Processing (Super) Timeframes in JANA2

Nathan Brei

Jefferson Lab

21 January 2025

Streaming Data Processing: A comparison

Traditional

- Data acquired in online workflows
- Data is stored as large files in hierarchical storage
- Offline workflows process data
- Batch queue-based resource provisioning
- Discrete, coarse-grained processing units (files and datasets)
- Decoupling from real-time data acquisition

Streaming

- Quasi-continuous flow of fine-grained data
- Dynamic flexibility to match real-time data inflow
- Prompt processing is crucial for data quality and detector integrity
- Processing full data set quickly to minimize time for detector calibration and deliver analysis-ready data

Advantages of streaming data processing

Simplified readout

No custom trigger hardware and firmware

Holistic detector information

Build events with holistic detector information

Continuous data flow

Detailed knowledge of backgrounds and enhanced control of systematics

The JANA2 reconstruction framework

- ePIC's reconstruction software, *EICrecon*, needs to support batched event processing today and streaming event processing tomorrow. It is built on top of JANA2

- JANA2 is a scalable, modern C++ reconstruction framework designed for both batched and streaming event processing. Internally, it uses dataflow parallelism to provide efficient and flexible multithreading

- JANA2 evolves in response to ePIC's needs

JANA2 component interfaces

- **JEventSource:** A component for reading (raw) event data from a file or socket and emitting it into the JANA2 processing topology. Example: JEventSourcePODIO

- **JEventProcessor:** A component for writing (processed) event data. JANA2 will create this data on-demand by (recursively) calling the corresponding JOmniFactories.

- **JOmniFactory: A component that abstracts running an algorithm and producing some output collections. The user requests input collections, parameters, services, and resources, and JANA2 injects them.**

- **JService**: Singleton helper components, usually for obtaining additional data keyed off of run number. Examples: Geometry, calibrations, logging.

JOmniFactory

- Interface for using EICrecon algorithms inside JANA2
- Extends JMultifactory
- Provides EICrecon-specific logger
- Provides optional ConfigT structure
- Uses the curiously recurring template pattern
- Wiring is set by the JOmniFactoryGenerator

```C++
struct ClusterConfig {
    double offset = 0;
}

struct ClusterFac: public JOmniFactory<ClusterFac, ClusterConfig> {

    PodioInput<Cluster> m_protoclusters_in {this};

    PodioOutput<Cluster> m_clusters_out {this};

    ParameterRef<double> m_offset {this, "offset", config().offset};

    void Configure() { }

    void ChangeRun(int32_t run_nr) { }

    void Execute(int32_t run_nr, uint64_t evt_nr) {
        for (auto proto : *m_protoclusters_in()) {
            auto cluster = m_clusters_out()->create();
        }
    }
};
```

JOmniFactoryGenerator

- Holds wiring information for each instance of an JOmniFactory
- Parameter values are type-safe thanks to ConfigT
- Allows us to maintain a small number of factory classes
- Parameters, input, and output tags can be overridden on the command line

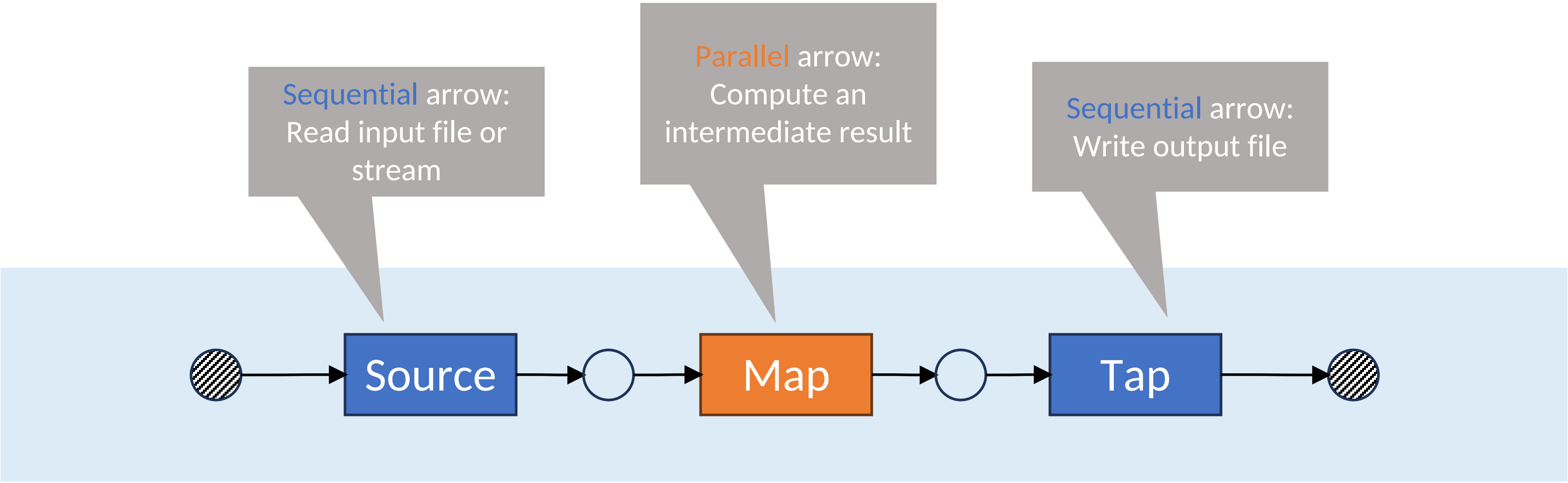
C++

```cpp

extern "C"
void InitPlugin(JApplication *app) {
    InitJANAPlugin(app);

    auto cluster_gen = new JOmniFactoryGeneratorT<ClusterFac>(
                             "clusterizer",     // prefix
                             {"protoclusters"}, // inputs
                             {"clusters"},      // outputs
                             {.offset=1000});   // configs

    app->Add(cluster_gen);
}
```

bash

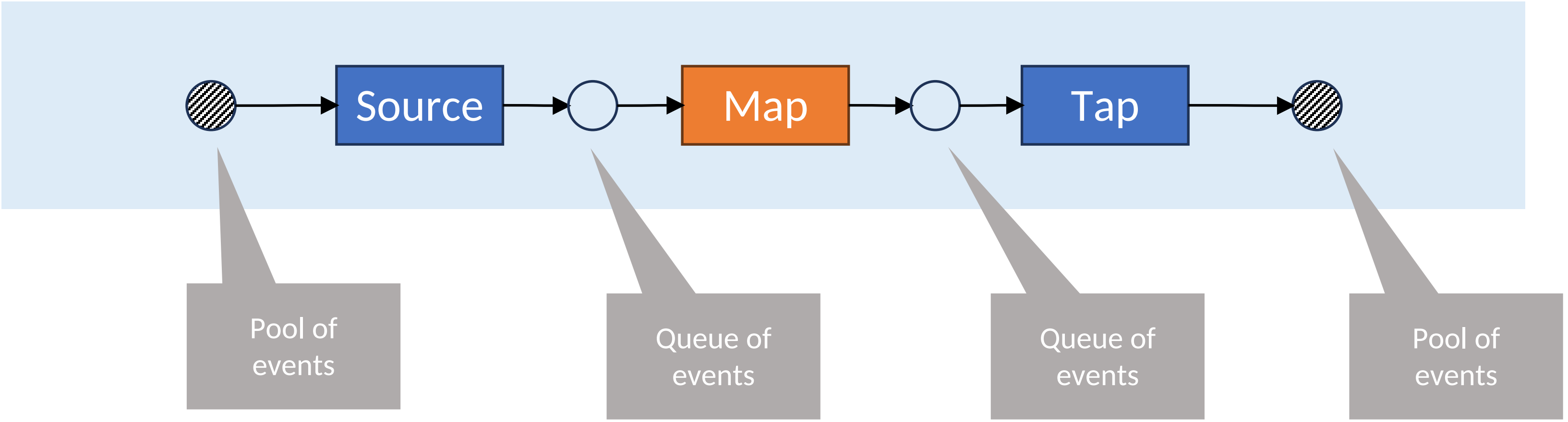
```bash
# Override this wiring using the following parameters:
# - clusterizer:InputTags
# - clusterizer:OutputTags
# - cluserizer:offset

eicrecon -Preco:clusterizer:InputTags="smeared_protoclusters" input.root
```

How JANA2 works internally – Formalism



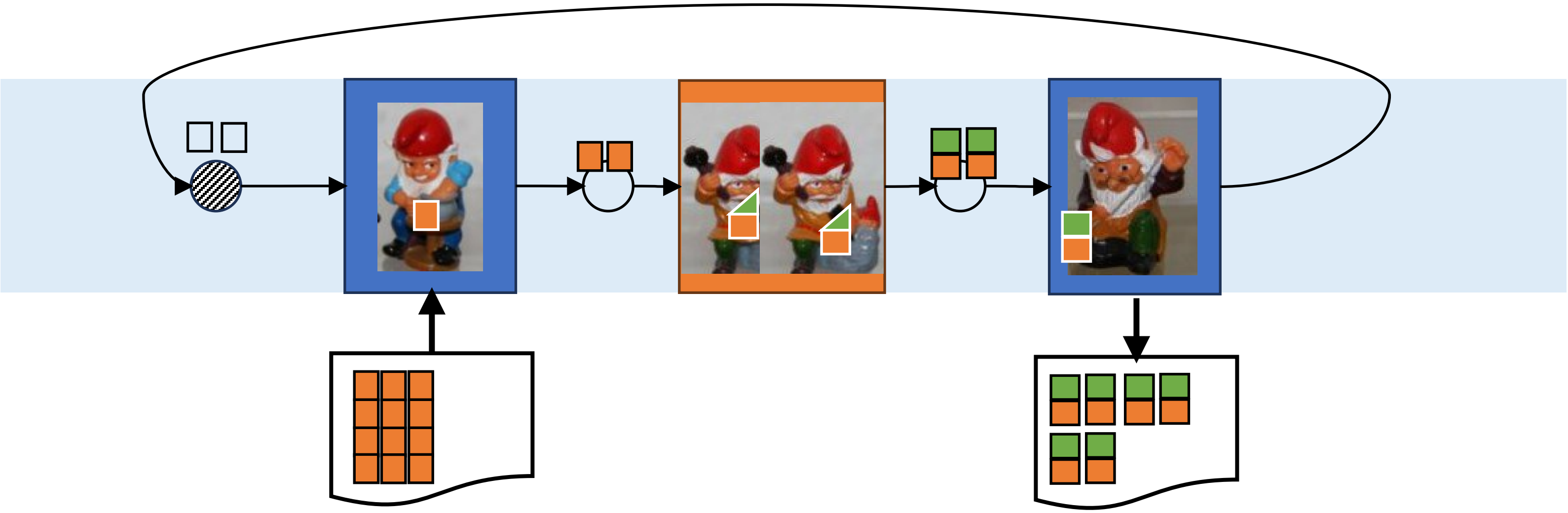
- Dataflow-parallel **processing topology** consisting of **arrows, queues,** and **pools**
- Arrows represent fixed tasks which may be sequential or parallel
- Arrows may have multiple queues and pools for their inputs and outputs
- Queues allow asynchronous processing so that no thread is directly waiting for a computation to finish

How JANA2 works internally – Formalism



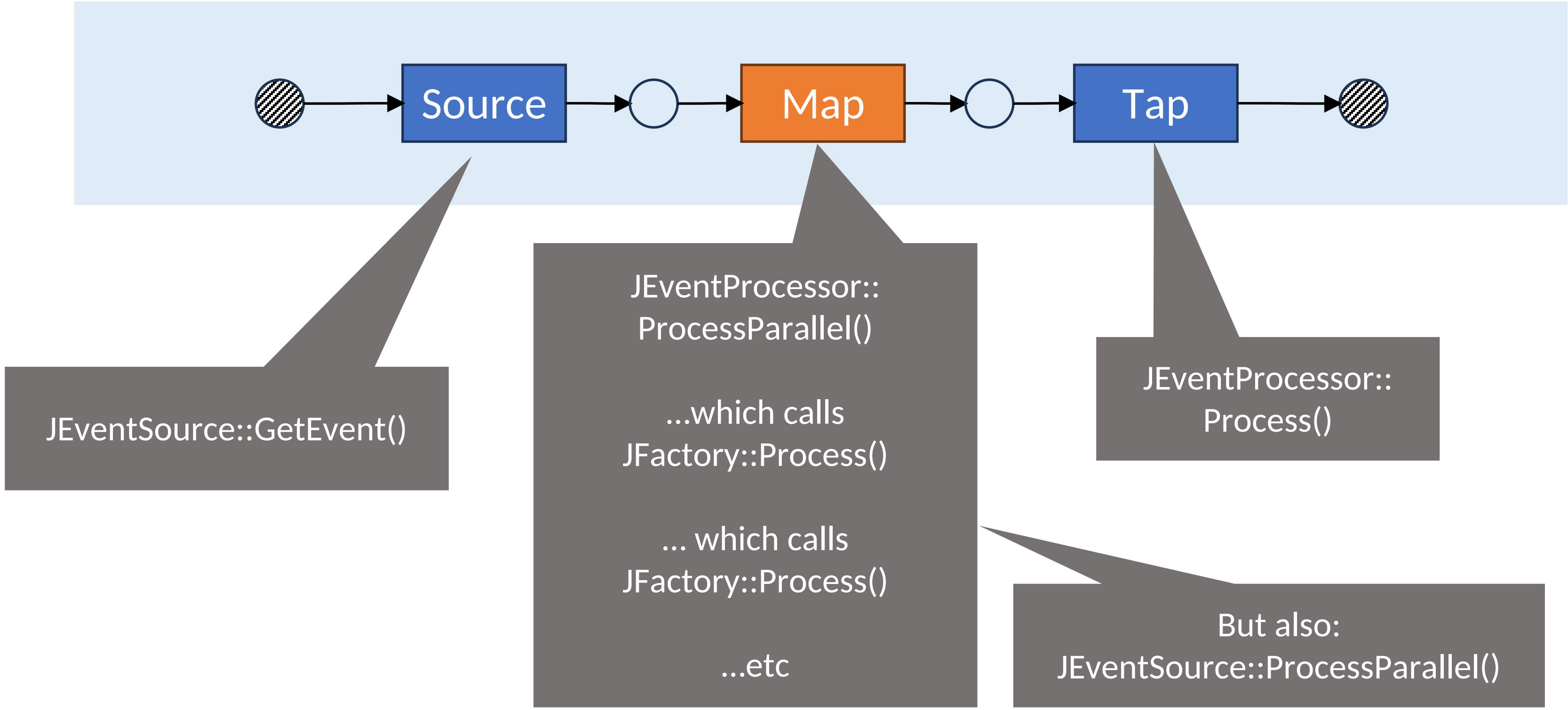
- Dataflow-parallel **processing topology** consisting of **arrows, queues,** and **pools**
- Arrows represent fixed tasks which may be sequential or parallel
- Arrows may have multiple queues and pools for their inputs and outputs
- Queues allow asynchronous processing so that no thread is directly waiting for a computation to finish

How JANA2 works internally – Cartoon



How JANA2 Components map to Arrows

- The user doesn’t interact with topologies or arrows directly
- Instead, the user provides JANA with components such as JEventSources, JEventProcessors, JFactories
- Components are **decoupled** from each other. “**Only communicate through the data model**”
- JANA2 assigns the components’ callbacks to arrows in the processing topology



JEventSource::GetEvent()

JEventProcessor::
ProcessParallel()

...which calls
JFactory::Process()

... which calls
JFactory::Process()

...etc

JEventProcessor::
Process()

But also:
JEventSource::ProcessParallel()

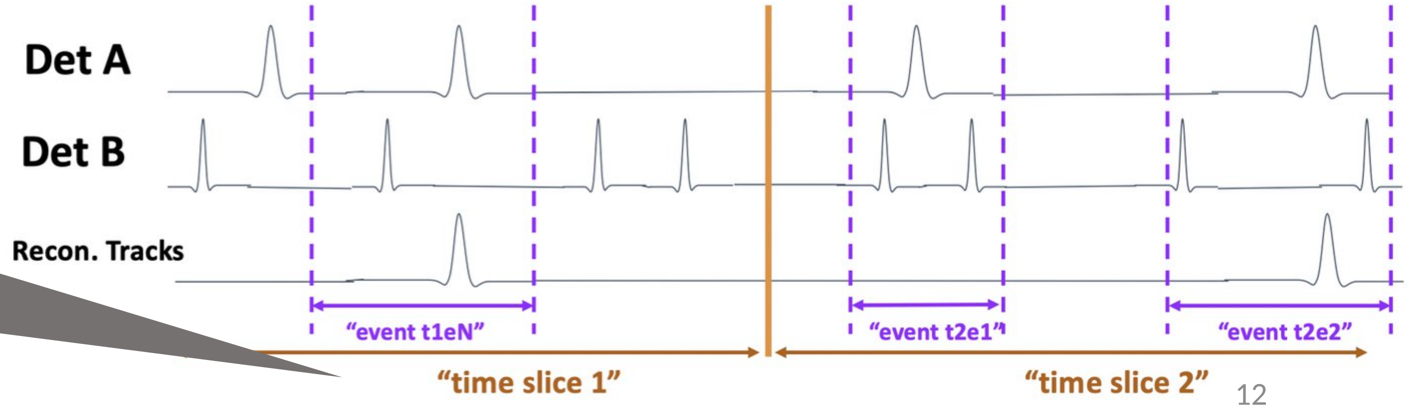
Event levels

- JANA2 has a JEvent abstraction which previously meant both
  1. A container of intermediate data that is used as JANA's unit of parallelism
  2. A physics event
- Now, JEvent strictly means (1).

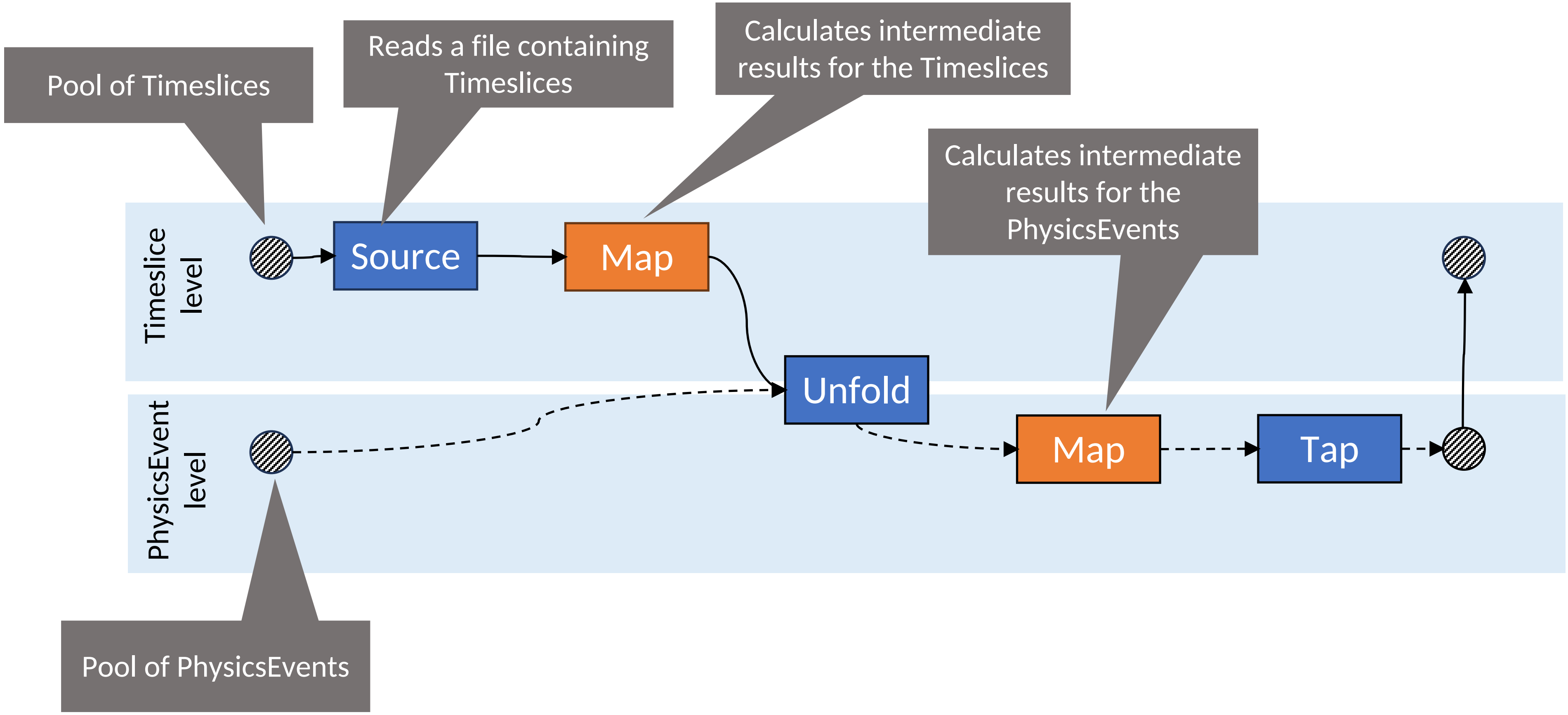
- Each JEvent is *tagged* (not typed!) as belonging to some JEventLevel.
- For now, JEventLevel is an enum, although user-definable event levels may be supported in the future.
- JANA2 doesn't assume that all event levels are hierarchical, e.g. that one physics event fits inside exactly one block, or even fully ordered. Instead, users establish that relationship explicitly.

```
enum class JEventLevel {
    Run,
    Subrun,
    Timeslice,
    Block,
    SlowControls,
    PhysicsEvent,
    Subevent,
    Task,
    None
};
```

"PhysicsEvents" and "Timeframes" are simply d**ifferent partitionings** of the time domain. As such, the JANA2 framework should handle these cases **symmetrically** to the maximum extent possible.



Generalizing to two event levels



Pool of Timeslices

Reads a file containing Timeslices

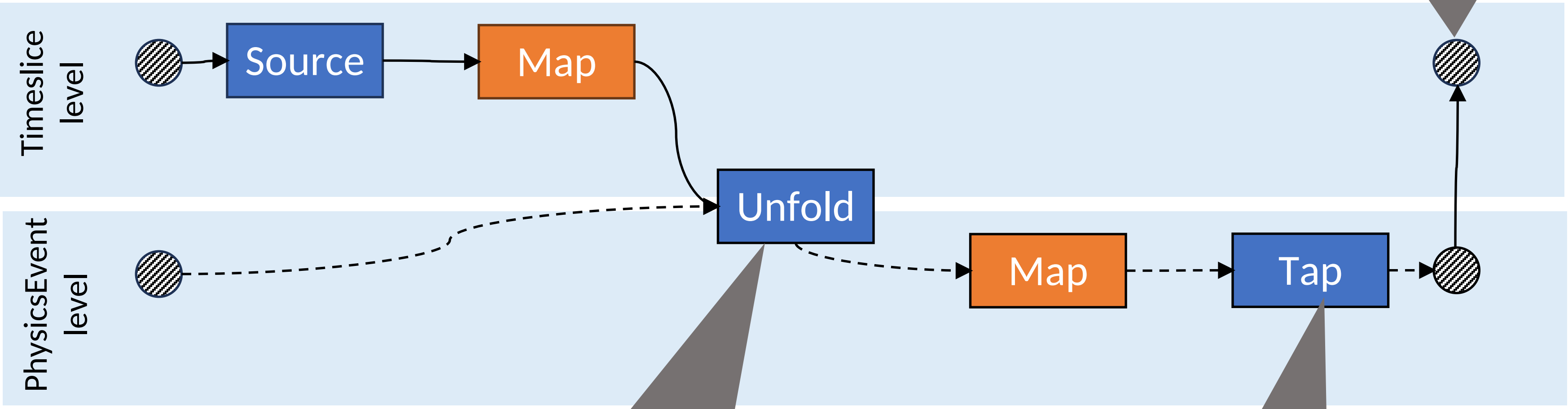
Calculates intermediate results for the Timeslices

Calculates intermediate results for the PhysicsEvents

Pool of PhysicsEvents

Generalizing to two event levels

PhysicsEvent pool knows how to recycle Timeslices once their reference count reaches zero



Splits Timeslices into PhysicsEvents. Keeps the Timeslice around and lets the PhysicsEvent reference it.

This uses a new type of component!

Writes PhysicsEvents (and possibly Timeslices) to file

Introducing JEventUnfolder component

```
Result Unfold(
    const JEvent& parent,
    JEvent& child,
    int child_index) override;
```

```
enum class Result {
    NextChildNextParent,
    NextChildKeepParent,
    KeepChildNextParent
};
```

- JEventUnfolder looks and feels very similar to a JOmniFactory
- Users may declare Parameters, Services, Resources, Inputs, Outputs, or access everything through JApplication/JEvent
- No Generator needed as there will only be one instance active for any given level, same as JEventProcessors

- Provides an **Unfold** callback
  - Name comes from functional programming and stream processing
  - Unfold handles both “splitting” and “merging” streams
  - Returns a Result code indicating whether the parent and child belong together
  - We never need to have all PhysicsEvents corresponding to one Timeslice in memory at once

- Inputs come from the parent event (e.g. Timeslice)
- Outputs are inserted into the child event (e.g. PhysicsEvent)
- The child event keeps a pointer to the parent event around, so that any factory can access Timeslice-level data

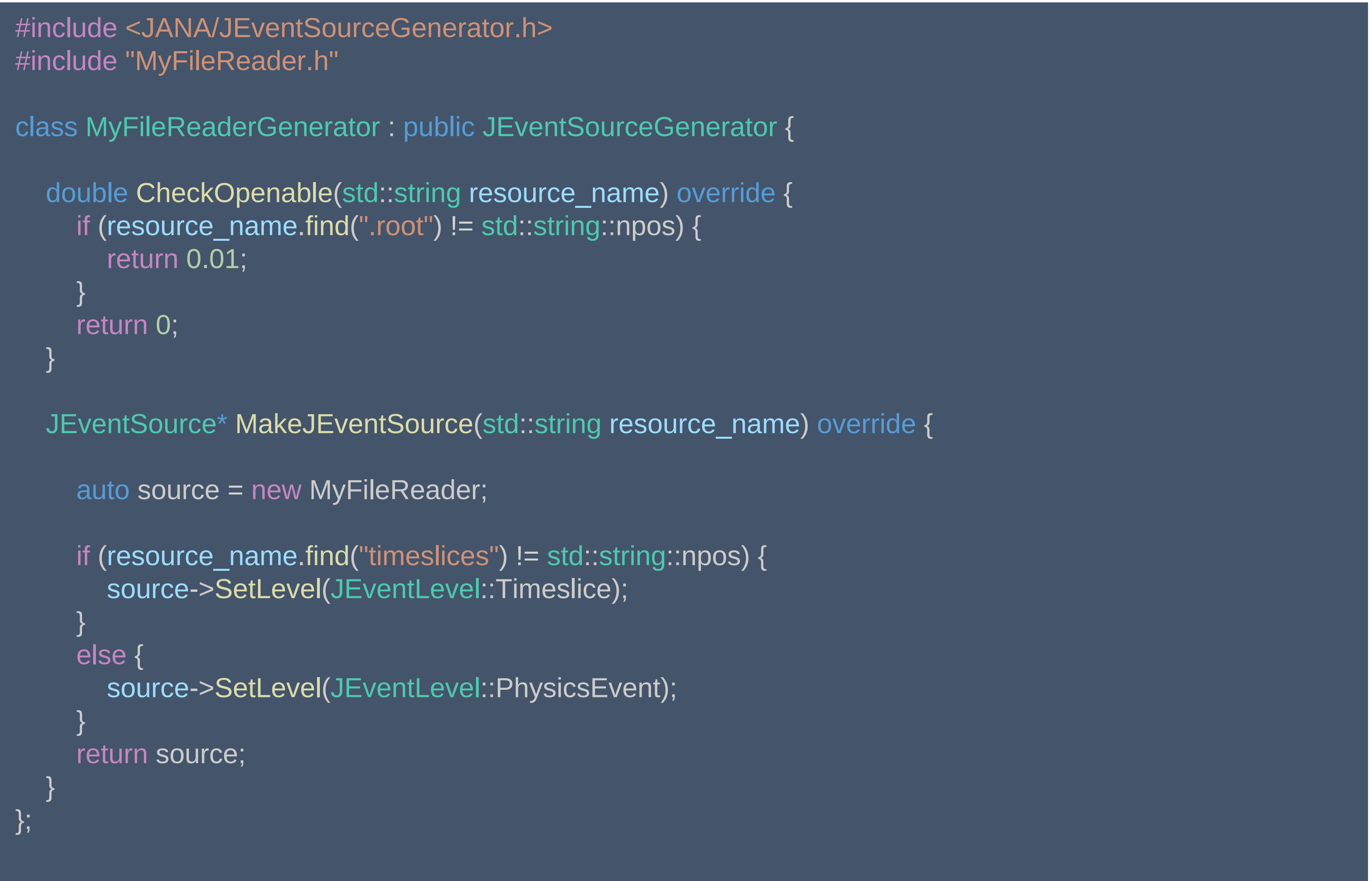
What does this mean for our Factories?

- OmniFactories look almost exactly the same as before
- OmniFactories each belong to a particular event level. All of their outputs belong to that level.
- OmniFactory::Input helper now takes event level as an optional parameter
- Event level information can be applied **entirely** at the JOmniFactoryGenerator level
- The same algorithm and factory can be wired and reconfigured for different event levels

```
struct MyProtoclusterFactory
  : public JOmniFactory<MyProtoclusterFactory> {

PodioInput<ExampleHit> hits_in {this};
PodioOutput<ExampleCluster> clusters_out {this};

void Configure() {
}

void ChangeRun(int32_t run_nr) {
}

void Execute(int32_t run_nr, uint64_t evt_nr) {
    ...
}
```

```
// Factory that produces timeslice-level protoclusters
// from timeslice-level hits
app->Add(new JOmniFactoryGeneratorT<MyProtoclusterFactory>(
    { .tag = "timeslice_protoclusterizer",
      .level = JEventLevel::Timeslice,
      .input_names = {"hits"},
      .output_names = {"ts_protoclusters"}
    }));

// Factory that produces event-level protoclusters
// from event-level hits
app->Add(new JOmniFactoryGeneratorT<MyProtoclusterFactory>(
    { .tag = "event_protoclusterizer",
      .input_names = {"hits"},
      .output_names = {"evt_protoclusters"}
    }));
```

What does this mean for JEventSources?



```
#include <JANA/JEventSourceGenerator.h>
#include "MyFileReader.h"

class MyFileReaderGenerator : public JEventSourceGenerator {

    double CheckOpenable(std::string resource_name) override {
        if (resource_name.find(".root") != std::string::npos) {
            return 0.01;
        }
        return 0;
    }

    JEventSource* MakeJEventSource(std::string resource_name) override {

        auto source = new MyFileReader;

        if (resource_name.find("timeslices") != std::string::npos) {
            source->SetLevel(JEventLevel::Timeslice);
        }
        else {
            source->SetLevel(JEventLevel::PhysicsEvent);
        }
        return source;
    }
};
```

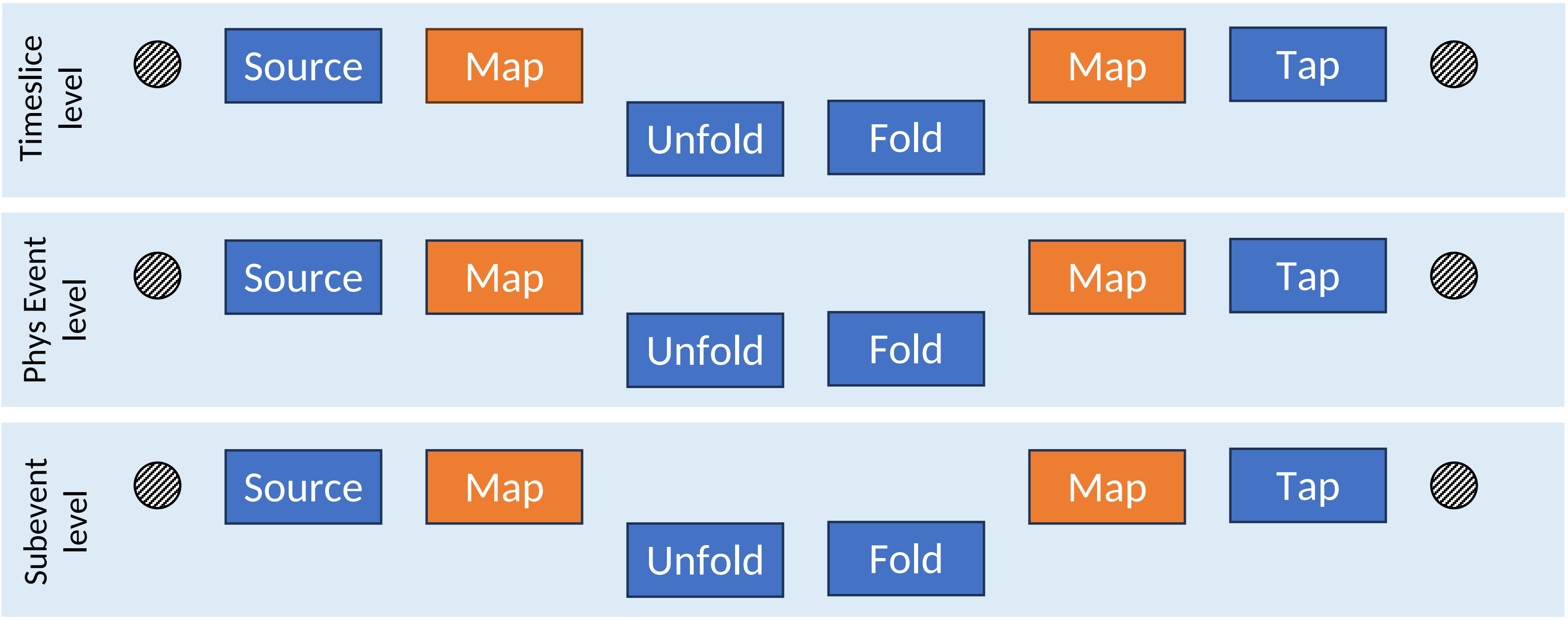
- JANA2 can figure out that the input file contains timeslices from inside the JEventSourceGenerator

- This means that this critical information is already known before the time of topology construction

- The topology builder is able to decide what topology to build based off what components were provided.

- The same PODIO event source class can be reused for files containing timeslices vs physics events with minimal modification

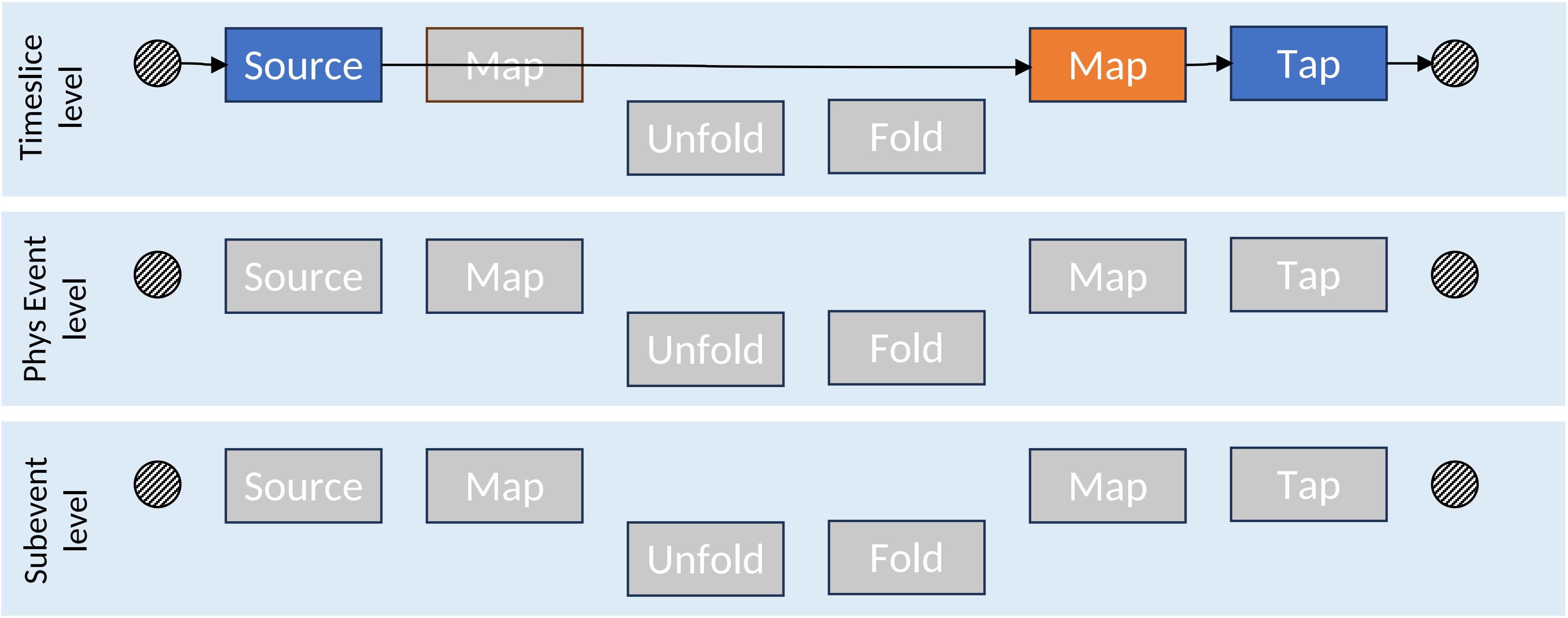
Generalizing further



- Source calls
  - JEventSource::GetEvent()
- Map calls
  - JOmniFactory::Process()
  - JEventProcessor::ProcessParallel()
  - JEventSource:: ProcessParallel()
  - JEventUnfolder:: ProcessParallel()
  - JEventFolder:: ProcessParallel()
- Tap calls
  - JEventProcessor::Process()
- Unfold calls
  - JEventUnfolder::Unfold()
- Fold calls
  - JEventFolder::Fold()

- The arrows in the further generalized topology (abstractly) form a grid:
  `{Source, Map1, Unfold, Fold, Map2, Tap} x {Timeslice, PhysicsEvent, Subevent,…}`
- Depending on which components the user provides, JANA2 can activate and wire the arrows automatically
- This wiring could also be specified manually

Basic topology



Timeslice level

Source

Map

Unfold

Fold

Map

Tap

Phys Event level

Source

Map

Unfold

Fold

Map

Tap

Subevent level

Source

Map

Unfold

Fold

Map

Tap

User provides:

- JEventSource [Timeslice]
- JEventProcessor [Timeslice]
- JFactory [Timeslice]

→ Timeslice

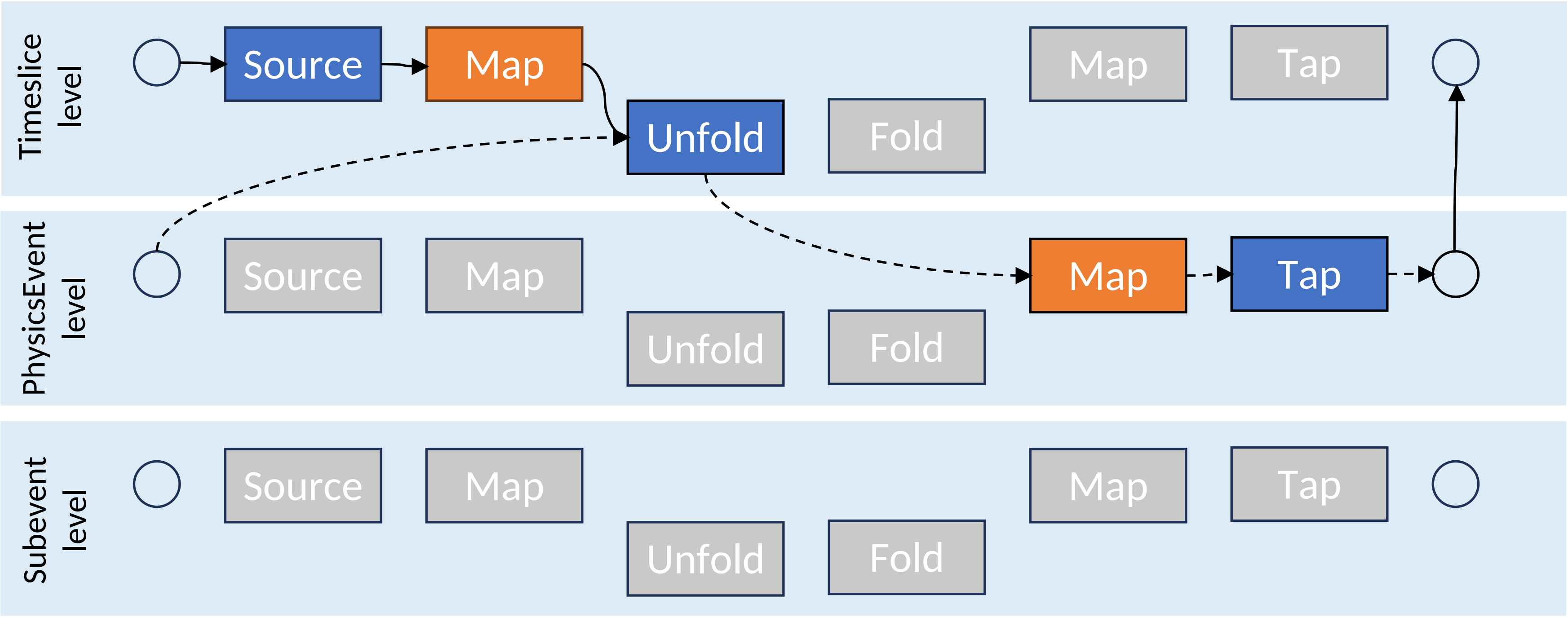
- - -→ Event

-------→ Subevent

Parallel

Sequential

Timeslice splitting topology



Only one wiring usually makes sense for each combination of components the user may add!

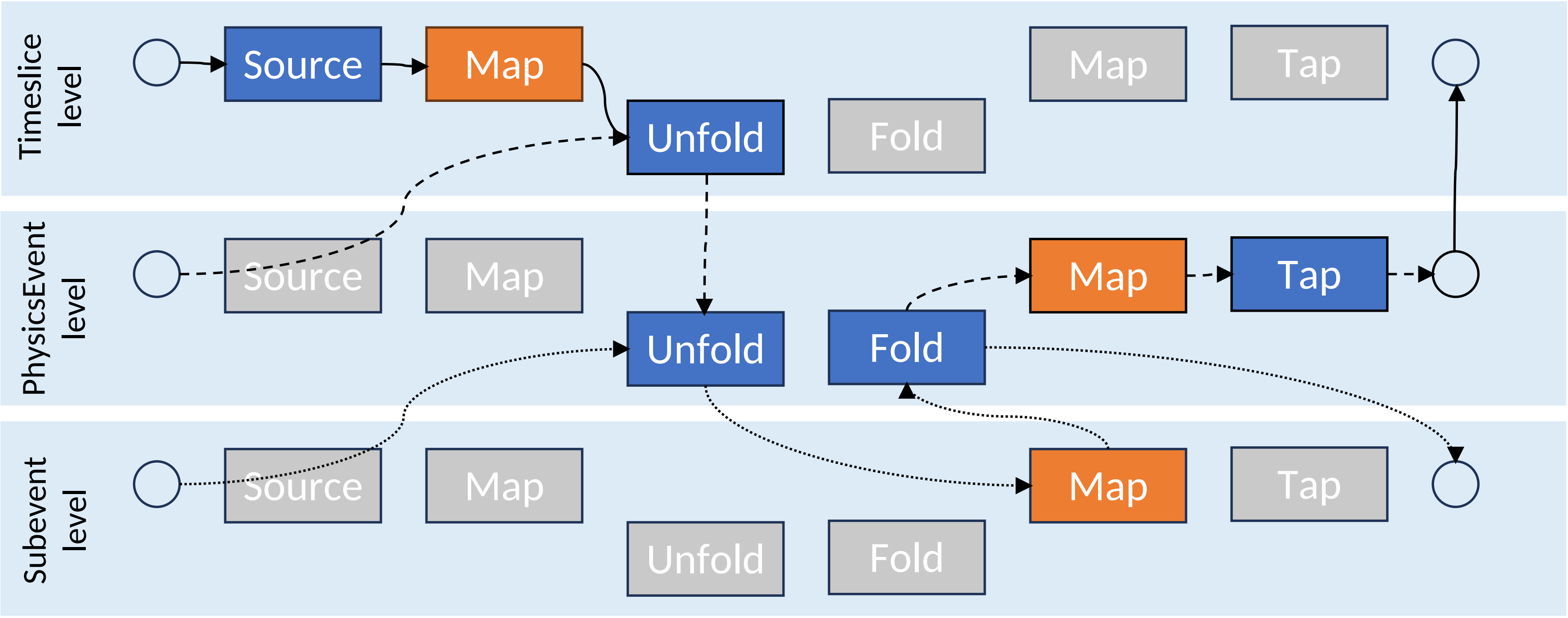
User provides:

- JEventSource [T]
- JFactory [T]
- JEventUnfolder [T -> P]
- JEventProcessor [P]
- JFactory [P]

⟶	Timeslice
- - -▸	Event
·······▸	Subevent

🟧 Parallel 🟦 Sequential

Timeslices + subevents topology

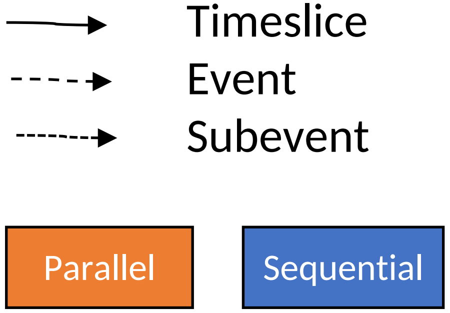


User provides:

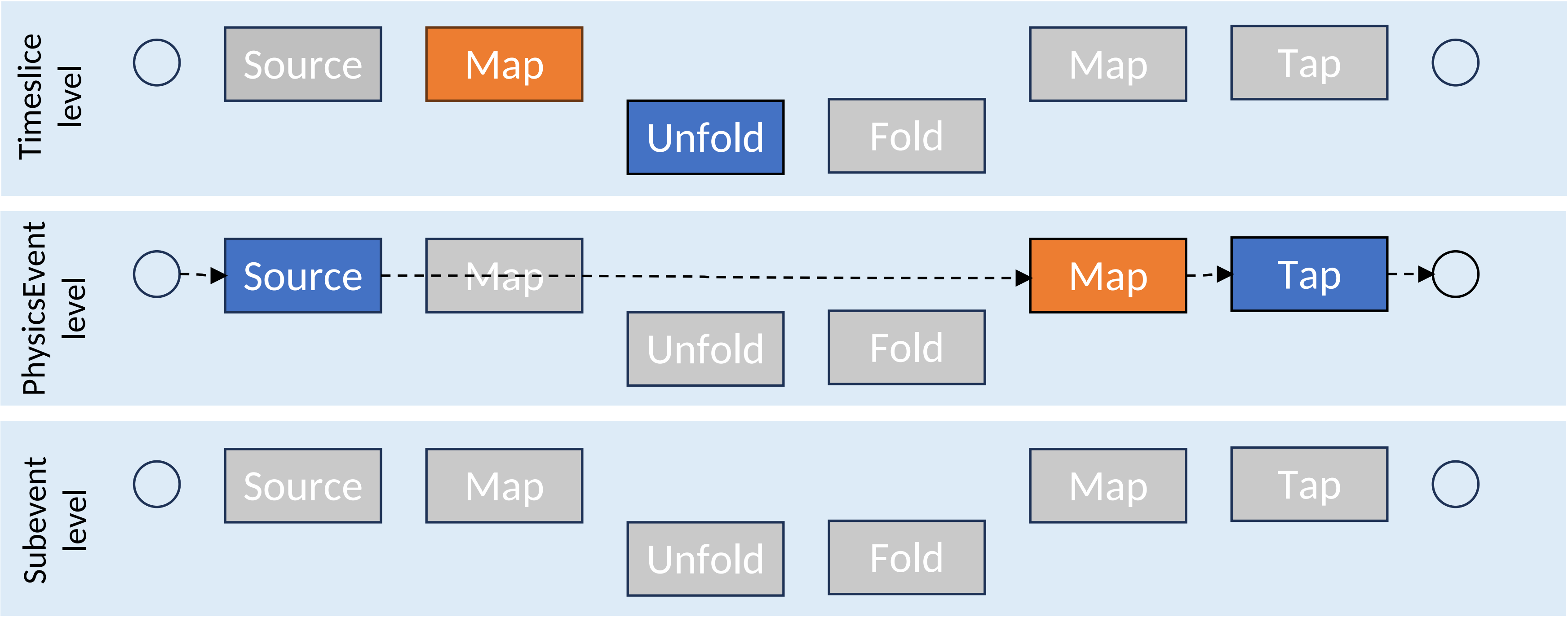
- JEventSource [T]
- JEventProcessor [P]

- JEventUnfolder [T -> P]
- JEventUnfolder [P -> S]
- JEventFolder[S -> P]

- JFactory [T]
- JFactory [P]
- JFactory [S]



What happen if the user provides “extra” components?



User provides:

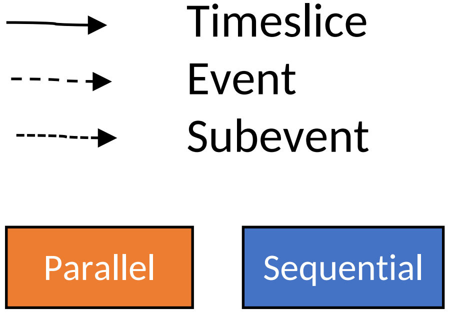
- JEventSource [P]
- JEventProcessor [P]
- JEventUnfolder [T -> P]

IGNORED!

- JFactory [T]

IGNORED!

- JFactory [P]



What does this mean for EICrecon?

- We can define our factories and algorithms once
- We can add generators that wire them differently for the timeslice input files and for physics input files
- These wirings can live side-by-side without interfering with each other
- We can define our PODIO event source and processor once
- We can add a generator that configures the source's event level
- The topology builder choose which topology to build based off of which components (most notably, sources) are present
- **No additional configuration necessary! Eases the transition from events to timeslices**

Memory management – Concept

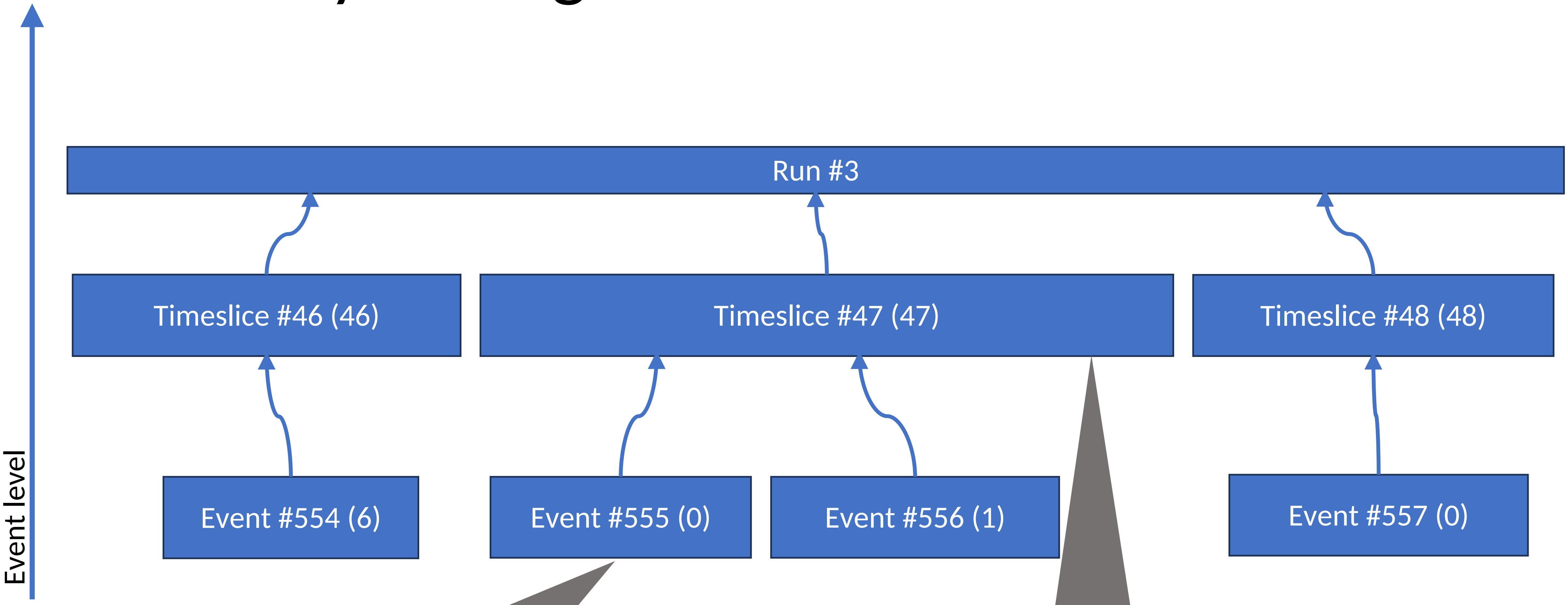
As of right now:

- Parents have shared-ptr-like semantics (except they are recycled to a pool)
- Parents always outlive their children
- Events can have multiple parents
- Parents are uniquely identified by their event level: “Diamond inheritance” not permitted
- To get data from a parent, you have to ask for the parent explicitly (no searching or “importing into the global namespace”)

Future improvements:

- Event sources will eventually be able to emit events that already have parents
- Data in adjacent timeslices will be accessible via a ‘sibling’ reference, analogous to parents except weak-ptr-like

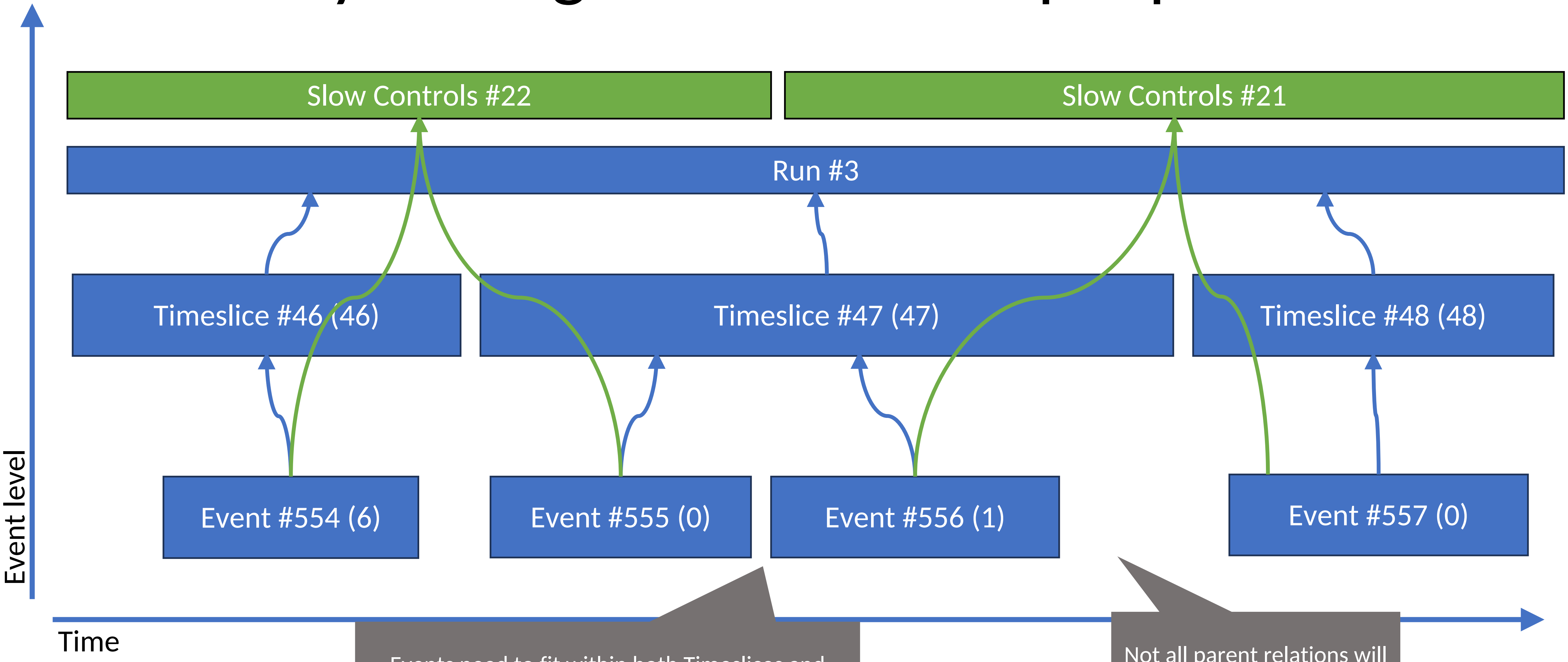
Memory management – Parent relation



PhysicsEvent 555 can reference Timeslice 47 and Run 3

Timeslice 47 will stay in memory until PhysicsEvent 555 and 556 are recycled

Memory management – Multiple parents



Events need to fit within both Timeslices and SlowControls, but SlowControls and Timeslices can overlap!

Not all parent relations will necessarily come from the Unfolder!

Current status

- An end-to-end working example of timeframe splitting is already present in JANA2’s master branch
  - src/examples/TimesliceExample
  - https://github.com/JeffersonLab/JANA2/

- EICrecon has a skeleton for timeframe splitting as a WIP PR
  - https://github.com/eic/EICrecon/pull/1510
  - Proof-of-concept for TDR: Kolja, Shuji, Barak
  - Generated data files containing “wide events” with background
  - Goal: test tracking accuracy without requiring realistic timeframe splitting logic
  - Developing realistic timeframe splitting logic is non-trivial

Thank you!