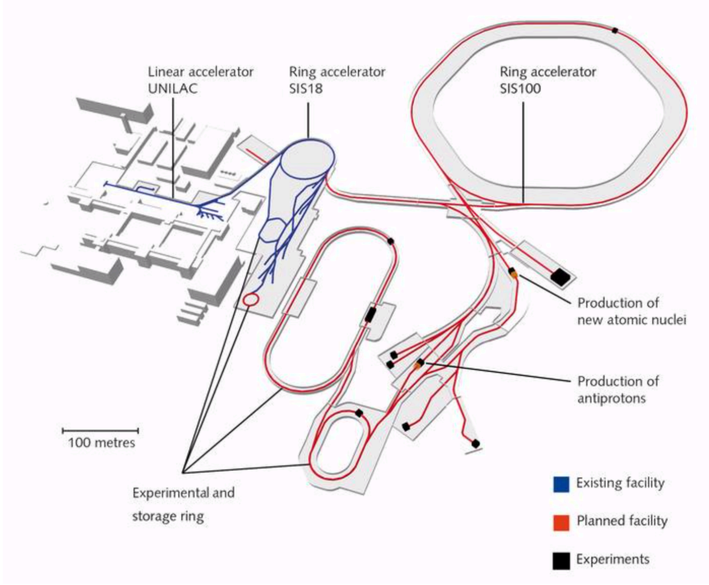
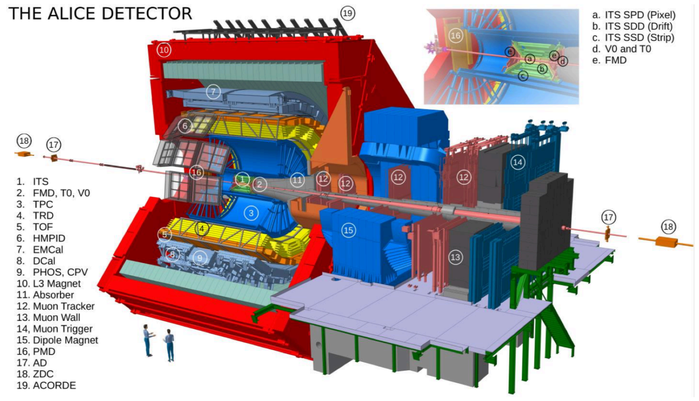
ALICE / FAIR FRAMEWORK EFFORTS

Giulio Eulisse - CERN

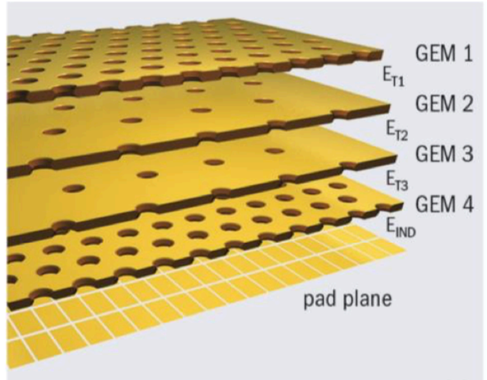
ALICE - FAIR FRAMEWORK COLLABORATION

- **Goal:** develop and support common software solutions for the Run3 of the ALICE LHC experiment and the upcoming experiments at the Facility for Antiproton and Ion Research in Europe (FAIR) being built at GSI.
- Based on the experiences of **ALICE HLT** in Run1 / Run2 and the of the **FairRoot** framework.
- One of the examples of fruitful collaboration on Software Frameworks & Toolkits in HEP.
- I modestly contribute to it as part of the CERN ALICE Team, in particular to the so called Data Processing Layer.





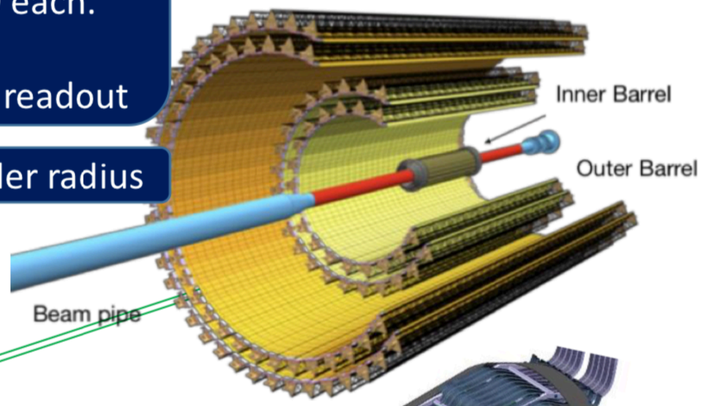
ALICE HW upgrades



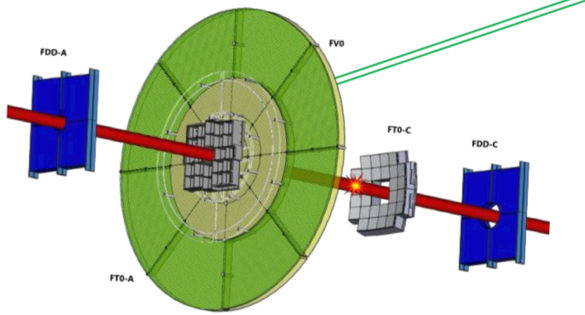
TPC MWPC readout → 4 layer GEM
(Intrinsic ion backflow ~99% blocking)
5MHz continuous sampling

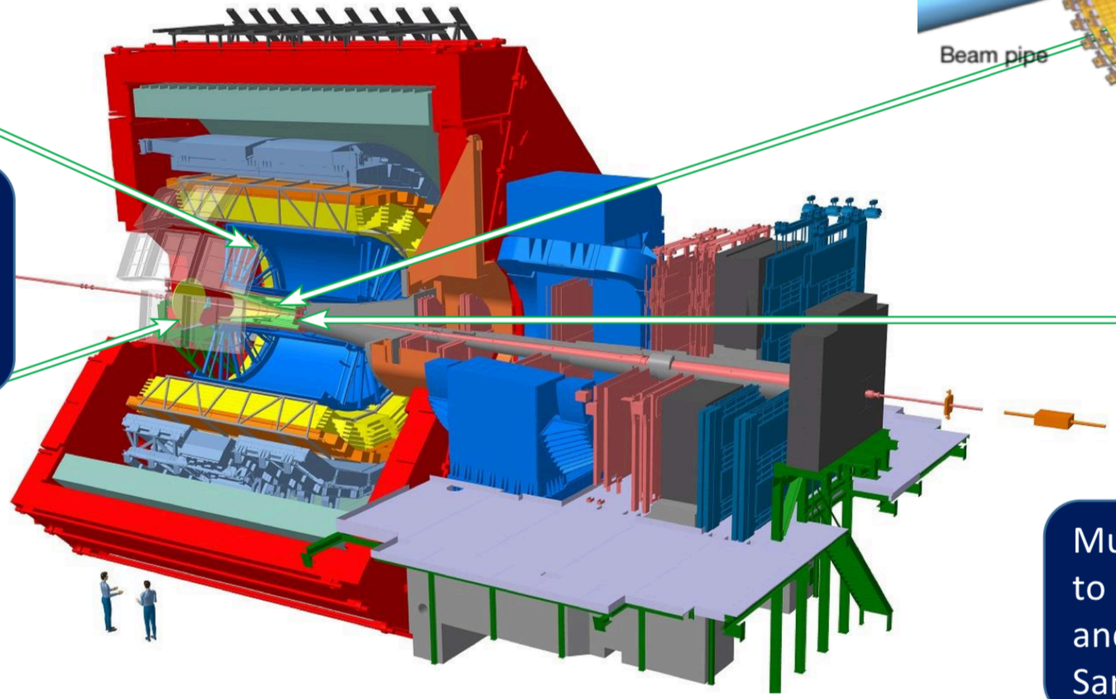
New Si Inner Tracker: 10 m^2 of MAPS with 29x27μm^2 pixel size
3 inner layers ~0.3% X0 each.
Closer to the beam
50-500 kHz continuous readout

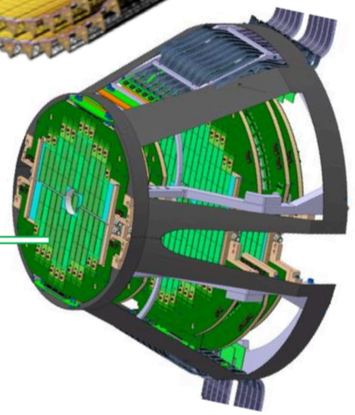
New beam pipe of smaller radius



Fast Interaction Trigger (FIT) detector
Scintillator (FV0, FDD) + Cerenkov (FT0) detectors to provide Min.Bias trigger for detectors with triggered R/O



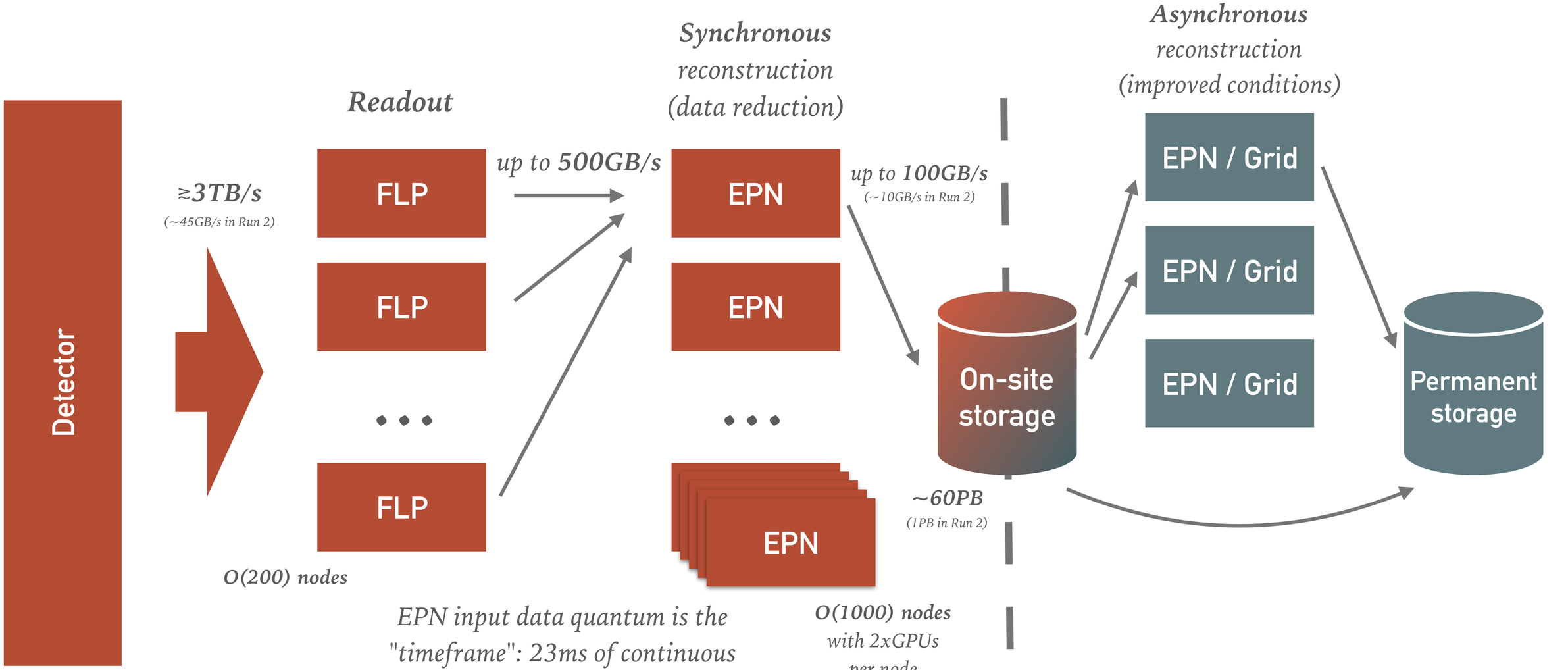




Muon Forward Tracker
to match muons before and after the absorber.
Same Si chips as new ITS

See CHEP2019 Plenary: ALICE continuous readout and data reduction strategy for Run3

ALICE in Run 3: Point 2

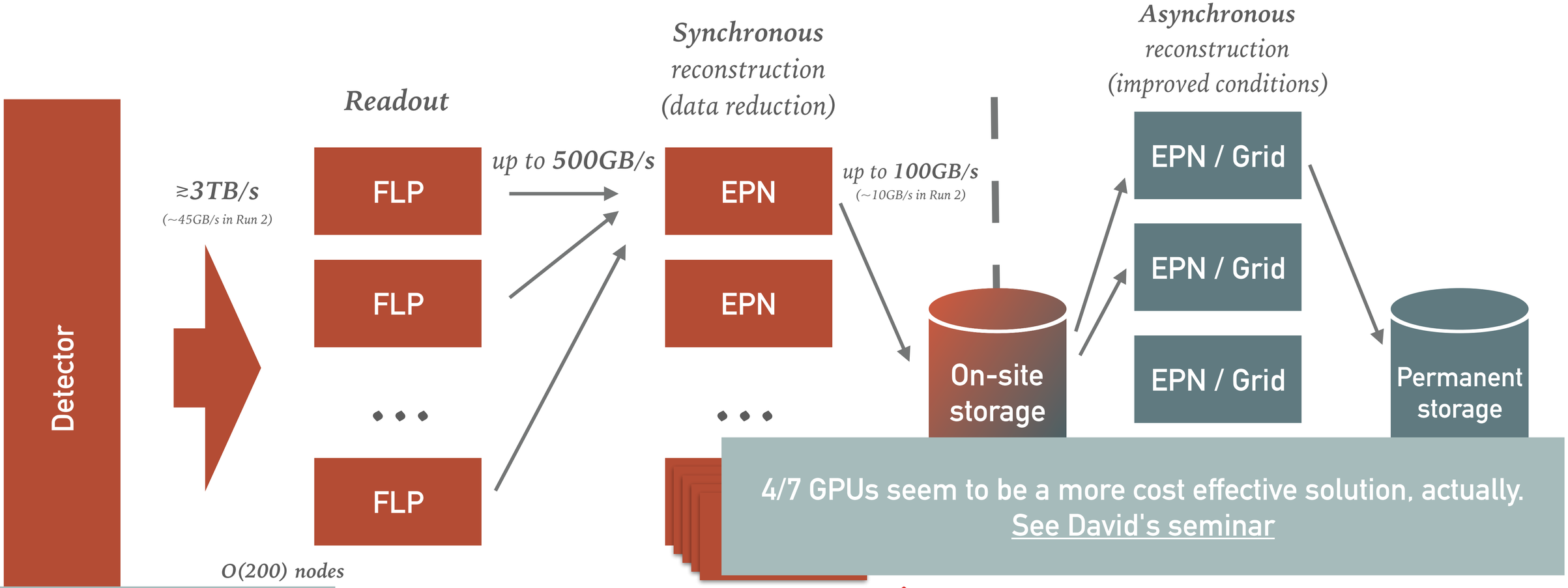


EPN input data quantum is the "timeframe": 23ms of continuous readout data. ~10GB

BEAM ON: data reduction

BEAM OFF: improved calibration

ALICE in Run 3: Point 2



4/7 GPUs seem to be a more cost effective solution, actually.
See David's seminar

O(200) nodes

Now ~11ms to fit memory due to strategy change in handling TPC clusters.

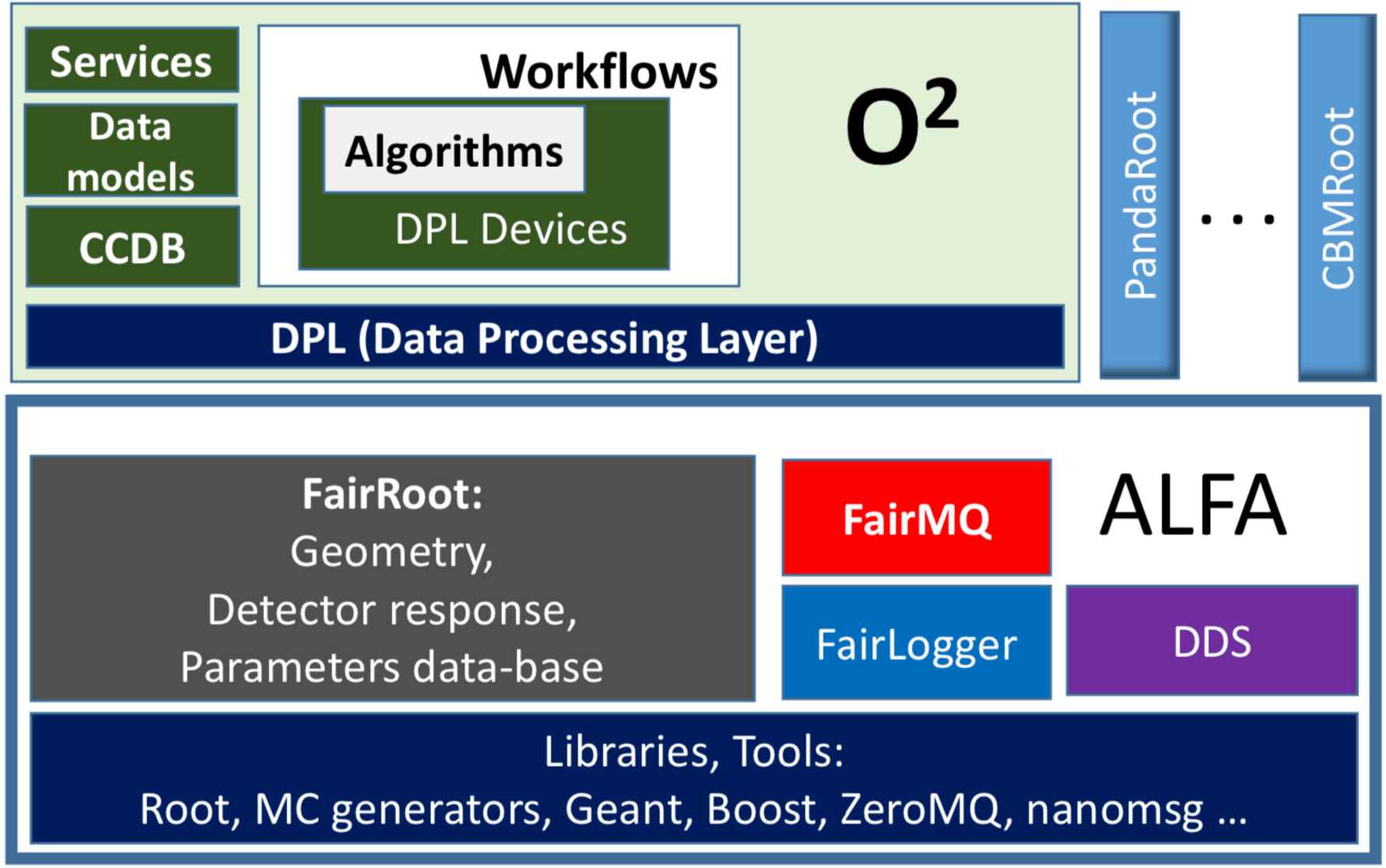
EPN input data quantum is the "timeframe": ms of continuous readout data. ~10GB

O(1000) nodes with GPUs

BEAM ON: data reduction

BEAM OFF: improved calibration

THE BIG PICTURE



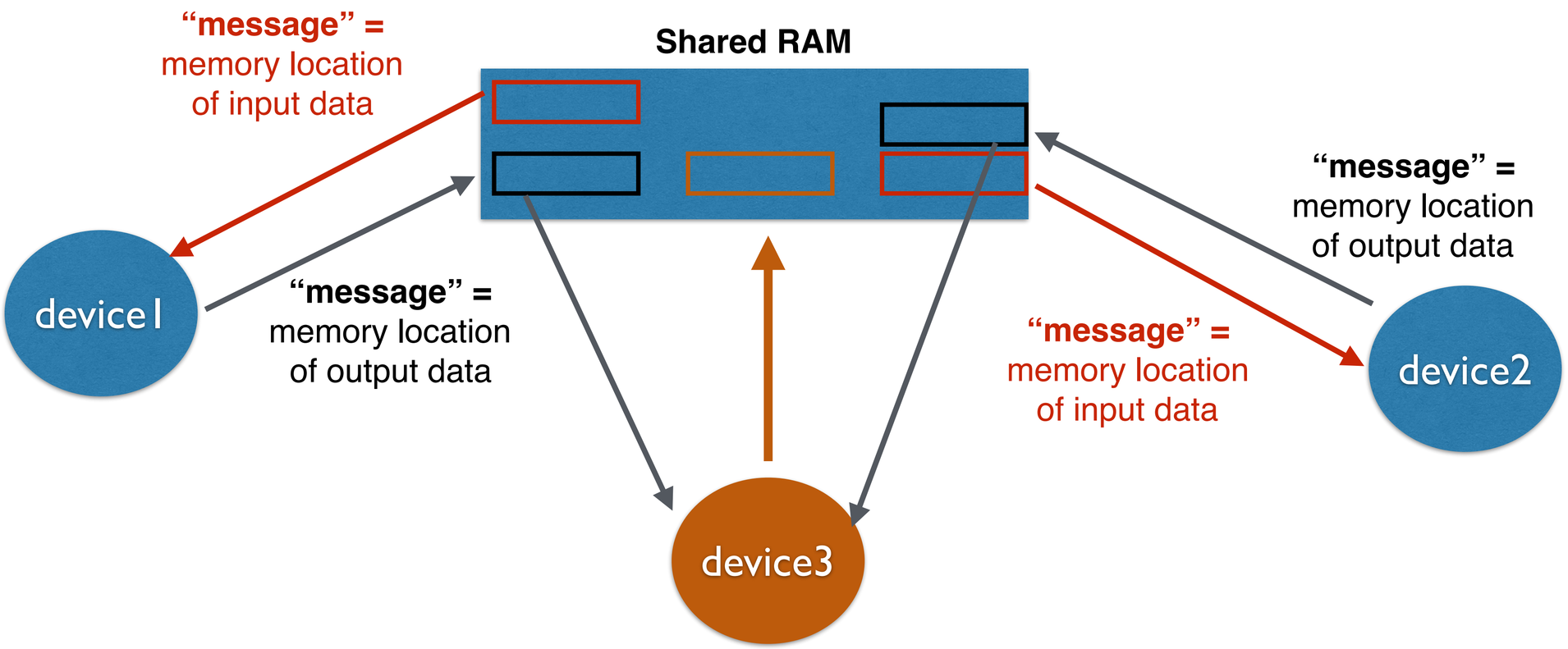
ALICE O2: Software Framework in one slide

Transport Layer: ALFA / FairMQ[1]

- **Standalone processes** *for deployment flexibility.*
- **Message passing** *as a parallelism paradigm.*
- **Shared memory** *backend for reduced memory usage and improved performance.*

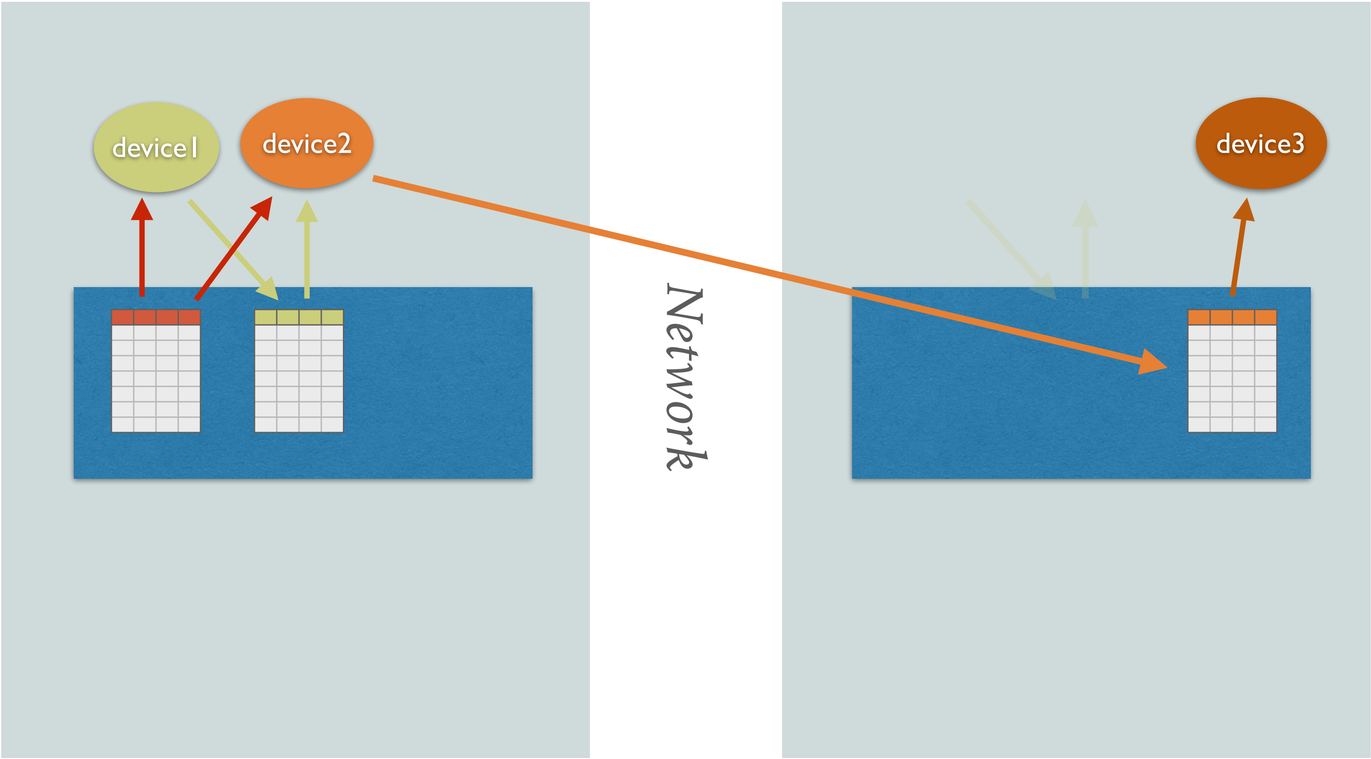
ALFA / FairMQ Framework: General Idea

Data processing happens in separate processes, called **devices**, exchanging data via a shared memory backed **Message Passing** paradigm.



ALFA / FairMQ Framework: General Idea

Seamless and homogeneous support for multi-node setups using one of the network enabled message passing backends, e.g. InfiniBand with RDMA.



SOFTWARE STACK BEHIND FAIRMQ

- Support for multiple message passing OpenSource libraries: **ZeroMQ,** nanomsg.
- In-house developed C++ bindings (**FairRootGroup/asiofi**) to OFI libfabric for InfiniBand support.
- Adoption of **boost::interprocess** for the shared memory backend.
- Support for multiple message serialisation (or not) protocols. Transport is agnostic about actual message content, allowing implementor to use their preferred technology: protobuf, flatbuffers, detector specific, **Apache Arrow**.
- State machine with pluggable support for deployment / control services: DDS, O^2 AliECS, PMIx or "standalone".
- See Mohammad CHEP 2019 talk.

ALICE O2: Software Framework in one slide

Transport Layer: ALFA / FairMQ[1]

- **Standalone processes** *for deployment flexibility.*
- **Message passing** *as a parallelism paradigm.*
- **Shared memory** *backend for reduced memory usage and improved performance.*

ALICE O2: Software Framework in one slide

Data Layer: O2 Data Model

Message passing aware data model. Support for multiple backends:

- **Simplified, zero-copy** *format optimised for performance and direct GPU usage. Useful e.g. for TPC reconstruction on the GPU.*
- **ROOT based serialisation.** *Useful for QA and final results.*
- **Apache Arrow based.** *Useful as backend of the analysis ntuples and for integration with other tools.*

Transport Layer: ALFA / FairMQ[1]

- **Standalone processes** *for deployment flexibility.*
- **Message passing** *as a parallelism paradigm.*
- **Shared memory** *backend for reduced memory usage and improved performance.*

O2 DATA MODEL

A timeframe is a collection of `(header, payload)` *pairs. Headers defines the type of data. Different header types can be stacked to store extra metadata (mimicking a Type hierarchy structure). Both header and payloads should be usable in a* ***message passing*** *environment.*

Different payloads might have ***different serialisation strategies****. E.g.:*

- *TPC clusters / tracks:* ***flat POD data with relative indexes****, well suitable for GPU processing.*
- *QA histograms:* ***serialised ROOT*** *histograms.*
- *AOD:* ***some columnar data format****. Multiple solutions being investigated.*

ALICE O2: Software Framework in one slide

Data Processing Layer (DPL)

Abstracts away the hiccups of a distributed system, presenting the user a familiar "Data Flow" system.

- ***Reactive-like design*** *(push data, don't pull)*
- ***Declarative Domain Specific Language*** *for topology configuration (C++17 based).*
- ***Integration*** *with the rest of the production system, e.g. Monitoring, Logging, Control.*
- ***Laptop mode****, including graphical debugging tools.*

Data Layer: O2 Data Model

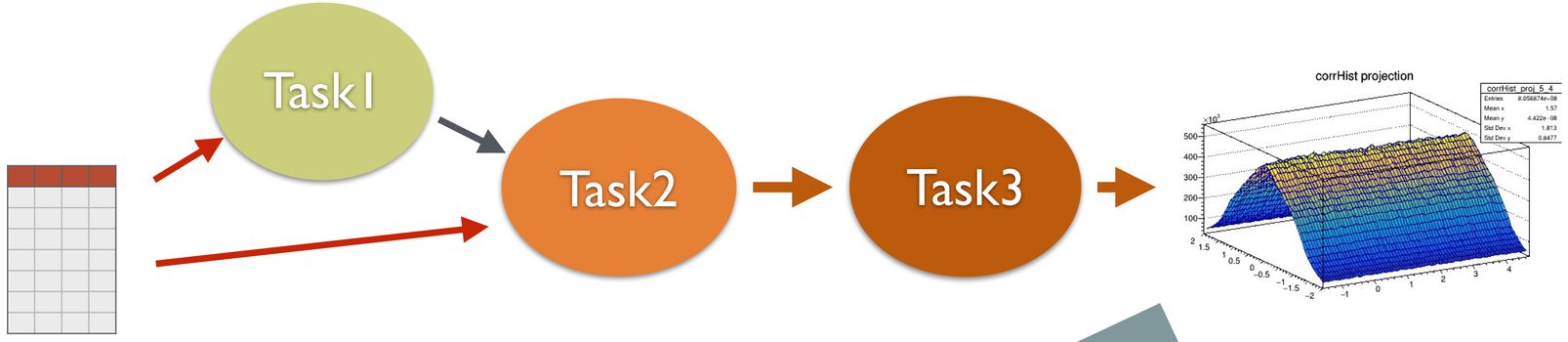
Message passing aware data model. Support for multiple backends:

- **Simplified, zero-copy** *format optimised for performance and direct GPU usage. Useful e.g. for TPC reconstruction on the GPU.*
- **ROOT based serialisation.** *Useful for QA and final results.*
- **Apache Arrow based.** *Useful as backend of the analysis ntuples and for integration with other tools.*

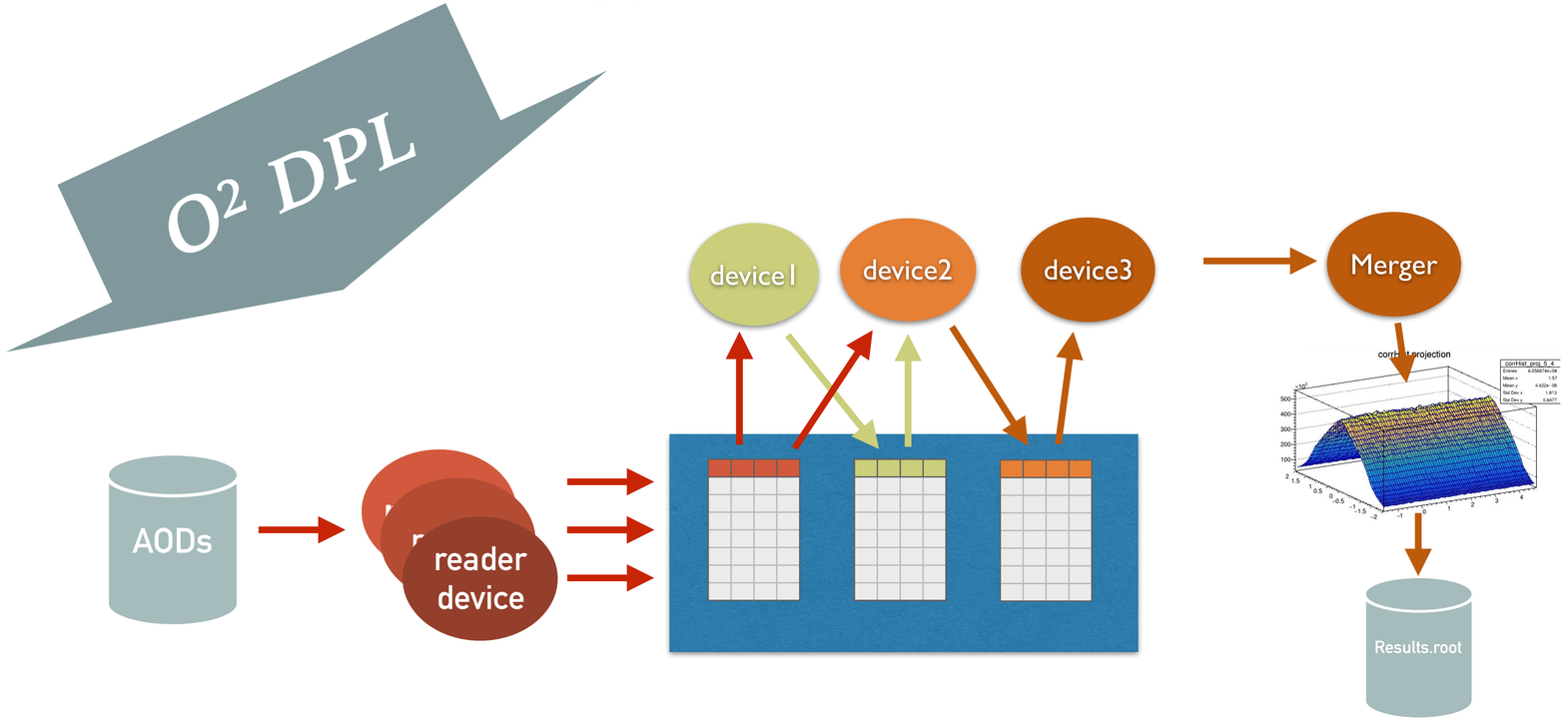
Transport Layer: ALFA / FairMQ[1]

- **Standalone processes** *for deployment flexibility.*
- **Message passing** *as a parallelism paradigm.*
- **Shared memory** *backend for reduced memory usage and improved performance.*

AliceO2 Data Processing Layer



User provides a description in terms of tasks and physics quantities.

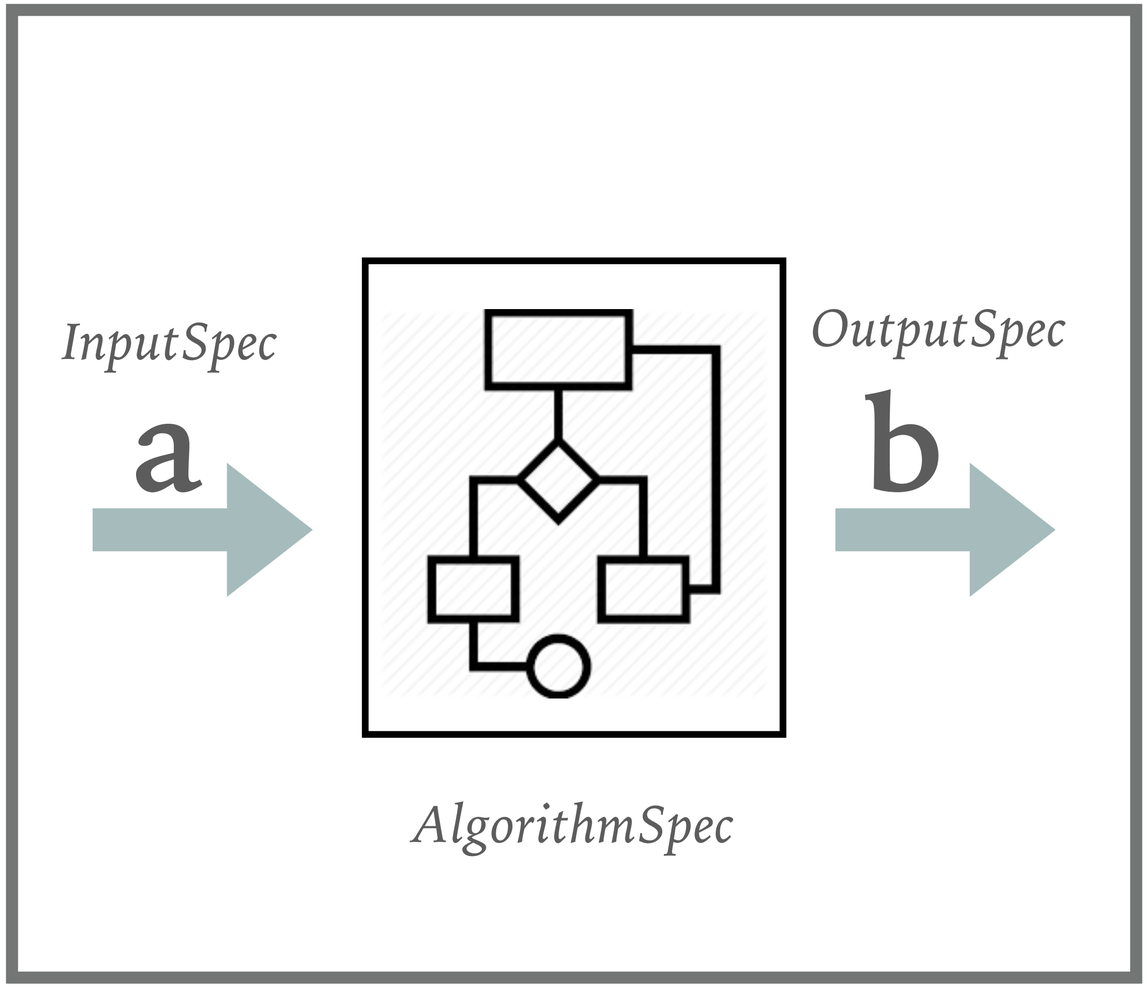


AliceO2 Data Processing Layer (DPL) translates the implicit workflow(s) defined by physicists to an actual FairMQ topology of devices, injecting readers and merger devices, completing the topology and taking care of parallelism / rate limiting.

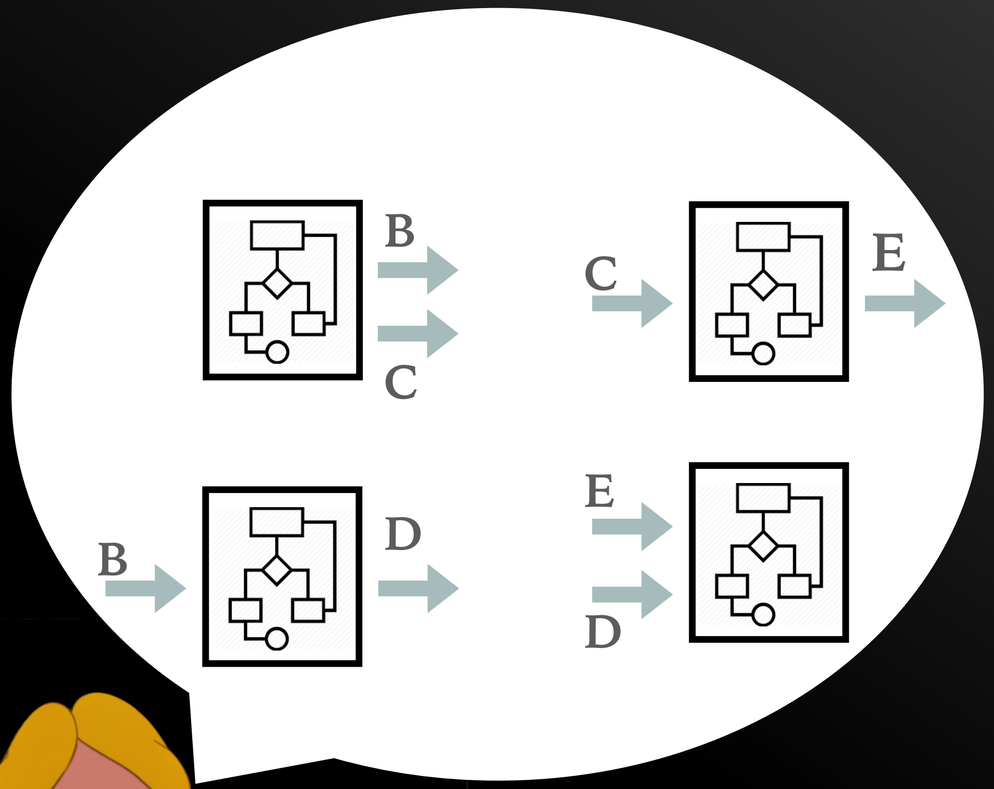
DATA PROCESSING LAYER: BUILDING BLOCK

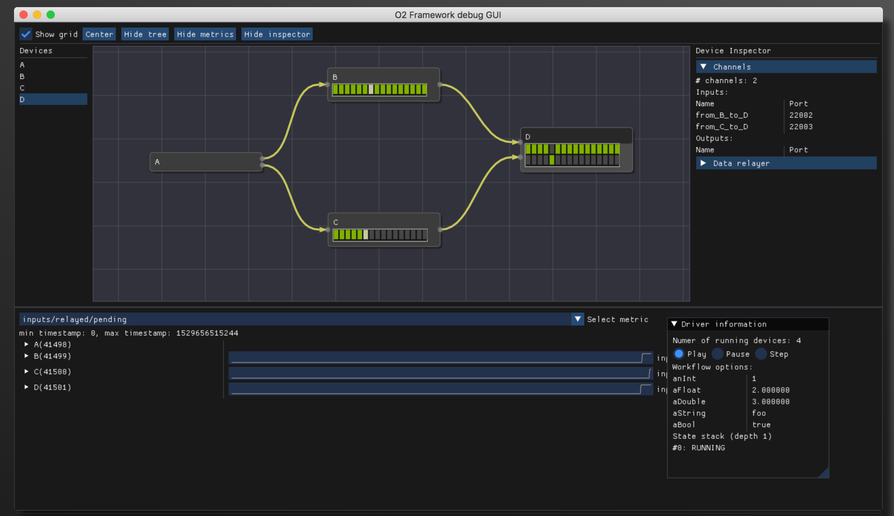
A DataProcessorSpec *defines a pipeline stage as a building block.*

- *Specifies* ***inputs and outputs*** *in terms of the O2 Data Model descriptors.*
- *Provide an implementation of how to act on the inputs to produce the output.*
- *Advanced user can express possible data or time parallelism opportunities.*



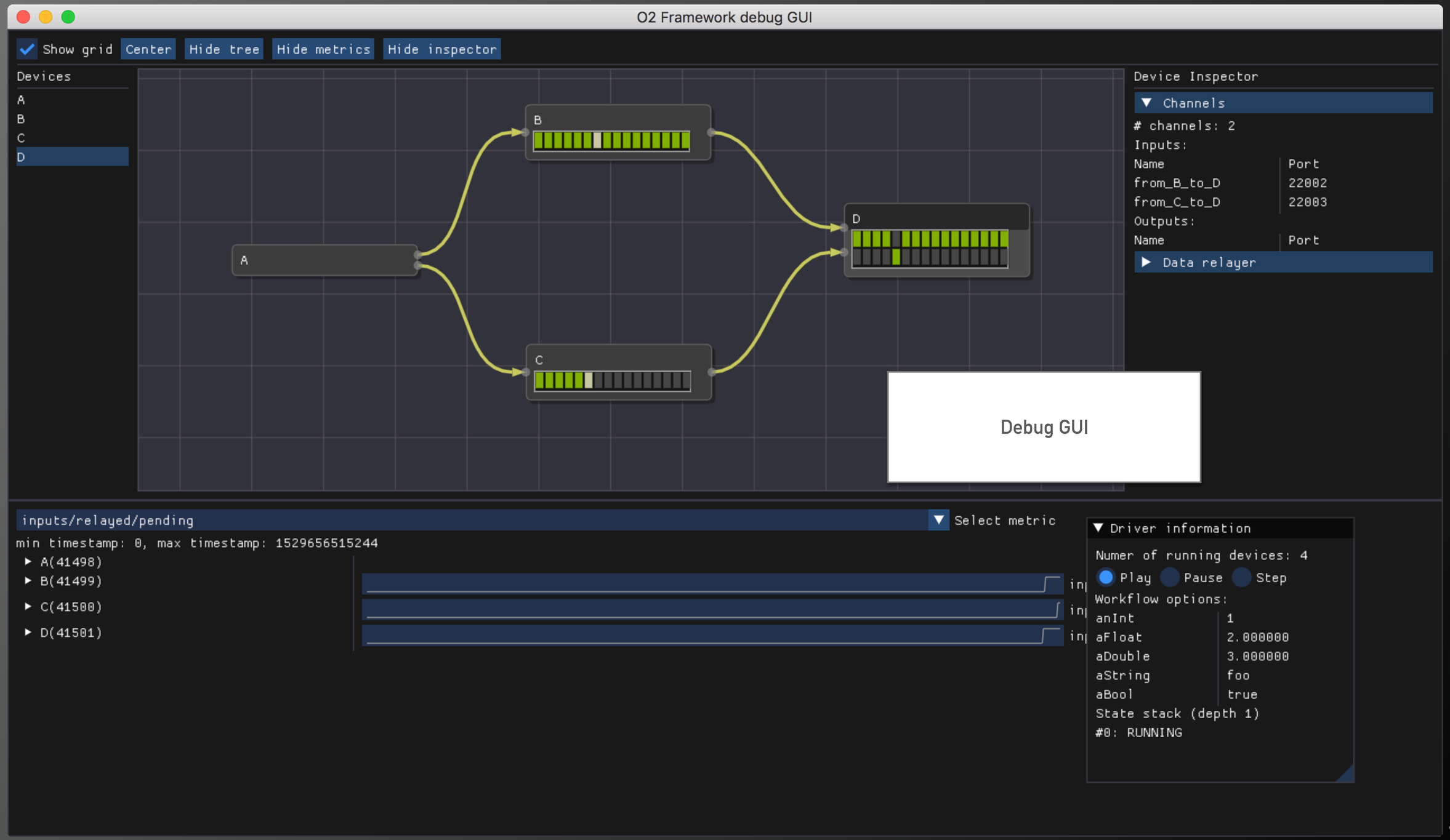
DATA PROCESSING LAYER: IMPLICIT TOPOLOGY

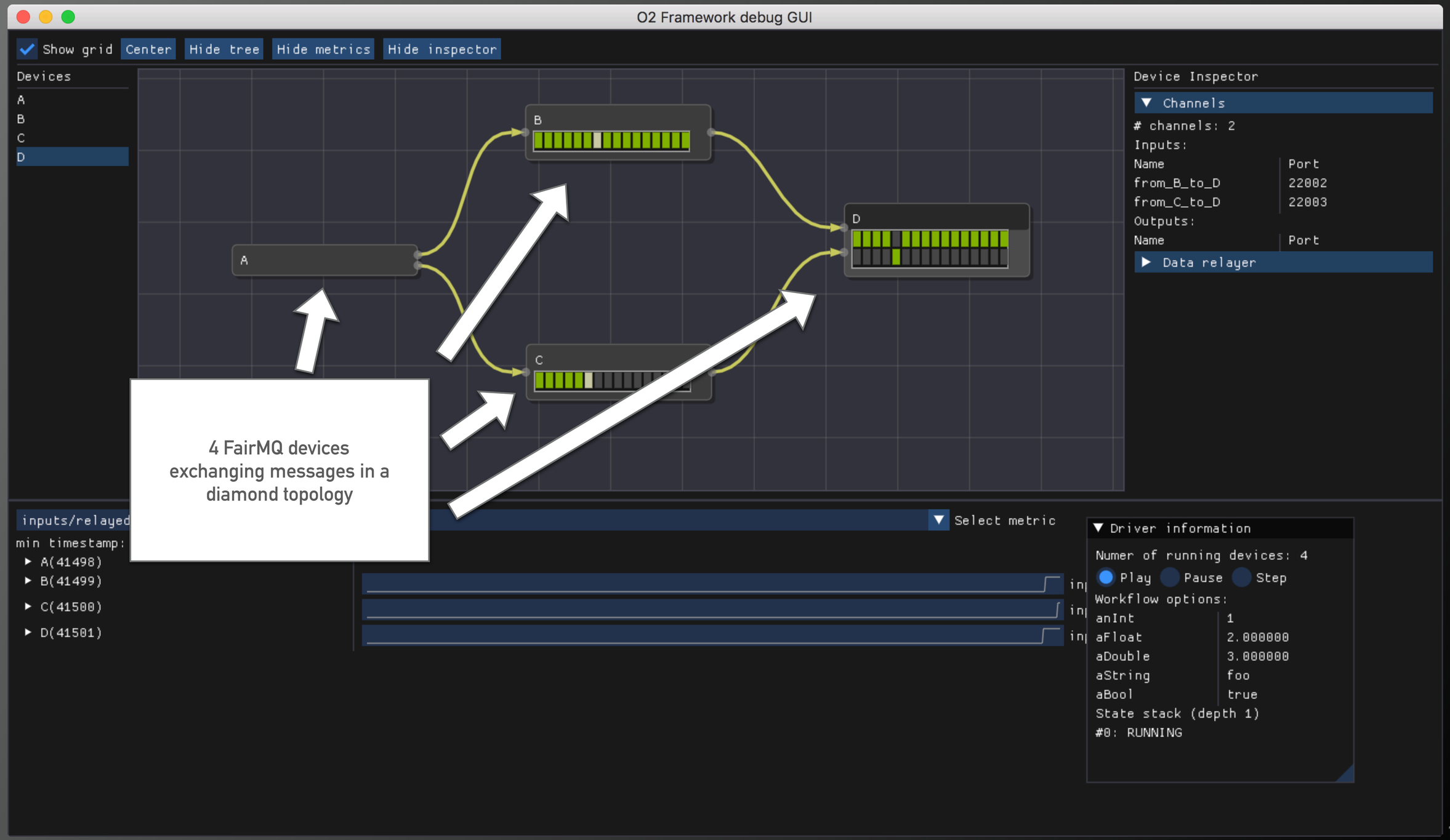


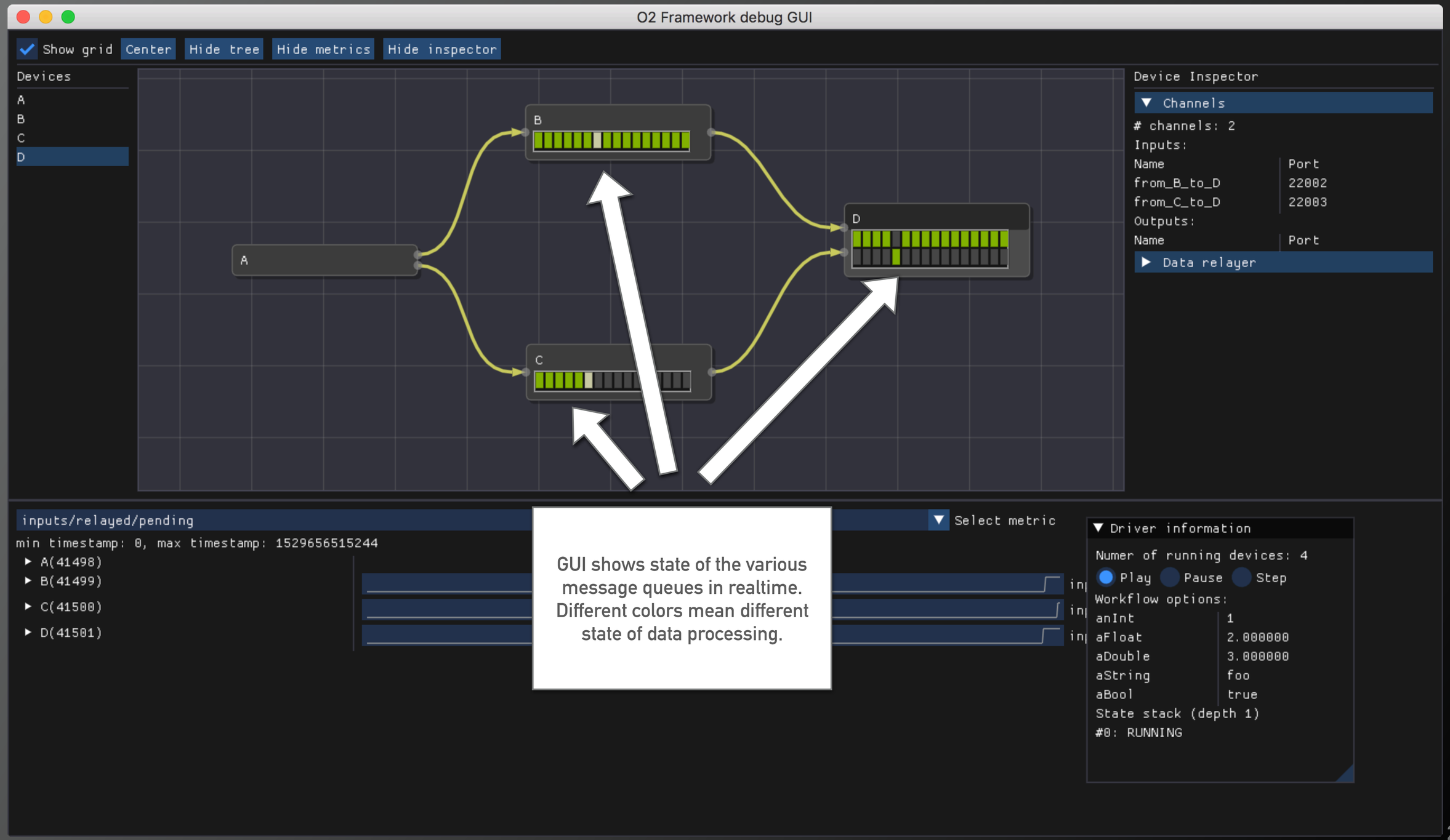


Data Processing Layer

Topology is defined implicitly.
Topological sort ensures a viable dataflow is constructed (no cycles!).
Laptop users gets immediate feedback through the debug GUI.
Service API allows integration with non data flow components (e.g. Control)


O2 Framework debug GUI
Show grid
Center
Hide tree
Hide metrics
Hide inspector
Devices
A
B
C
D
A
B
C
D
Device Inspector
Channels
channels: 2
Inputs:
Name
Port
from_B_to_D
22002
from_C_to_D
22003
Outputs:
Name
Port
Data relayer
Debug GUI
inputs/relayed/pending
Select metric
min timestamp: 0, max timestamp: 1529656515244
A(41498)
B(41499)
C(41500)
D(41501)
Driver information
Numer of running devices: 4
Play
Pause
Step
Workflow options:
anInt
1
aFloat
2.000000
aDouble
3.000000
aString
foo
aBool
true
State stack (depth 1)
#0: RUNNING


O2 Framework debug GUI
Show grid
Center
Hide tree
Hide metrics
Hide inspector
Devices
A
B
C
D
Device Inspector
Channels
channels: 2
Inputs:
Name
Port
from_B_to_D
22002
from_C_to_D
22003
Outputs:
Name
Port
Data relayer
inputs/relayed
min timestamp:
A(41498)
B(41499)
C(41500)
D(41501)
Select metric
Driver information
Numer of running devices: 4
Play
Pause
Step
Workflow options:
anInt
1
aFloat
2.000000
aDouble
3.000000
aString
foo
aBool
true
State stack (depth 1)
#0: RUNNING
4 FairMQ devices exchanging messages in a diamond topology

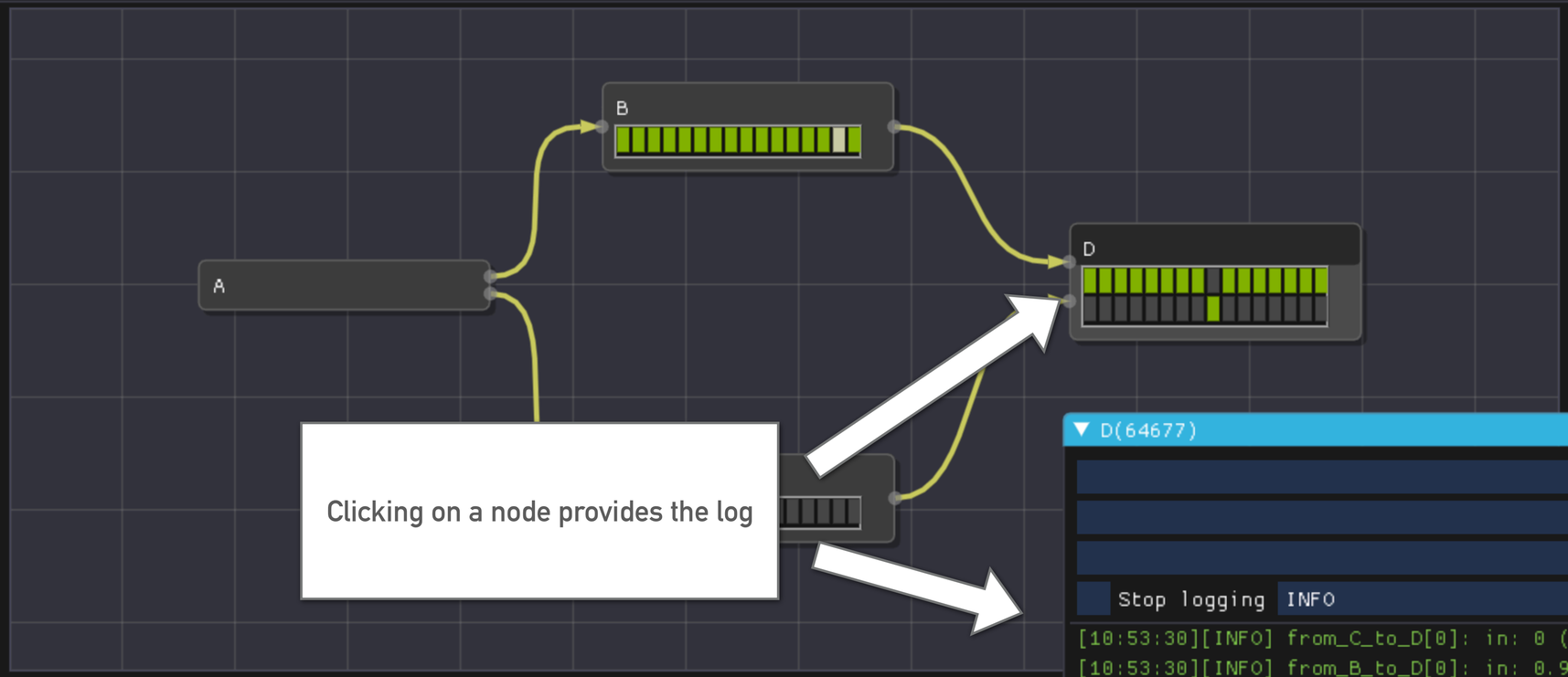


GUI shows state of the various message queues in realtime. Different colors mean different state of data processing.

O2 Framework debug GUI

Show grid | Center | Hide tree | Hide metrics | Hide inspector

Devices
- A
- B
- C
- D



Clicking on a node provides the log

Device Inspector

▼ Channels

channels: 2

Inputs:

Name	Port
from_B_to_D	22002
from_C_to_D	22003

Outputs:

Name	Port

▶ Data relayer

▼ D(64677)

Log filter

Log start trigger

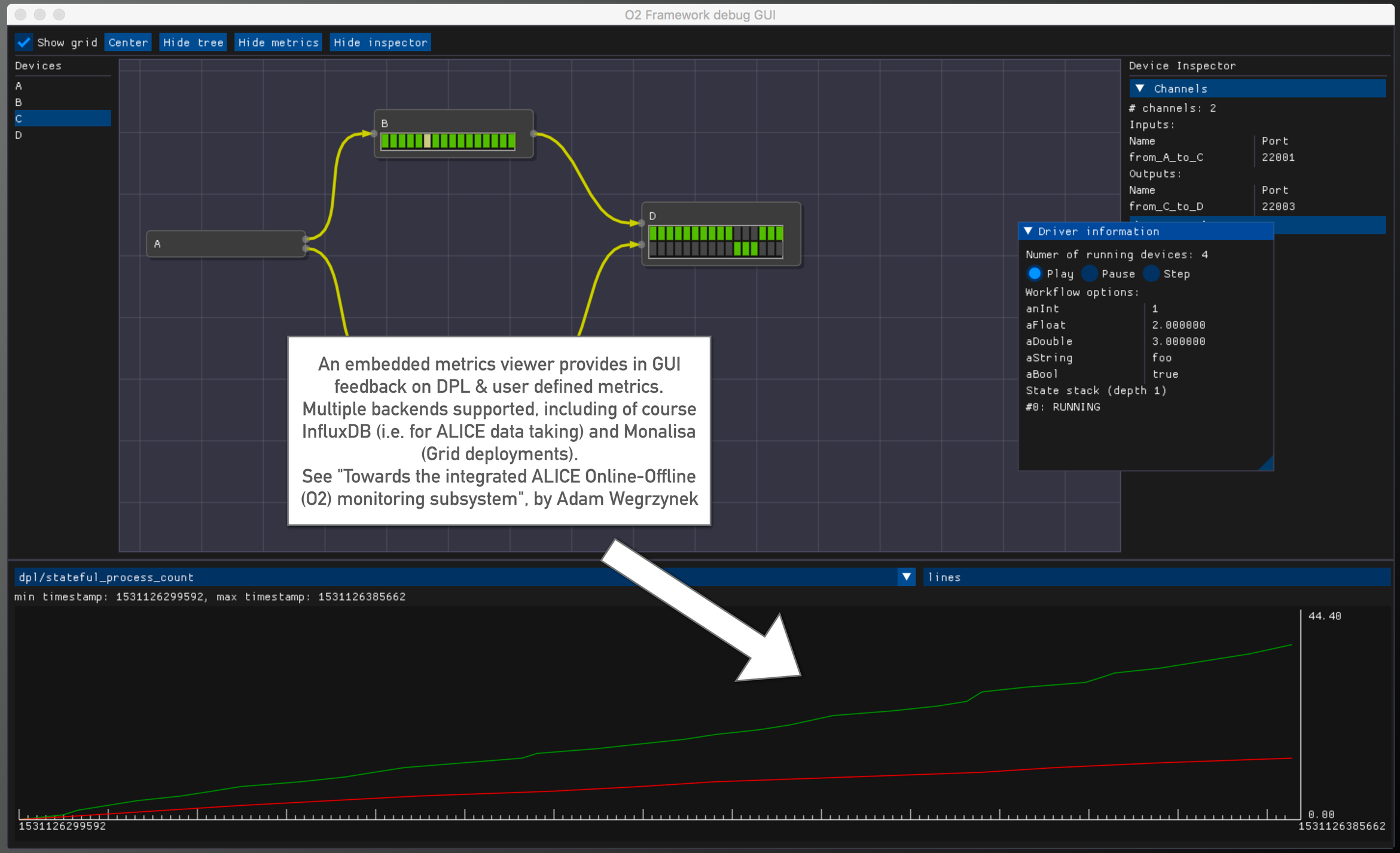
Log stop trigger

Stop logging | INFO | Log level

```
[10:53:30][INFO] from_C_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:30][INFO] from_B_to_D[0]: in: 0.999001 (0.000131868 MB) out: 0 (0 MB
[10:53:31][INFO] from_C_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:31][INFO] from_B_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:32][INFO] from_C_to_D[0]: in: 1 (0.000132 MB) out: 0 (0 MB)
[10:53:32][INFO] from_B_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:33][INFO] from_C_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:33][INFO] from_B_to_D[0]: in: 1 (0.000132 MB) out: 0 (0 MB)
[10:53:34][INFO] from_C_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:34][INFO] from_B_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:35][INFO] from_C_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:35][INFO] from_B_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:36][INFO] from_C_to_D[0]: in: 0 (0 MB) out: 0 (0 MB)
[10:53:36][INFO] from_B_to_D[0]: in: 1 (0.000132 MB) out: 0 (0 MB)
[10:53:37][INFO] from_C_to_D[0]: in: 0.995025 (0.000131343 MB) out: 0 (0 MB
[10:53:37][INFO] from_B_to_D[0]: in: 1.99005 (0.000262687 MB) out: 0 (0 MB)
```

- ▶ A(64674)
- ▶ B(64675)
- ▶ C(64676)
- ▶ D(64677)

Workflow options:



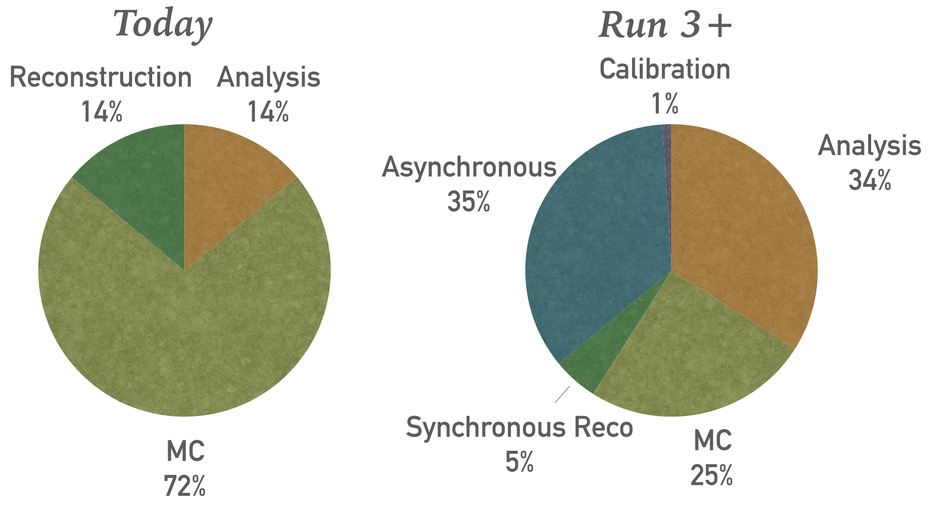
An embedded metrics viewer provides in GUI feedback on DPL & user defined metrics. Multiple backends supported, including of course InfluxDB (i.e. for ALICE data taking) and Monalisa (Grid deployments).
See "Towards the integrated ALICE Online-Offline (O2) monitoring subsystem", by Adam Wegrzynek

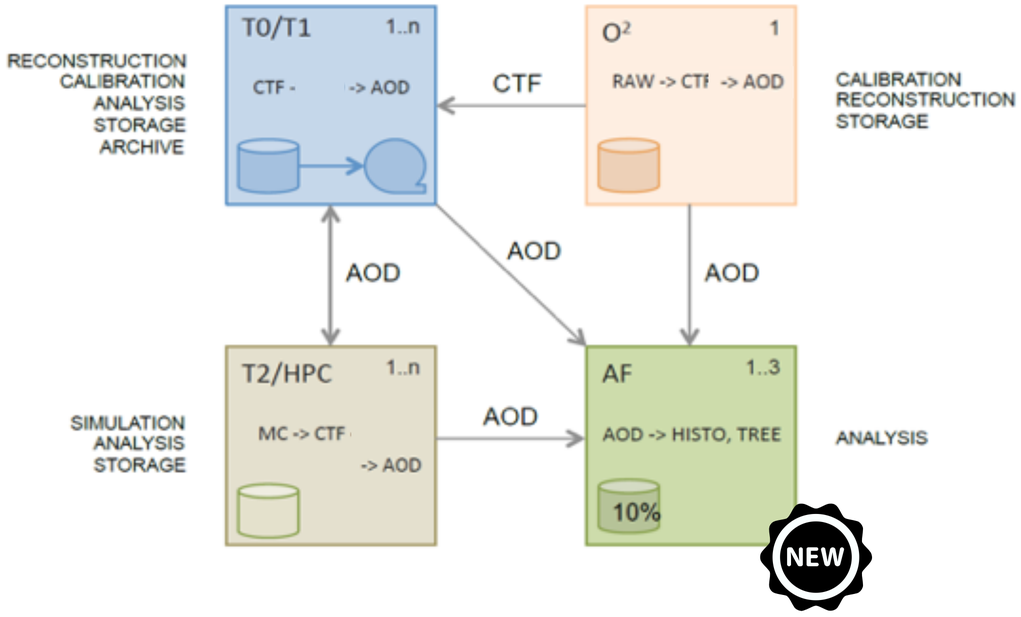
Shameless plug: Data Analysis Using ALICE Run3

Analysis Model: Run 3

Solid foundations: *the idea of organised analysis (trains) will stay. Improve on the implementation.*

- ***x100*** *more collisions compared to present setup,* ***AOD only.***
- *Initial analysis of 10% of the data at fewer* ***Analysis Facilities,*** *highly performant in terms of data access.*
- *Full analysis of a validated set of wagons on the Grid* ***⇒ Prioritise processing according to physics needs.***
- ***Streamline data model****, trade generality for speed, flatten data structures.*
- ***Recompute*** *quantities on the fly rather than storing them. CPU cycles are cheap.*
- ***Produce highly targeted ntuples*** *(in terms of information needed and selected events of interest) to reduce turnaround for some key analysis.*
- ***Goal is to have each Analysis Facility go through the equivalent of 5PB of AODs every 12 hours (~100GB/s).***





Building an Analysis Framework for the Years to Come

Homogeneity: *use the same message passing architecture which will be used for data taking to ensure homogeneity, integration and provide easy access to parallelism for the analysis tasks.*

Fast: *simplify the Analysis Data Model to achieve higher performance (e.g. via reducing I/O cost, vectorisation) for critical usecases.*

Familiar: *hide as much as possible the internal details and expose an API which provides a classic Object Oriented "feeling".*

Modern: *follow developments in ROOT and provide an easy way to access modern ROOT tools like RDataFrame.*

Open to the rest of the world: *consider integration with external analysis frameworks (e.g. Python Pandas) and ML toolkits (e.g. Tensorflow) as a requirement.*

Data Model for Analysis

Flat tables: in order to minimise the I/O cost and improve vectorisation / parallelism opportunity data will be organised in memory as column-wise collections (Tables) holding the various entities. Frontend API will still allow for nested collections but the backend will map them to a set of chunked columns.

Relational: relationships between entities are expressed in a relational manner (e.g. via indexes between tables) or as optional values (optimised via a bitmask). Frontend will still allow references, however pointers are banned from the backend.

Shared memory / message passing friendly: if we want our analysis framework to be a good citizen in the O2 world, we need the data model and the backend to be optimised for shared memory backed message passing, so that we are not hit by serialisation / deserialisation costs.

Apache Arrow: a few technical details

In-memory column oriented storage (think TTrees, but shared memory friendly). *Full description: https://arrow.apache.org/docs/memory_layout.html. Data is organized in Tables. Tables are made of Columns. Columns are (<metadata>, Array). An Array is backed by one or multiple Buffers.*

Nullable fields. *An extra bitmap can optionally be provided to tell if a given slot in a column is occupied.*

Nested types. *Usual basic types (int, float, ..). It's also possible (via the usual record shredding presented in Google's Dremel paper) to support nested types. E.g. a String is a List<Char>.*

No (generic) polymorphism. *The type in an array can be nested, but there is no polymorphisms available (can be faked via nullable fields & unions).*

Gandiva: JIT compiled, vectorised, query engine now available in upstream.

Investigating suitability for ALICE Run3 Analysis needs.

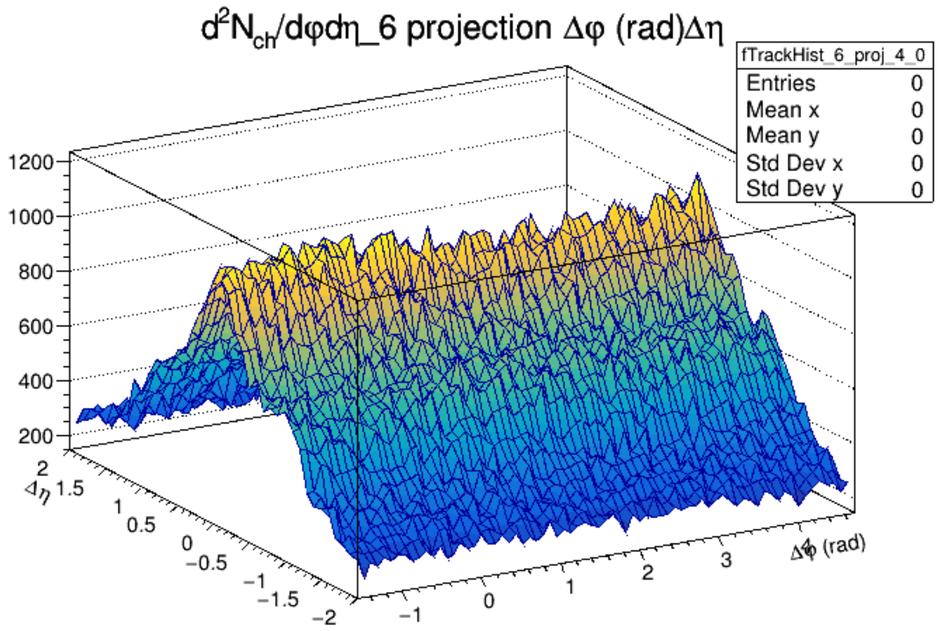
A Trivial Analysis

- Define a standalone workflow
- Define an AnalysisTask
- Define outputs, filters, partitions.
- Subscribe to the tracks for a given timeframe
- Compute (e.g.) φ from the propagation parameters
- Fill a plot

```
#include "Framework/runDataProcessing.h"
#include "Framework/AnalysisTask.h"
#include "Framework/AnalysisDataModel.h"
#include <TH1F.h>

using namespace o2;
using namespace o2::framework;

struct ATask : AnalysisTask
{
  OutputObj<TH1F> hPhi{TH1F("phi", "Phi", 100, 0, 2 * M_PI)};
  Filter ptFilter = aod::track::pt > 1;
  Partition pos = aod::track::x >= 0;

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

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

...and One Step Beyond...

```
void process(aod::Collision const& collision, aod::Tracks const& tracks)
{
  LOGF(info, "Tracks for collision: %d", tracks.size());

  for (auto it1 = tracks.begin(); it1 != tracks.end(); ++it1)
  {
    auto& track1 = *it1;
    if (track1.pt() < 0.5)
      continue;

    double eventValues[3];
    eventValues[0] = track1.pt();
    eventValues[1] = collision.v0mult();;
    eventValues[2] = collision.positionZ();

    same->GetEventHist()->Fill(eventValues, AliUEHist::kCFStepReconstructed);
    //mixed->GetEventHist()->Fill(eventValues, AliUEHist::kCFStepReconstructed);

    for (auto it2 = it1 + 1; it2 != tracks.end(); ++it2)
    {
      auto& track2 = *it2;
      if (track1 == track2)
        continue;
      if (track2.pt() < 0.5)
        continue;

      double values[6];

      values[0] = track1.eta() - track2.eta();
      values[1] = track1.pt();
      values[2] = track2.pt();
      values[3] = collision.v0mult();

      values[4] = track1.phi() - track2.phi();
      if (values[4] > 1.5 * TMath::Pi())
        values[4] -= TMath::TwoPi();
      if (values[4] < -0.5 * TMath::Pi())
        values[4] += TMath::TwoPi();

      values[5] = collision.positionZ();

      same->GetTrackHist()->Fill(values, AliUEHist::kCFStepReconstructed);
      //mixed->GetTrackHist()->Fill(values, AliUEHist::kCFStepReconstructed);
    }
  }
}
```



Courtesy of Jan-Fiete Grosse-Oetringhaus

Strategy underneath the Example

This is of course something very trivial, but it well illustrates the pursued strategy:

- **Task based API**: *reproduce run 1 and 2 analysis task concept to make transition easier. Task members are extracted to define outputs, filters, selections.*
- ***O^2* DPL underpinnings**: *this is actually an O^2 DPL workflow, heavy-lifting to map it to the message passing topology is taken care of by the framework.*
- **Type-safe**: *users subscribe to the inputs they need, in a type safe manner.* `aod::Tracks` *is a an actual type, which the DPL maps automatically to messages matching the associated Data Header.*
- **Arrow Skins**: *users are exposed to a familiar Imperative / "Object Oriented" API to access physics objects. In reality they act on an Apache Arrow backed* ***AoSoA*** *data store, on top of which the Framework allows to construct "Skins". Similar to LHCb SOAContainer or CMS FWCore/SOA.*
- **Generic**: *the signature of the* ***process*** *method is what drives the subscription to data (via template magic). E.g. to get all the tracks associated to a given collision:*

```
void process(aod::Collision& collision, aod::Tracks const& tracks)
```

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.

What about RDataFrame?

RArrowDS:

ALICE donated to ROOT a datasource allowing integration of Arrow and RDataframe.

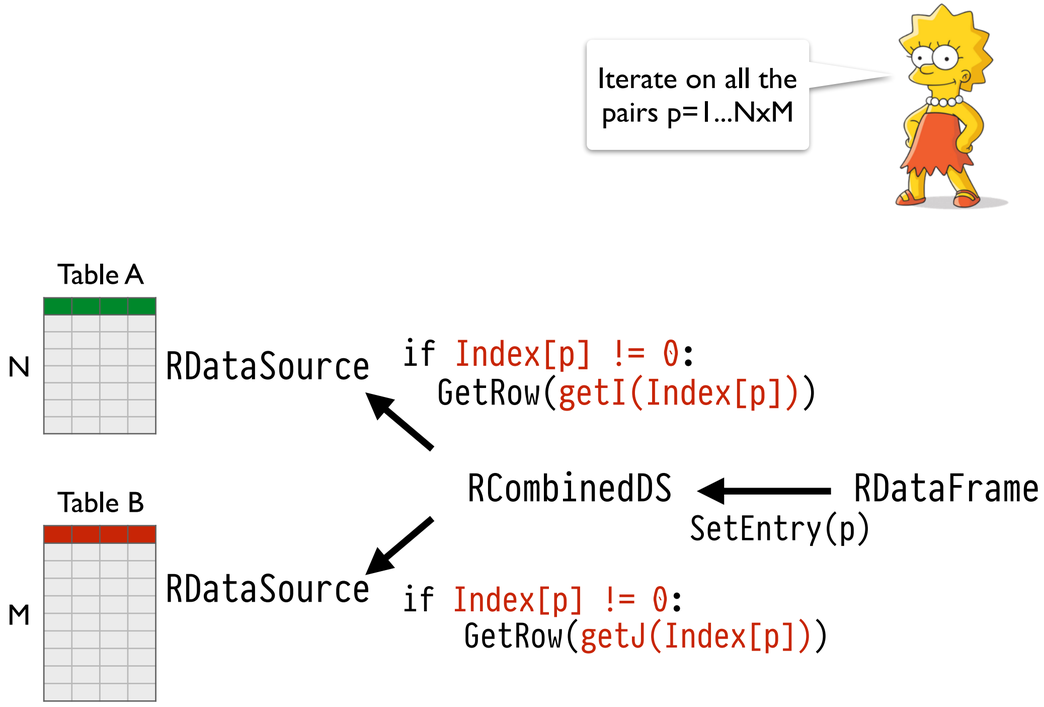
```
auto source = std::make_unique<RArrowDS>(tracks.asArrowTable(), std::vector<std::string>{});
RDataFrame rdf(std::move(source));
```

RCombinedDS:

We are investigating using a special RDataSource to combine two or more tables (or one with itself), effectively allowing double loops and generic JOIN operations within an RDataFrame. Challenge is to limit how many entries we keep in memory e.g. for event mixing.

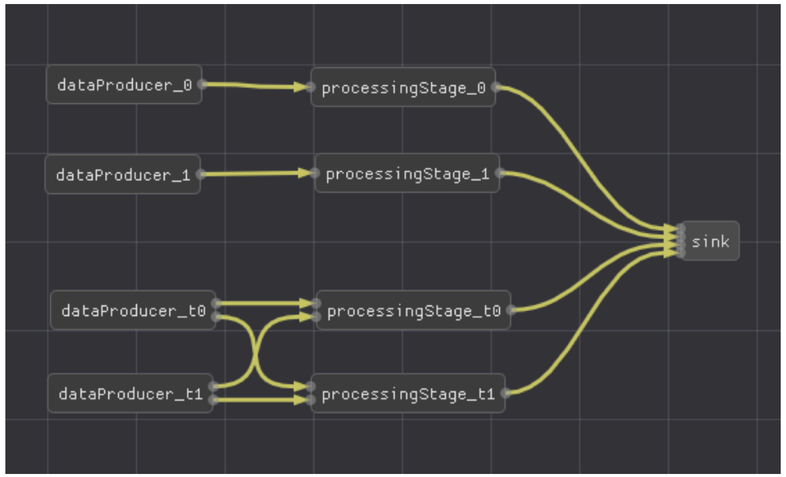
RDataFrame helpers:

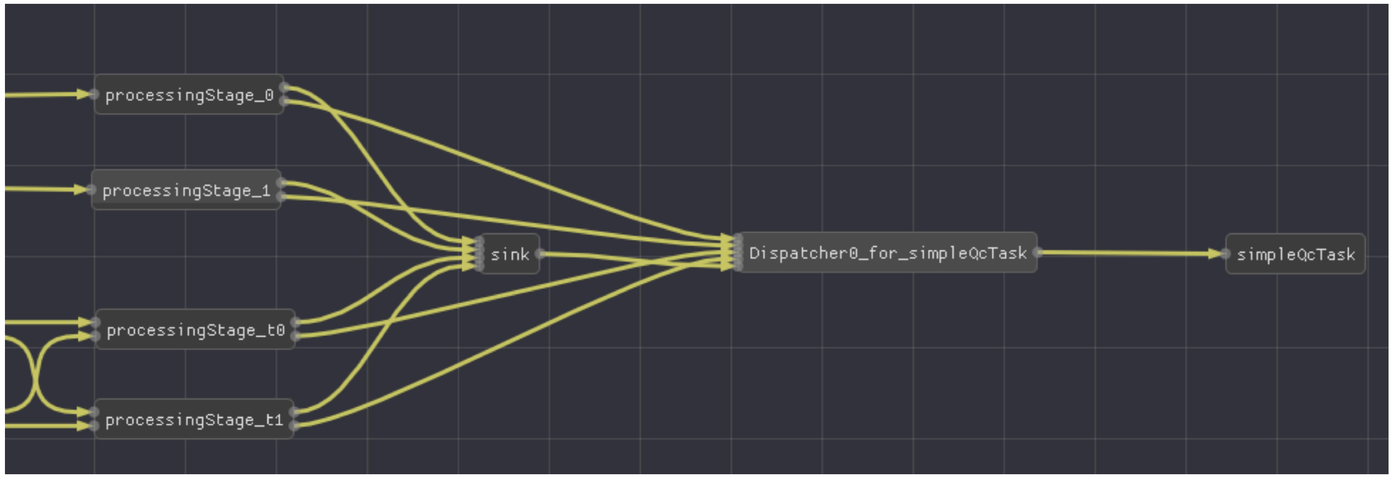
We have developed a few helpers which simplify the creation of RCombinedDS hierarchies for common physics usecases.

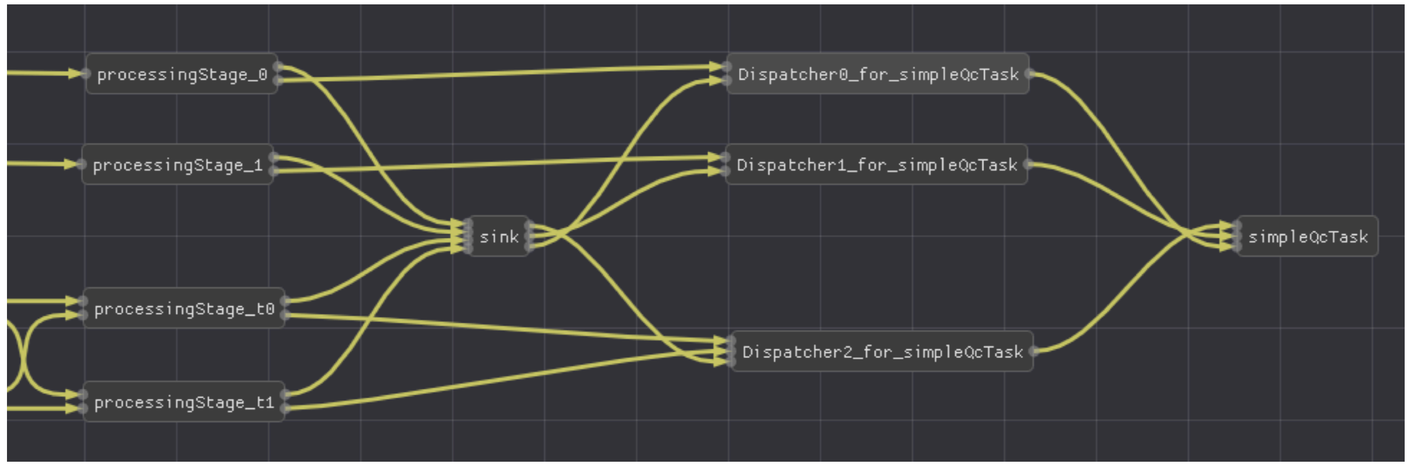


COMPOSABLE WORKFLOWS

Declarative configuration allows for easy customisation: e.g. adding a (one or more) dispatchers for QA.







Workflows are executables. Piping on the shell multiple executables builds the closure workflow.

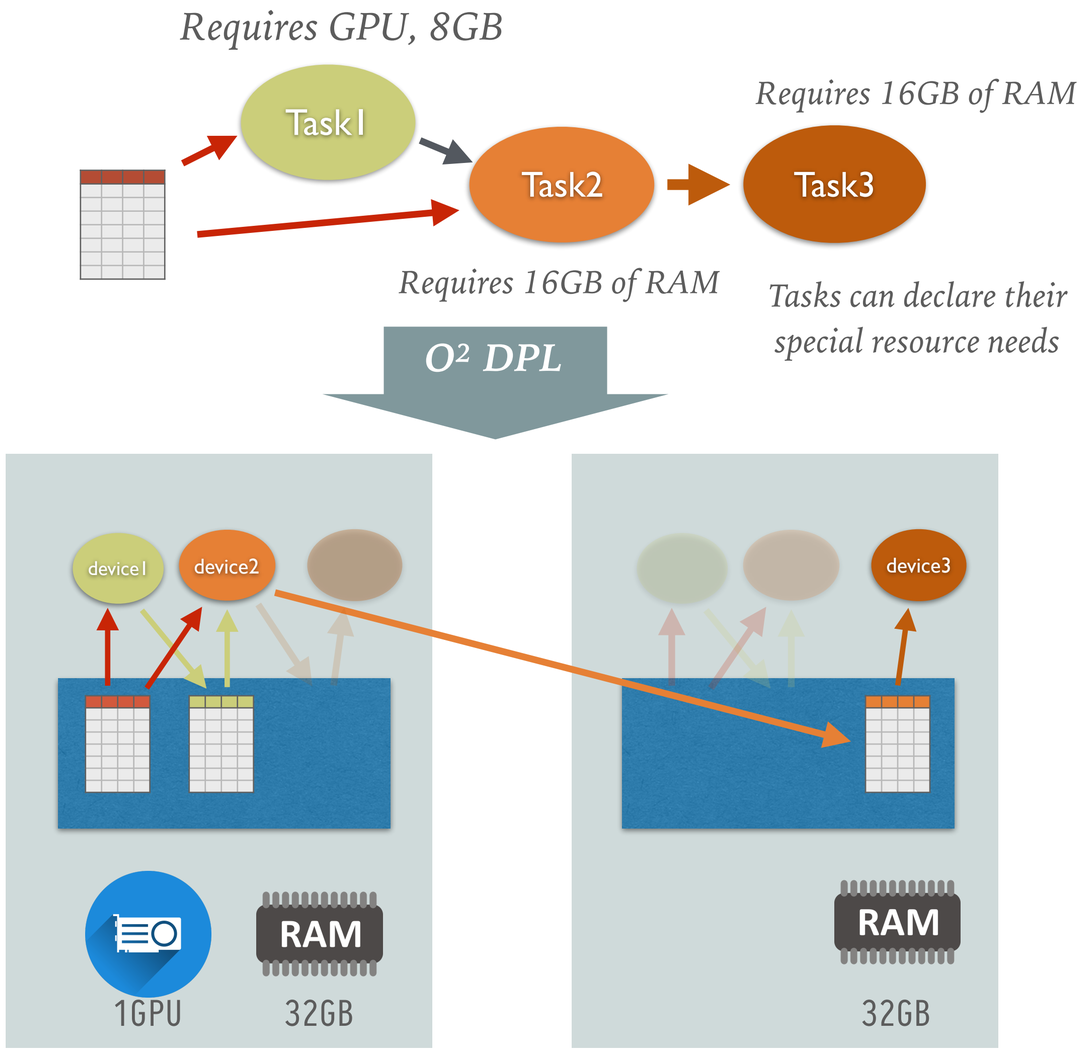
reconstruction-workflow | qc-workflow

Heterogeneous Computing Support

The mapping of an analysis workflow on top of a topology of message passing entities has the advantage to fit well physically / logically heterogeneous architectures.

Simple Multi Node support: *the current code can in particular already take advantage of multi-node setups (e.g. using Kubernetes ReplicaSet), without the need of an additional orchestrator entity. Each Replica knows the full topology and uses the same deterministic resource scheduling algorithm, resulting in seamless deployments for a low number of distinct nodes.*

Asymmetric nodes: *we are exploring using the same approach to model logically separated resources like GPU or NUMA.*



Resources can be either physically separated, or logically different domains within the same box.

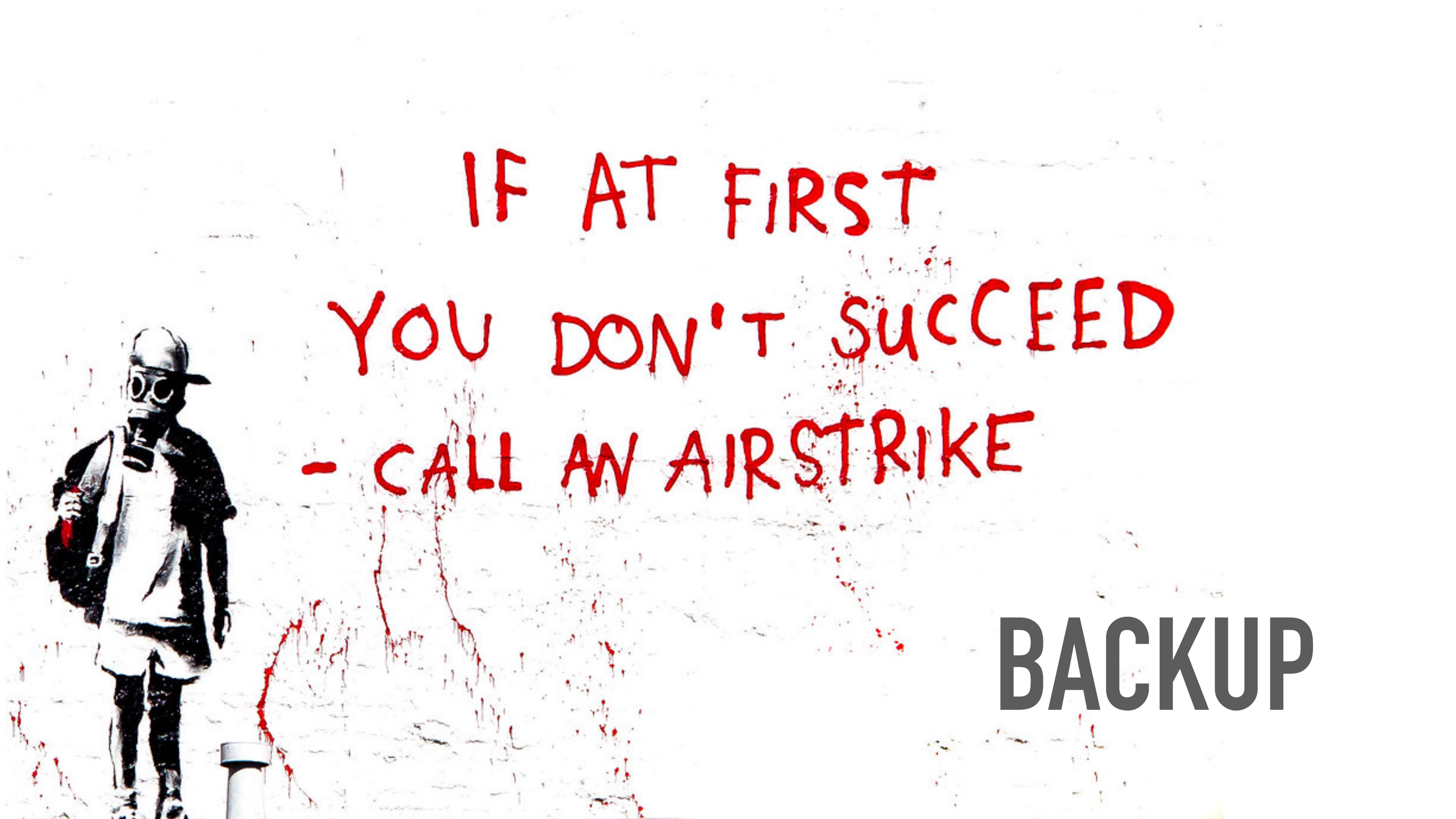
ASK NOT WHAT HSF CAN DO FOR YOU—ASK WHAT YOU CAN DO FOR HSF

Topics about which we are very happy to discuss / participate:

- How to evolve RDataFrame so that it can be the natural choice for actual HI analysis. (See my RCombinedDS pull request).
- How to better integrate our frameworks and tools with those which build on top / support Apache Arrow, e.g. NVidia Rapids, Tensorflow.
- How do we exploit RNtuple and next gen ROOT I/O in our framework.
- Everyone seems to have their own (Ao)SoA implementation, we have one based on Apache Arrow and Gandiva, is there room for a framework agnostic implementation which satisfies everyone?
- How to best deploy to Kubernetes / PMIx enabled facilities.
- News in the field of Message Passing / Actor Model frameworks.

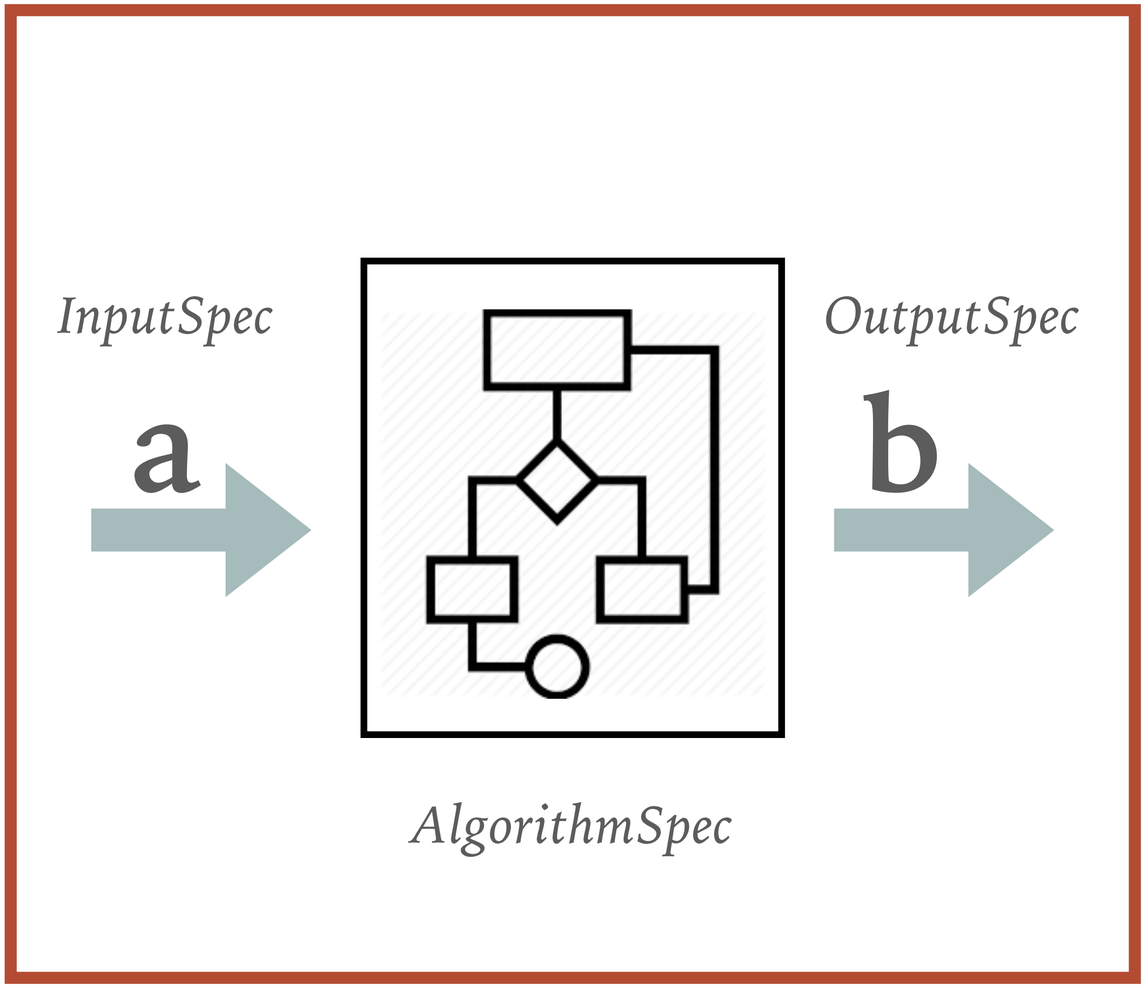
GEEKY STUFF AND SHAMELESS PLUGS

- The GUIs you have seen are based on https://github.com/ocornut/imgui of , we have our own simple wrapper at https://github.com/AliceO2Group/DebugGUI.
- We have some support for DTrace / SystemTap / Signpost probes and some custom Apple Instruments plugin. We are happy to discuss that as well.
- If anyone is interested, we will split off in a standalone component our own Arrow based (Ao)SoA implementation soon.



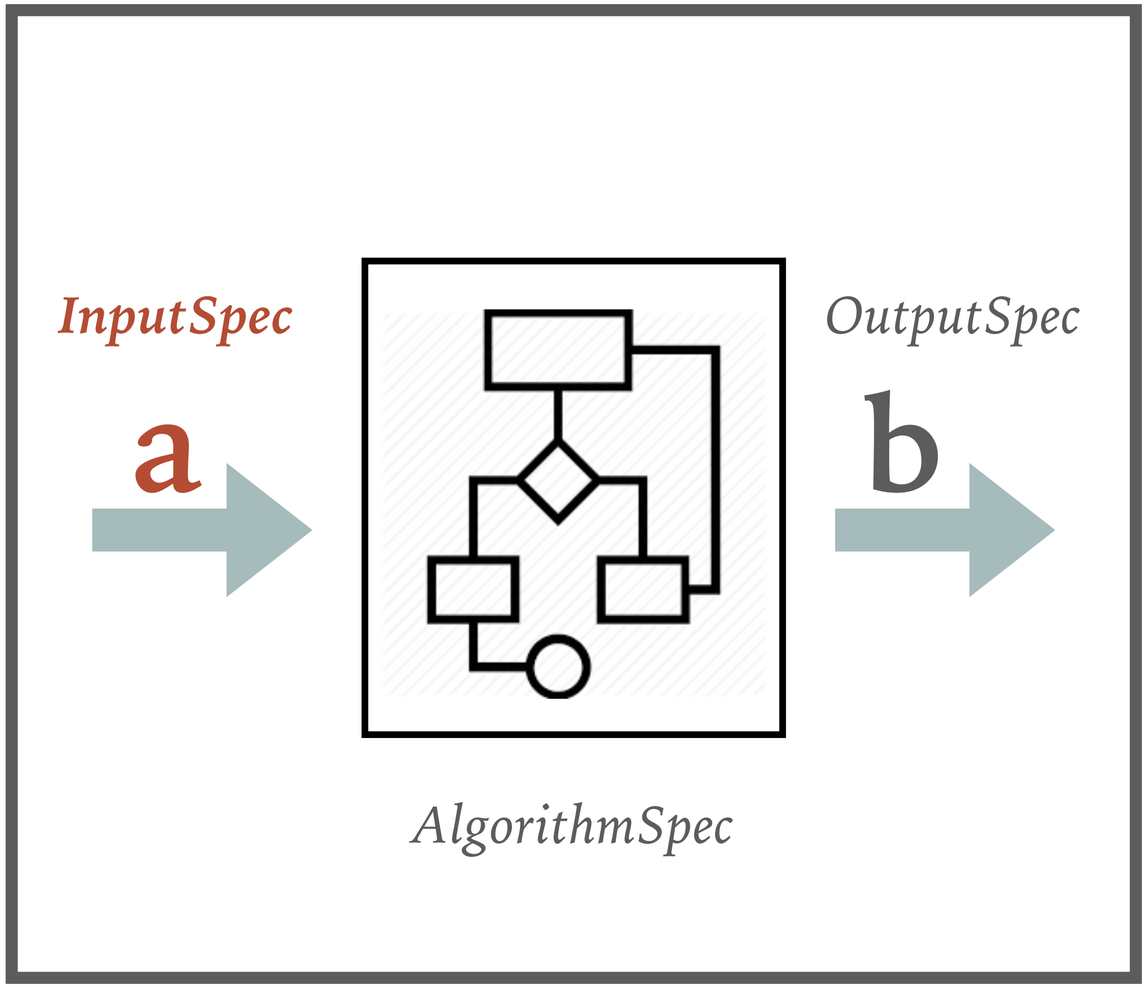
BACKUP

DATA PROCESSING LAYER: HOW



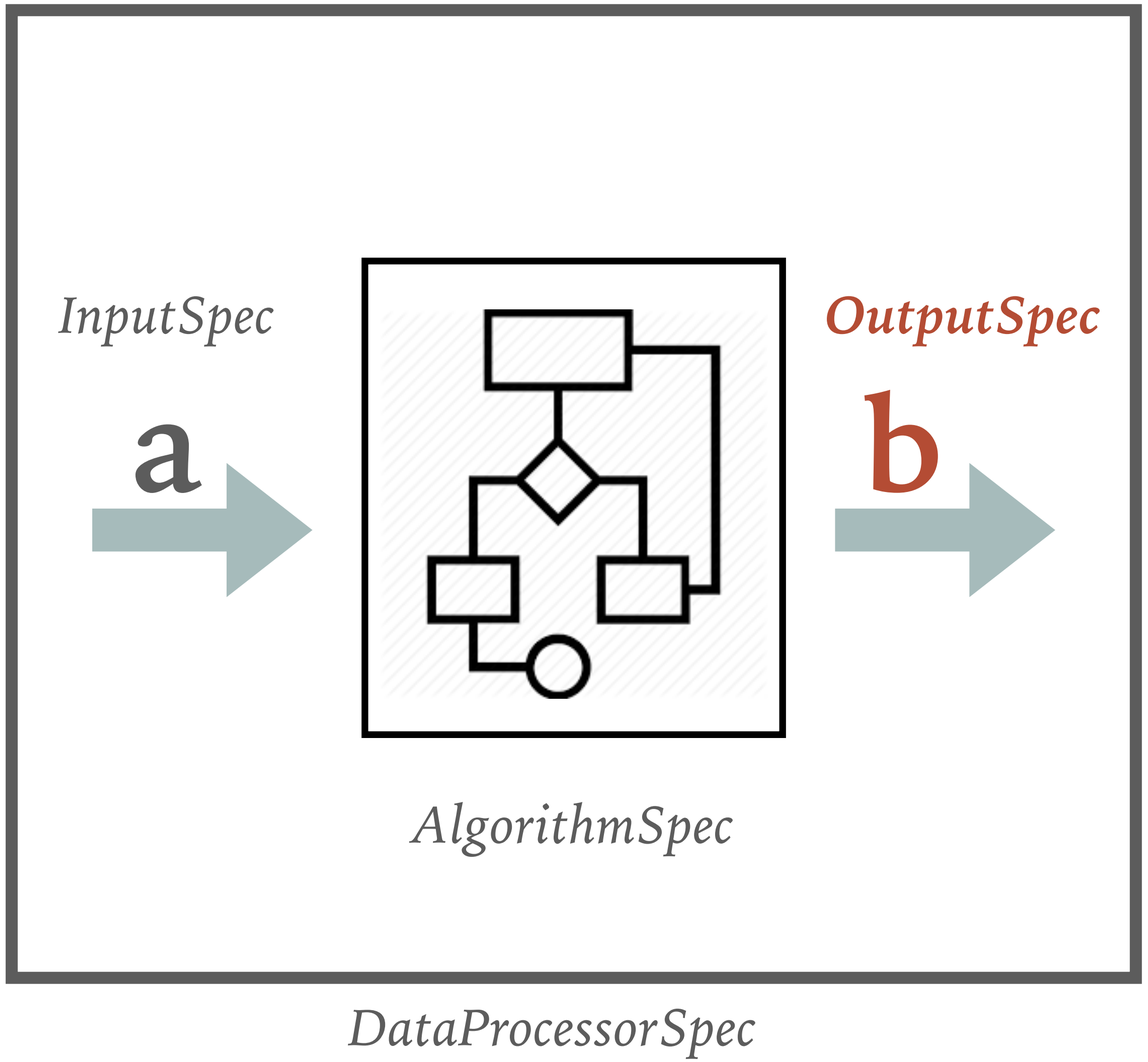

```
DataProcessorSpec{
  "A",
  Inputs{
   InputSpec{"a", "TPC", "CLUSTERS"}
  },
  Outputs{
   OutputSpec{{"b"}, "TPC", "TRACKS"}
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
       auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
```

DATA PROCESSING LAYER: HOW



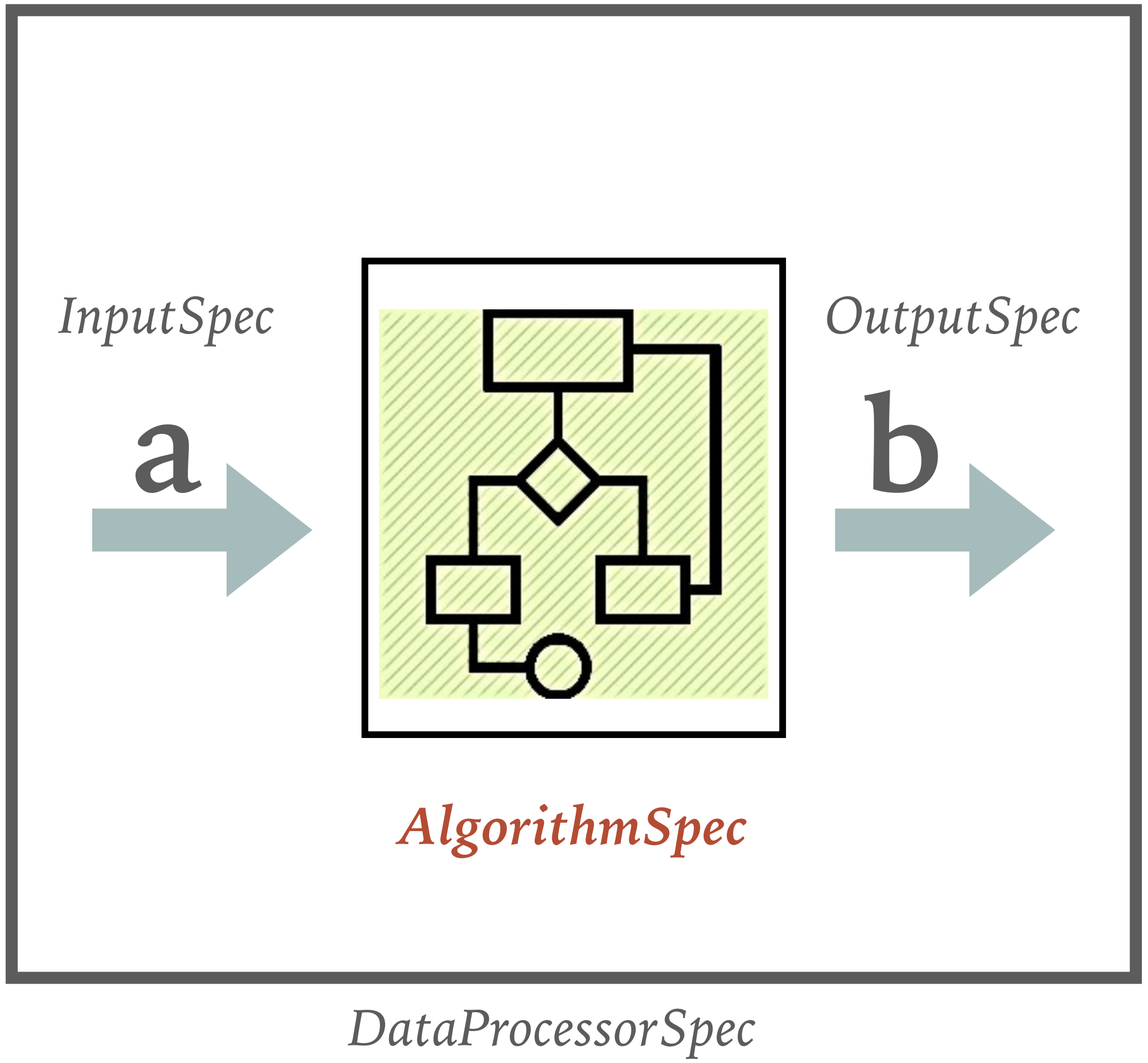

```
DataProcessorSpec{
  "A",
  Inputs{
   InputSpec{"a", "TPC", "CLUSTERS"}
  },
  Outputs{
   OutputSpec{{"b"}, "TPC", "TRACKS"}
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
       auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
```

DATA PROCESSING LAYER: HOW




```
DataProcessorSpec{
  "A",
  Inputs{
   InputSpec{"a", "TPC", "CLUSTERS"}
  },
  Outputs{
   OutputSpec{{"b"}, "TPC", "TRACKS"}
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
       auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
```

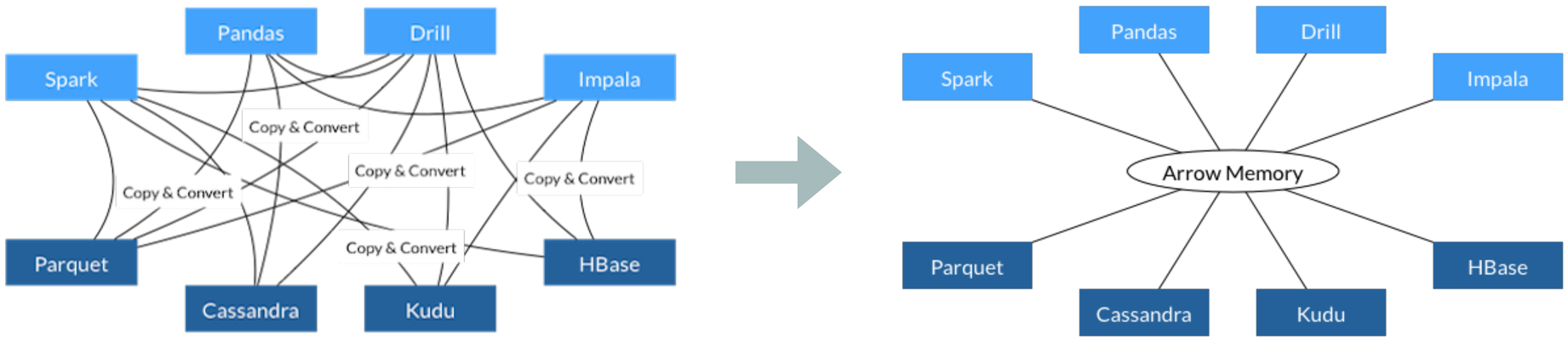
DATA PROCESSING LAYER: HOW




```
DataProcessorSpec{
  "A",
  Inputs{
   InputSpec{"a", "TPC", "CLUSTERS"}
  },
  Outputs{
   OutputSpec{{"b"}, "TPC", "TRACKS"}
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
       auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
```

Apache Arrow: a possible solution for In-Memory Columnar Format

"Cross-language development platform for in-memory columnar data."



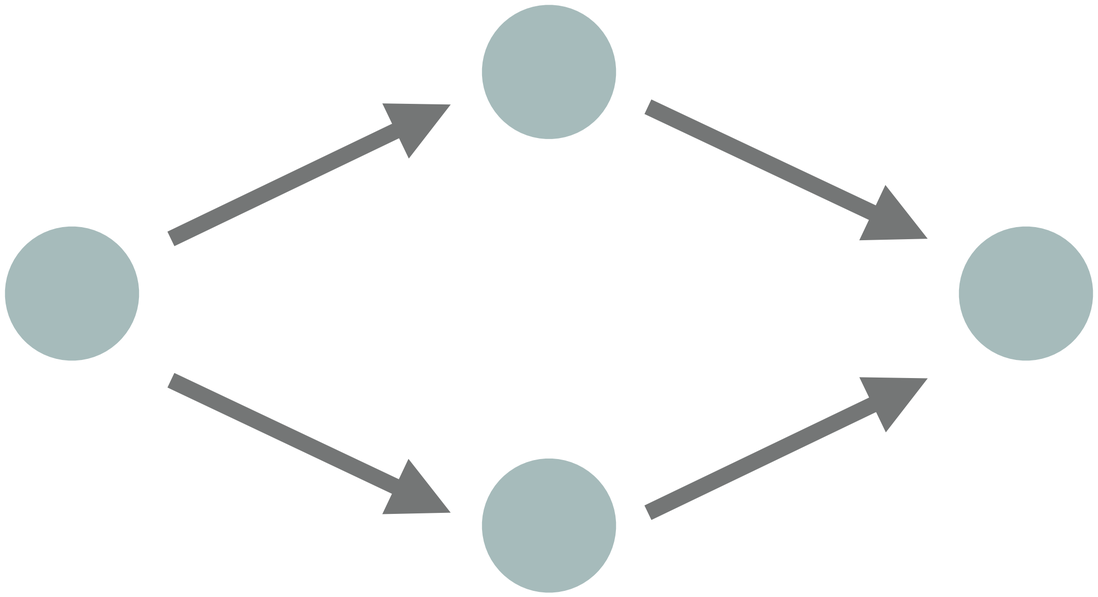
Well established. *Top-Level Apache project backed by key developers of a number of opensource projects: Calcite, Cassandra, Drill, Hadoop, HBase, Ibis, Impala, Kudu,* ***Pandas****,* ***Parquet****, Phoenix,* ***Spark****, and Storm,* ***Tensorflow****.*

Very active. *298 contributors, https://github.com/apache/arrow*

O2 design friendly. *Message passing / shared memory friendly. Support for zero-copy slicing, filtering.*

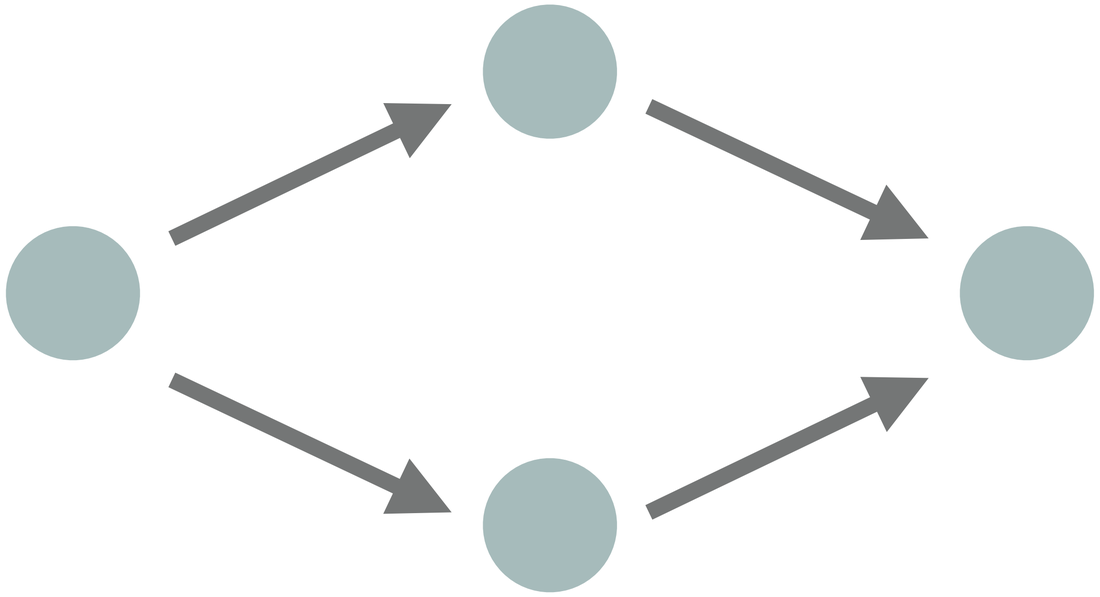
Multinode Support

o2-diamond-worklow

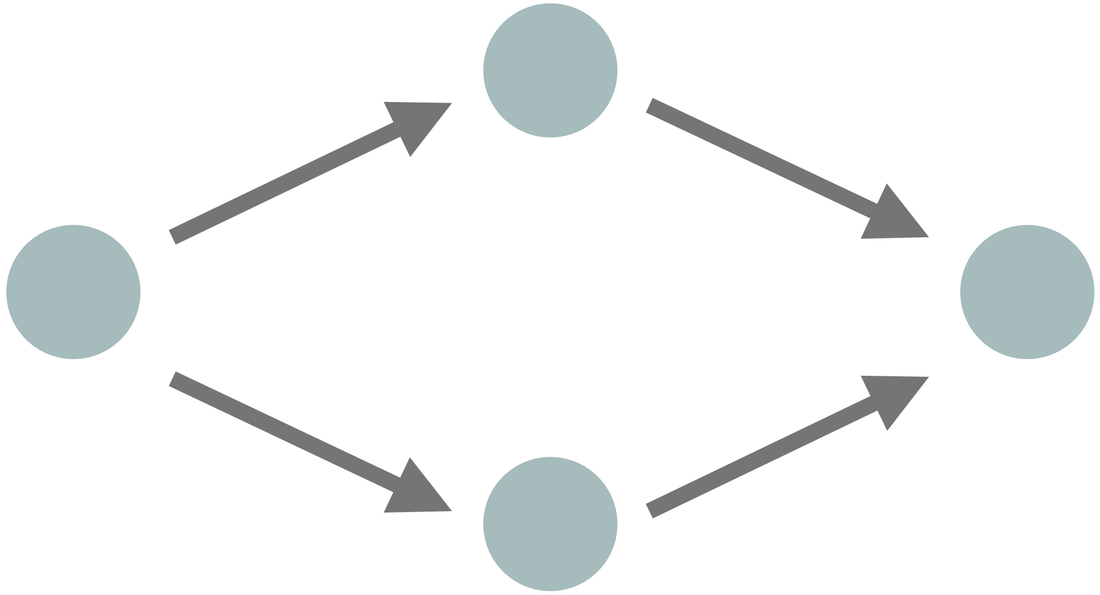


Multinode Support

```
o2-diamond-worklow --hostname hostname1 \
                   --resources hostname1:1:8000,hostname2:1:8000
```



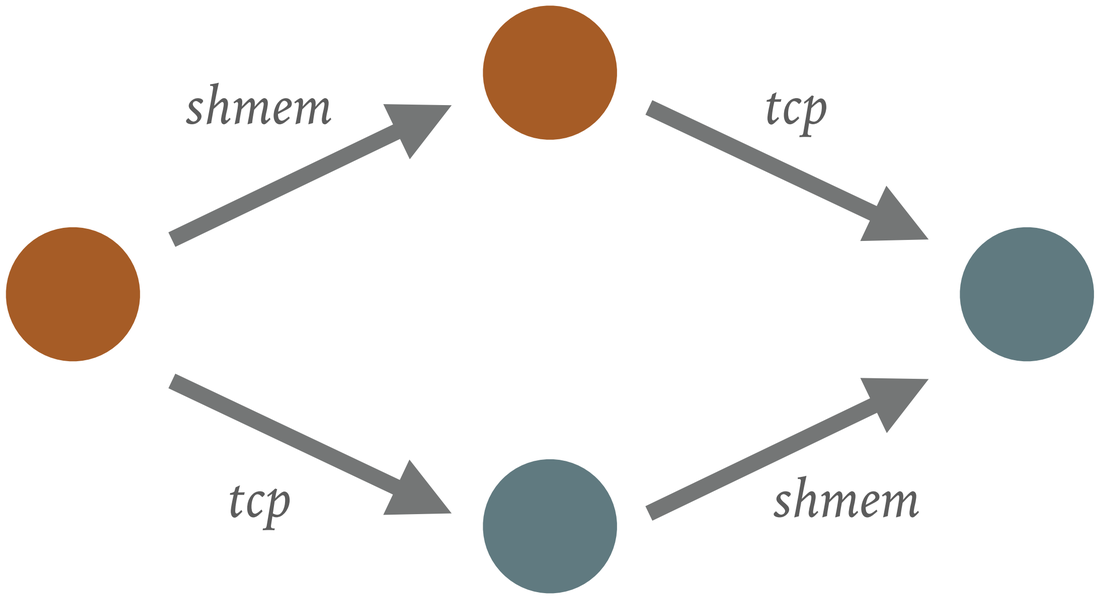
```
o2-diamond-worklow --hostname hostname2 \
                   --resources hostname1:1:8000,hostname2:1:8000
```



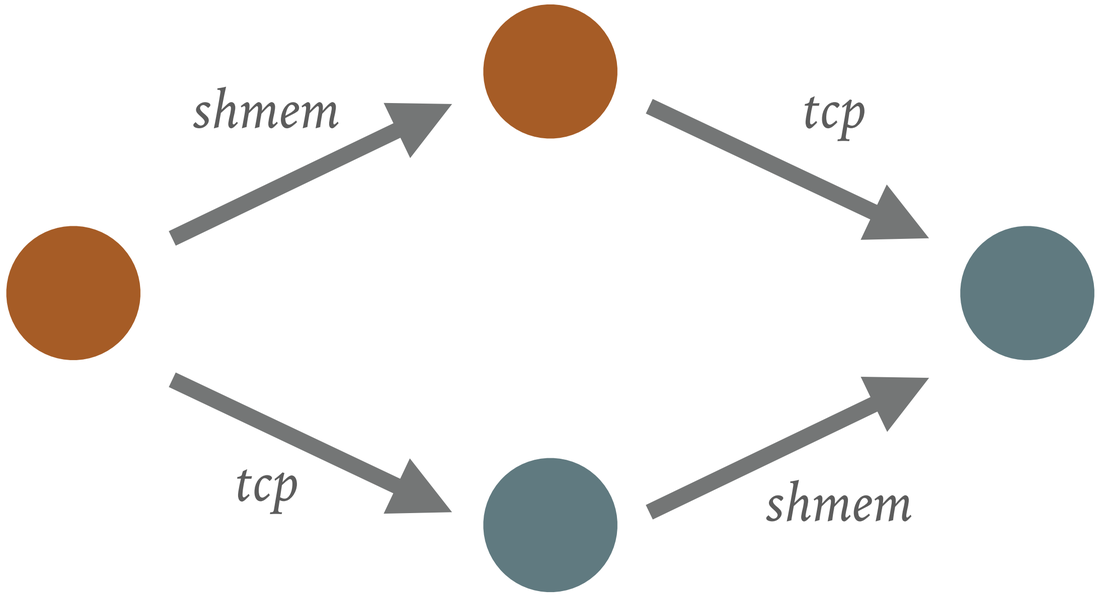
The topological sort builds the same topology from the workflow on both nodes.

Multinode Support

```
o2-diamond-worklow --hostname hostname1 \
                   --resources hostname1:1:8000,hostname2:1:8000
```



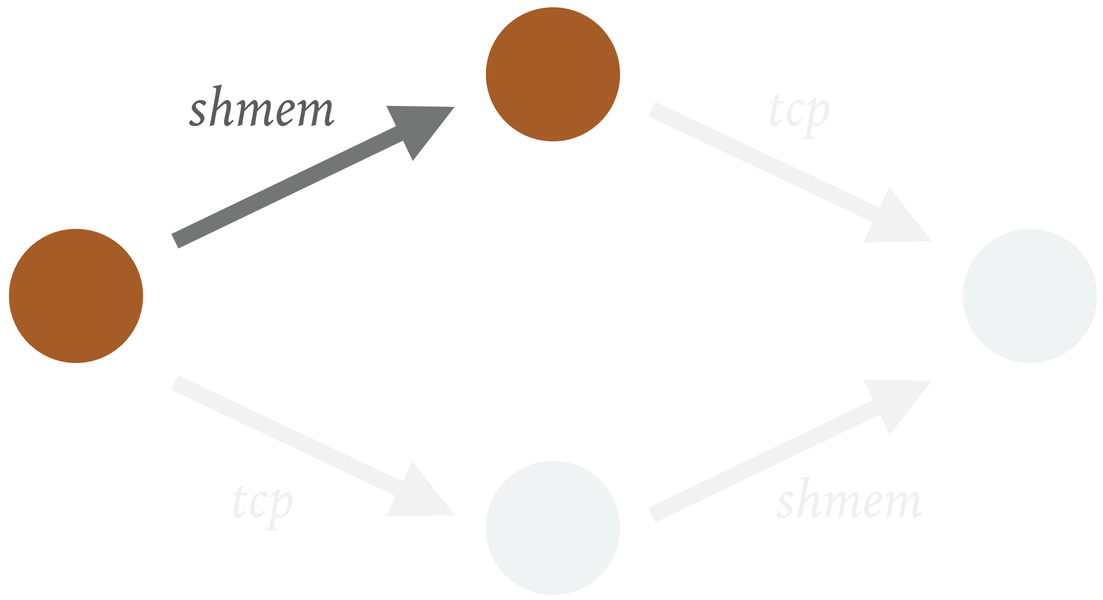

```
o2-diamond-worklow --hostname hostname2 \
                   --resources hostname1:1:8000,hostname2:1:8000
```



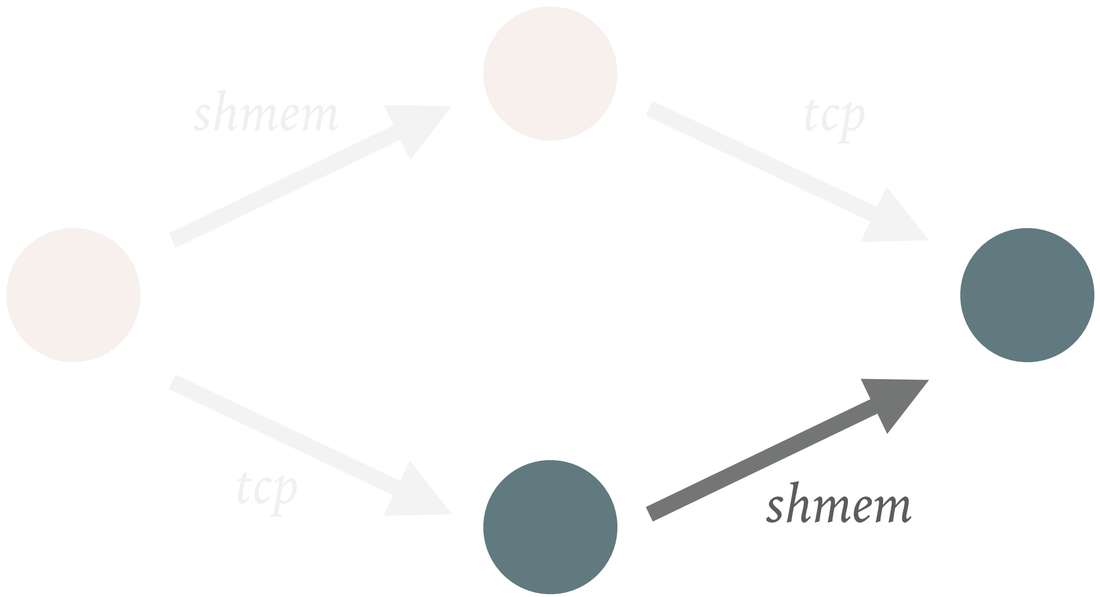
The resource mapping algorithm is stable so each nodes knows what should run where and what backend should be used.

Multinode Support

```
o2-diamond-worklow --hostname hostname1 \
                   --resources hostname1:1:8000,hostname2:1:8000
```




```
o2-diamond-worklow --hostname hostname2 \
                   --resources hostname1:1:8000,hostname2:1:8000
```

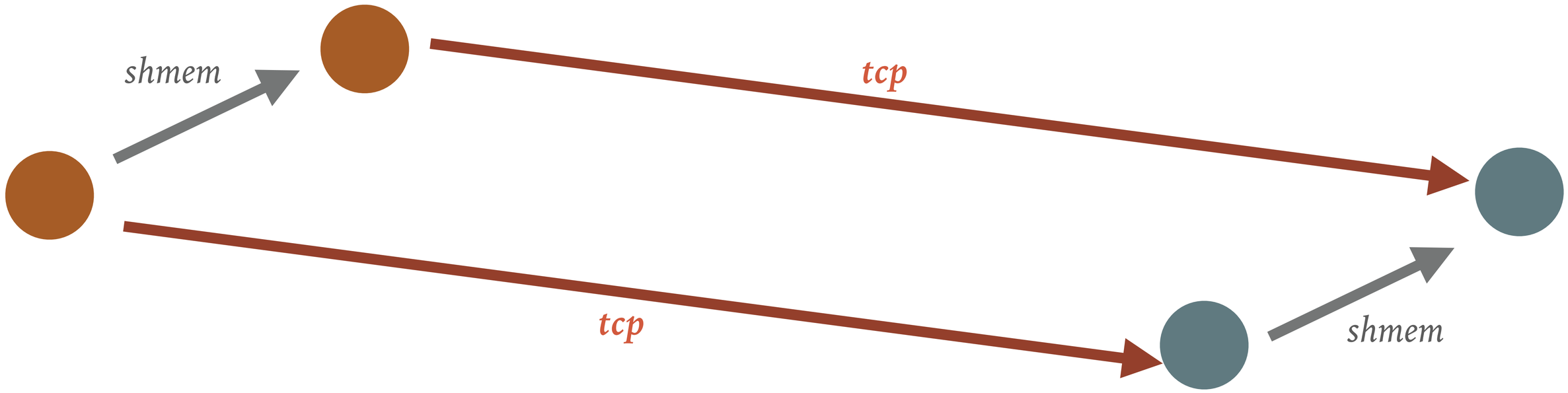


Each node runs only its part of the configuration

Multinode Support

```
o2-diamond-worklow --hostname hostname1 \
                   --resources hostname1:1:8000,hostname2:1:8000
```

```
o2-diamond-worklow --hostname hostname2 \
                   --resources hostname1:1:8000,hostname2:1:8000
```



...and because the resource mapping is stable and deterministic, each node knows where to find their external connections.

Single loop with RDataFrame

Get an RDataFrame iterating on candidates obtained via O2

```
auto candidates = o2::analysis::doSingleLoopOn(input);
```

Select Good candidates

```
auto filtered = candidates.Filter("cand_type & 1");
```

Fill an histogram

```
auto h1 = filtered.Histo1D("inv_mass");
```

Draw it

```
h1->Draw();
```

Single loop with RDataFrame

Get an RDataFrame iterating on candidates obtained via O2

```
auto candidates = o2::analysis::doSingleLoopOn(input);
```

Select Good candidates

```
auto filtered = candidates.Filter("cand_type & 1");
```

Fill an histogram

```
auto h1 = filtered.Histo1D("inv_mass");
```

Draw it

```
h1->Draw();
```

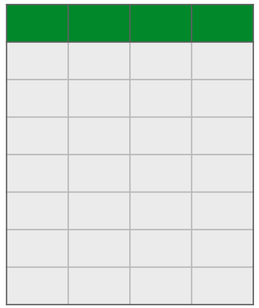
Event loop actually runs here.

RDataFrame internals

Iterate on all the items 1...N

```
table.GetRow(i)
```

```
for i in 1..N:
  datasource.SetEntry(i)
```



RDataSource ← RDataFrame ←

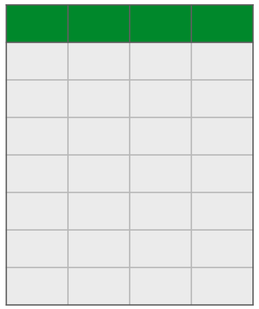
The user does not see the internals. All the gymnastic is hidden inside a framework provided helper function

```
auto candidates = o2::analysis::doSingleLoopOn(input);
```

Properly setup RDataFrame

Input subscribed from DPL

RDataFrame internals



Actual table stored in memory / disk / created on the fly. The actual backend will depend on the context.

`RDataSource`

RDataSource abstracts from the user the act of fetching row from a table. Different implementations depending on the backend.

`RDataFrame`

RDataFrame abstracts from the user how to iterate on a set of rows and how actions are performed on them.

Role of the analysis framework: provide helpers to construct useful views on the data, using the above building blocks.

Double loop with RDataFrame

Get an RDataFrame iterating on the possible **pairs** of candidates with the **same evtID**

```
auto pairs = o2::analysis::doSelfCombinationsWith(input, "d0", "evtID");
```

Select Good candidates

```
auto filtered = pairs.Filter("(d0_cand_type & 1) && (d0bar_cand_type & 1)");
```

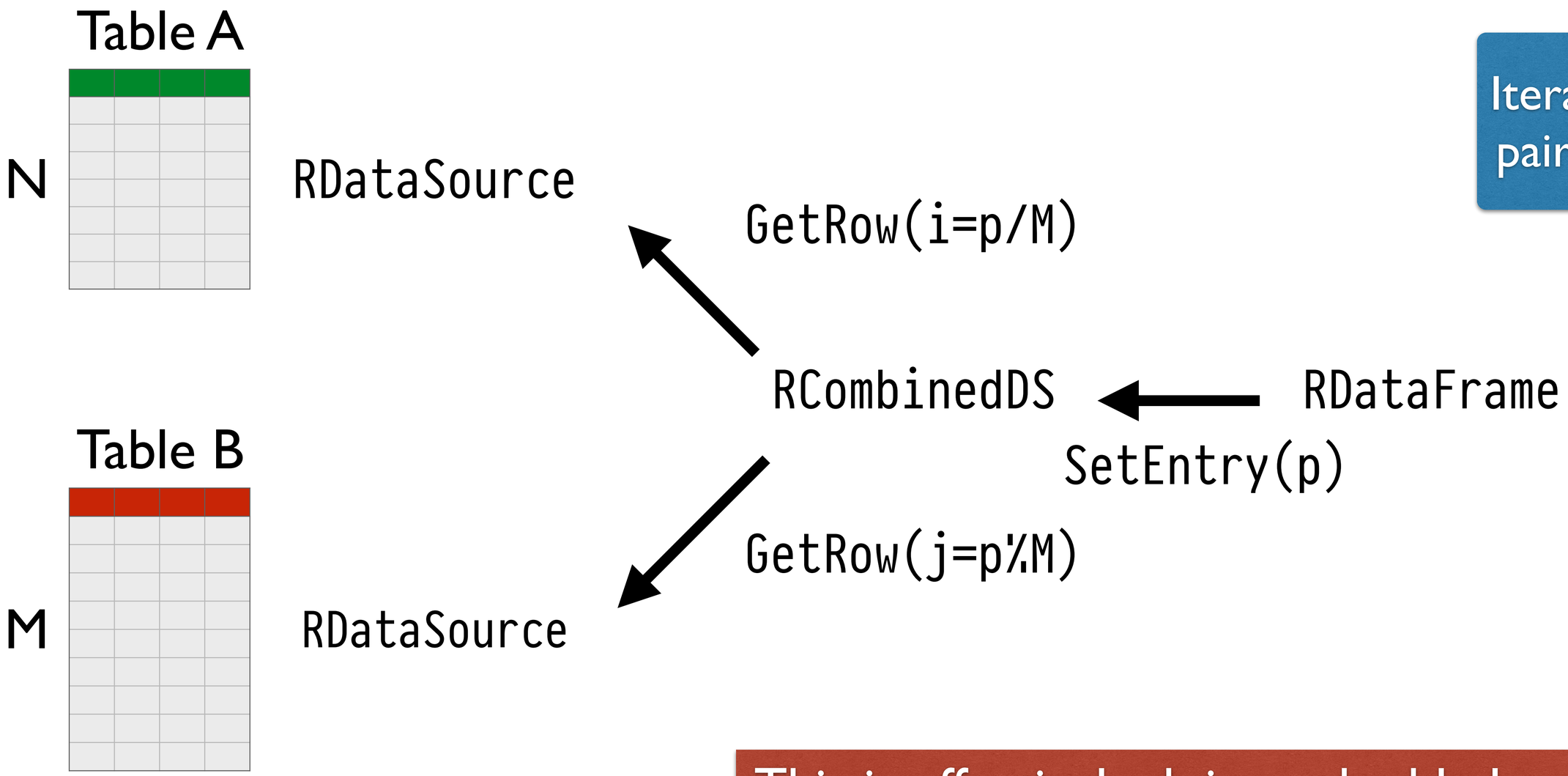
Define extra variables

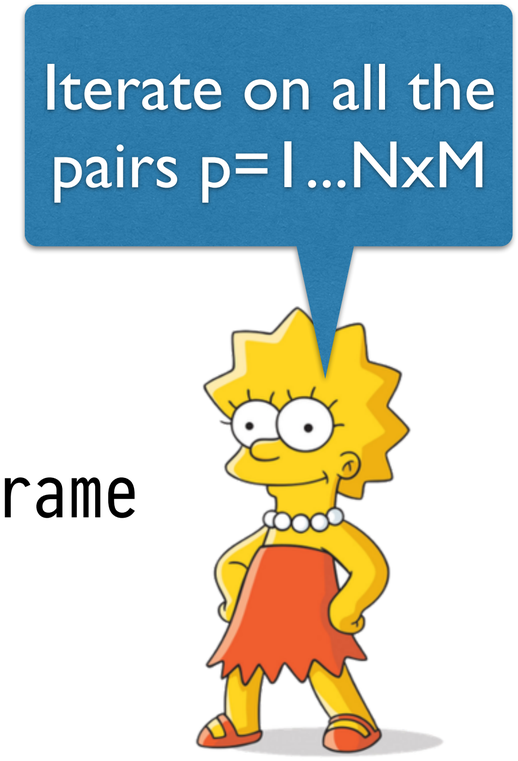
```
auto deltas = filtered.Define("Dphi", "d0_phi_cand - d0bar_phi_cand")
                      .Define("Deta", "d0_eta_cand - d0bar_eta_cand");
```

Create your histograms

```
auto h2 = deltas.Histo1D("Dphi");
auto h3 = deltas.Histo1D("Deta");
```

RCombinedDS: combining multiple datasources



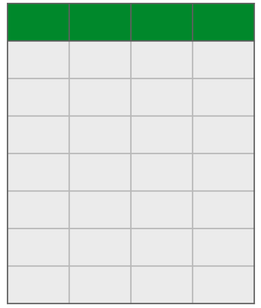


This is effectively doing a double loop on all the possible row pairs of the table A and B.

RCombinedDS: generalisation

Table A

N

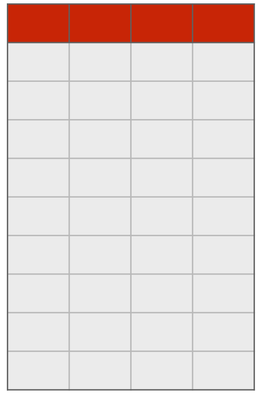


RDataSource

```
if Index[p] != 0:
  GetRow(getI(Index[p]))
```

Table B

M



RDataSource

```
if Index[p] != 0:
  GetRow(getJ(Index[p]))
```

RCombinedDS

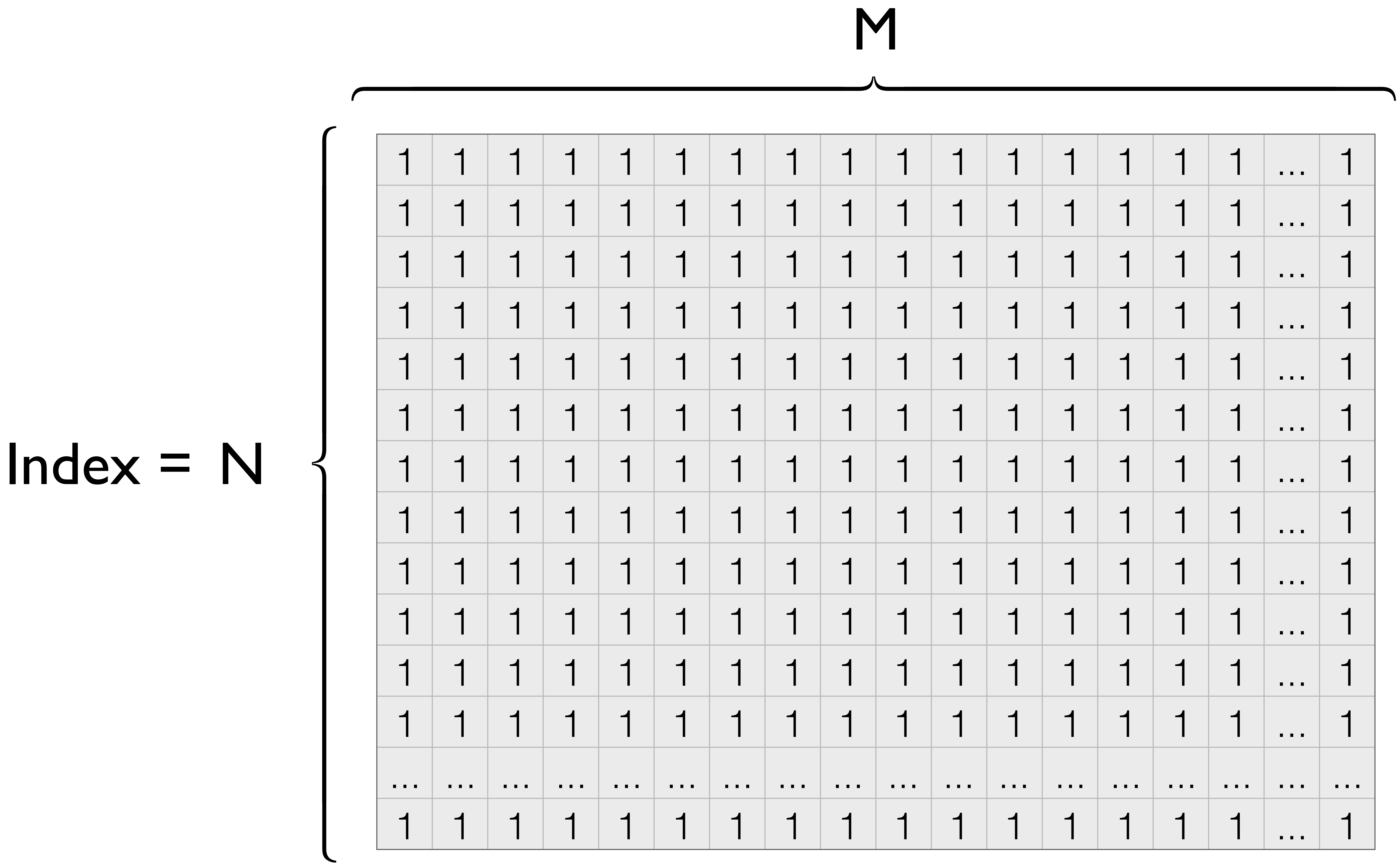
RDataFrame

SetEntry(p)

Iterate on all the pairs p=1...NxM

We can generalise the mechanism by having an index to select the pairs to yield.

RCombinedDS: full double loop

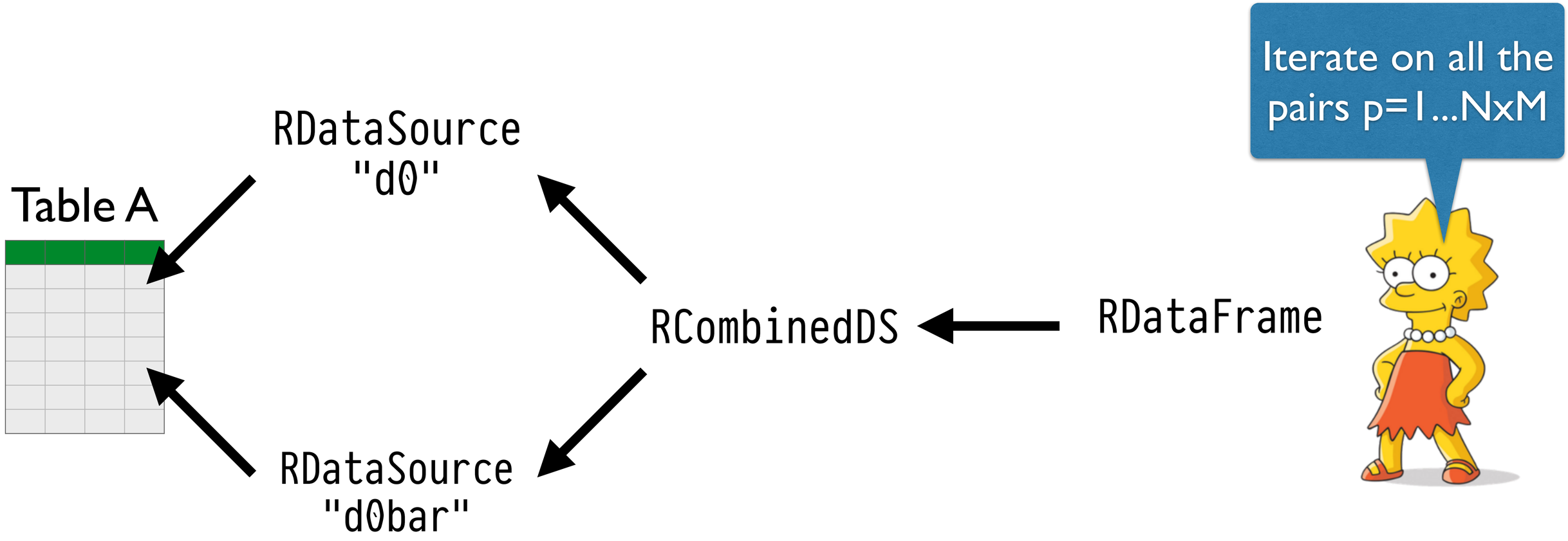


M

Index = N

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1

RCombinedDS: same source



Two RDataSource can actually point to the same table.

RCombinedDS: double loop without repetitions

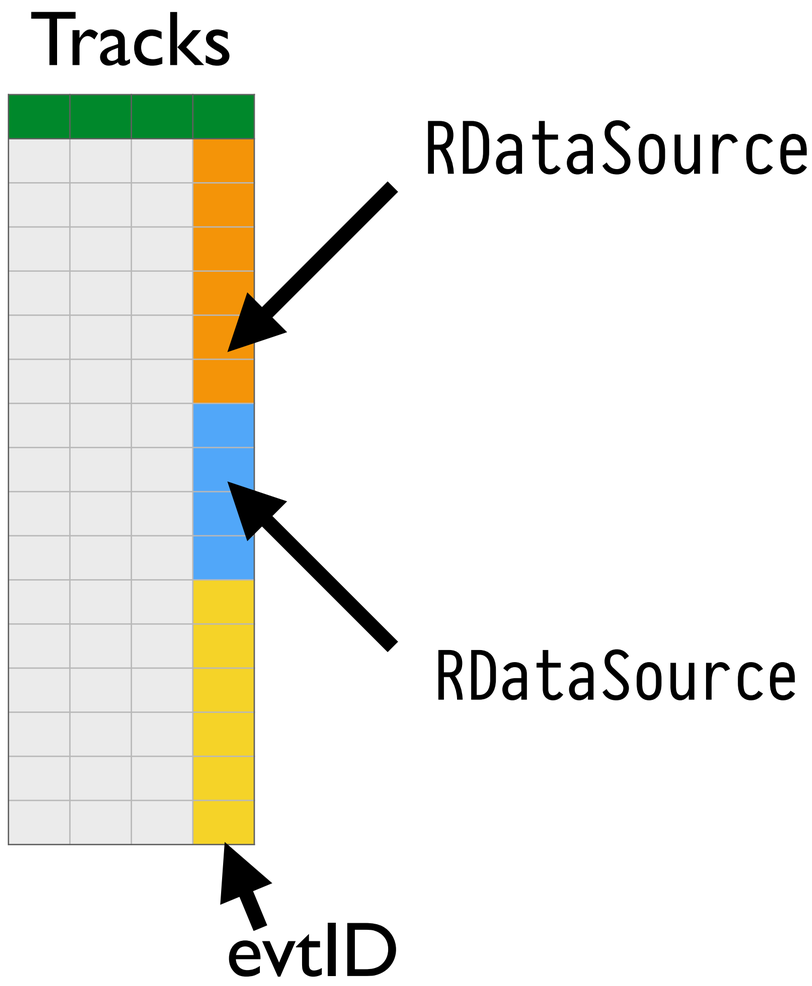
N

Index = N

0	1	1	1	1	1	1	1	1	1	1	1	...	1
0	0	1	1	1	1	1	1	1	1	1	1	...	1
0	0	0	1	1	1	1	1	1	1	1	1	...	1
0	0	0	0	1	1	1	1	1	1	1	1	...	1
0	0	0	0	0	1	1	1	1	1	1	1	...	1
0	0	0	0	0	0	1	1	1	1	1	1	...	1
0	0	0	0	0	0	0	1	1	1	1	1	...	1
0	0	0	0	0	0	0	0	1	1	1	1	...	1
0	0	0	0	0	0	0	0	0	1	1	1	...	1
0	0	0	0	0	0	0	0	0	0	1	1	...	1
0	0	0	0	0	0	0	0	0	0	0	1	...	1
0	0	0	0	0	0	0	0	0	0	0	0	...	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	0	0	0	0	0	0	0	...	0

A strictly upper triangular square matrix index can represent all the possible pairs of rows within the same table, avoiding repetitions.

RCombinedDS: generalisation

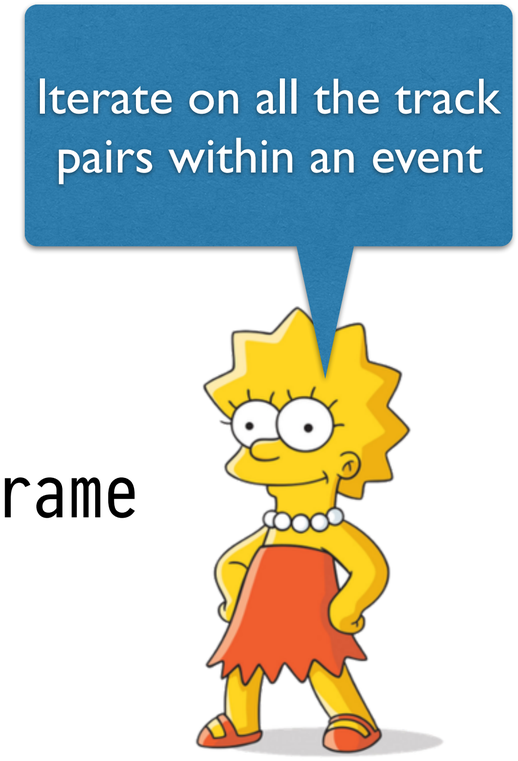


RDataSource

RDataSource

RCombinedDS

RDataFrame



A column can define categories of rows, e.g. the event id.

RCombinedDS: double with loop within a category

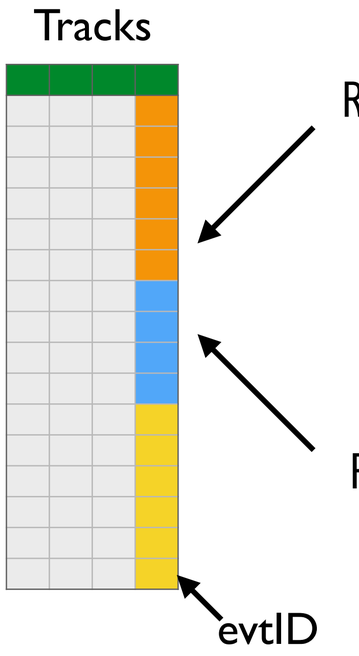
N

Index = N

0	1	1	1	1	1	0	0	0	0	0	0	...	0
0	0	1	1	1	1	0	0	0	0	0	0	...	0
0	0	0	1	1	1	0	0	0	0	0	0	...	0
0	0	0	0	1	1	0	0	0	0	0	0	...	0
0	0	0	0	0	1	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	1	1	1	0	0	...	0
0	0	0	0	0	0	0	0	1	1	0	0	...	0
0	0	0	0	0	0	0	0	0	1	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	1	...	1
0	0	0	0	0	0	0	0	0	0	0	0	...	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	0	0	0	0	0	0	0	...	0

An index (similar to) a block diagonal matrix represents combinations within the same category. Most natural category is the event, but RCombinedDS is not limited to that (e.g. for Event Mixing)

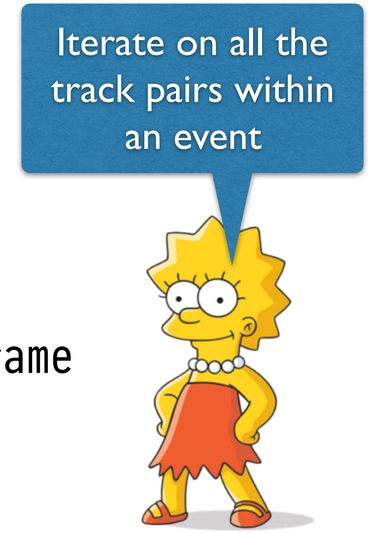
RCombinedDS: helper functions



RDataSource

RDataSource

RCombinedDS ← RDataFrame



The user does not see the internals. All the gymnastic is hidden inside a framework provides helper function

```
auto pairs = o2::analysis::doSelfCombinationsWith(input, "d0", "evtID");
```

Properly setup RDataFrame

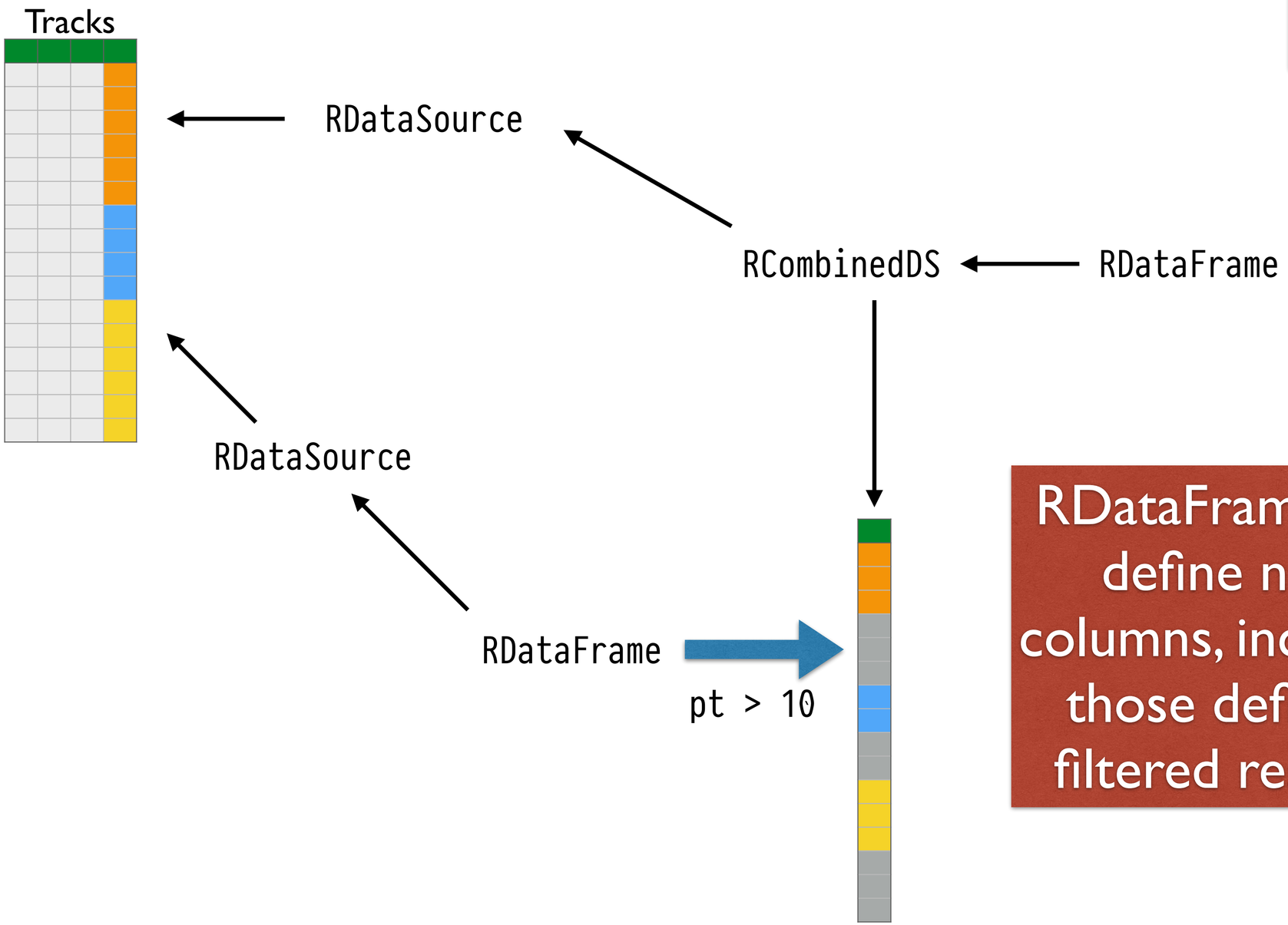
Input subscribed from DPL

mnemonic for the table

Column to use for category

RCombinedDS: filtered collections

Iterate on all the track pairs within an event with pt > 10



RDataFrame can define new columns, including those defining filtered results.

RCombinedDS: filtered collections

N

Index = N

1	0	0	0	0	0	0	0	0	0	0	0	...	0
0	1	0	0	0	0	0	0	0	0	0	0	...	0
0	0	1	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	1	0	0	0	0	0	...	0
0	0	0	0	0	0	0	1	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	0	0	...	0
0	0	0	0	0	0	0	0	0	0	1	0	...	0
0	0	0	0	0	0	0	0	0	0	0	1	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	0	0	0	0	0	0	0	...	0

A diagonal matrix can represent a filtered collection (obviously the actual code does not really use one!)

RCombinedDS: putting everything together