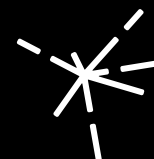


**ALICE**  
OO 5.36 TeV  
[CDS link](#)

ÖAW

ÖSTERREICHISCHE  
AKADEMIE DER  
WISSENSCHAFTEN



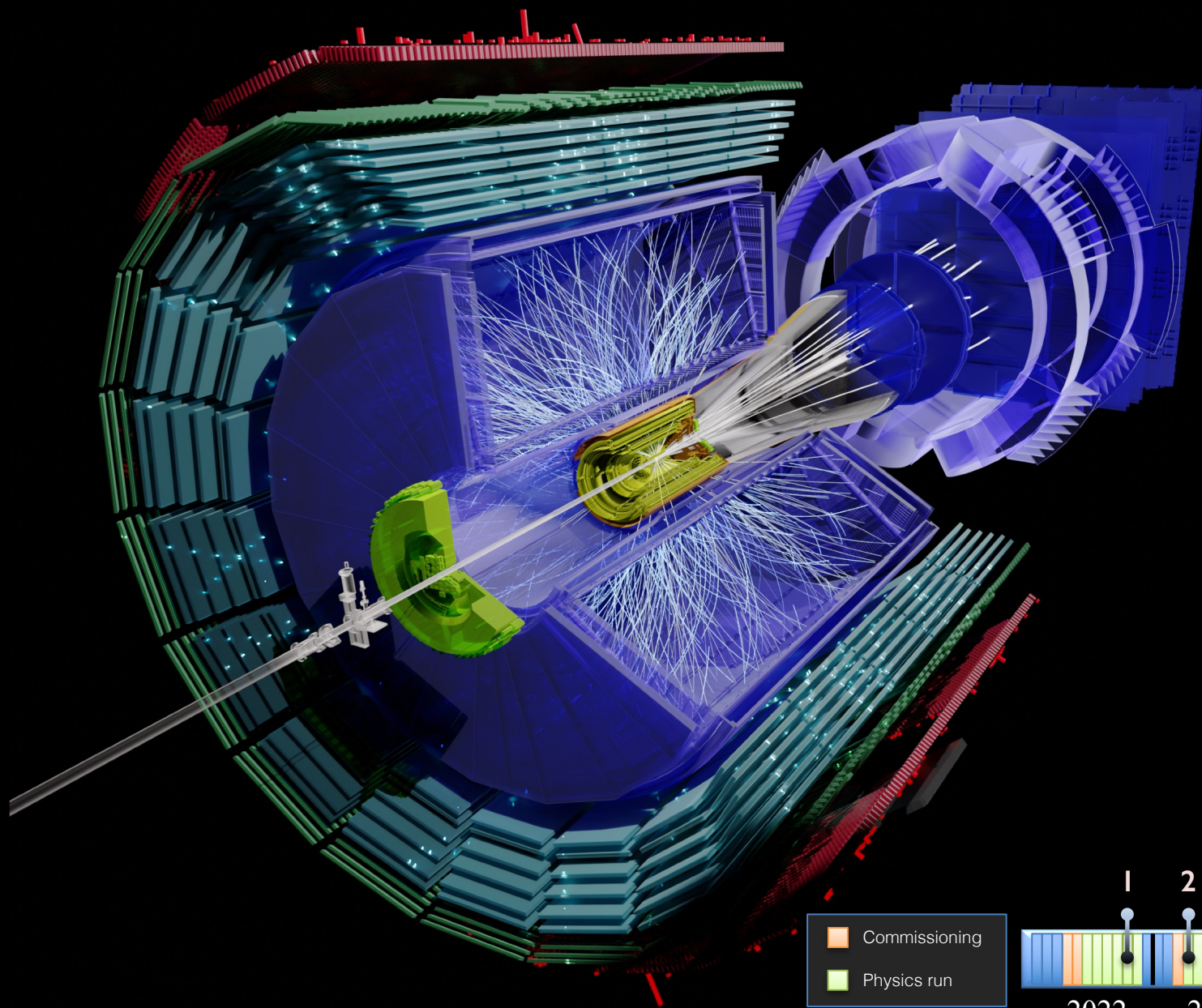
MARIETTA BLAU  
INSTITUTE FOR  
PARTICLE PHYSICS

O2 Analysis Tutorial 5.0:

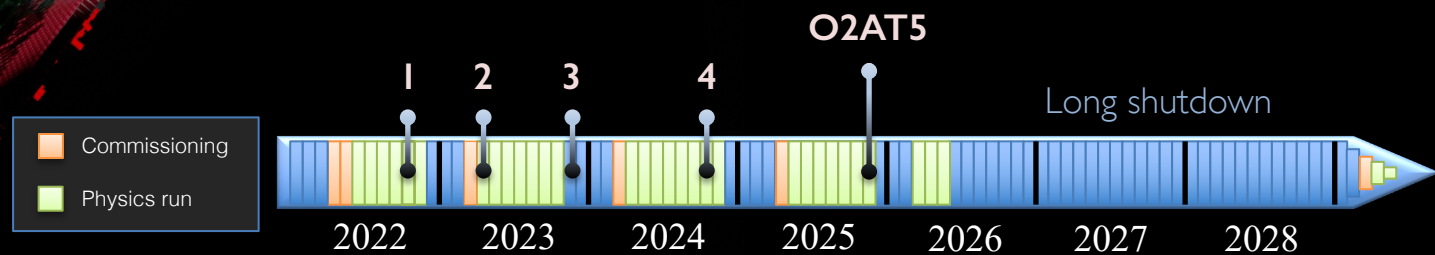
# Introduction to O2/O2Physics

10<sup>th</sup> November 2025





Where are we today?



Latest LHC schedule: <https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm>

# O2 Analysis tutorial 5.0: what to expect of this week

(Thank you very much to the entire crew!)

Day	Content	Room booking
<b>7th Nov</b> Friday	<b>Introduction:</b> aliBuild, git, O2/O2Physics quick start ( <a href="#">done already!</a> )	<b>Virtual</b>
<b>10th Nov</b> Mon Morning	<b>Lectures:</b> Introduction, hyperloop, DPG talks	<b>I 60/I-009</b>
<b>10th Nov</b> Mon Afternoon	<b>Hands-on:</b> (a) basic analysis, (b) simulation analysis, (c) derived data	<b>222/R-003</b>
<b>11th Nov</b> Tue Morning	<b>Lectures:</b> PID, strangeness, Run 2 analysis, derived data, ML, software triggers	<b>222/R-003</b>
<b>11th Nov</b> Tue Afternoon	<b>Hands-on:</b> (a) templates and compile-time polymorphism (b) Dedicated simulations talk	<b>222/R-003</b>
<b>12th Nov</b> Wed Morning	<b>PWG-HF:</b> framework, example D-meson analysis	HF: <b>I 3/2-005</b>
<b>12th Nov</b> Wed Afternoon	<b>PWG-DQ:</b> the DQ analysis framework <b>PWG-LF:</b> identified particle analysis	DQ: <b>53/R-044</b> LF: <b>I 3/2-005</b>
<b>13th Nov</b> Thu Morning	<b>PWG-CF:</b> Flow and femtoscopy (parallel sessions) <b>PWG-EM:</b> dileptons and the use of calorimetry	CF: <b>Virtual only</b> EM: <b>4/3-004</b>
<b>13th Nov</b> Fri Afternoon	<b>PWG-JE:</b> Jet analysis within O2 <b>PWG-UD:</b> ultra-peripheral and diffractive event analysis	JE: <b>Virtual only</b> UD: <b>4/3-004</b>
<b>14th Nov</b> Fri Morning	<b>Machine learning:</b> boosted decision trees and neural networks	ML: <b>I 3/2-005</b>

Don't worry: All sessions will be recorded and made available on zoom in case you miss them  
And: to accommodate as much as possible various time zones, we'll provide the recordings as quickly as possible (< 24 hours)



# The plan for the general hands-on sessions

- **Monday afternoon, first exercise: Getting started**

- In this exercise, we'll make very simple modifications to a simple example task and learn **how to run an analysis task over real Pb-Pb data** and handle helper tasks
- Rationale: this should be very simple and is **meant to be a 'hello world' session'**

- **Monday afternoon, second exercise: Calculation of an efficiency**

- In this part, we'll explore **simulated Pb-Pb data** and will introduce the concept of links between tables.
- **Target: calculation of an efficiency for pions, kaons and protons**
- Sets the stage for the PID talks on Tuesday morning

- **Monday afternoon, third exercise: Two-particle correlation analysis**

- As a last point, we will elaborate on **how to write a data model** and **how to use derived data**
- **Target: first steps of a two-particle correlation analysis**
- This exercise will contain an example derived data file as well as an illustration of the large gains due to that

- **Tuesday afternoon, fourth exercise: Compile-time polymorphism**

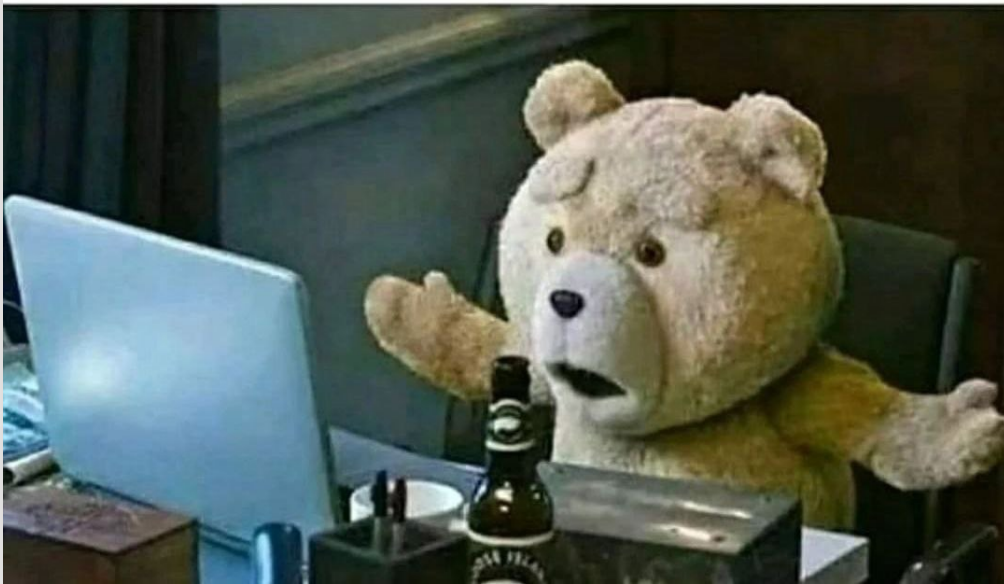
- Here, we'll teach you **good coding tips to design a clean and fast analysis task** without code duplication
- This session is new and strongly recommended: suggested by users!



# Support venues: in general and for the tutorial

don't hesitate to ask for help!

## Me When I Copy The Exact Same Code From Tutorial And It Doesn't Work



## Communicate! Mattermost®

How and where can I communicate about analysis?

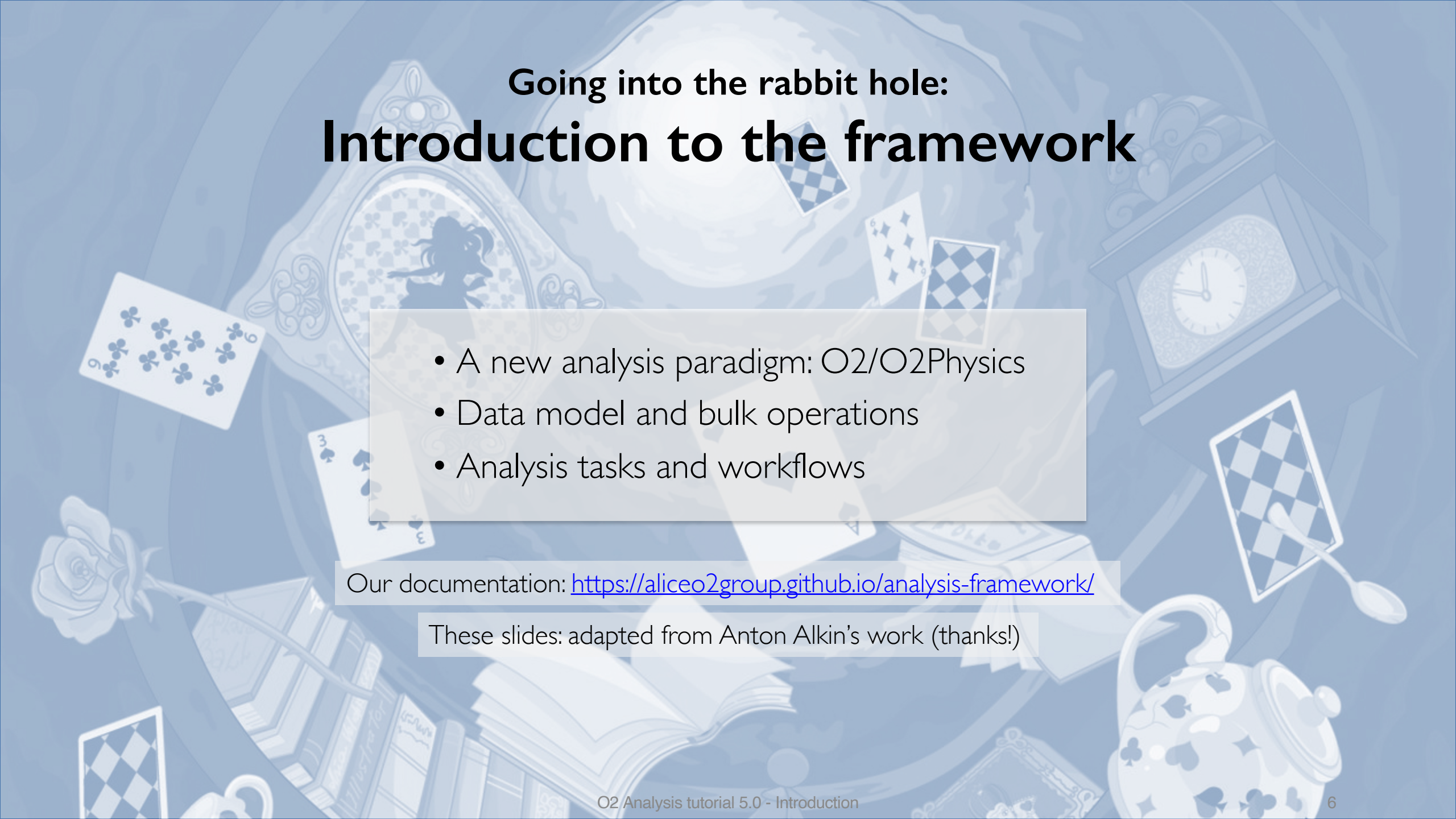
- [O2 analysis](#): general support
- [O2 hyperloop operations](#): running trains
- [O2 analysis announcements](#): general news
- [O2 analysis tutorial](#): help with this tutorial

Legacy communications, grid issues:

- [alice-project-analysis-task-force@cern.ch](mailto:alice-project-analysis-task-force@cern.ch)



[O2 framework documentation](#)

The background of the slide is a light blue illustration with a whimsical, Alice in Wonderland theme. It features a large, ornate clock on the right, a teapot with floral patterns at the bottom right, a playing card (the Queen of Hearts) on the left, and a small figure of a girl (Alice) in the center. The overall style is soft and artistic, with a focus on the clock and the playing cards.

# Going into the rabbit hole: Introduction to the framework

- A new analysis paradigm: O2/O2Physics
- Data model and bulk operations
- Analysis tasks and workflows

Our documentation: <https://aliceo2group.github.io/analysis-framework/>

These slides: adapted from Anton Alkin's work (thanks!)



# A new analysis paradigm

- AliPhysics: object-oriented, arrays of structures-based
  - ultimately: restrictive for analysis speed!
- Major switch to **arrow tables** <https://arrow.apache.org/>
  - structures of arrays-based, enormous processing speed unlocked!



# A new analysis paradigm

- AliPhysics: object-oriented, arrays of structures-based
  - ultimately: restrictive for analysis speed!
- Major switch to **arrow tables** <https://arrow.apache.org/>
  - structures of arrays-based, enormous processing speed unlocked!
- Cross-language development platform for in-memory data
- Columnar memory layout → optimized for bulk operations
- Native vectorized optimization of analytical data processing
- Supports zero-copy reads for lightning-fast data access
- Implementation in O2 developed by Giulio et al.
- All data is organized into split (but linked) tables





# A new analysis paradigm

- AliPhysics: object-oriented, arrays of structures-based
  - ultimately: restrictive for analysis speed!



- Major switch to **arrow tables** <https://arrow.apache.org/>
  - structures of arrays-based, enormous processing speed unlocked!

- Cross-language development platform for in-memory data
- Columnar memory layout → optimized for bulk operations
- Native vectorized optimization of analytical data processing
- Supports zero-copy reads for lightning-fast data access
- Implementation in O2 developed by Giulio et al.
- All data is organized into split (but linked) tables
- Root files are still the I/O backend of the new framework!
- Data is stored in multiple TTrees in TFiles
- O2 AODs (AO2Ds) can be inspected with the browser
- Tables are read from columnar trees during the analysis



# A new analysis paradigm

- AliPhysics: object-oriented, arrays of structures-based
  - ultimately: restrictive for analysis speed!



- Major switch to **arrow tables** <https://arrow.apache.org/>
  - structures of arrays-based, enormous processing speed unlocked!

- Cross-language development platform for in-memory data
- Columnar memory layout → optimized for bulk operations
- Native vectorized optimization of analytical data processing
- Supports zero-copy reads for lightning-fast data access
- Implementation in O2 developed by Giulio et al.
- All data is organized into split (but linked) tables
- Root files are still the I/O backend of the new framework!
- Data is stored in multiple TTrees in TFiles
- O2 AODs (AO2Ds) can be inspected with the browser
- Tables are read from columnar trees during the analysis

→ ok, but what are tables, David?

Show me what this means...

Collision table	Vertex Z
Row 0	6.52
Row 1	1.85
Row 2	-3.73

Track table	Collision index	pT	$\phi$	$\eta$
Row 0	0	1.75	0.02	-0.51
Row 1	0	0.38	1.32	0.32
Row 2	1	0.92	-0.75	0.44
Row 3	1	2.63	0.66	-0.01
Row 4	1	1.65	-0.23	-0.14
Row 5	1	1.32	0.62	0.09
Row 6	2	0.21	1.43	0.30





# A new analysis paradigm

- AliPhysics: object-oriented, arrays of structures-based
  - ultimately: restrictive for analysis speed!



- Major switch to **arrow tables** <https://arrow.apache.org/>
  - structures of arrays-based, enormous processing speed unlocked!

- Cross-language development platform for in-memory data
- Columnar memory layout → optimized for bulk operations
- Native vectorized optimization of analytical data processing
- Supports zero-copy reads for lightning-fast data access
- Implementation in O2 developed by Giulio et al.
- All data is organized into split (but linked) tables
- Root files are still the I/O backend of the new framework!
- Data is stored in multiple TTrees in TFiles
- O2 AODs (AO2Ds) can be inspected with the browser
- Tables are read from columnar trees during the analysis



→ ok, but what are tables, David?

Show me what this means...

Collision table		Vertex Z	
Row 0		6.52	2 tracks
Row 1		1.85	4 tracks
Row 2		-3.73	1 track

Track table	Collision index	pT	$\phi$	$\eta$
Row 0	0	1.75	0.02	-0.51
Row 1	0	0.38	1.32	0.32
Row 2	1	0.92	-0.75	0.44
Row 3	1	2.63	0.66	-0.01
Row 4	1	1.65	-0.23	-0.14
Row 5	1	1.32	0.62	0.09
Row 6	2	0.21	1.43	0.30

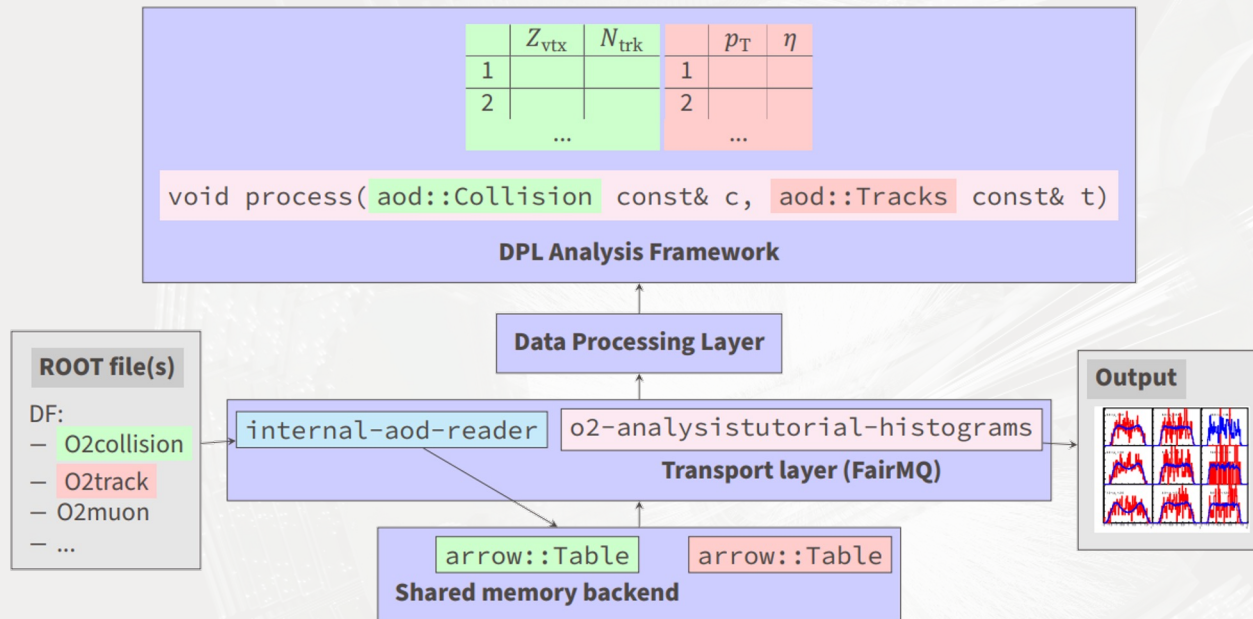
Diagram showing relationships: Row 0 and Row 1 of the Collision table point to Row 0 of the Track table. Row 2 of the Collision table points to Row 2 of the Track table. Row 3 of the Collision table points to Row 4 of the Track table.

Relationships done via index!

Processed/correlated quickly with arrow  
(i.e., phew, we don't have to do that)

- Reversed access hierarchy: e.g. Tracks **refer** to Collisions, previously Collisions **contained** Tracks

# The implementation in practice: from input to output!



- Operation based on collections of tasks that exchange, re-utilise and produce **tables in shared memory**
- Def.: **workflows** are a collection of tasks that operate to get you your results!
  - Service tasks: event selection, PID, ...
  - Analysis tasks: spectra, 2-particle correlation, ...
- Data streaming on demand allows for optimised execution!

- The upside: **analysis tasks still exist** as such, and still follow **a standard recipe**!
- The usual “event loop” approach is possible! But there are better, faster ways...
- **Declarative tools are provided** so that **explicit (imperative) loops are not required**
  - This is much better! Please use this whenever possible → more in the hands-on sessions
  - Further tools are continuously being developed on-demand → better flexibility, performance, and more!



# The data model: processing and organization

Track table	Collision index	pT	$\phi$	$\eta$
Row 0	0	1.75	0.02	-0.51
Row 1	0	0.38	1.32	0.32
Row 2	1	0.92	-0.75	0.44
Row 3	1	2.63	0.66	-0.01
Row 4	1	1.65	-0.23	-0.14
Row 5	1	1.32	0.62	0.09
Row 6	2	0.21	1.43	0.30

## Data processing basics

- **Dataframe**: a self-contained unit of processing (following Run 3 requirements)
- Tasks process one **Dataframe** at once and send out their outputs **per-Dataframe**
- **Data is immutable**, Tasks can only create new Tables or histograms, but not modify existing ones
- **Flattened data structures** allow to leverage **memory streaming**

## Data structure organization

- Arrays (referred to as **Columns**) are organized in **Tables**
- Each **Table row corresponds to a physical entity**: a Collision, a Track etc.
- Tables may relate to other **Tables** through **Index Columns** by referring to concrete row numbers
- **Tables** that correspond row-to-row and have a same number of rows can be **Joined**
- **Tables** that correspond row-to-row but have a different number of rows can be tied through an **Index Table**

# Column types in our framework

	X	$\alpha$	$f(X, Z, m)$	Index	$Z = X \sin \alpha$
1				2	
2				3	
	Static Arrow::Array (type, type[N], vector<type>)		Dynamic lambda function not stored in memory calculated on demand	Index Arrow::Array (int, int[2], vector<int>)	Expression Arrow::Array created in memory with Gandiva

- Static column:** straightforward, stored in memory, use **.getter()** method to get content
  - Basic data container, can be used in Expression column definitions
- Dynamic column:** accessed with **.getter(args)**, can have (optional) arguments
  - C++ [lambda function](#): fully general method, cannot be used in Expression column definitions
- Index column:** used to refer to other tables
  - numerical index accessible via **.getterId()** or **.getterIds()** (slice, array)
  - actual de-reference accessed via **.getter()**
  - can be typecast into a derived type (join, etc) via **.getter\_as<SomeType>()**
- Expression column:** any derived quantity directly calculable from static column content
  - Behaves just like static columns once created!

# Declaring columns and tables

## Columns

Regular `DECLARE_SOA_COLUMN(Name, getter, type);`

Index      Simple `DECLARE_SOA_INDEX_COLUMN(Origin, getter);`

            Slice `DECLARE_SOA_SLICE_INDEX_COLUMN(Origin, getter);`

            Array `DECLARE_SOA_ARRAY_INDEX_COLUMN(Origin, getter);`

            Self self-versions of all three

Dynamic `DECLARE_SOA_DYNAMIC_COLUMN(Name, getter, Lambda);`

Expression `DECLARE_SOA_EXPRESSION_COLUMN(Name, getter, type, expression);`

## Tables

Regular `DECLARE_SOA_TABLE(Name, Origin, Descr, Column1, Column2, ...);`

Extended `DECLARE_SOA_EXTENDED_TABLE(Name, Base, Descr, ExprCol1, ExprCol2, ...);`

Index `DECLARE_SOA_INDEX_TABLE(Name, Key, Descr, IndexCol1, IndexCol2, ..);`

- **Columns**: are the actual 'variables' of relevance
- **Tables**: collections of columns

→ More in the hands-on!



# Looping over a table: using iterators

- Every table has an automatically defined iterator: `soa::Table::iterator`
- An [iterator](#) is used to access table content:

```
for (auto const& track : tracks) {  
    histogram.fill(track.pt());  
}
```

- An iterator can be used to access individual columns by calling corresponding getters
  - roughly [equivalent to an object in the old framework](#) - collision, track, etc.
  - The iterator can be [incremented](#) and [decremented](#),
  - The iterator can be [moved](#) to a certain row
  - The iterator can be [copied](#) and [compared](#) to other iterators (via `!=`)
- Tables that contain a `soa::Index<>` column have access to enumerating methods:
  - `.index()`, `.globalIndex()`, `.filteredIndex()`

# Looping over a table: using iterators

- Every table has an automatically defined iterator: `soa::Table::iterator`
- An `iterator` is used to access table content:

```
for (auto const& track : tracks) {  
    histogram.fill(track.pt());  
}
```



- **const**: helps with compiler optimizations
- **&**: use the iterator by reference → faster!

- An iterator can be used to access individual columns by calling corresponding getters
  - roughly `equivalent to an object in the old framework` - collision, track, etc.
  - The iterator can be `incremented` and `decremented`,
  - The iterator can be `moved` to a certain row
  - The iterator can be `copied` and `compared` to other iterators (via `!=`)
- Tables that contain a `soa::Index<>` column have access to enumerating methods:
  - `.index()`, `.globalIndex()`, `.filteredIndex()`

# Bulk operations: what are those?

From commerce/trading:

- “Bulk operations are actions that are performed on a large scale.”



...That's actually *not that far from what we mean* here!

- Bulk operations on flat tables can leverage modern CPU vectorization (SIMD: single instruction, multiple data)
  - Example: AVX-512 instructions operate on 512 bits (e.g. operate on 16x 32-bit quantities with one instruction!)
  - Significantly increased processing speed!
- dedicated tools available in O2/O2Physics for such operations using arrow tables

- Expression columns: their calculation proceeds in bulk, faster than other methods
- Filtering: tables can be filtered, producing smaller tables that obey certain criteria
  - Only the filtered subset of the table is available when processing data
- Partitioning: tables can be split ('partitioned') into different parts, each part obeying certain criteria
  - All parts are available when processing data



# Getting starting with bulk operations: defining expressions

```
• DECLARE_SOA_EXPRESSION_COLUMN(Pt, pt, float, nabs(1.f / aod::fwdtrack::signed1Pt));
```

- Filter `f = nabs(aod::track::eta) < etaMax && aod::track::pt > ptMin;`
- Filter `g = (aod::track::flags & someBit != 0);`
- Partition `negative = aod::track::signed1Pt < 0;`

- Almost [arbitrary C++ expressions with columns as operands](#) (arithmetic and bitwise operations)
  - More examples in the hands-on session!
- Can be used to define [filters](#) and [partitions](#), [expression](#) columns
- Several [math functions can be used](#) (absolute value, trigonometric, square/cubic root, log/exp etc.)
- Conditional expressions are available: `ifnode(condition, true_exp, false_exp)`
- These are recipes, actual computation only happens when needed
  - Where and how do these matter? → **they get used inside analysis tasks**

# Filtering and partitioning tables: the differences

## Filters

```
Filter f1 = nabs(aod::track::eta) < 1.f;  
Filter f2 = nabs(aod::track::dcaZ) < 1.f;  
void processA(Filtered<Tracks>  
              const& tracks)  
void processB(  
    Filtered<Join<Tracks,TracksDCA>>  
              const& tracks)
```

- Filters are combined together and automatically applied to compatible tables
- One cannot simultaneously access full table and filtered part
- Filtered<> tables will interact with grouping

## Partitions

```
Partition<Filtered<Join<Tracks,TracksDCA>>>  
    p = nabs(aod::track::dcaZ) < 1.f;  
void process(Tracks const& allTracks) {  
    for (auto const& track : allTracks) {}  
    for (auto const& trackInCut : p) {}  
}
```

- Partitions are independent<sup>a</sup>
- They will *not* interact with grouping
- One can access the full table independently of a Partition

<sup>a</sup>They can interact with Filters if defined over a Filtered<> type

→ More in the hands-on!

# Writing an analysis task

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

using namespace o2;
using namespace o2::framework;

struct ATask {
    HistogramRegistry histos{"histos", {}};

    init(InitContext const&) {}; //← configure, create specifics
    process(inform framework on what data you want to run) {
        //process data: this is where the magic happens
    };
};

WorkflowSpec defineDataProcessing(ConfigContext const& cfgc) {
    return WorkflowSpec{adaptAnalysisTask<ATask>(cfgc)};
}
```

- This looks similar to what we had before! [UserCreateOutputObjects](#) and [UserExec](#)
- ...not so fast: the [process](#) call has a very specific way of operating.
  - The framework needs to be told which data you need! (minimalistic = better)
  - This is what is called “[subscription](#)”



# Looping over all tracks

```
struct ATask {  
    HistogramRegistry histos{"histos", {}};  
  
    init(InitContext const&) {  
        histos.add("hPhi", "hPhi", kTH1D, {{100, 0., o2::constants::math::TwoPI}});  
    };  
    process(aod::Tracks const& tracks) {  
        for (auto const& track : tracks) {  
            histos.fill(HIST("hPhi"), track.phi());  
        }  
    };  
};
```

- This is a simple example in which we **loop over all tracks!**
- However, note that the tracks will not be divided in tracks belonging to separate events.
  - This loops over all tracks, **literally!**

# Looping over all tracks, retaining knowledge of events

```
struct ATask {
    HistogramRegistry histos{"histos", {}};

    init(InitContext const&) {
        histos.add("hPhi", " hPhi ", kTH1D, {{100, 0., o2::constants::math::TwoPI}}});
        histos.add("hEvCount", "hEvCount", kTH1D, {{1, 0., 1.}}});
    };

    process(aod::Collision const& collision, aod::Tracks const& tracks) {
        histos.fill(HIST(" hEvCount"), 0.5);
        for (auto const& track : tracks) {
            histos.fill(HIST("hPhi"), track.phi());
        }
    };
};
```

- Here, by specifying the iterator to collisions (`collision`) first, before tracks, we are telling the framework that we'd like to get the process function called for every event
- The tracks available at each call will only belong to the correct event - automatically

# Defining axes in a cleaner way: the AxisSpec

```
struct ATask {
    HistogramRegistry histos{"histos", {}};

    init(InitContext const&) {
        AxisSpec phiAxis = {100, 0.f, o2::constants::math::TwoPI};
        histos.add("hEvCount", " hEvCount", {kTH1D, {{1, 0.0f, 1.0f}}});
        histos.add("hPhi", "hPhi", {kTH1D, {phiAxis}});
    };
    process(aod::Collision const& collision, aod::Tracks &tracks) {
        histos.fill(HIST("hEvCount"), 0.5f);
        for (auto const& track : tracks) {
            histos.fill(HIST("hPhi"), track.phi());
        }
    };
};
```

- **Axis definitions:** can also be done in a simple way using **AxisSpec objects**
  - But what if I wanted to **configure the output** somehow? Number of bins, for instance?



# Configurables: configuring your task as you like

```
struct ATask {  
    Configurable<int> nBinsPhi{"nBinsPhi", 100, "N bins in phi histo"}  
  
    HistogramRegistry histos{"histos", {}};  
  
    init(InitContext const&) {  
        AxisSpec axisPhi = {nBinsPhi, 0.f, o2::constants::math::TwoPI};  
        histos.add("hEvCount", " hEvCount", {kTH1D, {{1, 0.0f, 1.0f}}});  
        histos.add("hPhi", "hPhi", {kTH1D, axisPhi});  
    };  
    process(aod::Collision const& collision, aod::Tracks &tracks) {  
        histos.fill(HIST("hEvCount"), 0.5f);  
        for (auto const& track : tracks) {  
            histos.fill(HIST("hPhi"), track.phi());  
        }  
    };  
};
```

- Any task can have [configurables](#) (type: int, float, bool or std::string) to define behaviour
- This example: we will be able to [define the number of bins](#) of the phi histogram

# ConfigurableAxis: directly configurable, best flexibility

```
struct ATask {
    Configurable<int> nBinsPhi{"nBinsPhi", 100, "N bins in phi histo"}
    ConfigurableAxis axisPt{"axisPt", {VARIABLE_WIDTH, 0.0f, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f,
    0.6f, 0.7f, 0.8f, 0.9f, 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f, 1.9f, 2.0f,
    2.2f, 2.4f, 2.6f, 2.8f, 3.0f, 3.2f, 3.4f, 3.6f, 3.8f, 4.0f}, "pt axis"};

    HistogramRegistry histos{"histos", {}};

    init(InitContext const&) {
        histos.axisPhi = {nBinsPhi, 0.f, o2::constants::math::TwoPI};
        histos.add("hEvCount", " hEvCount", {kTH1D, {{1, 0.0f, 1.0f}}});
        histos.add("hPhi", "hPhi", {kTH1D, axisPhi});
        histos.add("hPt", "hPt", {kTH1D, axisPt});
    };

    process(aod::Collision const& collision, aod::Tracks &tracks) {
        histos.fill(HIST("hEvCount"), 0.5f);
        for (auto const& track : tracks) {
            histos.fill(HIST("hPhi"), track.phi());
            histos.fill(HIST("hPt"), track.pt());
        }
    };
};
```

- For maximum configurability, including convenient handling of fixed vs variable binning: **ConfigurableAxis**

# Configuring to your liking: a full menu

**Simple** `Configurable<type> var{"name", defaultValue, "description"};`  
integers, **float**, **double**, **bool** and string

**1D Array** `Configurable<std::vector<type>> var{"name", defaultValue, "description"};`  
int32, **float**, **double** and string

**2D Array** `Configurable<Array2D<type>> var{"name", defaultValue, "description"};`  
int32, **float** and **double**

**Labeled Array** `Configurable<LabeledArray<type>> var{"name", defaultValue, "description"};`  
special 2D Array with string labels for rows and columns

**Custom struct** `Configurable<Custom> var{"name", defaultValue, "description"};`  
any **struct** with ROOT dictionary defined (with corresponding LinkDef file)

**Axis** `ConfigurableAxis var{"name", binning, "description"};`  
histogram axis specification



# Filtering tables for interesting information only

```
struct ATask {  
    Filter<aod::Tracks> ptFilter = track::pt > 1.0f;  
  
    Configurable<int> nBinsPhi{"nBinsPhi", 100, "N bins in phi histo"}  
    ConfigurableAxis axisPt{"axisPt", {VARIABLE_WIDTH, 0.0f, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f,  
0.6f, 0.7f, 0.8f, 0.9f, 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f, 1.9f, 2.0f,  
2.2f, 2.4f, 2.6f, 2.8f, 3.0f, 3.2f, 3.4f, 3.6f, 3.8f, 4.0f}, "pt axis"};  
  
    HistogramRegistry histos{"histos", {}};  
    init(InitContext const&){ /* omitted for brevity, check previous slides */ };  
  
    void process(aod::Collision const& collision, soa::Filtered<aod::Tracks> &tracks) {  
        histos.fill(HIST("hEvCount"), 0.5f);  
        for (auto const& track : tracks) {  
            histos.fill(HIST("hPhi"), track.phi());  
            histos.fill(HIST("hPt"), track.pt());  
        }  
    };  
};
```

- A certain task can operate solely on a **filtered sample of the full table** if desired!
- For instance: only **high- $p_T$  tracks** in this case (filtering happens automatically with a single line added!)
- This is superior to doing an imperative filter (if) because it uses a (declarative) database query: **faster!**

# Partitioning information: getting different samples

```
struct ATask {
    SliceCache cache;
    Partition<Tracks> leftTracks = track::eta < 0.0f;
    Partition<Tracks> rightTracks = track::eta >= 0.0f;

    Configurable<int> nBinsPhi{"nBinsPhi", 100, "N bins in phi histo"}
    ConfigurableAxis axisPt{"axisPt", {VARIABLE_WIDTH, 0.0f, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f, 0.6f, 0.7f, 0.8f,
0.9f, 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f, 1.9f, 2.0f, 2.2f, 2.4f, 2.6f, 2.8f, 3.0f, 3.2f,
3.4f, 3.6f, 3.8f, 4.0f}, "pt axis"};

    HistogramRegistry histos{"histos", {}};
    init(InitContext const&){ /* omitted for brevity, check previous slides */ };
    void process(aod::Collision const& collision, aod::Tracks const& tracks) {
        histos.fill(HIST("hEvCount"), 0.5f);
        auto ltThisColl = leftTracks->sliceByCached(aod::track::collisionId, collision.globalIndex(), cache);
        auto rtThisColl = rightTracks->sliceByCached(aod::track::collisionId, collision.globalIndex(), cache);
        for (auto const& track : ltThisColl) {
            histos.fill(HIST("hPhi"), track.phi());
            histos.fill(HIST("hPt"), track.pt());
        }
    };
};
```

- If you don't want to just filter, but select separate [parts](#) of the table: use [partitioning](#)
- For instance: [tracks with negative and positive eta](#) can be grouped!
  - WARNING: partitions are NOT grouped according to an iterator in the data subscription!
  - You need to do it manually: see example above + hands-on session
- Again, this is superior to doing an imperative filter (if) because it uses a (declarative) database query: [faster!](#)

# Filtering and partitioning, all at once

```
struct ATask {
    SliceCache cache;
    Filter<Tracks> ptFilter = track::pt > 1.0f;
    Partition<Tracks> leftTracks = track::eta < 0.0f;
    Partition<Tracks> rightTracks = track::eta >= 0.0f;

    Configurable<int> nBinsPhi{"nBinsPhi", 100, "N bins in phi histo"}
    ConfigurableAxis axisPt{"axisPt", {VARIABLE_WIDTH, 0.0f, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f, 0.6f, 0.7f, 0.8f,
0.9f, 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f, 1.9f, 2.0f, 2.2f, 2.4f, 2.6f, 2.8f, 3.0f, 3.2f,
3.4f, 3.6f, 3.8f, 4.0f}, "pt axis"};

    HistogramRegistry histos{"histos", {}};
    init(InitContext const&){ /* omitted for brevity, check previous slides */ };

    void process(aod::Collision const& collision, soa::Filtered<aod::Tracks> const& tracks) {
        histos.fill(HIST("hEvCount"), 0.5f);
        auto ltThisColl = leftTracks->sliceByCached(aod::track::collisionId, collision.globalIndex(), cache);
        auto rtThisColl = rightTracks->sliceByCached(aod::track::collisionId, collision.globalIndex(), cache);
        for (auto const& track : ltThisColl) {
            histos.fill(HIST("hPhi"), track.phi());
            histos.fill(HIST("hPt"), track.pt());
        }
    };
};
```

- If you filter and partition, then the **partitions** are auto-defined with logical ANDs with the filter
- In **colloquial language**: it's as if you apply the filter first and then divide the remaining data into parts
- Again, this is superior to doing an imperative filter (if) because it uses a (declarative) database query: **faster!**



# Subscribing to more complicated objects: joining tables

```
struct ATask {
    Configurable<int> nBinsPhi{"nBinsPhi", 100, "N bins in phi histo"}
    ConfigurableAxis axisPt{"axisPt", {VARIABLE_WIDTH, 0.0f, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f, 0.6f, 0.7f, 0.8f,
    0.9f, 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f, 1.9f, 2.0f, 2.2f, 2.4f, 2.6f, 2.8f, 3.0f, 3.2f,
    3.4f, 3.6f, 3.8f, 4.0f}, "pt axis"};

    HistogramRegistry histos{"histos", {}};

    init(InitContext const&){ /* omitted for brevity, check previous slides */ };

    void process(aod::Collision const& collision, soa::Join<aod::Tracks, aod::TracksExtras> const& myTracks) {
        histos.fill(HIST("hEvCount"), 0.5f);
        for (auto const& track : myTracks) {
            histos.fill(HIST("hPhi"), track.phi()); //property in Tracks
            // here, you will have access to track properties! TPC crossed rows, etc etc
        }
    };
};
```

- Tables of [equal size](#) can be [joined](#) and processed as one
- This is super common! For instance, any extra stuff like PID info, track info, etc will be in other tables
- See the [table reference guide](#) for more info

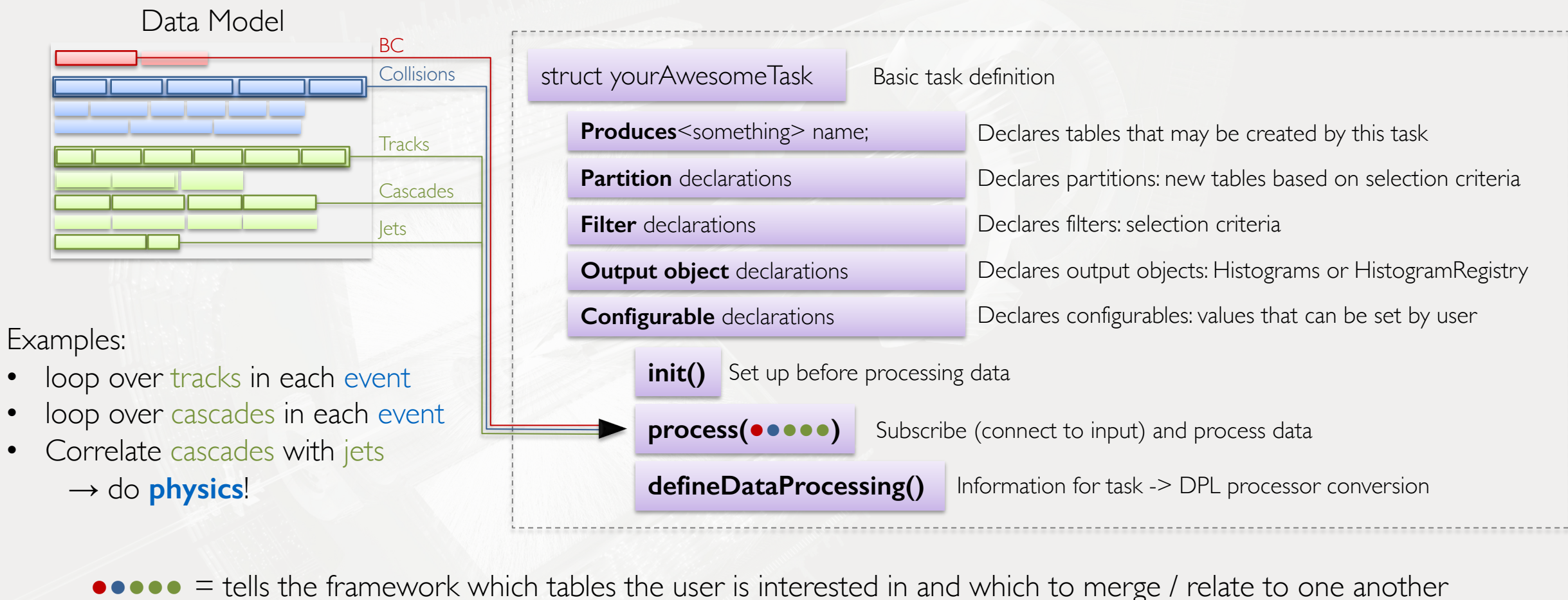
Examples in the hands-on sessions: it will be easier to follow in practice!

# Using multiple process functions

```
struct ATask {  
    (...)  
    void processOne(aod::Collision const& collision, aod::Tracks const& myTracks) {  
        // do something  
    };  
    PROCESS_SWITCH(ATask, processOne, "Do processing one", true);  
  
    void processTwo(aod::Collision const& collision, aod::Tracks const& myTracks) {  
        // do something else  
    };  
    PROCESS_SWITCH(ATask, processTwo, "Do processing two", false);  
};
```

- Process switches
  - Boolean configurable with a special purpose: enabling/disabling process functions
- Can only be set via JSON or on Hyperloop
- Since the task's inputs are determined by the process functions signatures, these switches can be used to control what inputs are required by the task (e.g. generator level information for MC input)
- Note that currently these switches are independent between the tasks and need to be consistently changed for each task in a workflow
- Note that currently all tasks need to know about the configuration of all other tasks in a workflow, the full configuration needs to be supplied to each entry

# In a nutshell: the general analysis task structure

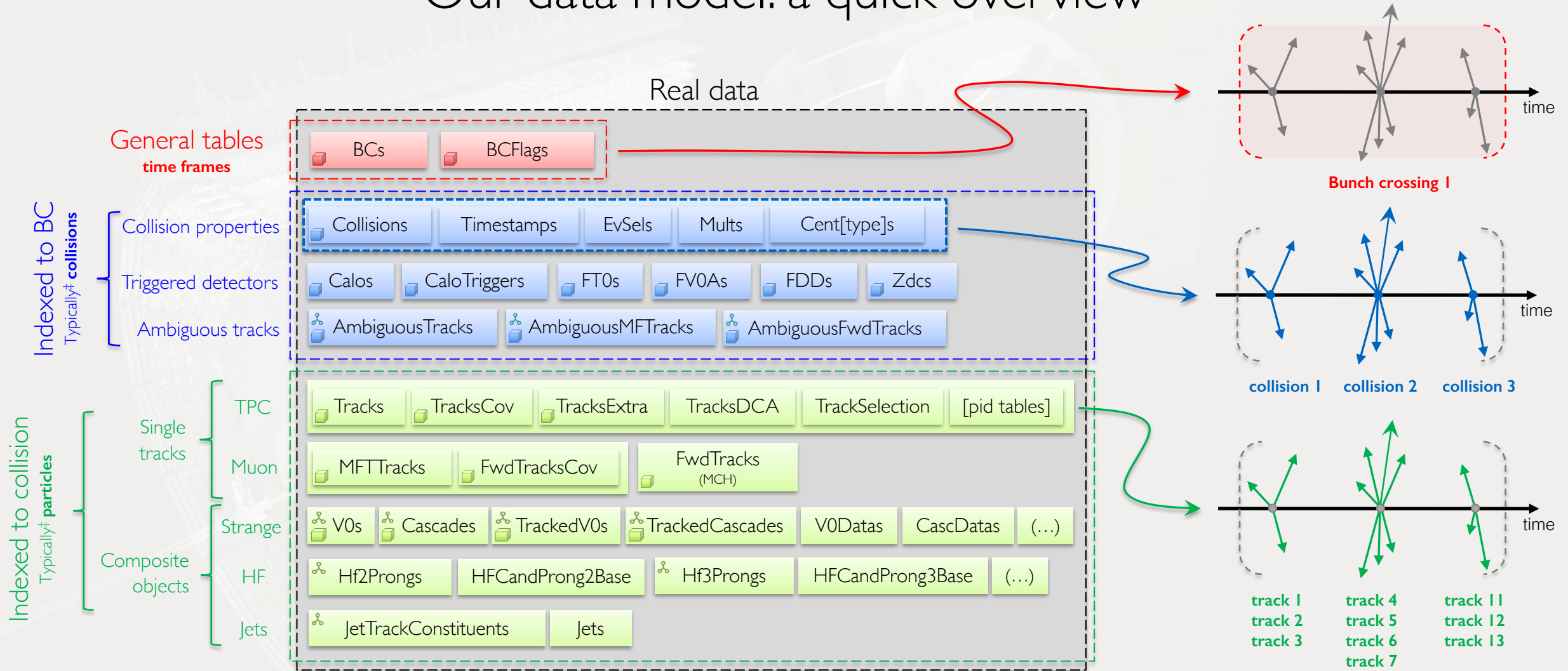


## Processing order:

- **init()** called first, then **process** functions called in sequence in the order their switches are declared
- At **init()** time, configurables are already set to their requested value



# Our data model: a quick overview



† simplified: some extended tables were omitted for brevity. Tracks are stored at the innermost update (TracksIU).

‡ typically → not always, e.g. ambiguous tracks belong to BCs because they're not assigned to specific collisions.



Table containing only indices  
(meant to link multiple tables)

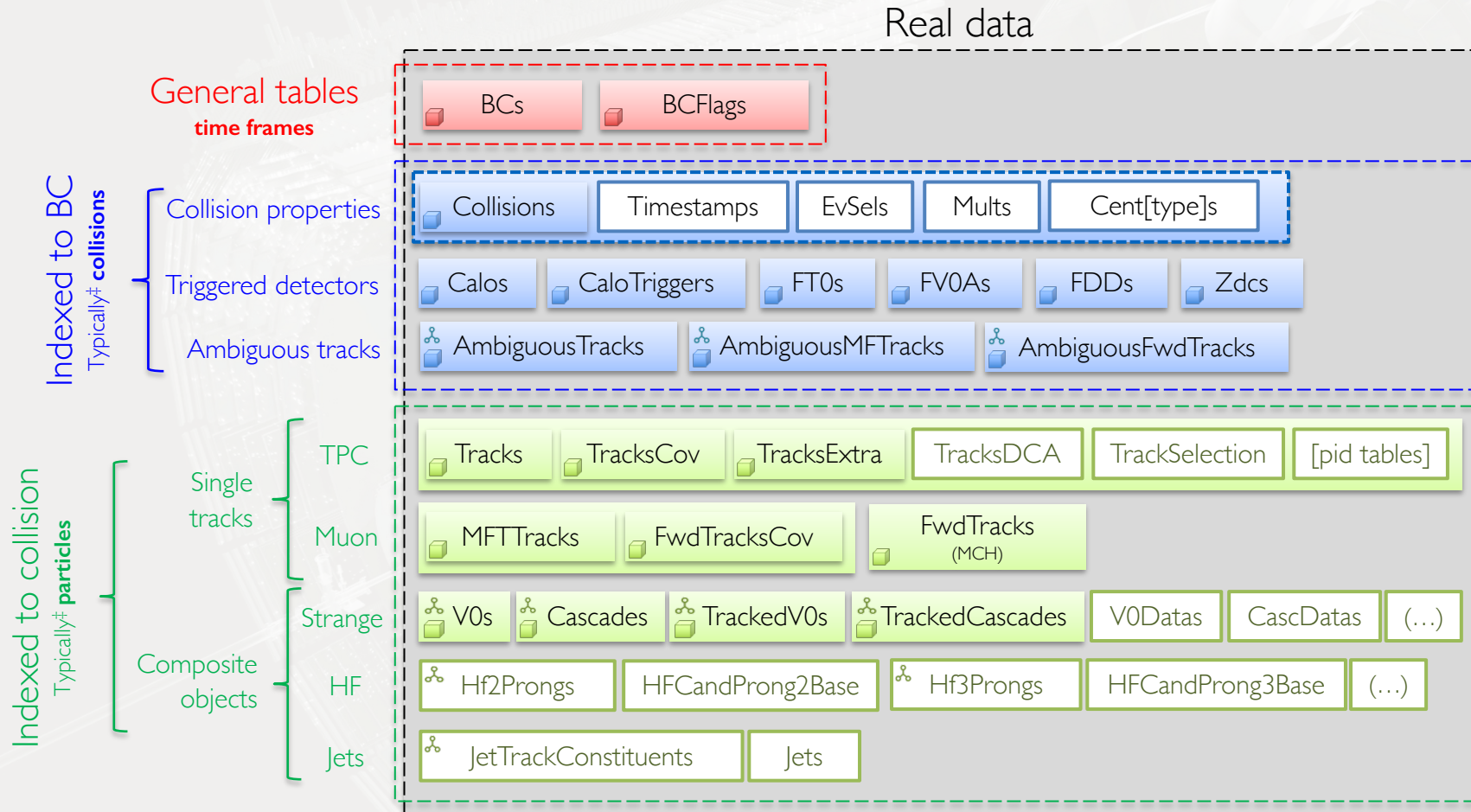


Tables saved in standard AO2Ds  
(others: generated on demand by helper tasks)



ALICE

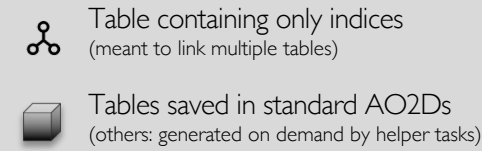
# Our data model: a quick overview



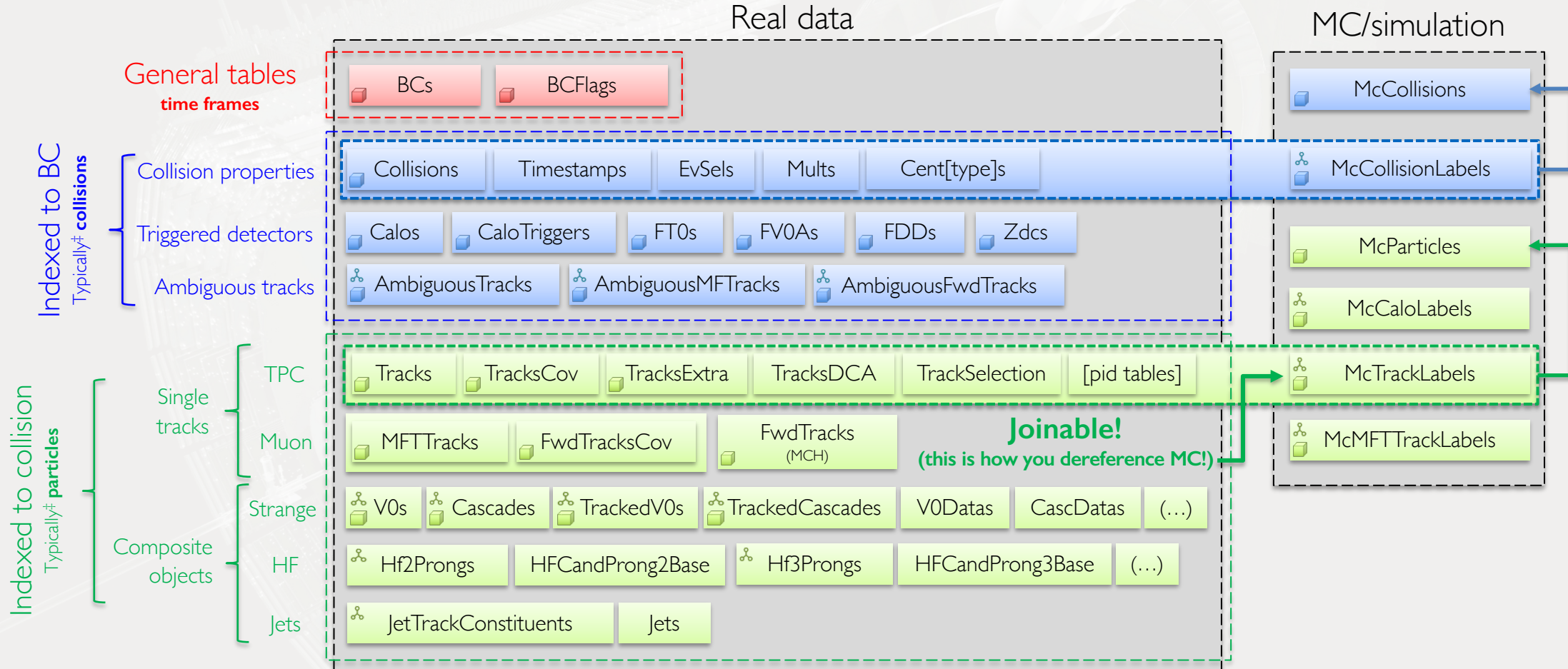
- Some tables are saved in standard AO2Ds, while some others are produced by general and PWG-specific tools to save disk space
- **No need to reinvent the wheel:** check if specific tools already exist for your purpose!

† simplified: some extended tables were omitted for brevity. Tracks are stored at the innermost update (TracksIU).

‡ typically → not always, e.g. ambiguous tracks belong to BCs because they're not assigned to specific collisions.





# Our data model: a quick overview



† simplified: some extended tables were omitted for brevity. Tracks are stored at the innermost update (TracksIU).

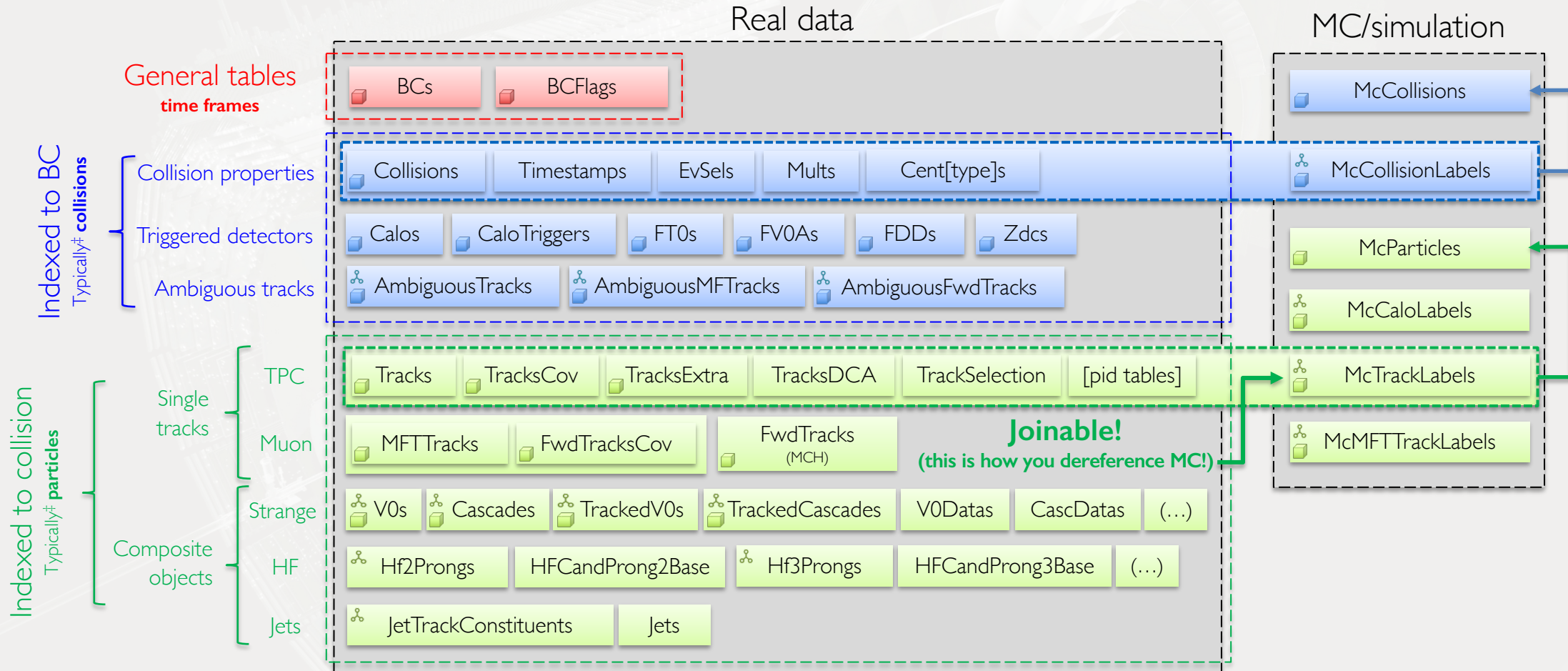
† typically → not always, e.g. ambiguous tracks belong to BCs because they're not assigned to specific collisions.

 Table containing only indices  
(meant to link multiple tables)

 Tables saved in standard AO2Ds  
(others: generated on demand by helper tasks)



# Our data model: a quick overview



$\dagger$  simplified: some extended tables were omitted for brevity. Tracks are stored at the innermost update (TracksLU).

$\dagger$  typically  $\rightarrow$  not always, e.g. ambiguous tracks belong to BCs because they're not assigned to a specific collision.

What if I want to create a table with track properties of my own?

Table containing only indices

Tables saved in standard AOZDS (others: generated on demand by helper tasks)



# Creating your own tables

or: the moment when things get really interesting

myTable.h

```
#include "Framework/ASoA.h"
#include "Framework/AnalysisDataModel.h"
namespace o2::aod {
namespace my_table {
DECLARE_SOA_COLUMN(MyValue, myValue, float, "myValue");
} //end myTable namespace
DECLARE_SOA_TABLE(MyTable, "AOD", "MYTABLE", my_table::MyValue);
}

struct ATask {
    Produces<aod::MyTable> thisTableHere;
    (...)
    process(aod::Collision const& collision, soa::Join<aod::Tracks, aod::TracksExtras> const& myTracks) {
        registry.fill(HIST("hCandidateCounter"), 0.5);
        for (auto const& track : myTracks) {
            registry.fill(HIST("phi"), track.phi()); //property in Tracks
            registry.fill(HIST("length"), track.length()); //property in TrackExtras
            thisTableHere( track.phi() + o2::constants::math::PI ); //this fills our new table!
        }
    };
};
```



This operation is flexible! We can then use the extra table for filtering (ultra fast), manipulating, etc and be very modular! In this case, this new table can be joined with tracks (same size)

# Creating tables of different types

**Produces**<DerivedTable> cursor;

- **Derived tables** are directly filled, row by row, by calling the cursor, created by Produces<> template. Table is only created after the filling task finishes.

**Spawns**<ExtendedTable> handle;

- User-defined **extended tables** need to be requested by adding Spawns<> template to the task. The table is created before the task is run and is accessible through the handle variable that acts as a pointer.

**Builds**<IndexTable> handle;

- User-defined **index tables** need to be requested by adding Builds<> template to the task. The table is created before the task is run and is accessible through the handle variable that acts as a pointer



# Derived table handling

- **Writing tables to disk**

- Any table that is accessible by its type can be written to disk at the end of processing by using:
  - `--aod-writer-keep` command line option (See docs for more options)
- This is mainly useful for storing skims and ML training data
- Tables are stored as ROOT trees

## Using tables in processing

- Any table that is accessible by its type and has been created by means of `Produces<>`, `Spawns<>` or `Builds<>` can be subscribed by other tasks in the workflow
- It behaves exactly as the tables that were read from AOD file and can be subjected to the same operations
- A typical usage is joining the data tables with those produced by helper tasks (e.g. track DCA, PID, track and event selection)

→ More in the hands-on!

# Running analysis in practice and assembling a workflow

## Helper tasks

- Various helper tasks, like [event](#) and [track selection, PID](#), etc., define corresponding tables
- These can be easily accessed in your task by including corresponding headers
- [PWGs maintain their own data model extensions](#) also defined in headers
- To use the outputs of the helpers tasks, [add them to the workflow](#)
- This is done by [adding the task's binary to the command](#) with | (called “pipe”)
- [Several tasks can be piped](#) together (note that full configuration needs to be provided to all entries)

```
o2-analysis-a --configuration json://file.json |\no2-analysis-b --configuration json://file.json |\no2-analysis-c --configuration json://file.json --other-option
```

→ More in the hands-on!

# A quick overview of general helper tasks (core service wagons)

a non-exhaustive list: your favorite PWG may have many more

Executable name	Function	HY wagon	
o2-analysis-event-selection-service	Provides timestamps to BCs (joinable with BCs) Provides event selection tables (joinable with Collisions)	<a href="#">link</a>	DPG talk later today
o2-analysis-multcenttable	Provides multiplicity tables (joinable with Collisions) Provides centrality tables (joinable with Collisions)	<a href="#">link</a>	
o2-analysis-qvector-table	Provides Q Vectors (joinable with Collisions)	<a href="#">link</a>	
o2-analysis-track-selection	Provides standard track selection criteria	<a href="#">link</a>	
o2-analysis-propagationservice (Aggregates propagation CPU)	Provides Tracks (tracks at the primary vertex) Provides/builds V0 information Provides/builds Cascade / KF Cascade / Tracked Cascade information	<a href="#">link</a>	PID talks tomorrow
o2-analysis-pid-tpc-service	Provides TPC <b>PID</b> (joinable with Tracks)	<a href="#">link</a>	
o2-analysis-pid-tof	Provides TOF <b>PID</b> (joinable with Tracks)	<a href="#">link</a>	
o2-analysis-pid-its	Provides ITS <b>PID</b> (joinable with Tracks)	<a href="#">link</a>	
o2-analysis-lf-strangeness-tof-pid	Provides TOF <b>PID</b> for V0s, Cascades (joinable: standard V0s/Cascades)	<a href="#">link</a>	

Nota bene: this has recently changed and **a lot of core service tasks were merged into fewer, much more optimized tasks**. These use **less memory and less CPU** to do the same and follow as much as possible **autodetection principles**, with a true autodetection of input data type still pending some framework developments.



# The O2Physics hints & tips shortlist



## General tips:

- Use **derived data** (standalone tables) → responsible resource consumption
- Use **declarative programming** whenever possible → faster processing
- Produce **minimal data** when populating tables → enable **large-scale trains**
- Use existing **common tools** when doing analysis → no need to reinvent the wheel
- Use **modularity** to ensure no unnecessary processing happens → enable **large-scale trains**
- Add **comments to code**: as much as needed, as little as possible → we're building code together

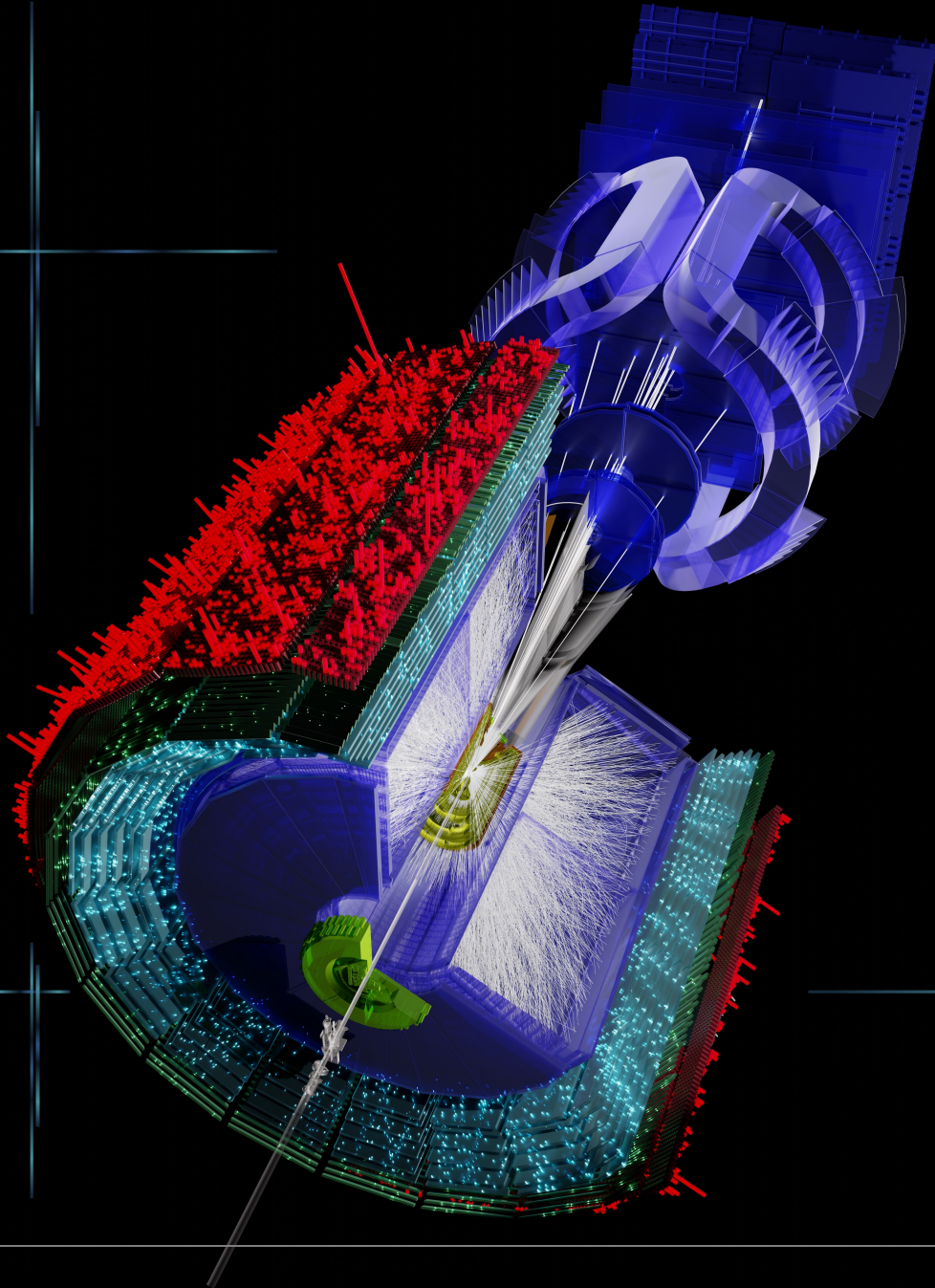
## Private execution tips:

- **Cache the CCDB** for faster testing, especially with unstable / high-latency connections
  - `export IGNORE_VALIDITYCHECK_OF_CCDB_LOCALCACHE=1`
  - `export ALICEO2_CCDB_LOCALCACHE=/home/daviddc/ccdb`
  - But: beware ignoring any CCDB updates as a consequence
- **Use pipelining** whenever processing heavy tasks for fast multi-core processing
  - See example this afternoon (instant gratification: `--pipeline=<device-name>:N` to run with N cpus)



See Nicolas' talk!

+ watch out for more useful information in the other lectures and hands-on sessions!



# Summary

The Analysis Framework contains a **wide set of basic tools**

A number of **declarative features** are already implemented

“Traditional” **object-like interface is provided** to iterate over tables

Work is ongoing to **simplify syntax** and **extend functionality**

Large effort by PWGs to create **dedicated frameworks**

Please use the **official documentation [1]**

- it has a lot of information already!
- and if it missing a piece, we need the feedback

[1] <https://aliceo2group.github.io/analysis-framework/>

*Thank you!*



A detailed 3D rendering of the ALICE detector at the Large Hadron Collider. The image shows the complex arrangement of various detector components, including the central barrel calorimeter, the forward calorimeters, and the particle tracking systems. The word "Backups" is overlaid on the central part of the detector.

# Backups

# In-place table handling: all inside your analysis task!

## Attaching **dynamic** columns on-the-fly

- Inside a process function it is possible to add extra **dynamic** columns to an existing table object
- ```
auto newTable = soa::Attach<OldType, DC1<>, DC2<>, ...>(oldTable);
```
- This is confined to a process function
- The **dynamic** columns need to be properly defined and bound to data columns in the table

## Attaching **expression** columns on-the-fly

- Inside a process function it is possible to add extra **expression** columns to an existing table object
- ```
auto newTable = soa::Extend<OldType, EC1, EC2, ...>(oldTable);
```
- This is confined to a process function
- The **expression** columns need to be properly defined
- The new columns are created in memory and can be used to apply **filters** or create **partitions**



In-place bulk operations: don't do `if` inside `for` loops if possible

## In-place filters and partitions

- Existing **table** objects can be filtered in-place inside a process function

- ```
auto filteredTable = soa::select(oldTable, aod::track::pt > 1.f);
```

- ```
Partition<OldType> part = aod::track::pt > 1.f;  
part.bindTable(oldTable);
```