ANALYSIS FRAMEWORK

Giulio Eulisse

A Trivial Analysis...

- Define a standalone workflow
- Define an AnalysisTask
- Get all the tracks for a given timeframe
- Compute (e.g.) φ from the propagation parameters
- Fill a histogram previously declared as OutputObj

```cpp
#include "Framework/runDataProcessing.h"
#include "Framework/AnalysisTask.h"

#include <TH1F.h>

using namespace o2;
using namespace o2::framework;

class ATask : public AnalysisTask
{
 public:
  OutputObj<TH2F> hPhi{TH1F("phi", "Phi", 100, 0., 2. * M_PI, 102, -2.01, 2.01)};

  void process(aod::Tracks const& tracks)
  {
    for (auto& track : tracks) {
      float phi = asin(track.snp()) + track.alpha() + M_PI;
      hPhi->Fill(phi);
    }
  }
};

WorkflowSpec defineDataProcessing(ConfigContext const&)
{
  return WorkflowSpec{
    adaptAnalysisTask<ATask>("mySimpleTrackAnalysis", 0)
  };
}
```

Architecture in One Slide

Data model is built from **flat tables** and associations between them.

An **object oriented API layer** is built on top of the flat model.

Analysis are organised in "**Tasks**". Tasks are automatically arranged in a workflow via DPL.

Two parts for each task:

- **Declarative**: outputs, filters, partitions, simple mapping operations. The more you can describe your analysis as declarative, the better optimisation chances.
- **Imperative**: the "process" method. This part is meant to simplify porting from AliPhysics and in general to provide a more Object Oriented look and feel to the user.

The arguments of the process method or the inputs of declarative statements are used to **subscribe to the needed data** (and not more).

Subscribing to Data

Process all the tracks in a timeframe:

```
void process(o2::aod::Tracks const& tracks) {
...
}
```

Process one track in a timeframe at the time (notice missing `s`):

```
void process(o2::aod::Track const& track) {
...
}
```

Process one collision at the time, with the associated tracks:

```
void process(o2::aod::Collision const& collision, o2::aod::Tracks const& tracks) {
...
}
```

Process one collision at the time, with the associated tracks and extras:

```
void process(o2::aod::Collision const& collision, Join<o2::aod::Tracks, o2::aod::TracksExtra> const& tracks) {
...
}
```

Merging Tables

It is possible to use the Join and Concat (and soon Merge) helpers to subscribe to table combinations.

Join: *allows you to join two (or more) tables row-wise. E.g. to access Tracks and TracksExtra properties from the object:*

```
process(Join<Tracks, TracksExtra> const& fullTracks);
```

Concat: *selects the **common columns of two tables** and create a new table with the union of their rows. Order will be preserved.*

Merge: *will select all the unique columns of two tables, and create a new table with the union of their rows. If a table does not have a column, empty "NULL" cells will be created as required and a "hasProperty()" method will be provided to probe values.*

Histograms

```
struct ATask {
  OutputObj<TH1F> phiH{TH1F("phi", "phi", 100, 0., 2. * M_PI)};

  void process(aod::Tracks const& tracks)
  {
    for (auto& track : tracks) {
      phiH->Fill(track.phi());
    }
  }
};
```

Declare histogram in output

Fill it through smart pointer.

At the end of processing histograms will be written in an AnalysisResults.root

Filters and Partitions

```
struct ATask {
  Filter ptFilter = (aod::track::pt > 1) && (aod::track::pt < 10);
  Partition etaPartition = aod::track::eta > 0;

  void process(aod::Tracks const& tracks)
  {
    for (auto& positiveTrack : etaPartition(tracks)) {
      for (auto& negativeTrack : (!etaPartition)(tracks)) {
    ...
    }
   }
  }
};
```

Define filter

Define partition

Only subscribe to filtered tracks

Iterate on a partition or its conjugate

Arrow Skins: Data Definition Example

```
#include "Framework/ASoA.h"

namespace o2::aod
{
namespace track
{
DECLARE_SOA_COLUMN(CollisionId, collisionId, int, "fID4Tracks");
DECLARE_SOA_COLUMN(Alpha, alpha, float, "fAlpha");
DECLARE_SOA_COLUMN(Snp, snp, float, "fSnp");
//...
DECLARE_SOA_DYNAMIC_COLUMN(Phi, phi,
  [](float snp, float alpha) { return asin(snp) + alpha + M_PI; });
} // namespace track

using Tracks = soa::Table<track::CollisionId, track::Alpha,
                          /* ... */,
                          track::Snp, track::Tgl,
                          track::Phi<track::Snp, track::Alpha>>;

using Track = Tracks::iterator;
}
```

Column

The smallest component is the Column, which is a type mapped to a specific column name.

Table

A Table is a generic union of Column types.

Dynamic Columns

Non persistent (i.e. calculated) quantities can be associated with a table in the form of a so called dynamic column.

Object

An object is actually an alias to the simultaneous iterators over the columns involved in a given table row.

Creating new Data

```
namespace o2::aod {
namespace etaphi {
DECLARE_SOA_COLUMN(Eta, eta, float, "fEta");
DECLARE_SOA_COLUMN(Phi, phi, float, "fPhi");
} // namespace etaphi
DECLARE_SOA_TABLE(EtaPhi, "RN2", "ETAPHI",
                  etaphi::Eta, etaphi::Phi);
} // namespace o2::aod

struct ATask {
  Produces<aod::EtaPhi> etaphi;

  void process(aod::Tracks const& tracks)
  {
    for (auto& track : tracks) {
      auto phi = asin(track.snp()) + track.alpha() + M_PI;
      auto eta = log(tan(0.25 * M_PI - 0.5 * atan(track.tgl())));
      etaphi(phi, eta);
    }
  }
};

struct BTask {
  void process(aod::EtaPhi const& etaPhis) {
    ...
  }
};
```

Define EtaPhi table

Declare we will produce EtaPhi

Fill the table (for now, orders matters)

Subscribe to the newly
created data in a different task.

ANALYSIS FRAMEWORK: COMBINATIONS

Combinations enumerator: *substitutes double / multiple loops with a generator for the pairs / tuples.*

From:

```
for (auto it0 = tracks.begin(); it0 != tracks.end(); ++it0) {
  auto& track0 = *it0;
  for (auto it1 = it0 + 1; it1 != tracks.end(); ++it1) {
    auto& track1 = *it1;
    ...
  }
}
```

to:

```
for (auto &[track0, track1] : combinations(tracks, tracks)) { ... ; }
```

ANALYSIS FRAMEWORK: CONFIGURABLE PARAMETERS

Configurable parameters for AnalysisTasks:

```
struct Task {
  Configurable<float> myCut{"myCut", 1.f, "Some user defined cut"};

  void process(Track const& track) {
    if (track.pt() > myCut) {
      ...
    }
  }
};
```

Can be customised on the command line (`--myCut 2.`) and will be configurable from the new train the infrastructure.

ANALYSIS FRAMEWORK: CONFIGURABLE PARAMETERS

It will be possible soon to use Configurables in filters

```
struct Task {
  Configurable<float> myCut{"myCut", 1.f, "Some user defined cut"};
  Filter myFilter = track::pt > myCut;

  void process(Track const& track) {
  ...
  }
};
```

If you will do this, it will be MUCH faster.

ANALYSIS FRAMEWORK: INDEX COLUMNS

Index columns and "row" getters.

```
namespace track {
DECLARE_SOA_INDEX(Collision, collision);
}
...
DECLARE_SOA_TABLE(Tracks, "AOD", "TRACKPAR", track::CollisionId, ...);
...
process(Track const&track) {
  track.collision().posX();
}
```

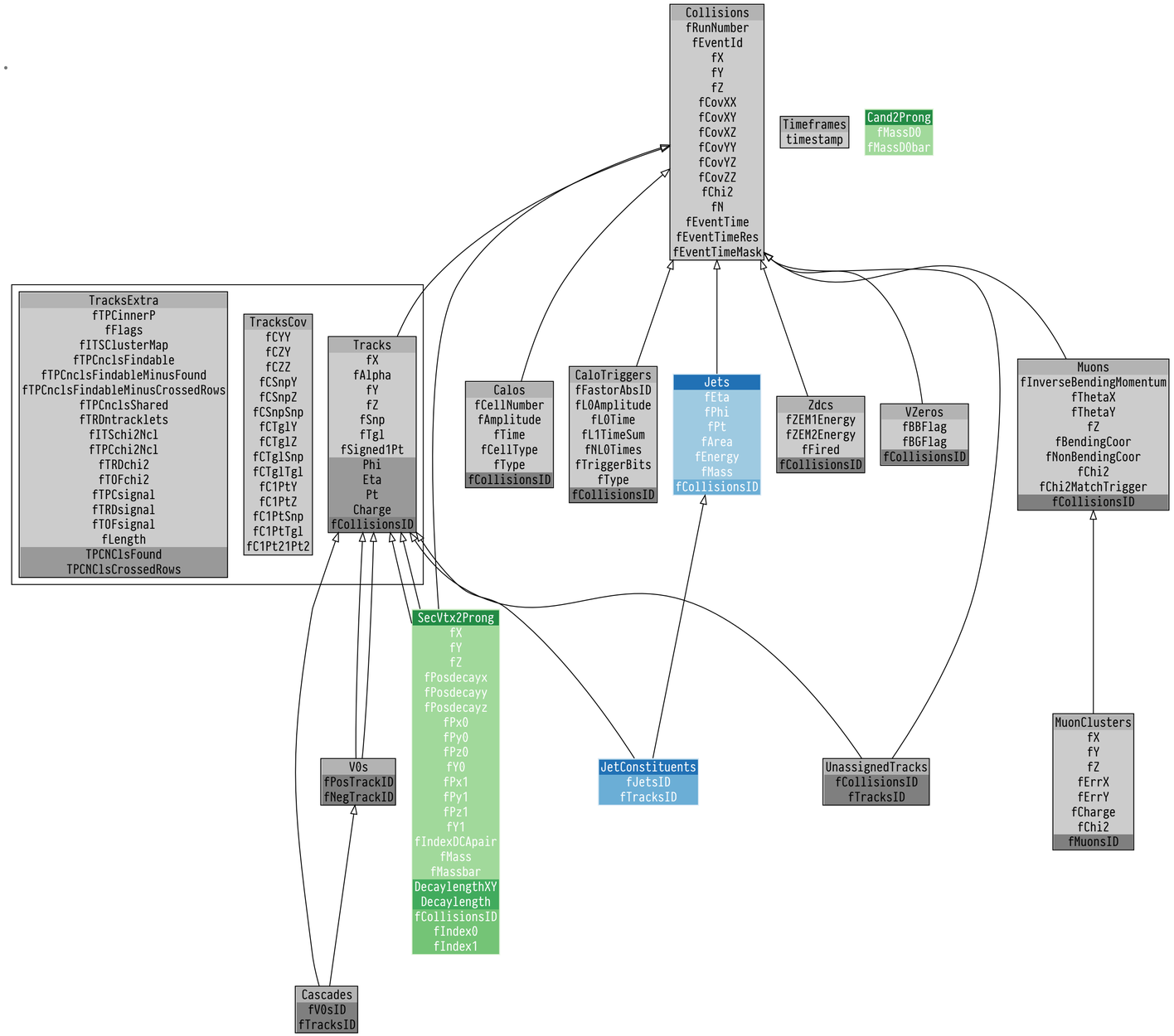
Generic indices:

This can be used also to have multiple indices to the same table:

```
DECLARE_SOA_INDEX_COLUMN_FULL(Pos, pos, int, Tracks, "fPosTrackID");
DECLARE_SOA_INDEX_COLUMN_FULL(Neg, neg, int, Tracks, "fNegTrackID");
```

ANALYSIS SCHEMA (SO FAR)

- *Persistent Tables (i.e. Peter's Converter)*
- *Vertexer HF Task (i.e. Gian Michele's)*
- *Jet Finder Task (i.e. Jochen's)*



Extra Material

Design document for the Framework at https://github.com/AliceO2Group/AliceO2/blob/dev/Framework/Core/ANALYSIS.md

Code samples in https://github.com/AliceO2Group/AliceO2/tree/dev/Analysis/Tutorials

Other presentations:

- Original presentation with the vision https://indico.cern.ch/event/710009/contributions/2922639/
- Presentation earlier this year together with Gian Michele https://indico.cern.ch/event/804873/contributions/3348872/
- Presentation in the Physics Week https://indico.cern.ch/event/797644/contributions/3476295/
- CHEP presentation https://indico.cern.ch/event/773049/contributions/3476164/attachments/1936146/3212074/2019-11-chep-data-analysis.pdf