RDataFrame

Zdeněk Hubáček

Bílý Potok, WEJČF 2020

Komentář k PBSPro

- ./mcprogram –Nevents 10000 –Seed 1

- for i in {1..10}
  do
  ./mcprogram –Nevents 1000 –Seed $i
  done

- Smyčka v čemkoliv – bash/python/C++ … jen potřebujete najít místo, kde to rozdělit

RDataFrame

Zdeněk Hubáček

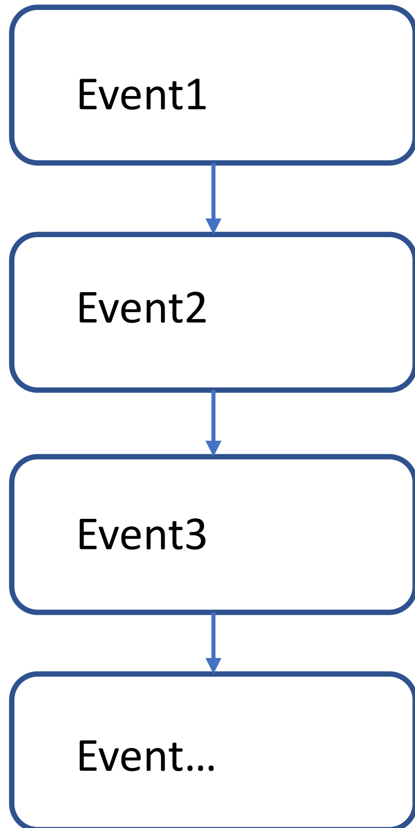
Bílý Potok, WEJČF 2020

Jak zpracovat velké množství dat (TTree)

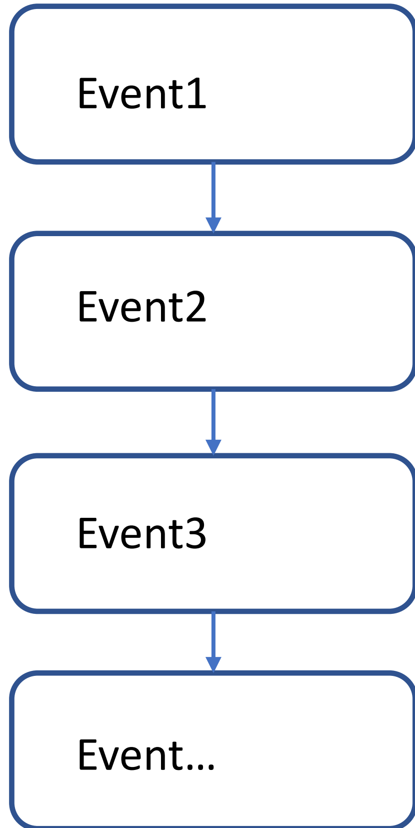
- Z prezentace o PBSPro – jednotlivé eventy zpracováváme nezávisle, sériově za sebou, je výhodné rozdělit úlohu na menší díly a běžet více jobů vedle sebe nezávisle a výsledky sečíst
- Dnes mají počítače více CPU/výpočetních jader, "frčí" paralelní počítání
- RDataFrame je novinka v poslední verzi ROOTu - paralelní zpracování dat z ntuplů – ROOT 6.18/00 a novější
  - Na sunrise:

```
export LCGENV_PATH=/cvmfs/sft.cern.ch/lcg/releases
export PATH=/cvmfs/sft.cern.ch/lcg/releases/lcgenv/latest:${PATH}
eval "`lcgenv x86_64-centos7-gcc8-opt all`"

# try LCG_96
eval "`lcgenv -p LCG_96 x86_64-centos7-gcc8-opt ROOT`"
```

Koncept



Koncept



Batch

Job1

Job2

Job…

Event1

Event…

Event…

Event2

Event…

Event…

Event3

Event…

Event…

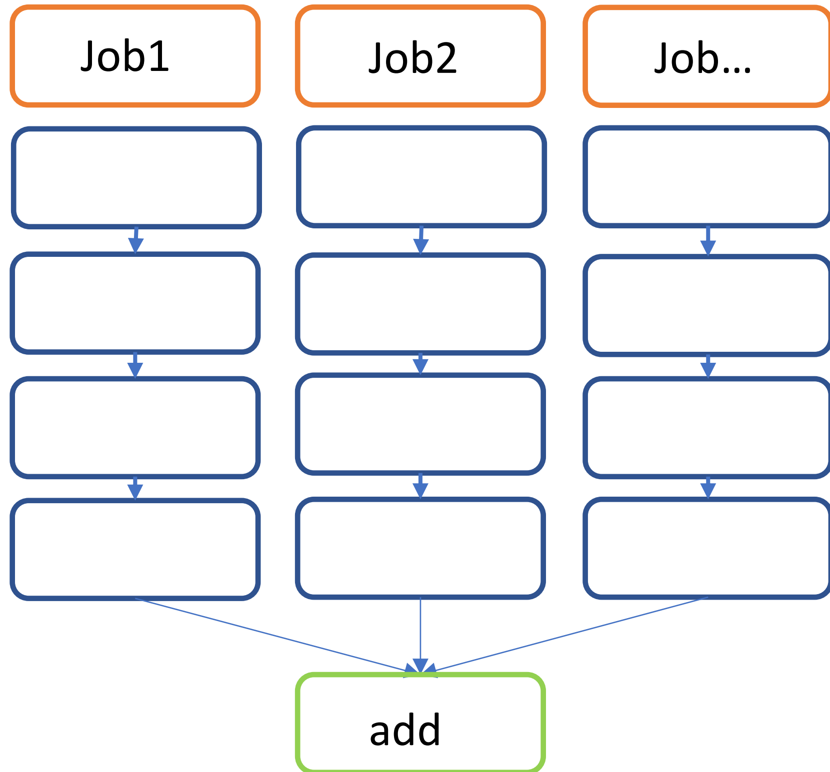
Event…

Event…

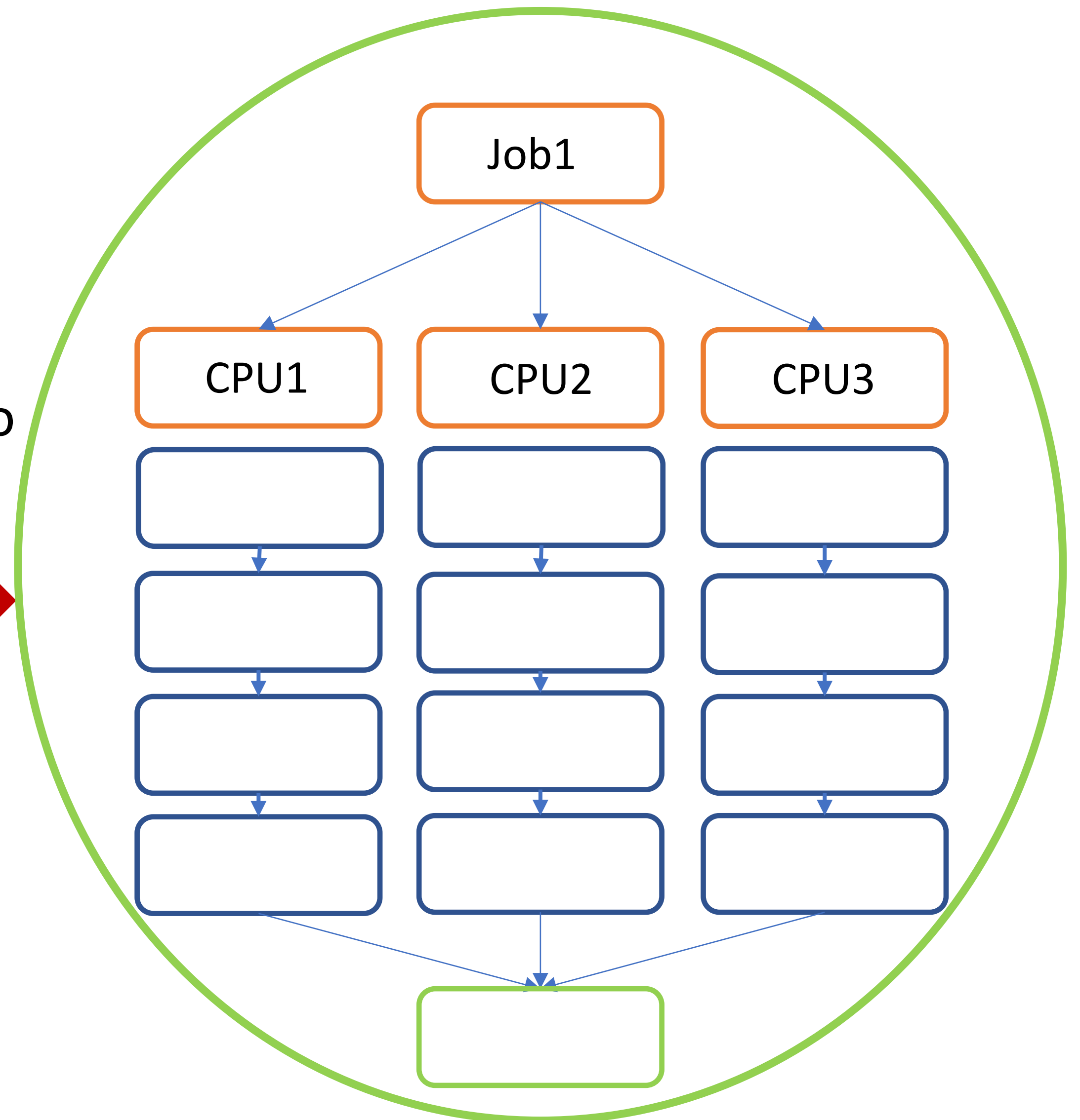
Event…

‘hadd‘

Koncept



Mám víc procesorů, nešlo by to udělat jednoduše/automaticky



Nešlo by to sečíst “samo“?

Parallel computing (nebo aspoň trochu)

Paralelní počítání

- Většina našeho počítání/zpracování dat není paralelní počítání (aspoň prozatím)
- https://en.wikipedia.org/wiki/Parallel_computing

- Pro nás:
  - Matematické knihovny, SIMD (https://en.wikipedia.org/wiki/SIMD)
  - Postarat se, aby se více vláken (threads, CPU) naráz nepokusilo zároveň zapsat do stejného histogramu (kusu paměti)

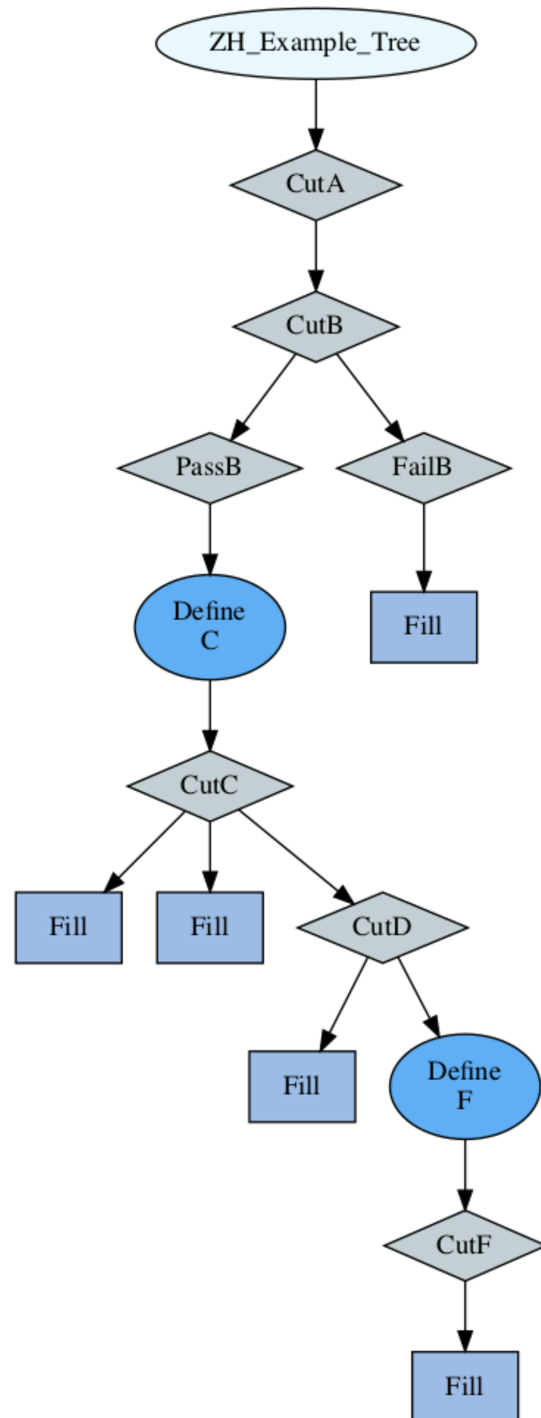
Zpátky k analýze ntuplů

- Způsobů jak udělat smyčku přes všechny eventy v ntuplech je celá řada
- Ignorujme pracovní frameworky a ntuply specifické pro každý experiment – něco na úrovni “flat“ ntuplů/TTree

```
void create_tree()
{
  TFile *f = TFile::Open("test_tree.root","RECREATE");
  TTree *t = new TTree("ZH_Example_Tree","ZH_Example_Tree");

  float A,B;
  double D,E;
  int X,Y,Z;

  t->Branch("A",&A);
  t->Branch("B",&B);
  t->Branch("D",&D);
  t->Branch("E",&E);
   t->Branch("X",&X);
  t->Branch("Y",&Y);
  t->Branch("Z",&Z);

  for (int event = 0; event < 1000; event++)
    {
      A=gRandom->Rndm()*10;
      B=gRandom->Gaus(1,2);
      D=gRandom->Uniform(10);
      E=gRandom->Poisson(3.);

      X=event;
      Y=(int) (event-50)/2;
      Z=event*2-20;
      t->Fill();
    }
  t->Write();
  f->Close();
}
```

Jednoduchý TTree, pár náhodných větví A,B,D,X,Y,Z…

Zadání

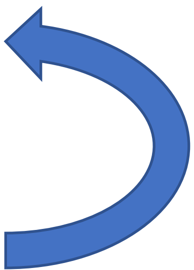


- Mějme nějaké ntuple/TTree
- Na něm chceme udělat "analýzu" – pár cutů: CutA, CutB…
- V závislosti na výsledku cutu naplnit pár histogramů
- …

TTree::MakeClass("ClassLoop")

- Vygeneruje kostru – ClassLoop.h, ClassLoop.C

```
class ClassLoop {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

// Fixed size dimensions of array or collections stored in the TTree if any.

   // Declaration of leaf types
   Float_t         A;
   Float_t         B;
   Double_t        D;
   Double_t        E;
   Int_t           X;
   Int_t           Y;
   Int_t           Z;

   // List of branches
   TBranch        *b_A;   //!
   TBranch        *b_B;   //!
   TBranch        *b_D;   //!
   TBranch        *b_E;   //!
   TBranch        *b_X;   //!
   TBranch        *b_Y;   //!
   TBranch        *b_Z;   //!

   ClassLoop(TTree *tree=0);
   virtual ~ClassLoop();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```



Kostra sama propojí větve s proměnnými

```
void ClassLoop::Loop()
{
   if (fChain == 0) return;
   TH1D *h1 = new TH1D(.....)
   ....

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      if (cutA)
       {
         if (cutB)
           {
             float C = DefineC();
             if (cutC)
               {
                 h1->Fill(X);
                 h2->Fill(Y);
                 if (cutD)
                   {
                     h3->Fill();
                     float F = DefineF();
                     if (cutF)
                       {
                         h4->Fill();
                       }
                   }

               }

           ...
           }
         else
           {
           ...
```

TTree::MakeClass(“ClassLoop“)

- Vygeneruje kostru – ClassLoop.h, ClassLoop.C

```
class ClassLoop {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

// Fixed size dimensions of array or collections stored in the TTree if any.

   // Declaration of leaf types
   Float_t         A;
   Float_t         B;
   Double_t        D;
   Double_t        E;
   Int_t           X;
   Int_t           Y;
   Int_t           Z;

   // List of branches
   TBranch        *b_A;   //!
   TBranch        *b_B;   //!
   TBranch        *b_D;   //!
   TBranch        *b_E;   //!
   TBranch        *b_X;   //!
   TBranch        *b_Y;   //!
   TBranch        *b_Z;   //!

   ClassLoop(TTree *tree=0);
   virtual ~ClassLoop();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

```
void ClassLoop::Loop()
{
   if (fChain == 0) return;
   TH1D *h1 = new TH1D(.....)
   ....

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      if (cutA)
        {
          if (cutB)
            {
              float C = DefineC();
              if (cutC)
                {
                  h1->Fill(X);
                  h2->Fill(Y);
                  if (cutD)
                    {
                      h3->Fill();
                      float F = DefineF();
                      if (cutF)
                        {
                          h4->Fill();
                        }
                    }

                }

              ...
            }
          else
            {
              ...
```

TTree::MakeClass(“ClassLoop“)

- Vygeneruje kostru – ClassLoop.h, ClassLoop.C

```
class ClassLoop {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

// Fixed size dimensions of array or collections stored in the TTree if any.

   // Declaration of leaf types
   Float_t         A;
   Float_t         B;
   Double_t        D;
   Double_t        E;
   Int_t           X;
   Int_t           Y;
   Int_t           Z;

   // List of branches
   TBranch        *b_A;   //!
   TBranch        *b_B;   //!
   TBranch        *b_D;   //!
   TBranch        *b_E;   //!
   TBranch        *b_X;   //!
   TBranch        *b_Y;   //!
   TBranch        *b_Z;   //!

   ClassLoop(TTree *tree=0);
   virtual ~ClassLoop();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

```
void ClassLoop::Loop()
{
   if (fChain == 0) return;
   TH1D *h1 = new TH1D(.....)
   ....
```

Definice histogramů, atd…

```
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      if (cutA)
       {
         if (cutB)
           {
             float C = DefineC();
             if (cutC)
               {
                 h1->Fill(X);
                 h2->Fill(Y);
                 if (cutD)
                   {
                     h3->Fill();
                     float F = DefineF();
                     if (cutF)
                       {
                         h4->Fill();
                       }
                   }


               }

             ...
           }
         else
           {
             ...
```

TTree::MakeClass(“ClassLoop“)

- Vygeneruje kostru – ClassLoop.h, ClassLoop.C

```
class ClassLoop {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

// Fixed size dimensions of array or collections stored in the TTree if any.

   // Declaration of leaf types
   Float_t         A;
   Float_t         B;
   Double_t        D;
   Double_t        E;
   Int_t           X;
   Int_t           Y;
   Int_t           Z;

   // List of branches
   TBranch        *b_A;   //!
   TBranch        *b_B;   //!
   TBranch        *b_D;   //!
   TBranch        *b_E;   //!
   TBranch        *b_X;   //!
   TBranch        *b_Y;   //!
   TBranch        *b_Z;   //!

   ClassLoop(TTree *tree=0);
   virtual ~ClassLoop();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

```
void ClassLoop::Loop()
{
   if (fChain == 0) return;
   TH1D *h1 = new TH1D(.....)
   ....

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
```

Hlavní smyčka

```
        if (cutB)
         {
           float C = DefineC();
           if (cutC)
            {
              h1->Fill(X);
              h2->Fill(Y);
              if (cutD)
               {
                 h3->Fill();
                 float F = DefineF();
                 if (cutF)
                  {
                    h4->Fill();
                  }
               }

            }

           ...
         }
        else
         {
           ...
```

TTree::MakeClass(“ClassLoop“)

- Vygeneruje kostru – ClassLoop.h, ClassLoop.C

```
class ClassLoop {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

// Fixed size dimensions of array or collections stored in the TTree if any.

   // Declaration of leaf types
   Float_t         A;
   Float_t         B;
   Double_t        D;
   Double_t        E;
   Int_t           X;
   Int_t           Y;
   Int_t           Z;

   // List of branches
   TBranch        *b_A;   //!
   TBranch        *b_B;   //!
   TBranch        *b_D;   //!
   TBranch        *b_E;   //!
   TBranch        *b_X;   //!
   TBranch        *b_Y;   //!
   TBranch        *b_Z;   //!

   ClassLoop(TTree *tree=0);
   virtual ~ClassLoop();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

```
void ClassLoop::Loop()
{
   if (fChain == 0) return;
   TH1D *h1 = new TH1D(.....)
   ....

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      if (cutA)
       {
         if (cutB)
           {
             float C = DefineC();
             if (cutC)
               {
                 h1->Fill(X);
                 h2->Fill(Y);
                 if (cutD)
                   {
                     h3->Fill();
                     float F = DefineF();
                     if (cutF)
                       {
                         h4->Fill();
                       }
                   }

               }

           ...
      }
```

“Analýza“

TTree::MakeClass smyčka

root> .L ClassLoop.C

root> ClassLoop analyza //vyrobit jednu instanci třídy ClassLoop (tady jde zadat ukazatel na jiný strom nebo TChain)

root> analyza.Loop();   // Loop on all entries

TTree::MakeSelector

- t->MakeSelector("OrigSelector","**LEGACY**")

Jméno selectoru

Starší verze bez TTreeReader – definice všech TBranch a jejich nastavení...

- Vygeneruje opět OrigSelector.h, OrigSelector.C

MakeSelector

```
class OrigSelector : public TSelector {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain

// Fixed size dimensions of array or collections stored in the TTree if any.

   // Declaration of leaf types
   Float_t         A;
   Float_t         B;
   Double_t        D;
   Double_t        E;
   Int_t           X;
   Int_t           Y;
   Int_t           Z;

   // List of branches
   TBranch        *b_A;   //!
   TBranch        *b_B;   //!
   TBranch        *b_D;   //!
   TBranch        *b_E;   //!
   TBranch        *b_X;   //!
   TBranch        *b_Y;   //!
   TBranch        *b_Z;   //!

   OrigSelector(TTree * /*tree*/ =0) : fChain(0) { }
   virtual ~OrigSelector() { }
   virtual Int_t   Version() const { return 2; }
   virtual void    Begin(TTree *tree);
   virtual void    SlaveBegin(TTree *tree);
   virtual void    Init(TTree *tree);
   virtual Bool_t  Notify();
   virtual Bool_t  Process(Long64_t entry);
   virtual Int_t   GetEntry(Long64_t entry, Int_t getall = 0) { return fChain ? …
fChain->GetTree()->GetEntry(entry, getall) : 0; }
   virtual void    SetOption(const char *option) { fOption = option; }
   virtual void    SetObject(TObject *obj) { fObject = obj; }
   virtual void    SetInputList(TList *input) { fInput = input; }
   virtual TList  *GetOutputList() const { return fOutput; }
   virtual void    SlaveTerminate();
   virtual void    Terminate();
```

- Init() – možnost změnit nastavení větví, vypnutí větví, které nejsou potřeba
- Notify() – možnost změnit/nastavit něco při otevření nového souboru
- SlaveBegin() inicializace histogramů
- Bool_t OrigSelector::Process(Long64_t entry) je hlavní smyčka

MakeSelector

- Umí běžet paralelně na více jádrech (PROOF)

- Root> TChain *chain = new TChain(…);
- Root> OrigSelector *analyza = new OrigSelector();
- Root> chain->Process(analyza);

TTreeReader

```
class Selector : public TSelector {
public :
   TTreeReader     fReader;  //!the tree reader
   TTree          *fChain = 0;   //!pointer to the analyzed TTree or TChain

   // Readers to access the data (delete the ones you do not need).
   TTreeReaderValue<Float_t> A = {fReader, "A"};
   TTreeReaderValue<Float_t> B = {fReader, "B"};
   TTreeReaderValue<Double_t> D = {fReader, "D"};
   TTreeReaderValue<Double_t> E = {fReader, "E"};
   TTreeReaderValue<Int_t> X = {fReader, "X"};
   TTreeReaderValue<Int_t> Y = {fReader, "Y"};
   TTreeReaderValue<Int_t> Z = {fReader, "Z"};


   Selector(TTree * /*tree*/ =0) { }
   virtual ~Selector() { }
   virtual Int_t   Version() const { return 2; }
   virtual void    Begin(TTree *tree);
   virtual void    SlaveBegin(TTree *tree);
   virtual void    Init(TTree *tree);
   virtual Bool_t  Notify();
   virtual Bool_t  Process(Long64_t entry);
   virtual Int_t   GetEntry(Long64_t entry, Int_t getall = 0) { return fChain ? 
fChain->GetTree()->GetEntry(entry, getall) : 0; }
   virtual void    SetOption(const char *option) { fOption = option; }
   virtual void    SetObject(TObject *obj) { fObject = obj; }
   virtual void    SetInputList(TList *input) { fInput = input; }
   virtual TList  *GetOutputList() const { return fOutput; }
   virtual void    SlaveTerminate();
   virtual void    Terminate();

   ClassDef(Selector,0);
```

Bez použití LEGACY flagu použije TTreeReader

Vsuvka: Imperativní programování

- Předchozí příklady analýzy odpovídají něčemu, čemu se říka **imperativní programování**
- Jedno z programovacích paradigmat - neboli způsobů, jak jsou v programovacím jazyku formulována řešení problémů.
- Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit

Deklarativní programování

- **Deklarativní programování** je založeno na myšlence programování aplikací pomocí definic co se má udělat a ne jak se to má udělat (opak imperativního)
- Zjednodušeně to lze popsat tak, že imperativní programy obsahují algoritmy, kterými se dosáhne chtěný cíl, zatímco deklarativní jazyky specifikují cíl a algoritmizace je ponechána programu (interpretu) daného jazyka.
- C++ umožnuje jak imperativní tak deklarativní způsob

Příklad

```
int values[4] = { 8, 23, 2, 4 };
int sum = 0;
for (int i = 0; i < 4; ++i)
        sum += values[i];
int temp = values[0];
for (int i = 0; i < 3; ++i)
        values[i] = values[i + 1];
values[3] = temp;
```

- Imperativní zápis
- Co ten kód dělá?

Příklad

```
// Declare array
int values[4] = { 8, 23, 2, 4 };

// Calculate sum
int sum = 0;
for (int i = 0; i < 4; ++i)
        sum += values[i];

// Rotate array items one slot left.
int temp = values[0];
for (int i = 0; i < 3; ++i)
        values[i] = values[i + 1];
values[3] = temp;
```

- Přidání komentářů pomůže k vysvětlení, co kód dělá

Příklad

```
int values[4] = { 8, 23, 2, 4 };
int sum = SumArray(values);
RotateArrayIndices(values, -1);
```

- Deklarativní příklad, který dělá to samé
- Jednodušší k pochopení
- Komentáře („deklarativní“) nejsou potřeba

RDataFrame

ROOT's **RDataFrame** offers a high level interface for analyses of data stored in **`TTree`**s, CSV's and other data formats.

In addition, multi-threading and other low-level optimisations allow users to exploit all the resources available on their machines completely transparently.
Skip to the class reference or keep reading for the user guide.

In a nutshell:

```
ROOT::EnableImplicitMT(); // Tell ROOT you want to go parallel
ROOT::RDataFrame d("myTree", "file_*.root"); // Interface to TTree and TChain
auto myHisto = d.Histo1D("Branch_A"); // This happens in parallel!
myHisto->Draw();
```

Calculations are expressed in terms of a type-safe *functional chain of actions and transformations*, **`RDataFrame`** takes care of their execution. The implementation automatically puts in place several low level optimisations such as multi-thread parallelisation and caching.

https://root.cern/doc/master/classROOT_1_1RDataFrame.html

RDataFrame

Deklarativní způsob

- RDataFrame (datový rámec?) je hlavní objekt, na kterém uživatel definuje sérii transformací (analýza), Framework se pak postará o všechno ostatní (I/O, smyčka přes eventy atd..)

`RDataFrame` is built with a *modular* and *flexible* workflow in mind, summarised as follows:

1. **build a data-frame** object by specifying your data-set
2. **apply a series of transformations** to your data
   a. **filter** (e.g. apply some cuts) or
   b. **define** a new column (e.g. the result of an expensive computation on branches)
3. **apply actions** to the transformed data to produce results (e.g. fill a histogram)

The following table shows how analyses based on `TTreeReader` and `TTree::Draw` translate to `RDataFrame`. Follow the crash course to discover more idiomatic and flexible ways to express analyses with `RDataFrame`.

Jednoduché příklady

<table>
<tr><th>TTreeReader</th><th>ROOT::RDataFrame</th></tr>
<tr><td>

```
TTreeReader reader("myTree", file);
TTreeReaderValue<A_t> a(reader, "A");
TTreeReaderValue<B_t> b(reader, "B");
TTreeReaderValue<C_t> c(reader, "C");
while(reader.Next()) {
   if(IsGoodEvent(*a, *b, *c))
      DoStuff(*a, *b, *c);
}
```

</td><td>

```
ROOT::RDataFrame d("myTree", file, {"A", "B", "C"});
d.Filter(IsGoodEvent).Foreach(DoStuff);
```

</td></tr>
<tr><th>TTree::Draw</th><th>ROOT::RDataFrame</th></tr>
<tr><td>

```
auto t = file->Get<TTree>("myTree");
t->Draw("x", "y > 2");
```

</td><td>

```
ROOT::RDataFrame d("myTree", file);
auto h = d.Filter("y > 2").Histo1D("x");
```

</td></tr>
</table>

Začínáme

- Všechno schované v ROOT:: prostoru jmen (nezapomenout na **using namespace ROOT;** // RDataFrame's namespace)
- Jak vytvořit hlavní RDataFrame – konstruktor nabízí řadu možností
  - **RDataFrame dataframe("treeName", "file.root");**
  - ….
  - **std::vector<std::string> filelist = …**
  - **RDataFrame dataframe2(("treeName", filelist);**

pohodlné

Co se dá s RDataFrame udělat

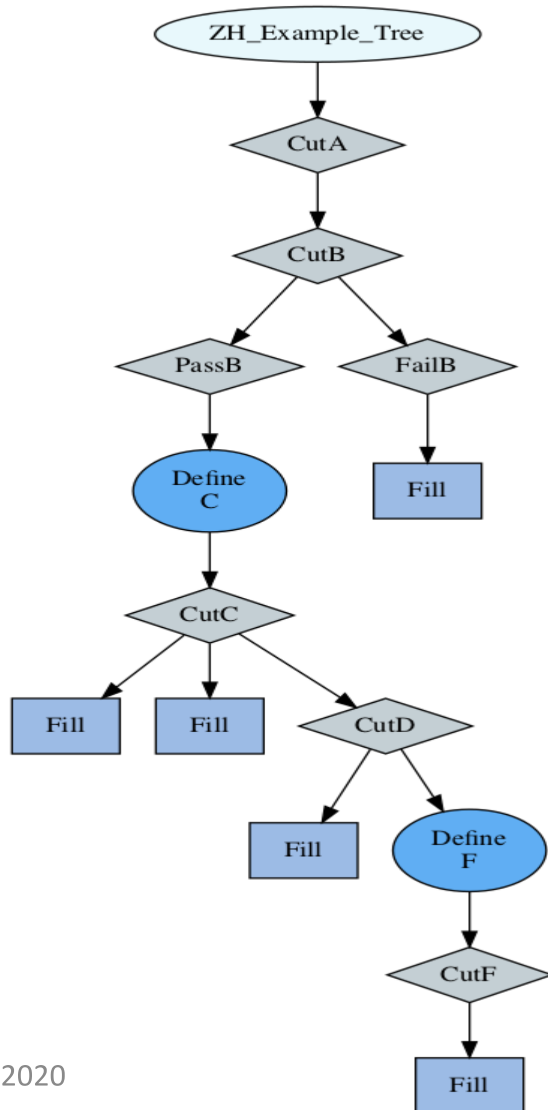
- **Transformace** (*transformations*) – způsob, jakým manipulovat s daty
  - **Define** – přidání nového sloupce (proměnné do ntuplů)
  - **Filter** – filtrování (cut)
  - …
  - Vrátí novou upravenou DataFrame
- **Akce** (*actions*) – způsob jak z dat dostat výsledky
  - **Histo1D** – naplnit histogram
  - **Count** – spočítat počet událostí
  - … (spousta dalších předdefinovaných, možnost přidat i vlastní)
  - Vrátí smart pointer na daný výsledek
  - Působí jen na události, které prošly posledním filtrem
  - Lazy/instant actions

Jak naplnit histogram

```
RDataFrame dataframe(”ZH_Example_Tree", ”test_tree.root");
auto h = dataframe.Histo1D("A");
h->Draw();
```

- Histo1D je akce (*action*), vrací smart pointer (RResultPtr), ve kterém je zabalený TH1D*
- POZN: RDataFrame nepoužívá TH1D* přímo, pokud chcete vytvořit vlastní histogram (například chcete změnit binning) tak je potřeba použít ROOT::RDF::TH1DModel*

Jak udělat první filter (cut)



- **RDataFrame dataframe(”ZH_Example_Tree", ”test_tree.root");**

  auto c = dataframe.Filter("A>5").Count();
  std::cout << *c << std::endl;

- POZN: DataFrame::Filter vytvoří novou DF (transformace), na ní chceme akci Count (tj. celou analýzu by šlo v principu napsat na jeden řádek (I když nepřehledně))

Filter

- Filtr (cut) je cokoliv, co vrátí true/false
- True projde dál (false je zahozeno)

- Nejjednodušší forma je výraz (*expression*) jako u TTree->Draw, možnost využít i spočítané nové proměnné, ale nepřehledné a trochu zpomalující – lepší využít vlastní funkci, functor, lambda funkci…

Vsuvka – lambda funkce v c++11 a novější

- “Nepojmenovaná funkce“
- Co je v C++ **[](){}();** ?

Vsuvka – lambda funkce v c++11 a novější

- “Nepojmenovaná funkce“
- Co je v C++ `[](){}();` ?
- ”an in-place definition and immediate call of an anonymous void function which takes no parameters and does nothing. It is also completely legal.”

Zkuste si přímo v ROOTu ;-)

Lambda výrazy v C++

C++11 přinesla integraci lambda výrazů přímo do standardu C++. Základní myšlenkou lambda výrazu v C++ je, že dokáže velmi elegantně nahradit jednoduché funkce. Obvykle se jedná o operace jako je porovnání prvků, provedení určité matematické operace, atp.

Obecný lambda výraz má následující syntax:

```
[](input_paramter_declaration)->returned_type {body_of_the_lambda_expression}(parameters)
```

Pojďme si jej ukázat na příkladu.

Vytvořeme lambda výraz, který provede součet dvou čísel.

```
[](int a,int b)->int{return a+b;}(2,4);
```

Jak je z příkladu patrné, hranaté závorky budou vždy beze změny. Následuje kulatá závorka, ve které jsou deklarovány vstupní hodnoty pro labda výraz (v našem případě 2 celočíselné hodnoty). Následuje šipka a deklarace typu návratové hodnoty (celé číslo). Ve složených závorkách je samotný lambda výraz (v zásadě si jej představme jako tělo funkce/metody). Zcela na konci v kulatých závorkách jsou pak vždy reálné vstupní hodnoty. Tento lambda výraz má tedy hodnotu 6.

Nyní jej zasaďme do příkazu.

Pokud bychom chtěli výsledek lambda výrazu uložit do proměnné, pouze lambda výraz přiřadíme:

```
int result=[](int a,int b)->int{return a+b;}(2,4);
std::cout << result << std::endl;
```

https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=32147

Tj. náš filtr

- **RDataFrame dataframe("ZH_Example_Tree", "test_tree.root");**

  auto CutA = [](double x) { return x > 5.; };
  auto passedA = dataframe.Filter(CutA, {"A"}).Count();
  std::cout << *passedA << std::endl;

- {"A"} specifikuje na které větvi chci filtr použít
- **auto passedAFrame = dataframe.Filter(CutA,{"A"},"CutA"));**
  - Možnost využít passedAFrame dále v analyze
  - Automatická počítadla prošlých událostí (poslední parameter je jméno počítadla)

Definování nové proměnné

- **auto latestDF = passedAFrame.Define("C","A+B");**
- Definice nové proměnné C, možnost využití v další DF (latestDF)
- Opět možnost využít (lambda nebo vlastní funkci k definici nové proměnné

- **auto sqrtSum = [](double x, double y) { return sqrt(x*x + y*y); };**
  auto zMean = d.Define("z", sqrtSum, {"x","y"}).Mean("z");
- Vše ostatní v analýze už je to samé
  - .Define().Filter().Define().Filter()….

Going Parallel

- Stačí zadat na začátku **ROOT::EnableImplicitMT();**
- Dá se **ROOT::EnableImplicitMT(ncores);**
- Všechno uvnitř je na paralelní běh připraveno, ale o bezpečnost uživatelských akcí a transformací je potřeba se postarat!

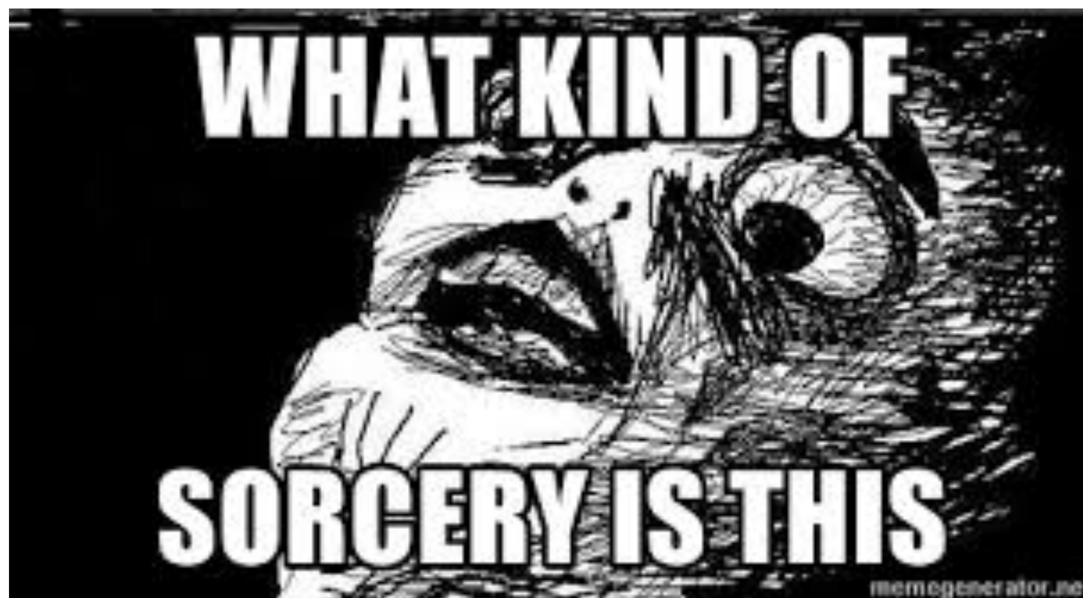
- std::mutex a podobně (to už je mimo tuto úvodní prezentaci)

Třída RVec – něco jako std::vector<> s extra vlastnostmi

- Příklad – máme v ntuplech větve s miony:
  - **std::vector<short> mu_charge {1, 1, -1, -1, -1, 1, 1, -1};**
  - **std::vector<float> mu_pt {56, 45, 32, 24, 12, 8, 7, 6.2};**
  - **std::vector<float> mu_eta {3.1, -.2, -1.1, 1, 4.1, 1.6, 2.4, -.5};**
- A chceme vybrat "dobré" miony
  - **std::vector<float> goodMuons_pt;**
  - **const auto size = mu_charge.size();**
  - **for (size_t i=0; i < size; ++i) {**
  - **if (mu_pt[i] > 10 && abs(mu_eta[i]) <= 2. && mu_charge[i] == -1) {**
  - **goodMuons_pt.emplace_back(mu_pt[i]);**
  - **}**
  - **}**

Tohle by šlo zjednodušit na 1 řádek

- Když se místo std::vector<float> použije ROOT::RVec<float>, tak jde
- **auto goodMuons_pt = mu_pt[(mu_pt > 10.f && abs(mu_eta) <= 2.f && mu_charge == -1)]**



RDataFrame standardně využívá RVec místo std::vector

- **auto GoodJetsPt = [](ROOT::RVec<float> &pt, ROOT::RVec<float> &y, float &ptcut, float &rapcut)**
- **{**
- **return pt[(pt > ptcut && abs(y)<rapcut)];**
- **};**

- Tj. tohle jde pěkně využít v Define, Filter,…

Finální analýza

```
ROOT::RDataFrame HlavniDataFrame("ZH_Example_Tree","test_tree.root");
auto TotalEvents = HlavniDataFrame.Count();
std::vector<std::string> colNamesStart = HlavniDataFrame.GetColumnNames();


auto passedA  = std::make_unique<RNode>(HlavniDataFrame.Filter(filterA,{"A"},"CutA"));
auto cutBf    = std::make_unique<RNode>(passedA->Filter("return true;","CutB"));
auto passedB  = std::make_unique<RNode>(cutBf->Filter(filterB,{"B"},"PassB"));
auto failedB  = std::make_unique<RNode>(cutBf->Filter(failB,{"B"},"FailB"));
auto latestDF = std::make_unique<RNode>(passedB->Define("C","A+B"));
auto passedC  = std::make_unique<RNode>(latestDF->Filter("C>5","CutC"));
auto passedD  = std::make_unique<RNode>(passedC->Filter("D<3","CutD"));
auto defineF  = std::make_unique<RNode>(passedD->Define("F","C*A-B"));
auto passedF  = std::make_unique<RNode>(defineF->Filter("F>1","CutF"));

std::vector<std::string> colNamesEnd = passedF->GetColumnNames();
passedF->Count();

auto h1 = passedC->Histo1D("X");
auto h2 = passedC->Histo1D("Y");
auto h3 = failedB->Histo1D("Z");
auto h4 = passedD->Histo1D("Y");
auto h5 = passedF->Histo1D("X");
```

Finální analýza

```
ROOT::RDataFrame HlavniDataFrame("ZH_Example_Tree","test_tree.root");
auto TotalEvents = HlavniDataFrame.Count();
std::vector<std::string> colNamesStart = HlavniDataFrame.GetColumnNames();

auto passedA  = std::make_unique<RNode>(HlavniDataFrame.Filter(filterA,{"A"},"CutA"));
auto cutBf    = std::make_unique<RNode>(passedA->Filter("return true;","CutB"));
auto passedB  = std::make_unique<RNode>(cutBf->Filter(filterB,{"B"},"PassB"));
auto failedB  = std::make_unique<RNode>(cutBf->Filter(failB,{"B"},"FailB"));
auto latestDF = std::make_unique<RNode>(passedB->Define("C","A+B"));
auto passedC  = std::make_unique<RNode>(latestDF->Filter("C>5","CutC"));
auto passedD  = std::make_unique<RNode>(passedC->Filter("D<3","CutD"));
auto defineF  = std::make_unique<RNode>(passedD->Define("F","C*A-B"));
auto passedF  = std::make_unique<RNode>(defineF->Filter("F>1","CutF"));

std::vector<std::s
passedF->Count();

auto h1 = passedC->Histo1D("X");
auto h2 = passedC->Histo1D("Y");
auto h3 = failedB->Histo1D("Z");
auto h4 = passedD->Histo1D("Y");
auto h5 = passedF->Histo1D("X");
```

Není explicitně nutné, zde jen pro přehlednost, lepší řetězit..

Finální analýza

```
ROOT::RDataFrame HlavniDataFrame("ZH_Example_Tree","test_tree.root");
auto TotalEvents = HlavniDataFrame.Count();
std::vector<std::string> colNamesStart = HlavniDataFrame.GetColumnNames();


auto passedA  = std::make_unique<RNode>(HlavniDataFrame.Filter(filterA,{"A"},"CutA"));
auto cutBf    = std::make_unique<RNode>(passedA->Filter("return true;","CutB"));
auto passedB  = std::make_unique<RNode>(cutBf->Filter(filterB,{"B"},"PassB"));
auto failedB  = std::make_unique<RNode>(cutBf->Filter(failB,{"B"},"FailB"));
auto latestDF = std::make_unique<RNode>(passedB->Define("C","A+B"));
auto passedC  = std::make_unique<RNode>(latestDF->Filter("C>5","CutC"));
auto passedD  = std::make_unique<RNode>(passedC->Filter("D<3","CutD"));
auto defineF  = std::make_unique<RNode>(passedD->Define("F","C*A-B"));
auto passedF  = std::make_unique<RNode>(defineF->Filter("F>1","CutF"));

std::vector<std::string> colNamesEnd = passedF->GetColumnNames();
passedF->Count();

auto h1 = passedC->Histo1D("X");
auto h2 = passedC->Histo1D("Y");
auto h3 = failedB->Histo1D("Z");
auto h4 = passedD->Histo1D("Y");
auto h5 = passedF->Histo1D("X");
```

Histogram je akce, možno vytvořit kdekoliv, akce je spuštěna, když je potřeba

Finální analýza

```
ROOT::RDataFrame HlavniDataFrame("ZH_Example_Tree","test_tree.root");
auto TotalEvents = HlavniDataFrame.Count();
std::vector<std::string> colNamesStart = HlavniDataFrame.GetColumnNames();


auto passedA  = std::make_unique<RNode>(HlavniDataFrame.Filter(filterA,{"A"},"CutA"));
auto cutBf    = std::make_unique<RNode>(passedA->Filter("return true;","CutB"));
auto passedB  = std::make_unique<RNode>(cutBf->Filter(filterB,{"B"},"PassB"));
auto failedB  = std::make_unique<RNode>(cutBf->Filter(failB,{"B"},"FailB"));
auto latestDF = std::make_unique<RNode>(passedB->Define("C","A+B"));
auto passedC  = std::make_unique<RNode>(latestDF->Filter("C>5","CutC"));
auto passedD  = std::make_unique<RNode>(passedC->Filter("D<3","CutD"));
auto defineF  = std::make_unique<RNode>(passedD->Define("F","C*A-B"));
auto passedF  = std::make_unique<RNode>(defineF->Filter("F>1","CutF"));

std::vector<std::string> colNamesEnd = passedF->GetColumnNames();
passedF->Count();

auto h1 = passedC->Histo1D("X");
auto h2 = passedC->Histo1D("Y");
auto h3 = failedB->Histo1D("Z");
auto h4 = passedD->Histo1D("Y");
auto h5 = passedF->Histo1D("X");
```

Pro kontrolu možno získat seznam všech proměnných

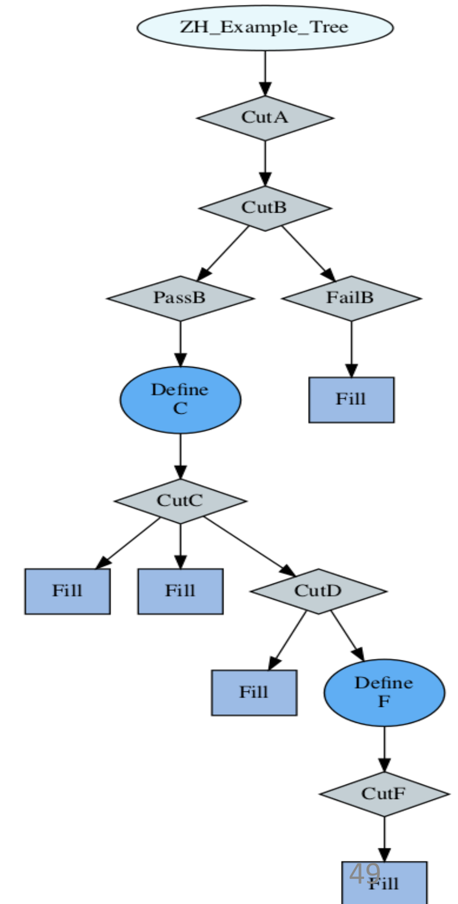
Počítadla

```
auto allCutsReport = HlavniDataFrame.Report();
allCutsReport->Print();
```

```
root [1] rdataframe_test()
Vetve na zacatku:
A B D E X Y Z
===
Vetve na konci:
C F A B D E X Y Z
===
CutA      : pass=1000     all=1000     -- eff=100.00 % cumulative eff=100.00 %
CutB      : pass=1000     all=1000     -- eff=100.00 % cumulative eff=100.00 %
PassB     : pass=496      all=1000     -- eff=49.60 % cumulative eff=49.60 %
FailB     : pass=504      all=1000     -- eff=50.40 % cumulative eff=50.40 %
CutC      : pass=212      all=496      -- eff=42.74 % cumulative eff=21.20 %
CutD      : pass=61       all=212      -- eff=28.77 % cumulative eff=6.10 %
CutF      : pass=61       all=61       -- eff=100.00 % cumulative eff=6.10 %
```

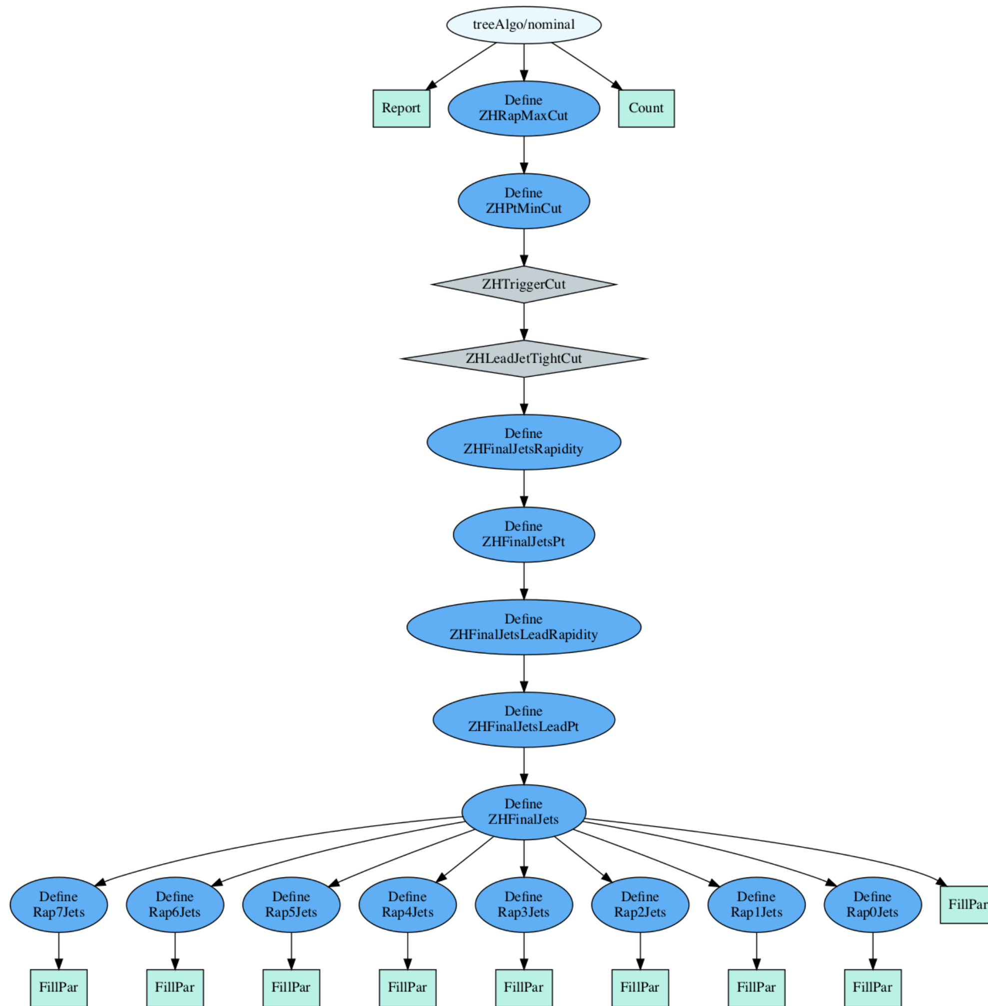
Graf

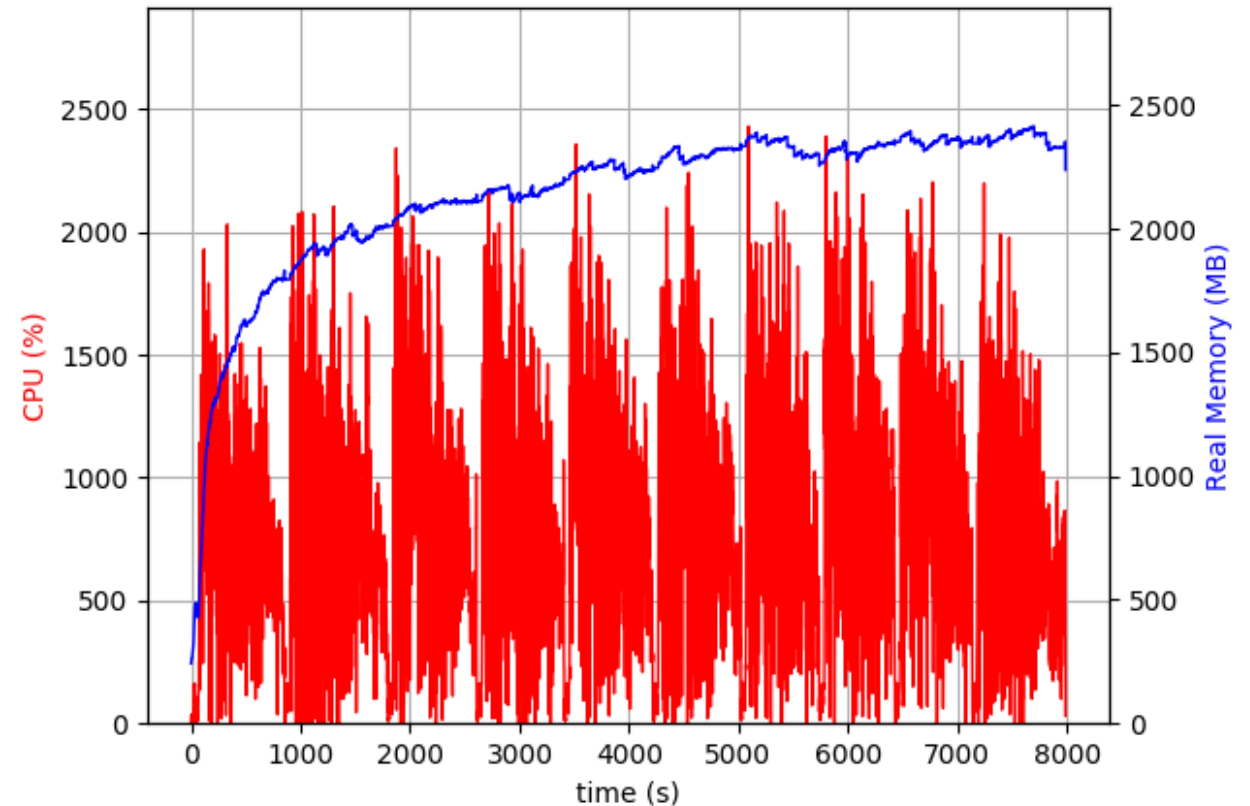
- **ROOT::RDF::SaveGraph(HlavniDataFrame,"test_tree_graph.dot");**
- https://en.wikipedia.org/wiki/DOT_(graph_description_language)
- graphviz balíček (linux, macOS,…)
- **dot -Tpng test_tree_graph.dot -o test_tree_graph.png**



Performance example

psrecord pythonovský balíček, který umožňuje monitorovat běžící proces





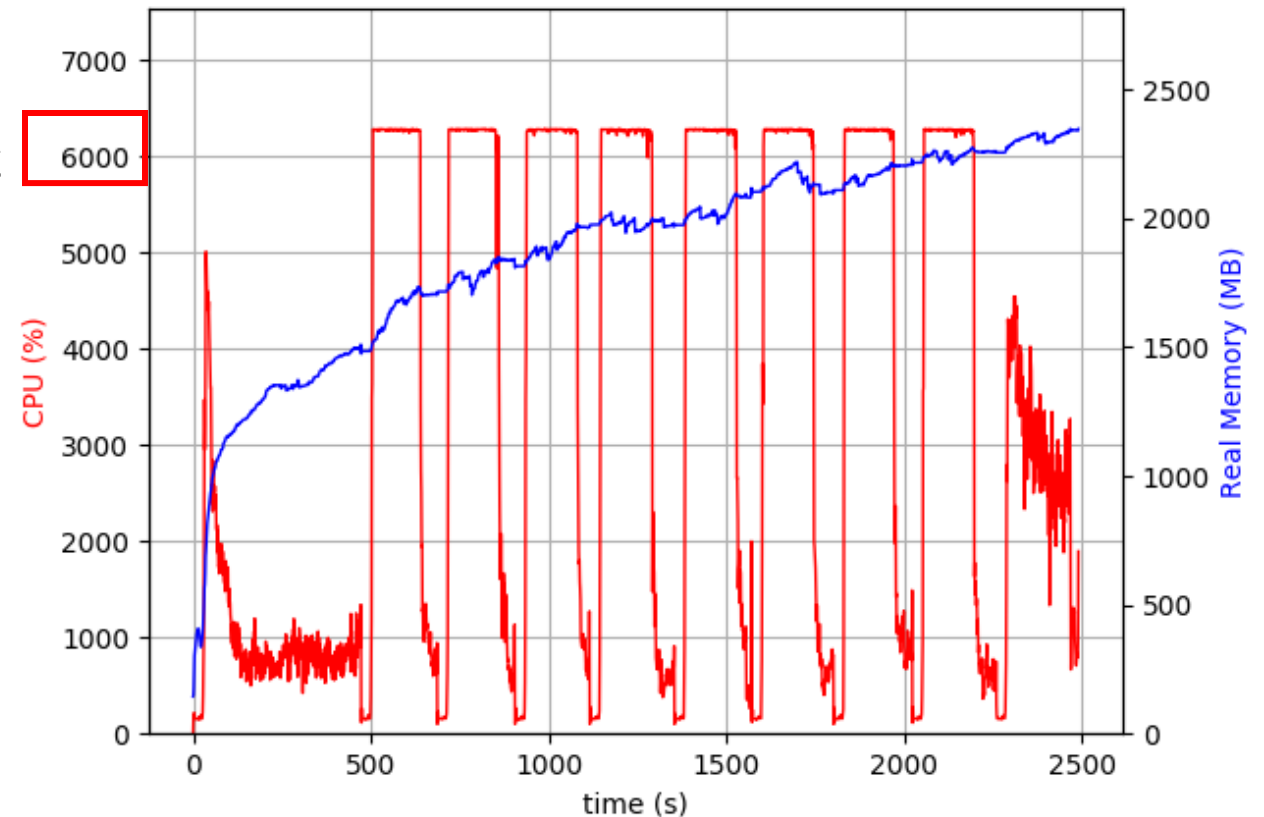
370M eventů zpracovaných na jednom počítači za 3h

Performance example

- Jeden počítač = 64 jader, 128GB paměti
- Zpomaleno rychlostí sítě (počítač v Praze, data v CERNu)

Performance example

- Jeden počítač = 64 jader, 128GB paměti
- Zpomaleno rychlostí sítě (počítač v Praze, data v CERNu)
- Zkopírování dat do Prahy = 370M eventů/hodinu



Shrnutí

- RDataFrame je nový způsob zpracování ntuplů v ROOT 6.18
- Využívající nejnovější vlastnosti C++11/14/17, paralelizace,…
- Spousta možností stojících za vyzkoušení