Development of a new indexing system for ATLAS entry data

Nikita Dulin[1]
[1]University of Chicago, 5801 S Ellis Ave, Chicago, IL 60637

August 17, 2018

Abstract.

The ATLAS experiment at CERN uses ROOT to process and analyze data. ROOT stores data in structures known as TTrees and TBranches. Currently, ATLAS uses the entry number into these containers for referencing recorded event data. This can pose a problem if the final output file is written by a different process than those producing the data, as entry numbers are reset while merging. This has prevented ATLAS from using more recent ROOT developments that utilize in memory data collection (such as TParallelMergingFile). New code was added to Athena, the data processing framework for ATLAS, and existing code was modified in order to associate index numbers with recorded events and to permit new referencing capabilities.

INTRODUCTION

The ATLAS experiment [1] has recorded approximately 200-300 petabytes of event data since its inception. This has allowed researchers the ability to make groundbreaking discoveries, such as the detection of a particle consistent with a standard model Higgs boson in 2012. Managing this data, however, can take quite a bit of time and processing power due to its size. It is often faster to use multiple clients to reconstruct raw data and to merge the produced data while it is still in memory. Due to the nature of the indexing of event data at the ATLAS experiment, this is currently not a viable option. ATLAS uses ROOT to store all of its derived data in structures called “TTrees” and “TBranches”. These data storage structures use “entry numbers” for access, which get reset after a merge. If merging were to occur in memory, it would be impossible to discern which event data came from which clients, which is necessary in order to be

able to keep track of the data. The current method at ATLAS of merging includes having each client write their data to disk before reopening the files and merging them. It would be much more efficient to perform in memory merges, especially given the size of the data. The solution is to add a separate way to index that does not get reset upon merging.

BACKGROUND

A. ROOT

ROOT is a C++ based data analysis framework [2] developed primarily at CERN for the purpose of handling large sets of data for physics analysis. ROOT can also create various types of 2D and 3D plots and histograms, which make it not only popular but very useful in the physics community. Some example plots produced by ROOT are shown in Figure 1.

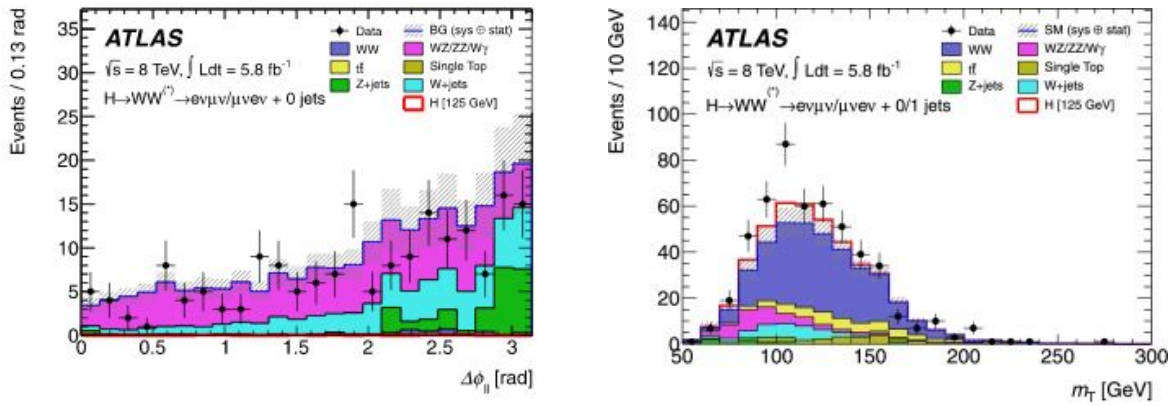

Figure 1. Some plots presented in the ATLAS publication announcing the discovery of the Higgs boson.[3]

1. ROOT data management

ROOT stores its data column-wise in structures called “TTrees” and “TBranches”. A TTree can have an unlimited number of TBranches. For example, ATLAS’s main data tree uses approximately 2000 TBranches. In the context of physics analysis, each TBranch corresponds to a particular characteristic of whatever is being measured or detected, and each is filled entry by entry. Within any TTree, each TBranch must have the same number of entries, and each entry is identified by an “entry number”. The first entry number in any TTree or TBranch is zero, and each subsequent entry number is one greater than before. A good way to think of these data sets is as sets of n-tuples, where n is the number of branches. There are as many n-tuples as entries, and each component of any n-tuple is given by the value found at the entry number of that particular n-tuple in each TBranch.

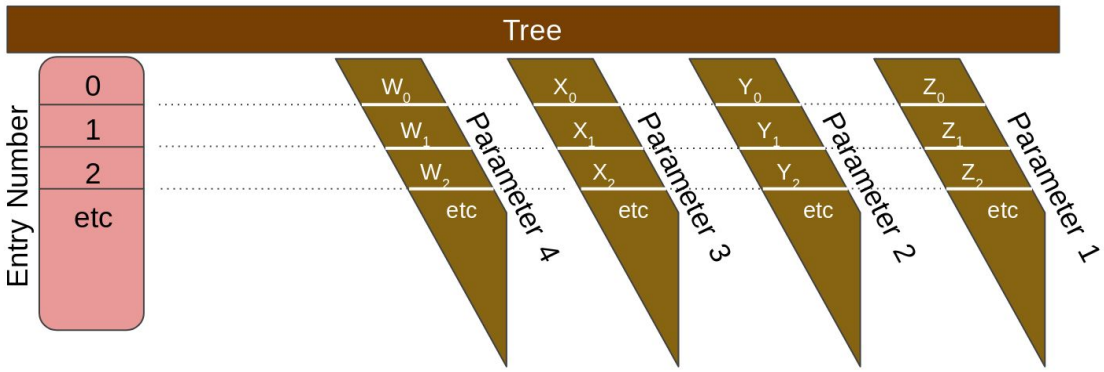

Figure 2. Graphical representation of ROOT data structures TTree and TBranch.

B. ATLAS

ATLAS is one of the four experiments running at the Large Hadron Collider at CERN, Geneva. The ATLAS experiment records a few thousand specially selected particle collisions (events) every second, which get sent to offline servers around the world for analysis. This adds up to quite sizable data sets. While there are no precise calculations, it is estimated that ATLAS is outputting approximately 100PB of event data per year. ATLAS stores its event data in one tree with approximately two-thousand branches. The only currently implemented way to recall events and event objects is using the entry number.

PROBLEM

ROOT has functionality (TParallelMergingFile class) that allows for in-memory merges. In-memory file merging is much faster than writing to files before merging. Writing to disk can be time-consuming, especially considering compression (and decompression, since the files must be reopened in order to be merged). With parallel in-memory merging, there would be several clients sending produced data to a server to be merged. The ATLAS collaboration would benefit from this functionality, but given that the only indexing system for recorded event data is with the entry number, in-memory merging is not currently possible.

The problem lies in the ability to track event data being produced by clients. Unlike having each client write to disk first, thus allowing one to know which files came from which client, before merging, it is impossible to tell which event data comes from where from an in-memory merge. This is because when clients send data to a server to be merged, the server decides in which order to merge all of the incoming data. Moreover, since entry numbers in a tree count up sequentially from zero, there can be no duplicates, and thus the entry numbers of the data are reset. What was entry number 0 on one client may now have a different entry number.

If someone were to notice a particular event reproduced on one client and wished to recall it on the final output file, they would be unable to do so, unless they sorted through all of the data looking for that one particular event manually. In other words, the ability to reference data from the clients is broken this way, thus preventing in-memory merges for ATLAS data.

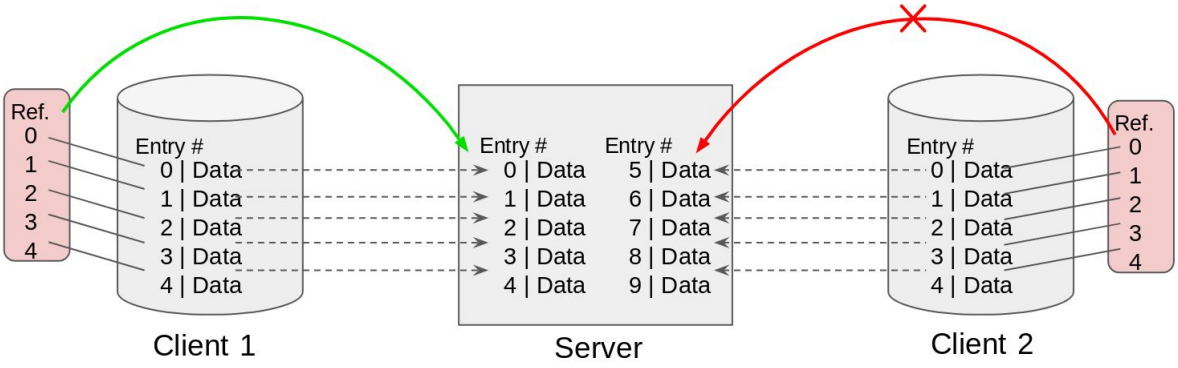

Figure 3. Graphical representation of broken in-memory merge described in PROBLEM section.

SOLUTION

The solution to the issue is to add a separate branch to all of the trees in ATLAS' data sets. This branch, which we can refer to as an index branch, will be filled at the same time the other branches in the tree are filled. Each index branch entry will have a unique number that will not be reset or modified upon merging by the server. Moreover, the index branch entry will depend on the server producing the data in order to maintain uniqueness and to be able to differentiate between client data on the server. Finally, to use this branch as an index, we utilize ROOT's TTreeIndex class, which marks the branch as an index.

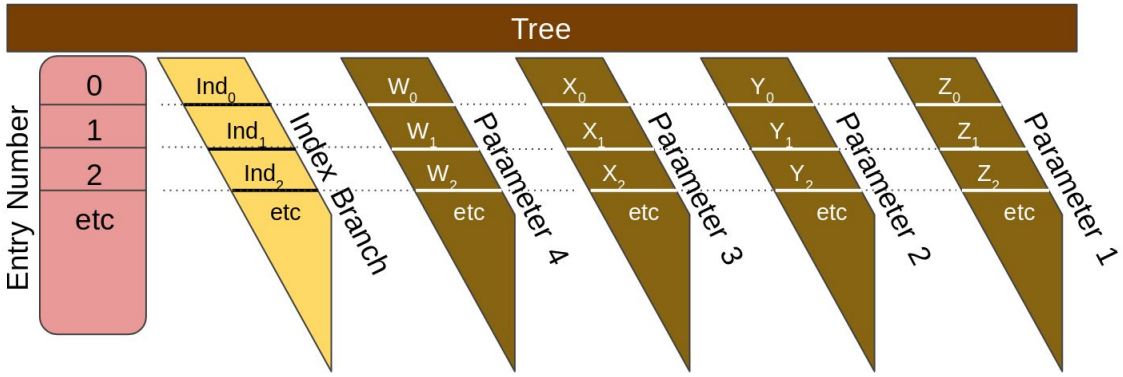

Figure 4. Graphical representation of ROOT TTree structure with added index branch.

As mentioned previously, the entry numbers at the server will count sequentially up from zero, they, as a whole, will not correspond with any of the entry numbers at any of the clients, and thus the data will be jumbled. Now, however, any given entry will now have an entry in the index branch, which we can refer to as an index number, associated with it at the server. Since this index number does correspond with the same index number associated with that entry at the client, one will be able to identify this entry at the server.

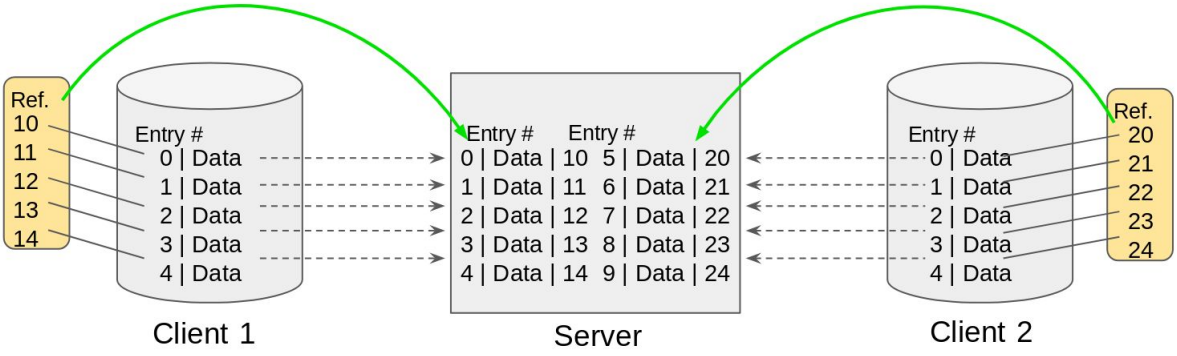

Figure 5. Graphical representation of fixed in-merge as a result of added indexing capability.

TTreeIndex class functions allow us to use our index branch as an index. Once we declare that our index branch will be used to index our TTree entries, we can use the index number through TTreeIndex functionality to return the entry number in the server TTree that corresponds to the index number. Since we now have the entry number, we can use that to retrieve the data we are looking for.

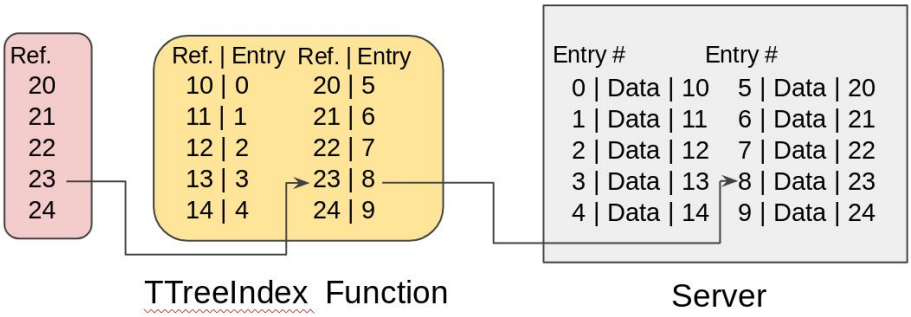

Figure 6. Graphical representation of reading using TTreeIndex functionality.

IMPLEMENTATION

This section will go more into detail of how code was added or edited to include this functionality. For those on the ATLAS collaboration who have access to the Athena codebase github, the changes can be found at https://gitlab.cern.ch/atlas/athena/merge_requests/12983.

A. Athena and POOL

ATLAS's data processing framework is called Athena. All of the raw data collected by the ATLAS experiment goes through Athena wherever it's sent to be reconstructed. POOL is an abstraction layer which allows for various functionalities or technologies to be used with the data. Currently, the only technology currently used is called RootTreeContainer. RootTreeContainer facilitates the creation and filling of TTrees and TBranches from input data. It does not implement any special indexing functionality. Rather than directly editing it, however, it was smarter and safer to create a new functionality / technology to incorporate our indexing capability.

B. RootTreeIndexContainer

New storage functionality, named RootTreeIndexContainer, was added and utilized through POOL. RootTreeIndexContainer inherits from RootTreeContainer, but a few functions were modified to include the indexing capability described above. The first of these functions was called transAct.

1. Creating, filling, and assigning the index branch

The function transAct is called, amongst other times, whenever a TTree is created, whenever a TBranch is created, and whenever a TTree/TBranch is filled. We utilized this convenience by incorporating the creation of the index branch in transAct so that the first branch added to any new tree is the index branch. Then, since transAct is called whenever any entries are added to a tree, the index branch is filled as well. How index numbers are assigned is described in the following section.

To assign the index branch to be used for indexing was a simple matter of calling a TTreeIndex function called BuildIndex. TransAct is called one final time just before sending the data to the server to be merged. On this last call, BuildIndex was added to assigned the added branch as the index branch.

2. Assigning index numbers

The function nextRecordId does not exist in RootTreeContainer; it was added to RootTreeIndexContainer and determines with what to fill the index branch. The index number associated with any entry is a concatenation of the process ID (PID) of the client which is producing the data and the entry number of the entry to which the particular index number will be assigned.

Although PIDs are unique, there can be instances of overlap if we only add the entry number to them, since PIDs are so small. The solution to this issue is to bit shift the PID by 32 bits, after which we add the entry number. This way, even though we add the entry number, there will never be any overlap between two clients.

3. Reading from index

Since the index branch had already been assigned at the client to be used for indexing, there are TTree functions that can now be used. Since RootTreeContainer is already programmed to use the entry number to read, it is only necessary to use the index number to retrieve the entry number associated with that entry in the merged TTree at the server. Calling GetEntryNumberWithIndex on the TTree using the index number returns the entry number. After that, the entry number can be used to read the data as it is done in RootTreeContainer.

CONCLUSION

When RootTreeIndexContainer is formally implemented, it will allow ATLAS data to be managed much more efficiently. Moreover, it will give analysts a new method of managing data, which could be helpful in and of itself. There is still testing to be done to confirm the correct operation of RootTreeIndexContainer and to confirm that there is not a considerable performance loss. Preliminary testing, which I have not reported here due to the lack of rigor of the tests, has not shown any considerable loss in performance. Moreover, the functionality proves to be working correctly when writing locally to a file. The ability to use this functionality in a client/server setup is not entirely

operable, due to certain incompatibilities with the current code. These are small bugs, however, and can be fixed in a short time.

ACKNOWLEDGMENTS

I would like to acknowledge my mentor and supervisor Peter van Gemmeren in the High Energy Physics division at ANL. He was happy to help me through every step of the learning process. I learned a lot through the summer and was able to accomplish something tangible, and I would not have been able to do so without his guidance.

I would also like to acknowledge David Malon at ANL for being at my disposal for any questions I had regarding ROOT or Athena while Peter was unavailable. David was very helpful in walking through and helping me to understand any code that I was working with.

APPENDIX A: Entire code of RootTreeIndexContainer.cpp

```
/*
 *   Copyright (C) 2002-2017 CERN for the benefit of the ATLAS collaboration
 *   */
// Local implementation files
#include "RootTreeIndexContainer.h"
#include "StorageSvc/Transaction.h"
#include "RootDataPtr.h"
#include "RootDatabase.h"
// Root include files
#include "TTree.h"
#include "TBranch.h"
using namespace pool;
using namespace std;

RootTreeIndexContainer::RootTreeIndexContainer() : RootTreeContainer(), m_index_ref(nullptr),
m_index_foreign(nullptr), m_index_multi(0), m_index(nullptr) {
  m_index = new long long int;
  m_index_multi = getpid();
}
/// Standard destructor
RootTreeIndexContainer::~RootTreeIndexContainer() {
  delete m_index; m_index = nullptr;
}
long long int RootTreeIndexContainer::nextRecordId()    {
  long long int s = m_index_multi;
  s = s << 32;
  if (m_tree != nullptr) {
          if (m_index_foreign != nullptr) {
          s += m_index_foreign->GetEntries();
          } else {
          m_index_foreign = (TBranch*)m_tree->GetBranch("index_ref");
          if (m_index_foreign != nullptr) {
          s += m_index_foreign->GetEntries();
          }
          }
  }
  return s;
}
DbStatus RootTreeIndexContainer::transAct(Transaction::Action action) {
  if (action == Transaction::TRANSACT_COMMIT) {
          if (m_tree == nullptr) return Error;
          if (m_tree->GetName()[0] != '#') {
          if (m_index_foreign == nullptr && m_tree->GetBranch("index_ref") == nullptr) {
          m_index_ref = (TBranch*)m_tree->Branch("index_ref", m_index);
          }
          if (m_index_ref != nullptr && RootTreeContainer::size() > m_index_ref->GetEntries()) {
```

```
            *m_index = this->nextRecordId();
            m_index_ref->SetAddress(m_index);
            if (!m_treeFillMode) m_index_ref->Fill();
            }
            }
  }
  DbStatus status = RootTreeContainer::transAct(action);
  if (action == Transaction::TRANSACT_FLUSH) {
            if (m_tree->GetName()[0] != '#') {
            if (m_index_ref != nullptr && m_tree->GetEntryNumberWithIndex(nextRecordId()) == -1) {
            m_tree->BuildIndex("index_ref");
            }
            }
  }
  return status;
}

DbStatus RootTreeIndexContainer::loadObject(DataCallBack* call, Token::OID_t& oid, DbAccessMode
mode) {
  if ((oid.second >> 32) > 0) {
            long long int evt_id = m_tree->GetEntryNumberWithIndex(oid.second);
            if (evt_id == -1) {
            m_tree->BuildIndex("index_ref");
            evt_id = m_tree->GetEntryNumberWithIndex(oid.second);
            }
            if (evt_id >= 0) {
            oid.second = evt_id;
            }
  }
  return RootTreeContainer::loadObject(call, oid, mode);
}
```

APPENDIX B: Other framework code changes to Athena codebase related to added the added functionality

Note: Superfluous line breaks, which exist in the code, have been removed in several places.

Database/APR/RootStorageSvc/src/RootOODb.cpp

```
@@ -15,12 +15,14 @@
#include "RootDomain.h"
#include "RootKeyContainer.h"
#include "RootTreeContainer.h"
#include "RootTreeIndexContainer.h"
#include "StorageSvc/DbInstanceCount.h"

// declare the types provided by this Storage plugin
DECLARE_COMPONENT_WITH_ID(pool::RootOODb, "ROOT_All")
DECLARE_COMPONENT_WITH_ID(pool::RootOOKey, "ROOT_Key")
DECLARE_COMPONENT_WITH_ID(pool::RootOOTree, "ROOT_Tree")
DECLARE_COMPONENT_WITH_ID(pool::RootOOTreeIndex, "ROOT_TreeIndex")
using namespace pool;

@@ -51,13 +53,16 @@ IDbDatabase* RootOODb::createDatabase() {
/// Create Root Container object
IDbContainer* RootOODb::createContainer(const DbType& typ) {
  if ( typ.match(ROOTKEY_StorageType) ) {
    return new RootKeyContainer();
  }
  else if ( typ.match(ROOTTREE_StorageType) ) {
    return new RootTreeContainer();
  }
  else if ( typ.match(ROOTTREEINDEX_StorageType) ) {
    return new RootTreeIndexContainer();
  }
  else if ( typ.match(ROOT_StorageType) ) {
    return new RootTreeContainer();
  }
```

Database/APR/StorageSvc/src/DbContainerImp.cpp

```
@@ -67,7 +67,7 @@ DbStatus DbContainerImp::close() {
/// In place allocation of raw memory for the transient object
void* DbContainerImp::allocate(unsigned long siz, DbContainer& cntH, ShapeH shape)  {
  DbObjectHandle<DbObject> objH(cntH.type());
  Token::OID_t objLink(cntH.token()->oid().first, nextRecordId());
  DbHeap::allocate(siz, &cntH, &objLink, &objH);
@@ -85,7 +85,7 @@ DbStatus DbContainerImp::allocate(DbContainer& cntH, const void* object,
ShapeH shape, Token::OID_t& oid) {
if ( object )  {
  oid.first  = cntH.token()->oid().first;
  oid.second = nextRecordId();
```

```
  if ( m_stack.size() < m_size+1 )  {
    m_stack.resize(m_size+1024);
}
```

Database/AthenaPOOL/AthenaPoolCnvSvc/src/AthenaPoolCnvSvc.cxx

```
@@ -1100,6 +1100,9 @@ StatusCode AthenaPoolCnvSvc::decodeOutputSpec(std::string& fileSpec,
pool::DbType& outputTech) const {
  } else if (fileSpec.find("ROOTTREE:") == 0) {
    outputTech = pool::ROOTTREE_StorageType;
    fileSpec.erase(0, 9);
  } else if (fileSpec.find("ROOTTREEINDEX:") == 0) {
    outputTech = pool::ROOTTREEINDEX_StorageType;
    fileSpec.erase(0, 14);
  }
```

REFERENCES

[1] ATLAS Collaboration. JINST 3 S08003 (2008).

[2] R. Brun, F. Rademakers, Nucl. Instrum. Methods A 389 (1997) 81. http://dx.doi.org/10.1016/S0168-9002(97)00048-X.

[3] ATLAS Collaboration, G. Aad et al., Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC, Phys. Lett. B 716 (2012) 1–29, arXiv:1207.7214 [hep-ex].