Native Language Integrated Queries with CppLINQ in C++

Vassil Vassilev

PH-SFT, CERN

Type-safe SQL in C++ = CppLINQ


```
#include ”cpplinq/linq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
long sum_primes() {
  // boost::counting_iterator
  auto xs = int_range(0, 100);
  using namespace cpplinq;
  return from(xs).where(is_prime).sum();
}
```

Language Integrated Queries

- Native support of SQL-like syntax
- Operate on collections using relational algebra operations.
- Iteration fundamental to the design on frameworks collections and a cornerstone for STL.
- C++11 provides even more iteration simplifiers such as range-based iteration.
- C++ & STL are LINQ-friendly

CppLINQ Dialects/Implementations


```
#include ”cpplinq/linq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
long sum_primes() {
  // same as boost::counting_iterator
  auto xs = int_range(0, 100);
  using namespace cpplinq;
  return from(xs).where(is_prime).sum();
}
```

MS RX, *cpplinq*

```
#include ”cpplinq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
long sum_primes() {
  // same as boost::counting_iterator
  auto xs = int_range(0, 100);
  using namespace cpplinq;
  return from(xs) >> where(is_prime) >> sum();
}
```

boolinq

Usage

#include LINQ header file

```
#include ”cpplinq/linq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
long sum_primes() {
  auto xs = int_range(0, 100); // == boost::counting_iterator
  using namespace cpplinq;
  return from(xs).where(is_prime).sum();
}
```

Enables LINQ

Sums all prime numbers in [0, 100)

Usage


```
#include ”cpplinq/linq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
using namespace cpplinq;
long sum_squared_primes() {
  auto xs = int_range(0, 100); // boost::counting_iterator
  return from(xs)
           .where(is_prime).select([](int x){return x*x}).sum();
}

std::vector<int> take_first_n_primes (int howMany) {
  auto xs = int_range(0, -1); // 0 to MAX(int)
  return from(xs).where(is_prime).take(howMany).to_vector();
}
```

C++11 lambda

Event Data Queries in ROOT


```
...
enum ParticleType {
   Photon = 0,
   Electron,
   Pion
};
struct Particle {
  ROOT::Math::XYZVector  fPosition; // vertex position (Gaus)
  ROOT::Math::PtEtaPhiMVector fVector; // particle vector
  int  fCharge; // particle charge
  ParticleType fType; // particle type
};
struct EventData {
  std::vector<Particle> fParticles; // Poisson distributed mean 40
};
...
```

Event Data Queries in ROOT


```
void PtPos(TFile* myFile) {
  // Calculate the pt (momentum) for all positive charged particles.

  TTreeReader tree("tree", myFile);
  TTreeReaderArray<Particle> particles(tree, "fParticles");

  using namespace cpplinq;
  auto PosPts = from(tree).select([&particles](Long64_t entry) {
     return from(particles).
       where([](const Particle& p) { return p.fCharge > 0; }).
       select([](const Particle& p) { return p.fVector.Pt(); }).sum();
     });
  auto h = new TH1F("ptSum", "Sum p_T of events; p_T [GeV]", 200, 0, 500);
  for(double pt : PosPts)
    h->Fill(pt);
}
```

```
root [4] TimeZad2()
Real time 0:00:16, CP time 16.970
root [5] TimeZad2NoLINQ()
Real time 0:00:16, CP time 16.410
root [6]
```

Event Data Queries in ROOT


```
void Do(TTree* tree) {
  // Select x and y pos of events which have at least 4 particles,
  // which have at least N GeV.
  TTreeReaderArray<Particle> particles(tree, "fParticles");

  using namespace cpplinq;
  typedef decltype(from(particles)) t_particleItr;
  auto goodParticles =
      from(tree)
      .select([&](Long64_t entry) { return from(particles); })
      .where([&](const t_particleItr &plist) {
            return plist.where([&](const Particle &p){
              return p.fVector.E()>N;
            }).count() >= 4;});
  // iterate and fill a histogram
}
```

```
root [3] Time(Zad1)
Real time 0:00:18, CP time 18.360
root [4] Time(Zad1NoLINQ)
Real time 0:00:17, CP time 17.980
```

Event Data Queries in ROOT


```
void PtPos(TTreeReader& tree) {
  // Select pt of the leading electron of all events which have at least 2 pions.
  TTreeReaderArray<Particle> particles(tree, "fParticles");

  using namespace cpplinq;
  auto result =
      from(from(tree).where([&particles](Long64_t entry) {
        bool hasAtLeast2 = from(particles)
          .where([](const Particle& p) { return p.fType == Pion; }).count() > 2;
        auto electrons = from(from(particles)
          .where([](const Particle& p) {return p.fType == Electron;}))
          .select([](const Particle& p) {return p.fVector.Pt();});

        // Missing implementation of default_if_empty().
        if (hasAtLeast2 && !electrons.empty()) return electrons.max();
        return 0;
      })).where([](int charge){return charge != 0;});
   auto h = new TH1F("pt", "pt of leading e; p_T [GeV
   for (double pt : result) h->Fill(pt);
}
```

```
Real time 0:00:16, CP time 16.920
root [8] TimeZad3NoLINQ();
Real time 0:00:16, CP time 16.040
root [9]
```

CppLINQ in ROOT6's prompt, just works


```
root [0] .I ../cpplinq/Ix/CPP/src/
root [1] #include "EventData.h"
root [2] TFile *myFile = TFile::Open("~/Dropbox/ACAT14/CppLINQ/ppt/samples/eventdata_s99.root");
root [3] TTreeReader tree("tree", myFile);
root [4] #include "cpplinq/linq.hpp"
root [5] using namespace cpplinq;
root [6] from(tree).count()
(typename std::iterator_traits<iterator>::difference_type) 200
root [7] from(tree).take(20)
(linq_driver<linq_take<class cpplinq::iter_cursor<class TTreeReader::Iterator_t> > >) @0x7fc0c0aec2d0
root [8] from(tree).take(20).to_vector()
```

LINQ Operators

- groupby
- select
- select_many
- where
- aggregate
- any
- all
- cast
- contains
- count
- element_at
- empty
- first
- last
- max
- min
- sum
- take
- to_vector
- late_bind
- from
- …

Debugging your code

- Sometimes can become a frustrating experience
  One needs a good compiler diagnostics for templated instantiations.
- Complex template mechanics at the backstage
  Higher compile times, faster to write.
- Could be improved by using various techniques

Advantages: Computation Abstraction

Going from imperative to declarative paradigm:

- Allows to describe the result of the computations instead of the computations themselves.
- Opens a new world of optimizations/scheduling
- Allows communication of the ‘expected’ results to a smart batch system doing complex optimizations and scheduling

Performance

- CppLINQ's implementation is based on std::iterator and templates
- Well-written queries have similar performance to an imperative implementation

Future Improvements


```
void PtP
  // Sel                    tron of all                    e at least 2 pions.
  TTreeR                    cles(tree,

  using
   auto res
      from(from(tree).where([&particles](Long64_t entry) {
        bool hasAtLeast2 = from(particles)
          .where([](const Particle& p) { return p.fType == Pion; }).count() > 2;
        auto electrons = from(from(particles)
          .where([](const Particle& p) {return p.fType == Electron;}))
          .select([](const Part cle& p) {return p.fVector.Pt();});

        // Missing implement          of default_if_empty().
        if (hasAtLeast2                          turn electrons.max();
        return 0;
      })).where([](int c                          ;}).count();

   printf("Events with more than 2 pions: %d\n", result);
}
```

Extra iteration because of the missing implementation of default_if_empty()

Remove useless details

Find a way to hide the types in lambda expressions

Future

- Expression trees
- Late binding
- Implement cpplinq.remote for PROOF-like systems
- Parallel CppLINQ

Thank you!

Acknowledgments: Thanks to the work of Gordon Watts on C# LINQ with ROOT.

Further resources:

https://github.com/vgvassilev/RxCpp