Timeframe-level reconstruction in JANA2

Nathan Brei

Jefferson Lab

25 July 2024

Overview

Recap of work since February

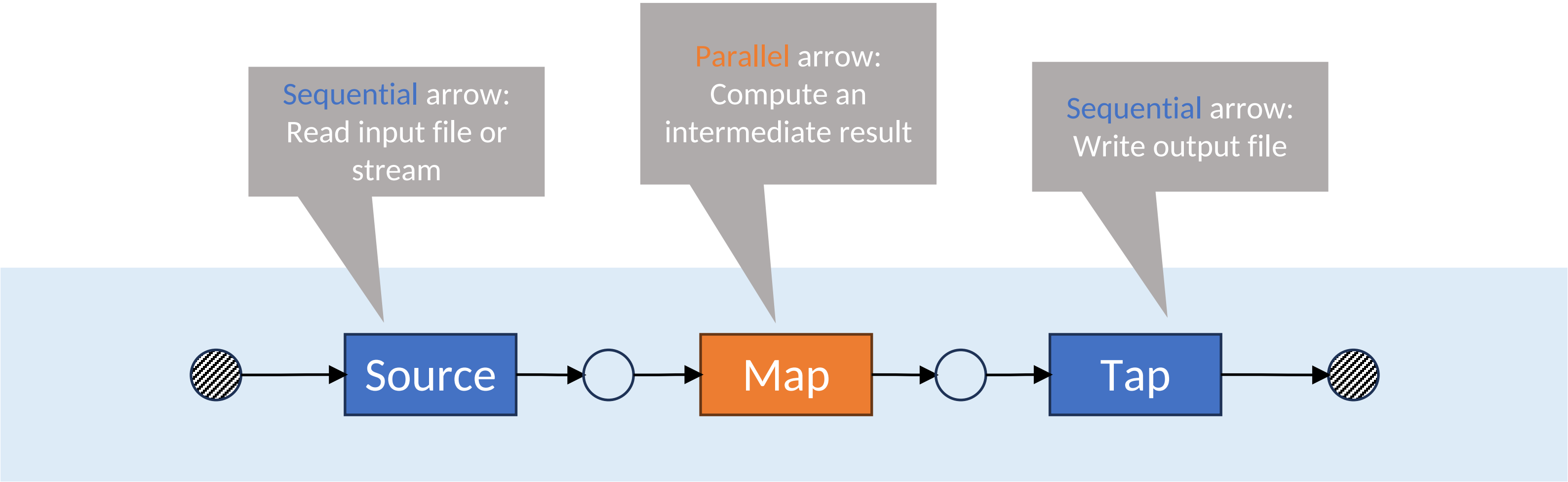
- Quick primer on JANA2 parallelism internals
- Event levels
- New components: Unfolders and Folders
- Consequences for existing components: (Omni)Factories and EventSources
- Dynamic wiring of processing topology
- Memory ownership options

Current status

Ongoing work

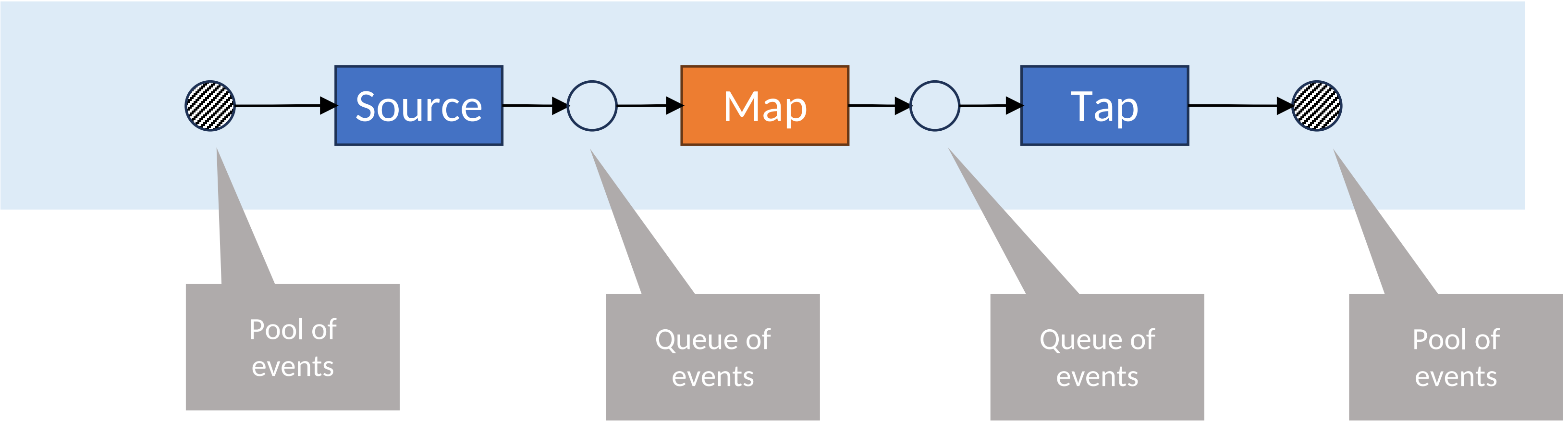
- Generalized event and run numbers
- Slow controls: interleaving vs side-loading
- Event classification and filtering

How JANA2 works internally – Formalism



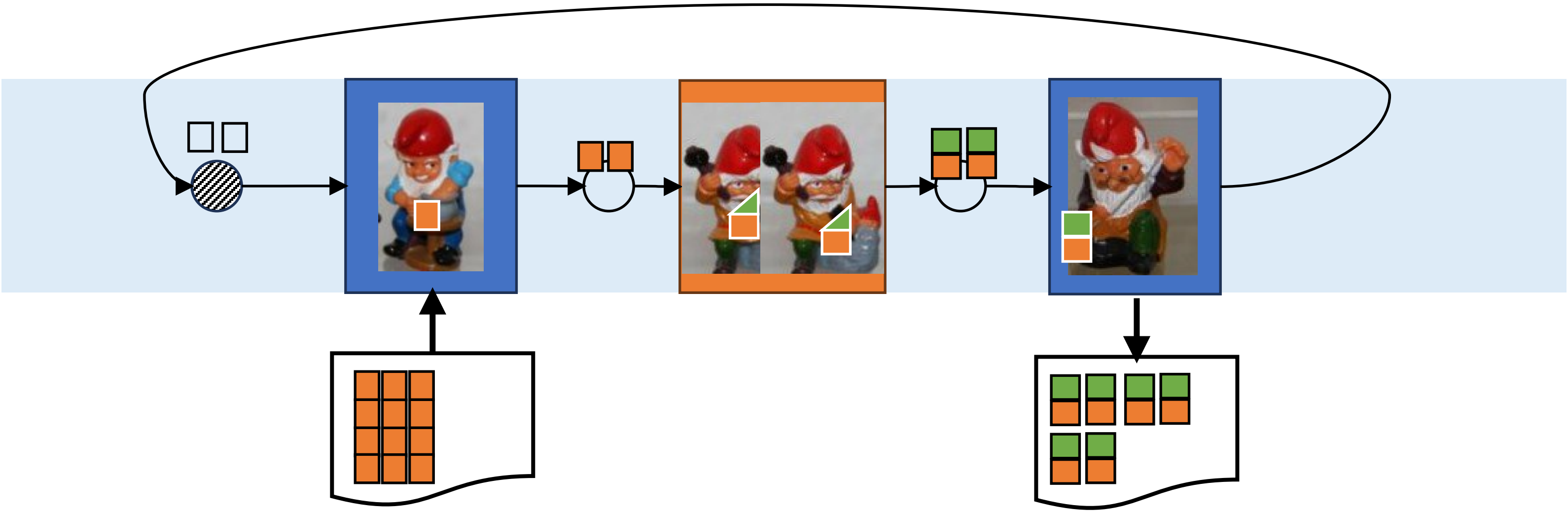
- Dataflow-parallel **processing topology** consisting of **arrows, queues,** and **pools**
- Arrows represent fixed tasks which may be sequential or parallel
- Arrows may have multiple queues and pools for their inputs and outputs
- Queues allow asynchronous processing so that no thread is directly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

How JANA2 works internally – Formalism



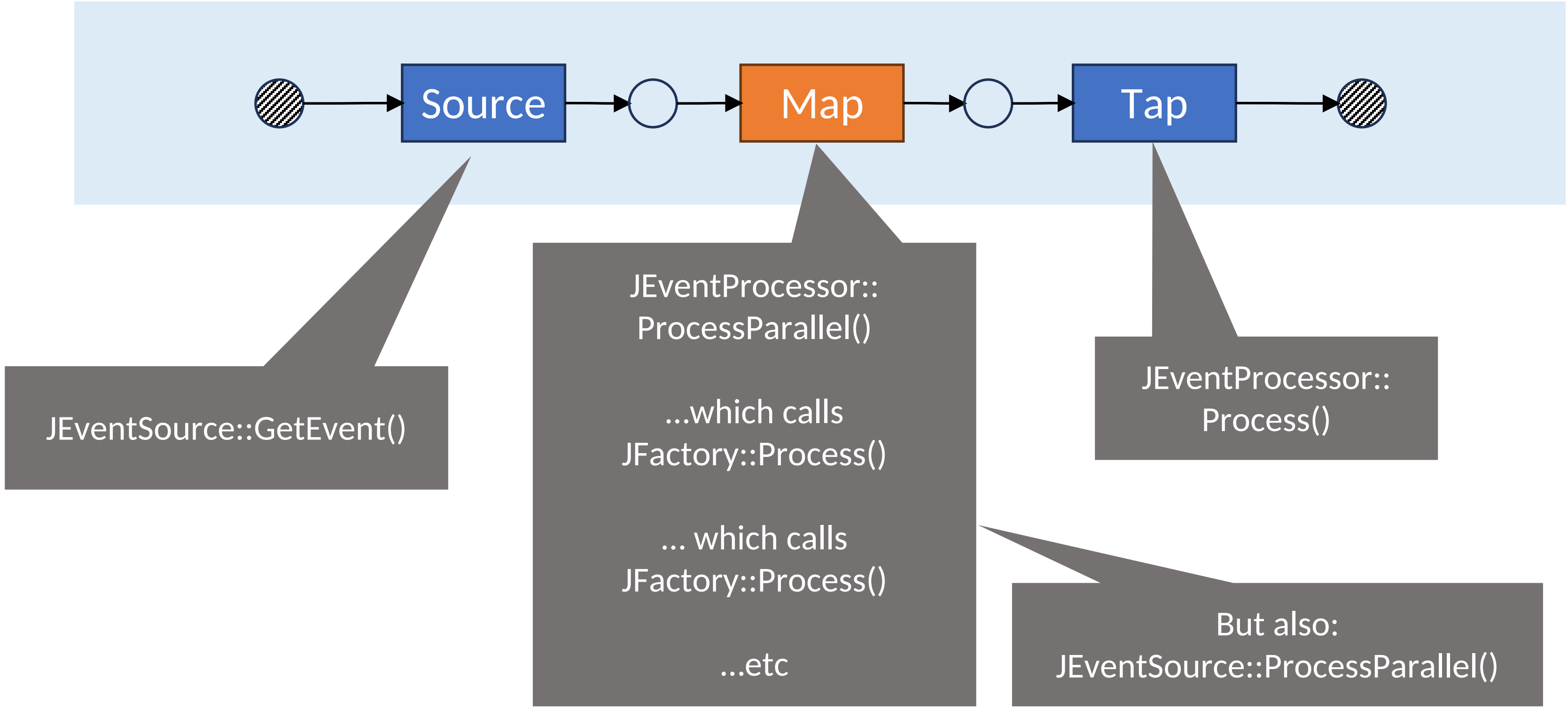
- Dataflow-parallel **processing topology** consisting of **arrows, queues,** and **pools**
- Arrows represent fixed tasks which may be sequential or parallel
- Arrows may have multiple queues and pools for their inputs and outputs
- Queues allow asynchronous processing so that no thread is directly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



Source

Map

Tap

JEventSource::GetEvent()

JEventProcessor::
ProcessParallel()

...which calls
JFactory::Process()

... which calls
JFactory::Process()

...etc

JEventProcessor::
Process()

But also:
JEventSource::ProcessParallel()

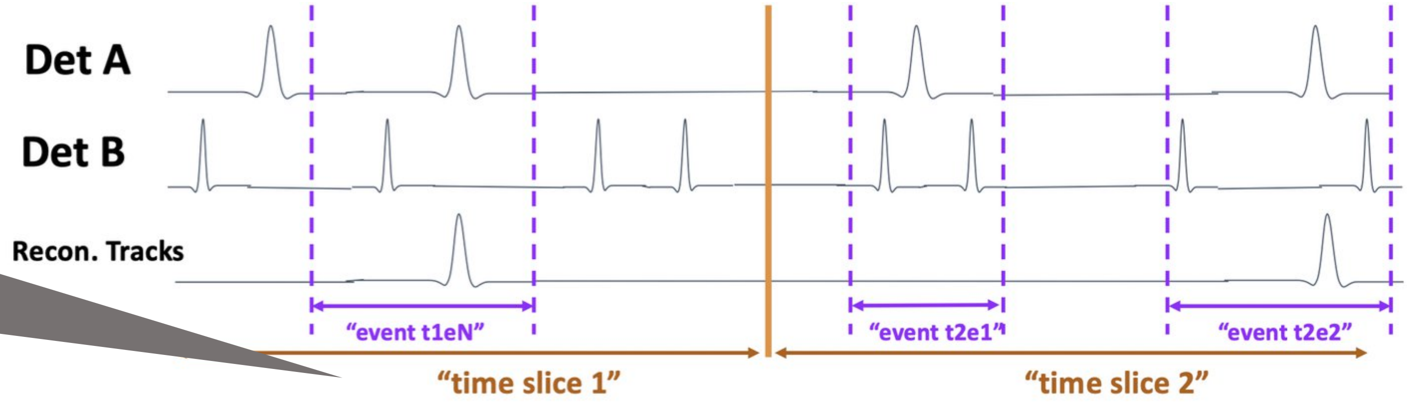
Event levels

- 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).

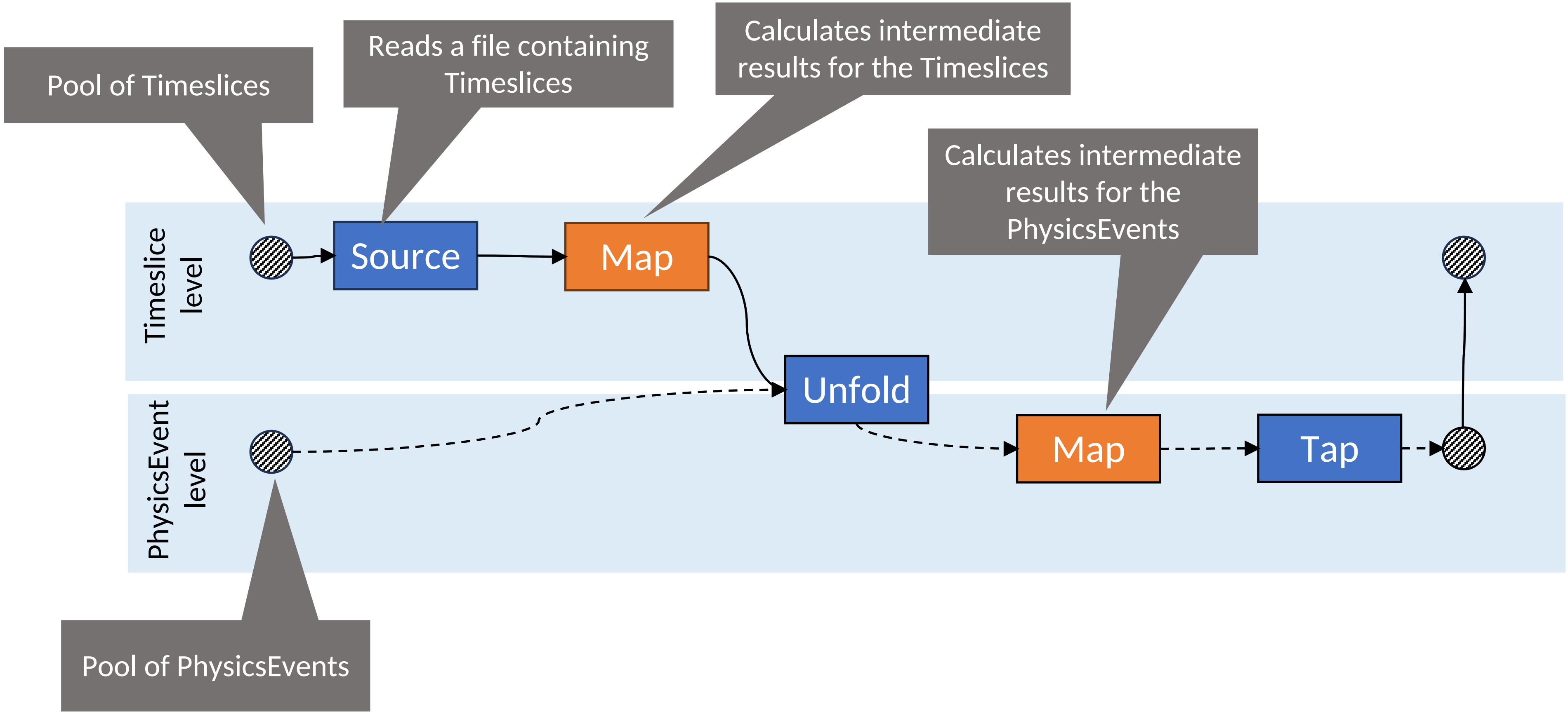
- Each JEvent is *tagged* (not typed!) as belonging to some JEventLevel.
- For now, JEventLevel is an enum, although user-definable event levels may be supported in the future.
- JANA2 doesn't assume that all event levels are hierarchical, e.g. that one physics event fits inside exactly one block, or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

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

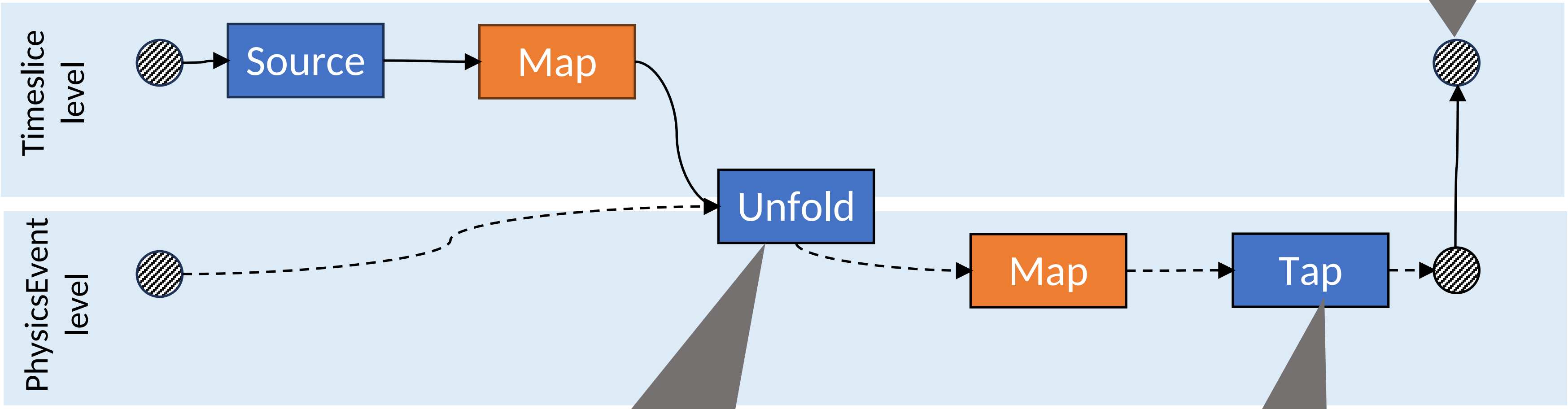
"PhysicsEvents" and "Timeframes" are simply **different partitionings** of the time domain. As such, the JANA2 framework should handle these cases **symmetrically** to the maximum extent possible.



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
Timeslice level
Source
Map
Unfold
PhysicsEvent level
Map
Tap
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

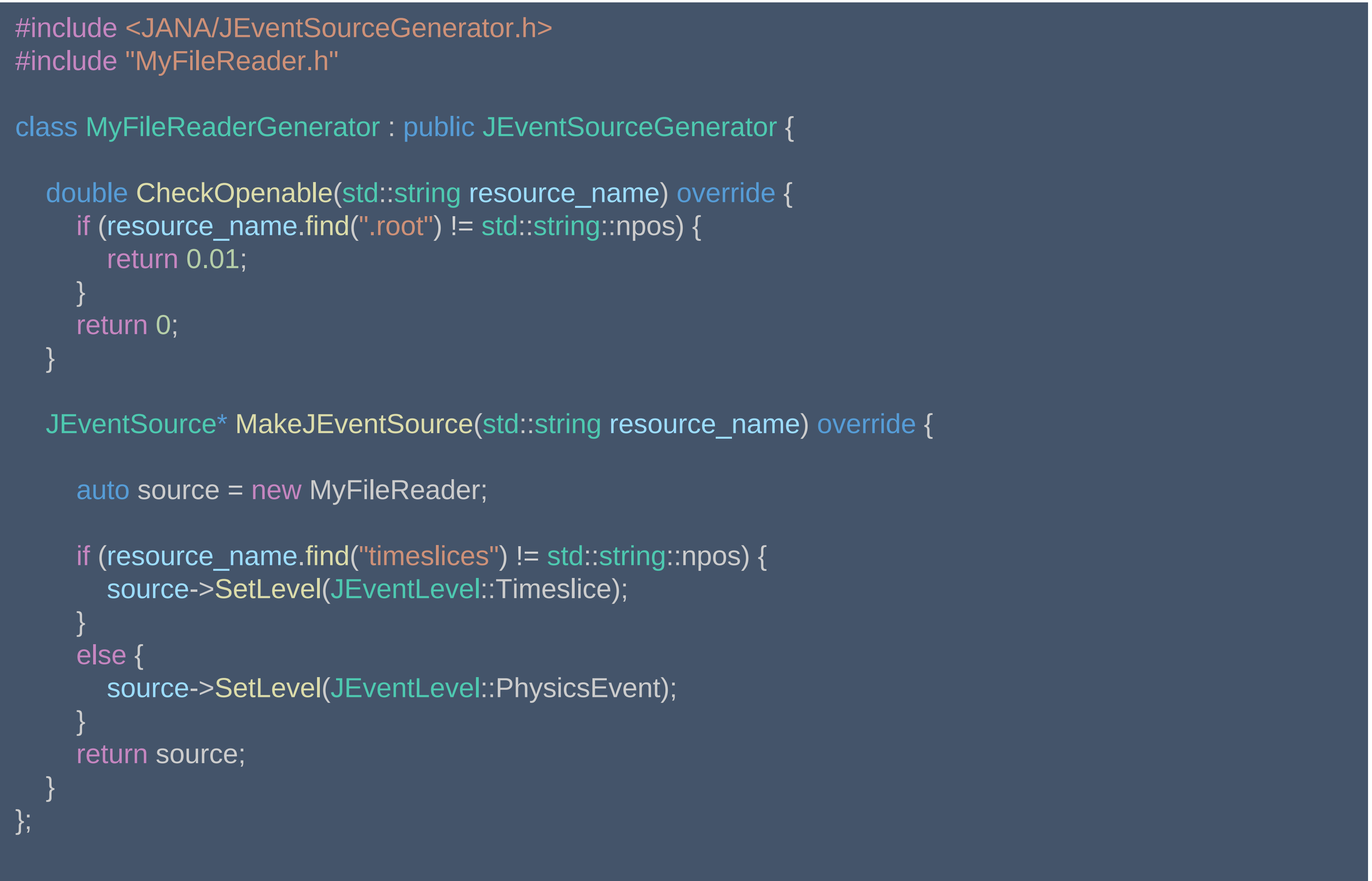
What does this mean for our Factories?

- 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;
    }
};
```

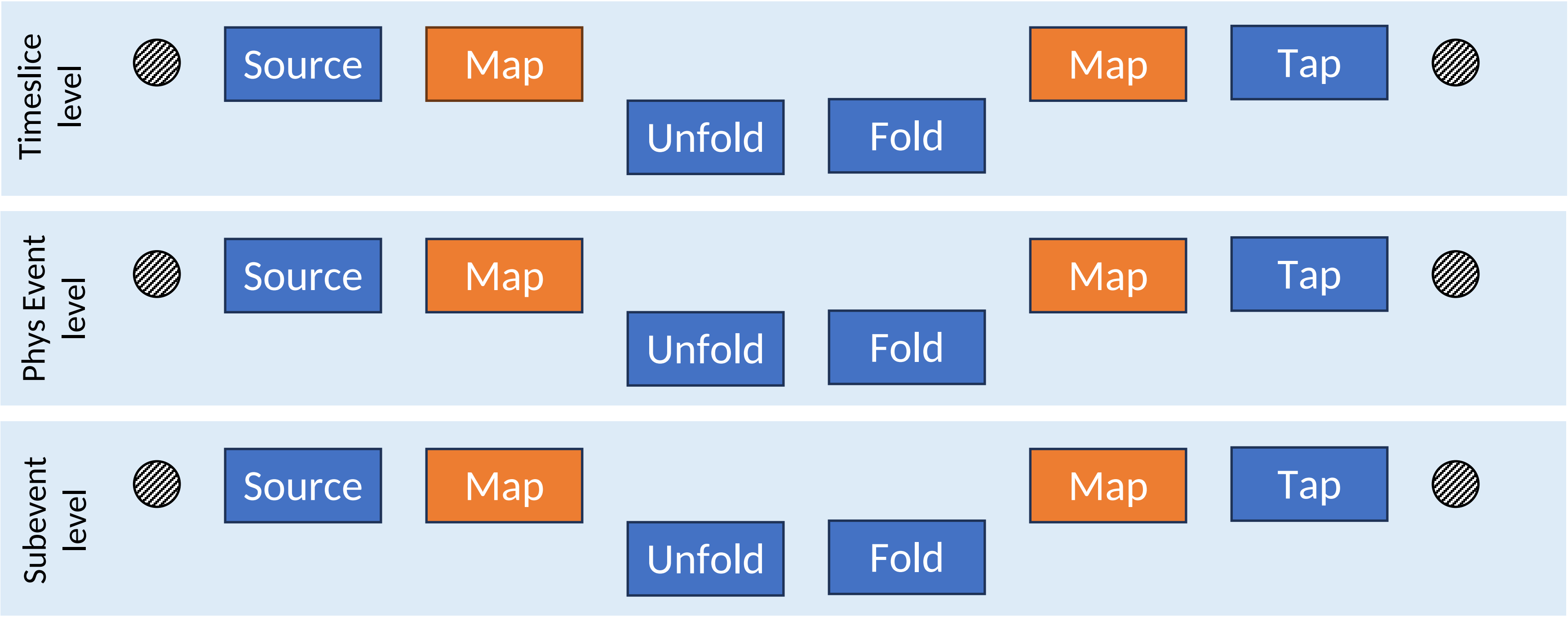
- JANA2 can figure out that the input file contains timeslices from inside the JEventSourceGenerator

- 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

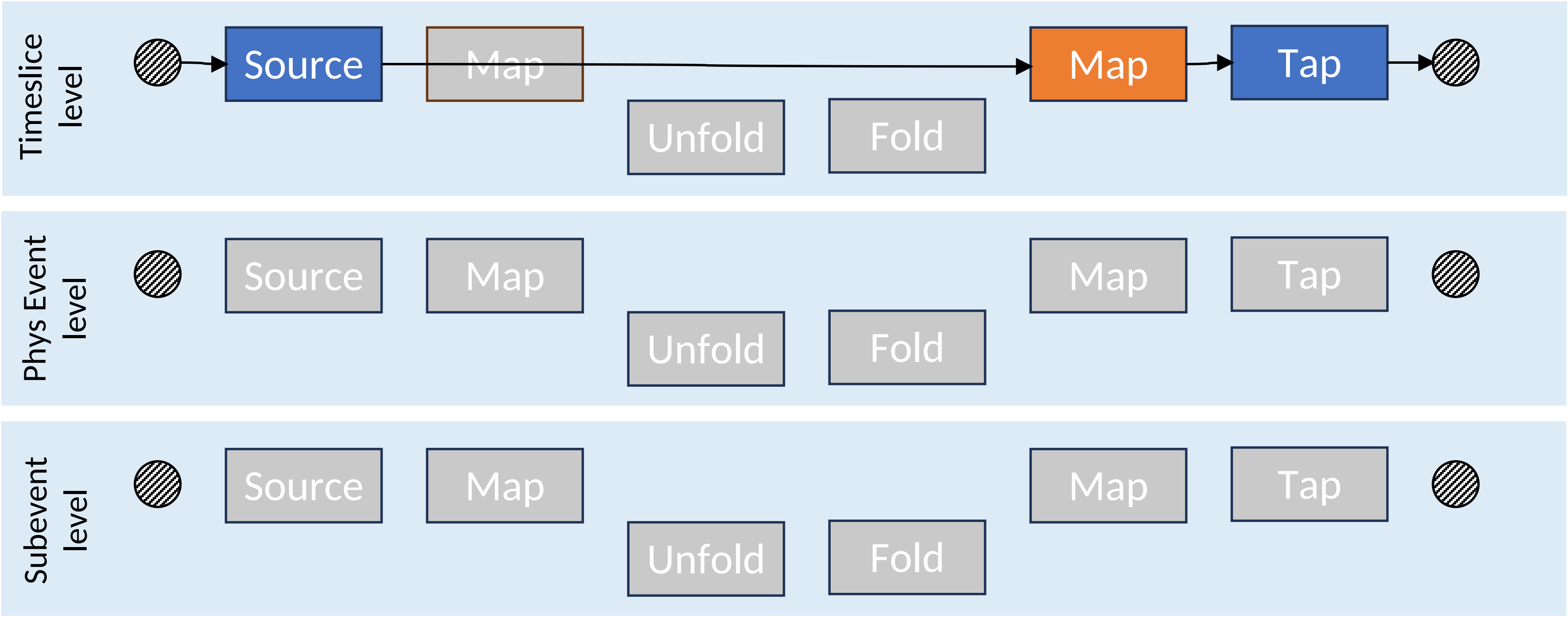
Generalizing further



- 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()

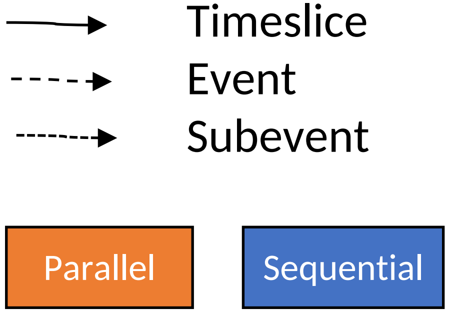
- The arrows in the further generalized topology (abstractly) form a grid:
  `{Source, Map1, Unfold, Fold, Map2, Tap} x {Timeslice, PhysicsEvent, Subevent,…}`
- Depending on which components the user provides, JANA2 can activate and wire the arrows automatically
- This wiring could also be specified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

Basic topology



User provides:

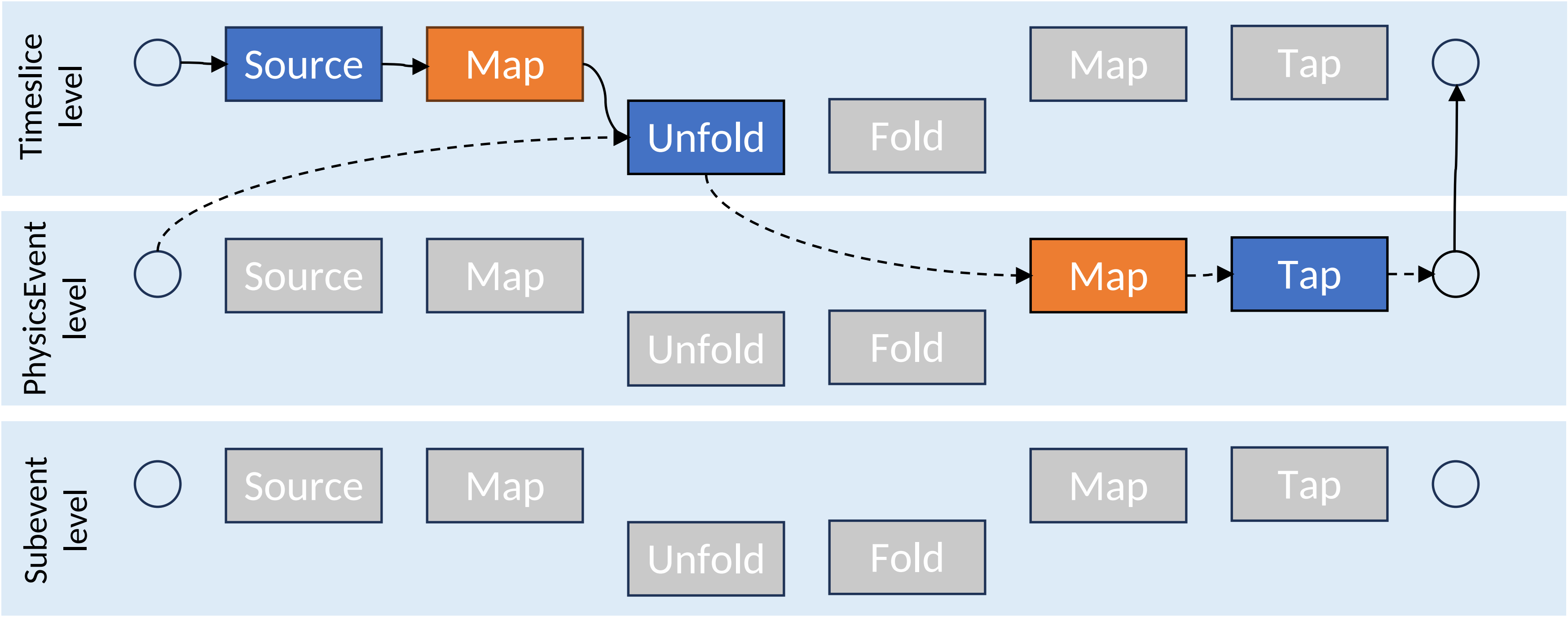
- JEventSource [Timeslice]
- JEventProcessor [Timeslice]
- JFactory [Timeslice]



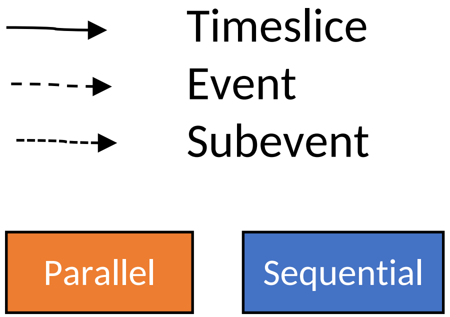
Timeslice splitting topology

User provides:

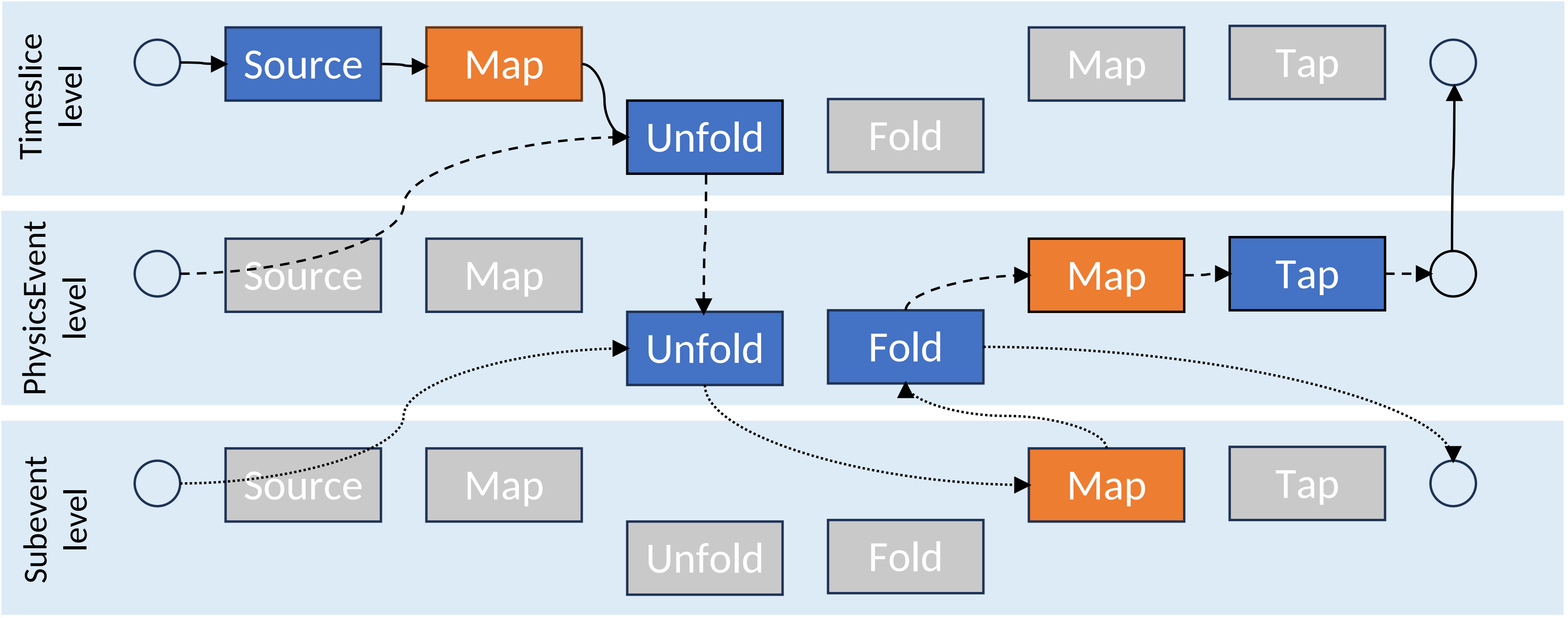
- JEventSource [T]
- JFactory [T]
- JEventUnfolder [T -> P]
- JEventProcessor [P]
- JFactory [P]



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



Timeslices + subevents topology



User provides:

- JEventSource [T]
- JEventProcessor [P]

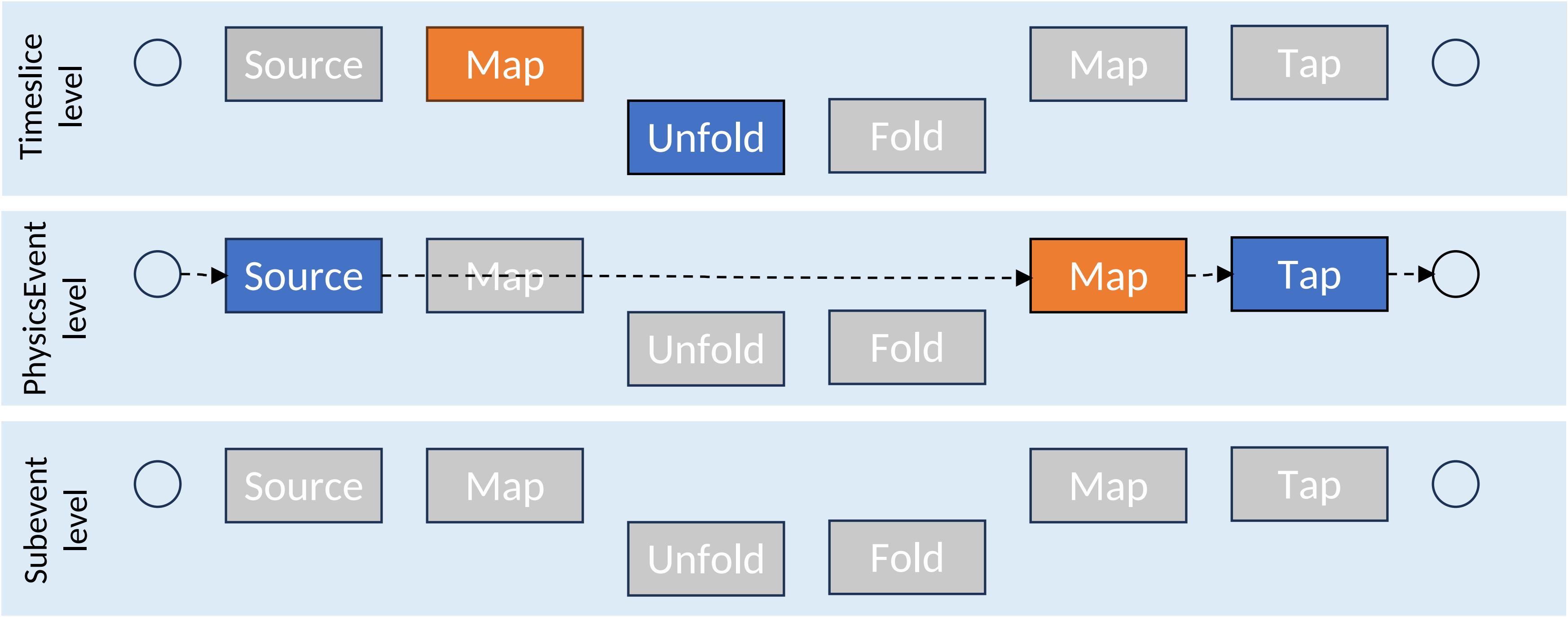
- JEventUnfolder [T -> P]
- JEventUnfolder [P -> S]
- JEventFolder[S -> P]

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

→ Timeslice
⇢ Event
⇢ Subevent

🟧 Parallel 🟦 Sequential

What happen if the user provides "extra" components?



User provides:

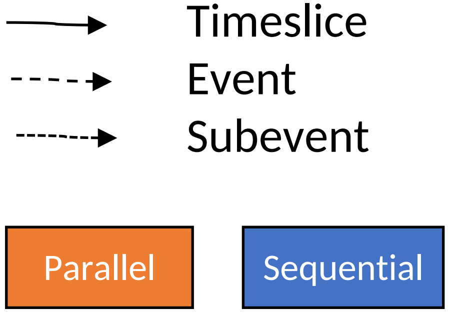
- JEventSource [P]
- JEventProcessor [P]
- JEventUnfolder [T -> P]

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

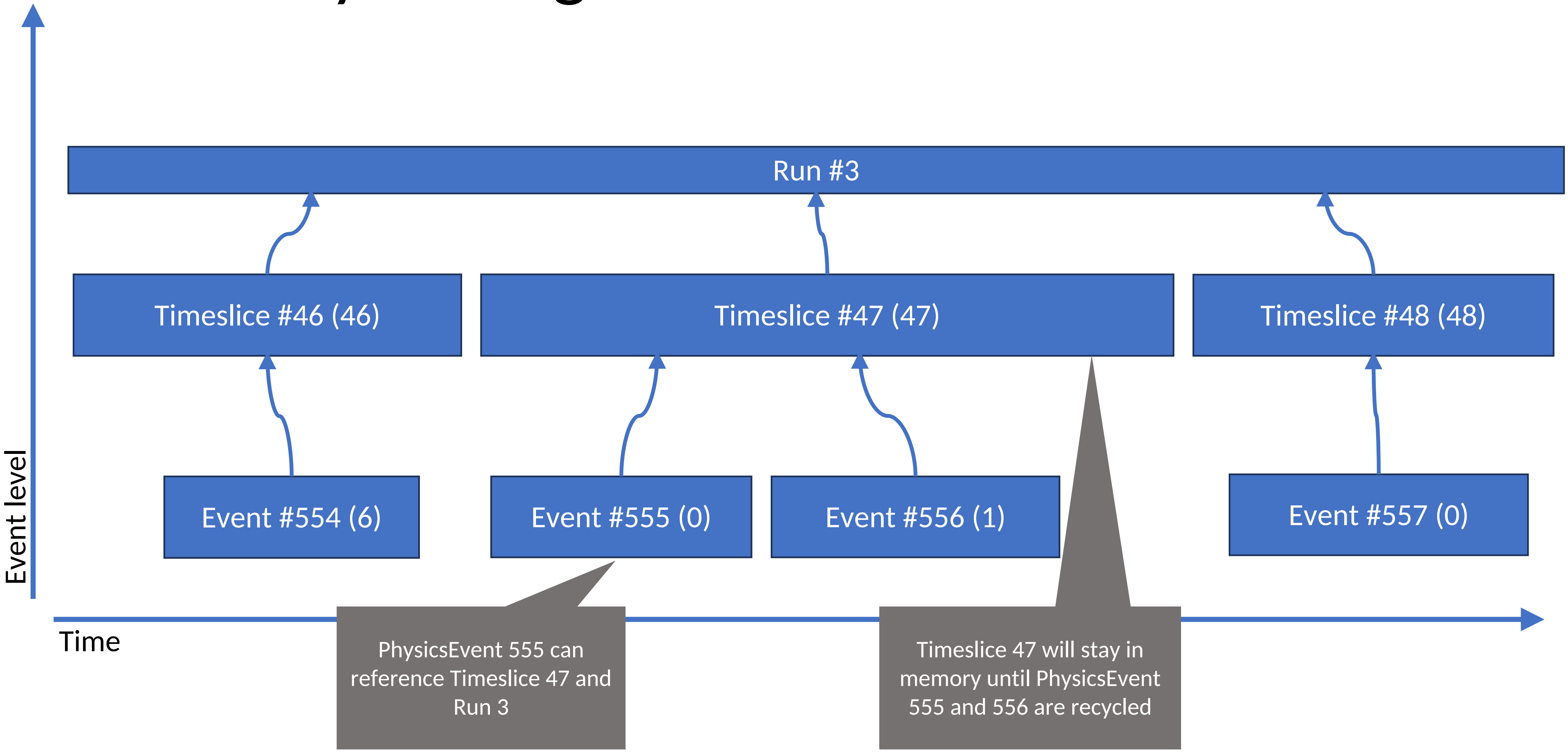
As of right now:

- 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

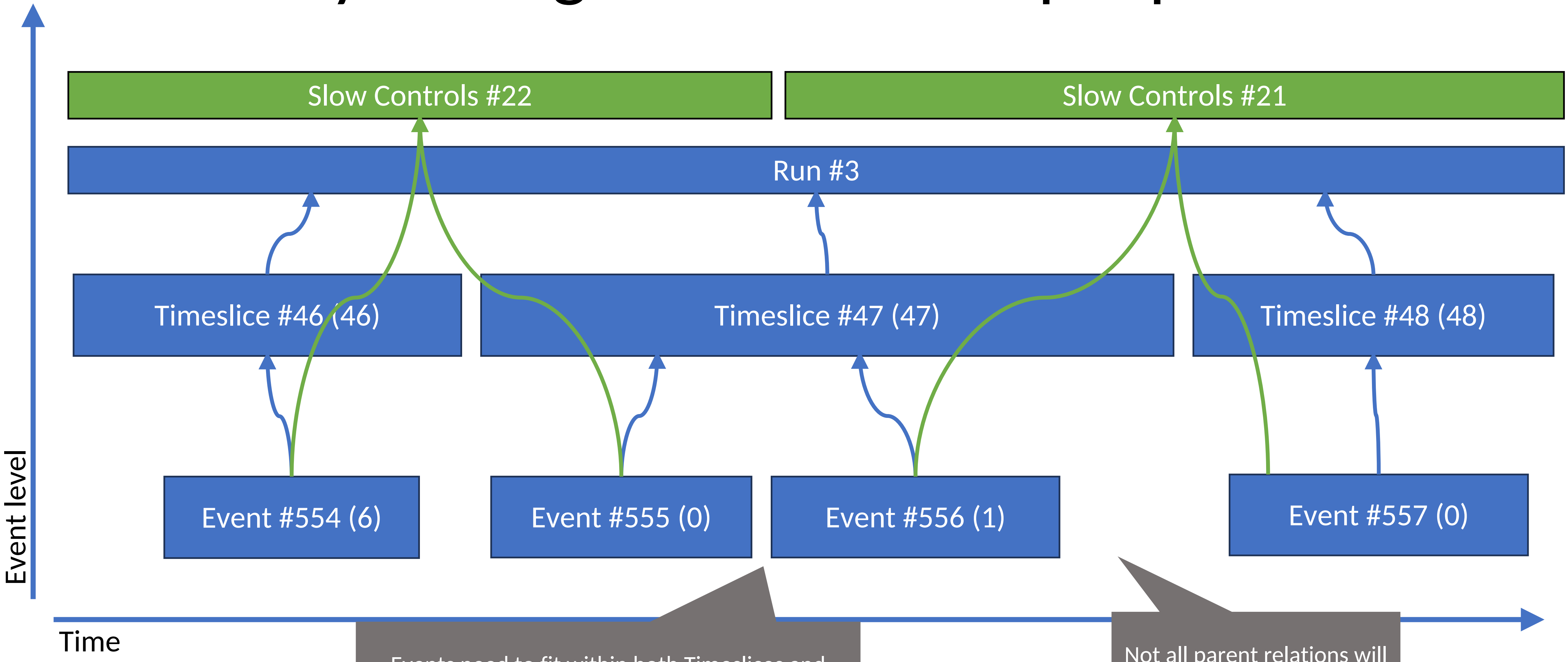
Memory management – Parent relation



PhysicsEvent 555 can reference Timeslice 47 and Run 3

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

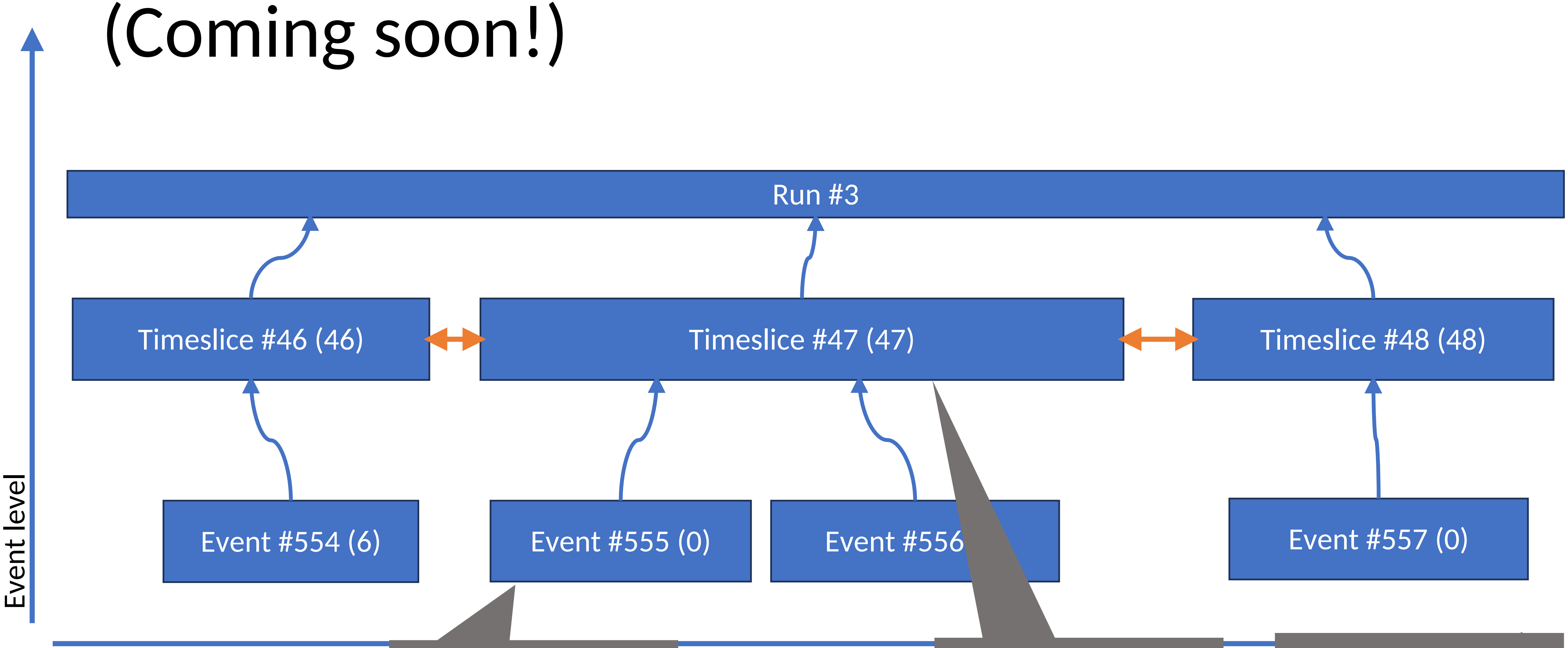
Memory management – Multiple parents



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!

Memory management – Sibling relations (Coming soon!)



PhysicsEvent 555 can reference Timeslice 47 *and* 46 and 48

Timeslice 47 has to stay around until both 46 and 48 are ready to be recycled

This is no longer mere reference counting!

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

Generalized event and run numbers

- Introduce *Event Key* abstraction which generalizes the concept of event and run number to streaming scenarios
- Will eventually replace the awkward arguments to JOmniFactory::Execute

- Event number: For each level **inside** our unfold/fold hierarchy, we can have:
  - Absolute number: Starts at 0, increments by 1
  - Relative number: Starts at 0 for each parent, increments by 1
  - User key: Could be anything, bunch crossing number in practice

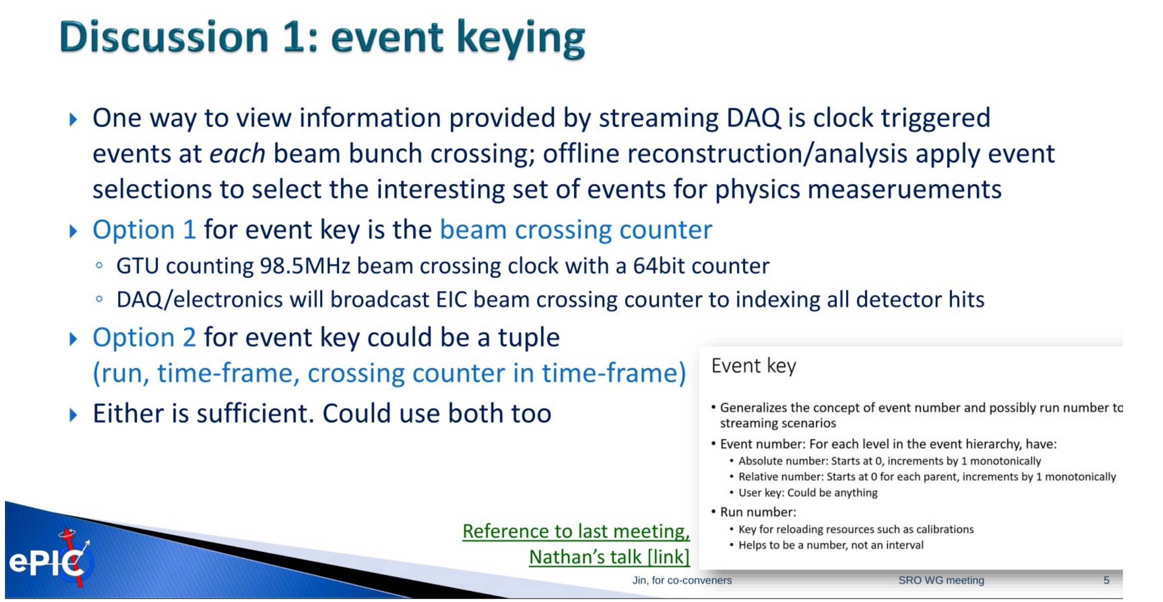
- Run number: Separate numbers for each parent level **outside** of the unfold/fold hierarchy
  - Key used for loading calibrations/conditions
  - Principle: Take advantage of the symmetry between “side-loading data from a database” and “retrieving data from events that live at a different level but were intermingled in the event stream”, e.g. BOR, slow controls
  - Might all end up being intervals of bunch crossing numbers in practice

ePIC’s event and run number

Notably productive discussion here, thanks to Jin Huang:

https://indico.bnl.gov/event/22949/

- Do NOT not to align time frame length with respect to the EIC beam rotation
- Primary event key is 64bit beam crossing counter
- Run structure driven by configuration changes; also continuous readout information on beam/detector monitoring



Discussion 1: event keying

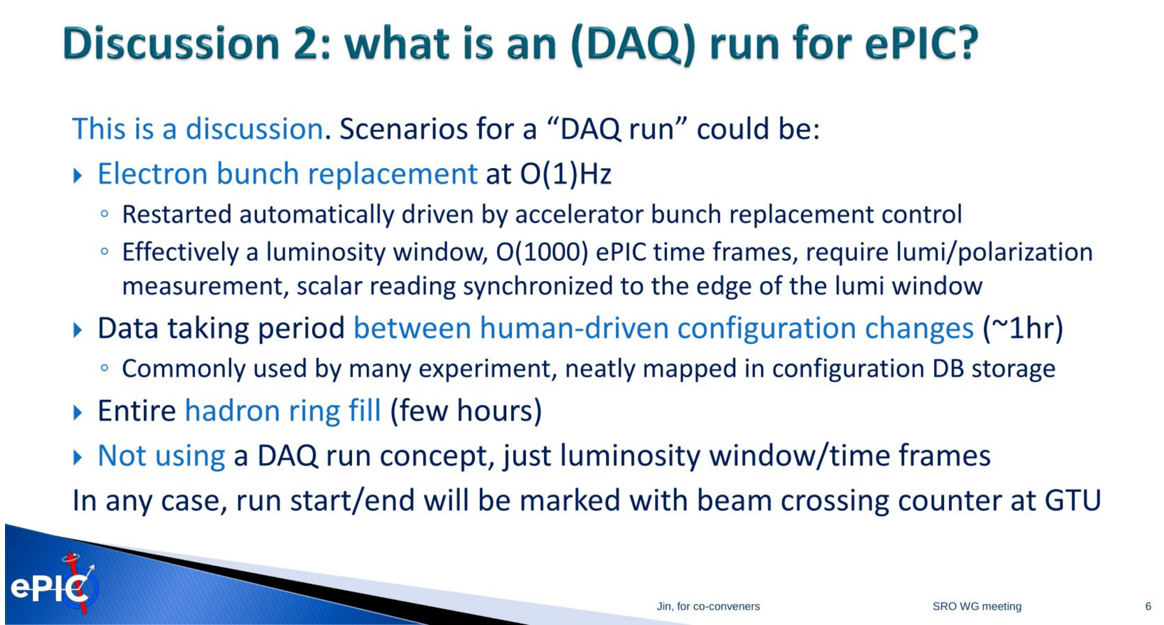
- One way to view information provided by streaming DAQ is clock triggered events at *each* beam bunch crossing; offline reconstruction/analysis apply event selections to select the interesting set of events for physics measeruements
- Option 1 for event key is the beam crossing counter
  - GTU counting 98.5MHz beam crossing clock with a 64bit counter
  - DAQ/electronics will broadcast EIC beam crossing counter to indexing all detector hits
- Option 2 for event key could be a tuple (run, time-frame, crossing counter in time-frame)
- Either is sufficient. Could use both too

Event key

- Generalizes the concept of event number and possibly run number to streaming scenarios
- Event number: For each level in the event hierarchy, have:
  - Absolute number: Starts at 0, increments by 1 monotonically
  - Relative number: Starts at 0 for each parent, increments by 1 monotonically
  - User key: Could be anything
- Run number:
  - Key for reloading resources such as calibrations
  - Helps to be a number, not an interval

Reference to last meeting,
Nathan’s talk [link]

Jin, for co-conveners SRO WG meeting 5



Discussion 2: what is an (DAQ) run for ePIC?

This is a discussion. Scenarios for a “DAQ run” could be:

- Electron bunch replacement at O(1)Hz
  - Restarted automatically driven by accelerator bunch replacement control
  - Effectively a luminosity window, O(1000) ePIC time frames, require lumi/polarization measurement, scalar reading synchronized to the edge of the lumi window
- Data taking period between human-driven configuration changes (~1hr)
  - Commonly used by many experiment, neatly mapped in configuration DB storage
- Entire hadron ring fill (few hours)
- Not using a DAQ run concept, just luminosity window/time frames

In any case, run start/end will be marked with beam crossing counter at GTU

Jin, for co-conveners SRO WG meeting 6

Event Key status

- A prototype is present in the JANA2 master branch
  - https://github.com/JeffersonLab/JANA2/blob/master/src/libraries/JANA/Utils/JEventKey.h

- However, not all component interfaces support it yet, and no components use it in practice
- You are welcome to poke at it and provide feedback, but I want to think it through a bit more before people start using it
- “Key” consideration is handling an *interleaved* event stream, e.g. slow controls
- Additional work needed to create an event source that emits events at more than one level.

Event classification/filtering

- Users should be able to *classify* an event. Components can then *filter* which events get processed based off of that classification
- Examples: DIS, DVCS, background, (slow controls, BOR)
- This functionality has always existed in JANA, but with flaws
- **Idea**: A new component, JEventClassification or JEventFilter, produces a classification analogously to how JFactories produce collections
- **Idea**: Factories, unfolders, folders, and JEventProcessors can declare that they short-circuit depending on one or more of these classifications

Challenges

- Classification needs to be represented and persisted in the data model
  - Data file closure/checkpointing: JANA2 should be able to read any data file it has written, and resume computation. This means that need to be either recomputable or cleanly extractable from the data model
  - This drives the decision to make JEventFilter be its own component type

- Event classifications are not inherently mutually exclusive.
  - Rationale: There may be multiple interactions in a single bunch crossing.
  - Rationale: Different classifications may be computed at different points in the compute graph. This plays nicely with both JANA2 and Podio's memory/mutability semantics
  - Intuition: "Event *contains* at least one DVCS interaction", not "event *is* a DVCS interaction". This also makes much more sense working with timeframes or other higher event levels, which contain many many interactions.
  - Problem: Skip("Background") behaves counterintuitively if both "Background" and "DIS" are set to true. Only way around this is to define `BackgroundOnly = !DIS and !DVCS and !SC`

Design decisions

- How complex does the short-circuiting logic need to be? Which constructs are most useful?
  - **`FilterAny(“DIS”, “DVCS”);`**
  - `Skip(“BackgroundOnly”); SkipAny(“BackgroundOnly”, “SlowControls”);`
  - `Filter(Not(“BackgroundOnly”))`
  - Lower bound is filtering for a single classification (no NOT, no Skip())
  - Upper bound on complexity is presumably Conjunctive Normal Form (AND of ORs)
- Are string types enough?
  - Other options are bitfields, wrapped bools (“strong types”), enums
  - Key consideration is open-world assumption due to plugin architecture
- Do we need consecutive “event numbers” for each classification? (I’m hoping not)
- Is “Classification/Filter” the best terminology? What other jargon would you use?
- How much of this lives in reconstruction, vs should be left up to analysis?

Thank you!