ongoing developments
  - D3PD <–> TAG integration

Analysis Caches



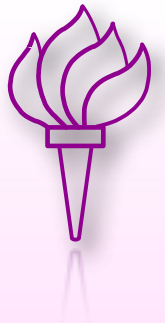
- Status of the releases intended for current analyses:
  - **AtlasProduction-16.0.X.Y**: Was kept mostly up to date with the tags not violating frozen T0 policy, swept from the TopPhys cache. Was not used for physics analyses on release 16 data.
  - **TopPhys-16.0.X.Y.Z**: Full support for the release has been dropped recently. Would've required too much effort to back-port all developments from 16.6.X.Y. Still the recommended release for data10 analysis with slightly older tags than 16.6.X.Y.
  - **AtlasProduction-16.6.X.Y**: Kept fully up to date with the newest version of all the D3PDMaker packages. No problems with frozen T0 yet.
  - **AtlasAnalysis-17.0.0**: Newest tags regularly collected/swept into it. Usually without performing functionality tests.
  - **16.6.X.Y.Z caches**: Not under PAT management. Usually special tags only for group specific D3PDMaker packages.

Truth dumper status



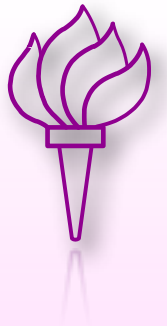
- Much work recently on unifying the way MC information is dumped by various groups
- Collected the truth handling tools from the various packages into `TruthD3PDAnalysis` and `TruthD3PDMaker`. (Very group specific tools did stay in the respective packages.)
- Dumping full MC information works quite nicely already.
- Writing analysis specific information is not solved yet completely. Usually quite special selections required.
  - Did start to come up with a framework for this though…

`Reco_trf.py` job configuration



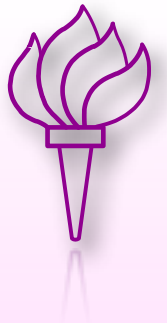
- Starting with NTUP_EGAMMA and NTUP_SUSY we started migrating the configuration of jobs to resemble the configuration of DESD/DAOD jobs
  - The D3PD developer creates a job0 fragment that sets up that particular D3PD output (e.g. SUSYD3PDMaker/SUSYD3PD_prodJobOFragment.py)
  - Have to make sure that the separate job0 fragments don't break each other
  - Will have to make sure they have **no** effect on each other (not yet the case)
- Simplifies the task of creating DPD trains

Code validation tightening



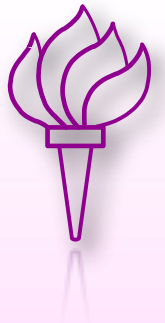
- D3PDMaker code quite entangled
  - High level packages all use the same low level ones, often relying on the internal structure of the D3PDObject configurables
  - An unfortunate change in one of the core packages can easily break multiple D3PD output formats
- Created a new AMI configuration (q125) that exercises all production D3PD formats on a data AOD (TWiki)
  - Dynamically updated when new production D3PDs are created
  - After a bigger change the developers have to demonstrate that the q125 test still succeeds
    - If not, they have to update all dependent packages
  - Can't introduce new ERROR messages to the jobs
- Very good first step, but more will be needed…
  - One test running on a MC AOD (q126)
  - Maybe a test on data ESD? (Performance D3PDs still need ESDs.)

D3PDReader goals



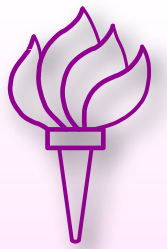
- Provide a general skeleton for the physics groups that makes it easy to read D3PD files in a performant way
  - D3PD datasets are becoming larger and larger, it becomes important to process them with the highest speed possible
  - Code has to work in any established C++ analysis framework, users don't have to start from scratch when using it
- Provide an interface for common tools for handling information coming from D3PDs
  - But don't develop a "mini-EDM", as those have never worked out (maintenance…)

D3PDReader implementation



- Written as a code generator, similar to TTree::MakeClass (TWiki)
  - The generator code sits in a CMT package (PhysicsAnalysis/D3PDMaker/D3PDMakerReader) to give it easy access to some libraries needed for the generation –> You need the offline SW environment for code generation
  - The generated code only depends on ROOT however
- Doesn't provide an event loop!
  - The generated classes only take care of accessing the information of a given TTree entry –> Can be integrated into practically any analysis code.
- Can be copied into existing analysis projects, or compiled as a standalone library (helper scripts provided)

D3PDReader usage (1)



- Not really documented at the moment, will have to improve on it...
- Simple example exists in the generator package (D3PDMakerReader/test/test.cxx)
  - The user has to access the D3PD TTree by hand

```
TFile* ifile = TFile::Open( “d3pd.root”, “READ” );
TTree* itree = ( TTree* ) ifile->Get( “d3pd” );
```

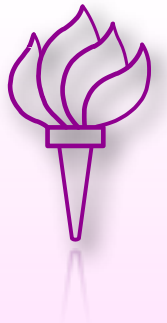
  - Tell the reader objects that they should use this TTree

```
Long64_t entry = 0; ///< Variable specifying which event to look at
D3PDReader::EventInfoD3PDObject evinfo( entry );
evinfo.ReadFrom( itree );
```

  - Run the event loop by hand

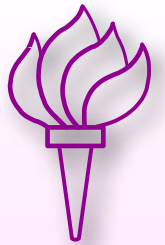
```
for( entry = 0 ; entry < itree->GetEntries(); ++entry ) {
   std::cout << “Processing event: ” << evinfo.EventNumber() << std::cout;
}
```

D3PDReader usage (2)



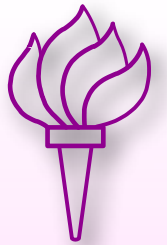
- Can be used very easily from a MakeClass class or a TSelector based class
  - However not much point in using it in MakeClass. Can use a standalone C++ function with that much effort.
  - No simple example “on the market”, but the implementation should hopefully be obvious from the standalone example
- Used extensively in SFrame these days
  - The already quite good analysis codes achieved a minimum of a 2x speed increase
  - Using it mostly with PROOF in SFrame analysis jobs

D3PDReader feature requests



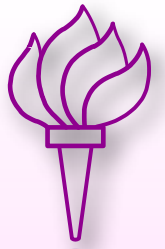
- Request for adding some built-in monitoring for the classes
  - Since branches are loaded only on demand, can be used to see which branches are used in various analyses, how much data is read from them per event, etc.
  - Monitoring code will only be compiled on demand (through pre-compile macro probably), as it will probably be too much of a performance loss for regular analysis jobs
- Would like to make them usable from Python. Not impossible, but will require a bit of re-design. (In own test achieved a ~100x speed increase over a simple PyROOT script.)

D3PD <-> TAG



- Tools exist for translating the properties of all major (and many hardly used) objects into primitive (D3PD) variables
  - Performance groups heavily involved with maintaining these tools
  - Performance group representatives already familiar with the D3PDMaker framework
  - Group involvement in TAG making code much lower
- We could re-use some of the D3PDMaker “filler” tools to fill the properties of offline objects into TAG
  - If the performance group provides a way of filtering the “good” D3PD objects, we can make use of that as well

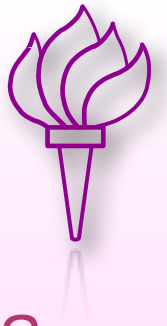
D3PD <-> TAG



- Needs some extension to the core D3PDMaker code
  - Scott created the interfaces such that none of the actual tools will have to change
  - Will try to implement it myself fairly soon
- Then we will just need to replace (some of) the existing TAG builder algorithms with their D3PDMaker counterparts
- Storing D3PD event selection results in TAG?
  - Could we extract information about the event selection results from ProdSys group productions?
    - Check overlaps between D3PD types
    - "I need events with XXX and YYY. Which D3PDs contain these events?"

Already covered by Amir...

Summary



- D3PDMaker operating in very stable mode since a while
- Some modifications needed in order to safely merge D3PD production jobs into trains, but we're almost there
- D3PDReader code already used by some adventurous groups
  - Very positive feedback so far
  - But the current users have in general good skills with C++...
- Starting the actual work with the D3PD <-> TAG cooperation