Report of the DØ Data Format Working Group

F. Déliot, H. Greenlee, S. Kulik,
A. Lyon, S. Protopopescu, G. Watts

June 22, 2004

Contents

1 Introduction and Charge

The full text of the charge to the DØ Data Format Working Group can be found in Ref. [1]. The charge contains the following two main points:

- Review currently available root-based data formats and associated analysis algorithms. Understand the rationales, pros and cons of each data format.

- Develop and implement a root-based data format incorporating desirable features of existing root-based formats and analysis tools, taking into account the needs of algorithm developments and physics analyses as well as the computing resources (storage, access time, how it scales with large dataset etc.) required for analyses.

This report constitutes the fulfillment of item one of the charge.

The original charge specified deadlines of May 1, 2004 for a plan (i.e. this report) and June 30, 2004 for complete implementation. The May 1 deadline was missed, obviously, and the June 30 deadline does not seem possible either. We think that the real, hard deadline is to have the common root format ready to go into production at about the time p17 d0reco is going into production, whenever that is.

Here then is a list of basic requirements for a common analysis format.

1. Content: Inclusive enough to support most DØ analyses.

2. Size: Events should be as small as possible. We have not been given a a firm target size, but p14 thumbnails are about about 20 kB/event for data. We note that in p17 the dst and tmb data tiers will likely be retired in favor of the so-called tmb++ data tier, which is an enlarged version of the current thumbnail. There is a need for a smaller "microdst" data tier that is smaller than tmb++, which can be kept on disk.

3. Read speed: There is a need for a data format that can be read much more quickly than thumbnails.

4. Development cycle: There is a need for a data format that has a rapid development cycle as compared to the DØ framework.

More detailed formal requirements can be found later in this document. But we regard the above points as the overriding goal of the common analysis format effort. We note that the charge specifies that the common analysis format be developed using root. However, if it were possible to fulfill all of the above requirements using a non-root format, such a format would be worth considering too.

There are other potential benefits of developing a "common analysis format," besides the ones listed above. Here are some:

- Reduce the software development and maintenence effort of physics groups and analyzers by providing a centrally maintained analysis format and program.
- Facilitate the sharing of data and analysis algorithms among analysis groups.
- Reduce confusion among analyzers (i.e. analyzers will know that the DØ experiment is committed to maintaining the standard root format for the lifetime of the experiment).

Even if no other benefits were realized except the main four listed above, the common analysis format would still be a success.

1.1 Common Format Root Tuple Use Cases

Here are some possible use cases involving new common format root tuples. In these scenarios, it is assumed that centrally produced root tuples exist and are accessible through sam.

1. Analyzer has a working analysis that runs off thumbnails using the framework and doesn't want to change anything. It will still be possible to work from thumbnails. However, tmb reading speed, which is currently of order 10 events/sec. for the tmb+ format (see Secs. 4, 5.2) will get slower due to the larger tmb++ format. Also tmb++ files probably will not be on disk.
2. Analyzer has a working analysis that runs off thumbnails, which are used to make some custom-format root tuples. Analyzer modifies his tuple-maker to run from common format root-tuples instead of thumbnails. The interactive part of his analysis doesn't change. The benefit to the analyzer is faster I/O for the batch part of the analysis.

3. Analyzer needs to use a custom-format root tuple for the interactive part of his analysis. Analyzer writes a program to read common format root tuples (based on standard example), and makes his own tuples.

4. Analyzer wants to use centrally-produced root tuples to produce final histograms. Common root tuples may not be accessible interactively (i.e. maybe only through sam).

5. Analyzer has a dataset that is a small fraction of a CSG skim. Analyzer writes a skimming/tuple-making program (based on standard example) to run over centrally-produced root tuple, writing selected events in the common tuple format to his local disk.

6. Analyzer has a large dataset, but needs only a small fraction of the data from each event in the common format root tuple. Analyzer configures his skimming program to only read in and save only the required branches on his local disk.

7. Analyzer needs results of some special algorithms or processing in his tuple that are too costly to run interactively. Analyzer reads centrally produced root tuples and adds a branch (using standard procedure), and saves the resulting tuples to his local disk.

8. Analyzer needs data that are in tmb++, but not in centrally produced root tuples, but which have software support in the common tuple maker (but which are turned off by default). Analyzer runs the common tuple-maker from tmb++, turning on the required branches in rcp, and saving the result on disk.

9. Analyzer needs data that are in tmb++, and which lack software support in the common tuple maker. Analyzer can a) do analysis directly from tmb++, b) read data from tmb++, use standard tuple-maker to fill standard branches, and adds his own branch(es).

It is in DØ's interest that most analyzers eventually switch to using centrally produced root tuples as their main input, whether or not they use the common tuple format for the interactive part of their analysis.

2 DØ Data Tiers and Processing Chain

This section summarizes how data is transformed from raw data to physics results (plots, etc.).

2.1 Reconstruction

Reconstruction is the first step in the DØ data processing chain (after recording). The DØ reconstruction program (d0reco) converts raw data into reconstructed data (dsts and thumbnails). Raw data, dst, and thumbnail are all stored as d0om files (see Sec. 4).

The reconstruction step is done on a reconstruction farm. Reconstruction is the most costly step in the data processing chain in terms of cpu time. In the case of data, this step can only be done easily at Fermilab due to the requirement of having fast access to large calibration databases.

2.1.1 Dsts and Thumbnails

From the start, all versions of d0reco have produced a data tier called the dst consisting of edm chunks containing the results the various reconstruction algorithms. Beginning with production release p11, a new data tier called the thumbnail (or tmb) was introduced. The thumbnail format contains a subset of the data in the dst, with most data being packed into a single chunk (the `ThumbNailChunk`), plus a few other chunks. The thumbnail format quickly grew into the so-called tmb+ format with the inclusion of `CalDataChunk`, allowing rereconstruction of calorimeter data from the thumbnail. Beginning with production release p17, it is intended that the thumbnail will undergo further expansion into the so-called tmb++ format with the inclusion of preshower and track cluster chunks, allowing nearly full rereconstruction from the thumbnail, without the need of accessing calibration data. At that time, routine production and storage of dsts on the farm by d0reco will probably cease.

2.2 Post-tmb Processing — The Common Sample Group

Eventually, the necessity of a post-tmb processing step became evident. The Common Sample Group was formed to manage post-tmb processing. Three

types of standard post-tmb processing are currently being used in DØ: fixing, skimming, and standard object corrections, each of which are described below.

2.2.1 Fixing

In order to get the latest vertexing, the latest calorimeter corrections and bug fixes, DØ performed the fixing of its dataset reconstructed with the p14 release versions of d0reco. Fixing is a way of re-reconstructing events starting from thumbnail alternatively from running d0reco again. After applying corrections to the CalDataChunk, all the reconstruction code except tracking and muon finding is rerun. Different versions of d0reco have slightly different imperfections which have been fixed by different configurations and/or versions of the fixing program (`TMBFixer`). This fixing also led to a more uniform dataset among the data processed with the following release versions of d0reco: p14.03, p14.05.00, p14.05.02 and p14.06.00.

Two fixing passes were performed. The first one (pass-1) aimed to correct the vertex reconstruction and to fix calorimeter hardware problems. It was necessary to perform it on p14.03, p14.05.00 reconstructed data and data reconstructed with p14.05.02 from dst at remote farms. The second pass (pass-2) will be run on all p14 dataset (prior to p14.07.00) to apply the T42 algorithm [4] in killing mode for calorimeter noise suppression and the latest noise cell killer. It will also fix the latest cps and vertex bugs (for p14.03 and p14.05.00 data).

The fixing was run centrally by the Common Sample Group [2] on CAB [3] (pass-1) starting from thumbnails out of d0reco and writing out new corrected thumbnails back into SAM. For pass-2, investigations to run the fixer also on the UTA farm are underway.

Fixing pass-1 has been run from October 2003 to January 2004. Here are the steps involved in `TMBfixer` pass-1:

- read a thumbnail file
- recreate a CalDataChunk corrected for:

  Shared energy problems which are not fixed in p14 [5]

  Tower 2 problem [6]

  Calorimeter cable swap

  Massless Gap x2 bug

- reconstruct primary vertices (2-pass vertexing) [7]
- fix ChargedParticle extrapolation to cps and association with vertices
- reconstruct taus, jets, EM objects, missing ET and muon links
- make a new ThumbNailChunk
- write out a new thumbnail file

Pass-1 fixed in total 388.8 million events (174.7 million events for p14.03, 123.5 for p14.05.00 and 90.1 for p14.05.02 from dst). Pass-1 used between 100 and 150 CPU slots on CAB (out of 340) and the typical speed of `TMBfixer` pass-1 was 1 s/event. In this process, the fixing process has reached a maximum global efficiency of 85% where the efficiency is here defined as the number of processed event over the number of events that should be processed (taking 1s/event with 100 to 150 CPU).

2.2.2 Skimming

The goal of skimming is to reduce the dataset to be processed in each analysis. To avoid work duplication, it is done centrally by the Common Sample Group [2]. The resulting thumbnail skims are stored in SAM.

For the p14 dataset used for winter conferences 2004, 524.3 million events have been processed and 23 skims have been written out (see Common Sample Group web page for details [2]). The largest skims were the bMU skim representing 32.8% of the dataset (3678 GB) and the 1EMloose skim with 14.4% of the dataset (1746 GB).

The skimming code runs at a speed of approximatively 0.3s/event.

At the same time as physical streams are written out, a bunch event flag is set in the evpack event header of each processed events [9]. Those event flags allow users to filter events very quickly according to those flags because events don't need to be unpacked to be selected. The goal is to have the event flags run just after d0reco in p17 release versions.

2.2.3 Common Object Corrections: d0correct

The standard correction package d0correct is aimed at gathering all the post-reconstruction object corrections as well as object certification cuts in a coherent and well-versioned way. This package doesn't contain any code (except the one to remove duplicate events which appeared in the skims after the

skimming process). Package d0correct calls for the post processing method for each object, currently muons, electrons, jets and missing energy.

Here are a rough description of what is involved in this post processing:

- Muons: remove duplicate muons if any, set muon quality criteria according to the certification (loose, medium and tight muons), add useful variables for analysis
- EM objects: apply EM energy scale, latest geometrical corrections and Hmatrix calculations, apply cuts to define good EM objects
- Jets: apply jet energy scale, apply certified cuts to remove bad jets, remove electron like jets
- Missing energy: compute missing transverse energy taking into account certified electrons, jets and muons, apply energy scale corrections

Package d0correct doesn't contain any executable. It exists as a framework package that is linked into various executables. It is run before the different root output makers. Currently it is linked with tmb_analyze, Athena, wz_analyze and AADST (only MuoCandidate). It works on thumbnail, reads the uncorrected object chunks and recreates new corrected chunks. Currently the possibility to write those corrected chunks into a new corrected thumbnail exists for all chunks except the jet chunks.

D0correct version v00-00-06-5 runs at 0.1s/event.

2.3 Analysis Formats

The d0om data format (thumbnail in particular) is not, and was never intended to be, a data format optimized for data analysis. At the beginning of Run II, DØ decided to standardize on edm and d0om for reconstruction and archival data storage, and root for data analysis. Although there were, and are, people who lament the fact that the archival and analysis universes could not be merged into a single data format, we do not propose to change that at this late date. The group charge specifically recognizes the validity of the dual format model. The vast majority of people who are doing analysis in DØ currently are converting thumbnails into one or another type of root tree or tuple (we use the terms "tuple" and "tree" interchangeably in this document, and indeed, root does not distinguish between the two formats).

The following is a list of data formats that were specifically mentioned by physics conveners:

- thumbnail
- tmb_tree.
- top_tree.
- Athena.
- AADST.
- wz_analyze.
- qcd_analyze.
- Diffractive root tuple (extension of qcd_analyze)
- higgs_skim.
- higgs_multijet.

In addition, there are individuals who are maintaining their own tuple formats, not mentioned in the above list.

Most of the above formats are root tuples. The only exceptions are AADST, which is its own binary format, and of course thumbnails. Some of these data formats are described in more detail below.

2.4 D0root

This section contains a brief description of the d0root software package. This section is included because d0root is in some sense part of the DØ analysis chain due to the fact that some important algorithms are only available in d0root (such as b-tagging).

The d0root package is not an analysis format, since it is not tied to any one format, but can be used with several different formats (currently, tmb, tmb_tree, and top_tree). d0root is a toolkit which consists of an object model for physics objects (e.g. jets, tracks, vertices), and high-level algorithms that run on these objects. To make use of any d0root algorithm, it is first necessary to import data into d0root's object classes.

D0root includes algorithms for primary and secondary vertexing, and b jet tagging. In some cases (primary vertexing), the d0root versions of these algorithms exist in competition with corresponding algorithms that run in d0reco. In other cases (secondary vertexing and b-id), these algorithms exist only in the d0root universe.

D0root was invented to allow algorithm development to take place in root (as compiled root macros). The algorithms themselve are implemented as ordinary c++ classes which are dependent on a limited set of root utility classes. The d0root algorithms can be called from any c++ program, including standard DØ framework programs. Furthermore, as noted above, standard ways have been developed for importing edm objects into the d0root object model.

Nevertheless, the d0root way of doing things has been both controversial and somewhat problematic for several reasons.

- *Maintainability.* The translators, the code that moves data from the TMB, top_tree, or tmb_tree format to d0root, must be maintained. Frequent upgrades and active development mean that people must be devoted to this task over the long term. Further, infrastructure must exist to compare the various format outputs to assure the results are the same.

- *Inability to Write Results to edm.* Currently, when running on the TMB, there is no code that will save the results back to edm. For example, given that a bID algorithm has been run, it would be advantagous if its results could be written to a edm chunk. While this makes no sense when running on the top_tree or the tmb_tree, this is capability should be added for various important algorithms like bID. Authors of d0root claim this isn't technically difficult; just a matter of finding an interested party with the time.

- *Algorithms are difficult to configure.* As with any complex algorithm, the bID algorithms have a large number of input parameters. d0root currently has no general mechanism to handle this. The equivalent in the framework package world is the combination of the RCP files and the framework constructors. While for bID algorithms work is in progress to mitigate this (the btags_cert package, which has RCP like parameter setting and a uniform interface), there is no general solution.

2.5 PAX

PAX (Physics Analysis eXpert) is a physics analysis toolkit that has been developed by soon-to-be DØ collaborator Martin Erdmann and collaborators at Karlsruhe University [8]. PAX maintains a high-level event model consisting of particles (4-vectors), vertices (3-vectors), and the relationships among these objects. In this sense, PAX bears some resemblence to d0root. PAX allows different "interpretations" of the same event to be in memory at the same time. These different interpretations might contain different types of data (e.g. a calorimeter-based view vs. a track-based view of the same event), or they might correspond to alternative event hypotheses (e.g. different associations of jets to top quarks). PAX includes hooks for implementing analysis algorithms in terms of one or more interpretations, possibly generating new interpretations.

PAX's objects are simple, by design. In order to maintain a connection to more detailed, and detector specific, information the relationship mechanism is again used. In this sense PAX would be used in conjunction with an existing data format like TMB or any of the root based formats.

PAX is built on top of either root or another toolkit, from which it gets utility classes (e.g. TLorentzVector), graphics, and object persistence. PAX is currently being used (by some) in CDF and CMS.

We think that PAX looks interesting, especially since the main author is joining DØ. It would be worthwhile to have PAX available to DØ as a DØ or FNAL product for people who want to use it.

3 Root

Early on, DØ decided to standardize on the root software package for data analysis in Run II. Root is endorsed and supported by Fermilab, and is widely accepted by High Energy Physics experiments and laboratories worldwide. There is little reason to think that the decision by DØ to rely on root for analysis was incorrect or should be revisited.

Root itself is a multifaceted software package which supplies software with the following features:

- Data visualization (histograms, plots).
- Data analysis (statistics, fitting).

- Presentation graphics.
- Graphical user interface.
- Object persistence.
- Tuples (trees).
- Utilities (collections, physics vectors).
- Scripting (cint).

In root, tuples or trees are represented by class `TTree`. A TTree can be viewed as an ordered collection of events containing objects of the same type. Each event of a root tree consists of a collection of branches (class `TBranch`). Each branch can contain either a flat list of simple variables (scalars or fixed- or variable-length arrays of atomic types) or a single class. A tree that consists only list-type branches is conceptually identical to a paw column-wise ntuple. Indeed, this type of tree was invented to provide a compatibility mode with paw. The list-type tuple has the advantage that a tuple can be defined programmatically, i.e. there is no need to have a class header available at compile time, or to have a class definition in root's class dictionary. The class-type branch is more in keeping with object-oriented programming philosophy, with the benefits and costs that go with this programming style.

There is little difference in the physical data format or performance between list-type branches and class-type branches. That is, the choice between the two types of branches is a mainly a question of the programming interface.

3.1 MakeClass

One of the easiest and most popular ways to read a root tuple is to use root's TTree::MakeClass function to generate a skeleton macro tailored for a given tuple. MakeClass generates a single header and a single implementation file which contain a struct that can hold the tuple data for one event, and a skeleton event loop. MakeClass has the advantage that the files that it generates are completely self-contained. No configuration is required. The user does not have to know anything about how the tuple was originally genated. In principle, MakeClass can work for either list-type of class-type tuples. In

either case, the tuple data is flattened into a single struct containing basic types and (fixed length) c arrays.

MakeClass does have its problems, however. It sometimes generates code that doesn't compile (even if it can be interpreted by cint). Since MakeClass does not use dynamic collections, it is forced to guess the maximum size for the c arrays that it generates. It often guesses wrong. MakeClass puts all of the variables in the tuple together into a single struct, rather than putting variables from separate branches in separate structs. With MakeClass, the definition of the tuple data is not well-separated from methods that the user may need to supply or modify. MakeClass has no provision for schema-evolution. That is, the skeleton macro generated by MakeClass will typically only work for the tuple that it was generated from, but not for other versions.

There are many ways that MakeClass could be improved. For example, it could generate multiple structs, or even the original class structure. It could use dynamic collections instead of fixed-size arrays. The root developers are well aware of the limitations of MakeClass, and are in fact planning on releasing an improved version. An improved version of MakeClass would be welcome.

The alternative to using MakeClass is for users to write macros based on headers and implementations provided by the tuple author. This approach has the advantage of maintaining the original class structure of the tuple data. The main disadvantage is this approach is that the user is confronted with a more complex configuration problem than MakeClass. The user is typically required to setup the header include path and may be required to load shared libraries, etc.

3.2 Branch Splitting

Branch splitting means that branch elements (leaves) are buffered to disk separately. When reading one has the option of reading certain branches, or leaves, and not others (selective reading). Root is quite flexible in defining how to split branches, but by default branches are split to the maximum possible depth (i.e. all the way down to the leaf level).

There is the potential for large improvements in reading speed by reading branches selectively. Unfortunately, selective reading is not particularly easy to use, as the user is required to laboriously turn on or off individual branches by hand. The number of branches and subbranches can be quite large. Furthermore, selective reading is bug-prone. In a complicated macro,

it may not be easy to determine which branches are actually used. There is usually no indication that a required branch has not been read, except incorrect results. In this sense, root is less advanced than paw, which read ntuple columns selectively automatically. Selective reading may become automatic in a future version of root.

4 D0om

D0om is DØ's c++ object persistence mechanism. DØ uses d0om to store raw and reconstructed event data. D0om is documented elsewhere [10]. This section is included as a brief introduction to d0om.

One of the main features of d0om is the fact that it is divided into a front end and a back end. The front end consists of the object model and I/O classes (in short, the programming interface). The front end has nothing to do with the physical format of persistent data. The d0om back end is responsible for object I/O and is (mostly) invisible to the programmer. The back end determines the physical format of d0om files.

The back ends that are in use for storing DØ event data are based on the dspack software package. Dspack was originally written in fortran as a memory management and I/O package (somewhat similar to zebra, for those who remember Run I). Dspack was chosen as the back end because it has several desirable features. Dspack incorporates an object model that maps naturally onto c++ objects (including things like dynamic collections and pointers). Dspack also has the built-in ability to read and write the objects that it manages into flat files.

Most data in DØ is currently stored in the evpack format using the evpack d0om back end. Evpack is a derivative of dspack that stores standard dspack physical records inside its own record structure. As compared to dspack, evpack adds various features, including compression, random access, and fast skimming based on event flags [9].

One undesirable feature of either back end (dspack or evpack) is the fact that any I/O necessarily involves double-copying of data, first between c++ objects and dspack-managed objects, and second between dspack memory and flat records. When reading, d0om has the ability to defer the conversion of some dspack objects to c++ objects. But dspack itself has no ability to defer construction of dspack objects or to read only part of the data in an event (with the limited exception of evpack's fast-skimming feature).

Table 1: Breakdown of cpu usage for reading thumbnails.

Step	CPU time (ms)	Fraction (%)
Read data into DSPACK objects	3.6	4
Convert data into c++ objects	7.7	8
Unpack ThumbNailChunk	85.0	88
Total	96.3	100

Finally, it must be noted that when reading thumbnails there is an additional time that is needed for unpacking. Thumbnail unpacking is not part of d0om proper, but must be considered as part of the cost of reading thumbnails. Table 1 gives the breakdown of cpu usage in t04.04.00 into the three main components of dspack I/O, c++ object conversion, and unpacking[1]

Observe that Thumbnail reading is dominated by thumbnail unpacking and not d0om I/O per se. The unpacking time in Table 1 is specifically the time used by framework package UnpThumbNailPkg. It does not include bit-unpacking (e.g. converting short fixed point numbers back to floating point), which is included in the 8% d0om c++ object instantiation overhead. The unpacking time used by UnpThumbNailPkg is the time necessary to convert thumbnail physics objects into dst physics objects and chunks.

Recent profiling studies by Scott Snyder indicate that that thumbnail unpacking is dominated by a single (fixable) hot spot in the jet unpacking. If jet unpacking is turned off (rcp parameter d0Jet=false), the unpacking time is reduced from 85.0 msec to 21.8 msec. The next hot spot is "charged particle extrapolation," which does some significant relcalculations involving charged particles and vertices. If charged particle extrapolation is turned off (rcp parameter d0ChXtrap = false), the unpacking time goes down to 10.9 msec per event. There are no more significant hot spots after this. With jet unpacking and charged particle extrapolation turned off, the total cpu time to read one event is about 3.6 msec (dspack) + 7.7 msec (d0om) + 10.9 msec (unpacking) = 22.2 msec. This is about a factor of five improvement over thumbnail reading in its current default configuration, which is about as good as thumbnail reading is ever likely to get.

[1]Times were measured on dragon-clued0, a 1.6 GHz Athlon

5 Current Analysis Formats

5.1 Overview

5.1.1 Reco_analyze

Historically, reco_analyze was the first root-tuple to be supported by DØ. Reco_analyze was an open format, meaning that each d0reco algorithm had complete freedom to define its own branch in the reco_analyze tuple, which it would register with a central tuple manager. Since reco_analyze came early, these algorithm branches were often geared toward debugging and algorithm development rather than physics analysis, although people did use them for physics analysis too.

The main disadvantage of the reco_analyze tuple was that it grew up organically to meet the needs of algorithm developers, with no one in charge of its overall design. As a result of its development history, it became too large to be seriously considered as a "microdst" format for physics analysis.

In d0library terms, reco_analyze had its own cvs package, which did not contain any code, but which supplied a top-level rcp with links to various algorithm rcps, plus a prelinked executable.

Operationally, reco_analyze was a list-of-variables (paw-type) type of tuple. The reco_analyze tuple was written through the `HepTuple` interface, and could be stored either as a paw- or root-tuple. This meant that any root-specific features of tuples had to be avoided. One oddity of the `HepTuple` interface was that variables had to be unique among all branches, which sometimes caused problems. Certainly, such an approach (i.e. using `HepTuple`) would not be recommended today.

Analyzers accessed the reco_analyze tuple exclusively via the MakeClass method. The header file generated by MakeClass was large and unwieldy, and it took a long time to compile.

5.1.2 TMB Tree Root Format

The tmb tree root format shadows closely the thumbnail container classes, with usually a one to one correspondence between a DØ physics object class and a TMB root class. For historical reasons, having mainly to do with the state of root at the time the project was started, there are structural differences between the way the data is organized in the thumbnail and in

the tmb tree. The tmb root tree follows closely the edm chunk structure, except there is no concept of chunk, instead:

1. Every object inside a chunk has a tmb tree object as a counterpart
2. If a chunk has information in only one container object (or is the only container object) its counterpart is a branch with one object.
3. There is only one branch containing all objects of a certain type. Same type objects coming from different chunks carry an algorithm name.
4. If there is more than one type of object in a chunk each type ends up in a separate branch.
5. Vectors or lists are replaced by *TClonesArrays*.
6. LinkIndex is replaced by *TRef*

Chunk classes are mapped into one or more root branches. A choice was made to use TClonesArrays to store collections of objects. This is encouraged in root instead of STL containers for efficiency and speed of access. Typically there is one branch with one TClonesArray containing all objects of a given type, so for example all thumbnail Jet objects have a corresponding tmb tree TMBJets object stored in one TClonesArray and that TClonesArray is the root tree branch Jets. In the thumbnail the jets produced by one reconstruction algorithm are in a separate instance of a JetChunk. In root tmb tree all jets are in the same TClonesArray and each TMBJets carries an algorithm identifier. This same structure is used for the other objects produced by multiple reconstruction algorithms, such as TMBVrts (Vertex) and EMparticle (TMBEmcl).

To save space some DØ physics objects have been split, reflecting more closely how the data is actually stored in the thumbnail: ChargedParticles have the basic information in TMBTrks, additional information for isolated tracks is stored in TMBIsoTrks (every TMBTrks has a valid TRef to a TMBIsoTrks if it is an isolated track). All EMparticle information is in TMBEmcl except the associated calorimeter cells. All calorimeter cells used in EMparticles are stored in TMBEmcells and the TMBEmcl has a TRefArray with TRef's to the associated TMBEmcells. Note that TRef's (unlike LinkIndex) are not templates and thus are not as safe to use —one cannot explicitly tell what kind of object a TRef points to. Therefore, whenever

possible, classes have methods that return a pointer, rather than a TRef. A TRef is meant for internal storage only.

Just like thumbnail physics objects inherit D0PhysObj so tmb tree root objects inherit TPhysObj. This enforces uniformity in access methods for quantities that are equivalent in each physics object (momenta, phi, eta, etc.). There has been some effort to have the same name for methods retrieving the same information from thumbnail objects and tmb tree objects but this has not been strictly enforced. In some cases information from thumbnail objects is given back as an instance of a DØ class. It was consciously decided, though, to minimize the coupling between the tmb tree root classes and d0library classes so that analyzers would need only a very small subset of the d0library. This not only makes it easy for analyzers to copy the necessary code to local platforms but also speeds considerably the compilation and linking of macros with shared object libraries. It would be possible for the root tmb tree classes to use many of utility DØ classes if they were kept in a few utility libraries decoupled from edm/d0om/rcp.

One aim of the tmb tree root file was to have all the information available in the thumbnail file. However, in the intervening time the thumbnail has been expanding to include enough information to redo much of the event reconstruction. The tmb root tree did not expand accordingly, there are no tmb tree classes to store calorimeter cell or tower information (with the exception of TMBEmcells), nor classes to store track hits. It is straightforward to code tmb tree classes handling such information but it would lead to an increase of event size by a factor of 5 to 10. It is unlikely that analyzers will want to pay the price to have that data available in root tree format for every event, but may want to have the option of having that information for some events. We should note that the tools do not exist to access detector geometry information with root so the utility of calorimeter cells and track hits outside a DØ framework program is limited.

Tables 2 and 3 list the tmb tree classes and corresponding edm classes.

The branch names are the same as the class but dropping the prefix TMB. Whenever there is more than one instance of a class the instances are in a TClonesArray. The table do not include the bcjet classes as the d0reco bc classes are being abandoned.

The tmb tree root files are generated with the **TMBAnalyze_x** program. The p16 (and later) versions runs **d0correct** plus packages that convert thumbnail information → tmb tree information.

Table 2: TMB Tree Classes I

Interface classes	
TMB class	**edm class**
TPhysObj	D0PhysObj
Physics Objects + associated objects	
TMB class	**edm class**
TMBCps	CPSCluster
TMBFps	FPSRecoCluster
TMBEmCells	EMCells
TMBEmcl	EMparticle
TMBJets	Jet
TMBLeBob	LeBob
TMBMet	missingET
TMBMuon	MuonParticle
TMBTaus	Tau
TMBTrks	ChargedParticle with TRef to
TMBIsoTrks	Isolated ChargedParticle info.
TMBVrts	Vertex
TMBTRefs	LinkedPhysObj

The tmb tree code is in 5 libraries:

- **tmb_tree**: all container classes + example macros
- **tmb_tree_maker**: fills all containers except MC, bcjet and trigger
- **tmb_tree_trigger_maker**: fills all trigger containers
- **tmb_bcjet**: fills all bcjet classes. This is obsolete, b id group has decided to drop all b jet classes generated by d0reco.
- **mc_analyze**: fill MC container classes

Every TMB container class is filled by a different instance of a maker, ie TMBTaus are filled by TMBTausMaker, etc. All makers follow the same basic template with some variations depending on whether objects are in a TClonesArray or not.

Table 3: TMB Tree Classes II

General Information	
TMB class	**edm class**
TMBCalQual	cal. quality from CalDataChunk
TMBGlob	global info
TMBHist	HistoryChunk
Trigger Information	
TMB class	**edm class**
TMBTrig	trigger names (fired)
TMBL1AndOr	AndOr terms
TMBL1CalTower	CalTrigTower
TMBL1Muon	l1mu_reco
TMBL2EM	l2calemp_reco
TMBL2Jet	l2caljetp_reco
TMBL2Muon	l2gblMuon_reco
TMBL3ToolResults	tool definition with TRefs to
TMBL3PhysicsResult	L3 physics object
MC Information	
TMB class	**edm class**
TMBMCevtInfo	MCevtInfo + chunk info
TMBMCpart	MCparticle
TMBMCvtx	MCvertex

To analyze a p16 (or earlier) tmb root tree file all the code one needs, besides root, is in the tmb_tree and kinem_util libraries. Versions after p16 also require met_util library. The tmb_tree/doc/README.txt has precise instructions and tmb_tree/macros has example macros for analysis. The documentation is in

`http://www-d0.fnal.gov/nikhef/?tmb_tree/ClassIndex.html`

It does not have yet a complete description of class methods but the plan is to expand it to have similar documentation to that available for root classes (using root documentation tools).

5.1.3 Top Tree Root Format

TopTree root tuple is a derivative of the analysis_example family of root tuples. It consists of 11 branches that could be divided in two logical categories. The first group of branches represents physics objects like electrons, muons, taus, and triggers. These branches are TopEvent branch, Objects branch, and MissingEt branch.

The second category of branches are specific to various top quark decay signatures, and are filled for events that satisfy certain criteria. These branches are EJets, MuJets, EMU, DiMuon, DiEM, AllJets and Properties. The Properties branch has variables relevent for kinematic constrained fit for top pair production events.

All branches consist of either simple type variables (int,float) or arrays of objects (TClonesArray). The objects are simple structures with access methods and little if any methods that do something more complicated than simply return a value of the class variable.

This simplicity allows to work with the TopTree either using a MakeClass() generated access code or to link object definitions with the analysis code and extract objects from the root tuple directly (top_tree_reader). More information about the later approach can be found in Ref. [11].

The output of top analyze is highly configurable and many branches and variables inside branches could be dropped from being saved in a root tuple to save space.

Below is more detailed description of branches.

- TopEvent Branch.

  TopEvent branch has variables like run number and event and tick number, primary vertex position calculated with two algorithms (standard and two-pass), list of L1, L2, L3 trigger names, L1 prescales and fired L1 AndOr terms, solenoid polarity, luminosty blocks and calorimeter quality flags.

- Objects branch.

  Object branch is the largest one. It consists of arrays of classes (TCloneArrays). One array for each object. The objects are EM Particles, Muons, Jets, BadJets (jets that fail jetID cuts), Vertex, NewVertex (two pass vertex), Calorimeter cells (optional, basically content of CalDataChunk), Calorimeter Towers (optional), Tracks, Track Clusters,

MC Particles, MC Vertex, TrackJets, Secondary Vertex Array, Secondary Vertex probability, PreShower, Tau.

Each Object, for instance EM Particle, consists of multiple variables (of simple — int,float) type. Some variables that were included in EM objects by demand of the people doing Electron ID optimizations. As a result, in addition to standard 4-momentum and EM cluster quality variables (EM fraction, isolation, HMatrix, Electron likelihood) EM objects contains all variables used to calculate the likelihood like energy in all EM layers, distance to CPS (central pre-shower) cluster, number of strips in the preshower cluster and others.

Also instead of links to objects in TrackArray, EM object constains variables that describe momentum and coordinates of the track associated with the calorimeter EM cluster.

And finally variables (bool) that define tight electron and electron within good fiducial region of the calorimeter. Tight electron means an electron that passes EM fraction, isolation and likelihood cuts. The variable is included to guarantee common definition of tight ("good" electron) in all analysis.

Other objects in Object Branch has similar structure. They include various variables taken directly from corresponding EDM Chunks and also various different quantities usefull for object ID optimization studies.

Links between objects are implemented as indices into arrays in Object Branch, or relevant quantities are directly included into the object description itself (like in the above EM object example).

- Trigger Branch.

  Trigger Branch is optional. It can be disabled. It consists of L1, L2, L3 tool names, L1, L2, L3 objects (reconstructed by the trigger tools) and indices that allow to identify which tool reconstructed each object. Some of the objects are : L2 jet, L2 muon, L2 EM, L2 Track, L3 Track, L3 Jet, L3 EM, L3 Muon, L3 Missing Et.

- Missing Et Branch.

  Missing Et branch includes various calculation of Missing Et, with or without EM and Jet Energy Scale corrections. Missing Et branch does

not have all the definitions present in the EDM MissingETChunk.

- Property Branch.

  Property branch includes variables outputed by hitfit package that is used in top mass fitting analysis.

- EJets, MuJets, EMU, DiMuon, DiEM and AllJets Branches.

  These branches include definitions of variables used in analysis of different $t\bar{t}$ decay channels. The variables included here are : aplanarity, sphericity, W mass, Wp_T, invariant dilepton mass, some of all jet p_Ts (jets satisfying certain fiducial cuts) and others. Most of the variables could be computed from the content of Object branch, but are included here to help the analyzers and also to enforce the same definitions of these variables in all the analysis.

5.1.4 Athena

Athena is a general purpose root tuple developed by members of the Higgs dilepton physics subgroup [12]. Athena handles all standard physics objects used by DØ, including b-id. All standard object corrections are applied using the d0correct framework package. Several b-tagging algorithms are applied, including the certified versions of impact parameter (JLIP, not CSIP), secondary vertex (SVT), as well as (uncertified) soft-lepton-tagging (SLT). The b-tagging algorithms are applied using the framework-d0root interface. Athena also contains higl-level branches geared toward dielectron and dimuon analyses.

Athena has its own cvs package, which has never been released (as of the writing of this document), but which can be checked out of cvs. The maintainers also supply a tarball for use on non-DØ machines. Athena is documented in its own web page [12].

Operationally, Athena is a list-type of tuple. Links between branches are made using simple indices. Athena but supplies its own header and implementation files, or it can be used with MakeClass. With the supplied header, variables from separate branches are in separate structs (unlike MakeClass). The user does not need to modify the supplied header or implementation file. Tuple variables are stored in global variables that are accessed from an analysis macro through the supplied header. Athena uses method TLeaf::SetAddress to establish the mapping between leaves in memory and

leaves in the tuple by name. In this way, backward compatibility is maintained with respect to old tuple files if the tuple definition in the Athena software package changes.

5.1.5 wz_analyze

wz_analyze is a simple, list-of-variables type root-tuple maker developed by the WZ group after the support for reco_analyze was dropped. It has been successfully used for analysis due to its easy usage and because it is easily expandable. wz_analyze is using d0correct to access the standard DØ certified objects. The wz_analyze root-tuples are in part generated centrally by the WZ group and can be used as a starting point to create user specific root-tuples. Users are analyzing them using MakeClass.

wz_analyze contains well documentated branches [14] with informations about event, tracks, vertex, muons, EM objects, taus, trigger (L1, L2 and L3 informations), jets, missing energy and MC quantities (no b tagging informations are implemented). It writes tuples through the `HepTuple` interface using simple arrays of integers or floats. Each tuple branch can easily be switched on or off. A bunch of specific cuts can also be set via RCP to reduce the size of the root-tuples (a minimum p_T cut on the tracks for instance). Links between the branches are made using simple indices.

wz_analyze has also the ability to deal with bad runs lists (i.e. skip events/runs/LBNs if they are bad to be used for WZ analyses). It also reads the beam spot position database (ASCII file) from AADST and contains the mapping between AndOr trigger bits and AndOr trigger terms for a given run to store the AndOr names information directly into the root-tuple (using array of integers).

5.1.6 qcd_analyze

qcd_analyze is a list-of-variables type root-tuple maker developed by the QCD group. It is also used by the Jet Energy Scale group. qcd_analyze root-tuples are simple root-tuples generated centrally by the QCD group and are analyzed using MakeClass. qcd_analyze is writing tuples through the `HepTuple` interface.

Each branch in the root-tuples can easily be switched on or off by RCP. There are three versions of root-tuples:

- a full version (used for JES out-of-cone studies) containing all the calorimeter cells and calorimeter trigger towers
- a medium version without the calorimeter cells
- a short version without the calorimeter cells and trigger towers.

All versions contain informations about event, jets (0.5 and 0.7 cone), EM objects, muons, missing energy, vertex and trigger [13]. The diffractive root-tuple is an extension of the long version adding FPD informations (hit fibers, segments and track information in the FPD, timing info in FPD).

There are no links between branches as they are not needed. qcd_analyze is not interfaced with d0correct as the JES corrections are stored in the root-tuples and applied "by-hand" in the analysis codes. The qcd_analyze root-tuples suffer however from some redundancy in the stored informations.

5.1.7 AADST

The AADST format is a special purpose format that has been developed by a group of individuals who wanted to do b physics using tracks and muons [15]. The AADST format was originally developed as an outgrowth of the Guennadi Borissov's AA track reconstruction package as a way to save tracks found by AA. AA later grew into a full physics analysis package, BANA. BANA can be run as a stand alone program, reading previously made AADST files, or in the DØ framework. Framework package AAna imports reconstructed objects (tracks and muons) from standard edm chunks into static memory, and optionally runs various BANA physics algorithms. BANA algorithms can also be called as subroutines.

The focus of the BANA developers has always been physics analysis rather than algorithm development. In order to do b physics, the BANA team has developed algorithms for primary and secondary vertexing, mass-constrained fitting (J/ψ, K^0_S, etc.), and other algorithms. This effort to develop BANA algorithms has been carried out independently of the b-id and vertexing algorithm groups, and outside of the DØ framework, although, as noted above, there is a framework interface that allows BANA algorithms to be used in framework programs.

The philosophy behind BANA is remarkably similar to d0root. Both efforts allow algorithm development, as well as physics analysis, to take place outside the DØ framework, although both have framework interfaces. There

is also considerable duplication of algorithms between d0root and BANA, although BANA does not include a version of the d0root algorithm that is of greatest interest outside the b physics group, namely b-tagging of jets. BANA and d0root also have in common that there is no way to save the results of their algorithms in edm chunks.

BANA has attracted considerable support within the b physics group. BANA algorithms are used to select several b-physics CSG skims. The b physics group currently maintains centrally produced samples of AADST files.

Most people who are doing physics analysis using BANA run BANA stand alone, reading previously created AADST files, and making private tuples or histograms. There is a perception (probably true) among many members of the b physics group that the BANA system is the quickest and easiest way to produce b physics results.

The reason for the popularity and success of the BANA/AADST system is probably due to the BANA algorithm package rather than any particular advantage of the AADST physical format itself. It is certainly true that the reading speed and development cycle of the BANA/AADST system are superior to the DØ framework and thumbnails, but the same is true of root tuples.

All of the software associated with the BANA resides in the AATrack cvs package, along with the AA track reconstruction algorithm. The BANA group maintains a beam position database, which is stored as an ascii file inside the AATrack cvs package. The beam position database has proved useful to the entire DØ experiment, including people who are not dirctly concerned with BANA.

5.1.8 edmroot

`edmroot` [17] is a package for saving DØ EDM chunk object instances directly into a Root file for easy access and manipulation within Root.

The problem that `edmroot` was designed to solve is that of algorithm development within Root and how to incorporate those algorithms into Reco easily. Root is attractive for algorithm development because it allows for interactive data analysis where one can easily create plots on the fly. Root also allows the user to write in less strict "interpreted" C++ (though it may be debated if this near-C++ is really a good thing for the developer). The problem is that the user's Root format typically looks very different than the

schema of the D0 data chunks used by Reco. Thus porting such algorithms into Reco, where they belong, has been problematic.

The Analysis Scenario Task Force [18], reviewed the situation as well as considered future implications and proposed three possible solutions.

1. Allow for parallel development of algorithms on the thumbnail (EDM) path and the Root path, thus maintaining the status quo. While this solution would allow for the ease of use of Root and having two analysis systems for cross checks, it was rejected because

   - No code could be shared between the development paths.
   - Writing the necessary and similar code for EDM objects and Root tree objects would be effort intensive.
   - Algorithms developed within Root but seen as desirable for Reco would require extensive translation and debugging, delaying their implementation.
   - The Root Tree generation code would be completely separate from the EDM chunk code, even though they would be similar if not identical in function.

2. Develop a format independent analysis code. This solution involves writing an intermediate interface layer that users and developers would utilize for data access. The interface to data would be the same regardless of whether the actual data format is Root or EDM (thumbnails). That is there would be backends to bring the data from these formats into the common interface. While this solution has the advantages of allowing for shared tools between the data tiers and utilizing the investment in the TMBTree Root format, it too was rejected. The maintenance of the intermediate layer was seen as too large an effort. Also, there was no mechanism to prevent developers from still developing algorithms on bare EDM or Root, leading to code that would be incompatible with the system.

3. Use DØ physics chunk objects directly within Root. Although technically challenging, this solution is the one that was ultimately accepted by the task force. The actual EDM objects are to be stored in Root, and so there is no parallel development path. There is also no intermediate interface layer to maintain. The user/developer would have

the full interactive power of Root to explore the data. Changes to the contents of the chunks are automatically propagated to Root, so no updating of Root specific code is needed. Finally, algorithms written within Root can be easily ported to work within Reco, because the algorithm code would need only trivial changes to be made compatible with the DØ framework. This solution became `edmroot`.

Marc Paterno was tasked with implementing the low level infrastructure to make `edmroot` possible, and the project was completed at the end of 2003. Chunk instances are written to a Root file with a DØ framework executable including the `tree_writer_pkg` package. This package writes a chunk object into a Root branch. Note there is no translation; the actual chunk instance is saved within the Root file. In order to use the Root file, a version of Root that has been statically linked with the chunk and `edmroot` libraries must be utilized. Then interactive use is easy as shown in the following example:

```
t->Draw("jc7.ptr()->AnyMemberFunction()")
```

`t` is the Root tree object. `jc7` is a chunk (on a Root Branch) which is accessed using the `ptr()` method. Now all member functions of the chunk may be exploited directly from within Root.

Since Root is storing the actual chunk object instances, some small changes the chunk code are necessary. Root dictionaries of the chunk objects must be produced using `rootcint`. Normally doing so is trivial (the build system already does most of the work), but since `rootcint` is not an actual C++ parser, it sometimes cannot handle DØ's more complicated C++ code. Fortunately, one of the primary Root developers is resident at Fermilab and has provided very fast and valuable support. Instances of such `rootcint` failures are being actively investigated and in some cases are already corrected in the newest version of the utility.

Unfortunately, these small coding changes seem to be a barrier to widespread use of `edmroot`. Since it's announcement in late 2003, no chunks have been made compatible with the system, making it difficult to evaluate. Though some interest in this package has been revived and currently Adam Lyon and Marc Paterno are actively converting a few chunks. There is also the beginnings of an effort to bring some developers together and convert more chunks. It is not known how large `edmroot` files will be, nor have any timing studies been done yet.

Note that `edmroot` is mainly for algorithm development and while it is conceivable that one could use an `edmroot` file for physics analysis, that is not the system's primary purpose. One would imagine that instead the physics analyzer would prefer a custom highly tailored Root tuple format with their specific variables and calculated quantities or perhaps a common root format for those too busy or inexperienced to write their own format.

5.2 Performance Tests

Readback tests were run on many of the root-tuple formats. While little difference was observed between the various different root formats, an order of magnitude speed difference was observed between TMB and root-based formats. The speed of the root-based formats was most directly correlated to the amount of data-per-event required.

5.2.1 Tests

For all tests an event loop was used that cycled through all events in a data file. The root object TStopwatch was used to measure clock time.

Two type of read back tests were run. Both tests had the very simple task of plotting only the jet p_T of all the JCCB jets in the event. The first test reads in all data weather or not it is required to plot the Jet p_T (the tracks, the jet η, etc.). The second test reads back as little data as possible to make the same plot. Note that how much data read back depends upon the tree's *split level*. The higher the split level, the simpler it was to read back only a small portion of the data.

Second, an effort was made to read back both Monte Carlo and Data. Approximatly 10,000 events of each were used, where availible (TMBTree, TMB, and top_tree). In other cases, files that were availible were used for the tests.

Where possible both a *MakeClass* and an Object Oriented readback were used. The *MakeClass* method uses an infrastrucutre generated by the the MakeClass method of TTree. This method creates a simple loop, and attaches each leaf in the tree to a particular variable. This makes for a rather simple framework and can be used without additional code.

The Object Oriented readback requires the source code for the classes that were written out. This must be compiled and linked into root (usually done via a shared library). In this case, readback fills each object's member

Table 4: Rates acheived for the various formats and read back methods. Entries labeld *N/A* were not able to be filled usually for technical reasons. For example, QCD, Athena, and Higgs tests were run on availible datasets rather than a uniform set. See text for further details.

Format and Method	Data		MC	
	Full Event	Jets Only	Full Event	Jets Only
Top Tree Objects	254 Hz	N/A	172 Hz	N/A
Top Tree MakeClass	355 Hz	4616 Hz	241 Hz	58 kHz
TMB Tree Objects	157 Hz	1561 Hz	62 Hz	250 Hz
TMB Tree MakeClass	N/A	N/A	N/A	N/A
QCD MakeClass	441 Hz	7236 Hz	N/A	N/A
Athena MakeClass	910 Hz	14kHz	N/A	N/A
Higgs MakeClass	7407 Hz	27 kHz	N/A	N/A
TMB	11.4 Hz	15.4 Hz	4 Hz	3.6 Hz

variables, and the framework accesses TClonesArray's of these filled objects. The benifits of this vs MakeClass are discussed in Section 3.1.

All the code is availible for inspection – it can be found in the cvs package *d0dfwg_tests*.

5.2.2 Test Results

Table 4 shows the raw results. All of these tests were run on a dual AMD 2800+ clued0 machine *husky-clued0*. A check was made to ensure no one else was using ether the machine or the disks hung off it (via NFS).

Table 5 catalogues the per event sizes of each format. These are calculated by using the *ls -l* command to determine the size and then divideing by the number of events in the file. Note that not all formats were writing out all data. For example, the top_tree used for these tests did not contain the trigger information. The MC format is often bigger because it also contains extra MC particle information.

Some results are striking. First, the anomolously high rate for the jets-only readback for the top_tree and the Higgs Multijet samples. This has to do with the split levels the trees were produced with, which allowed one to readback only the jet p_T's and nothing else.

Table 5: Event sizes for the various formats, for the data and for the MC format.

Format	Size (KB) Data	MC
tmb	22.8	40.9
tmb (no raw data chunks)[a]	11.0	27.7
top_tree	9.3	22.9
TMBTree	18.5	51.1
QCD	21.1	N/A
Athena	6.32	N/A
Higgs	0.6	N/A

[a]Drop CalDataChunk, CalNadaChunk, and L1L2Chunk

Further, there are no MakeClass results for the TMBTree data format because root cannot produce a proper MakeClass source code due to the complexity of the missing E_T objects.

Finally, the jet-only tests for the object oriented readback tests are missing because there was no simple way to enable this feature without writing new code for the various frameworks. While this can be done, it was felt that this involved too much work and the expected results would be similar to the MakeClass speed increases.

A breakdown of the sizes for a TMB file was done by the group. The results are presented in Table 6. The author of the L1L2 trigger chunk has notified us that the chunk is under revision and should decrease in size. Table 7 shows the sizes of each component of the ThumbNailChunk. The EM information, taking up almost 50% of the space, is largly filled with zeros and other null values. This file is the TMB+ (the TMB with the CalDataChunk added in). It is likely that the experiment will move to the TMB++ which will be larger still. The increased size will mostly be driven by the addition of track clusters.

5.2.3 Discussion

A few things stick out from these test results. First, and most important, there is not much different between the various root formats. From the point of view of using one or the other, there isn't much difference. For example,

Table 6: Sizes of edm chunks (uncompressed) in a TMB file. These numbers are an average over approxminately 6000 events.

Chunk Name	Chunks/Event	Bytes/Chunk	Bytes/Event	Percent
HistoryChunk	3.00	86	260	0.53
CalNadaChunk	1.00	641	641	1.30
ThumbNailChunk	1.00	22255	22255	45.08
L1L2Chunk	1.00	16948	16948	34.33
CalDataChunk	1.00	9056	9056	18.34
TMBTriggerChunk	1.00	208	208	0.42
Total	8.00	6171	49370	100.00

the large differences in some of the limited readback tests can be explained away by the split level of the trees being read. On the other hand, the speed difference between the root readback and the TMB is significant.

It is not likely that a real analysis would read back only the variables required. Special code must be written for each variable when running in this mode, and if a variable is forgotten no error message is produced. More likely one will enable or disable the read-back of a collection of variables. For example, if an analysis dosen't require tracks, no tracking variables might be read in.

The second observed effect is that large root trees, with a large number of leaves, take a great deal of CPU time when transitioning between files. Most of this is presumably taking up by moving the leaf branch addresses. Any root-tree production should endevor to make a few large files rather than lots of small ones. TMBTree's, for example, were in two files while the top_tree provided the same events in a single file. For some short runs this was seen to make a 30% difference.

5.3 Summary

This section contains a feature comparison and assessment of some of the formats that are currently in use. Here are some of the features that were considered.

- Inclusiveness.

Table 7: Sizes of the components of the ThumbNailChunk (uncompressed) in a TMB file. These numbers are an average over approxminately 6000 events.

TMB Attribute name	Objects/event	Bytes/object	Bytes/event	Fraction
mets	1.00	636	636	2.86
envID	1.00	8	8	0.04
chiso	6.25	76	475	2.13
rcpid	1.00	8	8	0.04
cpscls	28.10	52	1461	6.57
l3s	1.00	2582	2582	11.60
rcpids	28.05	8	224	1.01
jets	9.91	72	713	3.21
vtxs	8.97	71	638	2.87
chparts	81.62	44	3591	16.14
jetAssoc	1.00	151	151	0.68
lebob	3.00	10	30	0.13
parentIDs	2.00	4	8	0.04
muonparts	1.32	192	253	1.14
__base1	1.00	12	12	0.05
taus	3.83	105	402	1.81
version	1.00	4	4	0.02
parents	16.82	4	67	0.30
emparts	11.93	856	10220	45.92
fpscls	1.66	43	71	0.32
vtxtypes	5.12	4	20	0.09
Total	215.58	100	21579	96.96

Does this format include all reconstructed physics objects? Does it use d0correct? What about trigger and MC information? Is there any low level information (like raw data)?

- Tuple package.

  Does it use root, or something else? For building tuples, does it access root through the HepTuple interface?

- User interface.

  Is it list-type or class-type? Does it supply its own header, implementation, and/or shared libraries? Does it work with MakeClass?

- Performance.

  How fast is it (see Sec. 5.2)?

- Size.

  How big is each event (see Sec. 5.2)?

- D0library coupling.

  How strongly coupled is this format to d0library and the DØ software environment? Does it have a rapid development cycle?

- Development status.

  Is this package stable and mature, or is it in need of development? Is it well-documented?

Table 8 contains a feature comparison of various analysis packages.

6 Collaboration Comments and Feedback

In the course of our investigations, the Data Format Working Group has solicited input from DØ collaborators regarding the proposed common analysis format. Many people have indeed expressed their opinions. Various e-mail discussions about the common analysis format are viewable on the Data Format Working Group web page [16]. This section attempts summarize the comments that have been received.

Table 8: Feature comparison of analysis packages.

DØ Format	File format	Supplied header?	Works with MakeClass?	Physics objects	d0library coupling
thumbnail	d0om	Classes	N	all[a]	Strong
tmb_tree	root tree	Classes	N	all[a]	Weak
top_tree	root tree	Classes[b]	Y[c]	all	Weak
Athena	root tree	Structs	Y[c]	all	Weak
reco_analyze	root tree[d]	N	Y	all	None
wz_analyze	root tree[d]	N	Y	partial	None
qcd_analyze	root tree[d]	N	Y	partial	None
AADST	special		N	μ, tracks	None
edmroot	root tree	Classes	N	all	Strong

[a] No b-tagging
[b] If read using `top_tree_reader`
[c] Can be used with or without MakeClass
[d] HepTuple

On the issue of whether there should even be a common analysis format, there was not agreement. While some people expressed strong support for a common analysis format, others expressed relative contentment with the status quo of using thumbnails as the only common format. The latter group typically views the possibility of change from the negative perspective of how it might disrupt their analysis or otherwise make their life more difficult, rather than from the positive perspective of how it might make their analysis easier or faster. Naturally, this is a legitimate concern.

There are some things that everyone agrees on concerning any possible common analysis format. Everyone agrees that a common analysis format should have good performance, such as fast reading and skimming. It should be complete enough to support most analyses, without being bloated. Everyone agrees that a common analysis format needs good documentation. Most people want it to be easy to drop or add variables, and that the procedure for doing this is documented, with examples (at least, no one argued that it should be hard to add or drop variables). Finally, everyone agrees that a common analysis format should have a rapid development cycle, whether or not it is scripted.

Many people argued that the common analysis format should be decou-

pled from d0library. This concern is driven in part by concern about the speed of the development cycle, the asummption being that having analysis code decoupled from d0library will result in a faster development cycle than if it is strongly coupled. However, a greater concern seems to be the desire to do analysis on one's laptop or other non-DØ computer without having to install a lot of software or anything like a "DØ software development environment." Furthermore, people want there to be tangible support for this mode of operation in the form of documentation, examples, tarballs, etc., rather that have it be something that is possible in principle with users being left to work out the details for themselves.

On the issue of whether a common analysis tuple should be object-oriented or flat, there were people on both sides of the issue. The main concern of the people who argued for a flat structure seems to simplicity or ease of use. They like the idea of not having to deal with configuration issues, such as setting the include path or loading shared libraries. They also liked the idea of having all variables visible in a single header file.

The content of the common analysis format has up to now gotten relatively little attention beyond generalities. A few people expressed the desire to have calorimeter cell data inside root trees. The QCD group actually has this in their qcd_analyze root tuples.

Much of the commentary that the Data Format Working Group received is related in one way or another to the speed of the development cycle. Many people complained about the slowness of linking DØ framework programs. The desire to have a rapid development cycle is usually one of the main reasons why people gravitate to scripted systems, such as root. Quite a few people stated that they wouldn't mind using a linked program for their analysis, provided that the link time could be made much faster than it currently is for a DØ framework program. Of course, most people who use root are actually compiling and linking their macros into shared libraries anyway, using root (among other things) as a kind of light build system. Some people expressed a desire to get cint out of the picture entirely, and read root trees using a 100% linked program. Naturally, this is possible with any format. What is sometimes lacking is documentation and examples on how to do it. It was noted that programs built out of root macros lack many basic features that exist in the framework, such as an event class, and a system like rcp to control the program at run time. It could be argued that if you want all of the features of the framework, you should work in the framework and read thumbnails. The reason that people usually cite for not

doing this is that the development cycle is too slow.

One of the most controversial issues is algorithm development in root. The reason that is usually cited for wanting to do algorithm development in root is the slowness of the development cycle for the DØ framework. While there is indeed wide recognition that the slowness of the framework development cycle is a problem, there is no consensus on what to do about it. Some people, such as the d0root team, have used root as an alternative to the framework for algorithm development. Others (particularly the WZ group) are adament that algorithms must be developed inside the framework despite the difficulties, because the full thumbnail will always be needed for their analysis. Some people expressed a desire to have something like a dual development environment, such that code could be developed inside root using an edm-like interface, and then be used directly in or ported to the framework. Some people mentioned edmroot by name. Still others argued that what was really needed was an effort to speed up the d0library development environment, i.e. not to use root at all for algorithm development, but to make framework development more acceptable by speeding up the framework link time dramatically.

To summarize, some desirable aspects of a common data format are not controversial, like good performance and good documentation. As far as features and content, generally speaking there seem to be two schools of thought. One school wants everything as simple as possible for the sake of portability and ease of use. The other school wants a a feature-rich root environment that is as much like the framework as possible in terms of features and content. In its extreme form, this school wants a root data tier to become an alternative to thumbnails for algorithm development.

7 Requirements for a Common Analysis Format

In this section we list the formal requirements for a common analysis format. A short list of requirements has already been presented in Sec. 1. These requirements are included as the first four requirements in the following expanded list.

1. Content.

   (a) All reconstructed physics objects (electrons, muons, taus, jets, missing E_T, charged particles, vertices, b-id).
   (b) Full trigger information.
   (c) CSG event flags.
   (d) Full MC information.

2. Ability to do skimming without reading entire event (e.g. based on CSG event flags or other physics object attributes).
3. Small size (as small as possible, but not more than about 20 kB/event for data).
4. Fast reading speed (much faster than thumbnail reading speed of about 10 events/second).
5. Rapid analysis code development cycle (no linking or very fast linking).
6. Accessible via sam, including parallel projects.
7. Customizable. Easy to drop or add variables, including adding custom variables.
8. Portable to non-D0 computers.
9. Support for schema evolution (can read old tuples with newer software).
10. Data definition separate from user macros.
11. Easy to use.
12. Good documentation.

8 Recommendations

This section contains the recommendations of the DØ Data Format Working Group.

8.1 Thumbnail

We begin by examining how various analysis formats stack up against the requirements listed in Sec. 7. We first consider our current common analysis format, the thumbnail. Thumbnail content (requirement 1) is good, and getting better with the tmb++ format. Size (requirement 3) is getting larger with tmb++. However, thumbnail size could easily be made much smaller by simply dropping the raw data chunks. In fact, one could imagine having d0reco or the CSG skimming write two thumbnails: tmb++ and a smaller tmb without raw data. Read speed and development cycle for thumbnails (requirements 4 and 5) are poor.

Some have argued that what DØ needs is not a new root-based analysis format, but rather a focused effort to improve the analysis experience using thumbnails. Such an effort would necessarily have to address the thing that people complain about most, namely the framework development cycle, and especially framework linking time. It might be possible make some improvement by the use of computer science measures such as shared libraries or reducing coupling between d0library packages. In fact, progress in this direction was reported in the short term computing meeting at the 2004 Fresno Workshop in the Short Term Computing meeting by Gustaaf Brooijmans.

The second main problem with the thumbnail is the reading speed, which is currently of order 10 events/second. In Sec. 4, we pointed out that it may be possible to speed up thumbnil reading to order 30-50 events/second by fixing or turning off hot spots in the thumbnail unpacking. Another idea for speeding up thumbnail reading is to give users direct access to thumbnail objects from the ThumbNailChunk, or to store thumbnail objects in their own chunks, rather than recreating dst chunks. It is clear that large improvements in thumbnail reading speed are possible. On the negative side, thumbnail reading will probably never be as fast as root, and thumbnails don't have the flexibility that root has of reading only selected branches.

The other disadvantages with thumbnails as an analysis format, as compared to root, are ease of use and portability (strong coupling to d0library). It is unlikely that there will be major improvement in these areas.

We conclude that thumbnails in their present form are not the best choice for the common analysis format. Significant improvements in linking time and thumbnail reading speed appear to be possible. These improvements should be pursued whether or not the thumbnail is replaced as the common analysis format, as thumbnails will still be needed for algorithm development

and for some analyses. However, we think that, even with forseeable improvements in the thumbnail and framework, most people will continue to prefer root as an analysis format. That is, if DØ opts for the status quo option of keeping the thumbnail as the only common analysis format, individuals and physics groups will undoubtedly continue to do what they do now, which is to support their own root tuple formats.

8.2 Edmroot

The edmroot format can be viewed as a computer science solution to the problem of the slow framework code development cycle by allowing development of code that accesses edm chunks (including analysis code) in an cint interpreted or semi-interpreted environment. As an analysis format, edmroot suffers from some of the other defects of thumbnails, namely slow reading speed, strong coupling to d0library, and poor ease of use. Edmroot has in common with edm the problem that DØ's edm chunks (dst chunks and ThumbNailChunk) are not optimized for analysis. So, if edmroot were to be used as an analysis format, it would be desirable to develop a way to get fast access to thumbnail physics objects. We think that edmroot may have a future in DØ as an algorithm development tool, but it should not be used as the common analysis format.

8.3 Root Tree

Now we consider root tuples or trees (we use the two terms interchangeably) as candidates for the common analysis format. First of all, it is clear that a reasonably designed root tuple can satify the first four requirements (content, size, speed, rapid development).

Of the remaining requirements, one that is of particular concern for any root-based analysis format is requirement 6, accessibility via sam. This committee believes that making a parallel sam interface for root should not be all that difficult in principle. However, if for unforseen reasons it were to turn out to be impossible or impracticable to make a sam root interface, then the entire conclusion of this report would have to be rethought. Another issue is the fact that the development of a sam root interface will probablly require some negotiation between DØ and the Computing Division sam group. We do not think that the CD sam group would object in principle to having a parallel sam root interface. In fact, there might well be interest from other

non-DØ sam users in a parallel root interface. However, the sam group might not want to have the sole responsibility for developing and maintaining a sam root interface, or their timescale may not agree with DØ's. In that case, DØ should be prepared to support the development of a sam root interfacd with its own manpower, in consultation with CD sam experts.

With regard to the portability requirement (8), most root tree formats have minimal coupling to d0library (except edmroot). We regard coupling to a limited set of d0library packages as acceptable, provided that clear documentation and support (e.g. tarballs) exist for installing DØ analysis software on non-DØ computers. The remaining requirements should not present any insurmountable problems for a root tuple format.

8.3.1 Object Oriented Root Tree

Now, we examine the question of class-type vs. list-type root tuples. First, we note that the issue of class-type vs. list-type root tuples is not a black-and-white one. In terms of data storage, there is little difference between the two types of tuples. The interface through which the analyzer views the tuple data may be object-oriented to a greater or lesser degree, depending on the extent to which it contains a hierarchical, as opposed to flat, data organization, whether it provides non-trivial methods other than simple accessors, and the extent to which is incorporates root utility classes, such as TClonesArray or TRef, into its data model. Our opinion, generally speaking, is that object-oriented is better than flat. The argument that people usually make against class-type tuples or in favor of flat tuples is that objects are too complicated, or too hard to use, or too hard to learn, etc. While there is something to such arguments (or people wouldn't make them), we think that in the long run, people are more productive with objects than without them. We think that ease of use and learning curve issues can be mitigated, if not entirely eliminated, by having good documentation. For people who are determined to have a flat tuple, there is always MakeClass. One good thing about objects is that tree class header files are a source of documentation about tree contents.

Regarding MakeClass, we think that it is an advantage if a tuple format can work with MakeClass, but it is not an absolute requirement. In principle, it should be possible for any tuple format to work with MakeClass. That is, the only reason that a tuple format would not work with MakeClass is root bugs. However, the situation with MakeClass is evolving rapidly. The root

developers are working on an improved version of MakeClass. Furthermore, it should be possible for DØ to request the root team to get MakeClass to work with any particular tuple format. For these reasons, we think that it isn't worth making design concessions because of short term problems involving the current version of MakeClass. Once the common analysis format root tuple is relatively fixed, DØ should request the root developers to fix any problems that prevent MakeClass from working.

8.3.2 Algorithm Development

When the subject of a common analysis format came up, one of the first questions that people asked is whether this means that from now on algorithm development, data corrections, etc., will be done in root. The subject of algorithm development in root has been the subject of much discussion on the d0dfwg mailing list.

The short answer to the question of whether algorithm development should be shifted from thumbnail to root tuple is no. In a trivial sense, anyone who is writing root macros for the purpose of doing their physics analysis is doing algorithm development. One needs to distinguish, therefore, between "analysis algorithms," which are mainly of interest to one or a few analyses, and "reconstruction algorithms," which are of broad interest within the collaboration. Obviously, there is also a gray area. By these definitions, reconstruction algorithms include "late" algorithms, such as jet energy corrections and all of the corrections in the d0correct package. It also includes b-id. The speed with which an algorithm runs (whether it is fast or slow compared to thumbnail I/O speed) does not determine whether an algorithm should be classified as analysis or reconstruction. We believe that any algorithm that qualifies as a reconstruction algorithm should be able to be run as a package in the framework, under rcp control, getting its input from edm chunks, and storing its results in an edm chunk. The reason for this requirement is simple. Some analyses need to access data in the thumbnail that will not be in the common format root tuple. In order for such analyses to be able to benefit from DØ's full array of reconstruction algorithms, the results of these algorithms need to be available inside the framework. Furthermore, algorithms results need to be available in a chunk, and not just from a function call, because that is the only way DØ has to ensure that the algorithm has been run in a standard, certified way (i.e. with compatible, certified code and rcp parameters).

How will the requirement that reconstruction algorithms need to run in the DØ framework be enforced? If an algorithm is deemed to be of sufficiently broad interest by the DØ management that it qualifies as a reconstruction algorithm rather than simply as an analysis algorithm, then the DØ management will decree that the algorithm, and not just the analyses that use it, will need to be certified by an editorial board. It will be the responsibility of the editorial board to certify that the algorithm meets the requirements of a reconstruction algorithm, including that it runs in the DØ framework, and stores its results in a chunk.

The requirement that a reconstruction algorithm run in the DØ framework does not absolutely preclude doing algorithm development outside the framework (in root or elsewhere). The AA track reconstruction algorithm is an existence proof that a reconstruction algorithm can be developed outside the framework and successfully integrated into the framework. We think that algorithm development in root is feasible provided that an import and export mechanism exists between the framework and root, so that the same c++ code can run in the framework and in a compiled root macro.

An alternate method of doing algorithm development in root that is definitely unsatisfactory is the cut-and-paste method. This is the method where you develop an algorithm in root for a while, then you use a text editor to copy code into a framework package.

d0root Up to now, the main example and test case of an attempt to develop reconstruction algorithms in root is the d0root package of algorithms, which includes vertexing and b-id algorithms. Up to now, a framework import mechanism (edm-to-root) exists, but a framework export mechanism (root-to-edm) has not yet been developed. The algorithm leaders and developers who are responsible for d0root say that using root has made them much more productive. On the other hand, the cost of maintaining interfaces to multiple data formats has been a significant burden. We think that it is a waste of time to debate whether moving b-id algorithm development from the framework to root was a good idea. It is a done deal that d0root is what it is. We do not think that there is a fundamental flaw in the d0root strategy. The d0root algorithms need to be made to work as well as possible, and the framework export interface needs to be completed.

BANA BANA is another algorithm package that has been developed outside the DØ framework. Like d0root, BANA has a framework import mechanism, but no export mechanism. Some of the algorithms in the BANA package are clearly analysis-specific, such as reconstruction of particular b hadron exclusive final states. Other algorithms in the BANA package appear to qualify as reconstruction algorithms, and might indeed be of interest to people outside the b physics group if they had access to them. One clear example is primary vertexing, but it may not be the only one. We know of no reason why general-interest algorithms in the BANA package should not run in the framework.

8.3.3 Root Tree Contents

The general requirements for root tree contents are listed in requirement 1 in Sec. 7, which are reconstructed physics objects (including b-id), full trigger information, full mc information, CSG event flags. We purposely did not include a requirement for any raw data, such as calorimeter cells or `CalDataChunk`, in the content requirement, because we do not think centrally produced root tuples should include this information. We think that algorithms that require raw data qualify as reconstruction algorithms, and should run in the framework off of thumbnails, if not in d0reco. This being said, we have no objection to including software support in the tuple-making program for any data, including raw data, that analyzers find useful to have in their private tuples, provided that someone is willing to implement and maintain this software support. We just think that raw data should be excluded from any centrally produced tuples.

As to the question of whether the new common analysis format should be based on tmb_tree, or one of the other formats that have been developed, or redeveloped from scratch, our preference is that it be based on tmb_tree. Our main reasons for this are as follows:

- Tmb_tree was originally developed to include most of the contents of the thumbnail, rather than for any specific physics analysis.
- Tmb_tree is object-oriented. Tmb_tree objects derive from a common base class `TPhysObj`, which give access to the object contents as a `TLorentzVector`. This is a plus, and is something which is not true of the other formats.

Our preference for tmb_tree is not based on a preference for its contents, except for the general expectation that that it is already a general purpose format. We have not studied the content of any format in detail. For example, we have not determined why the tmb_tree is a factor of two larger per event than top_tree and a factor of three larger than Athena. The design process for the common analysis format should include a detailed review and comparison of similar objects and branches from several formats, with a view to including the best features and contents from each in the common analysis format. We also think that the tmb_tree physics objects should be compared with the dst version of the corresponding physics object with a view to eliminating merely gratuitous differences between the tuple and dst class interfaces.

Although we have not studied the contents of tmb_tree in detail, there are already some changes and improvements in content that we know should be made.

- The results of b-id algorithms need to be added.
- Add a branch containing CSG event flags.
- In cases where different incompatible versions of algorithms are used to reconstruct the same physics object (such as the different jet and EM cluster algorithms), the results of each algorithm should be stored in a separate branch.
- Some information is stored redundantly, for example p_T and η in addition to p_x, p_y, and p_z. We think that it would be better not to store information that can be easily recalculated.

Although we are saying that the common root format should be based on tmb_tree, we do not think that it will be possible to maintain backward compatibility with the current tmb_tree, either with respect to the interfaces of tuple objects or the data format.

8.3.4 Root Framework Infrastructure

One of the problems with the current tmb_tree maker, tmb_analyze, and, to our knowledge, with all current programs that make root tuples from thumbnails, is the fact that the tuple-making process is implemented in a

single framework package. All tuple branches are filled and then written out by a single package. Such a design offers little flexibility apart from being able to turn on or off predefined branches. In particular, such a design does not make it easy to add new branches. What is lacking is the ability to have several framework packages that can cooperate and share root data among themselves. One would like the ability to add a custom branch by adding a framework package to the tuple-maker, without the necessity of changing the standard tuple-making packages.

We propose that the current tmb_analyze framework package split into several framework packages. At a minimum, the tuple-writing part of the process should be separated from the branch-filling part. There should be a tuple-event class (something like a TMBEvent), that acts as a container for branch data, and that could be passed from package to package, somewhat like edm::Event is currently. Whether TMBEvent would move thorugh the framework work queue, or be shared in some other way, is yet to be determined. In this model, branch data classes would function somewhat similarly to edm chunks, probably deriving from a common DØ-supplied base class (possibly tmb_tree's TPhysObj). Basic root-data packages would include a root tuple event writing package, a sam-enabled root tuple event reading package, packages that fill or add branch data, and skimming packages. The TMBEvent class and the root-reading and writing packages should not be hard-coded for a particular set of branches. Such an architecture would offer great flexibility for producing or operating on root tuples in batch.

The current d0root algorithms, in addition to being able to export their results to an edm chunk, should also be able to export their results to a branch in TMBEvent. This would allow d0root to be easily merged into the common format tuple maker.

Finally, we note that having an infrastructure for handling root format data in the DØ program framework in batch in no way precludes reading common format root tuples interactively using normal root macros. Root macros would work as they do now, would access data as TChain or TTree objects, and would not make use of TMBEvent.

8.3.5 Documentation

It goes without saying that the common analysis format needs to have good documentation. In this section, we give some specific recommendations of what we think should be included.

Documentation should cover various analysis scenarios, including the following:

- Interactive analysis — reading tuples on disk.
- Batch analysis — reading tuples in sam.
- Reading tuples using a linked program — cint-less analysis.
- Using branch splitting. Reading partial event information.
- Running the tuple-maker from thumbnails.
- Making tuples from other tuples — skimming.
- Adding and dropping branches in the tuple-maker. How to write your own branch and add it to the common format tuple.
- How to do analysis on your Windows or linux laptop computer.

Generalities aside, the main documentation of analysis tuple content should be in the tuple class headers. Any other way of documenting tuple contents is bound to be too burdensome and difficult to maintain. An automatic documentation system that allow users to view hyperlinked class headers in a web browser, and which assists developers by automatically generating html would be highly advantageous. Root provides such a documentation system [19]. We recommend that the root documentation system be used to generate a browsable version of the headers. The resulting web pages would resemble root's own reference web pages.

8.4 Central Production of Common Format Root Tuples

In order for most of the benefits of common analysis format root tuples to be realized, and in particular for the common analysis format to serve as new "microdst" data tier, replacing the current thumbnail in this capacity, common analysis format root tuples need to be produced centrally. We have not completed a study of the impact of such production on the experiment. However, the general characteristics that such a production should have are clear:

- The Common Sample Group should be responsible for central production of common analysis format root tuples.
- The input for centrally produced root tuples should be CSG skim thumbnails. CSG skims would still need to be saved in tmb++ format for analyses that need full tmb++ information.
- Central root tuple production should incorporate standard physics object corrections via the d0correct framework package.
- Centrally produced root tuples should be stored in sam.

The venue for central production of root tuples, whether cab, remote farm sites, or the FNAL production farm, is yet to be determined.

8.5 Summary of Recommendations

Here is a summary of recommendations in this section.

1. The common analysis format should be a root tree.
2. DØ would benefit from an effort to speed up the framework development cycle (if successful), although it is beyond the charge for this group to specifically recommend that such a effort be launched.
3. Hot spots in thumbnail unpacking should be fixed.
4. There should be a root sam interface that supports parallel projects.
5. The common analysis format root tree should be lightly coupled to d0library, so that it is possible to do interactive analysis on non-DØ computers.
6. There should be a well-defined procedure for adding new branches to the common format root tuple.
7. The common root tree should fulfill the other requirements listed in Sec. 7.
8. The common root tree should be object-oriented.

9. It is an advantage, but not an absolute requirement, that the common root tree work with MakeClass. DØ should request that the root team get MakeClass to work with whatever the common analysis format turns out to be.

10. Algorithms that are of general interest should run in the framework.

11. Development can be done outside of the framework (in root or elsewhere), provided that a framework interface exists that allows the same code to run in the framework.

12. D0root and BANA algorithms that are of general interest should be integrated into the framework.

13. The common root tree should contain all reconstructed physics objects, including b-id, full trigger information, full mc information, and event flags.

14. The common root tree should not contain raw data (e.g. `CalDataChunk`), except as an option for private tuples.

15. The common root tree should be based on tmb_tree.

16. The contents of the common root tree need further review.

17. As compared to the current tmb_tree, the common root tree needs to have added the following information: b-id, event flags, trigger.

18. As compared to the current tmb_tree, the common root tree should have the jets and EM cluster branches split into separate branches for different algorithms.

19. Redundant information that can be easily recalculated should be removed from the common root tree.

20. An infrastructure should be developed for processing root-events in the framework, including a TMBEvent class and utility framework packages.

21. Documentation should cover the various analysis scenarios listed in Sec. 8.3.5.

22. The root documentation system should be used to genreate html documentation from tree headers.

23. Common analysis format root tuples should be centrally produced by the Common Sample Group.

As the common root tree format is the one that most collaborators will interact with, it is very important that a small group be identified as responsible for developing, maintaining, updating, and documenting the code. This same group should respond rapidly to questions and suggestions from users.

References

[1] http://www-d0.fnal.gov/cgi-bin/d0news?read__GENERAL__13240.

[2] Common Sample Group web page:
http://www-d0.fnal.gov/Run2Physics/cs/index.html.

[3] http://www.nuhep.nwu.edu/∼schellma/cab/cab.pdf.

[4] D0 Note 4146, Technical description of the T42 algorithm for the calorimeter noise suppression, Jean-Roch Vlimant, Ursula Bassler, Gregorio Bernardi, Sophie Trincaz-Duvoid

[5] D0 Note 4267, Correction of the energy sharing problem in the calorimeter data, Jan Stark.

[6] D0 Note 4268, Correction of the tower two problem in the calorimeter data, Jan Stark.

[7] D0 Note 4263, Revertexing in TMB's using offline Vertex Infrastructure, Gordon Watts.

[8] http://pax.home.cern.ch/pax.

[9] http://www-d0.fnal.gov/cgi-bin/cvsweb.cgi/io_packages/doc/EventFlags.txt?rev=1.2&content-type=text/vnd.viewcvs-markup.

[10] d0om/doc/d0om_user_guide.ps.

[11] http://www-clued0.fnal.gov/∼schiefer/top_tree_reader.html.

[12] http://www-d0.fnal.gov/∼suyong/athenaweb/athena.htm.

[13] http://www-d0.fnal.gov/∼demine/qcd_analyze_ntuple.htm.

[14] http://www-d0.fnal.gov/d0dist/dist/releases/development/
wz_analyze/doc/html/index.html.

[15] http://d0server1.fnal.gov/users/nomerot/Run2A/B_ANA.html.

[16] http://www-d0.fnal.gov/Run2Physics/working_group/data_format/.

[17] Marc Paterno,
http://www-cdserver.fnal.gov/cd_public/cpd/aps/mfp/web/index.htm
See edmroot and tree_writer_pkg links.

[18] Amber Boehnlein for the Analysis Scenario Task Force, ADM 11/8/2002,
http://ww-d0.fnal.gov/atwork/adm/d0_private/2002-11-08/anal_code_scen.ppt

[19] http://root.cern.ch/root/Documentation.html.