Using Functional Languages and Declarative Programming to Analyze Large Datasets

Gordon Watts

University of Washington

LINQToROOT

The Problem: I'm a Professor

Producing Credible Plots

Then

Monte Carlo ROOT TTree

Analysis Code

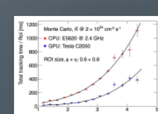
Now

Data ROOT TTree

Calculate Corrections

Analysis Code

Plots



Corrections might include p_T spectra, pile-up, η, etc.

A lot of code scattered in many files in different programming languages for some simple plot exercises!

I don't have the time!

Your post-doc was right, and don't say a thing…

Convince your grad student that you still have it…

How is a professor to survive?

Give up?
Have my students and post-docs do it all?

Or…

Write a new framework

Tune the framework to make plots

Remove as much boiler plate possible

Tune the framework to make plots

Remove as much boiler plate possible

```
using System.Linq;
using LINQToTreeHelpers;

namespace Simple
{
    class Program
    {
        static void Main(string[] args)
        {
            var ds = DataSetFinder.FindROOTFilesForDS("ggH12020");
            var q = HVData.QueryableCollectionTree.CreateQueriable(ds);

            var p = q
                .SelectMany(evt => evt.Jets)
                .Plot("jetpt", "Jet pT; pT [GeV]", 100, 0.0, 100.0, j => j.pt);

            var f = ROOTNET.NTFile.Open("junk.root", "RECREATE");
            p.SetDirectory(f);
            f.Write();
            f.Close();
        }
    }
}
```

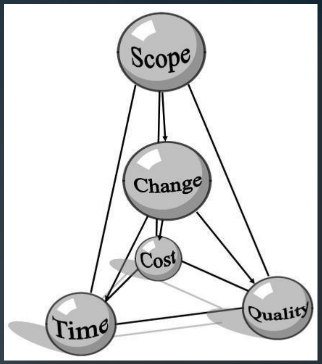
Runs over 50,000 events on a PROOF server back at UW

Setup

Plot jet p_T

Save the plot

Scope Of A Possible Solution



- Handle multiple passes
- Keep code that fills correction histograms near code that calculates the scale factors from those histograms
- PROOF!

Corrections, manipulating the plots (scaling, dividing, etc.)

- Support iterative development
  - We have moved back to the batch model of the pre-PAW* days

Mass running of 1000's of plots with lots of changes, means I make lots of mistakes or forget what I'm doing…

- Keep boiler plate code to a minimum, but be efficient
  - TSelector, proxies, etc.
  - But run in C++ for best speed

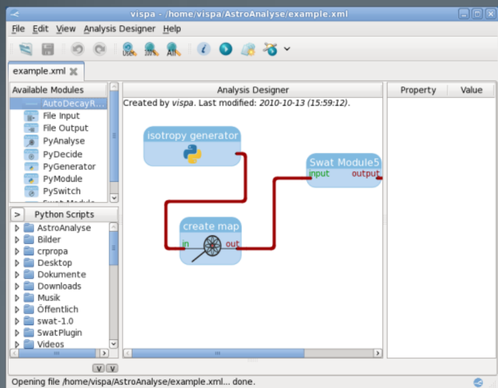
Too much code obscures the often "simple" science I'm trying to do

- Cuts, not algorithms

Not trying to invent the next b-tagging algorithm… at least, not yet…

PAW*=ROOT for really old people

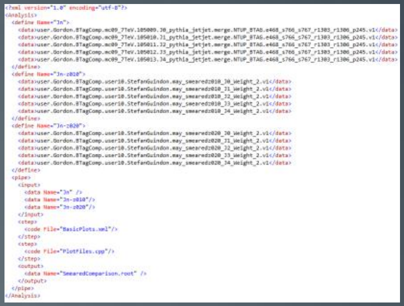
Don’t reinvent the wheel!



<u>Visual Programming</u>

- Text is what I learned in the 1970’s – why am I still using it?
- Control flow obvious to user
- Didn’t know about VISPA or others, so tried to roll my own
- Kept being forced back into actual text code.

“Right level of abstraction…”



<u>Workflow</u>

- Tried a Visual Workflow tool (ScientificWorkflow from MSR). Failed for similar reasons to my visual programming attempts, and also not really built for HEP data flows
- Tried a roll-my-own based on text
- Lots of inferred data flow, which worked well.
- But had to have separate files for each stage, and different languages too (C++, my XML language, python, etc.).
- Framework based on make-like utility

It Will Have To Be Text

Can't beat the information density and expressiveness of code!

Post histogram filling manipulations requires full power of a programming language

- Or an endless set of histogram manipulation, combination, and fitting primitives have to be written from scratch!

Expressions for filtering on or plotting require full power of programming language

- Otherwise will need to re-invent the wheel!
- Only way to run fast in ROOT is run in C++ - which has a "decent" amount of power

It Will Have To Be Code

ROOT Leads The Way: TTree::Draw

Problems:

1. All that boilerplate code needs to be abstracted away
2. Putting plot manipulation close to generating the plots

ROOT has the kernel of the solution:

```
CollectionTree->Draw("rpc_prd_phi", "rpc_prd_doublr>1")
```

- Implied loop!
- Filter and expressions for cutting built in
- Uses C++ and is fairly efficient (not quite at the metal)

- Composition is difficult at best (string manipulation!)
- Have to write a caching infrastructure
- If you have 10 plots and you are I/O bound it is not efficient
- etc.

C#: Language Integrated Query (LINQ)

Pulled from research in functional languages & put into an imperative language

```
var dsname = "JetStream";
var files = DataSetFinder.FindROOTFilesForDS(dsname);

var data = ROOTLINQ.QueryableCollectionTree.Create(files);

var p = from evt in data
        from j in evt.Jets
        where Math.Abs(j.Eta) < 2.0
        select j;
p.Plot("jetpT", "Jet pT", 100, 0.0, 100.0, j => j.Pt / 1000.0);
```

Get access to a TChain

All jets with $|\eta| < 2.0$

Plot p_T in GeV

Syntactic sugar… The compiler translates it to this (LINQ):

```
var p1 = data
    .SelectMany(evt => evt.Jets)
    .Where(j => Math.Abs(j.Eta) < 2.0)
    .Plot("jetpT", "Jet pT", 100, 0.0, 100.0, j => j.Pt / 1000.0);
```

Goal: Run against a TTree in C++ either locally or on PROOF!

C# to C++

```
var p1 = data
    .SelectMany(evt => evt.Jets)
    .Where(j => Math.Abs(j.Eta) < 2.0)
    .Plot("jetpT", "Jet pT", 100, 0.0, 100.0, j => j.Pt / 1000.0);
```

This is a C# lambda and must be translated into C++

Possible Ways To Do This:

- Modify the compilation process

  Requires detecting what code matters and where, storing it separately, and finding it at run-time, putting it back together in a TSelector and invoking ACLIC.

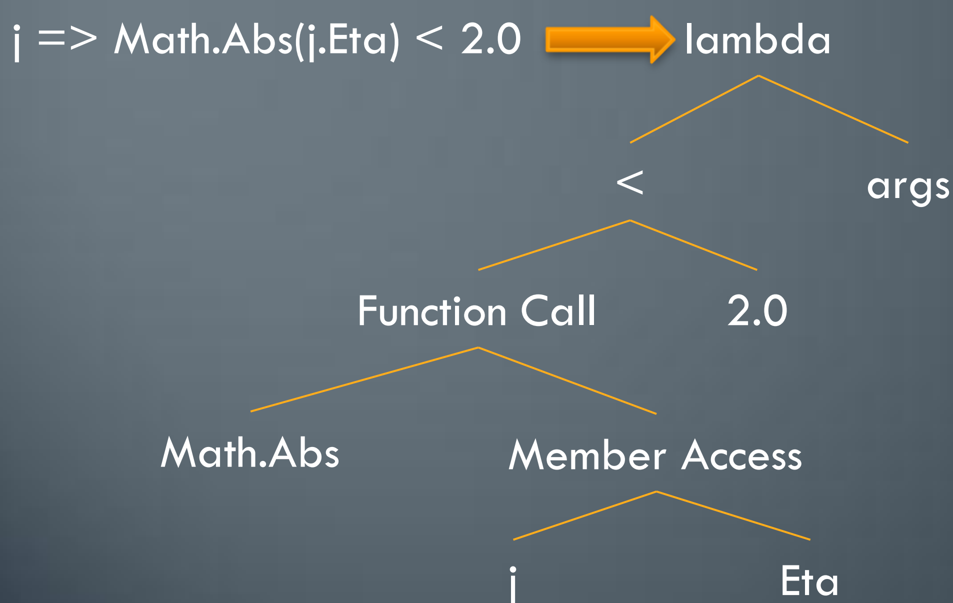
- Code as Data ⟵ C# 3.0 has decent support for this

  Requires language support (ala LISP), translating the data structures that represent the lambda function into C++, and putting that together in a TSelector and invoking ACLIC.

C# to C++

```
var p1 = data
    .SelectMany(evt => evt.Jets)
    .Where(j => Math.Abs(j.Eta) < 2.0)
    .Plot("jetpT", "Jet pT", 100, 0.0, 100.0, j => j.Pt / 1000.0);
```

Jet Where (**Expression<Func<Jet, bool>>** expr)



- Data structure is easily iterated over
- Support for the full expressions in this data structure
- No support for multi-line statements (C# language limitation)

C# to C++

```
var p1 = data
    .SelectMany(evt => evt.Jets)
    .Where(j => Math.Abs(j.Eta) < 2.0)
    .Plot("jetpT", "Jet pT", 100, 0.0, 100.0, j => j.Pt / 1000.0);
```

Plot predicate

- Creates a TH1F, sets up to fill it with jet p_T in units of GeV.
- Triggers C++ generation, ACLIC compilation, and TTree::Process to fill the histogram
- Returns the histogram, which can now be manipulated by the code

C# to C++

```
var p1 = data
    .SelectMany(evt => evt.Jets)
    .Where(j => Math.Abs(j.Eta) < 2.0)
    .Plot("jetpT", "Jet pT", 100, 0.0, 100.0, j => j.Pt / 1000.0);
```

1. The variable *data* derives from a Queriable<T> class. T is the type of object collection – CollectionTree.

2. Plot's signature means that it is called with an expression that contains the whole query:

```
public static ROOTNET.NTH1F Plot<TSource>
    (
    this IQueryable<TSource> source,
    string plotName, string plotTitle,
    int nbins, double lowBin, double highBin,
    Expression<Func<TSource, double>> xValue,
    Expression<Func<TSource, double>> weight = null)
```

An expression tree that represents the data, SelectMany, and Where calls

3. Plot calls a well known routine responsible for turning the expression tree into a result

C# to C++

Analysis Code ⟷ C# Libraries ⟷ re-linq ⟷ LINQToROOT Translator

LINQToROOT Translator ⟷ TTree::Process

Translating from the Compiler generated expression trees to something ready for C++ is non-trivial

The re-linq project is an open source project that provides much of the plumbing and takes care of many 'obvious' simplifications.

LINQToROOT is built on top of re-linq and is much simpler as a result.

http://relinq.codeplex.com/

Composability comes for free

Trivial to make a common selection and use it multiple times

Or to build the selection dynamically

```
var goodJets = from evt in qdata
               from jet in evt.Jets
               where (Math.Abs(jet.Eta) < 2.0)
               select jet;

if (doPtCut)
    goodJets = goodJets.Where(j => j.pt > ptcut);

goodJets.FuturePlot("jetpt", "Jet Pt", 100, 0.0, 100.0, j => j.pt / 1000.0).Save(f);
goodJets.FuturePlot("jetpt", "Jet Pt", 100, 0.0, 100.0, j => j.Eta).Save(f);
```

Or even functions for the cases where you need them…

```
Expression<Func<HVData.CollectionTreeJets, bool>> goodJet = j => j.pt > 10;

var goodJets = from evt in qdata
               from jet in evt.Jets
               where (Math.Abs(jet.Eta) < 2.0 && goodJet.Invoke(jet))
               select jet;
```

Composability is a big win

By far the easiest system I've used to build up and manipulate plots

Built cut-flow table analyzer:

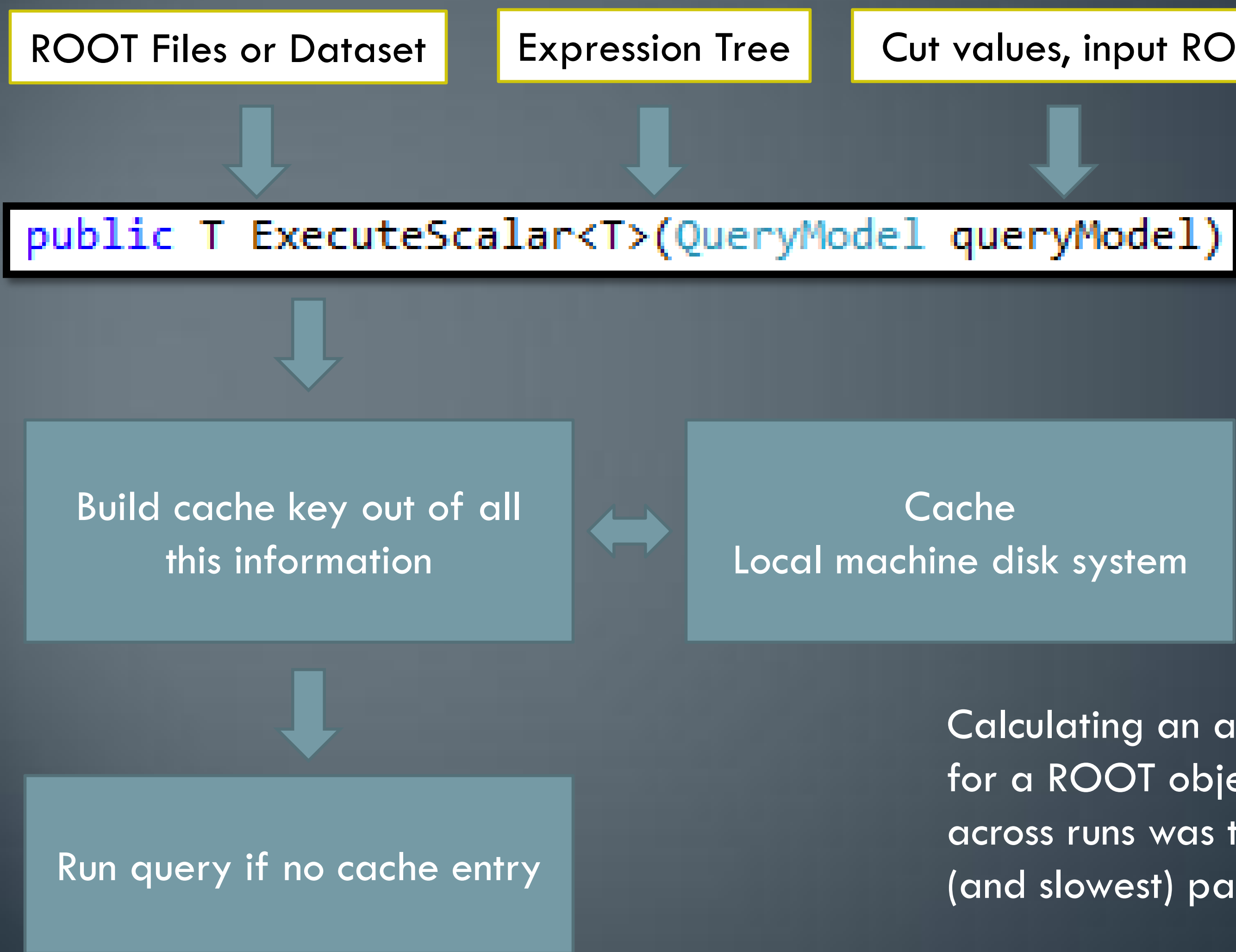
- Give it a list of cuts and plots to make
- It generates the plots after each set of cuts, and then plots them together for comparison
- Could even deal with event level cuts, and jet level plots

I quickly had over 1000 plots (not all useful!)

No other system or framework I've written or used has made it this easy.

I believe this is a direct consequence of the functional nature of LINQ.

Caching is almost free



Calculating an accurate hash key for a ROOT object that is constant across runs was the most difficult (and slowest) part.

TTree re-writing

ATLAS TTree's make minimal use of object-oriented features

LINQ is built to run against *structured* data
It can do unstructured data, but it isn't nearly as pleasant.

XML based translation system:

```
vector<float> jetAntiKt5_px;
vector<float> jetAntiKt5_py;
vector<float> jetAntiKt5_pz;
vector<float> jetAntiKt5_pT;
```

```
class Jet {
  float px, py, pz, pT;
};
Vector<Jet> Jets;
```

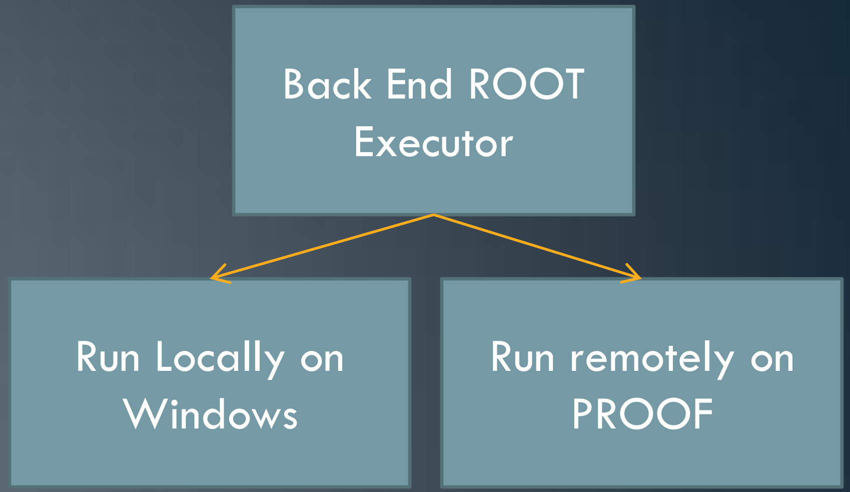
Translated into a C++ TSelector against the TTree's native format

You write your LINQ query against this object model

- Scanning program which will guess a TTree's structure and generate XML files that you can edit.
- Indirection is also supported

PROOF

- *Experimental* support as of version 0.5 of LINQToROOT
- If you try to run over a PROOF dataset, then the PROOF backend is used. Otherwise the local backed is used.
- Works…
- ROO T communication is not robust, and generates a huge amount of output making it very hard to figure out if anything went wrong
- Constant hangs on the server which could be due to how I am invoking it.



Only way to get high speed running on a large dataset!

Future Work

- Robustness
- Be able to close lid of laptop, walk to next meeting, and not loose a “long” running query.

I get very excited…

But it isn't without its problems…

Running Simultaneous Queries

```
particles
    .FuturePlot("mc_pT", "pT of all MC Particles; pT [GeV]; # Particles", 20, 0.0, 200.0, p => p.TLZ.Pt() / 1000.0)
    .Save(f);
particles
    .FuturePlot("pdgid", "PDG ID of the particles", 2000, -1000.0, 1000.0, p => p.PDGID)
    .Save(f);
particles
    .Where(p => p.vtxTermOK)
    .FuturePlot("decayLength", "The decay length of the particles; d [mm]", 100, 1.0, 0.0, p => p.vtxTerm.Mag())
    .Save(f);
```

The *FuturePlot* call queues a query, referencing a *Value* will run all queued queries on the *data* variable.

- Common task-based threading coding pattern
- You don't know if .Value will trigger a run accidentally: code is less obvious.
- Manipulation of the results is a little less natural…

```
var pnorm1 = p1.Apply(h => h.SetNormFactor(2.0));
var pnorm2 = p2.Apply(h => h.SetNormFactor(2.0));
var pratio = DivideBy(p1, p2);
```

"Monad Hell"

Some functional languages might offer a way out of this (F#)…

Sometimes you need C++

- ROOT is not functional, only functional expressions supported in LINQ
- You want to call a C++ routine that is your own code
- Some algorithms are easier to write in C++!

1. Direct mapping to existing C++ functions by a text file

```
include: cmath

Math Abs(System.Double) => std::abs(double)
Math Abs(System.Single) => std::abs(float)
Math Abs(System.Int32) => std::abs(int)
```

2. Include a C++ fragment

```
[CPPCode(IncludeFiles = new string[] { "TLorentzVector.h" },
    Code = new string[]{
        "TLorentzVector tlzUnique;",
        "tlzUnique.SetPtEtaPhiE(pt, eta, phi, E);",
        "CreateTLZ = &tlzUnique;"
    })]
public static ROOTNET.NTLorentzVector CreateTLZ(double pt, double eta, double phi, double E)
```

You can now call CreateTLZ in your LINQ query and the C++ code will be inserted

Quality of generated C++ code

- Multiple plots are intelligently combined
  - Plotting the jet p_T and η of a special subset of jets
- Results from queries within an event are cached
  - Plotting the jet p_T and η of the highest p_T jet in each event

But there are plenty of situations where you get this sort of thing:

```
var prs = particles.SelectMany(p => p)
    .Where(p => p.vtxTerm.Px() > 6.0)
    .Where(p => p.vtxTerm.Pt() > 5.0)
    .Count();
```

```
TVector3 aNTVector3_7((((*this).McEventCollection_p4_GEN_EVENT.m_genVertices.m_x)
    ((*this).McEventCollection_p4_GEN_EVENT.m_genVertices.m_x)[aInt32_6],
    ((*this).McEventCollection_p4_GEN_EVENT.m_genVertices.m_x)[aInt32_6]);
TVector3 *aNTVector3_8 = &aNTVector3_7;
if (((*aNTVector3_8).Px())>6.0)
{
    TVector3 aNTVector3_9((((*this).McEventCollection_p4_GEN_EVENT.m_genVertices.
        ((*this).McEventCollection_p4_GEN_EVENT.m_genVertices.m_x)[aInt32_6],
        ((*this).McEventCollection_p4_GEN_EVENT.m_genVertices.m_x)[aInt32_6]);
    TVector3 *aNTVector3_10 = &aNTVector3_9;
    if (((*aNTVector3_10).Pt())>5.0)
    {
        aInt32_11=aInt32_11+1;
    }
}
```

Created Twice

Using MakeProxy as the base interface. Accessing things repeatedly can be expensive

Recent analyses were the first time the code became CPU bound…

If you hate Microsoft and C#

But you like this approach…

The programming language you choose needs to be able to:

- Treat code as data (i.e. an Expression Tree)
  - This was free in C#
- Be well integrated with ROOT
  - I wrote a project that wraps ROOT in .NET (ROOT.NET, see poster)

Could you use raw C++?

- Use gcc's XML output, parse for the relevant expressions
- Want to make sure that you are independent of local version of ROOT and PROOF version of ROOT.

Could you use python?

- Not sure how you get around lack of Expression Tree's? Compile py code?

Conclusions

- Used for a number of Hidden Valley QCD background studies
  - This summer plan to use it for simple full analysis.
- Excellent for making plots, applying cuts, associating objects (jets, tracks, etc.)
  - Code is straight forward and easy to read
  - Probably not good for track finding and fitting type algorithms??
- What succeeded
  - Composability! Wow!
  - Boilerplate code dramatically reduced…
  - Time from new-project to first project is less than 5 minutes, if you know what you are doing.

Conclusions

- What needs work
  - "Monad-Hell"
  - Support for including common run-time C++ packages and libraries written by the experiment (i.e. good run list).
  - Improve time to up-and-running.
  - Fill in missing corners of LINQ translation, e.g. *joins*.
  - Output C++ optimization
- Future
  - Optimization of generated C++ code
  - Stabilizing PROOF support
  - Mostly driven by what I need in my analysis…
  - Could you do this in a language like python?
- Open Source: http://linqtoroot.codeplex.com/, and also on nuget.