Support for MiniAOD Transformer for ServiceX

Haoran Sun

University of Washington

Mentors: Prof. Gordon Watts, Dr.Alex Held, Dr. Oksana Shadura, Dr. Ben Galewsky, Mason Proffitt

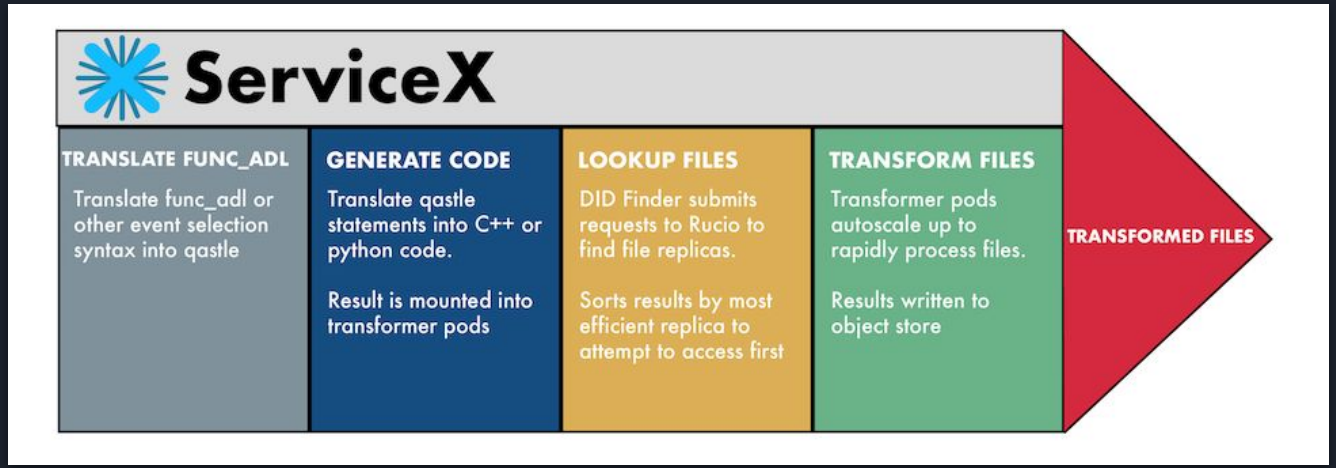
Content Overview

- What is ServiceX?
- What is Func_adl?
- Architecture of func_adl
- Result & Conclusion

ServiceX

Data extraction and delivery service for HEP event data

Location, Extraction, filtering, and transformation on the data



https://iris-hep.org/projects/servicex

Func_adl

SQL-Like Declarative Language

Extracts and filters data from files

```
from func_adl_servicex import ServiceXSourceXAOD

ds = "mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.DAOD_STDM3.e3601_s2576_s2

f_ds = ServiceXSourceXAOD(ds)
r = f_ds \
    .SelectMany('lambda e: e.Jets("AntiKt4EMTopoJets")') \
    .Select('lambda j: j.pt() / 1000.0') \
    .AsPandasDF('JetPt') \
    .value()
print(r)
```

- Jet Collection Selection
- Select pt() attribute of each jets
- Store filtered data as pandas dataframe
- Execute the query

Previously supported flat root file, ATLAS xAOD file, and CMS run1 AOD file.

This project enabled CMS run2 miniAOD file as the backend.

(Code snippet from: https://servicex.readthedocs.io/en/latest/user/requests/)

Architecture of func_adl

Func_adl & Abstract Syntax Tree(AST)

- Tree representation of query
- Structure of the query

Query

```
f_single.SelectMany('lambda e: e.Muons("slmmedMuons")')
        .Select('lambda m: m.pt()')
```

AST rep:

```
Call(func=Name(id='SelectMany', ctx=Load()),
     args=[Call(func=Name(id='EventDataset', ctx=Load()),
                args=[],
                keywords=[]),
           Lambda(args=arguments(posonlyargs=[],
                                 args=[arg(arg='e')],
                                 kwonlyargs=[],
                                 kw_defaults=[],
                                 defaults=[]),
                  body=Call(func=Name(id='Select', ctx=Load()),
                            args=[Call(func=CPPCodeValue(),
                                       args=[Constant(value='slmmedMuons')],
                                       keywords=[]),
                                  Lambda(args=arguments(posonlyargs=[],
                                                        args=[arg(arg='m')],
                                                        kwonlyargs=[],
                                                        kw_defaults=[],
                                                        defaults=[]),
                                         body=Call(func=Attribute(value=Name(id='m',
                                                                             ctx=Load()),
                                                                  attr='pt',
                                                                  ctx=Load()),
                                                   args=[],
                                                   keywords=[]))],
                            keywords=[]))]
     keywords=[])
```

Func_adl & Abstract Syntax Tree(AST)

Query

EventCollectionSpecification

```
f_single.SelectMany('lambda e: e.Muons("slmmedMuons")')
        .Select('lambda m: m.pt()')
```

```
def get_collection(self, md: EventCollectionSpecification, call_node: ast.Call):
    r'''
    Return a cpp ast for accessing the jet collection with the given arguments.
    '''
    # Get the name jet collection to look at.
    if len(call_node.args) != 1:
        raise ValueError(f"Calling {md.name} - only one argument is allowed")
    if not isinstance(call_node.args[0], ast.Str):
        raise ValueError(f"Calling {md.name} - only acceptable argument is a string")

    # Fill in the CPP block next.
    r = cpp_ast.CPPCodeValue()
    r.args = ['collection_name', ]
    r.include_files += md.include_files
    r.link_libraries += md.libraries

    self.get_running_code_CPPCodeValue(r, md)
    r.result = 'result'
```

AST rep:

```
Call(func=Name(id='SelectMany', ctx=Load()),
     args=[Call(func=Name(id='EventDataset', ctx=Load()),
                args=[],
                keywords=[]),
           Lambda(args=arguments(posonlyargs=[],
                                 args=[arg(arg='e')],
                                 kwonlyargs=[],
                                 kw_defaults=[],
                                 defaults=[]),
                  body=Call(func=Name(id='Select', ctx=Load()),
                            args=[Call(func=CPPCodeValue(),
                                       args=[Constant(value='slmmedMuons')],
                                       keywords=[]),
                                  Lambda(args=arguments(posonlyargs=[],
                                                        args=[arg(arg='m')],
                                                        kwonlyargs=[],
                                                        kw_defaults=[],
                                                        defaults=[]),
                                         body=Call(func=Attribute(value=Name(id='m',
                                                                             ctx=Load()),
                                                                  attr='pt',
                                                                  ctx=Load()),
                                                   args=[],
                                                   keywords=[]))],
                            keywords=[]))]
     keywords=[])
```

Executor()

cms_miniaod_executor(): Simplify AST, traverses AST, turn it into C++ code.

apply_ast_transformation():simplify AST

query_ast_visitor(): Object that traverses through AST
- Subclass of ast.NodeVisitor
- Extracts and accumulates c++ information

generated_code(): Stores accumulated information

```
class generated_code:
    def __init__(self):
        self._block = block()
        self._book_block = block()
        self._class_vars = []
        self._scope_stack = (self._block,)
        self._include_files = []
        self._link_libraries = []
```

Inject C++ Code

write_cpp_files()

```
def write_cpp_files(self, ast: ast.AST, output_path: Path) -> ExecutionInfo:
    r"""
    Given the AST generate the C++ files that need to run. Return them along with
    the input files.
    """
      ...
      ...
    qv = self.get_visitor_obj()
      ...
      ...

    class_decl_code = qv.class_declaration_code()
      ...
      ...
    # The replacement dict to pass to the template generator can now be filled
    info = {}
    info['class_decl'] = class_decl_code
```

C++ Template:

```
   ...
   virtual void
endLuminosityBlock(edm::LuminosityBlock const &,
edm::EventSetup const &);

   TTree *myTree;

   {% for l in class_decl %}
   {{l}}
   {% endfor %}

};

Analyzer::Analyzer(const edm::ParameterSet & iConfig)
   ...
```

Run Generated C++ File

EDAnalyzers

Part of CMS Software (CMSSW)

Modules that allow read-only access to the CMS Event

Executed in the container(cmssw_7_6_7) by runner.sh

EDAnalyzer: https://cms-opendata-workshop.github.io/workshop2021-lesson-cmssw/03-edanalyzers/index.html

Results & Conclusions

Query -> Generated Code

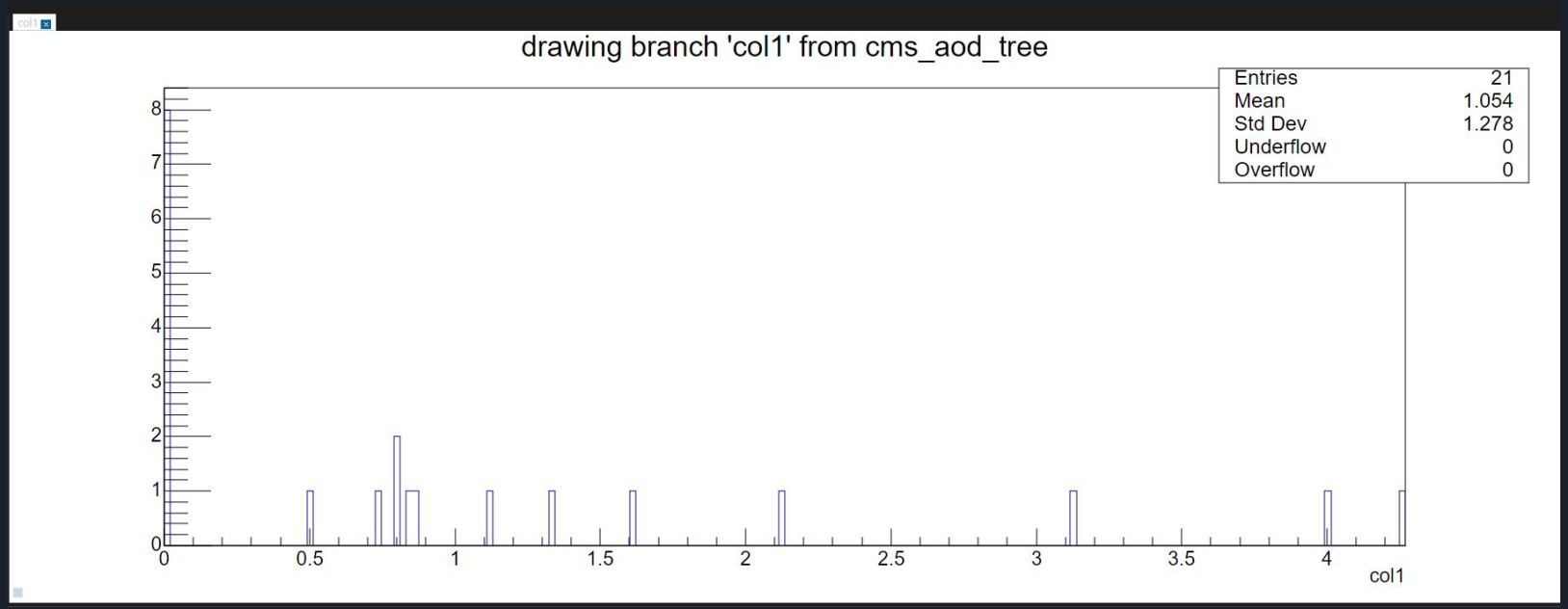
Query Code

```
f_single.SelectMany('lambda e: e.Muons("slmmedMuons")')
        .Select('lambda m: m.pt()')
```

Generated C++ Code

```
. . .
edm::EDGetTokenT<pat::MuonCollection> token0;
. . .
{
  {
    // Inside Constructor
    token0 = consumes<pat::MuonCollection>(edm::InputTag("slimmedsMuons"));
  }
}
. . .
 {
   Handle<pat::MuonCollection> muons1;
   {
     Handle<pat::MuonCollection> result;
     iEvent.getByToken(token0, result);
     muons1= result;
   }
   for (auto &i_obj1 : *muons1)
   {
     _col13 = i_obj2.pt();
     myTree->Fill();
   }
 }
 . . .
```

Output root file



Approach

- Study the structure of the func_adl code
- Identify the difference between CMSAOD and miniAOD
- Write C++ file manually in container, find the correct form of EDAnalyzer

Approach

CMS MiniAOD

```
. . .
edm::EDGetTokenT<pat::MuonCollection> token0;
. . .
{
  {
    // Inside Constructor
    token0 = consumes<pat::MuonCollection>(edm::InputTag("slimmedsMuons"));
  }
}
. . .
 {
   Handle<pat::MuonCollection> muons1;
   {
     Handle<pat::MuonCollection> result;
     iEvent.getByToken(token0, result);
     muons1= result;
   }
   for (auto &i_obj1 : *muons1)
   {
     _col13 = i_obj2.pt();
     myTree->Fill();
    }
 }
 . . .
```

CMS AOD

```
. . .
. . .
. . .
{
  {
    // Inside Constructor
    . . .
  }
}
. . .
 {
   Handle<pat::MuonCollection> muons1;
   {
     Handle<pat::MuonCollection> result;
     iEvent.getByLabel(“globalMuons”, result);
     muons1= result;
   }
   for (auto &i_obj1 : *muons1)
   {
     _col13 = i_obj2.pt();
     myTree->Fill();
    }
 }
 . . .
```

Approach

```python
def get_collection(self, md: EventCollectionSpecification, call_node: ast.Call):
    r'''
    Return a cpp ast for accessing the jet collection with the given arguments.
    '''
    # Get the name jet collection to look at.
    if len(call_node.args) != 1:
        raise ValueError(f"Calling {md.name} - only one argument is allowed")
    if not isinstance(call_node.args[0], ast.Str):
        raise ValueError(f"Calling {md.name} - only acceptable argument is a string")

    # Fill in the CPP block next.
    r = cpp_ast.CPPCodeValue()
    r.args = ['collection_name', ]
    r.include_files += md.include_files
    r.link_libraries += md.libraries

    self.get_running_code_CPPCodeValue(r, md)
    r.result = 'result'
```

```python
def get_running_code_CPPCodeValue(self, cpv: cpp_ast.CPPCodeValue, md: EventCollectionSpecification):
    # Used to build CPPCodeVlue for miniAOD
    cpv.running_code = self.get_running_code(md.container_type)
    # Specify the token name and type
    token_variable = crep.cpp_variable(self.t_name, gc_scope_top_level, ctyp.terminal(md.container_type.token_type()))
    # value of initializing the token
    token_init = f'consumes<{md.container_type.type}>(edm::InputTag(collection_name))'
    # add both token declaration and initializtion to CPPCodeValue.fields for building the cpp files
    cpv.fields.append((token_variable, token_init))
```

Future Work

- Enable the support for more HEP data formats, such as NanoAOD
- Add more methods types so more functions of EDAnalyzer will be supported

Acknowledgement

I would like to thank all my mentors:
Prof. Gordon Watts,
Dr.Alex Held,
Dr. Oksana Shadura,
Dr. Ben Galewsky,
Mason Proffitt
for their huge help on this project!

Also, I would like to thank the generous help from

Baidyanath Kundu,
Dr. David Lange

Thank you!

Questions?