

Recommendations Regarding Common Analysis Format Content

D0 Data Format Working Group

November 12, 2004

Contents

1	Introduction	3
2	Overview of Tmb_tree	3
2.1	Executable, framework packages and rcps	3
2.1.1	Structure of TMBAnalyze_x	4
2.1.2	Overview of tmb_tree Content	4
3	General Considerations Regarding Root Data Formats	6
3.1	Redundant Attributes	6
3.2	CAF Data Model and Standard Root Classes	6
3.2.1	TObject	6
3.2.2	Root collections: TObjArray and TClonesArray	7
3.2.3	Smart Pointers: TRef and TRefArray	8
3.2.4	TMBLorentzVector	8
3.3	Schema Evolution	9
4	Specific Branches	9
4.1	Global Event Data	9
4.1.1	TMBGlob — Global Event Object	9
4.1.2	TMBHist — Processing History	9
4.1.3	TMBTrig — Fired Triggers	10

4.2	Monte Carlo Information	10
4.2.1	TMBMCpart — Monte Carlo Particles	11
4.2.2	TMBMCvtx — Monte Carlo Vertices	12
4.2.3	TMBMCevtInfo — Global Monte Carlo Information	12
4.2.4	Particle and Parton Jets	12
4.3	Reconstructed Physics Objects	13
4.3.1	TMBTrack and TMBIsoTrack — Charged Particles	13
4.3.2	TMBVertex — Vertices	16
4.3.3	TMBEmcl — Electrons and Photons	18
4.3.4	TMBEmCells — Calorimeter Cells	18
4.3.5	TMBMuon — Muons	20
4.3.6	TMBJets — Jets	25
4.3.7	TMBLeBob — Unclustered Energy	27
4.3.8	TMBTaus	27
4.3.9	TMBMet — Missing E_T	29
4.3.10	TMBCps — Central Preshower	30
4.3.11	TMBFps — Forward Preshower	33
4.3.12	TMBTRefs — Overlapping Physics Objects	34
4.4	B -Tagging	34
4.4.1	Primary Vertices for b -tagging	36
4.4.2	V^0 's	37
4.4.3	Filtered Tracks	38
4.4.4	Track Jets	38
4.4.5	Secondary Vertices	39
4.4.6	b -Tagging Results	39
4.5	Trigger Results	42
4.6	Branch and Class Names	42

Table 1: Framework packages in `TMBAnalyze_x`.

CVS Package	Framework Package
<code>tmb_tree_maker</code>	<code>TMBTreePkg</code>
<code>tmb_tree_maker</code>	<code>TMBCorePkg</code>
<code>tmb_bcjet</code>	<code>TMBBCJetPkg</code>
<code>mc_analyze</code>	<code>TMBTreeMCPkg</code>
<code>tmb_analyze</code>	<code>TMBRefsPkg</code>
<code>tmb_tree_trigger_maker</code>	<code>TMBTriggerPkg</code>

1 Introduction

The Report of the DØ Data Format Working Group [1] recommended, among other things, that the proposed Common Analysis Format (CAF) be based on the `tmb_tree`. The basic reasons for this recommendation were that the `tmb_tree` is general purpose and object-oriented. The report further recommended a review of the contents of the `tmb_tree` without any requirement of backward compatibility of the CAF with respect to the current `tmb_tree`. This document contains the result of this review. Here we give our recommendation of what the content of the CAF should be.

2 Overview of `Tmb_tree`

This section contains an overview of the existing (pre-CAF) `tmb_tree`, its organization, branches, framework and cvs packages, and relationship to the DST and TMB.

2.1 Executable, framework packages and rcps

`Tmb_tree` root files are built by the executable `TMBAnalyze_x` executable, which is part of cvs package `tmb_analyze`. The top level rcp is called `runTMBTreeMaker[SAM][_MC].rcp`. The packages invoked by the standard top level rcp (i.e. those that are specifically related to making the `tmb_tree`) are shown in Table 1. The first package, `TMBTreePkg`, is the overall guiding package (it somewhat resembles the standard framework `Controller` package). The remaining packages are concerned with filling one or more branches.

2.1.1 Structure of `TMBAnalyze_x`

Before moving on to a consideration of the content of the `tmb_tree`, we include here a brief commentary on the structure and design of `TMBAnalyze_x`.

The branch-filling packages construct one or more branch-filling “maker” objects (base class `TBMaker`), which register themselves with the single `TMBTreePkg` instance using static/global methods. This design is good in the sense that it allows the content of `tmb_tree` to be extended without changing any existing code (this is one of the requirements for the CAF). However, there are some negative aspects and missing features in the current design. For example:

1. The fact that `TMBTreePkg` is a singleton means that it is not possible to write more than one `tmb_tree` at a time. It should be possible to write multiple `tmb_tree`'s, each with different branches and event selection.
2. The current `TMBAnalyze_x` does not generate metadata for storing `tmb_tree`'s in `sam`.
3. The current `TMBAnalyze_x` only runs on `d0om` data, and the maker objects only have access to data stored in `edm` chunks. It would be useful for `TMBAnalyze_x` to run on either `d0om` or `root` data, and for maker objects to have access to both `edm` chunks and `root` branches.

The Data Format Working Group intends to address these limitations, however the proposed solutions are not the subject of this document.

2.1.2 Overview of `tmb_tree` Content

The branches of the current `tmb_tree` are summarized in Table 2.

The largest framework package is `TMBCorePkg`, which fills branches for reconstructed physics objects and raw data. Package `TMBTreeMcPkg` fills Monte Carlo information. Package `TMBBCJetPkg` is obsolete, having been replaced by `d0root`-based *b*-tagging algorithms. `TMBBCJetPkg` is commented out of the standard `runTMBTreeMaker.rcp` in the latest test releases, but is still present by default in the `p14` and `p16` versions of `tmb_analyze`. We will have nothing further to say about `TMBBCJetPkg` in this document, but we are proposing a replacement (see Sec. 4.4). The trigger branches created by package `TMBTriggerPkg` are likewise in need of major revision. We are proposing a replacement based on the `TrigSimCert` package (see Sec. 4.5).

Table 2: Branches in `tmb_tree`.

Framework Package	Branch	Tree Object	DST Object	Chunk(s)
TMBCorePkg	Trks	TMBTrks	ChargedParticle	ChargedParticleChunk
	IsoTrks	TMBIsoTrks	ChargedParticle	ChargedParticleChunk
	Vrts	TMBVrts	Vertex	VertexCollChunk
	Emcl	TMBEmcl	EMparticle	EMparticleChunk
	EmCells	TMBEmCells	EMparticle	EMparticleChunk
	Muon	TMBMuon	MuonParticle	MuonParticleChunk
	Jets	TMBJets	Jet	JetChunk
	LeBob	TMBLeBob	LeBob	JetChunk
	Taus	TMBTaus	Tau	TauChunk
	Met	TMBMet	MissingET	MissingETChunk
	Glob	TMBGlob		TMBTriggerChunk
	Trig	TMBTrig		L1L2Chunk, L3Chunk
	Hist	TMBHist		HistoryChunk
	CalQual	TMBCalQual		CalDataChunk Calt42Chunk Calt4_25Chunk FPSClusterChunk CPSClusterChunk
	Fps	TMBFps		
	Cps	TMBCps		
TMBBCJetPkg	BCJets	TMBbcJet	bcJet	bcJetChunk
TMBTreeMcPkg	MCpart	TMBMCpart		MCKineChunk
	MCvtx	TMBMCvtx		MCKineChunk
	MCevtInfo	TMBMCevtInfo		MCKineChunk
TMBTrefsPkg	TRefs	TMBTRefs		LinkedPhysObjChunk
TMBTriggerPkg	L1CalTile	TMBL1CalTower		L1L2Chunk
	L1CalTwr	TMBL1CalTower		L1L2Chunk
	L1CalEMTwr	TMBL1CalTower		L1L2Chunk
	L1Muon	TMBL1Muon		L1L2Chunk
	L1AndOr	TMBL1AndOr		L1L2Chunk
	L2Jet	TMBL2Jet		L1L2Chunk
	L2EM	TMBL2EM		L1L2Chunk
	L2Muon	TMBL2Muon		L1L2Chunk
	L3ToolsResults	TMBL3ToolsResults		L3Chunk

Apart from these major revisions, we believe the existing branches of the current `tmb_tree` can provide a reasonable basis for the CAF.

3 General Considerations Regarding Root Data Formats

3.1 Redundant Attributes

The current `tmb_tree` classes are shot through with redundant attributes. A common situation that arises is storing the 3-momentum of physics objects in both Cartesian coordinates (p_x, p_y, p_z) and Cylindrical coordinates (p_T, η, ϕ) . Given that it is faster to do a recalculation, even one involving a transcendental function call, than it is to read four bytes of data from disk, we recommend that in most cases these redundant attributes should be eliminated. The dropped attributes can be replaced by methods, if necessary.

The only concrete advantage of keeping redundant attributes, that we know of, is having interactive access to these quantities through the root browser. Thanks to Axel Naumann, the root browser has been extended to allow browsing of const methods that can be called with zero arguments (as is already the case using method `TTree::Draw`), effectively removing this incentive to keeping redundant attributes. The only caveat is that the classes involved must be known to root (shared library loaded, etc.).

The specific case of storing the 4-momentum of physics objects will be handled by class `TMBLorentzVector` (see Sec. 3.2.4), which will be used as a base class for objects for which this makes sense.

3.2 CAF Data Model and Standard Root Classes

This section describes the basic requirements for CAF classes.

3.2.1 TObject

`TObject` is the standard root base class. `TObject` provides a standard set of features for all root objects. `TObjects` can be printed (`TObject::Print`), drawn (`TObject::Draw`), persisted (`TObject::Streamer`, `TObject::Write`), cloned (`TObject::Clone`), stored in root collections (`TObjArray`,

`TClonesArray`), and pointed to by root smart pointers (`TRef`). Default implementations of most standard `TObject` methods are supplied by the root precompiler `rootcint`. It is a basic requirement that the top level class of a root tree branch inherit from `TObject`.

Each `TObject` carries a two-word overhead of persistent data (before compression), namely, a unique id. and a status bit word. Since both words are integers, compression of the `TObject` overhead tends to be good. Because of the `TObject` overhead, it is normally preferred not to use root classes as internal persistent attributes inside branch classes, unless there is a specific reason for doing so.

3.2.2 Root collections: `TObjArray` and `TClonesArray`

Root has several different kinds of collection classes, including its own versions of most of the STL collection classes. Except for the top level collection of a tree branch, which is always `TClonesArray` for array-type branches, there is little reason to be concerned about root collections, or to prefer root collections to STL collections.

One thing that all root collections have in common is that they inherit from base class `TCollection`. This base class provides standard ways of iterating all root collections, for example, as well as inheriting from `TObject`. Another thing that all root collections have in common is that they hold the objects they contain by reference (as `TObject*`), rather than by value, as in the case of STL collections. Since root collections are not templated, extracted objects must be cast to the correct type.

Root collections may or may not own the objects they contain (ownership can be controlled by method `TCollection::SetOwner`). Root provides some features and tools to automate memory management. Memory management in root nevertheless remains complicated and error prone (as is true of C++ generally).

The basic variable size array collection is called `TObjArray`, which corresponds roughly to the STL collection `vector<TObject*>`. Root collection `TClonesArray` inherits from `TObjArray` and has the same general structure, but can only be used for identically typed objects (`TObjArray` has no such restriction). As compared to `TObjArray`, `TClonesArray` has several optimizations and restrictions, the main optimization being that different attributes are streamed in separate buffers, allowing (usually) greater compression.

3.2.3 Smart Pointers: TRef and TRefArray

TRef is the root smart pointer class, which can be used to point to any object that inherits from **TObject**. **TRef** is dereferenced by calling method **TRef::GetObject**. As with root collections, **TRef** is not templated, so a cast to the correct type is required when dereferencing.

TRef inherits from **TObject**, and has a persistent size of three words (before compression), with two of the persistent words coming from the **TObject** base class. In the case of **TRef**, the unique id. part of the **TObject** stands for the object being pointed to, rather than to the **TRef** itself.

Class **TRefArray** is a special type of collection for **TRefs** only. **TRefArray** has a per-element overhead of only two persistent words, as compared to three words for a single **TRef**. The per-element overhead consists of the **TObject** part of the stored **TRefs**, the non-**TObject** part being shared by all of the elements in the collection. Thus, **TRefArray** is preferred to any other way of storing a collection of **TRefs** (root collection, STL collection, or bare C++ array).

3.2.4 TMBLorentzVector

The standard root 4-vector class **TLorentzVector** was considered and rejected as a common base class for physics objects, mainly because its methods are not virtual. The data format group decided to develop our own 4-vector class **TMBLorentzVector** as the standard base class for physics objects that have a 4-momentum. **TMBLorentzVector** is similar to root's **TLorentzVector**, but has mostly virtual methods. Another optimization of **TMBLorentzVector** as compared to **TLorentzVector** is the use of mass rather than energy as the fourth component in the persistent representation. This representation give better compression, since $D\mathcal{O}$ tree data often consists of many objects with identical masses.

Table 3: Attributes of `TMBGlob`.

Attribute(s)	Keep?
Run number (int)	Yes
Event number (int)	Yes
Store number (int)	Yes
Luminosity block (int)	Yes
Tick number (int)	Yes
Solenoid polarity (int, 0=forward, 1=reverse)	Yes
Toroid polarity (int, 0=forward, 1=reverse)	Yes
Solenoid current (float)	Yes
Toroid current (float)	Yes
Event flags (vector<int>, current length 4)	Add
Muon quality (6 ints)	Add
Calorimeter quality (? ints)	Add

3.3 Schema Evolution

4 Specific Branches

4.1 Global Event Data

This section describes global event produced by `d0reco` in the `tmb.tree`. This information is very simple, and not very big.

4.1.1 TMBGlob — Global Event Object

Branch `Glob` (class `TMBGlob`) is the main repository for “global event data.” The global event data stored in this branch originates in the `Global` object stored in `TMBTriggerChunk`. The header `Global.hpp` can be found in `d0library` package `config_base`. The contents of `TMBGlob` are shown in Table 3. We are recommending expanding the current branch by the addition of event flags, muon and calorimeter quality words.

4.1.2 TMBHist — Processing History

Branch `Hist` (class `TMBHist`) stores processing history from `HistoryChunks`. The contents of `TMBHist` are shown in Table 4. We do not recommend any changes.

Table 4: Attributes of TMBHist.

Attribute(s)	Keep?
Number history chunks (int)	Yes
Program name (string)	Yes
Program version (string)	Yes

Table 5: Attributes of TMBTrig.

Attribute(s)	Keep?
Number of triggers (int)	Yes
Trigger name (string)	Yes
L3 pass flag (bool)	Yes
L2 pass flag (bool)	Yes
L1 pass flag (bool)	Yes
L3 unbiased flag (bool)	Yes
L2 unbiased flag (bool)	Yes
L2 bit number (int)	Yes
L1 bit number (int)	Yes
L1L2Chunk flag (bool)	Yes

4.1.3 TMBTrig — Fired Triggers

Branch `Trig` (class `TMBTrig`) stores the list of fired triggers. This branch is intended to provide event selection based on trigger. The contents of `TMBTrig` are shown in Table 5. This branch gets its data either from `L1L2Chunk` and `L3Chunk` (if present), or `TMBTriggerChunk`. We do not recommend any changes.

4.2 Monte Carlo Information

The information in the three Monte Carlo branches mirrors information in `MCKineChunk`. Dst or thumbnail files have several `MCKineChunks`, one for each generated event (usually one hard scatter and several minimum bias events), plus one `MCKineChunk` that is added by geant. The Monte Carlo branches contain the union of the various `MCKineChunks`, and also retain enough information to determine which `MCKineChunk` each particle and vertex originated from.

Table 6: Attributes of `TBMCPart`.

Attribute(s)	Description	Keep?
p_x, p_y, p_z	Cartesian 3-momentum	Yes (<code>TMBLorentzVector</code>)
p_T, η, ϕ	Cylindrical 3-momentum	No
E	Energy	Yes (<code>TMBLorentzVector</code>)
q	Charge	Yes
id	Particle type (pdg id.)	Yes
Associations	Initial vertex (<code>TRef</code>)	Yes
	Final vertex (<code>TRef</code>)	Yes

Due to the large number of Monte Carlo particles in a typical MC event, the MC particle and vertex branches are important contributors to the overall size of MC `tmb_trees`. Therefore, it is important not to waste space for these branches.

4.2.1 `TBMCPart` — Monte Carlo Particles

Monte Carlo particles are stored in branch `MCpart`. The attributes of `TBMCPart` are shown in Table 6. Currently, `TBMCPart` stores two redundant representations of the 3-momentum, one of which should be removed. The mass and charge of the particle can be obtained from the pdg id. via built in root class `TParticlePDG`. However, the energy attribute needs to remain to handle the case of off-shell or wide particles. The charge attribute could be eliminated, but we recommend keeping it because it should be highly compressible, and because of the possibility of particles with pdg ids. unknown to root.

Another change that might be considered is eliminating some of the `TRefs`. Currently, bidirectional associations exist between Monte Carlo particles and vertices. One might consider replacing some of these bidirectional associations with unidirectional (forward only) associations. However, we recommend that the bidirectional associations be retained, as there can be a need to traverse the MC particle-vertex tree backward as well as forward (example: Did a Monte Carlo muon arise from the decay of a b quark-containing hadron?).

Table 7: Attributes of `TBMCvtx`.

Attribute(s)	Description	Keep?
x, y, z	Cartesian position	Yes
ct	Time	Yes
Associations	Parent particles (<code>TRefArray</code>)	Yes
	Daughter particles (<code>TRefArray</code>)	Yes

Table 8: Attributes of `TBMCevtInfo`.

Attribute(s)	Description	Keep?
r, e	Run & event number	Yes
n_R	Reaction number	Yes
σ	Cross section	Yes
w	Event weight	Yes
$\hat{q}^2, \hat{s}, \hat{t}, \hat{u}$	Relativistic invariants	Yes
N_{chunk}	Number of <code>MCKineChunks</code>	Yes
N_{P_i}	Number of particles in chunk i	Yes
N_{V_i}	Number of vertices in chunk i	Yes

4.2.2 `TBMCvtx` — Monte Carlo Vertices

Monte Carlo vertices are stored in branch `MCvtx`. The attributes of `TBMCvtx` are shown in Table 7. We do not recommend any changes in this class.

4.2.3 `TBMCevtInfo` — Global Monte Carlo Information

Monte Carlo global information is stored in branch `MCevtInfo`. The attributes of `TBMCevtInfo` are shown in Table 8. Since there is only one copy of this class in an event, it is insignificant in terms of its total contribution to the size of `tmb_tree`. We do not recommend any changes.

4.2.4 Particle and Parton Jets

Monte Carlo particle and parton jets are found by `d0reco` and stored in `JetChunk` along with calorimeter jets. In the old `tmb_tree`, particle and parton jets stored in `JetChunk` were stored in the `TBJets` branch along with calorimeter jets. In the CAF, particle and parton jets will still be

stored, except that the old `TMBJets` branch will be split by jet algorithm, meaning that particle and parton jets will have their own branches.

4.3 Reconstructed Physics Objects

4.3.1 TMBTrack and TMBIsoTrack — Charged Particles

Charged particle (track) objects are stored in branches `Track` and `IsoTrack` (renamed from `Trks` and `IsoTrks`). The former branch stores data for all tracks in an event, while the latter holds information for a subset of tracks that pass minimum p_T and isolation cuts. The `Track/IsoTrack` split mirrors the split between thumbnail objects `ChTmbObj` and `ChIsoTmbObj`, which is made already by `d0reco`. The cuts that determine which tracks qualify as high- p_T and isolated are included in the charged particle reconstruction rcv file `<chpart_reco ChPartReco>`.

The attributes of the current `TMBTrack` object are shown in Table 9. If we look at the variables in `TMBTrack`, we see that the most important variables are the `trf` track parameters and error matrix. Other obviously useful variables are the hit mask, track chisquare, and energy loss in the `smt` (the energy loss in the `cft` is currently unimplemented in `d0reco` and is of limited usefulness in any case). There are two representations of the track 3-momentum, in addition to the `trf` track parameters (for a total of three different representations of the 3-momenta). We recommend keeping the the Cartesian 3-momentum (as part of the `TMBLorentzVector` base class), and losing the the cylindrical 3-momentum and the last three `trf` track parameters, which are just another representation of the 3-momentum (however, the `trf` track parameters should be continue to be available as methods, since they will be needed for any fitting or propagation). Everything else except charge can be gotten rid of. There are several attributes having to do with the relationship of a track to primary and secondary vertices. We feel that these vertex-oriented attributes should be dropped due to the fact that tracks are logically prior to vertices. Quite often, the first thing that people do when analyzing `tmb_tree`'s is to throw away any existing vertices and re-find vertices. Therefore, the proper way to store the association between tracks and vertices is a one-way `TRef` from the vertex object to the track object.

The attributes of the current `TMBIsoTrack` object are shown in Table 10. There are several attributes relating tracks to preshower clusters. The associations, which are made in `d0reco` (in `ThumbNailPkg`), and which association

Table 9: Attributes of **TMBTrack**.

Attribute(s)	Description	Keep?
p_x, p_y, p_z	Cartesian 3-momentum	Yes (TMBLorentzVector)
p_T, η, ϕ	Cylindrical 3-momentum	No
E	Energy assuming m_{π^+}	Yes (TMBLorentzVector)
q	Charge	Yes
h	Hit mask (96 bits, 3 ints)	Yes
r	Radius (?) — not filled	No
$\Delta r, \sigma_r, \Delta z, \sigma_z$	L.P. & error to best vertex	No
$r_S, z, \phi_D, \tan \lambda, q/p_T$	Trf track parameters	Keep r_S, z
σ_{ij}^2	Trf error matrix (15 floats)	Yes
χ^2/dof	Track chisquare/dof	Yes
$\Delta E, \sigma_{\Delta E}$	Energy loss, error in cft, smt	Keep smt, drop cft
χ_{PV}^2	χ^2 wrt two pri. vertices	No
χ_{SV}^2	χ^2 wrt two sec. vertices	No
Associations	TMBIsoTrack (one TRef)	No
	Pri. vertices (TRefArray)	No
	Sec. vertices (TRefArray)	No

is one of the criteria for retaining preshower clusters in the thumbnail, should be kept, but the track-preshower residual and chisquare should be dropped, as they can easily be recalculated. There is also an association to the (non-isolated) track object, **TMBTrack**, which should be kept.

Another set of attributes has to do with calorimeter and tracking energy flow type information. This is the type of information that would potentially be useful for doing, say, lepton identification. The lepton branches have their own versions of the quality information they need, of course. But the information in **TMBIsoTrack** could potentially be useful for doing track-based lepton identification where no lepton object was found by **d0reco**. One example of track-based lepton identification is the third lepton in trilepton analyses. It is our view that the energy flow information in the current **TMBIsoTrack** class needs refining. The purely track-based energy flow (track-energy in cones) is easily recalculated and can be dropped. The calorimeter MTC information is certainly useful for muon identification. However, muon reconstruction already includes track-seeded muons (i.e. muons found from track and MTC information with no matching local muon track in the muon system). We

Table 10: Attributes of `TMBIsoTrack`.

Attribute(s)	Description	Keep?
r, ϕ, z	Position at preshower	To <code>TMBTrack</code>
$\Delta\eta, \Delta\phi$	PS clus. sep. (not incl.)	No
χ_{PS}^2	χ^2 wrt PS cluster	No
E_{Ti}, N_i	Track E_T , mult. in cone i	No
x, y, z	Position near vertex	No
MTC	Cal. MTC data (15 floats, 1 int)	No
E33 and E55	Calorimeter energy	No
EM (0.2, 0.3, 0.4, 0.5)	Calorimeter energy (4 floats)	Add
FH (0.2, 0.3, 0.4, 0.5)	Calorimeter energy (4 floats)	Add
CH (0.2, 0.3, 0.4, 0.5)	Calorimeter energy (4 floats)	Add
ICR (0.2, 0.3, 0.4, 0.5)	Calorimeter energy (4 floats)	Add
Associations	<code>TMBTrack</code> (one <code>TRef</code>)	Yes
	CPS clusters (<code>TRefArray</code>)	Yes
	FPS clusters (<code>TRefArray</code>)	Yes

do not think storing MTC information in `TMBIsoTrack` adds anything, so we recommend dropping it. Calorimeter energy flow and isolation information is useful for various types of lepton identification. Our recommendation is that the calorimeter energy flow information be modified from its current form to be the energy in various cones around the track at several depths in the calorimeter. This an expansion of what exists currently.

The remaining attributes of `TMBIsoTrack` are purely kinematic. One of these is the position of the track where it passes close to a vertex (we think). This can be dropped as it is easily recalculated by a short-distance propagation. This brings us to a really useful attribute in the `TMBIsoTrack` branch, which is the position of the track at the preshower (actually, it would be useful to have the track angles and momentum too, that is, all five track parameters and one surface parameter). This information is potentially useful, and it is not easy to recalculate correctly at the tree level. One can imagine, for example, that someone might want to know which tracks of any momentum fall within the cone of a jet, and this information might not be available in the jet branch. Our recommendation is that the track parameters at the preshower be stored in the `TMBTrack` branch.

Track propagation at the Tree Level It would be quite difficult to do a proper track propagation (e.g. using `DOPropagator`) at the tree level, as this would require bringing in a great deal of `d0library` code (magnetic field map, detector geometry, etc.), and would be slow as well. This is why we recommend storing the track parameters at the preshower in the track branch.

It is much easier to do a short-distance propagation with acceptable accuracy. A common instance of this is propagating the track parameters from the (0,0) dca surface, where they are calculated by `d0reco`, to a vertex (that is, to the (x_V, y_V) dca surface). Class `TMBTrack` should provide a method to do this (it doesn't currently). Such a method could be specialized for a dca surface, assume a uniform axial magnetic field, and make short-distance approximations. It need not import any `trf` code from `d0library`. The axial magnetic field polarity would have to come from an external source, as it is not stored in `TMBTrack`.

The Track-Vertex Measurement Problem Another problem that should be solved by a method (not by additional attributes) is the track-vertex measurement problem. That is, one should be able to calculate the 2D impact parameter and error matrix of any track with respect to any vertex. Note that 99% of the solution of this problem consists of propagating the track parameters and error matrix to the vertex dca surface, since, apart from a z offset, the first two `trf` dca track parameters are precisely the radial and z impact parameters, and the impact parameter error matrix is precisely the 2×2 upper left corner of the track parameter error matrix.

Vertex-Constrained Tracks A third method that we recommend adding to `TMBTrack` is a method to refine the estimate of track parameter by the use of vertex information. Equivalently, this can be thought of as constraining a track to a vertex or applying the Kalman smoothing algorithm to the vertex reconstruction problem (assuming that the vertex was originally reconstructed using a Kalman fit type of algorithm).

4.3.2 TMBVertex — Vertices

Reconstructed vertices are stored in class `TMBVertex` (formerly `TMBVrts`). The attributes of `TMBVertex` are shown in Table 11. One of our recommendations is that class `TMBVertex` remain as a base class for different kinds of

vertices, and that derived classes be added for primary vertices, secondary vertices, and V^0 's.

As far as additions, we recommend adding attributes for the total 4-momentum of the constituent tracks (via base class `TMBLorentzVector`) and total charge. This is mainly of interest for secondary vertices and V^0 's, but is not totally without interest for primary vertices, so we recommend adding this attribute to the base class.

The vertex class currently has a vertex-type attribute, which currently is only used to distinguish primary and secondary vertices (currently, secondary vertices are not present in `tmb_tree` because they are not implemented in the framework). In the future, there is the possibility to have multiple primary or secondary vertex algorithms. Our recommendation is to split the vertex branch according to algorithm, making this attribute unnecessary, so it should be dropped.

Another attribute that could be a candidate for dropping is the number of degrees of freedom, which is redundant with the size of the `TRefArray`. However, it makes sense to keep this attribute (or equivalent), so that the analyzer has the option of not reading in the `TRefArray`, which can be large compared to the rest of the vertex branch. We think that it is preferable to save this information in the form of the number of tracks rather than the number of degrees of freedom, which we feel is a more intrinsically interesting variable.

For the primary vertex class `TMBPrimaryVertex` (see Table 12), our recommendation is to add an attribute for the minimum bias probability, in addition to the base class attributes.

For secondary vertices (class `TMBSecondaryVertex`, see Table 13), we recommend adding as an attribute a reference to the associated primary vertex. The decay length and decay length significance should be available as methods.

Neutral V 's are a special case of secondary vertices. Class `TMBV0` should be derived from `TMBSecondaryVertex`, with an additional attribute for the pdg id. (see Table 14). In this case, it is desirable that the 4-momentum stored in the base class be calculated including a mass and primary vertex constraint. Note that the class `TMBV0` being proposed here is not a full-fledged system for dealing with fully reconstructed or identified particles, but is intended to solve the more limited problem of finding long-lived two-prong neutral V 's, namely, K_S^0 's and Λ 's. The $K_S^0//\Lambda$ case is somewhat special in the sense that the identity of the daughter particles (i.e. the tracks) can be

Table 11: Attributes of `TMBVertex`.

Attribute(s)	Description	Keep?
p_x, p_y, p_z, E	4-momentum	Add (<code>TMBLorentzVector</code>)
q	Total charge	Add
T	Vertex type	No (split branch by type)
x, y, z	Position	Yes
σ_{ij}^2	Position error matrix	Yes
χ^2	Chisquare	Yes
N_{dof}	Degrees of freedom	Yes (\rightarrow # of tracks)
Associations	Tracks (<code>TRefArray</code>)	Yes

Table 12: Attributes of `TMBPrimaryVertex`.

Attribute(s)	Description	Keep?
<code>TMBVertex</code>	base class	Yes
P_{MB}	Minimum bias probability	Add

unambiguously inferred from the identity of the parent, which is not true in the general case. A general system for dealing with identified or fully reconstructed particles (including, for example, B and D mesons), would need a way of attaching identity hypotheses to the constituent particles.

4.3.3 `TMBEmcl` — Electrons and Photons

4.3.4 `TMBEmCells` – Calorimeter Cells

The current version of `TMBTrees` optionally stores calorimeter cells for electromagnetic clusters. The code is part of `TMBEmclMaker` and, depending on a boolean flag, either produces a list of calorimeter cells which are part of a cluster, or the clusters themselves with references to the cells.

Each `TMBEmCells` object inherits from `TObject` and stores its coordinates

Table 13: Attributes of `TMBSecondaryVertex`.

Attribute(s)	Description	Keep?
<code>TMBVertex</code>	base class	Yes
Associations	Primary vertex (<code>TRef</code>)	Add

Table 14: Attributes of `TMBV0`.

Attribute(s)	Description	Keep?
<code>TMBSecondaryVertex</code>	base class	Yes
<code>PDGID</code>	Particle type	Add

(`ieta`, `iphi`, `ilayer`) as three integers and its energy as a float. The total size of each cell object is 24 bytes. See table 15 for details.

The references between electromagnetic clusters and cells is stored as a `TRefArray` in `TMBEmcl`.

The current implementation wastes a lot of space by having each cell inherit from `TObject` and storing the coordinates as 32 bit integers despite their restricted range. Some of this is compensated for by ROOT's compression algorithm. The `TObject` base class is needed for the `TRef` references from the EM clusters.

The current implementation only stores cells for EM clusters, not for any other physics object. It is not possible to store the full data chunk or killed cells (e.g. by Nada and/or CalT42).

We suggest to change the structure of the calorimeter cell branch in the following way:

- The cell class no longer inherits from `TObject`.
- The coordinates are stored as 3 bytes instead of 3 integers.
- An additional byte for flags is added (see below).
- A new container class which inherits from `TObject` is introduced. It contains a vector of all cells that have been stored for this event. A cell is identified by a 16 bit index in this container. The container is filled dynamically: as new cells are added they are either assigned a new index, or an existing index is returned. Every cell is stored only once.
- Every physics object that wants to store a list of cells, stores a list of cell indices instead of a `TRefArray`. It only references the container object, not every single cell. The combination of the container object plus the index is enough to find a pointer to a given cell.

Table 15: Attributes of `TMBEmcells`.

Type	Attribute	Description	New Type
int	ieta	eta coordinate (-40..40)	signed byte
int	iphi	phi coordinate (0..64)	unsigned byte
int	ieta	layer (1..17)	unsigned byte
float	energy	Energy	float
NEW	flags	Cell level flags	unsigned byte

The new cell size will be 8 bytes instead of 24. The size of the information in a physics object is roughly 2 bytes times the number of cells.

There should be additional RCP flags which can be enabled during tree generation for experts. The default will be to not generate the full data chunk or any of the Nada or CalT42 cells.

- `bool doCalData` - add cells from full `CalDataChunk`
- `bool doCalNada` - add cells from `CalNadaChunk`
- `bool doCalT42` - add cells from `CalT4_25Chunk`

Cells which do not come from the normal `CalDataChunk` will have an additional flag set, giving information on why they have been killed.

Additional physics objects apart from `TMBEmc1` can implement the option to store associated calorimeter cells using the same mechanism.

4.3.5 `TMBMuon` — Muons

All muon objects are stored in the `Muon` branch. The small number of muons in an event means that limiting the branch size is not the main goal in redefining the branch content, although certainly there are obsolete and unsupported variables that should be cleaned out. The main goal is to provide access via clear variable or method names to data needed by those using muons in analyses. Along these lines it was decided to add a structure that allows for a sensible storage of different types of muon momenta (e.g. local, central, global, etc.). Because the size is not so much an issue, it was also deemed appropriate, except in simple cases like momentum components, to bias on the side of storing data rather than relying on methods where values

Table 16: Momentum Structure of TMBMuon

Classes	Description	
Local	no track match or eloss correction	
LocalCorr	no track match with eloss correction	
Central	track values for matched muons	
CentralCorr	track values with corrections for CFT only tracks	
Global	fit values for matched muons	
SmearedMC	MC smeared values	
Members of Each Class	Description	Comments
px, py, pz	Cartesian 3-momentum	Access via Method
pT, η, ϕ	Cylindrical 3-momentum	Stored as Data
E and q	Energy and charge	Stored as Data
$pT_err, \eta_err, \phi_err$	associated errors	Stored as Data

would end up being calculated in two separate places (once in MuoCandidate code then again in the CAF provided method). Remember that use of methods versus data will be invisible to the user.

At the moment there are six classifications of muon momentum in use or to be used in time, local (not track matched), central (track matched – but using just the track values), global (track matched using the fitted local and central values), local uncorrected for energy loss, Monte Carlo smeared, and central corrected when the track has no SMT hits. For several reasons, including having higher level code while avoiding long lists of data members in the tree, it was decided to keep these different classifications of muons in separate objects. One example is to have a set of classes that inherit from `TMBLorentzVector` (from which `TMBMuon` also inherits) based on these muon types. Then, for example, a local muon’s pT could be accessed via

```
TMBMuon::Local::pT()
```

A general momentum value could also be returned as defined by the muon group. That is,

```
TMBMuon::pT()
```

Table 17: Attributes of TMBMuon

Attribute(s)	Description	Keep?
float tanl	$\tan(\lambda)$	Method
int nhit	numbers of wire and scint hits	Data
int nseg	muon types	Data
int ndeck	numbers of deck hits	Data
int nmtc	number of calorimeter muons	Data
float deltaPhi	ϕ diff of A seg and charged part	Data
float deltaEta	η diff of A seg and charged part	Data
float deltaDrift	z diff of A seg and charged part	Data
float etrack_best	energy in cells assoc w/MTC	Data
float chisq	χ^2 of track match	Data
int ndof	degrees of freedom	Data
float prob	probability of χ^2	Method
float chisqloc	χ^2 of local muon fit	Data
float xA, yA, zA	hit positions at A layer	Data
float zAtPca, err_zAtPca	z and error at pca	Data
float impPar, err_impPar	impact parameter and error	Data
float imparSig	impact parameter significance	Data
float dca	distance of closest approach	Data
float EinCone(1,15,2,4,6)	calo energy in ΔR cones	Data
float calnLayer	number of calo layers hit	Data
float caleSig	calo energy signature	Data
float calEta, calPhi	calo η and ϕ of muon	Data
float eloss	Run I energy loss in calo	Data
int wireHits(A,B,C)	A, B and C layer wire hits	Method
int scintHits(A,B,C)	A, B and C layer scint hits	Method
int wireHits(i,j,k,l)	i, j, k, and l deck wire hits	Method
bool is(Loose,Medium,Tight)	muon quality info	Data
int hasLocal	is there a local track?	Data
int hasCentral	is there a central track?	Data
int hasCal	is there a calo signal?	Data

Table 18: Attributes of TMBMuon cont.

Attribute(s)	Description	Keep?
float drJet5, etTrkCone5	isolation variables	Data
float etHalo, int nTtk5	isolation variables	Data
float bdl	$B dl$ of muon through Toroid	Data
int isMuonEventOK	check for crate errors	Data
TRef chptr	points to matching charged part	Data
TRef vtxref	points to matching vertex	Data
float detectorEta	detector eta	Data
float sctime(A,B,C)	scint times per layer	Data
int region, octant	region and octant	Data
bool isCosmic, isCosmicT	cosmic flags	Method

would return the track matched momentum or if there is no track match, the local corrected momentum as is done at the time of this writing. The exact implementation has yet to be worked out, but the user interface will be as stated. Table 16 shows the momentum structure for the muon branch in the CAF. As in other branches only the cylindrical momentum components will be stored as data, with methods providing access to all the others.

For muon branch content aside from the new momentum structure, see tables 17 and 18. Note that parentheses do *not* denote arrays, but are different endings, e.g. wireHits(A,B,C) means wireHitsA, wireHitsB, wireHitsC. The variables in the table are largely based on the current content of the tmb_tree. There are several additions to the list as well as renaming of variables to make them more descriptive. In no case, however, did a variable name *change* meaning – renaming means an old variable moved to an unused name or used in a different context. The table points out whether the value will be stored in the tree as data, or recalculated via a supplied method. Note that the six muon quality words (with information on readout errors) will be stored in the Global branch. See table 19 for a list of variables being removed. Note that some of these will still exist in other forms, for example, float pxA, etc. are migrating into the Local class.

In the end there might be a separation of branches into *user* and *expert* branches where the *expert* branch could be easily turned off for perhaps quicker run times and/or smaller output data files for users who don't need

Table 19: Attributes of TMBMuon being Removed

Attribute(s)	Description	Why Remove?
float hfrac_hit	fraction of hadronic calo cells hit	Unused
float etrack_hit	energy in calo cells hit	Unused
float hfrac_best	frac of had layers w/energy	Unused
float elast	energy in last had. layer	Unused
float e33	calo energy in 3x3 tower	Unused
float e55	calo energy in 5x5 tower	Unused
int categoryloc	types similar to nseg	Duplicate
int qualityloc	types of muons based on hit info	Duplicate
int statusloc	status of local muon fit	Obsolete
int centralmatch	number of central track matches	Unused
int centralrank	rank of track match	Unused
float qptloc	charge divided by pT	Easily Reproduced
float pxA, pyA, pzA	local fit at A layer	to Local class
float phiA, etaA	local fit at A layer	to Local class
int segIndex	index of segment	Obsolete
int TrkIndex	index of track in TrackChunk	Obsolete
bool isTightMuoTrack	is tight according to p10 or p11?	Obsolete
float pTCorr, int chargeCorr	corrected for CFT only track	to CentralCorr
float (pT,eta,phi)Central	central track pT, η , ϕ	Duplicate
int isAxialMatched	is there an axial track?	Obsolete
float road(EM,Fine)	unknown	Obsolete
float road(Course,OutFloor)	unknown	Obsolete
float scvelo	scint velocities	Unused
float sctimeBC	scint time for BC layers	Obsolete
int expWhits(A,BC)	wire hits in layers	Renamed
int expShits(A,BC)	scint hits in layers	Renamed

Table 20: Kinematical attributes of **TMBJets**.

Attribute(s)	Description	Keep?
p_T, p_x, p_y, p_z	Momenta	No, move to base class
E	Energy	No, move to base class
ϕ, η	Direction	No, move to base class
det. ϕ , det. η	p_T weighted location in terms of iphi and ieta	Yes

such detailed information. The separation of data into these branches will then need to be decided.

4.3.6 **TMBJets** — **Jets**

The attributes of a jet can be divided in four categories, descibed in Tables 20-23:

- Kinematical information.
- Quality/ID information.
- Jet Energy Scale information.
- Association to other physics objects.

Regarding the kinematical information, some of the variables stored in the **TMBTree** were redundant and could be computed from the other variables. This will be addressed in the new format with the new base class from which all physics objects inherit. Thus most of the kinematical variables stored in the **TMBJets** class can be removed.

For the Quality/ID information we propose to keep most of the variables present in the **TMBJets**. New variables can be added, for example variables having to do with energy flow or new and updated quality variables.

It is useful to store Jet Energy Scale corrections and smearing coefficients in the jet class to allow for systemtic studies. We propose to add variables containing this information, which was not present in the **TMBJets** class.

Jets are central objects within the event and can be associated with other physics objects such as tracks, primary and secondary vertices, trigger objects and so on. We propose to add all such useful associations to the jet class.

Table 21: Quality/ID attributes of TMBJets.

Attribute(s)	Description	Keep?
q	Sum of charge of associated charged particles	No
emf	E_T fraction in layers 1-7	Yes
emf1, emf2, emf3	E_T fraction in layers 1, 2, 3	Yes
icdf, ccmg, ecmg	E_T fraction in ICD, CC massless gap, EC massless gap	Yes
icrf	E_T fraction in ICD massless gap	Yes
fh1f, fh2f, fh3f	E_T fraction in Fine Hadronic layers 1, 2 and 3+4	Yes
chf	E_T fraction in Coarse Hadronic layers	Yes
emcc, hadcc	E_T fraction in the EM and hadronic part of the CC	Yes
emec, hadec	E_T fraction in the EM and hadronic part of the EC	Yes
hot	Ratio of hottest to next-hottest cell	Yes
mxET	Hottest cell E_T	Yes
etaW, phiW	Eta and phi RMS width	Yes
cpsE	De-ghosted associated CPS energy	Yes
sET, vPT	Scalar E_T and vector p_T	Yes
iPT	Initial E_T : valid only for cone jets	Yes
seedET	Seed E_T : valid only for cone jets	Yes
split_merge_word	Split/merge word: valid only for cone jets	Yes
Nitems	Number of towers	Yes
n90	Number of towers with 90% of p_T	Yes
ntrk	Number of associated tracks	No
nps	Number of associated preshower clusters	No
algoname	Algorithm name as JCCA, JCCB, etc.	No
flavor	Jet flavor in MC	Add

Table 22: Jet Energy Scale information in TMBJets.

Attribute(s)	Description	Keep?
jes_data_lq[3]	JES data correction for light jet, with stat. and syst. error	Add
jes_data_hq[3]	JES data correction for hf jet, with stat. and syst. error	Add
jes_mc_lq[3]	JES MC correction for light jet, with stat. and syst. error	Add
jes_mc_hq[3]	JES MC correction for hf jet, with stat. and syst. error	Add
jes[3]	JES correction currently applied to jet, with errors	Add
smear_coeff	Smearing coefficient applied to jet	Add

Table 23: Association of `TMBJets` to other objects.

Attribute(s)	Description	Keep?
Associations	Tracks (<code>TRefArray</code>)	Yes
Associations	Primary Vertex (<code>TRef</code>)	Yes
Associations	Track Jets (<code>TRefArray</code>)	Add
Associations	Matched Muons (<code>TRefArray</code>)	Add
Associations	Matched Electrons (<code>TRefArray</code>)	Add
Associations	Trigger Objects (<code>TRefArray</code>)	Add
Associations	b -ID Objects (<code>TRefArray</code>)	Add
Associations	PS clusters (<code>TRefArray</code>)	Add

4.3.7 TMBLeBob — Unclustered Energy

4.3.8 TMBTaus

`TMBTaus` contains information about tau candidate objects. The current attributes of `TMBTaus`, and our recommended modifications for CAF are shown in Table 24.

We propose to eliminate all duplicate attributes, including some that are uniquely determined by others. Some others that are there for historical reasons only shall be dropped as well. All commonly used accessor functions will be retained and several new ones will be added. Implementation of some existing accessors will have to change because of the above restructuring, but this will not require any change in the analysis code.

For the 4-momentum, the same coordinate system (Cartesian or Cylindrical) should be chosen for all objects. The energy `_E` of the calorimeter cluster can be calculated from the 3-momentum, and does not need to be retained as a data member. The charge `_charge`, will be calculated from the associated tracks. The number of tracks in 10° , 20° , 30° cones can also be calculated on the fly, if necessary. These attributed will therefore be dropped. The variables `_et_3`, `_et_7`, `_rms`, `_profile`, `_hot`, `_emf`, `_chf`, `_icdf` need to be retained as they contain important information about the shower shape and isolation. `_EM12_Et`, `_EM3_Et`, `_EM4_Et` and `_EM12_Et_core`, `_EM3_Et_core`, `_EM4_Et_core` also detailed information

¹The Δ 's are taken between the vector sum of tracks & the EM subcluster to calculate the invariant mass of the charged tracks and the π^0 s: `_e1e2` \times `_dalpha` = mass(track, cal cluster).

Table 24: Current attributes of TMBTaus.

Attribute(s)	Description	Keep?
<code>_px, _py, _pz</code>	Cartesian 3-momentum	No
<code>_pT, _eta, _phi</code>	Cylindrical 3-momentum	Yes
<code>_E</code>	Energy	No
<code>_charge</code>	Charge	No
<code>_ntrk</code>	number of tracks	Yes
<code>_rms, _profile, _hot</code>	Cal RMS, Profile, hot-cell ratio	Yes
<code>_emf, _chf, _icdf</code>	EM, CH, and ICD fractions	Yes
<code>_ntrk1, _ntrk2, _ntrk3</code>	# tracks in 10°, 20°, 30° cones	No
<code>_fsh</code>	Fisher discriminant (obsolete)	No
<code>_type</code>	tau decay hypothesis (1-4)	Yes
<code>_nnout[4]</code>	NN output for 4 types	Yes
<code>_flag</code>	Status word (Int)	Yes
<code>_et_3, _et_7</code>	E_T in 0.3 & 0.7 cones	Yes
<code>_EM12_Et, _EM3_Et, _EM4_Et</code>	E_T in EM layers 1+2, 3, & 4	Only EM12
<code>_EM12_Et_core, ... _EM4_Et_core</code>	Ditto in core cones	Ditto
<code>_tzDCA</code>	z of leading track at DCA	Yes
<code>_ett1, _ett2, _ett3</code>	p_T of 3 highest- p_T tracks	Yes
<code>_mtrk,</code>	Mass of all tracks	Yes
<code>_teta, _tphi</code>	η, ϕ of track (Σp)	Yes
<code>_tphiPS, _tphiEM3</code>	ϕ of leading track at PS, EM3	Yes
<code>_ettr</code>	p_T sum of non-leading tracks	Yes
<code>_empt, _emeta, _emphi, _emm</code>	p_T, η, ϕ , mass of EM subclusters	Yes
<code>_emcl_eta1, _emcl_eta1, _emcl_phi1</code>	E_T, η, ϕ of leading EM subcluster	Yes
<code>_emcl_eta2, _emcl_eta2, _emcl_phi2</code>	E_T, η, ϕ of 2nd EM subcluster	Yes
<code>_pseta, _psphi</code>	η, ϕ of PS cluster	Yes
<code>_dalpha</code>	$_dalpha = \sqrt{\frac{(\Delta\phi)^2}{\sin^2\theta} + (\Delta\eta)^2}$	Yes ¹
<code>_e1e2</code>	$\sqrt{p_T(\text{tracks}) \times E_T(\text{EMcal})}$	Yes ¹
Associations	Description	Keep?
<code>_vtxref</code>	Vertex (one TRef)	?
<code>_tracks</code>	Tracks (TRefArray)	Yes
<code>_cpsptr</code>	CPS clusters (TRefArray)	Yes
<code>_fpsptr</code>	FPS clusters (TRefArray)	Yes

about the shape of the shower in the EM layers, but are not being used. These can probably be dropped. The obsolete variable `_fsh` can be dropped. `_type` is an integer representing the categorization of the candidate based on hypothesized decay channel (presently 1-4). `_nnout[4]` are the outputs of each candidate from neural networks, one for each `_type`. Frequent re-training of the networks makes it necessary to retain all the input variables (see below), and member functions will be available for the user to re-evaluate the network functions with new sets of weights. Still, it is necessary to keep the original (reco) values of NN outputs since these are often used in selection criteria for streaming etc. Pointers to tracks and preshower clusters associated with the tau candidate need to be kept. The pointer to the associated vertex can be dropped if it can be readily accessed using the track pointer(s).

4.3.9 TMBMet — Missing E_T

Information about the missing E_T in a event is stored in the `Met` branch. The branch consists of a single object of type `TMBMet` which closely mimics the interface in the `met_evt` framework package.

The attributes of the current `TMBMet` are shown in Table 25. They contain all the information from the `d0reco` and `thumbnail` version.

The basic data structure used in the `TMBMet` object is the `BMetStruct`, defined in the `met_util` package. It consists of five floating point numbers of 32 bit each, representing the various components of missing and sum E_T :

- ME_x
- ME_y
- ME_z
- SE_T
- ME_T

The structure is repeated for different definitions of missing E_T . See [2] for the details. There are 29 versions of these, and most of them are only of interest to an expert user.

The corrected missing E_T is stored as a vector of `BMetQualInfo` objects (also defined in `met_util`). Each object is at least 164 bytes plus the size of a string identifying the jet algorithm. Its structure can be seen in Table 26.

Finally there are ring variables, four floats each, for up to 74 eta rings. It is not known if this information has been used in a RunII analysis so far.

The z coordinate of the vertex used to calculate the missing E_T is also kept as a single floating point value.

We suggest to split the missing E_T information into a user and expert branch and keep only the indicated variables in the user branch. All the other information will go into the expert branch. The missing E_T classes have been recently re-organized, so it contains no remnants of the past that can be easily removed.

All items marked **Yes** in Table 25 will be kept in the user branch. All items marked **No** will be in the expert branch. [One of the items marked **JES** will be kept in the user branch, whichever the Jet Energy Scale group believes to be most useful.]

For the `_metqualinfos` vector we suggest to keep all the information available in the thumbnails.

4.3.10 TMBCps — Central Preshower

For the CPS, contiguous strips above threshold are clustered into single layer clusters (SLCs). These SLCs are then divided into subclusters containing at most five strips. Subclusters that overlap in all three layers are used to create 3D clusters. 3D clusters associated with isolated tracks, taus, and EM objects are written to thumbnail. When writing CPS information to `tmb_tree`, each 3D cluster is represented by a `TMBCps` object. Each `TMBCps` has the following members:

- `r`
- `phi`
- `z`
- `xE`, `uE`, `vE`
- `matchQ`
- `matchEQ`
- `xSLC_Id`, `uSLC_Id`, `vSLC_Id`
- `xNStrips`, `uNStrips`, `vNStrips`

Table 25: Attributes of TBMet.

Type	Attribute	Description	User
BMetStruct	_met	CAL + ICD Towers	JES
BMetStruct	_metnoeta	CAL + ICD Towers, above tower threshold	JES
BMetStruct	_metweta	CAL + ICD Towers, in eta limits, over t threshold	JES
BMetStruct	_metT	CAL + ICD Towers, over t threshold	JES
BMetStruct	_metTM	CAL + ICD + Muon correction, over t threshold	No
BMetStruct	_metTAS	CAL + ICD over eta limit, over t threshold	No
BMetStruct	_metTBS	CAL + ICD below eta limit, over t threshold	No
BMetStruct	_metTAN	CAL + ICD over eta limit, below t threshold	No
BMetStruct	_metTBN	CAL + ICD T below eta limit, below t threshold	No
BMetStruct	_metC	CAL + ICD Cells	Yes
BMetStruct	_metCM	CAL + ICD Cells + Muon correction	No
BMetStruct	_metCAS	CAL + ICD C above eta limit, above c threshold	No
BMetStruct	_metCBS	CAL + ICD T below eta limit, above c threshold	No
BMetStruct	_metCAN	CAL + ICD T above eta limit, above t threshold	No
BMetStruct	_metCBN	CAL + ICD c below eta limit, below c threshold	No
BMetStruct	_metICD	ICD Cells	No
BMetStruct	_metNADA	NADA Cells	No
BMetStruct	_metMUON	Tight Muons	No
BMetStruct	_metD	CAL + ICD Cells layers 1-14 (no CH)	Yes
BMetStruct	_metDM	CAL + ICD c layers 1-14 (no CH) + Mu correction	No
BMetStruct	_metEM	EM layers (1-7)	No
BMetStruct	_metMG	Massless Gap layers (8 & 10)	No
BMetStruct	_metFH	FH layers (11-14)	No
BMetStruct	_metCH	CH layers (15-17)	No
BMetStruct	_metED	CAL + ICD Cells below eta limit.(no threshold)	No
BMetStruct	_metNG	negative cells	No
BMetStruct	_metT42	noise cells, based onthe CalT42 algorithm.	No
BMetStruct	_metA	META = METD + CH frac. of good jets	Yes
BMetStruct	_metB	METB = METD + CH frac. of good jets	Yes
Float_t	_Zvertex	Z of the vertex used to calculate mET	Yes
BMetQualInfo	_metqualinfos	vector of Corrected mET	Yes

Table 26: Attributes of `BMetQualInfo`.

Type	Attribute	Description
string	<code>_algo</code>	Jet Algo Name
<code>BMetStruct</code>	<code>_METcorr</code>	Final Corrected Missing ET
<code>BMetStruct</code>	<code>_CHcorr</code>	MissingET Corrections
<code>BMetStruct</code>	<code>_JEScorr</code>	MissingET Corrections
<code>BMetStruct</code>	<code>_EMcorr</code>	MissingET Corrections
<code>BMetStruct</code>	<code>_MUCorr</code>	MissingET Corrections
<code>BMetStruct</code>	<code>_MUCalcorr</code>	MissingET Corrections
<code>BMetStruct</code>	<code>_BJcorr</code>	MissingET Corrections
float	<code>_isophigoodjet</code>	MissingET isolation wrt good id objects
float	<code>_isophimu</code>	
float	<code>_isophiem</code>	
float	<code>_isophibadjet</code>	
float	<code>_isophiunclusteredenergy</code>	
float	<code>_isophibadtower</code>	MissingET isolation wrt bad Towers, Cells

- `dphi`
- `dz`
- `nn`
- `isLoose`
- `isTight`
- `p3v[3]`

The first three variables are the coordinates of the 3D cluster. The next three are the energies of the SLCs that make up the 3D cluster. `matchQ` is a χ^2 -like term that measures the quality of the spatial match of the three SLCs. `matchEQ` is an analagous quantity for the energy match of the three SLCs. The next three variables are the id numbers for the SLCs in the 3D cluster. The following three variables are the number of strips in each SLC. `dphi` and `dz` are the errors on the ϕ and z coordinates. `nn` is a neural net output to discriminate between EM objects and non-EM objects, and `isLoose` and `isTight` are boolean flags for picking EM-like clusters, based

Table 27: Average number of CPS clusters written per event.

Skim	Clusters
BID	4.7
EM1TRK	6.9
QCD	8.0

on the neural net output and the number of strips and energies in the SLCs. Finally, `p3v[3]` is a unit vector from the primary vertex to the 3D cluster, used for π^0 reconstruction.

The `(x,u,v)SLC_Id` variables identify the SLCs used in a 3D cluster, and could be used to identify 3D clusters that share SLCs. However, it is unlikely that this will be done, and it is recommended that these three variables be omitted.

`p3v[3]` can be calculated from the position of the 3D cluster and the desired vertex, and the algorithm is one that will not change over time, so it is suggested that this variable also be omitted.

All other variables should be kept. `r`, `phi`, `z`, `dphi` and `dz` are clearly necessary. `matchQ` and `matchEQ` are important quality variables useful in rejecting combinatoric background. The number of strips and energy in each SLC are used to distinguish EM-like objects from fakes. The neural net output is not easily calculated on the fly, and `isLoose` and `isTight` only require one bit each.

Since 3D clusters are only kept if they are associated with various physics objects, relatively few are written per event. Typical averages are shown in Table 27.

We would also like to consider the possibility of keeping all 3D clusters that are classified as `isLoose`. Using a diEM skim, we found that the number of clusters kept would go from 7.1 to 12.4.

In addition to the standard CPS data, the energy information for all strips that were read out in an event (i.e. - the contents of the `CPSDigiChunk`) should be available in an expert/developer branch.

4.3.11 TMBFps — Forward Preshower

As for the CPS, FPS clusters associated with isolated tracks, taus and EM objects are written to thumbnail. Information about FPS clusters are stored

in `TMBFps` objects, which contain the following members.

- `E`
- `phi`
- `r`
- `z`
- `nstr_(u,v)`
- `cntrd_(u,v)`
- `rms_(u,v)`
- `ecl_(u,v)`
- `index_cl`

`E` is the energy of the cluster, and `r`, `phi` and `z` are the coordinates of the cluster. The next variables are the number of strips, centroid, rms width and energy, respectively, of the `u` and `v` layer cluster. `index_cl` identifies which wedge the cluster is in.

The energy and position variables should definitely be kept for the common analysis format. The other variables are likely to be used only by experts, but since less than one FPS cluster is stored per event, their impact on the data size will be negligible. We therefore propose to keep the FPS variables as they are, and store the same clusters as for the thumbnail.

In addition to the standard FPS data, the energy information for all strips that were read out in an event (i.e. - the contents of the `FPSDataChunk`) should be available in an expert/developer branch.

4.3.12 TMBTRefs — Overlapping Physics Objects

4.4 *B*-Tagging

One of the shortcomings of $D\bar{O}$'s way of doing *b*-tagging is a lack of any standard way of storing persistent *b*-tagging information, either as an edm chunk, or as as a `tmb_tree` branch. The working model has been that everyone who

wants to do b -tagging has to run `d0root` macros at the analysis level, beginning with the refinding of primary vertices. This way of doing things has been developed to suit the needs of the developers of b -tagging algorithms, who say for their part that abandoning the $D\bar{O}$ framework has made them much more productive and efficient. Unfortunately, this way of doing things does not suit the needs of analyzers. For one thing, doing b -tagging at the analysis level is slow. For another, the analyzer is presented with an interface that is unnecessarily complex, and not well-documented.

The data format group proposes that the results of each stage of the lifetime b -tagging analysis chain be made persistent as an edm chunk and as a branch in the CAF. There are six such stages.

1. Refind primary vertices. This step is not part of the b -tagging algorithms proper, but for some time the primary vertex algorithm used for certified b -tagging has been different than the primary vertex algorithm that runs in `d0reco`.
2. V^0 -finding. The purpose of this step is to identify tracks originating from K_S^0 and Λ decays, so that such tracks can be removed from consideration as candidates tracks arising from the decay of b -quark-containing hadrons.
3. Track selection and V^0 -removal. The purpose of this step is to identify a sample of tracks originating from the hard scatter vertex to be used as input to the lifetime b -tagging algorithms.
4. Find track jets. Track jets are found using a cone algorithm using track as inputs. Calorimeter information is ignored, except that track jets are eventually matched to calorimeter jets using η - ϕ matching. Track jets are used to define “taggability” of calorimeter jets for all lifetime taggers (certified tag rate functions and scale factors are always defined relative to so-called taggable jets). Additionally, some taggers (namely SVT) make essential use of the track jets as input to the algorithm.
5. Find secondary vertices. Secondary vertices are searched for within track jets.
6. Apply tagging algorithm. There are currently four tagging algorithms implemented in `d0root`.

- The Secondary Vertex (SVT) algorithm uses secondary vertices as its primary input. Calorimeter jets are not used, except that the final results are applied to matched calorimeter jets.
- The Jet Lifetime Impact Parameter (JLIP) algorithm is based on the impact parameters of tracks. Tracks are treated independently. Calorimeter jets are required as an essential input to define the jet axis so that tracks can be classified as having positive or negative impact parameter. Matched track jets are not used directly, except to establish taggability.
- The Counting Signed Impact parameter (CSIP) is another impact parameter based algorithm. The same general considerations apply as for JLIP.
- The Soft Lepton Tag (SLT) algorithm is based on the finding of leptons in jets as evidence of a b -quark semileptonic decay. Currently, only muons are used. Most of the track-based infrastructure used by the lifetime taggers is not relevant for the SLT algorithm.

We recommend that the results of the above six analysis steps be saved in an edm chunk and in a `tmb_tree` branch. In addition, we recommend that a seventh chunk/branch be created to store the relationships among all of the objects and algorithms involved in b -tagging.

4.4.1 Primary Vertices for b -tagging

Among all of the `d0root` objects created for use in b -tagging, vertices (primary or secondary) are the only one for which there is already an edm chunk in existence, namely `VertexCollChunk`. The contents of `VertexCollChunk` consist of an integer vertex type, a vertex name, and a collection of `Vertex` objects.

The selection of vertex chunks for use as input for further reconstruction or analysis is done exclusively by means of the integer vertex type attribute rather than the vertex name attribute. The vertex type is used to distinguish primary and secondary vertices, as well as vertices reconstructed using different algorithms. Many packages assume that there will be only one `VertexCollChunk` of any given type, which restriction `d0root` must respect. C++ enums of all known vertex types are stored in the header file

Table 28: Attributes of `vertex::V0`.

Attribute(s)	Description
p_x, p_y, p_z	3-momentum
m	Mass
σ_m	Mass error
q	Total charge
x, y, z	Decay vertex
σ_{ij}^2	Vertex error matrix
χ^2	Vertex chisquare
PDGID	Particle type
Associations	Charged particles (<code>LinkIndex<ChargedParticle></code>)

`vertexutil/VertexType.hpp`. The type that corresponds to “physics primary vertices” is `VertexType=PRIMARY` or 3. The first vertex in this chunk is the one that has been selected as the best choice for the hard scatter vertex.

Primary and secondary vertices found by `d0root` can fit rather easily into the existing edm vertex infrastructure by simply defining new vertex types for each `d0root` vertex reconstruction algorithm. Creating additional `VertexCollChunks` will not interfere with any existing code provided that each algorithm gets a unique vertex type. Each algorithm that wants to create a chunk should register for a type by adding an enum to `VertexType.hpp`.

For the CAF, primary vertex objects will be stored using class `TMBPrimaryVertex` (see sec. 4.3.2), with different algorithms having different branch names.

4.4.2 V^0 's

The next step in lifetime b -tagging is search for fully reconstructable long-lived V^0 's (K_S^0 's and Λ 's). At the CAF level, these objects will be stored in class `TMBV0` (see Sec. 4.3.2). At the edm level, it will be necessary to define a new chunk. Rather than derive from class `vertex::Vertex`, we propose to make a new class `vertex::V0` (see Table 28). Class `vertex::V0Chunk` will consist of a collection of $V0$'s.

Table 29: Attributes of `bid::FilteredTrackChunk`.

Attribute(s)	Description
Primary Vertex	<code>LinkIndex<Vertex></code>
Charged particles	<code>vector<LinkIndex<ChargedParticle> ></code>

Table 30: Attributes of `bid::TMBFilteredTracks`.

Attribute(s)	Description
Primary Vertex	<code>TRef</code>
Charged particles	<code>TRefArray</code>

4.4.3 Filtered Tracks

The next step in lifetime b -tagging is track selection. This step consists of the following types of selections.

- Association to the hard-scatter primary vertex (2D impact parameter).
- V^0 removal.
- Quality selection (e.g. minimum number of SMT hits).
- Kinematic selection.

The output of this step consists of a collection of the selected tracks. It will be necessary to define a new edm chunk and a new branch class for the CAF. For concreteness, we propose the name `bid::FilteredTrackChunk` for the edm chunk, with contents shown in Table 29. For the CAF, we propose class `TMBFilteredTracks`, shown in Table 30.

4.4.4 Track Jets

The fourth step in lifetime b -tagging is the finding of track jets. Currently, track jets are defined in `d0root`, but there is no support for track jets in edm or `tmb_tree`. As with calorimeter jets, there is the potential to have different track jet algorithms. At the edm level, the results of different track jet algorithms will be stored in different chunks with the attribute identified by a chunk attribute (a string, presumably). At the CAF level, the track jet algorithm will be identified by the name of the branch.

Table 31: Attributes of `bid::TrackJet`.

Attribute(s)	Description
p_x, p_y, p_z, E	4-momentum
q	Total charge
status	Status word (includes taggability)
Primary Vertex	LinkIndex<Vertex>
Charged particles	vector<LinkIndex<ChargedParticle> >

Table 32: Attributes of `TMBTrackJet`.

Attribute(s)	Description
p_x, p_y, p_z, E	4-momentum (TMBLorentzVector)
q	Total charge
status	Status word (includes taggability)
Primary Vertex	TRef
Charged particles	TRefArray

The proposed edm track jet object is called `bid::TrackJet` and its contents are shown in Table 31. The CAF object is called `TMBTrackJet`, and its contents are shown in Table 32.

4.4.5 Secondary Vertices

The fifth step in lifetime b -tagging is secondary vertex finding. Many of the same general considerations apply to secondary vertices as to primary vertices (see Sec. 4.4.1). At the edm level, secondary vertices can be stored in `VertexCollChunk`, with appropriate vertex type attribute. At the CAF level, secondary vertices are stored in `TMBSecondaryVertex` (see Sec. 4.3.2).

4.4.6 b -Tagging Results

The sixth and final step in lifetime b -tagging is the application of the b -tagging algorithm proper. The tagging algorithms may make use of any of the results from the previous five steps, as well as any other results from the event.

Some results produced by the various tagging algorithms are generic, such as whether a given calorimeter jet is tagged tight, medium, loose, or not at all. Other results are algorithm specific. We propose to have a common

Table 33: Attributes of `bid::BTag`.

Attribute(s)	Description
status	Status word (includes loose/medium/tight, positive/negative, MC flavor)
Calorimeter Jet	<code>LinkIndex<Jet></code>
Track Jet	<code>LinkIndex<TrackJet></code>
MC Particle Jet	<code>LinkIndex<Jet></code>
Filtered Tracks	ChunkID

Table 34: Attributes of `TMBBTag`.

Attribute(s)	Description
status	Status word (includes loose/medium/tight, positive/negative, MC flavor)
Calorimeter Jet	<code>TRef</code> to <code>TMBJet</code>
Track Jet	<code>TRef</code> to <code>TMBTrackJet</code>
MC Particle Jet	<code>TRef</code> to <code>TMBJet</code>
Filtered Tracks	<code>TRef</code> to <code>TMBFilteredTracks</code>
Data TRF	Method or attribute
MC TRF	Method or attribute
Scale factor	Method or attribute

base class for the generic results, and derived class for the algorithm-specific results. The base class is called `bid::BTag` at the edm level, and `TMBBTag` at the CAF level. The contents of the base class are shown in Tables 33 and 34. Algorithm specific classes for SVT are shown in Tables 35 and 36. Algorithm specific classes for JLIP are shown in Tables 37 and 38. Algorithm specific classes for CSIP are shown in Tables 39 and 40. Algorithm specific classes for SLT are shown in Tables 41 and 42.

Table 35: Attributes of `bid::BTagSVT`.

Attribute(s)	Description
<code>bid::BTag</code>	base class
Secondary vertices	<code>vector<LinkIndex<Vertex> ></code>

Table 36: Attributes of `TMBBTagSVT`.

Attribute(s)	Description
<code>TMBBTag</code>	base class
Secondary vertices	<code>TRefArray</code>

Table 37: Attributes of `bid::BTagJLIP`.

Attribute(s)	Description
<code>bid::BTag</code>	base class
P_{JLIP}	Tagging probability
N_{Track}	Number of selected tracks.
Selected tracks	<code>vector<LinkIndex<ChargedParticle> ></code>

Table 38: Attributes of `TMBBTagJLIP`.

Attribute(s)	Description
<code>TMBBTag</code>	base class
P_{JLIP}	Tagging probability
N_{Track}	Number of selected tracks.
Selected tracks	<code>TRefArray</code>

Table 39: Attributes of `bid::BTagCSIP`.

Attribute(s)	Description
<code>bid::BTag</code>	base class
N_{Track}	Number of selected tracks.
Selected tracks	<code>vector<LinkIndex<ChargedParticle> ></code>

Table 40: Attributes of `TMBBTagCSIP`.

Attribute(s)	Description
<code>TMBBTag</code>	base class
N_{Track}	Number of selected tracks.
Selected tracks	<code>TRefArray</code>

Table 41: Attributes of `bid::BTagSLT`.

Attribute(s)	Description
<code>bid::BTag</code>	base class
N_{muon}	Number of associated muons.
Associated muons	<code>vector<LinkIndex<MuonParticle> ></code>

Table 42: Attributes of `TMBBTagSLT`.

Attribute(s)	Description
<code>TMBBTag</code>	base class
N_{muon}	Number of associated muons.
Associated muons	<code>TRefArray</code>

4.5 Trigger Results

[Section missing. Basic plan is to import contents of `trigsimcert` tree into CAF.]

4.6 Branch and Class Names

We suggest to change the names of branches in the files and C++ classes according to Table 43.

The changes are intended to move away from abbreviations and be internally consistent. For instance, classes for physics objects are all singular and fully spelled out.

Branch names are often used interactively, so they leave out any prefix and try to be short and descriptive.

Jet and EM branches will be split by algorithm. We suggest to use the normal Jet algorithm name for branches, e.g. `JCCA`, `JCCB`, etc. For EM branches we suggest to use `EMscone` and `EMcnn`.

References

- [1] D0 Note 4473, Report of the DØ Data Format Working Group.
- [2] DØ Note 4474, Missing ET Reconstructions in p17.

Table 43: Names of branches and C++ classes.

Old Class Name	Old Branch	New Class Name	New Branch
TMBGlob	Glob	TMBGlobal	Global
TMBHist	Hist	TMBHistory	History
TMBMuon	Muon	TMBMuon	Muon
TMBTrks	Trks	TMBTrack	Track
TMBIsoTrks	IsoTrks	TMBIsoTrack	IsoTrack
TMBVrts	Vrts	TMBVertex	Vertex
TMBEmcl	Emcl	TMBEM	(see text)
TMBJets	Jets	TMBJet	(see text)
TMBTaus	Taus	TMBTau	Tau
TMBLeBob	LeBob	TMBLeBob	LeBob
TMBMet	Met	TMBMet	Met
TMBTrig	Trig	TMBTrigger	Trigger