ALICE Run3 Analysis Framework

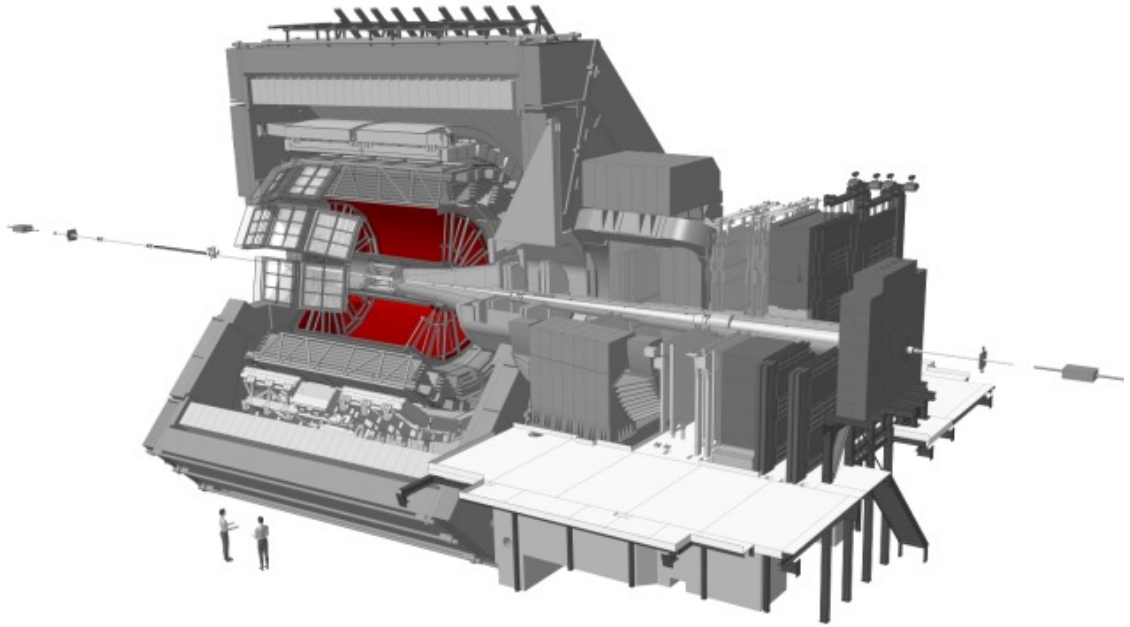
P. Hristov

HL-LHC Analysis Mini-Workshop

04.05.2021

A Large Ion Collider Experiment

Run3/4 goals



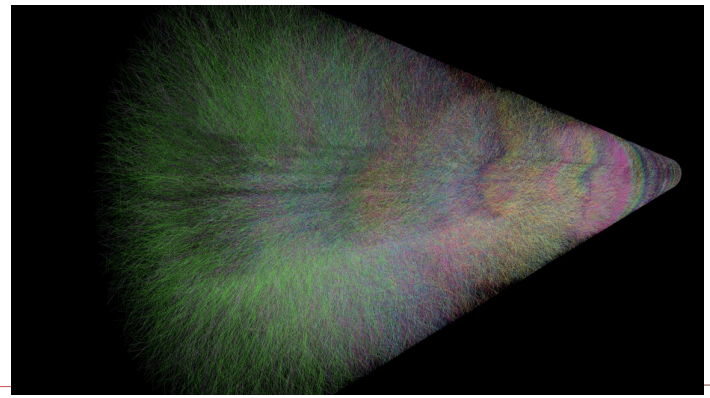
- ALICE: Dedicated experiment for heavy-ion physics
  - Study of strong interactions & QGP properties
  - Detection of low momentum particles, vertexing, PID
- Run3/4 main physics goals: rare signals with low S/B
  - dynamics of heavy flavour and quarkonia at very low-pT: thermalisation, recombination
  - vector mesons and low-mass di-leptons: chiral symmetry restoration, virtual thermal photons
  - precise measurement of light nuclei and hyper-nuclei: QGP nucleosynthesis, exotics
- Detector upgrade: ITS, TPC, MFT, FIT

Run 2: Triggered readout

- Pb-Pb Interaction Rate (IR) ~7-10 kHz, trigger rate < 1 kHz limited by **TPC** readout (<3.5 kHz) and bandwidth
- Collected luminosity (**L**): ~1 nb^{-1}

Run 3 + Run 4: Continuous readout , 10 ms timeframes

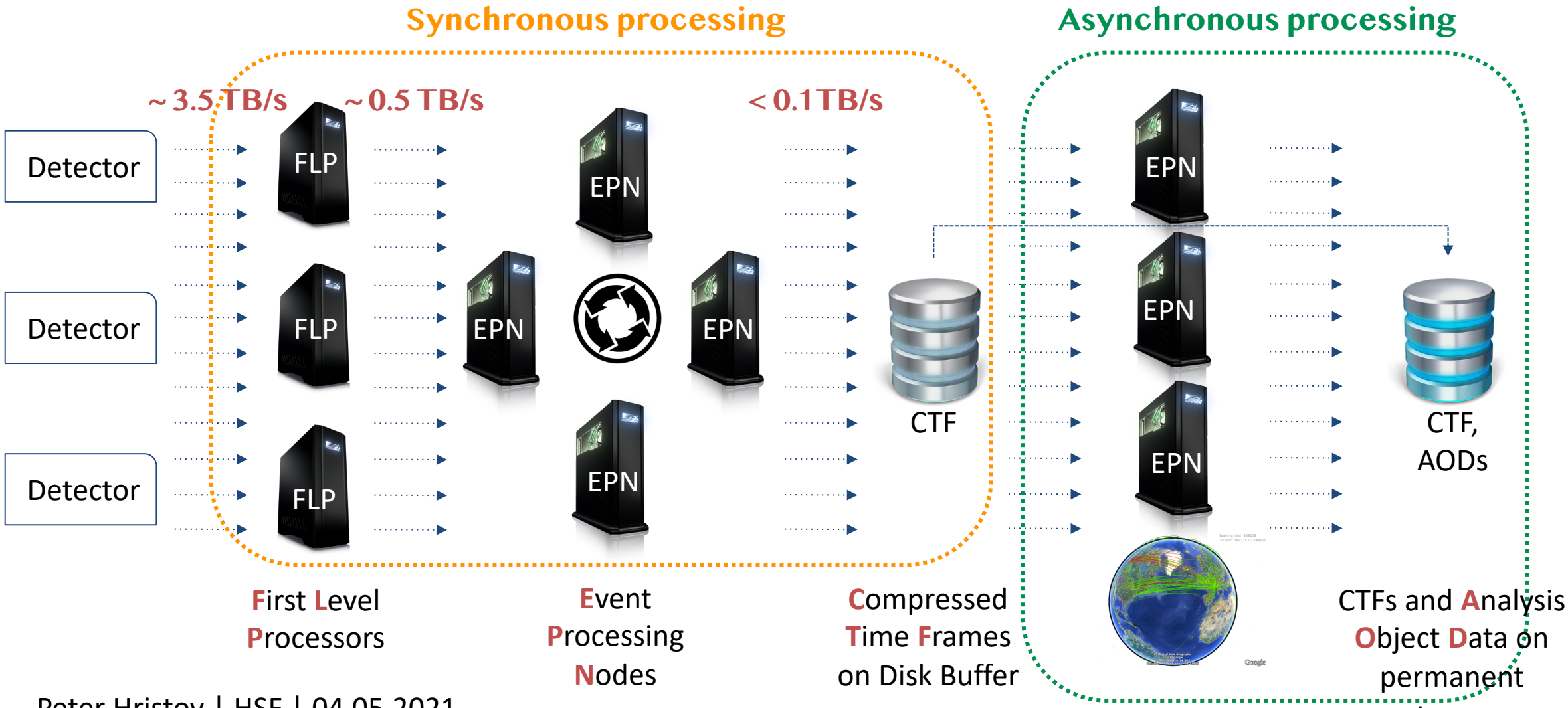
- LHC will deliver collisions at an IR of 50 kHz
- Goal: **L** ~6 nb^{-1} + 7 nb^{-1}



PbPb @50 kHz IR, 2 ms TF, TPC reconstructed tracks from different color-coded events

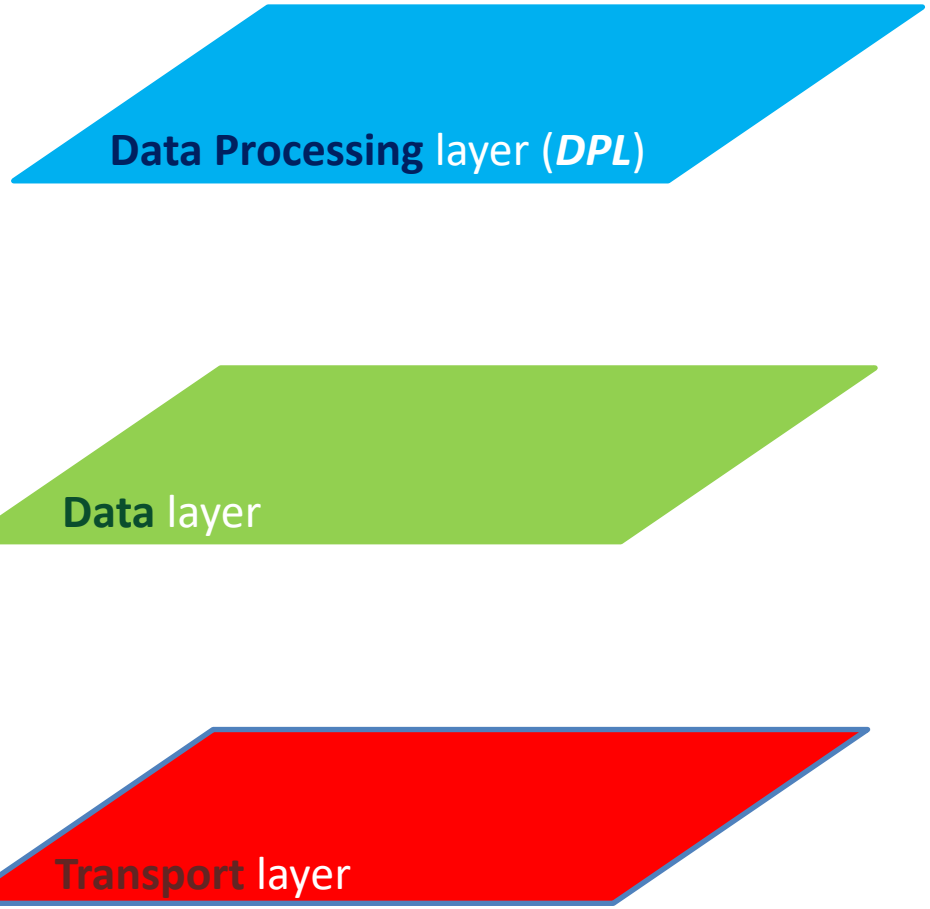
ALICE data processing in Run 3 + 4

Summary



O^2 processing model

ALICE Run 3 + 4 processing software (reconstruction) agnostic of where it is happening: synchronous phase, asynchronous phase, simulation…**analysis**



"Translator" of the user's computational problem in a low-level topology of devices exchanging messages

Declarative.
Reactive-like design (push data, don't pull).
Integration with the rest of the production system.

ALICE-specific description of the messages

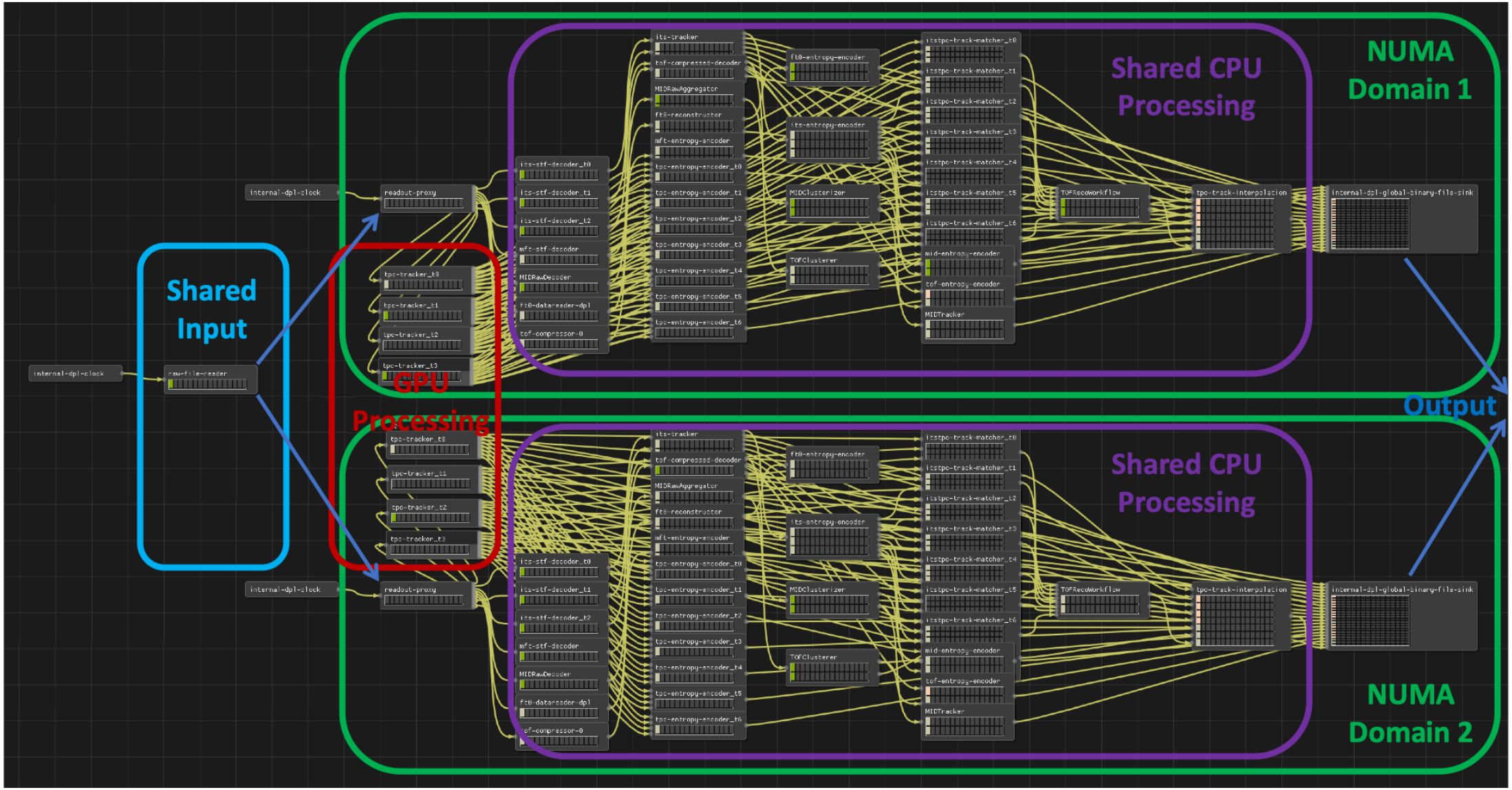
Header (extensible) + payload.
Computer language agnostic. Extensible. Suitable for GPU.
Multiple data formats and serialization methods: custom data structure GPU oriented; ROOT; Apache Arrow.

FairMQ message passing architecture

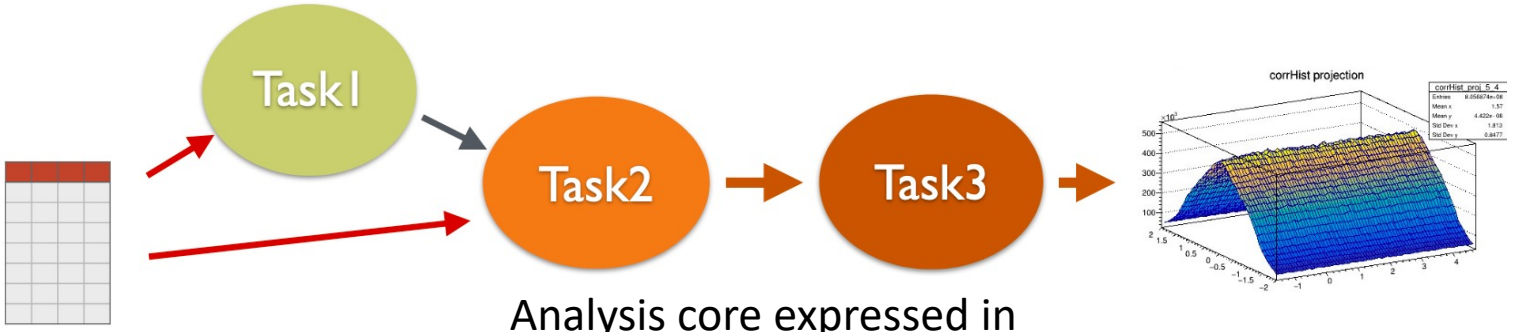
Standalone general processes (devices).
Shared memory backend.

O^2 processing model

ALICE Run 3 + 4 processing software (reconstruction) agnostic of where it is happening:
synchronous phase, asynchronous phase, simulation…**analysis**



Analysis Framework Overview



Data model for analysis based on **flat tables** arranged in a relational-database-like manner:

- minimise I/O cost
- improve vectorisation / parallelism

Apache Arrow hidden behind a classic C++ API

Analysis core expressed in the form of a **task**

- Successful from Run1/2
- Filters and selections
- Merging, concatenation of tables

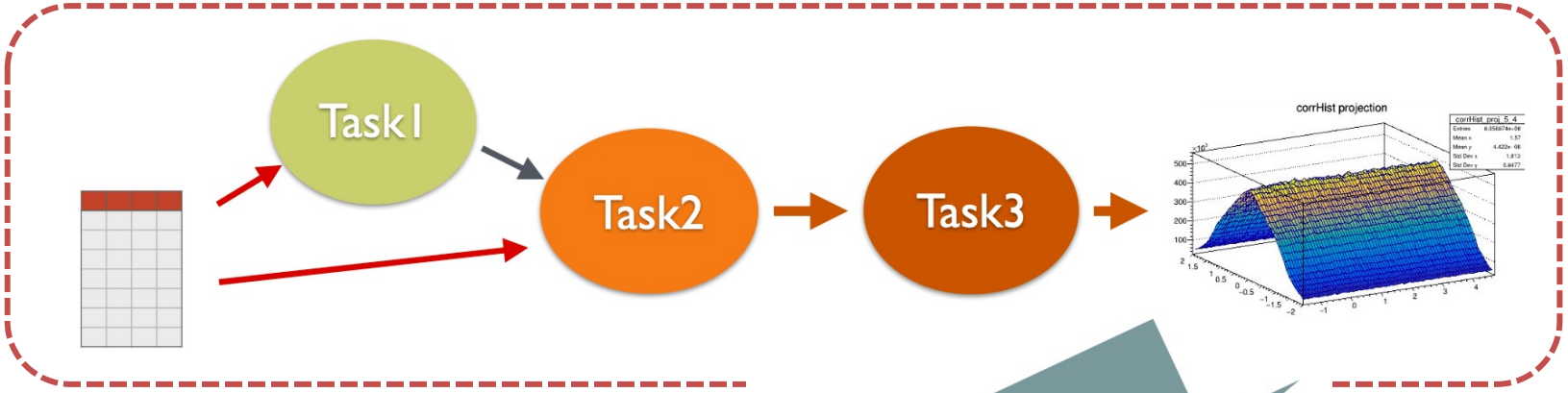
ROOT serialized output

O^2 Analysis model is

DECLARATIVE: the user will specify inputs, filters and outputs

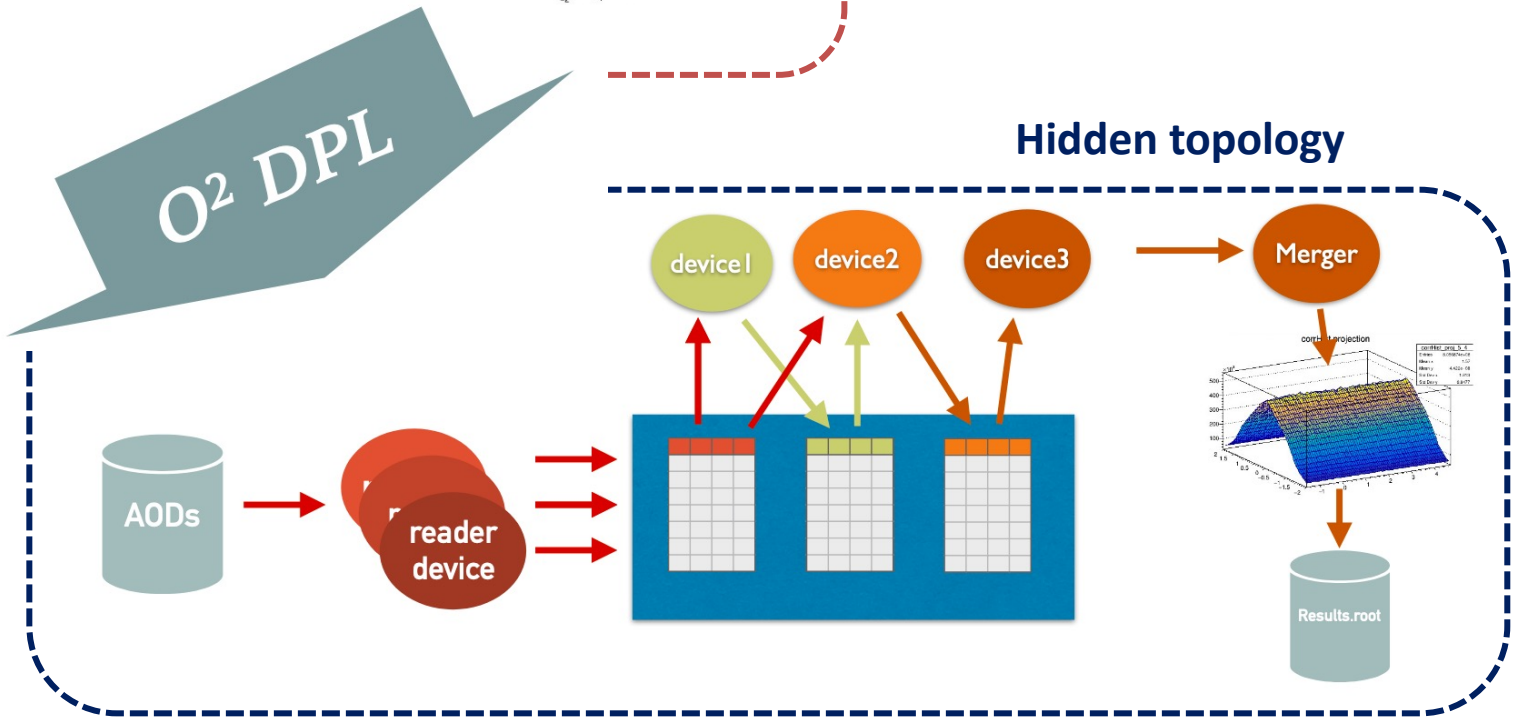
IMPERATIVE: the user will specify the processing algorithm

Analysis Framework Overview



User's responsibility

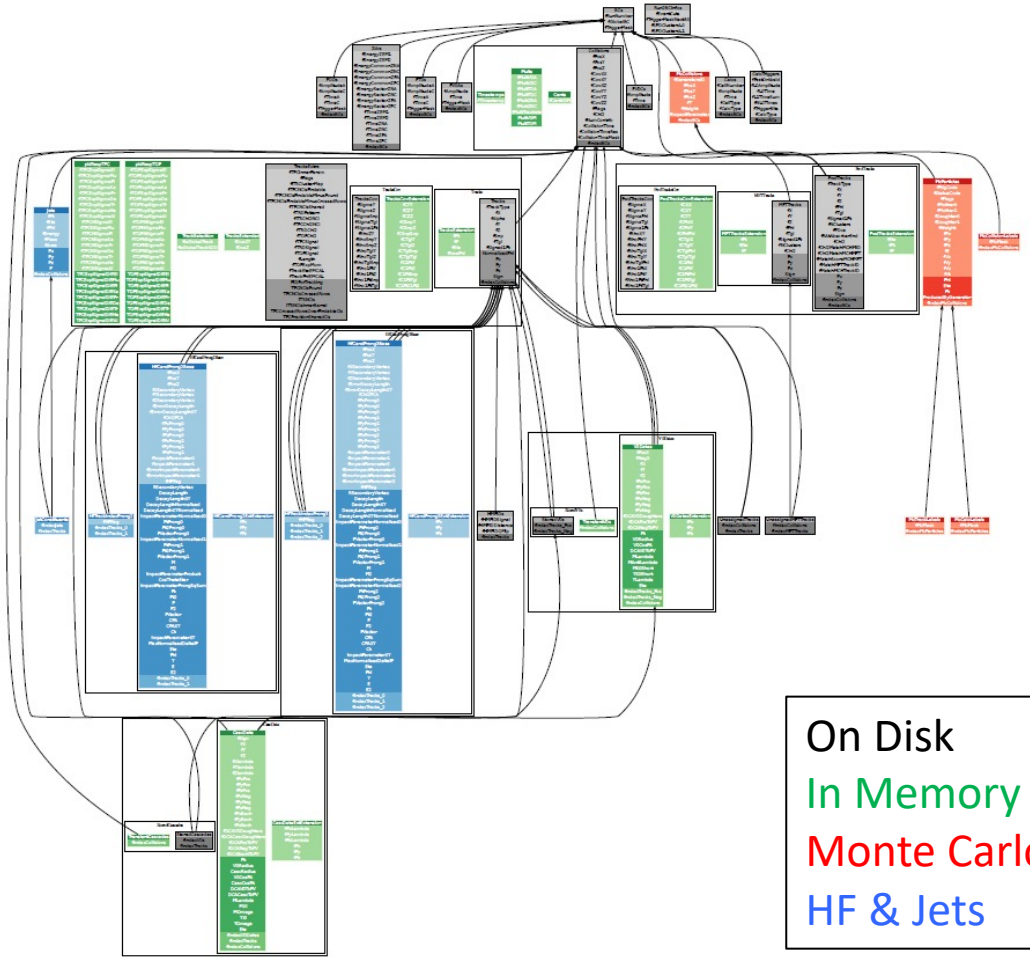
Data Processing Layer translates the implicit workflow(s) defined by physicists to an actual FairMQ topology of devices, injecting readers and merger devices, completing the topology and taking care of parallelism / rate limiting.



Data Model – Connected Tables

How the data is organized

- Structure-of-arrays instead of familiar from Run2 array-of-structures[a]
- Arrays[b] (referred to as Columns) are organized in logical Tables, corresponding to physical concepts of Collisions, Tracks etc.
- Tables may refer to other Tables through Index Columns that contain row numbers
- Tables that correspond row-to-row and have 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
- Significantly reduced size per collision
  - Factor 16 re Run 2 ESD | factor 5 re Run 2 AOD



[a] https://en.wikipedia.org/wiki/AoS_and_SoA
[b] https://arrow.apache.org/

Tables

Technical details

- A Table is defined as a unique C++ type, templated on Columns, themselves unique C++ types
- Underlying contiguous arrays in memory are immutable – no data is removed or copied in common operations such as Grouping, Filtering or Partitioning
- Tables can be read from or written into a ROOT TTree
- New Tables can be defined and created by users

```
namespace bc
{
        DECLARE_SOA_COLUMN(RunNumber, runNumber, int);
        DECLARE_SOA_COLUMN(GlobalBC, globalBC,
                                uint64_t);
        DECLARE_SOA_COLUMN(TriggerMask, triggerMask,
                                    uint64_t);
} // namespace bc

DECLARE_SOA_TABLE(BCs, "AOD", "BC", o2::soa::Index<>,
bc::RunNumber, bc::GlobalBC,
bc::TriggerMask);
using BC = BCs::iterator;
```

Columns

Technical details

Declaration

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

Index `DECLARE_SOA_INDEX_COLUMN(OriginTable,getter);`

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

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

Normal

- Named array
- Supported types: `int`, `float`, `double`, `bool`, `uint8_t`, arrays, e.g. `int[N]`
- `getter` returns a stored value

Index

- A Normal Column with `int` contents, containing row numbers referring to some other Table
- `getter` returns an accessor row of the indexed Table

Dynamic

- A function, defined on other Normal/Expression Columns with an arbitrary return type
- Is not backed by any memory contents

Expression

- A Normal Column that is calculated on the fly according to a recipe
- Does not exist in memory unless created by the framework or user

Tables

Operations

Declaration

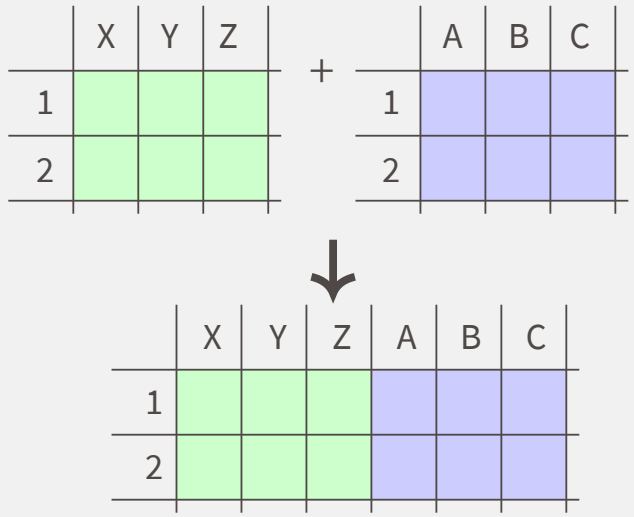
Normal `DECLARE_SOA_TABLE(Name, Origin, Descr,Column1, Column2, ...);`

Extended `DECLARE_SOA_EXTENDED_TABLE(Name,OrigTable,Descr,ExprColumn1, ExprCoulmn2, ...);`

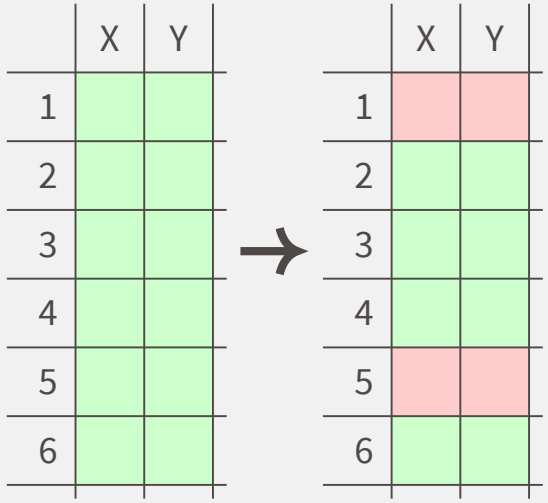
Index `DECLARE_SOA_INDEX_TABLE(Name, KeyTable, Descr, IndexColumn1, IndexColumn2, ..);`

Typical operations with tables

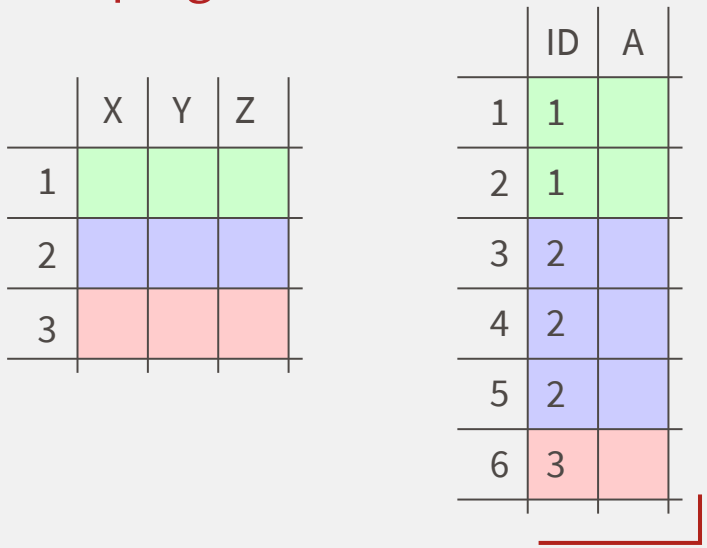
Join



Filter/Partition



Grouping



Tables

Stored, Extended and Index Tables

Stored

- Table types defined for tables, present in AOD files
- Not used directly, they represent the minimal set of stored quantities that fully define things like Tracks or Collisions

Extended

- Table types with Expression Columns added
- Can be predefined in Data Model (like Tracks) or user-defined
- User-defined Extended Tables need to be explicitly created
- Otherwise they are directly usable in Tasks

Index

- Special Table type that contains only Index Columns
- Need to be explicitly created
- Allow tying together non-joinable Tables (e.g. for accessing ZDC information tied to Collisions)

Analysis Task Interface: Factorization

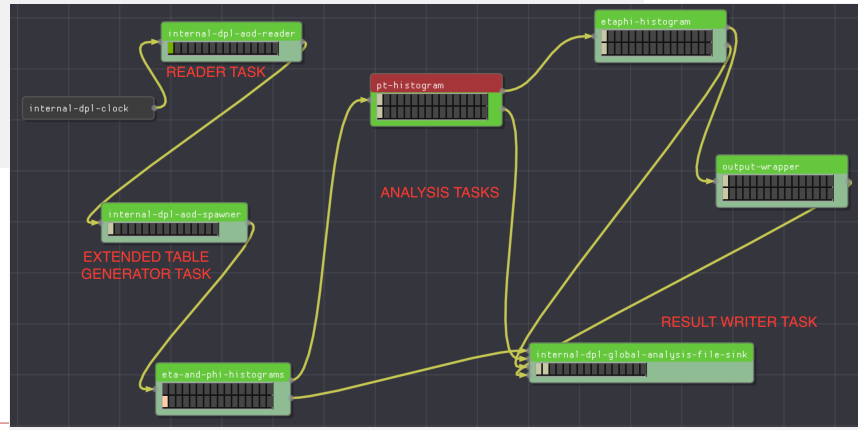
Tasks and workflows

Tasks

- Task is a C++ struct that has at least one of `init(InitContext&)` or `process(...)` methods declared
- Tasks can have Configurable data members, Filters and Partitions for subscribed data, Produces, Spawns and Builds declarations for new tables, OutputObj-wrapped analysis objects (any ROOT object) and whatever data members and methods required for analysis

Workflows

- Workflow is a collection of Tasks that subscribe to Tables and their modifications
- Workflow is automatically sorted to connect all inputs and outputs between user Tasks and Service Devices (like file reader or histogram writer)
- Workflows can be Merged and Serialized into json files
- Workflows can be configured in the command line or with a json file



Analysis Task Interface: Access to Data

Subscribing to Tables

Basic

```
process(TableA const& a, TableB const& b, ...)
```

Directly subscribes to pre-defined types TableA and TableB

```
process(TableA::iterator const& arow)
```

Directly subscribes to a *row* of TableA, meaning the process function will be invoked by the framework for each row of the table

Grouping

```
process(TableA::iterator const& arow, TableB const&, ...)
```

Subscribes to a TableA row and TableB, in the case when TableB has Index Column referring to TableA, it is *grouped* based on the index, meaning on each invocation of process method for a row of TableA, you only get a *subtable* of TableB that refers to that row; if the *first* argument of process is an iterator all other arguments need to be tables (not iterators)

Manipulation

```
process(Join<TableA,TableB>::iterator const& a, Filtered<Join<TableC,TableD» const& b, ...)
```

Subscribes to Tables A, B, C and D; A and B are joined (i.e. a single row now has access to columns from both), C and D are joined and then filtered (with a filter defined elsewhere in the task) and then grouped (if either C or D has index to either A or B); first argument can be filtered as well

Analysis Task Interface: Processing Data

Creating Tables

New or Extended Table

```
Produces<TableT> table;
```

Declares a table to be produced by the Task; `table` variable can be used to write a row of values (e.g. in a loop) `table(var1, var2, var3, ...)`

```
Builds<TableT> index;
```

Declares an Index Table to be built by the Task; the `index` variable is a pointer to the newly created table

```
Spawns<TableT> table;
```

Declares an Extended Table to be materialized by the Task; the `table` variable is a pointer to the newly created table

```
auto newtable = Extend<TableT, ColumnA, ColumnB,...>(table);
```

`newtable` Table is created, from `TableT table` instance, by adding expression columns calculated on the fly with a corresponding function

```
auto newtable = Attach<TableT, ColumnA, ColumnB,...>(table);
```

`newtable` Table is created, from `TableT table` instance, by adding dynamic columns

Analysis Task Interface: Processing Data

Task Output

Tables

- All Produced, Extended and Index Tables created by the Task are added to its outputs automatically
- This means that they can be saved into files or consumed by other Tasks, that subscribe to these Tables
- Tables are *the primary way* of information exchange between Tasks – the actual data exists in shared memory and *is not copied*
- The transient Tables, e.g. results of Filter, Partition or application of `Extend<>` are not propagated

Analysis Objects

- Analysis Objects can we added to Task outputs by wrapping them with `OutputObj<>`
- Most ROOT objects can be wrapped this way, including containers like `TList`
- HistogramRegistry is being developed[a] to offload histograms/objects creation and management to analysis framework itself, that will allow optimizations

[a] Several solutions already exist – they are under active development and will be eventually merged into one

Analysis Task Interface: Processing Data

Task Configuration

Declaration

Simple type

```
Configurable<float> pt_cut{"pt_cut",0.5,"Lowest pt"};
```

this defines named `pt_cut` variable, implicitly convertible to `float` that can be used in normal expressions *and* Expressions that define Filters and Partitions; the actual value can be set *externally*, either with command line option or in JSON configuration for the workflow

Compound type

```
Configurable<SomeStruct> some_cut{"some_cut",{0.23, 3, "str"},"cut object"};
```

this defines named `some_cut` variable, implicitly convertible to `SomeStruct` object with default settings; these settings can be modified *externally* with JSON configuration for the workflow

CCDB access

Task can access CCDB information through the Service mechanism

```
Service<o2::ccdb::BasicCCDBManager> ccdb;
void init(InitContext&) {
    auto p =
       ccdb->getForTimeStamp<Parametrization>("Analysis/PID/TPC/BetheBloch", -1);
}
```

Analysis Task Interface: Bulk Operations

Expressions

Operation "recipes"

- C++ expressions ("formulas") using Normal and other Expression Columns as variables[a]
- Unlike normal expressions, only define a *recipe* for the operations to be performed
- Can be used to define derived Expression Columns, Filters and Partitions naturally

[a] Dynamic Columns cannot be used as they do not correspond to any data in memory

Function menu

- All functions are prefixed with `n` to distinguish them from normal math
- `nsqrt(x), ncbrt(x)` – square and cubic root
- `nexp(x), nlog(x), nlog10(x)` – exponent, natural logarithm, base 10 logarithm
- `nsin(x), ncos(x), ntan(x), nasin(x), nacos(x), natan(x)` – trigonometry
- `npow(x,y), nabs(x)` – x^y, $|x|$

Analysis Task Interface: Bulk Operations

Filters, Partitions and Extended Tables

Filtering and Partitioning

```
Filter ptfilter = aod::track::pt > ptCut
```

Declares a Filter that operates on p_{T} column, it will be automatically applied to compatible table that is found within `Filtered<>` specification in `process(...)` signature; Filtered can be applied to a Join

```
Partition<myTracks> midPt = aod::track::pt >= ptl && aod::track::pt < pth;
```

Declares a Partition – a subtable of myTracks fulfilling the criterion that is made available to the user as `midPt` variable independently of the original full table; grouping is respected;

Combinations

```
using Track_binned = Join<Tracks,TrackBins>;
Tracks_binned tracks;
for (auto& [t0, t1] : selfCombinations("fBin", 5, -1, tracks, tracks)) {}
```

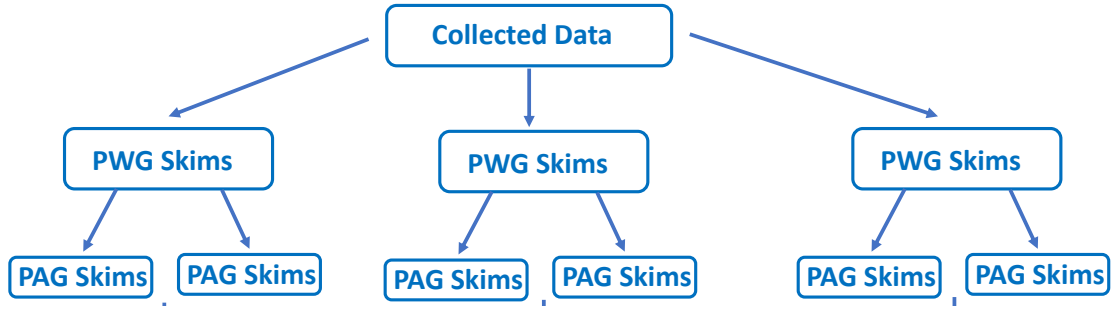
Some technical details

- Filters declared in a task will be automatically applied to *any* compatible `Filtered<>` found in process signature; several compatible Filters will be added together with logical 'and' `&&`
- Filtered tables can be further Partitioned

Skimming

Run3/4 concept

- Some objectives of the skimming strategy:
  - Regulate load on grid resources
  - Allow analysers as much freedom as possible to develop and optimise analyses
  - Optimise the frequency with which analysers can access grid resources
  - Allow for the possibility of meaningful local analyses
- Preliminary results
  - Reduction in data read load by a factor of 7-10
  - Reduction in CPU wall time by a factor of 2-10



Tiered Skimming Approach

- Each skim level will consist of a derived data set stored to disk
- Top level skims will be more inclusive – larger data sets that will be created with less frequency
- Lower level skims will be tuned to particular analysis groups – will allow analysers to utilise them with more frequency
- The skimming strategy will define the topology of the tirrs and the parameters within which they will need to operate

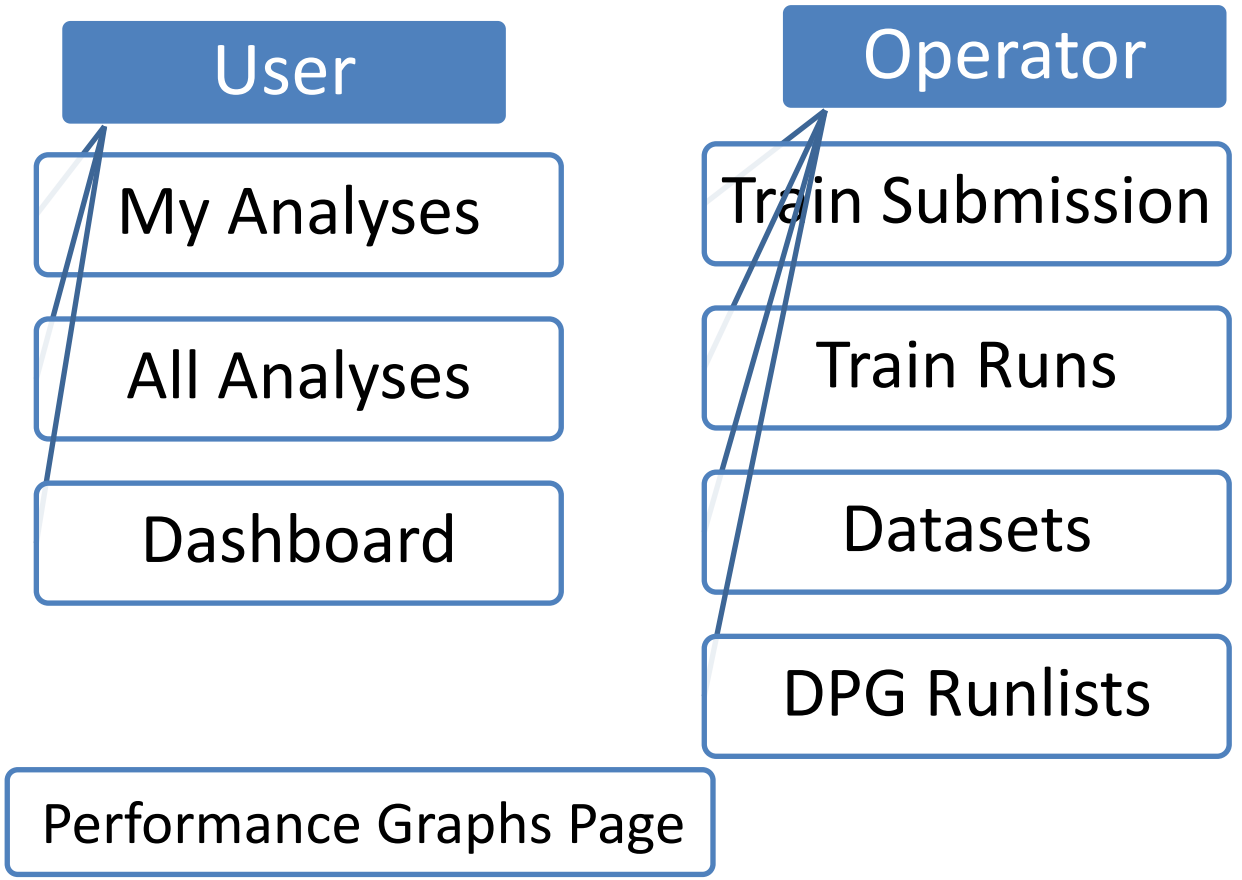
Train Framework for Run3/4

From LEGO to Hyperloop

- Rewrite of LEGO train framework
- Modern front-end technology (React)
- MonALISA backend as usual

- Builds on solid Run 2 concept

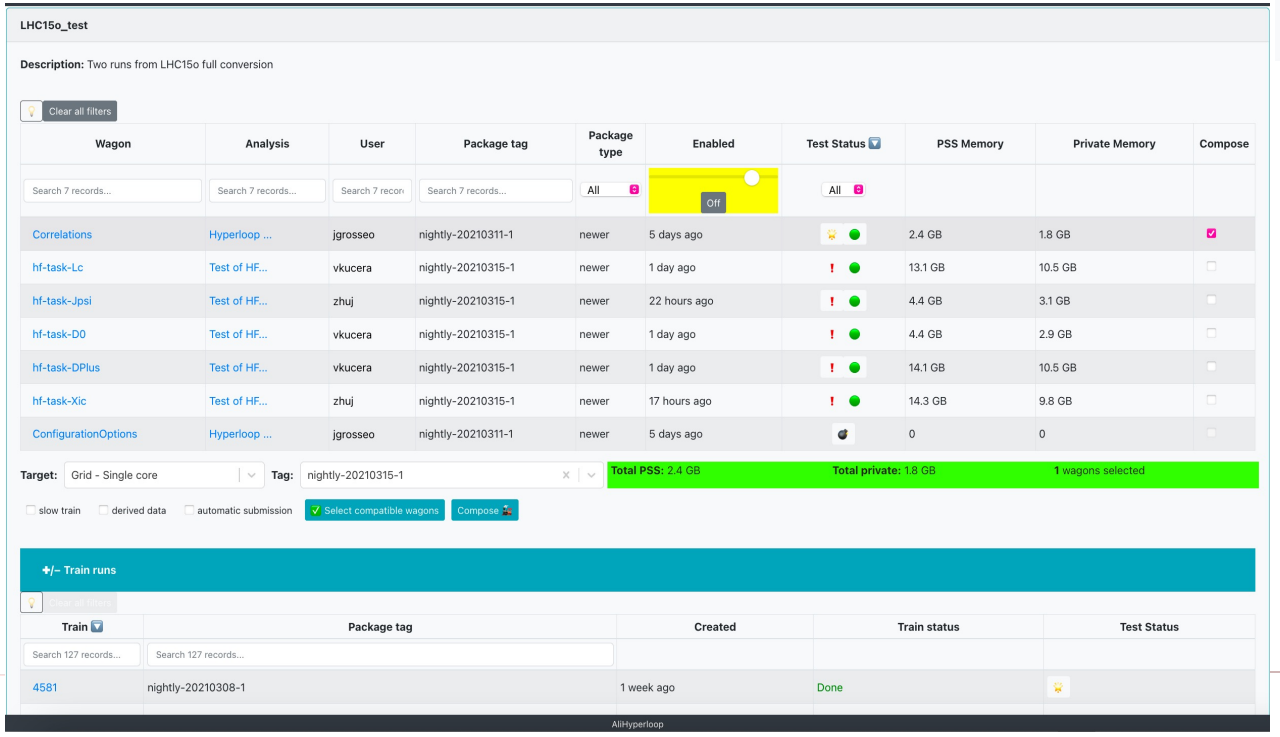
- Concept of analysis (defined on JIRA)
- Multiple analyzers use same wagons
- Direct interface to DPG (run lists)
- Trains across PWGs
- Professional operators instead of per PWG

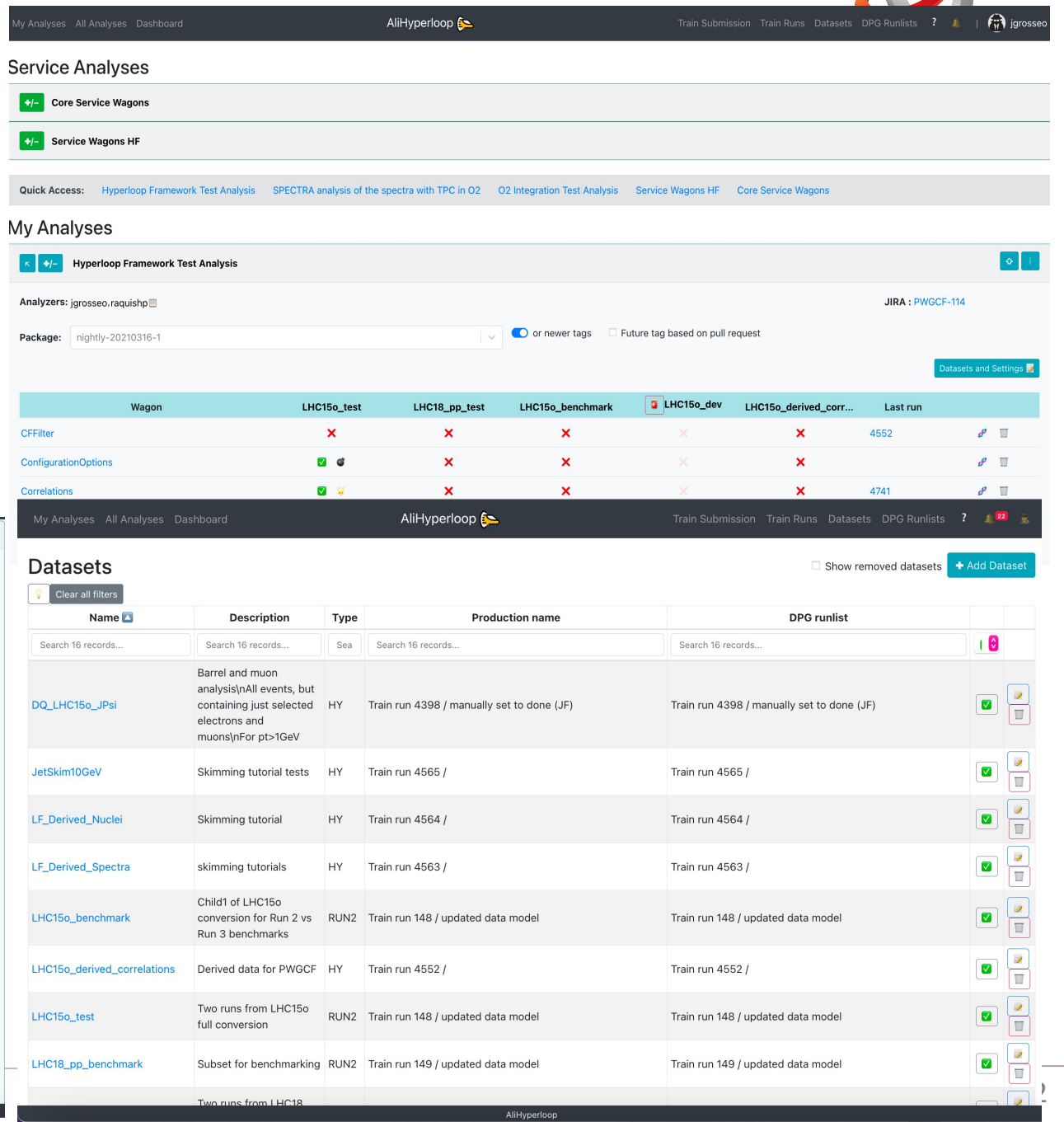


Train Framework for Run3/4

Web Interface

- Similar concept of enabling wagons
- Framework aware of parameters
- History of changes
- Interface for data sets, DPG run lists, train submission and train runs





Summary

Plans

- The Run3/4 ALICE Analysis Framework is designed to process the significantly increased amount of data
- The new flat data model is optimised for fast IO
  - We expect significant speed-up from the Root bulk IO
- The framework is still under development, but already provides the main features for declarative analysis
- Some preliminary benchmarking results show factor 3-10 higher event throughput
  - We expect the usage of skimmed data to bring additional x10 speed-up
- Ongoing work on benchmarking and optimisation

References

Additional Information

This presentation borrows slides from the following sources (many thanks to the authors!):

- C. Zampolli, ALICE data processing for Run 3 and Run 4 at the LHC, ICHEP'2020
- G.Eulisse, ALICE/Fair Framework Efforts, HSF Frameworks WG, Dec. 2019
- A. Alkin, O^2 Analysis Framework and Data Model, ALICE Physics Week, Sep. 2020. +Talk at vCHEP'2021!

The following links contain useful information and examples

- Documentation:
  https://aliceo2group.github.io/analysis-framework/
- Examples of tasks:
  https://github.com/AliceO2Group/AliceO2/tree/dev/Analysis/Tasks
- Tutorials:
  https://github.com/AliceO2Group/AliceO2/tree/dev/Analysis/Tutorials/src
  Small workflow examples that illustrate usage of the main features of the O^2 Analysis Framework in isolation (all need a data file input to function).