DATA ANALYSIS USING ALICE RUN 3 FRAMEWORK

Giulio Eulisse for the ALICE collaboration

ALICE Analysis Model: Run 1 / Run 2

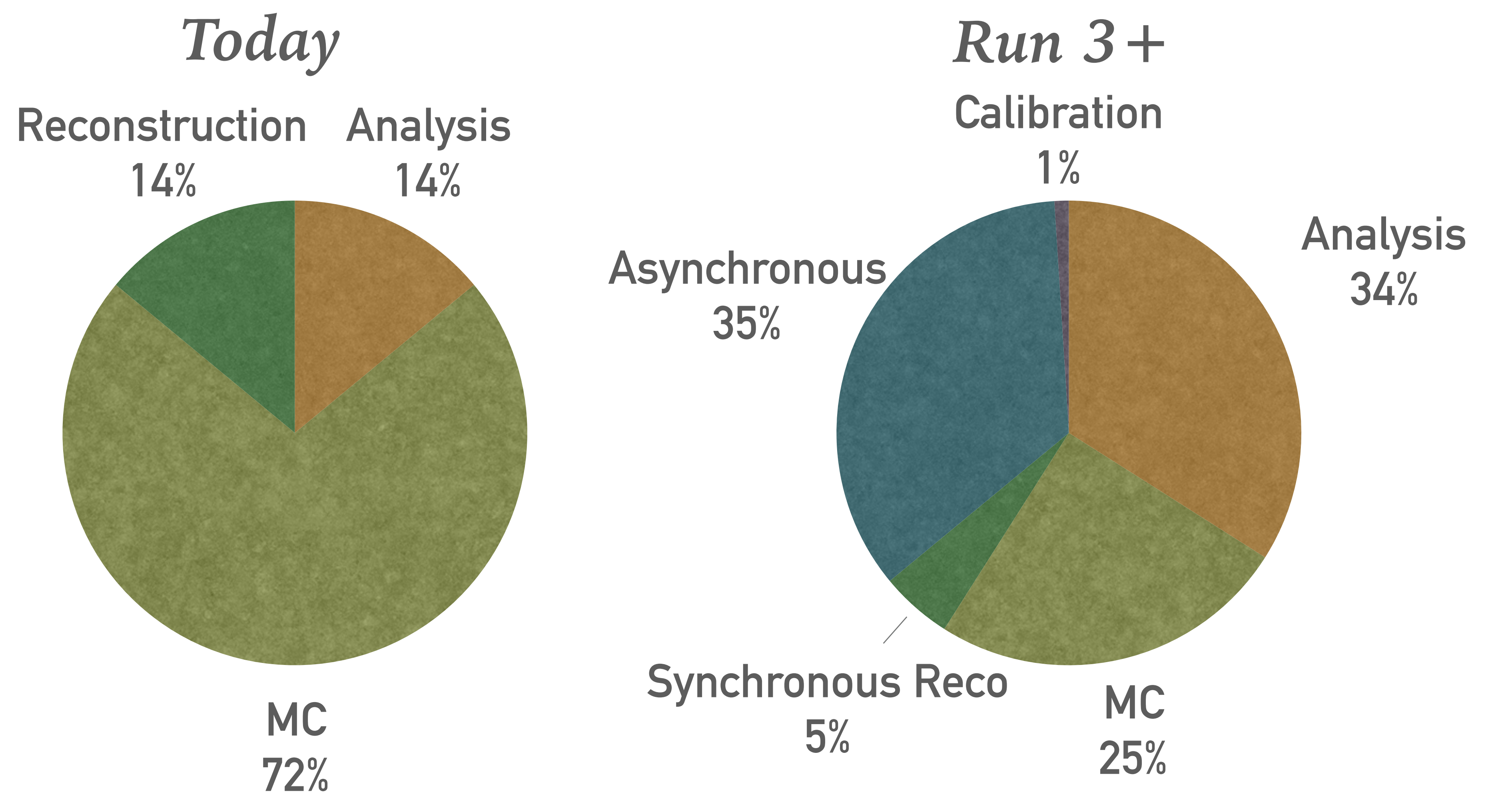
In order to offset the costs of reading data, ALICE has as strong tradition of organised analysis (i.e. trains):

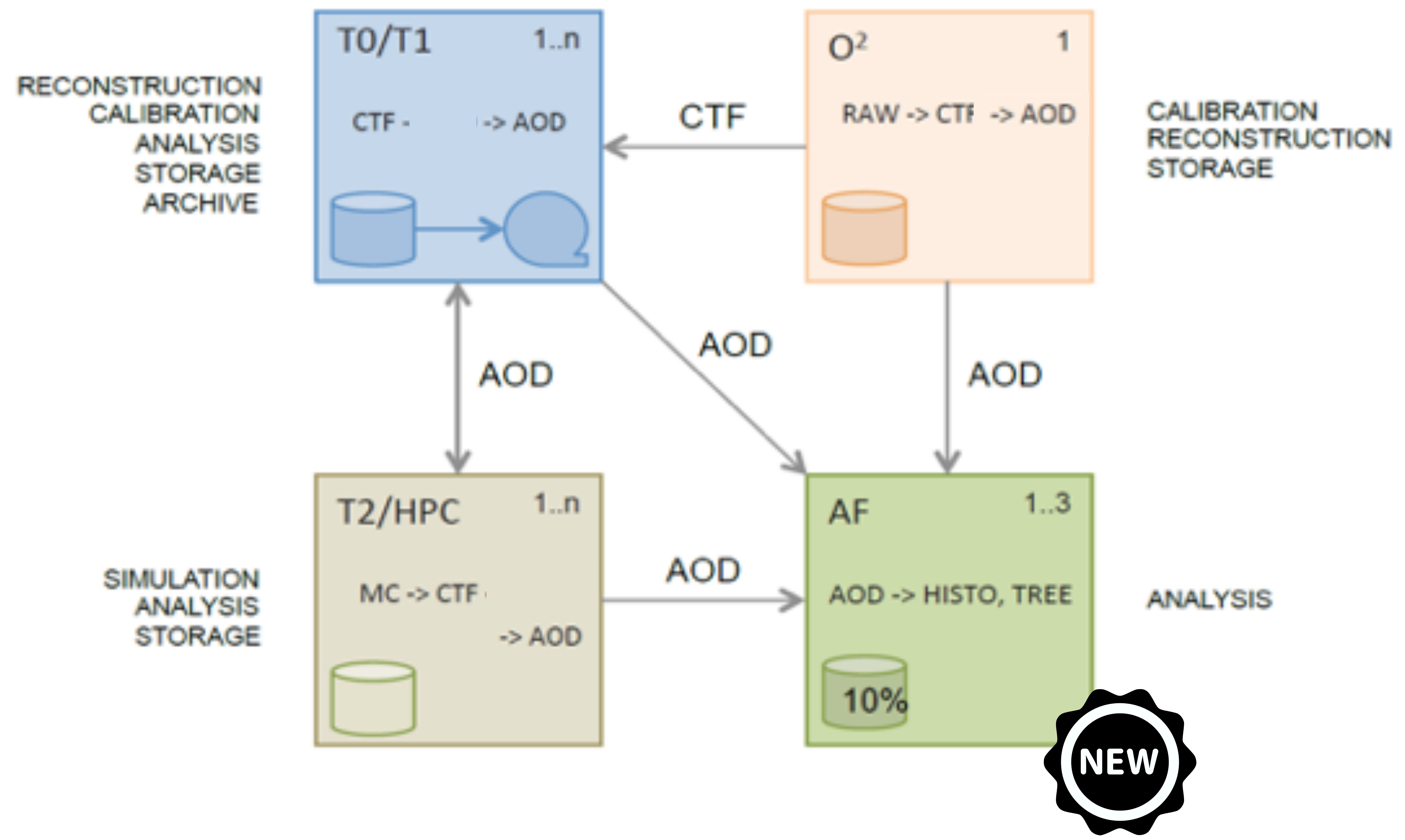
- Analysis performed on both ESD and AODs.
- Users provide tasks, "wagons", organised in "trains". Trains run on the Grid.
- Data are read only once per train, "wagons" get applied to it.
- Data are kept in a generic C++ object store, backed by ROOT.
- Slow sites / site issues dominate overall performance.
- Data-access and de-serialisation of complex object major single core performance offender.

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".*

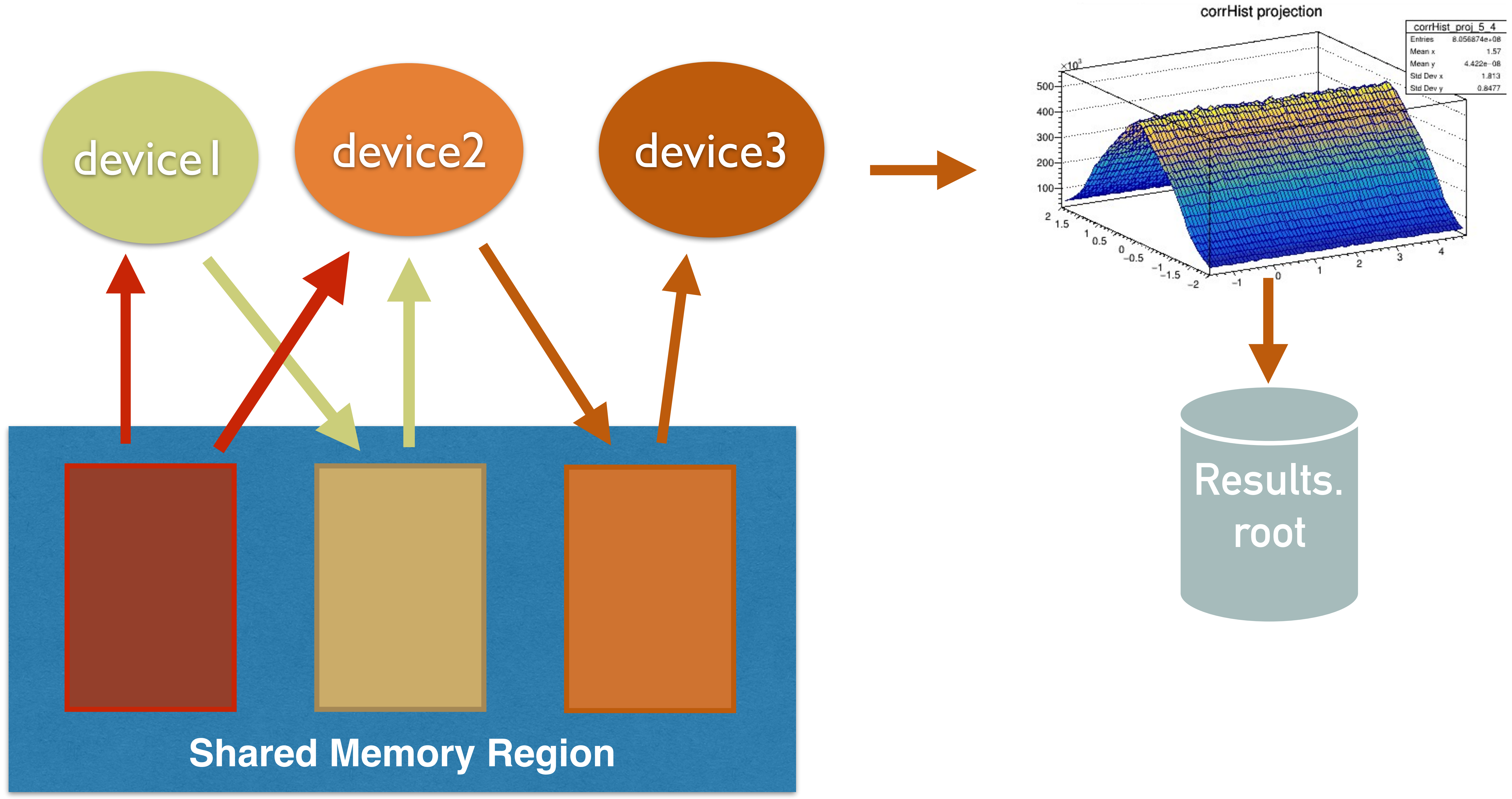
Modern: *follow developments in ROOT and provide an easy way to access modern ROOT tools like RDataFrame.*

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.*

02 / ALFA / FairMQ Framework: General Idea

FairMQ

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



"message" = memory location of one ***shared memory buffer***

02 / ALFA / FairMQ Framework: General Idea

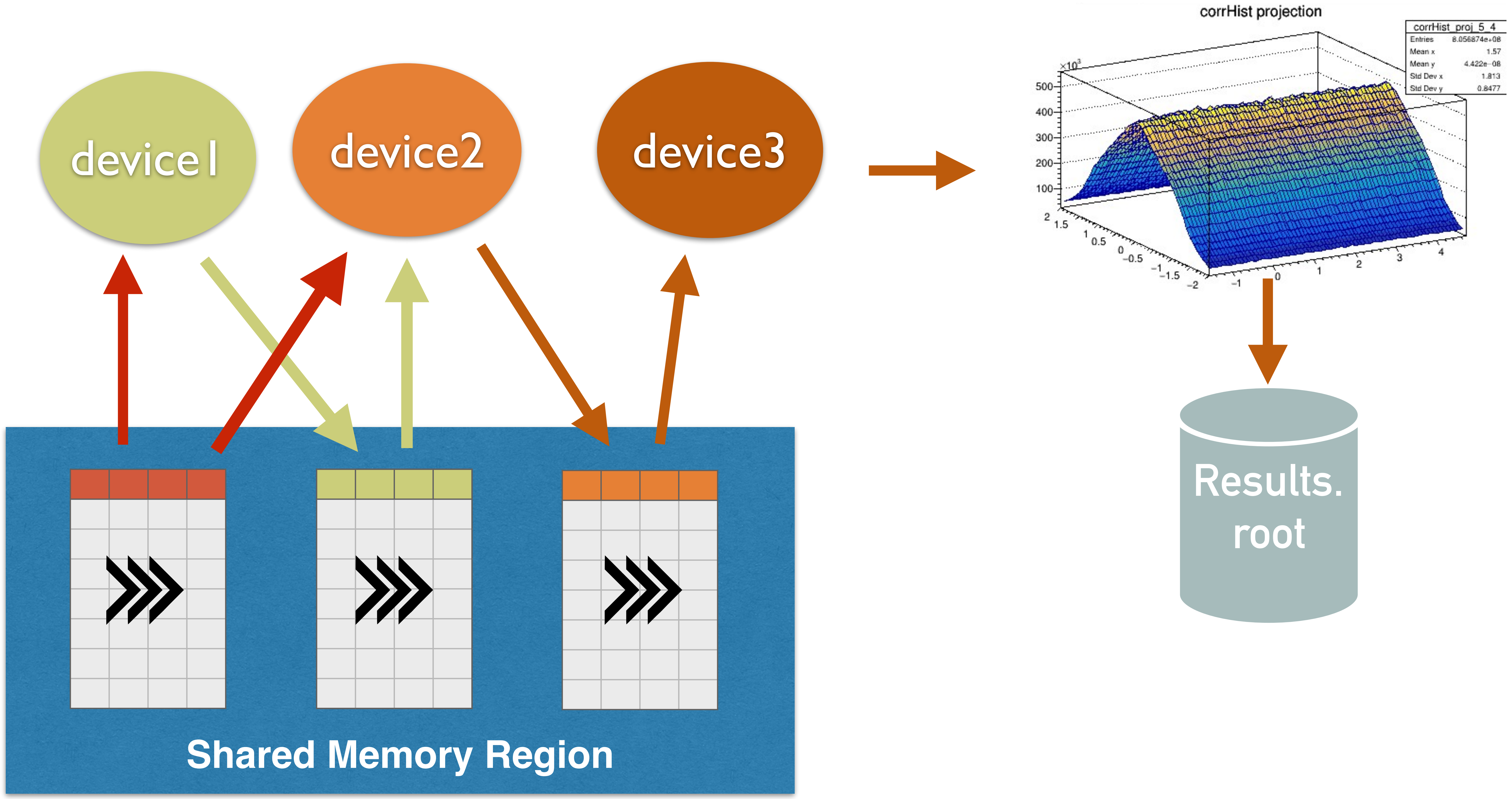
FairMQ

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

Apache Arrow

*For the Analysis Framework in particular we plan to use __**Apache Arrow**__ as backing store for the message passing.*

Arrow fits well to represent column oriented data, while providing some level of flexibility for nested data via the usual record shredding.



*“message” = memory location of one **Arrow Table***

02 / ALFA / FairMQ Framework: General Idea

FairMQ

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

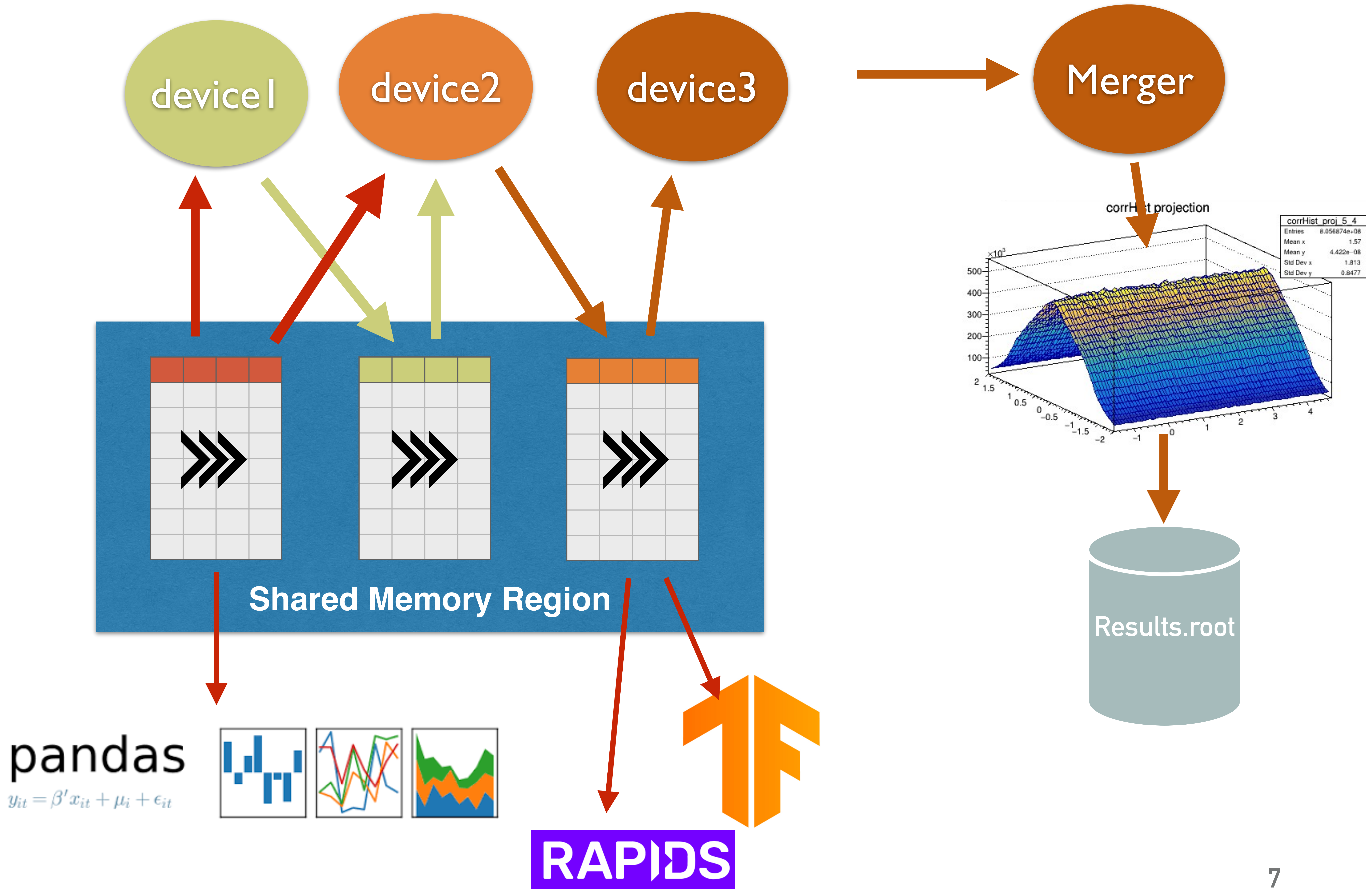
Apache Arrow

For the Analysis Framework in particular we plan to use ***Apache Arrow*** *as backing store for the message passing.*

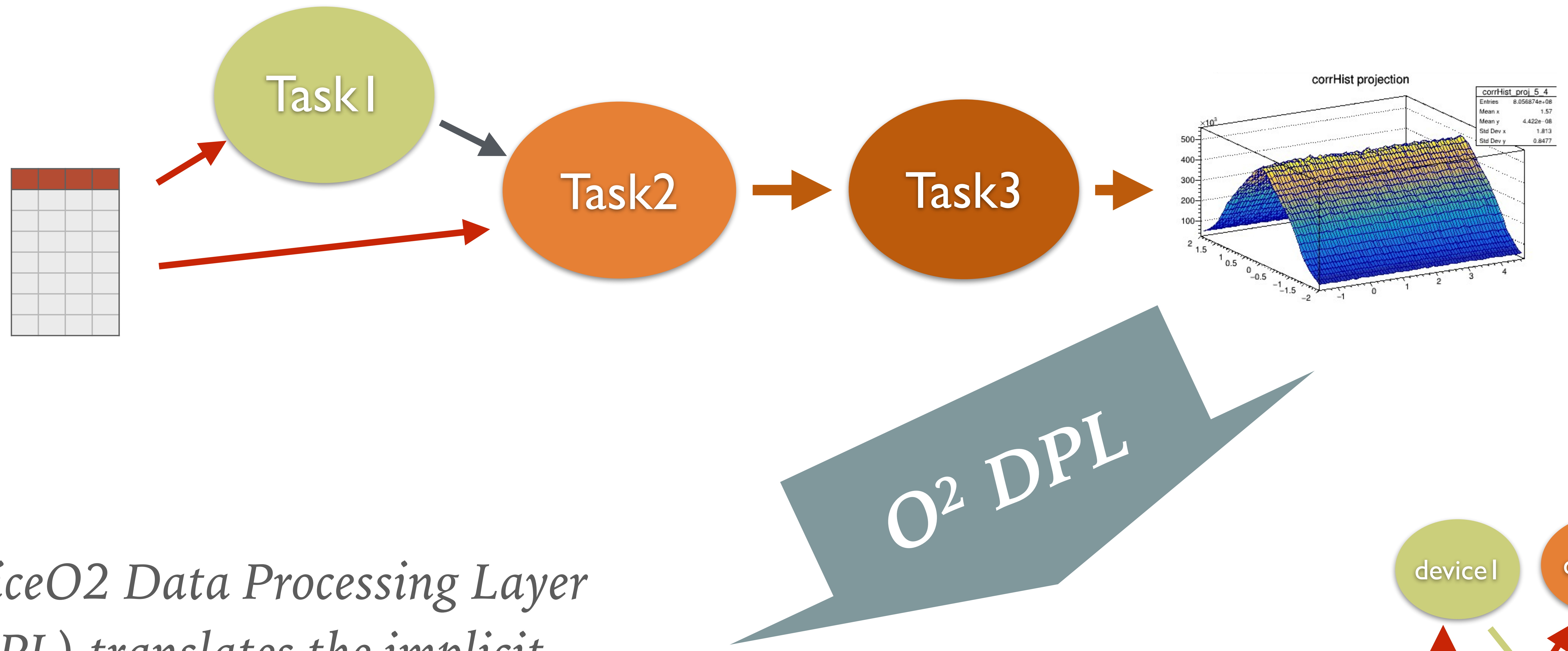
Arrow fits well to represent column oriented data, while providing some level of flexibility for nested data via the usual record shredding.

Interoperability

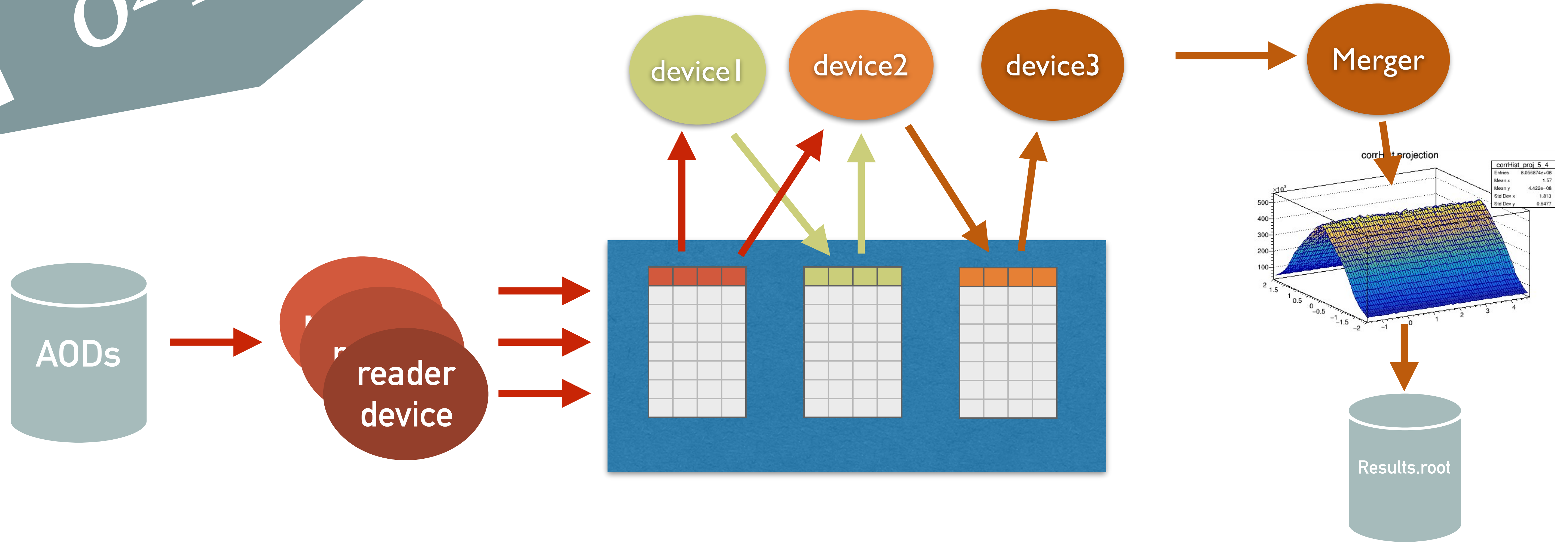
Using Apache Arrow allows for seamless integration with a larger ecosystem of tools, like Pandas or Tensorflow.



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.

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

```cpp
#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)
  };
}
```

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.*
- **O² DPL underpinnings**: *this is actually an O² 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

```cpp
#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.

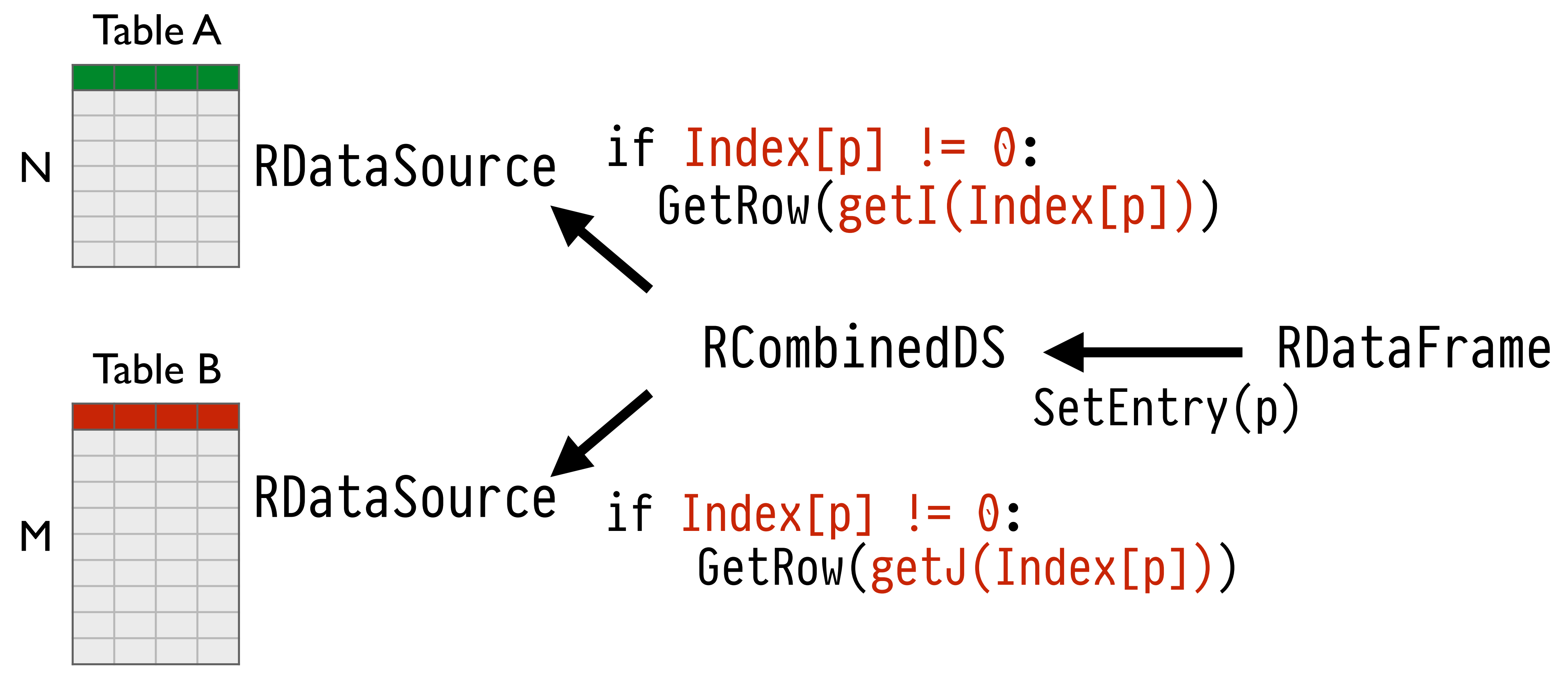
```
auto source = std::make_unique<RArrowDS>(tracks.asArrowTable(), std::vector<std::string>{});
RDataFrame rdf(std::move(source));
```

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.

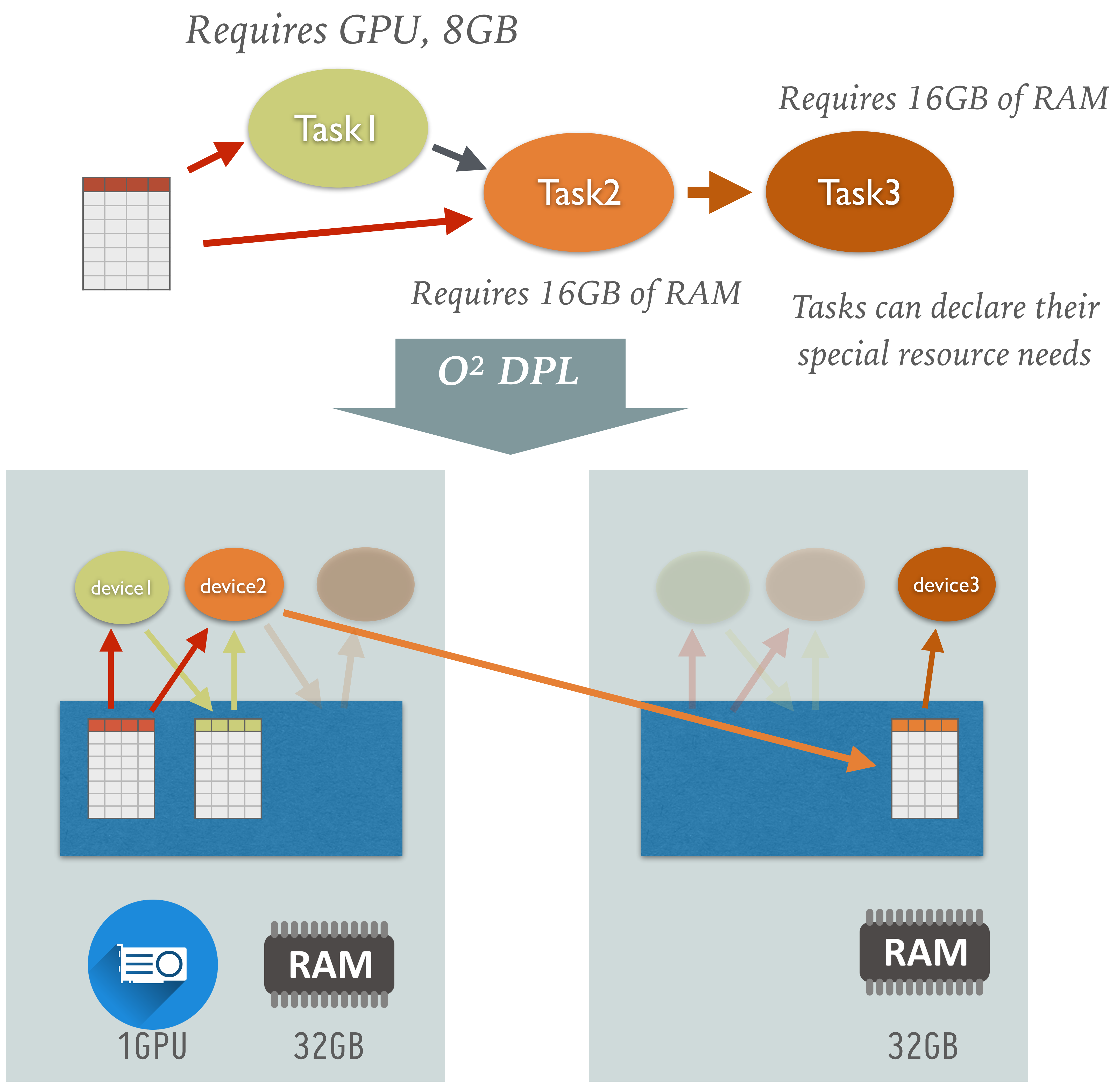


Heterogeneous Computing Support

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.

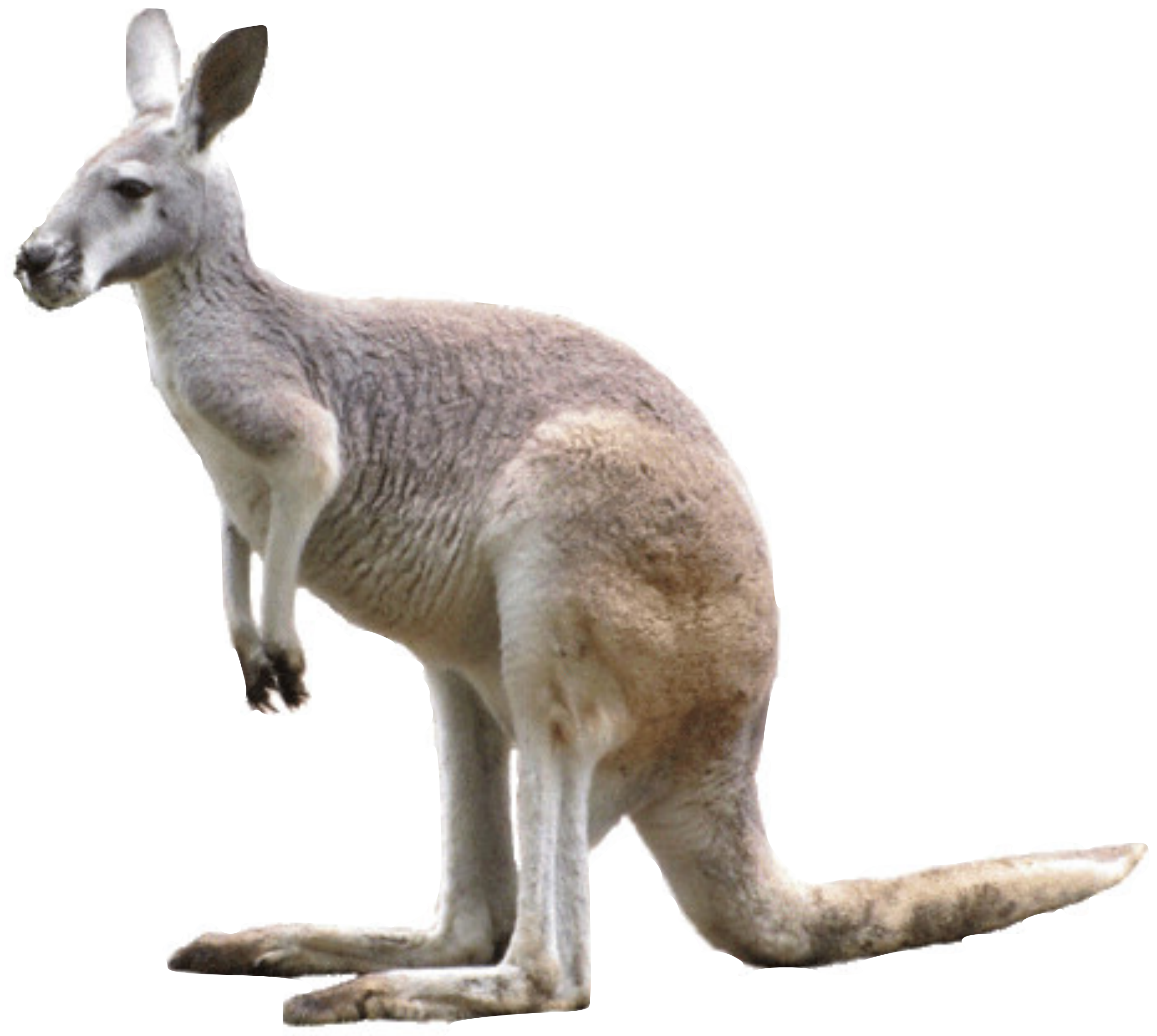
Summary

We are preparing the Analysis Framework for Run 3, exploring different possibilities:

- *Object oriented "**Skins**" as baseline*
- ***RDataFrame** integration for advanced users*
- *Seamless **Python integration** via **Apache Arrow***

All three solutions leverage on the work already being done as part of the more generic Data Processing Layer of ALICE O^2, which builds a workflow engine on top of the FairMQ message passing foundations.

Using Apache Arrow, as in memory backing store, will simplify interoperability with a number of OpenSource tools.



“

Never forget the cost of our choices.

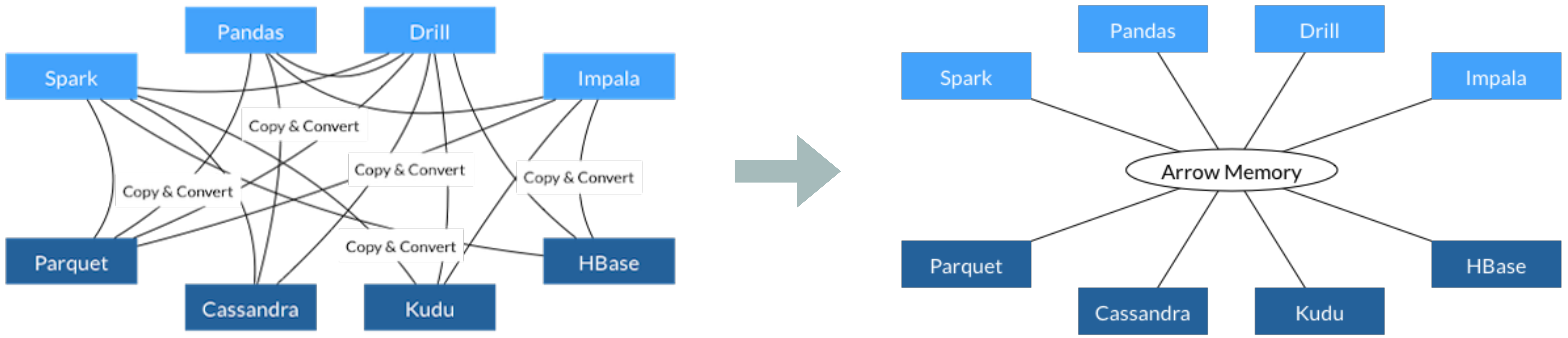
Adventures of Priscilla, Queen of the Desert

(… and good software engineering advice)

BACKUP

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****, Storm, and* ***Tensorflow****.*

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

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

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 organised in Tables. Tables are made of Columns. Columns are (<metadata>, Array). An Array is backed by one or multiple Buffers.*

Nullable fields. *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).*

Investigating suitability as ALICE Run3 Analysis foundation.

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)");
```

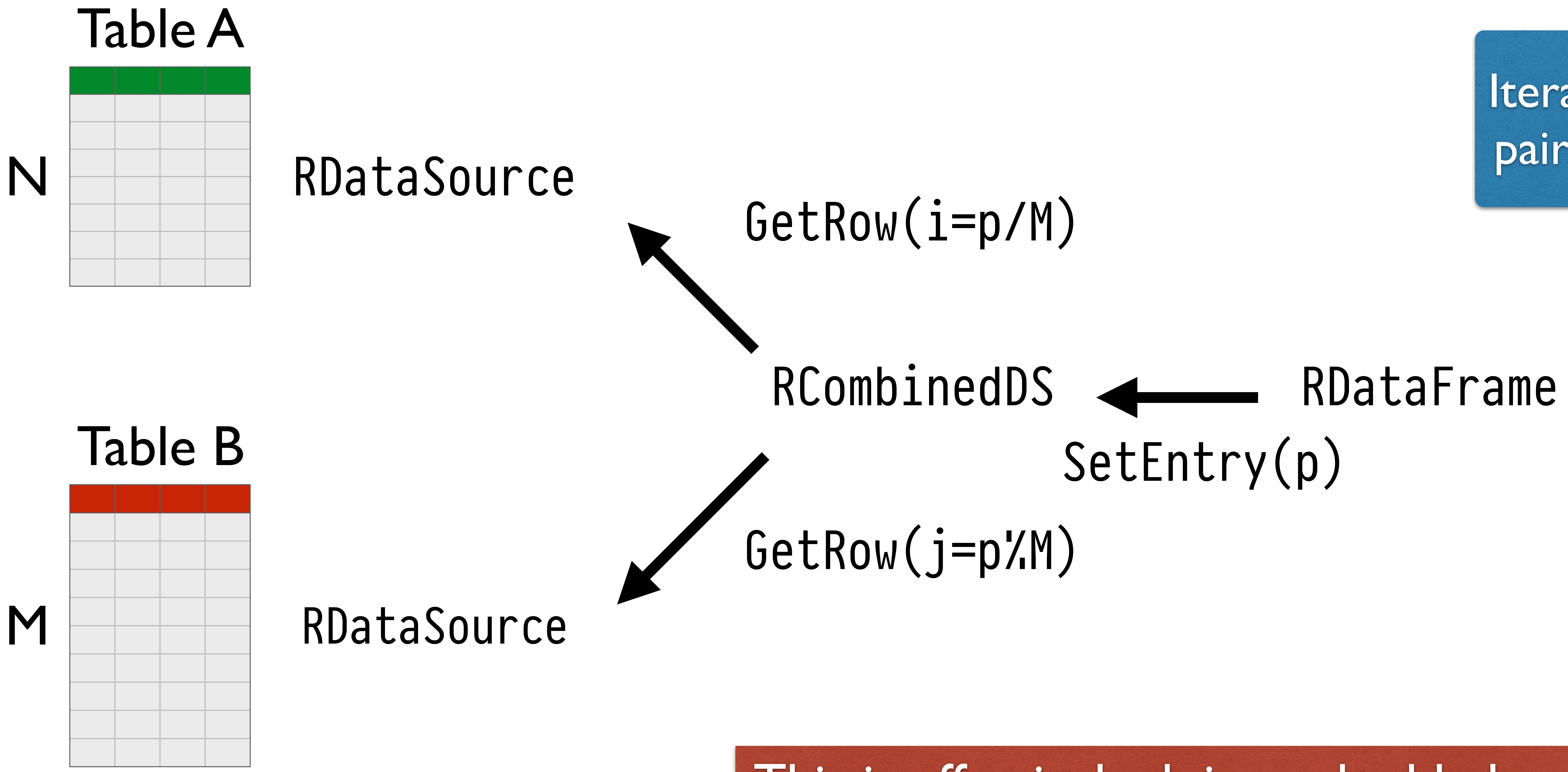
Define extra variables

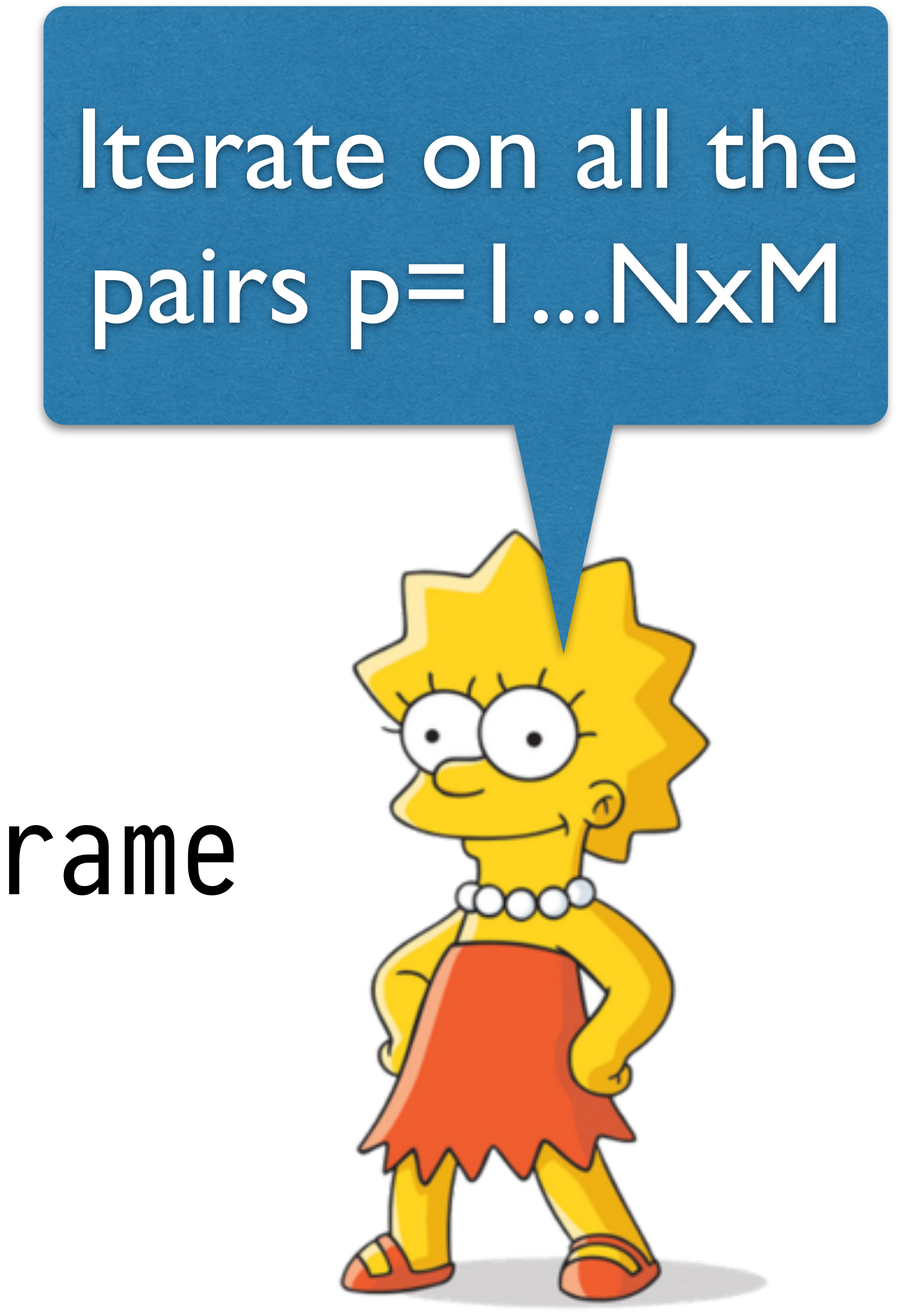
```
auto deltas = filtered.Define("Dphi", "d0_phi_cand - d0bar_phi_cand")
                      .Define("Deta", "d0_eta_cand - d0bar_eta_cand");
```

Create your histograms

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

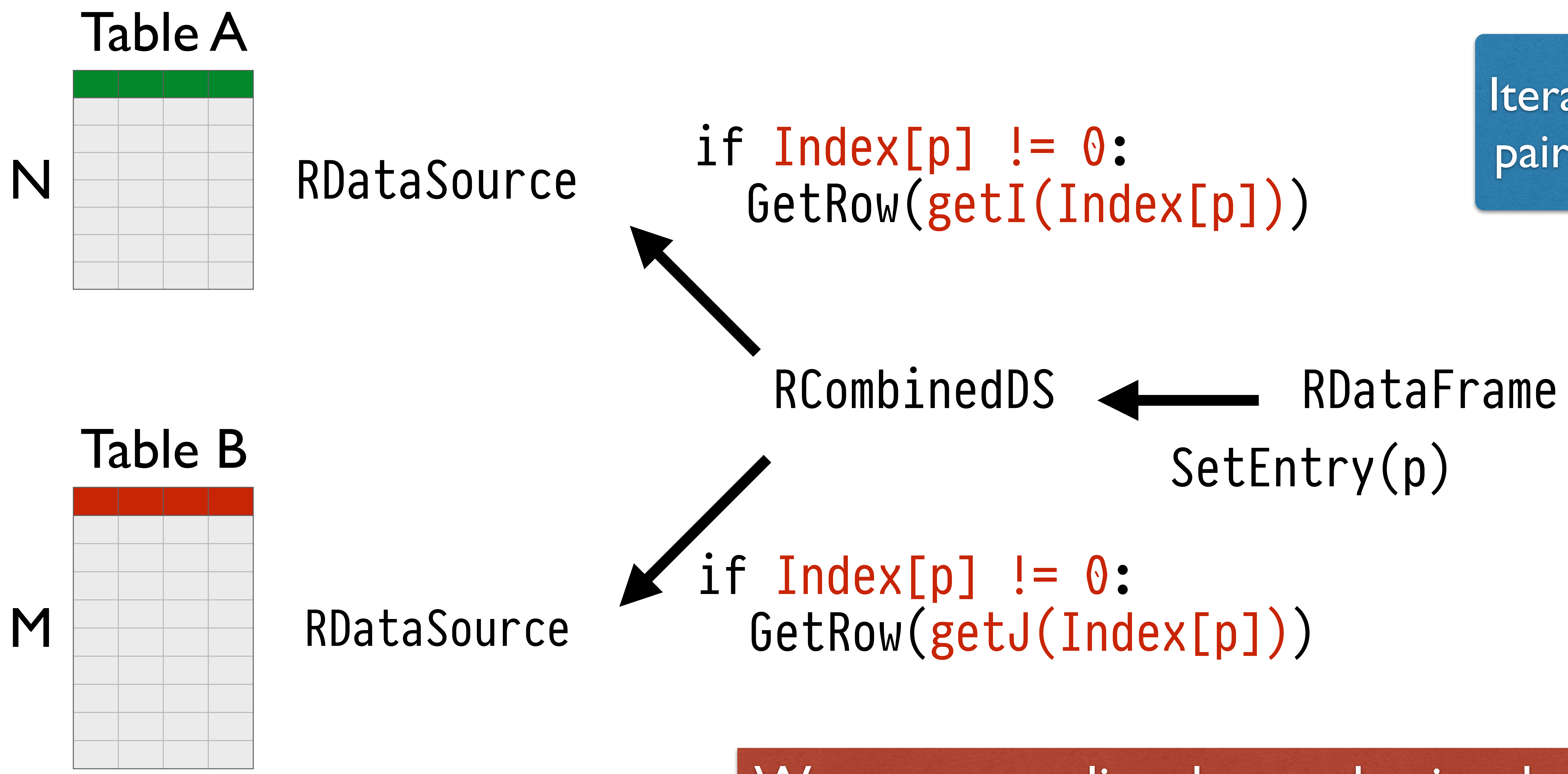
RCombinedDS: combining multiple datasources





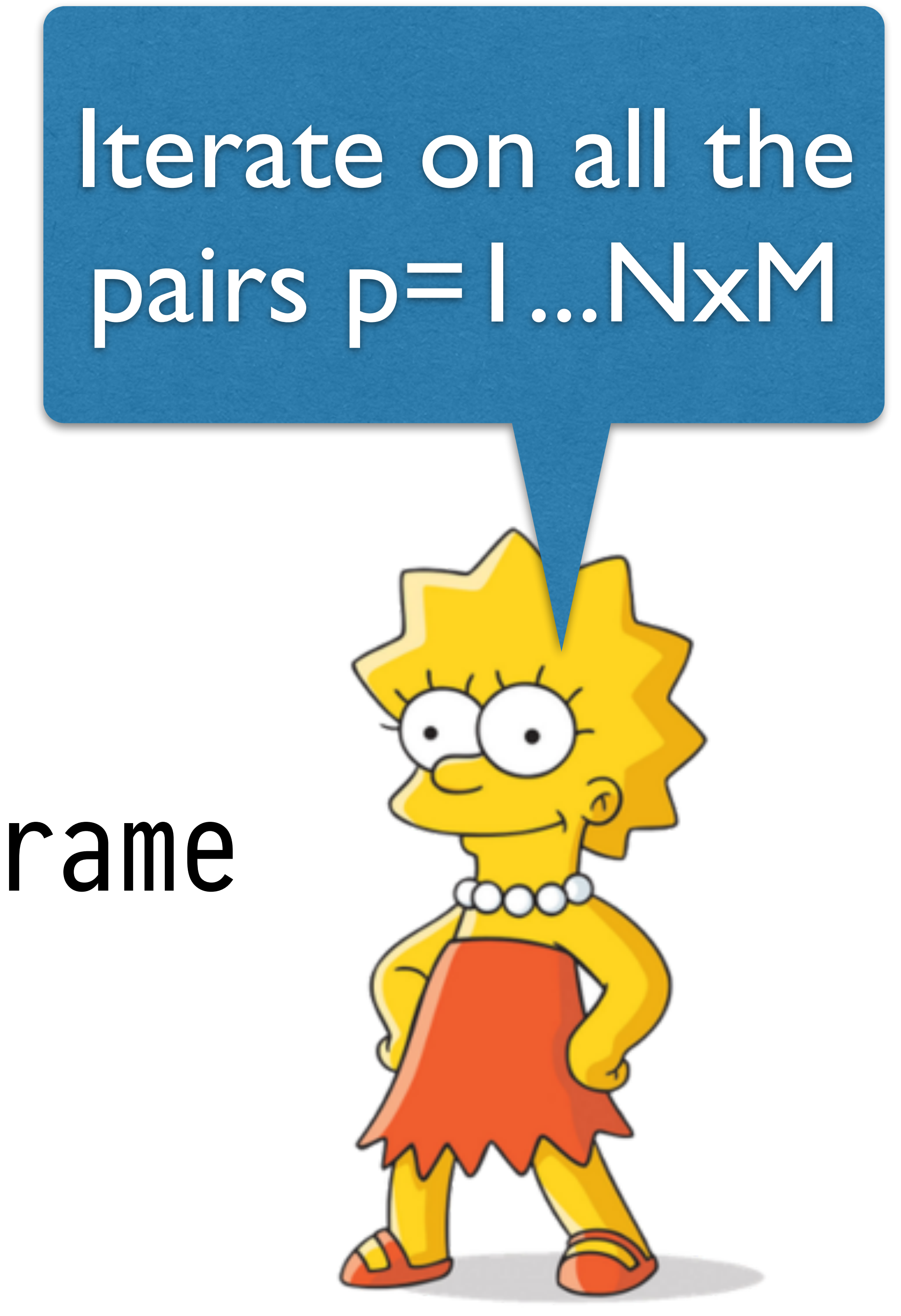
This is effectively doing a double loop on all the possible row pairs of the table A and B.

RCombinedDS: generalisation



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

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

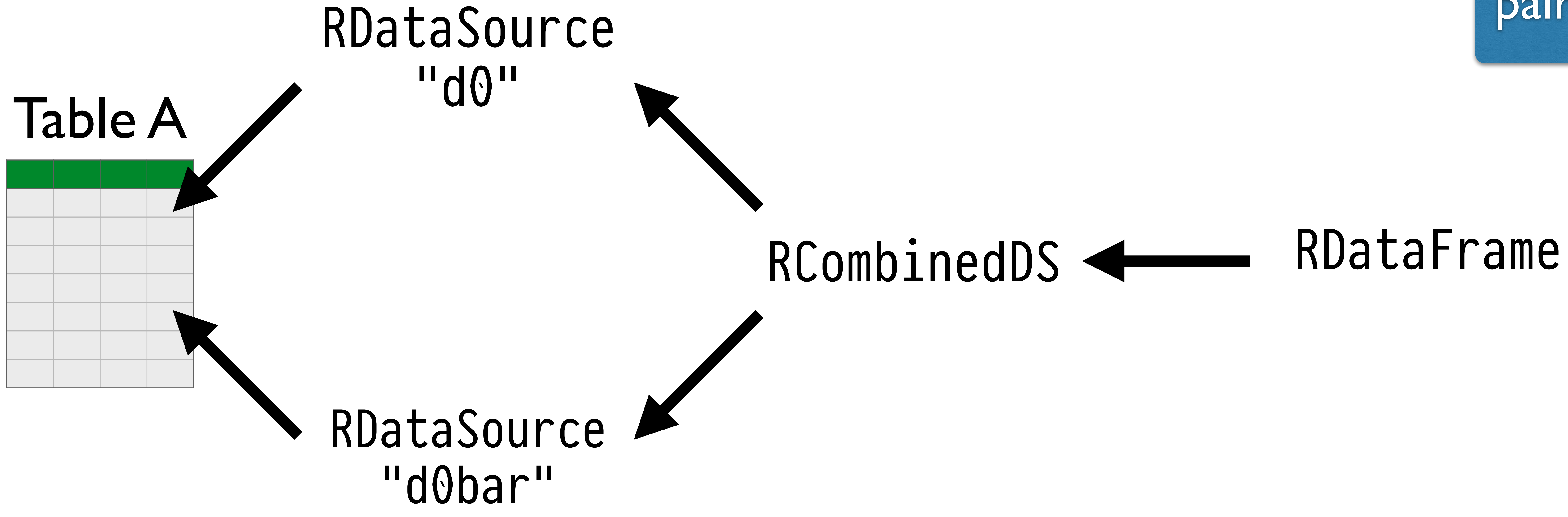


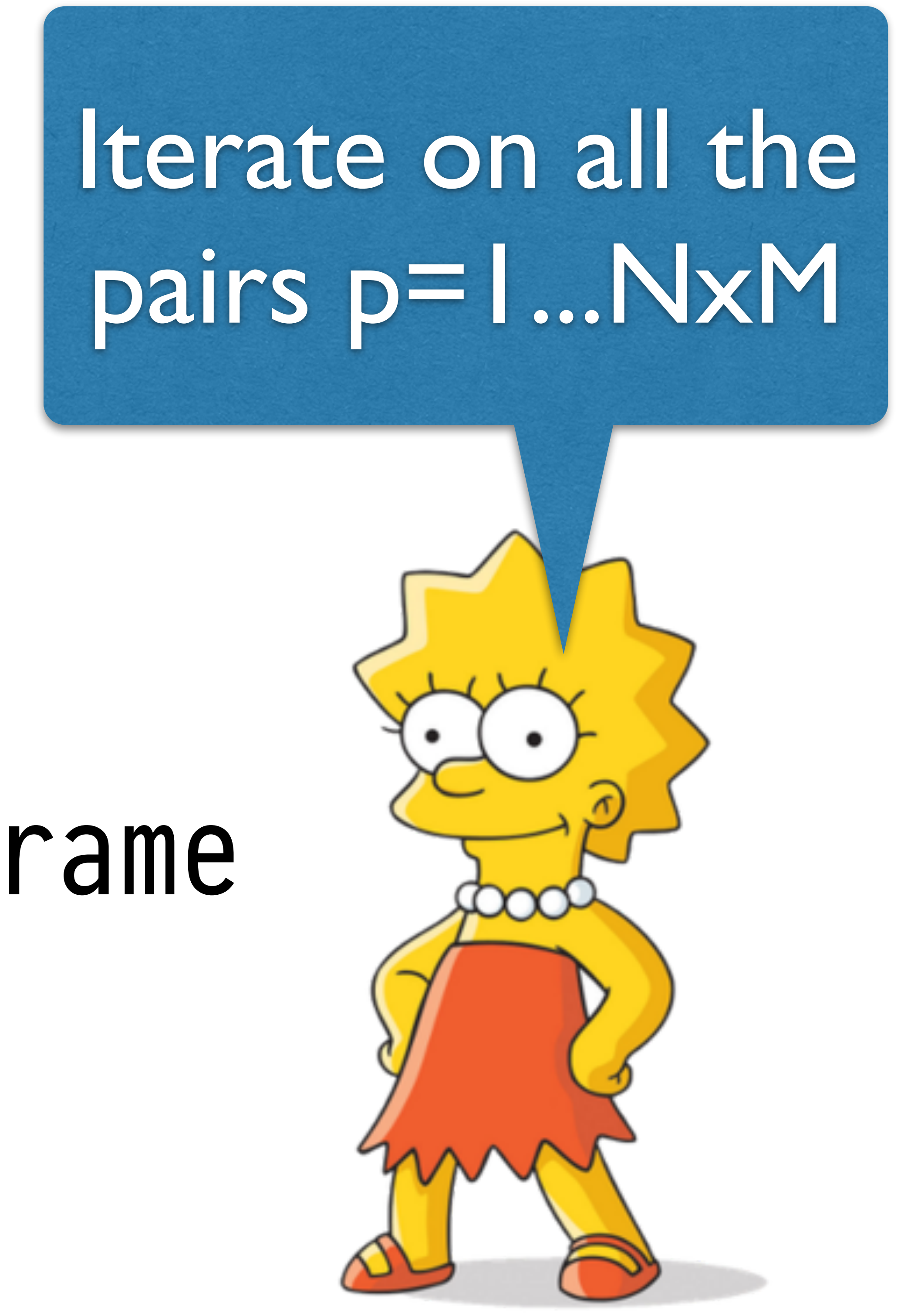
We can generalise the mechanism by having an index to select the pairs to yield.

RCombinedDS: full double loop

$$\text{Index} = N \left\{ \overbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ \hline
\end{array}}^{M} \right.$$

RCombinedDS: same source





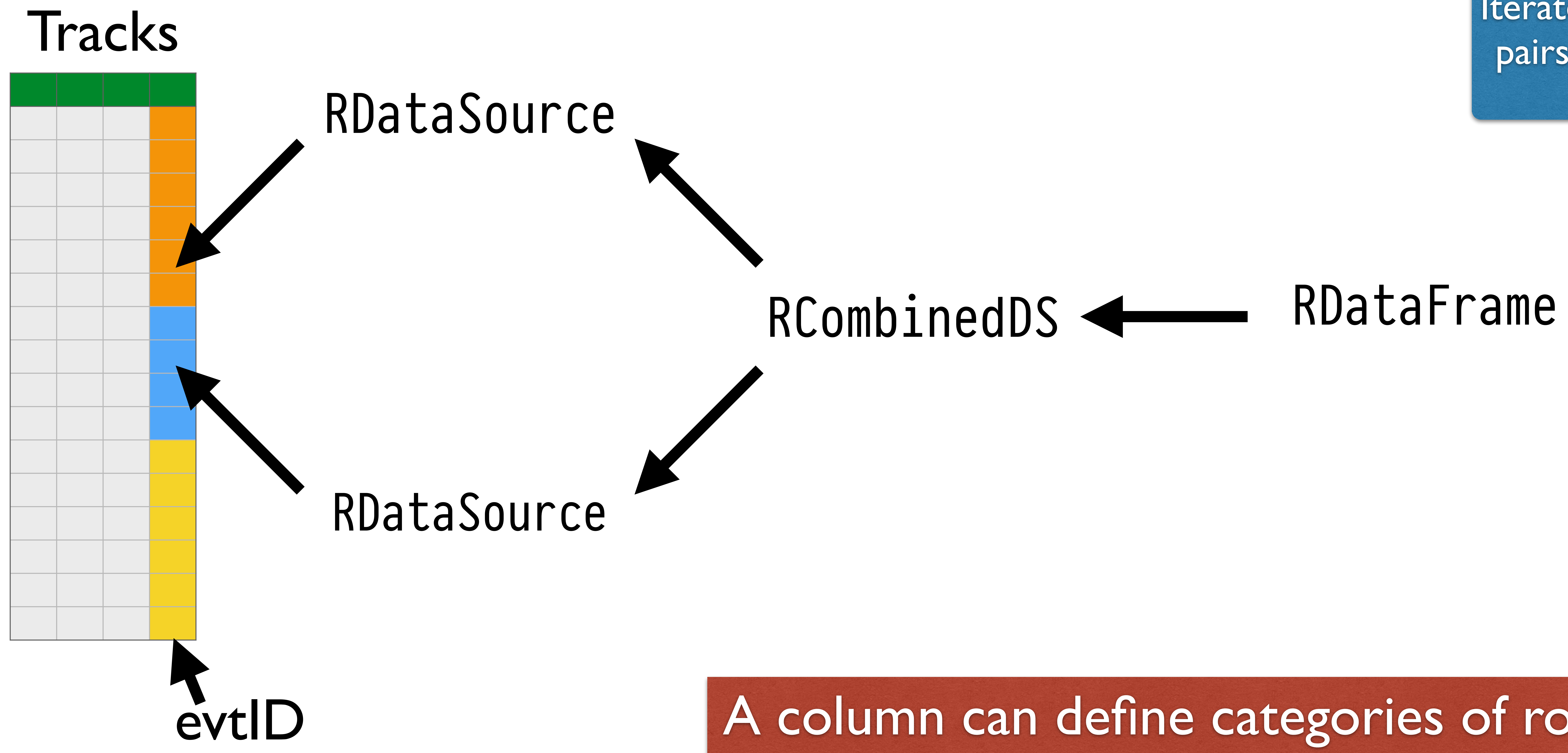
Two RDataSource can actually point to the same table.

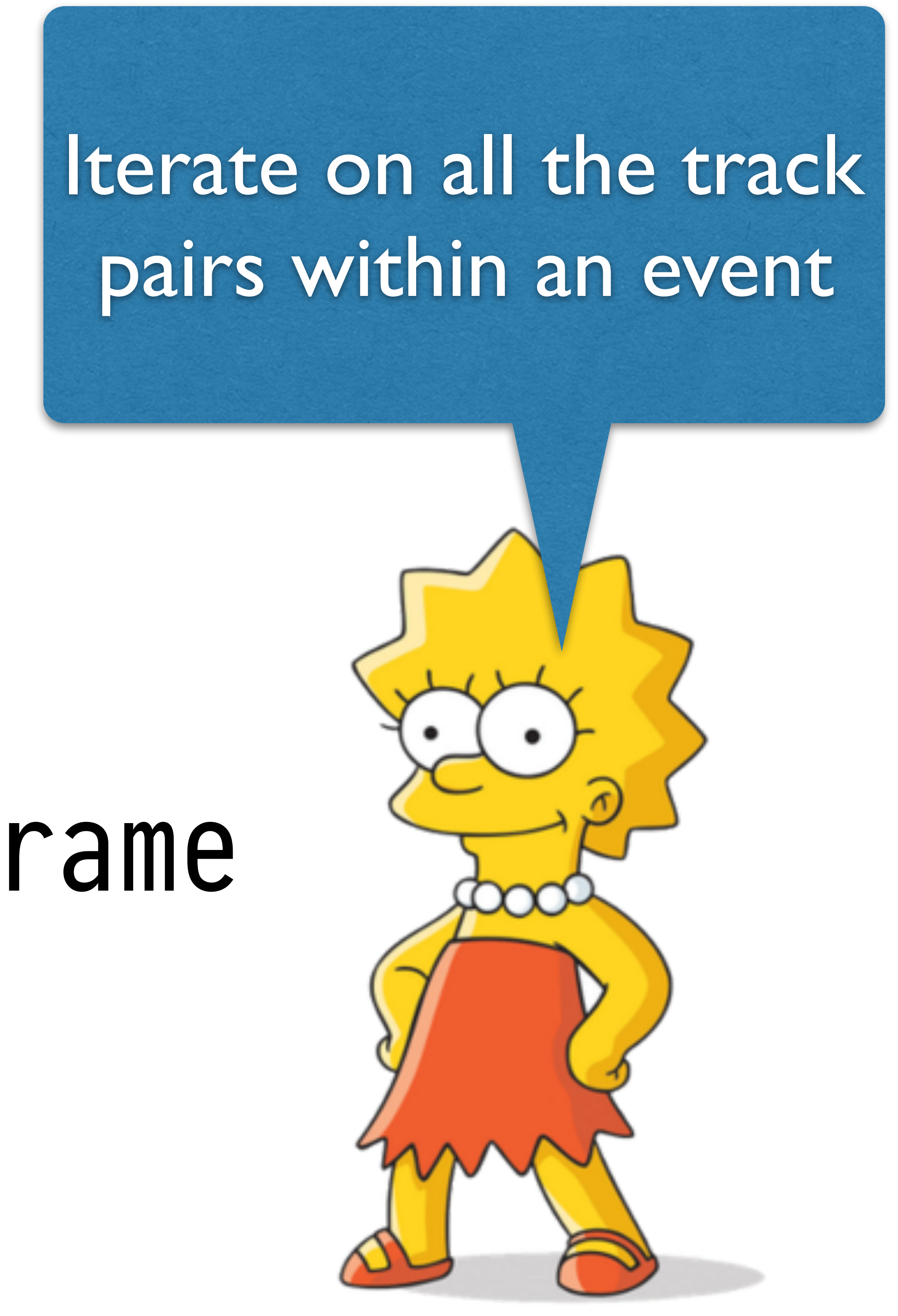
RCombinedDS: double loop without repetitions

$$\text{Index} = N \left\{ \overbrace{\begin{array}{cccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0
\end{array}}^{N} \right.$$

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

RCombinedDS: generalisation





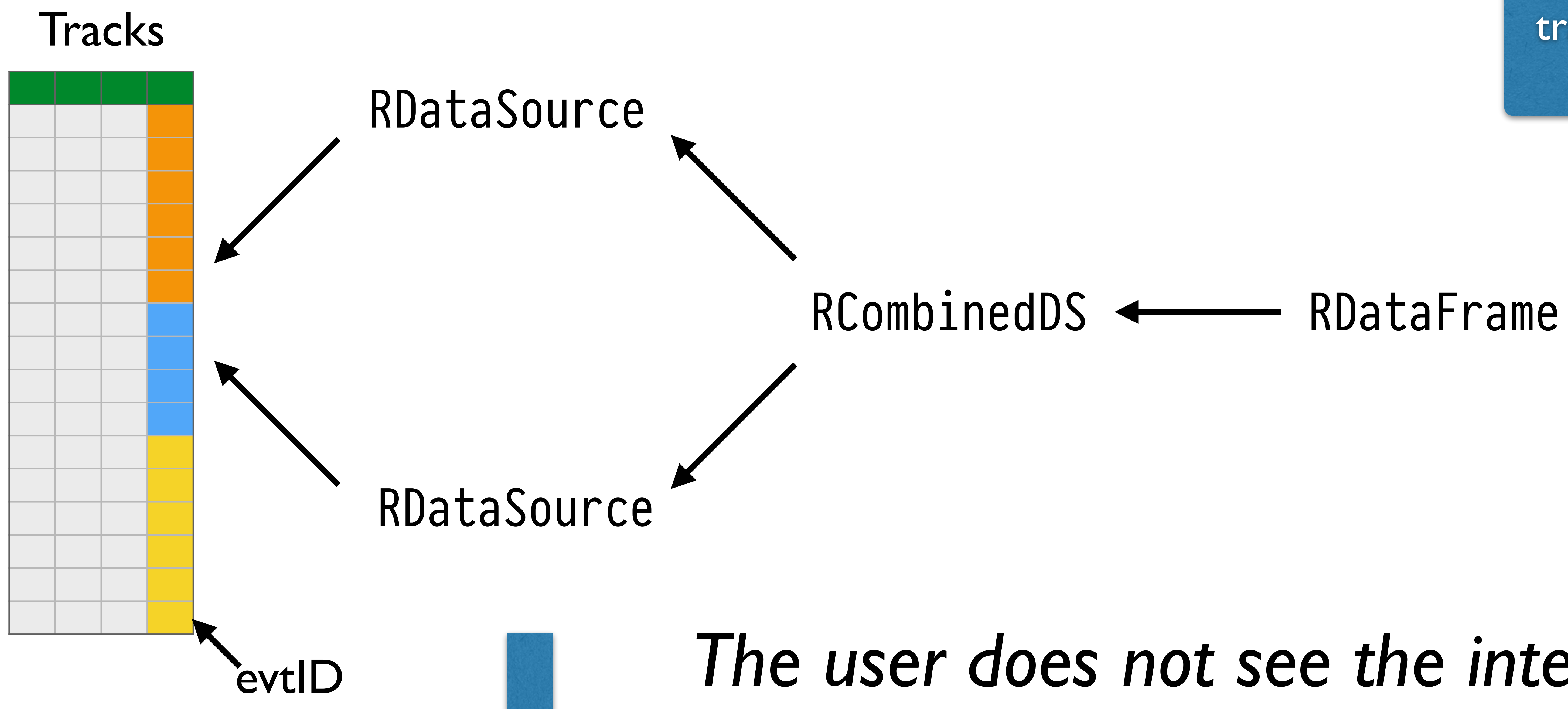
A column can define categories of rows, e.g. the event id.

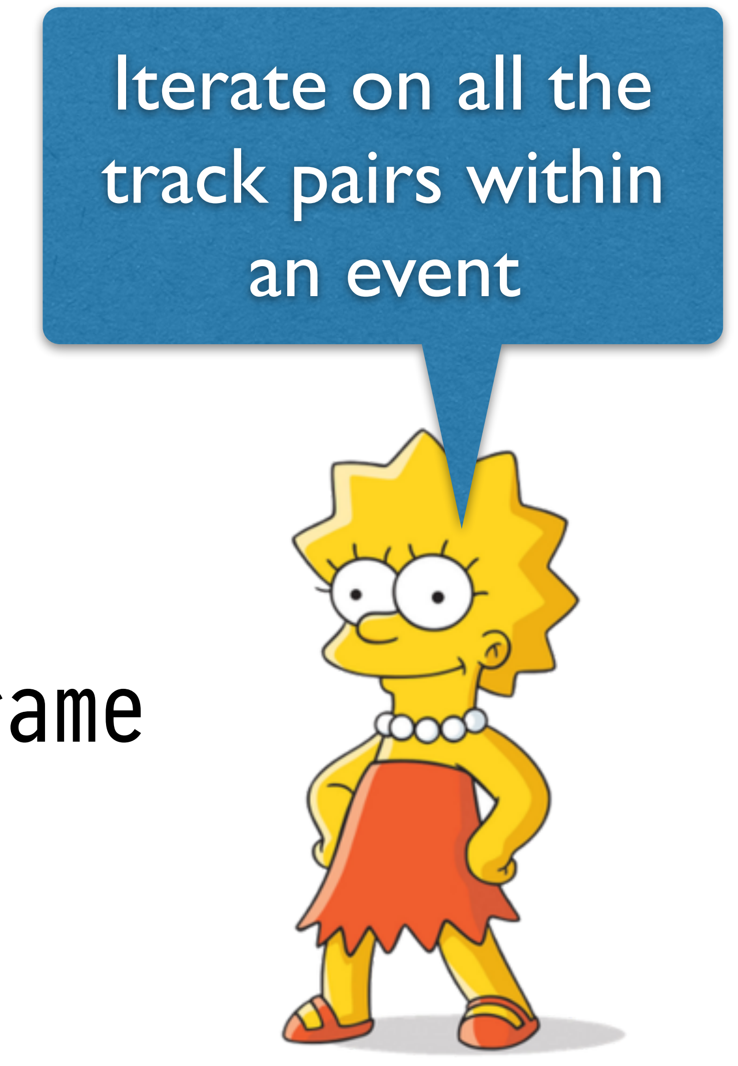
RCombinedDS: double with loop within a category

$$\text{Index} = N \left\{ \overbrace{\begin{array}{cccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0
\end{array}}^{N} \right.$$

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





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");
```

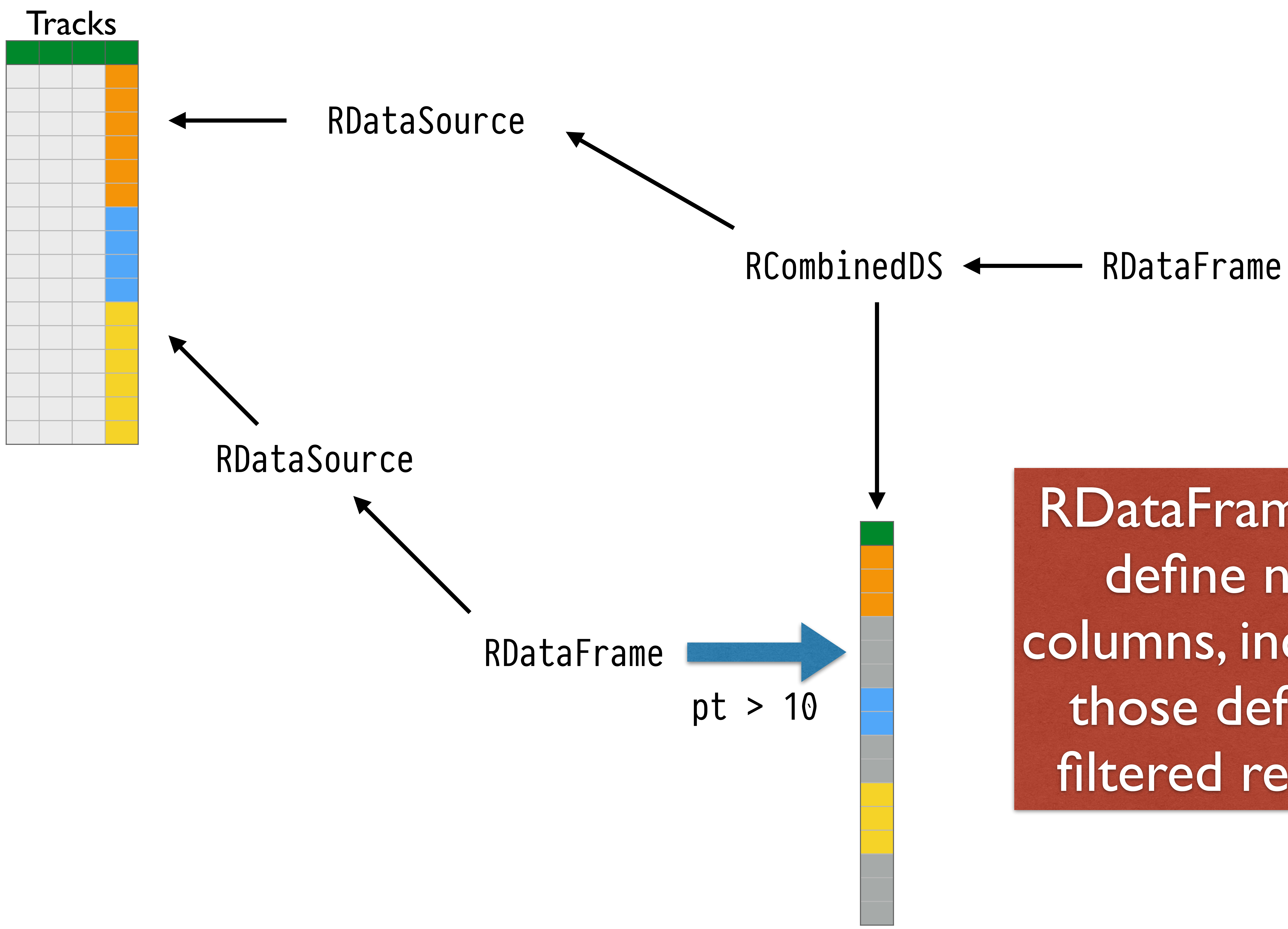
Properly setup RDataFrame

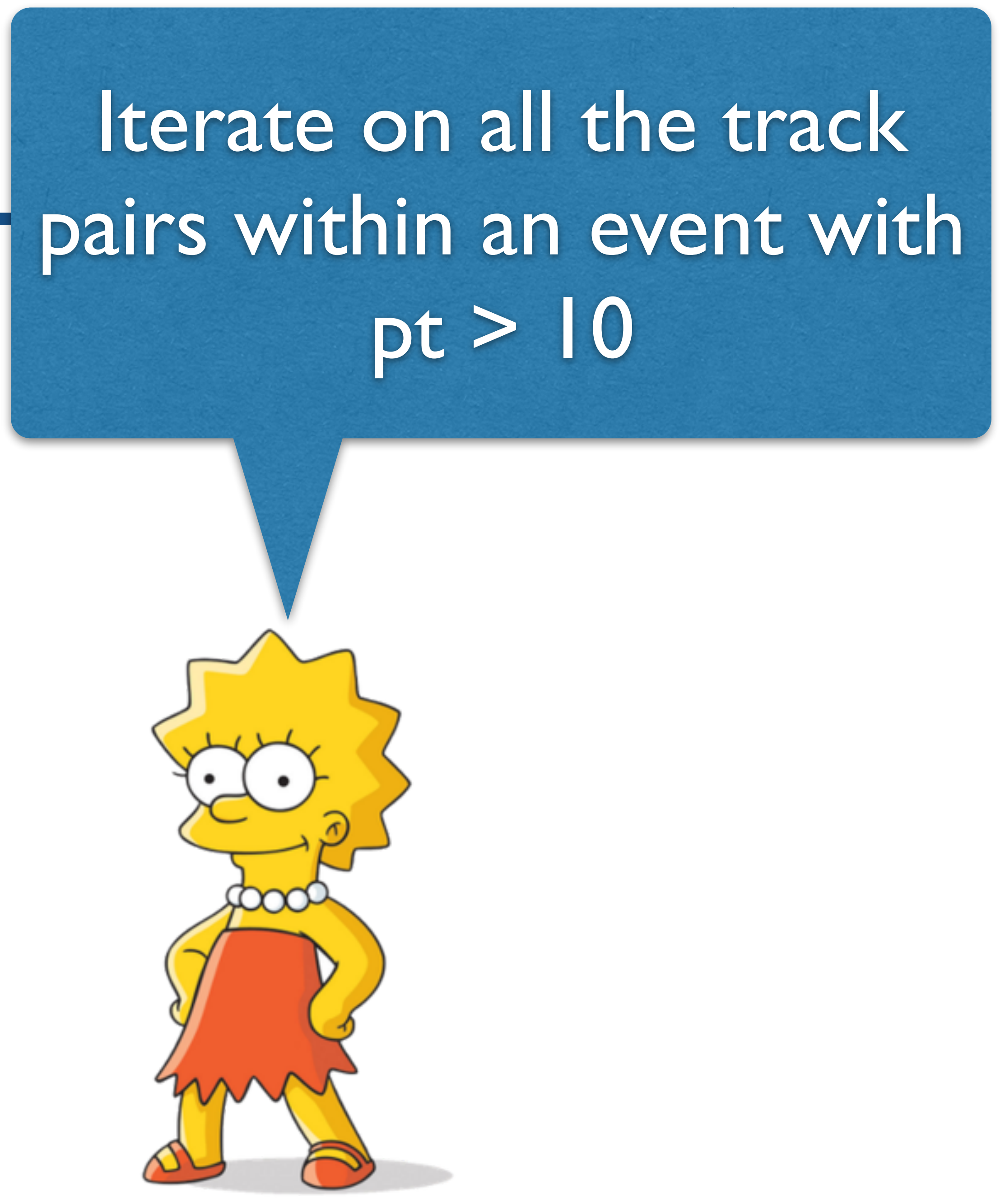
Input subscribed from DPL

mnemonic for the table

Column to use for category

RCombinedDS: filtered collections





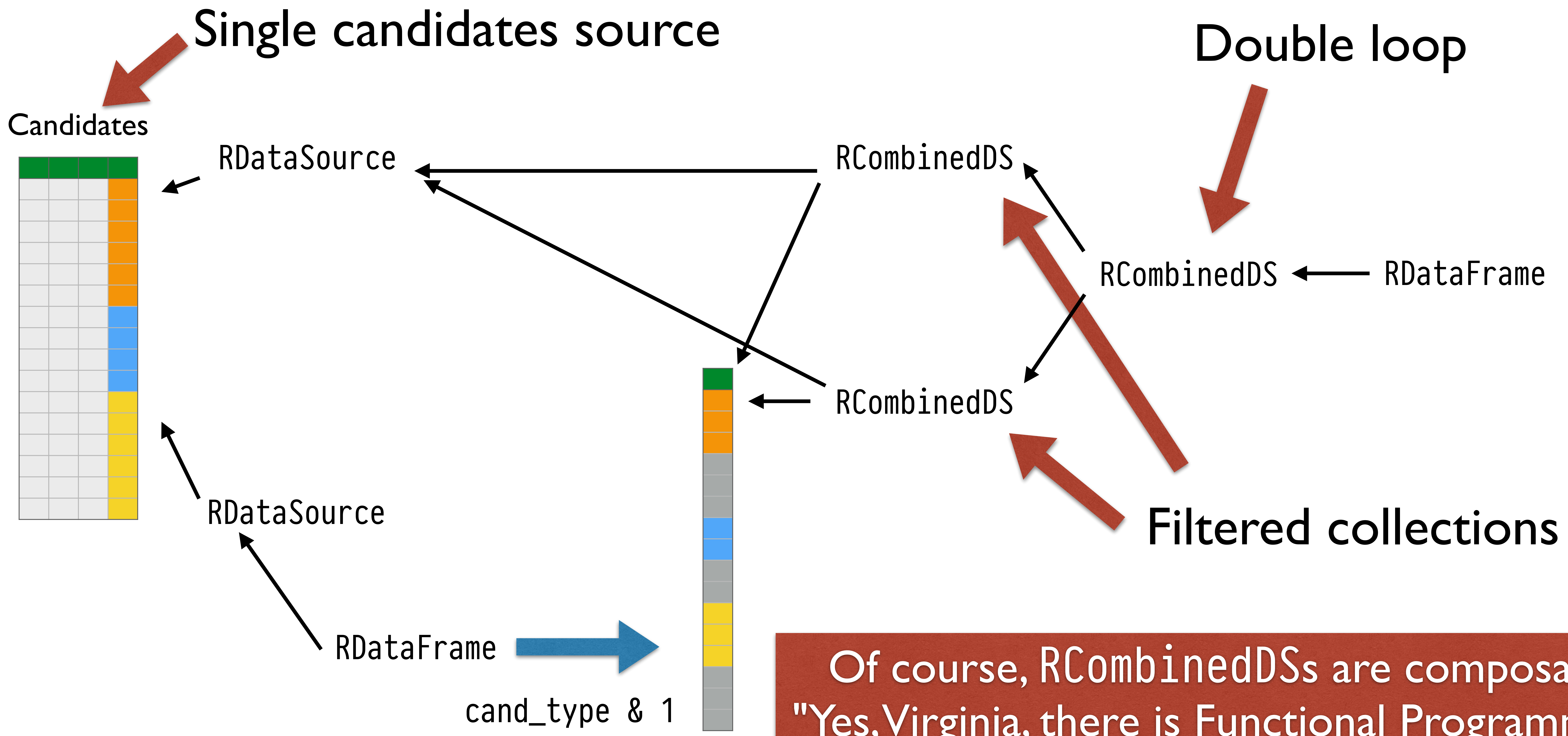
RDataFrame can define new columns, including those defining filtered results.

RCombinedDS: filtered collections

$$\text{Index} = N \left\{ \overbrace{\begin{array}{cccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0
\end{array}}^{N} \right.$$

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

RCombinedDS: putting everything together



Of course, RCombinedDSs are composable.
"Yes, Virginia, there is Functional Programming."

A Trivial Analysis

- Fill a plot

```
OutputObj<TH1F> hPhi{TH1F("phi", "Phi", 100, 0, 2 * M_PI)};


    hPhi->Fill(phi);
```

A Trivial Analysis

- Fill a plot
- Compute (e.g.) φ from the propagation parameters

```
OutputObj<TH1F> hPhi{TH1F("phi", "Phi", 100, 0, 2 * M_PI)};
```

```
    float phi = asin(track.snp()) + track.alpha() + M_PI;
    hPhi->Fill(phi);
```

A Trivial Analysis

- Fill a plot
- Compute (e.g.) φ from the propagation parameters
- Get all the tracks for a given timeframe

```
OutputObj<TH1F> hPhi{TH1F("phi", "Phi", 100, 0, 2 * M_PI)};

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

A Trivial Analysis

- Fill a plot
- Compute (e.g.) φ from the propagation parameters
- Get all the tracks for a given timeframe
- Define an AnalysisTask

```
struct ATask : AnalysisTask
{
  OutputObj<TH1F> hPhi{TH1F("phi", "Phi", 100, 0, 2 * M_PI)};

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

A Trivial Analysis

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

```
#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)};

  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)
  };
}
```