Questions from the Data Handling Technical Review Committee to ROOT Proposal and Answers to them

1 RESOURCE REQUIREMENTS

¿From the initial implementation plan described in CDF 4497 (Addendum 2) several areas of effort are identified:

A) Implementing C++ classes for ROOT persistent containers and optimizing their interface to TRYBOS

B) Implementing AC++ I/O modules for access to ROOT trees and conversion to TRYBOS

C) Implementing user interface code to have ROOT write separate branches to separate files

D) Implementing event/run and dataset databases and managers supporting split streams of RAW data.

E) Minimization of overlap between physics production streams

F) Optimization of Run II event format, in particular with respect to data compression

and there are additional areas of effort that are relevant in terms of a complete (i.e. beyond prototyping) implementation plan, including:

G) Interfacing to HSM (establishing that open/close commands will work, establishing naming conventions, etc.)

H) Realizing the usage of 'cloned' executables in order to optimize the use of a network-distributed disk repository (especially in terms of coordinating the output from multiple executables.)

We'd like to introduce the following notations:

- we will be calling HSS the generic (not specified yet) Hierarchical Storage System (HPSS, for example)
- we will call HSM (Hierarchical System Manager) the software managing the HSS
- we will call RUNDB all the databases used by the data handling system which do not belong to HSS. So RUNDB includes Event/Run database, File/Dataset database etc.
- *For the projects identified above and as well for many of the questions that follow it would be useful to have a better estimate of the resources required from the collaboration to develop and test prototypes and create and finally maintain the final products. We are looking for an expansion of information given on page 22 of CDF 4497.*

  Wherever possible please indicate who will do the work. How many people (individuals and/or FTEs, inside or outside of the data handling group) are needed, for how long, with what training (e.g. in ROOT, C++, Run 1 reconstruction software, Run 1 analysis) and based where (onsite, institution)?

-
  - **A)** has to be done in close cooperation with Trybos developers (Rob Kennedy). We could estimate amount of work required based on the available experience. At present initial implementation of ROOT interface classes is available for more than 100 different CDF YBOS banks, it took about 1 month to implement all of them. This work is very mechanical, once the data formats are defined, it could easily be divided between different people each spending a small part of his time to it.
  - **B) and C)** this work has to be done in cooperation with AC++ development team, the implementation is straightforward and the first version could be completed in 1 week.
  - **D)** implementation of RUNDB is a more serious effort, we consider implementation of RUNDB in cooperation with the ROOT team and an ORIACLE/SQL expert from the

computing division. MSQL-based implementation of RUNDB for NA49 took about one FTE-month (see **A1.3**, we want to be on the safe side and assume 3 FTE-months for this. This work should be done by a computer professional at FNAL. The precise specifications for this system must be discussed. A fine tuning of this database will require several iterations.

- **E)** requires more serious study of Run I CDF data and needs a physicist. We estimate that it would take 2 to 3 months for a physicist doing it about 50% of his time.
- **F)** should be done within the CDF collaboration. Many people have to be involved into this work. Input from the phyiscs groups is especially important for the data format optimization. We assume that this process will take several iterations but it doesn't seem to be on the critical path.
- **G)** we expect interface to HSM to be developed primarily by the ROOT team in close collaboration with an HSM expert. The amount of work seems to be quite modest, more accurate estimate could be done after the HSM is chosen.
- **H)** we see the development and optimization of the PROOF server to be done primarily by the ROOT team (this is already work in progress) in cooperation with CDF.

2 DESIGN AND IMPLEMENTATION QUESTIONS

We assume that something like the following functional subsystems are needed in order to build a minimally functioning data handling system:

a) File format and I/O routines to read/write that file format, providing interpretation, packing/unpacking, byte-swapping, header insertion, collection markers and counts, etc. in order to move bytes between disk and memory structures in a machine independent format - i.e. the equivalent of YBOS i/o or ROOT i/o or d0om/DSPACK in D0.

b) User API to access the objects associated with the data read/written using the file format - i.e. accessor methods, creation and deletion of objects, etc

c) User API to iterate over a collection of events, each themselves consisting of a specific collection of objects; to create a new collection of objects for each event; to delete a collection of objects; and to query up a collection of objects - i.e. a specific Physics Dataset.

d) Interface to Robotic Tape Storage to get files(objects?) which are to be added to the disk cache and to write out files(objects?) which are no longer required on disk.

e) Datafile manager to assist in determining when files are to be migrated to Tape/Robotic Storage, using various policies.

f) Mechanisms to ensure that all events and branches of events in files somewhere in the system (disk resident or tape resident) are known to the system, are where they are supposed to be, and knowledge of their whereabouts can be recovered in case of errors, power failures, disk full, etc. I.e. a metadata catalog and tools to manage it.

g) Luminosity databases or other files which must be accessed in conjunction with the ROOT format data files.

Q1: For each of the above functional subsystems can you say:

Q1.1 *whether they may be implemented using tools provided by ROOT - fully or partially?*

A1.1:
- a),b) and c) are fully provided by ROOT.
- d), e) is a job of HSM. A simple interface to HSM has to be developed, the implementation depends on the choice of HSM. In case of HPSS this interface could be provided by modification of the existing Root daemon (rootd) used by the TNetFile class to access remote files via a socket interface.
- f),g) is a part of RUNDB software.

Q1.2: *whether they might be just as appropriately implemented using tools or packages (or a database system) not part of ROOT?*

A1.2: we think it is essential for HSM and RUNDB to be decoupled from the ROOT system. Interfaces to these systems could be implemented in ROOT via additional classes, if necessary. Part of RUNDB software could be implemented using ROOT classes, but from the point of view of the architecture of the whole system its 3 major components should be decoupled.

Q1.3: *in terms of some abstract unit of work what is your estimate of the amount of work it would take to implement them.*

A1.3:

- a),b) and c) are implemented in ROOT and are supported by ROOT team. The CDF data model could be modified to take advantage of ROOT container classes. This work has already started, about 100 CDF Run 1 banks have their ROOT interfaces written. The work could be done in 2 steps:
  * at the 1st step we need to interface CDF data model to ROOT I/O to allow reading and writing data from AC++. This work could be done in about a month by somebody familiar with C++ and ROOT.
  * at the next step we need to write large number of methods which would provide non-generic access to the data stored in persistent ROOT objects. This doesn't affect step 1 and could be staged.
- d) and e) are more difficult to evaluate at this point because it depends on the choice of HSS. We insist that the HSM interface must be at the file level (not object level). In principle, the HSM layer should already be implemented in the HPSS API. Tests of this API do not require the connection to ROOT and are independent of the CDF object model in a first approximation. The interface between ROOT and the HSM API requires few changes in the "rootd" program itself (say one week for one ROOT expert working closely with an HSM expert). Most of the work is expected in the interfacing of a class on top of TChain, depending if a synchronous or asynchronous response is expected. This work could be staged, starting with something simple and gradually adapted to the architecture.
- f) and g) will represent more work. We suggest the RUNDB to be based on a relational data base (Oracle or MSQL). A similar project based on MSQL has been implemented within NA49 by P.Buncic in about one month. The RUNDB software requires write concurrency and locking + journaling mechanism. The NA49 project replaces the existing FATMEN and HEPDB systems and is more reliable. The ROOT team is willing to implement an interface to MSQL (TSQL class) within ROOT. Via TSQL, a user should be able to query the RUNDB data base with simple queries to extract for example the list of runs/files corresponding to some criteria. The result of the query could be a simple text file used as an input for ROOT to create a TChain object.

Q2: *Sections 3.2 and 4.1 of the document assume a non-exclusive streaming strategy is employed and that fixed collections of banks form particular 'tiers' of event data (to use Sequential Access Method terminology). These assumptions appear to be proposing basically the Sequential Access File model with, incidentally, the file format as ROOT file format. Comments please?*

A2: If we define Sequential Access File Model (SAM) as a model with data exchange unit between the different levels of storage hierarchy being a file (and not an object or a page), then the model we are proposing is a Sequential Access File Model. SAM model could be implemented with ROOT without any modifications of the ROOT software. ROOT-based model however has several important advantanges over the conventional (for example, YBOS-based) SAM model:

- it provides direct access to the events stored in the data file by storing the events in Trees;
- it allows to read in only those parts (branches) of event data which are necessary for the given job, thus maximizing I/O bandwidth;
- because of its multibuffer I/O, ROOT structure provides data compression with compression ratios much higher than it could be achieved within the conventional SAM model.

Q3: *Section 4.2 also appears to be exactly equivalent to one of the data tiers proposed in the Sequential Access Method - Thumbnail, or EDU5. Comment please?*

A3: Disk-resident part of the event data discussed in our proposal includes everything covered by the thumbnail data tier. However it is much more than just an equivalent of it:

- based on the results of conversion Run I BPAD's and PAD's into ROOT format we believe that about 15Kbytes of disk resident (compressed) event data which correspond to about 30 Kbytes of uncompressed data will satisfy most part of the Run II physics analyses, which is not true for thumbnail tier discussed in SAM. This would greatly simplify the data access during the physics analysis stage.
- also very important is that having all the data necessary for the analysis on disk would allow to reduce tape I/O to a great extent and to remove one of the potentially most serious bottlenecks in the data handling system.
- Within the ROOT-based model disk resident branches do not duplicate any other tier, but are just a disk-resident part of the reconstructed event data.

Q4: *Is there anything within the ROOT system itself which would facilitate efficient management of the proposed 10TBytes of disk cache? or is this a part of the data handling system which must exist and function independent of the file format used?*

A4: Resource management system is a part of HSM. ROOT provides tools to optimize management on a single file/chain of files level:

- A ROOT file has a directory system allowing direct access to one event, one branch or leaf. This directory is independent of the RUNDB. If a file is moved to a different machine, all the information to process this file is in the file itself.
- The TChain class is a utility to navigate in a list of files. A TChain object has the same query semantics as a TTree in a ROOT file. However the TChain mechanism is not a replacement for RUNDB. The two are complementary. From RUNDB, one must be able to automatically generate the TChain object. The TChain system has the advantage in the physics analysis stage to be autonomous on a local workstation. A physicist could make a query to RUNDB via the TSQL class to
  * stage the list of files via HSM for a given selection.
  * generate the text file of the TChain macro.

  then, these files (or events, selected from them) and the macro can be moved to the local workstation offering a simple analysis model independent of the more complex central data bases.

Q5: *How does the primary database interact with the Production Manager Software? Does the Production Management system write directly to the ROOT file system?*

A5: there should be an interface between Production farm and HSM/RUNDB (say, Spool Manager) which talks to HSM and RUNDB asyncronously with respect to what Production

Farm is doing. We do not suggest that Production management system writes directly to the file system managed by HSS. The reason for this is that HSS and RUNDB should be updated in sinc when new production output becomes available. Having a separate process (Spool Manager) talking to HSM and RUNDB would also simplify management of the Production Farm.

Q6: *What administrative tools are required to configure, control and monitor the database. Please summarize the status of this work. What additions are necessary for reasonable administration of a full system for CDF II? Please estimate the effort (number of FTE's) needed to administer the database system.*

A6: The HSM (HPSS) has its own management support. We assume this support to be provided by the computing division. The RUNDB software requires intervention from the CDF production system. If the system is based on Oracle or MSQL, report generators are already available from these systems. RUNDB software should be developed jointly by CDF and an DB expert in CD. This requires 1 FTE from CDF side.

3 EVENT/RUN CATALOG

In asking our questions we assume the following:

- Level 3 will scramble the order of events output from the online
- Production may further scramble the order
- It is possible for some data sets that several runs may need to be concatenated into a single file for tape-resident data (to reach the minimum optimal tape file size of 1 GB)

Q7: *Given these assumptions, we do not fully understand how single event pick access works. Note the example given for Event/Run database entries on page 7 of CDF 4497 and the absence of a single entry per event.*

A7.a: We see several possible solutions for pick mode. One of them could be implemented as follows:

- normally a ROOT datafile contains just one object - Tree with the event data, event numbers being scrambled within the file.
- after the file is closed, Production manager reads it once again and adds to the file one object per run - a balanced Tree, which contains an ordered array of event numbers and the sequential number of each event in the file (8 bytes per event total).
- this gives a straightforward implementation of pick mode with the number of comparisons proportional to $\log N$, which is exactly what any database software is doing.
- This sorted table can also be recomputed at any time from the Run/Event information in the event itself if these two attributes are conveniently stored into separate branches.
- Event/Run DB in this case contains just minimal and maximal event numbers in this file to define the files to be looked at and could me made very small with the data files containing part of the DB information, related to them.
- alternatively, one could store the objects described above in Event/Run DB, in this case the overall size ov Run II Event/Run DB is of the order of 20-30GBytes, i.e. could still easily fit into 1 disk.

The advantage of this approach is that it allows to make Event/Run DB small.

A7.b: Another solution is to use a commercial relational database (Oracle, for example) and its tools to keep pointers to each event in RUNDB. In this case Event/Run DB becomes large, but the data access in pick mode could possibly be made more efficient.

Q8: *It appears that the random access features provided by the ROOT file format, whereby the Kth element in any branch of a Tree can be accessed easily are not specially useful unless one has an good metadata catalog which maps the Run/Event selections one is looking for into the appropriate K for the appropriate Tree/File Chain. Does your proposal claim that:*

Q8.a: *this can easily and efficiently be done using the features provided in the ROOT file system and will work well when we have 1.8 billion events and several million files and tens of users locally and at remote institutions possibly wanting to browse this catalog?*

A8.a:

* Offline RUNDB could easily be implemented using the tools provided by ROOT system. **However we do not claim that this is the best possible solution** . NA49 experiment has implemented RUNDB based on mSQL on a timescale of 1 month. Using commercial relational databases like Oracle could also be more advantageous (see answers to similar questions). Performance in multiread mode is not an issue, so multiple users could easily browse the catalog without interfering with each other.
* Because each event is identified not by one, but by 2 numbers: its run and event numbers, 2*10**9 is not a characteristic parameter for the search problem. First one looks for a given run, then for a given event number within this run. Taking for an estimate 10^5 events per good run CDF will have about $n \cdot 10^4$ runs. Presence of 2 keys makes search much more simple and efficient procedure.
* talking about the organization of the files in the datasets, we note that existing facility of file chains (implemented in TChain class) has demonstrated to work well up to a few hundred files in a chain with files in the GB range. The ROOT team is planning to extend this hierarchy, so it would be possible to define TChain of TChains (technically this extention is fairly straighforward). This was not required up to now because the amount of disk space (say 300 GB for NA49 miniDSTs) is a natural limitation for today.

Q8.b: *ROOT implementation of such a catalog is merely a proof of principle? What limitations do you see in using ROOT for this cataloging feature in terms of searches over large amounts of data (unless all searches are limited to Run/Eventnumber metadata only) and multiple writers needing to update the catalog simultaneously? What limitations do you see in using ROOT for the cataloging feature in terms of access to information from remote institutions? Why would you not use a non-ROOT relational or object database for such a catalog function?*

A8.b: This certainly is a proof of principle, but not only. If CDF will decide to use the commercial database software (Oracle etc) or public-domain databases (MSQL) we'll face a problem of distributing the database to the remote sites. We could extract information from Oracle RUNDB and create read-only clones in ROOT format to be distributed to the remote institutions thus solving licencing problem.

Q9: *How does the Event/Run catalog handle reprocessing of data? Does it require that the full catalog be re-written? If so, does this require the data access system be shut down and for how long?*

A9: Reprocessing of the run adds to the RUNDB one more record of run processing history, which becomes the default one. This doesn't require rewriting the full catalog. With RUNDB software supporting write locks there is no need to disable the system because of the catalog update. When replacing the old versions of the datafiles on disk with the new ones one could disable the access to the corresponding part of central disk repository to avoid conflicts with the users. This doesn't require disabling the whole repository.

Q10: *What is the mechanism by which Event/Run catalog information is passed to the analysis job? Are scripts written out from the run-catalog management system? Does the input module of AC++ directly access the catalog? Who develops and maintains the run-catalog management system?*

A10: One could envisage different forms in which user specifies the requested data:

- debugging mode: several events from a given file are requested or several specified files are requested
- the whole dataset is requested
- event pick mode: events with given run/event numbers are requested

Translation of these requests into filenames and run/event numbers in these files to be read should be done by AC++. It could be a core of AC++ which talks to RUNDB, it could also be an input module. Implementation doesn't seem to be a problem, a decision on where to put this functionality has to be made in cooperation with AC++ developers.

- **Technical comment:** one of the possible implementations could be done via a TSQL class in the ROOT system. The list of files, TChains or TSelectionList objects for events to be processed could be automatically generated from RUNDB via a TSQL query.

Q11: *Do you envision the same administrative tools being used on the physics workgroup servers? For private workstation analysis?*

A11: yes. Because ROOT has already a client-server messaging system built-in, the TSQL class must work transparently in a networked heterogeneous environment. However, as explained later, the interface to both HSM and RUNDB must also be transparent.

Q12: *Have you considered the option of putting the Event/Run catalog into a commercial database such as Oracle?*

A12: Yes. As we explained above we see many advantages of using commercial databases like Oracle. However MSQL could also be an adequate candidate. One could also imagine having RUNDB an Oracle engine at FNAL with read-only clones of this database implemented via the public domain MiniSQL for the distribution to the CDF remote institutions.

4 RELATION TO USER SOFTWARE

Can you tell us where things stand on how the TRYBOS BANKS classes can be accommodated with ROOT? What changes if any are required? To be explicit:

Q13: *Section 3.1 of your document defines the data model of a CDF event as basically limited (as it is today using TRYBOS/YBOS) to a flat collection of banks. Would you envisage CDF removing some of these restrictions on its event data model if a ROOT data model and file format were adopted?*

A13: The ROOT system supports a general C++ object model. The ROOT container classes provide support ranging from for simple flat structures with linked, doubly linked lists to more powerful classes such as sorted lists, object arrays, balanced trees, etc. The CDF TRYBOS event model can be implemented in a straightforward way. The present CDF Run II data model has essentially been defined by 2 major boundary conditions:

– requirement to the system to be able to run FORTRAN modules, which were supposed to be the majority of the code.

– the only one persistency mechanism (YBOS/TRYBOS) was available.

Now a scenario with the majority of the code being written in C++ is becoming more and more realistic. This along with the OO persistency mechanism provided by ROOT would allow to remove some restrictions on the event data model. If there were just a few FORTRAN modules in the system, we could imagine CDF adopting a general C++ object model where FORTRAN modules request the necessary banks via a set of simple API. These API would convert C++ objects into the banks in the beginning of the module and copy them back in the end. ROOT data and persistency model give us an opportunity to rethink the existing CDF event structure, however ROOT in no way would limit us if CDF will decide to continue with today's flat model.

Q14: *Where are the limitations of the ROOT data model from your perspective? Does it support making persistent all possible C++ transient objects and object collections? What atomic data members does it support?*

A14: In ROOT, any transient object may be made persistent without changing the class model. This is an essential feature of the system, as opposed to ODBMS systems where differentiation between the transient and persistent capable classes is necessary because of performance-related reasons. A ROOT object becomes persistent with a user operation such as *object.Write* or *tree.Fill*. Vice-versa an object (or collections of objects) is read in memory via a user operation such as *key.Read* or *tree.GetEvent*. The Streamer function for each class is generated automatically by the **rootcint** preprocessor. This function may be modified by the user to support schema evolution. The atomic data members can be:

– any basic data type: char, short, int, float, double.

– any C structure

– any C++ class

– array of basic types

– any ROOT container (TObjArray, TList etc)

– a pointer to an object. Root I/O has a mechanism to save only one copy of an object in case of multiple references to this object.

– a pointer to a special Root container class TClonesArray. This class has proven to be very useful and efficient to support a large number of identical objects.

- **Limitations:** presently ROOT doesn't support persistent templated classes. Support for STL containers is planned to be added to **rootcint**, the preprocessor generating automatically the code for the Streamer function.

Q15: *C++ definitions for raw data classes (and presumably soon!? analysis banks) are maintained in the "BANKS" package. These only include information necessary for TRYBOS, and are set up in such a way that persistence is transparent to user code. What if anything must be done to these definitions in ROOT to make a class "persistent capable"? How would this be managed so that persistent and non-persistent versions track each other?*

A15: To make class T "persistent capable" user has to provide a T::Streamer method which does the conversion between the run-time and (persistent) representations of the class. Therefore any class described in the Banks package could be easily made compatible with the ROOT persistency mechanism in a way completely transparent for the user.

- We'd like to make the following comment. Offline software deals with 2 significanltly different types of objects:
  - the ones for which "run-time" and "on-tape" representaitons are the same, for example RAW data
  - more complex objects like tracks, for which we do not want to write all the (redundant) run-time information out (example: TRKS vs QTRK).

  TRYBOS doesn't distinguish between these 2 types, run-time and on-tape representations are the same for each object. For this reason banks give adequate run-time representation for the RAW data, but not for the more complex analysis objects, like tracks, which we want to compress when writing them out. Existence of routines like TRKSUN, CTCSUN in Run I code (which unpack the bank contents into the FORTRAN common blocks) is the best proof of it. ROOT persistency mechanism via Streamer functions makes a distinction between run-time and persistent representation of the object, which is an advantage over the TRYBOS persistency scheme.

Q16: *How would the above bank definition mechanism work in conjunction with Fortran code, which must access YBOS banks? Would you propose building a CDF i/o wrapper or layer and a new form of API (which hides ROOT i/o underneath) and which performs a copy of designated parts of a ROOT tree into the Ybos bank collection structure accessible by the Fortran, much like TRYBOS does things today? Or what would you propose for the mechanism by which the Reconstruction program or non-ROOT analysis program can get access to in-memory objects, collection of objects, or banks formed by reading a ROOT tree. Would a YBOS bank be a specially designated user defined class of data in ROOT?*

- With the good progress on track reconstruction front it becomes more and more real that the absolute majority of the offline code will be written in C++ and not in FORTRAN. To allow use of the FORTRAN modules (which are supposed to be the minor part of CDF offline code) we propose to use the scheme adopted initially for AC++ FORTRAN modules: each FORTRAN module in the beginning of the event processing requests the banks it is going to use. These banks are created in YBOS memory by special API. In the end FORTRAN module requests copying the necessary data back from the YBOS banks to the objects they should be stored in. In this scheme FORTRAN code does not require any modifications, the price to be paid is additional memory-to-memory copies.

Q17: *Where and how would the master definition of a ROOT object (bank) be maintained? Would this be in the form of C++ header files? How would the position of an object in a branch of a ROOT event tree be specified in a global way?*

A17.1 C++ header files are very appropriate place for maintaining the definitions of the objects (including ROOT persistent objects). These files must be correctly commented. If they are, the ROOT automatic documentation tools can be used to easily navigate in the class structure and hierarchy. The design of a ROOT Tree and associated branches are in the class code itself. This provides more flexibility as it can be under control of some production flag. Default methods are provided to generate automatically a ROOT Tree from the class definition. This is possible because ROOT has access to the full Run Time Type Information (RTTI) generated by the **rootcint** utility.

A17.2: There is a very important difference between ROOT persistent objects and (TR)YBOS banks: in case of banks all the data members are persistent by definition. This imposes a pretty tight limitation on the internal structure of the banks and thus on the data model:

  - banks can't contain pointers to other objects, ROOT objects - can;

- their contents has to be optimized with respect to requirement of data storage efficiency which pretty often contradicts the requirements of run-time efficiency: one doesn't want to store on tape px,py,pz,energy, phi,eta,mass of the track whereas at run-time it makes a lot of sense to calculate all the variables just once and to store them in a track data structure. ROOT objects do not have this limitation.

Q18.1: *For the AC++ ROOT I/O modules, how would it be determined what parts of the event to load (e.g. to avoid the parts on tape)?*

A18: Reading only parts of the event data is implemented in ROOT by activating/deactivating different branches. We suggest to use AC++ user interface to define set of branches to be activated in this job by default. The corresponding AC++ command could have a form

```
input file r58895.dat/branches=(head,trig,cal)
```

By default all the branches are read in memory. Different AC+ modules modules could also specify the additional branches they need to enable/disable. This feature could either be hardcoded or implemented via talk-to and would give a natural and elegant implementation of the deferred input. We'd like to mention that by moving this functionality to AppModule class (all the AC++ modules inherit from it) this feature could be made transparent not only to the users but also to the module writers.

Q18.2: *Would there be a convenient way, without requiring modification of existing TRYBOS based code, to avoid reading the complete disk resident event if not accessed by the code?*

A18.2: Yes, this is possible. See comments below.

- **Technical Comments:** CDF event could be described as follows:

```
class CDF_EVENT :  public TObject {
  TTpid* Tpid;                // TPID bank
  TLrid* Lrid;                // LRID bank
  ....
  TClonesArray* Cmuo;         // Array of CMUO banks
  ...
  TObjArray*    OtherBanks;
};
```

Most of the banks (described explicitly) could be accessed in direct mode using all the advantades of ROOT I/O, while the last data member allows to add to the event an arbitrary user-defined bank without changing I/O module. Data from the active branches are read in memory and copied into Trybos record. As Trybos itself doesn't know anything about the banks which should be present in the record, this partial read is fully transparent with respect to Trybos.

If the next analysis module in the processing chain activates another set of branches, ROOT reads only the part of the event data it hasn't read yet and hands the corresponding banks to Trybos. Thus the incremental read is also transparent to Trybos.

Q19: *What opportunities for deferred I/O are presented by suitable organization of an event tree into branches which are only actually read in when that part of the event is referenced? To make use of such a feature would it not again imply use of a CDF-specific API built on top of the root i/o API? What might such an API look like to the user? Would you envisage the current Trybos API as also being provided by this interface (only with a different implementation under the hood to that used in Trybos today)? Have you talked with Rob Kennedy about the feasibility and ease of doing this? What is the answer?*

A19: These questions are mostly answered in [A18]. The CDF-specific API required would be just an interface between the CDF framework and already existing ROOT API which would do the real work.

ROOT already supports branches of a Tree residing in different files. Only referenced branches (files) in the list of declared active branches will be read. The ROOT team is currently working on the PROOF (Parallel Root Facility) system to analyze the user analysis code to generate automatically the list of branches that are active.

We discussed the issues above with Rob Kennedy and concluded that this scheme suggested in **A18** could easily be implemented. We also discussed the ways of future optimization of the interfaces between ROOT, Trybos and AC++ , we didn't find any principal problems and came to the commmon understanding of the implementational issues involved.

Q20: *How exactly would you imagine keeping control of schema evolution? Is a schema version built into each ROOT object (bank) definition? Would the ROOT object/bank owner write their own Streamer method only in the case where the schema had evolved, and otherwise use the built-in streaming methods of ROOT? In the end would this not end up with everyone having to write Streamer methods - just like in the LWDB approach and the objectivity approach where the burden of converting from the persistent format to the in-memory format appears to be placed heavily on the bank/object owner?*

A20: Conversion between the persistent and run-time representation of the object is always necessary if disk/tape usage efficiency is an issue. Any offline system will end up with having many of them. Even in Run I system we had routines like TRKSUN etc, which unpack banks into FORTRAN common-blocks and functionally are equivalents of Streamer functions. The advantage of ROOT is that it defines a natural place where this conversion has to occur thus minimizing the probability of introducing a bug in a conversion routine.

Experience with ROOT applications so far indicates that the majority of Streamer functions is generated automatically. Obviously with time, the percentage of Streamer functions implemented by hand will grow. There is certainly a compromise between implementing a completly new class (or derived class) or introducing patches in an existing Streamer function.

Q21: *How much effort is required in using ROOT to evolve to Offline versions (viz. 7.23 to 7.24)?*

A21: We suggest to treat ROOT as an external package in Run II environment. We consulted Robert Harris (CDF release manager) on this issue and the common conclusion was than that it is a minor effort which could easily be done by CDF software management team.

Q22: *What are the compiler/C++ library/platform issues associated with the proposal? Which are supported? What C++ class library must be used? If not one of the currently supported ones, can that be hidden?*

A22: The ROOT system itself is currently available on most Unix platforms and WindowsNT/95. An OpenVMS version is currently under test at Austin University. The following compiler/machine options are supported:
Linux/Intel: Redhat 4.2 g++ 2.7
Redhat 5.0 g++2.7, g++2.8, egcs, KAI
Linux/Alpha: currently under test.
SGI Irix 5.3: Native CC
Irix 6.4: Native CC
Irix 6.2: g++, KAI port completed by Apr 24 1998
HP-UX: 9.05, 10.20 with CC and aCC compilers.

Solaris/Sun: with CC and gcc.
Solaris/Intel: with CC
Alpha/Unix 4.0; both CXX and g++
AIX4.1: xlC
Windows/NT/95: with VisualC++

Linux/KAI port is close to completion.

Binaries for nearly all these combinations are available from http://root.cern.ch

ROOT itself does not reference other class libraries. The only "system libraries" used by ROOT are X11 on Unix systems and WIN32 on WindowsNT.

5 HOW MUCH DOES A USER NEED TO KNOW?

Q23: *Please expand on what a physics group would do, what expertise would be needed, how long it would take to create a dataset, how long to remake, what happens after re-processing, how would other physics groups on different(?) servers use it?*

A23:
- As from the data handling point of view ROOT-based system is equivalent to Sequential Access Model (files are accessed sequentially, one after another) stripping/selecting the secondary dataset needs user to run the AC++ selection job and to specify the input and output files. This is exactly the same procedure we are doing with Run I AC.
- To automate stripping procedure in case both input and output datasets are ROOT chains one needs to develop naming conventions and conditions when one file in the output chain has to be closed and the next one - opened.
- Adding secondary and tertiary datasets to RUNDB should **not** be done automatically and requires administration.

As output of selection/stripping job is a file or a chain of files, users working on different platforms could access it in several ways.

- user can copy (by ftp or rcp, for example) the datafiles to local node and read them locally
- one can use ROOT daemon (rootd) and open the datafiles residing on a remote node via TNetFile class:

```
TNetFile* file = new TNetFile("root://cdfsga.fnal.gov/CDFrun1.root");
```

- one can use the PROOF service.

Q24: *Where does the interface between experienced managers and physics groups users fall, e.g. what files, data, etc. are handled by the central event processing/cataloging system, and where do the users take over?*

A24: Besides the purely administrative matter of deciding which files/datasets have to be maintained in the central repository and which do not need it, the difference between accessing the data residing in the central repository and the private data has to be handled by AC++ and is transparent for the user. This difference exists only if access to the data stored in the central repository has to be monitored.

Q25: *How does a user actually start accessing a Physics DataSet which is in a chain of files not resident on disk. What do they have to do before running their analysis job? What happens as a result of API calls within their analysis program?*

A25: The most interesting case here is a dataset maintained in the central repository, i.e. under the control of HSM. In this case AC++ job queries the File/Dataset DB to construct the chain of files to be processed. Then AC++ sends a request for these files to HSM. The rest has to be done by HSM software. HSM should inform AC++ when the first file is ready to be processed, so AC++ should not wait until the whole chain is staged to disk. HSM continues staging in asynchronous mode informing AC++ about the files ready to processing. In case of very large chains this could also help in optimizing the use of the disk space. This mechanism is hidden from the user, who only requests the dataset.

- **Technical Comment:** When a list of files is identified via RUNDB, a query is submitted to the HSM system directly by the TFile or TNetFile constructors.

Q26: *Where does information on luminosity reside in the system? If a user analyses a partial dataset, how is the luminosity determined?*

A26:
  - This information is part of RUNDB (its File/Dataset section). Extracts from RUNDB can be made periodically to generate a summary run information in the form of a simple ROOT Tree file.
  - ROOT provides an interesting possibility to do luminosity book-keeping at a file level. In addition to the Tree containing the events one could write into the data file a second object - a luminosity record, which could be accessed directly without reading the whole file. Luminosity record could be added to a ROOT file after the production. In this case all the secondary, tertiary and user-selected datasets would automatically contain in them the value of integrated luminosity corresponding to them.

- **Technical Comment):** In an analysis program, the user has the choice in making a direct query to RUNDB via TSQL or a query to the summary ROOT RUNs file via statements like: (case of ROOT Run summary file)

```
TFile   runs("CDFRUNS.root");             // connect the summary run info file
RunInfo *run = new RunInfo();             // create a local runinfo object
TreeRun.SetBranchAddress("RunInfo",&run); //connect data base to runinfo object
TreeRun.GetEvent(run_number);             // read run_number info into object run
float luminosity = run->Luminosity();     // get luminosity for this run

via statements like: (case of access to RUNDB via TSQL class)

TSQL sql;                                 // create an SQL object
float luminosity = sql.Query("select luminosity run_number");
```

Q27: *It has been stated that using ROOT is not possible without knowledge of C++. Does this statement remain strictly true if for some reason a user is not using ROOT as the replacement for PAW?*

A27: We see AC++-to-ROOT interface as hidden from the AC++ user. If a user works with only FORTRAN modules no knowledge of C++ is necessary.

Some basic knowledge of C++ is necessary to write C++ macros for ROOT. Many examples of simple analysis tasks (a la PAW) are already provided in the ROOT tutorials or test programs. Experience has shown this minimal level is not a problem and PAW users have no difficulty to move to C++ ROOT macros as a replacement of KUIP kumacs and COMIS Fortran. Some basic C++ courses must be organized with real CDF analysis examples.

Q28: *How will the operation of 'cloned' excutables accessing network-distributed disk repositories work for a single user? Transparently?*

A28: PROOF (Parallel ROOT Facility) is currently being developped by the ROOT team in the context of NA49. Other experiments, like PHOBOS at RHIC have also expressed interest in the system. PROOF would allow large databases to be analyzed in parallel on MPP's, SMP's or loosely coupled workstation- or PC-clusters. A PROOF session consists of one master server, running on the user workstation, and several slave servers, running on the CPU's local to the disk repository. To query a database containing many millions of objects the master sends small work packets (a packet contains an object index and the number of objects to be processed) to the slaves. After processing the objects in a packet a slave asks a new packet until all objects in the database have been processed. The packet size (number of objects) is dynamically determined by the master depending on the time a slave took processing a previous packet. Typically a large logical database consists of many physical databases distributed over the nodes in a cluster. The packet begin object index is determined such that the objects reside as much as possible in a database local to the slave. Using this adaptive work distribution technique the parallel performance of the system scales well up to a large number of nodes. The current implementation is on a farm of Pentium —— running Linux. Interaction between the master and the slave nodes is transparent for the user.

6 INTEGRITY OF DATABASE

Q29: *In general, compression algorithms result in less robust output data.*
- How easy is it to corrupt the ROOT file format?
- Does the data contain checksum information to sense such corruption?
- Does corruption result in the complete file being lost?

A29:

- The compression algorithm in ROOT is based on the standard GZIP algorithm. Not one single case of failure of this algorithm has been reported so far. The algorithm is really standard, well know and secure. It includes its own checksum mechanism.
- In ROOT files, directories/keys are never compressed. This means that if part of a file (i.e. disk sector) is corrupted, one can always recover what is not corrupted. An automatic recovery mechanism is built-in the ROOT file system (procedure TFile::Recover).
- Another important point not addressed by this question, but relevant. In case one writes a ROOT file or Tree, it may happen that the job aborts for multiple reasons (crash, overflow, time limit, space limit, power shutdown etc). ROOT has an autosave facility that guarantees the integrity of the file up to a given event number. Buffers still in memory may be lost.

Q30: *A fair amount of discussion in the proposal is given over to the importance and advantages of making full use of compression techniques. However one might ask is this crucial to the viability of the ROOT alternative; what happens if we are either unable or unwilling (assume for the sake of argument, for good reason) to either fully compress or minimize the overlaps?*

A30: Given the present estimates of the disk space available in Run II ROOT compression could be a very important factor. On the other hand, compression is an option that can be activated at the file level or at the branch level. A ROOT file can contain both compressed and uncompressed objects. In the same way, a ROOT Tree can contain compressed and/or uncompressed branches. Therefore if CDF decides not to use compression at all no changes are required in the software - only the compression flags should be set to OFF.

- **Technical Comments:**
  - According to the available benchmarks in non-compressed mode ROOT reads the data about 20% faster then with compression level = 1.
  - The Tree::Print function reports the compression factor on a per branch basis. The TFile::Map function gives full details of the compression for each logical record in a ROOT file.

Q31: *What happens with ROOT file format if a chain of files is defined and one of the files in the chain is missing or corrupted. What utilities or actions have to be provided to recover from this and carry on reading events from that chain?*

A31: When a TChain object is created, ROOT makes a first loop on all the files referenced in the chain. This first pass merges the Tree directories into one single directory. If a file is missing, an error is reported and the file is not included in the chain directory. Chain processing can continue with the missing file/files. User has the control via the return flag of TChain::Add to decide if he wants to stop processing or continue. Only events from the missing file will be missing in the collected statistics.

- **Technical Comments:** The chain processing strategy may be optimized if some (or all) of the files reside on HSM - depending on the overall size of the chain one may not want to prefetch all the files before the job starts processing the first one.

Q32: *What facilities are necessary to back-up disk resident data? Are special administrative tools required? What back-up strategy do you imagine for the disk-resident primary data?*

A32: One could imagine several possibilities:

case 1: if all the central disk repository is maintained by HSM with HSM distinguishing between "permanent" and "temporary" datasets on disk. In this case backing up the disk-resident data should be a part of HSM software. This would be our preferred solution.

case 2: if central disk repository is not a part of HSM, but keeps only permanent datasets which need to be backed-up, a special back-up utility has to be provided. This utility may be a simple interface to HSM supplying HSM with the names of files to be backed up.

7 IMPLICATIONS FOR REMOTE USERS

Q33: *What level of system support is required for a remote institution to install and maintain ROOT?*

A33: Installing ROOT for a supported platform is pretty straightforward. Installation steps include:

- copying ROOT distribution tarfile(5-7MBytes) from the ROOT distribution node (root.cern.ch), which contains prebuilt binaries, unzipping and untarring it.
- a single environment variable (ROOTSYS) must point to the current PRO version of the ROOT system. The control of what is PRO/NEW/OLD is under a user/lab control.
- Normally (from the experience) the installation takes about 10 minutes and does not require system privileges (installation could be done by the user and does not need a system manager)

- The source codes of ROOT could be installed using the same procedure.

The **rootdev** mailing list can be used to report problems to the ROOT team. Practice shows that the response time varies from several hours to 1 day. The **roottalk** list can be used to discuss items of general interest.

8 THE INTERFACE TO HSM

Q34: *Do you plan to pre-fetch files from the HSM, or will tape mounts be initiated when the first request for tape resident data is sensed? If the strategy is the first, what initiates the tape staging? If the strategy is the second, what performance penalty does it incur?*

A34: This is part of the HSM software. In case a single file has been requested and a TFile (Root file) object is created, the TFile constructor waits until the staging operation is completed. In case of a TChain, the request to the HSM manager could be implemented at the end of the chain definition so HSM does the optimization to minimize the number of tape mounts. The details of the interface depend on the choice of HSM.

Q35: *Are any resource management tools forseen to control requests to access the HSM? Suppose a user submits a job that requires 500 tapes to be mounted. Does the system sense this? Does it inform the user? Does it allow the job to continue under all conditions, or does such a request require administrative approval?*

A35: Data handling system should definitely have resource management tools. HSM resource management tools should either be a part of HSM software (in case it comes with HSM) or a part of CDF interface to HSM, in which case it should be developed in the collaboration. One of the strategies of disk space management is discussed in CDF-4497.

Q36: *How are files pre-fetched onto disk for Farm processing and which part of the proposed data handling system implements that functionality? At what point do newly created ROOT files get archived onto tape in the Robot and which part of the proposed data handling system implements that functionality?*

A36: There are 2 possible scenarios of production farm organization:

- In one of them (CDF Run I scenario), when the Production Manager distributes events to the processing nodes prefetching could be done by the Production Manager only.
- another scenario assumes that processing nodes receive not the single events but the data files. In this case prefetching in principle could be done by the corresponding node, but we believe that in this case it is also more efficient if it is done by the Production Manager, which distributes prefetched files to the processing nodes.

Archiving should be done in asynchronous mode with respect to production. When an output file is written and closed, ROOT dialogs with the HSM software to stage the file if it is necessary. If the HSM software is not clever enough to migrate data sets on statistics considerations, it should be easy to inform RUNDB by a manual intervention (or automatically from the production system) that a file has to be migrated to tape. A special Spool Manager (a daemon or a cron based utility) could look at regular intervals the files to be migrated. This process could also take care of combining data sets on the same tape. After the file is archived, Spool Manager updates RUNDB.

Q37: *Presumably the natural unit of tracking of work performed by a Farm system is a file? How do you envisage the tracking of Production activities to be done? Where do you keep track of exactly which version of the Reconstruction and which supporting calibration and alignment data were used in producing a certain ROOT format Dataset consisting of a chain of files? What happens when you reprocess the data.*

A37: This is the task of RUNDB. When a new output file is produced a corresponding processing record which includes all the relevant information about the alignment/calibration is added to RUNDB.

We assume that a unit of (re)processing is one run. When a run is reprocessed another version of run processing record is added to RUNDB which becomes the default one. In case of central RUNDB implemented in Oracle one could imagine extracting info out of it and creating read-only clones in ROOT format to be distributed to the remote sites not equipped with SQL experts.

In addition to the Tree with events one could also store in the datafile a file processing record (with luminosity etc info). In case of ROOT this record would be a second object in a file and could be retrieved in a direct access mode without reading the whole file. This gives a natural implementation of a file-based luminosity bookkeping.

Q38: *If all the objects for a RAW event were to stored together in the same file on the same tape this would facilitate production. Is this a problem?*

A38: Understanding the production needs we propose as a baseline option to keep all the RAW data on the same branch (together). If for the needs of reprocessing, for example, CDF will find it more efficient to split part of the RAW data into a separate branch (to minimize amount of tape mounts, to optimize disk space usage etc) ROOT provides a mechanism to do it.

- **Technical Comments):** With a ROOT Tree, the user can control the file where the branch resides. By default a branch is written in the file where the Tree header has been created. To assign a branch to a file, one writes a statement like:

```
branch.SetFile(file_name);
```

When a Tree containing branches residing in different files is read, the Tree header remembers the name of the file in which each branch has been written. The user has the possibility to change the name of a file, move branch files to a different machine, etc. Note that using the existing Tree::Merge function, several Trees can be merged into a new Tree, the original Tree having branches on different files or vice-versa.

Q39: *Is there any form of optimization of use of the Robot envisaged, in terms of organizing the work to minimize tape mounts? This would require an interface to the HSM's database of physical tapes. How would this be done and in what part of the proposed system?*

A39: Normally this should be part of the HSM software. As already stated above, if the HSM software is not clever, this optimisation could be added to RUNDB software.

- **Technical Comments):** In case the optimization is being done by HSM software, ROOT TChain class could be optimized to call a special HSM API which would pass to HSM names of the files included into the chain.

Q40: *What overheads (in writing and in reading) are incurred by writing out an event superbranch into n branches, each in a separate file? What are these overheads as a function of n? Have you, or anyone, performed these tests? Can we do so?*

A40: This overhead is minimal. There is practically zero overhead in ROOT itself: each I/O buffer has a file pointer associated with it, so the ROOT performance doesn't depend of whether all the pointers correspond to the same file or not. There might be an overhead related to the ability of operating system itself to support efficiently multiple parallel writes to the physically different locations on disk. For exactly the same reason if branches reside in different files there is a clear gain in **read** performance: data on the branches would be nicely buffered into contiguous blocks so it would be read in faster. The conclusion is:

- the read performance must be better.
- the write performance must be benchmarked in a real configuration. The tests performed by the ROOT team do not show a performance degradation in writing, but they were only tests and no measurement has been done in a real production configuration.

Q41: *The essential difference between the Lightweight database approach and a Sequential File Access approach using ROOT file format appears to be in the apparent fine tuning on an event-by-event basis, of the collection of banks which are present and read in when iterating over a collection of events. If one decided that this level of 'dynamic PAD definition' was indeed necessary and effective in optimizing disk usage please explain how an event tree suitably organized into branches, each written to different files, could emulate the effect described by the Lightweight database. (if possible?) Please comment on the reclustering/file rewriting software which would be needed.*

A41: "dynamic" reclustering of the data at the single event level which is the major distinctive feature of LWDB approach requires much more complicated I/O mechanism than that implemented in ROOT and the practical advantages of it still have to be demonstrated.

ROOT approach (based on many years of HEP practice) assumes that it is possible to cluster the data statically into pieces which logically have to be read/used together always or almost always. It is very efficient in case of a sequential loop on events. It remains very efficient in case on read only a subset of all events, or a subset of all event branches or the two combinations.

Having branches written into different files one could keep the mostly used branch(es) on local workstation with the rest of the data residing on the workgroup disk server. This would require only changing the procedure of parsing the filenames corresponding to the branches when reading the event Tree, so the remote branches would be associated with TNetFile's rather than with TFile's. Also the procedure of copying the Tree from one file to another should be modified to add an option of actual copying of only several branches and storing the filenames instead of the actual data for the rest of the branches.

This wouldn't be exactly equivalent to the "dynamic reclustering", but quite close to it.

9 GENERAL CONCERNS

It has been stated that the ROOT team has in the past expressed an interest in participating in a CDF II implementation of ROOT.

Q42: *Is this a true statement, and if so what does it mean? Who is now actively engaged in the ROOT team and do they have an interest in interfacing what could be claimed to be a stand-alone product with AC++, the event/run catalog, production, etc.*

A42: (by Rene Brun, leader of ROOT project)
The ROOT team is interested in a close collaboration with this CDF project. We see CDF as the first experiment collecting a huge amount of data similar to the LHC experiments within a factor 2 to 5. This will be a very valuable experience many years before LHC data taking. In particular, getting experience in interfacing ROOT with a RUNDB and HSM software is regarded as extremely important. The ROOT team is willing to invest time in development and support of this vital activity for their project.