EPJ Web of Conferences **337**, 01189 (2025) https://doi.org/10.1051/epjconf/202533701189
CHEP 2024

Reconstruction framework advancements to support streaming for the ePIC detector at the EIC

Nathan Brei[1,*], *Torri* Jeske[1], and *David* Lawrence[1]

[1]Thomas Jefferson National Accelerator Facility

Abstract. The ePIC collaboration adopted the JANA2 framework to manage its reconstruction algorithms. This framework has since evolved substantially in response to ePIC's needs. There have been three main design drivers: integrating cleanly with the Podio-based data models and other layers of the key4hep stack, enabling external configuration of existing components, and supporting timeframe splitting for streaming readout. The result is a unified component model featuring a new declarative interface for specifying inputs, outputs, parameters, services, and resources. This interface enables the user to instantiate, configure, and wire components via an external file. One critical new addition to the component model is a hierarchical decomposition of data boundaries into levels such as Run, Timeframe, PhysicsEvent, and Subevent. Two new component abstractions, Folder and Unfolder, are introduced in order to traverse this hierarchy, e.g. by splitting or merging. The pre-existing components can now operate at different event levels, and JANA2 will automatically construct the corresponding parallel processing topology. This means that a user may write an algorithm once, and configure it at runtime to operate on timeframes or on physics events. Overall, these changes mean that the user requires less knowledge about the framework internals, obtains greater flexibility with configuration, and gains the ability to reuse the existing abstractions in new streaming contexts.

1 Introduction

ePIC is the primary experiment at the Electron-Ion Collider [1], a highly integrated and multi-purpose experiment featuring state-of-the-art detectors and computing [2, 3]. One key requirement is a tight integration between detector and compute, with the turnaround from readout to analysis being just 2-3 weeks. This timeline is driven by alignment and calibrations. Three intersecting technologies are leveraged to meet this requirement: streaming readout, artificial intelligence, and heterogeneous computing.

Adopting a streaming data processing paradigm has wide ranging architectural consequences. With a traditional architecture, data is acquired in an online workflow and stored as large files in hierarchical storage. The processing of the data happens in a separate, offline workflow. This decoupling between online and offline allows for discrete, coarse-grained data units, which in turn allows for batch queue-based resource provisioning. In contrast, with a streaming architecture, fine-grained data is collected quasi-continuously. As the data inflow

*e-mail: nbrei@jlab.org

rate may vary, dynamic resource scheduling is necessary. Furthermore, fast processing of the entire dataset is necessary for alignment and calibration and for delivering analysis-ready data. In exchange for this additional complexity, streaming provides some essential advantages: the readout is much simpler, as no custom trigger hardware or firmware is required, and events may be built with holistic detector information. Ultimately these give physicists a deeper knowledge of backgrounds and an enhanced control of systematics.

The ePIC collaboration chose JANA2 [4] as its reconstruction framework. JANA2 is a scalable, modern C++ reconstruction framework that uses dataflow parallelism in order to support both traditional and streaming processing. It is a rewrite of JANA, which was developed for the GlueX experiment at Jefferson Lab. This meant the codebase was proven enough to be usable on day 1 for processing simulated data and developing the chain of reconstruction algorithms, while at the same time being clean and malleable enough to be heavily adapted for ePIC's future streaming needs. This enabled a unified software stack for different groups within ePIC operating on different timescales.

JANA2 has already undergone a variety of improvements in response to ePIC's needs, including cleanly integrating with Podio, and factoring out component configurations for better code reuse. The focus of this paper is on building out the interfaces for streaming readout, specifically those for performing reconstruction on timeslices and for splitting the timeslices into physics events, although the result has much wider generality.

2 Background

The JANA2 reconstruction framework is organized into two layers: a 'component' frontend layer and a 'processing topology' backend layer. *Components* are a family of interfaces for the user to implement. They are meant to be independent from each other; all communication between them is expressed through the data model. For this purpose, JANA2 provides a container called JEvent, which manages collections of data model objects. Crucially, a JEvent represents data that can be processed as a discrete, independent unit, and JANA2 uses it as its unit of parallelism. Traditionally, JEvents were used to represent physics events, but this changes substantially in the context of streaming readout.

The most commonly used component interfaces are JEventSource, JFactory, and JEventProcessor. A JEventSource reads from a file or socket and inserts the data into a JEvent. Thus users create a JEventSource for each input file or message format. A JFactory operates on an individual JEvent, running an algorithm to compute new results using the existing data (which might have come from the JEventSource or from other JFactories), and adding them to that JEvent. Finally, a JEventProcessor writes data from a JEvent to a file or socket. If that data hasn't already been created, JANA2 will run the corresponding JFactories to produce it. Together, these components are enough to create a basic reconstruction chain.

Under the hood, JANA2 builds a dataflow-parallel processing topology consisting of arrows, queues, and pools. Arrows represent fixed tasks that modify JEvents using the user-provided components, and may run either sequentially or in parallel. Rather than having a direct connection to each downstream arrow, the connection between two arrows is intermediated by a queue. This allows asynchronous processing, so that no worker thread is left waiting for a task to finish when there are other tasks available. The entry and exit vertices of the topology are pools, which recycle JEvents in order to avoid expensive initializations and to control the total number of in-flight events (and hence memory consumption). This setup is inspired by Kahn Process Networks [5].

The processing topology paradigm that JANA2 uses under the hood was designed for streaming readout from the beginning and had already been integrated successfully into an SRO project with TriDAS [6]. What it lacked was a coherent concept at the user-facing layer:

Components such as factories, processors, and sources could only operate on physics events, and everything else was implemented as custom arrows. Apart from being inconsistent, this design exposed a great deal of internal complexity, resulting in a higher learning curve and surface area for bugs. It also conflicted with JANA2's plugin architecture, as individual plugins cannot directly modify the processing topology while staying independent of each other. The ePIC streaming readout project presented a long-awaited opportunity to improve this design.

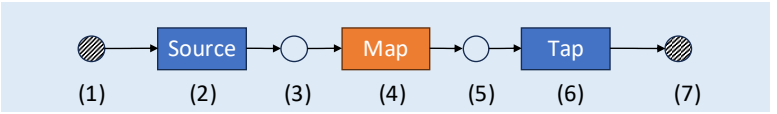

Figure 1. The simplest JANA2 processing topology. (1, 7) A pool of empty events. (2) An arrow that reads a (raw) event sequentially using a JEventSource. (3, 5) Event queues. (4) An arrow that calculates reconstruction results in parallel using JFactories. (6) An arrow that writes a (reconstructed) event sequentially using a JEventProcessor.

Figure 1 shows the typical topology that is used for traditional physics-event processing. It consists of three arrows, Source, Map, and Tap. JEvents start their life in the pool on the far left. When JANA2 fires the Source arrow, an empty JEvent container is popped from the pool. It is passed to the `JEventSource::Emit` callback which attempts to fill it with data from file. If that succeeds, the JEvent is pushed onto the downstream queue; otherwise it is returned to the pool. Next JANA2 fires the Map arrow, which calls `JEventProcessor::ProcessParallel()`, which in turn calls any necessary JFactories. Unlike the Source arrow, which is constrained to one worker thread at any time, multiple worker threads may run Map. Map always succeeds, upon which the JEvent is pushed downstream, triggering Tap, which calls `JEventProcessor::ProcessSequential()` to write the results for that event to file (sequentially). Finally Tap clears the JEvent and pushes it back to the pool where it originated.

3 Design Principles

Several key design principles were employed during this work. The first is preserving symmetry between interfaces. In the simplest case, an algorithm that might have been written for physics events might very well be repurposed for timeframes. The framework should handle both cases with maximal consistency and minimal reconfiguration. Similarly, timeframe or 'DAQ event' sizes are often driven by hardware details, and don't necessarily have a physical meaning from the reconstruction's perspective. The framework must be able to accommodate these without imposing constraints upstream.

A second principle is modeling the data stream as a (mostly) time-ordered sequence of signals. Informally we can think of these as 'hits', although within JANA2 they generalize to include reconstructed quantities such as tracks. In the most basic SRO setup, timeframes are a partitioning of this sequence, and physics events are a partitioning of the timeframe. This cleanly generalizes as a hierarchy of subsets, where the exact levels in the hierarchy depend on the particular experiment. In the simplest case, event building consists of drawing vertical bars on the time axis, as shown in Figure 2. However, there are two more complicated scenarios the framework must handle. Firstly, when detectors have different response latencies, the partition boundaries need to be shifted in accordance with each detector's physics.

This requires adding a new component interface for expressing this logic to the framework. Secondly, events may overlap, or experience 'edge effects', e.g. where part of a physics event is stranded on the wrong side of a timeframe boundary. This has consequences for data lifetimes, but nonetheless remains compatible with an event hierarchy.

A third principle, drawn straight from the ePIC Software Statement of Principles, is user-centered design. There are several aspects to this. One concrete goal is code reuse: Just as one factory implementation should be reusable for multiple detectors, so too should it be able to operate on either a timeframe or a physics event. Another goal is to enable configuration that doesn't require recompilation: ideally, the full ePIC reconstruction chain could be rewired from a text file. These goals dovetail with the stylistic choice within the JANA2 codebase to minimize use of templates and type system trickery for the sake of easier debugging. Together, they steer the design away from a sophisticated representation of an event hierarchy within the type system.

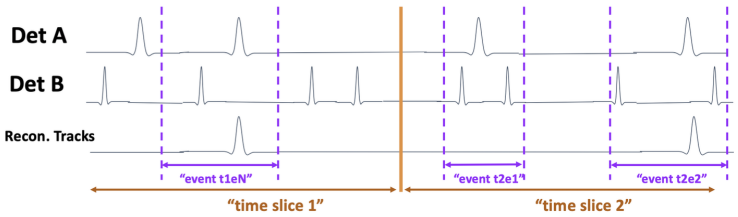

Figure 2. A cartoon view of event building, wherein a stream of detector hits and corresponding reconstructed values is recursively partitioned. This gives rise to the event hierarchy.

4 Event Levels

The first step to generalizing the JANA2 framework for streaming readout is to understand that JEvent represents a container for any data that is meant to be processed as a discrete independent unit, and does not always contain a physics event. Thus a JEvent can represent physics events and timeframes, but in principle also any other partitioning of the data stream. JEvents are tagged as belonging to a particular *event level*. These event levels form a hierarchy, which may be experiment-specific. The predefined event levels are:

```
enum class JEventLevel {
    Run, Subrun, Timeslice, Block, SlowControls, PhysicsEvent, Subevent, Task
}
```

While Timeslice, Block, PhysicsEvent, and Subevent are obvious, the rest deserve some explanation. Tasks are distinct from Subevents because Subevents may be understood to have a particular physical meaning in the context of a particular experiment, e.g. interactions, whereas Tasks specifically have no physical meaning and simply indicate very fine grained data to be processed in parallel. Run and SlowControls are more interesting. Both traditionally represent an *interval of validity* for data which is usually either side-loaded from a database or interleaved with the event stream. Both of these cases, however, are isomorphic with a streaming merge. Representing such intervals of validity explicitly within the framework allows for unified and considerably simpler handling of all three cases, and prevents users reaching for more brittle and less performant mechanisms, such as 'barrier events'.

Different experiments use different event hierarchies, and possibly use the same terms inconsistently. For instance, in some experiments, a Block might be bigger than a Timeslice, and there might be additional levels such as SuperTimeframe or DAQEvent. Consequently JANA2 supports both user-defined event levels and user-defined event level orderings. An additional complication is that event levels might only have a *partial* ordering. For instance, consider a PhysicsEvent that belongs to one Timeslice and one SlowControls interval, but the Timeslice and SlowControls have no relationship to each other.

5 Hierarchical data access and lifetimes

Having defined an event hierarchy, the next step is providing child events access to their parents' data. To prevent races, parent-level data must be immutable, and must have already been calculated upstream rather than on-demand. Because hierarchies are only partially ordered, any JEvent may have multiple parent events, though at most one at each level. For simplicity of implementation and clarity, the child JEvent does not 'inherit' the data collections of its parent. Instead, the caller must request the data from the parent explicitly.

Ownership and lifetimes of parent data are tricky. Any JEvent may either be independent, or be the parent of at least one other JEvent. If it is independent, its owner is whatever arrow or queue or pool currently has a unique reference to it, and its data gets cleared whenever it gets returned to the pool, so its data lifetime is the same as its in-flight time. If the JEvent is the parent of another JEvent, however, its ownership is shared among the children, and its lifetime must cover the lifetimes of all child events. Only when the last child event is returned to the pool, can the parent event be cleared and recycled as well. For this purpose JANA2 uses an internal reference counter. However, the condition of all children being finished is not sufficient: there might be more child events within the parent's interval of validity that have not yet been emitted. Thus there is an additional flag to indicate when a JEvent has been fully 'released' by its earlier owner so that the reference counting may come into play. The component interfaces are designed to keep this complexity encapsulated inside JANA2 itself.

The current design supports overlapping events by allowing references to hit objects to be handed over to multiple child events. However, it does not support edge effects, e.g. where part of a physics event is at the wrong side of a timeslice boundary. The design could be extended to handle this case, at the expense of additional complexity, by adding *sibling* relations analogous to parent relations, and deferring recycling until both siblings are also ready to be recycled. This way, a physics event would have access not only to its own parent timeslice, but also the adjacent timeslices. A simpler solution at the user level is to clone the boundary data from the timeslices on either side, at the cost of higher memory usage.

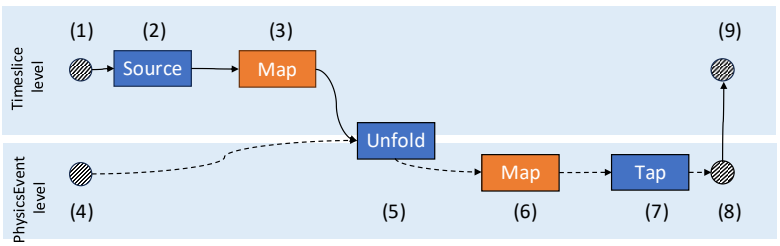

Figure 3. An example timeslice-splitting topology.

6 Unfolders and Folders

Figure 3 depicts a simple processing topology that accepts timeslices and splits them into physics events. While in the previous example, the source emitted PhysicsEvents, now it emits Timeslices. The first Map arrow runs any timeslice-level factories needed for splitting. A new arrow, named Unfold, merges the streams of full Timeslices and empty PhysicsEvents, producing a new stream of full PhysicsEvents that have a Timeslice as a parent. These are then passed to Map, which runs any PhysicsEvent-level factories needed for reconstruction. Finally, Tap writes out the reconstructed PhysicsEvents, and potentially the parent Timeslices as well. The PhysicsEvent is returned to its pool, and once the last PhysicsEvent for a given Timeslice is returned, that Timeslice is returned as well.

The key feature here is a new component interface, JEventUnfolder, which is JANA2's abstraction for splitting a stream of parent events into child events. It takes its name from functional programming, where it is a common pattern for operations that start with an initial value and repeatedly apply a function to generate new values until a termination condition is met. This terminology has also been adopted by FermiLab's Meld framework [7] for the same purpose. JANA2's implementation is slightly different than its functional programming namesake because all JEvents are pooled and recycled. Thus it consumes as input *two* streams (full parents and empty, parentless children) and produces one stream (full, parented children). JEventUnfolder's principal callback signature is:

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

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

Here, the parent event must be immutable because there may be other child events referencing it downstream. The child event must be mutable, on the other hand, because the JEventUnfolder needs to insert data collections into it, and there are no other references to the child event in the whole topology. The child's index, i.e. the number of children already emitted downstream with the same parent, is provided for convenience. The `Result` enum expresses termination but also memory ownership transfer, while keeping the implementation details out of the hands of the user. Intuitively it can be understood as "at least one of the two events will be released downstream." Splitting can be accomplished using just NextChildNextParent and NextChildKeepParent. The KeepChildNextParent case enables JEventUnfolder to be used for *joining* two (already full) streams (e.g. SlowControls and Timeslice) rather of splitting one into the other.

This callback signature was designed to minimize memory used at any point in time to at most one parent and one child. It also preserves liveness: the component will not deadlock even if the user configures JANA2 to run with exactly one parent and one child event in-flight. The Unfolder arrow runs sequentially because the JEventUnfolder component needs local state in order to track its progress within the parent event. Unlike other JANA2 components, there can only be one JEventUnfolder at a given level in a topology. This is because of the termination logic: while many JEventUnfolders could all theoretically modify a child event by looking at its parent event, only one gets to decide when the parent event has been fully processed.

The 'unfold' combinator has an inverse, 'fold' or 'reduce', which aggregates results from child events and appends them to the parent event. For instance, subevent-level results would be reduced into physics-event-level results for building histograms. (Arguably the histogram itself is conceptually a reduction from physics-event-level results to run-level results, although this is not how it is implemented in practice.) To do these reductions, JANA2 provides

JEventFolder. JEventFolder consumes one stream (of parented full children) and emits two streams (parents, and unparented children). Its principal callback signature is:

```
virtual void Fold(const JEvent& child, JEvent& parent, int child_index);
```

JEventFolder is simpler than JEventUnfolder because it doesn't need a termination condition. This also means that multiple JEventFolders at the same level could in principle be used simultaneously. The `Fold` arrow runs sequentially because the JEventUnfolder maintains local state. This local state is necessary due to the collection immutability constraints imposed by Podio. Waiting until the final child event has completed before writing accumulated collections to the parent event is desirable for correctness, as otherwise, user code could attempt to read collections upstream while they are being written downstream. While JEventFolder guarantees liveness, it has mildly less predictable memory usage than JEventUnfolder, because it may accumulate results from children with multiple parents at once.

7 General Processing Topologies

Figure 3 depicted a timeslice-splitting processing topology. The user-provided JEventSources, JEventProcessors, and JFactories have just one modification compared to their counterparts in the traditional processing topology of Figure 1 – they are tagged as belonging to an event level. As it turns out, the *wiring*, or connections between the arrows, can be determined automatically, so that no further configuration is required. This holds true for more complex processing topologies as well. Figure 4 shows a more complex topology that uses timeslices, physics events, and subevents.

The algorithm works on an imaginary grid of possible arrows. Each row in the grid is a level in the user's event hierarchy. Within each row, the possible arrows always have the ordering `Source, Map, Unfold, Fold, Map, Tap` and the only way to move between levels is via `Unfold` and `Fold`. Thus JANA2 can collect all user-provided components, sort them by level, and connect them in a single pass. This minimizes the necessary configuration and plays well with JANA2's plugin architecture. It has been demonstrated that JANA2 can correctly wire up either the traditional processing topology or the timeframe-splitting topology based only off of a flag in the input file. Nevertheless, there are certain cases where the wiring is ambiguous, and advanced users may wish to add custom arrow types which the algorithm cannot handle. For these cases, JANA2 allows the wiring to be specified explicitly.

8 Conclusion

The ePIC detector will use a streaming readout architecture. This places additional requirements on the reconstruction framework, in particular the ability to read and process timeframes, split them into physics events, and process and write the physics events. The ePIC collaboration chose the JANA2 reconstruction framework since it could be used in production immediately, while meanwhile being heavily adapted for ePIC's longer-term streaming needs. One major set of improvements introduced the concept of an event hierarchy and added the ability to use existing JANA2 components at different event levels. To move between streams of different levels, this added two new component abstractions, Folder and Unfolder, which among other uses, allow the user to express arbitrary logic for splitting timeframes into physics events. The overall design allows JANA2 to automatically wire the correct processing topology based off of the components that the user has provided, up to and including the input file itself, thereby minimizing extra configuration. Future work includes extending this design to explicitly support components that offload their computations, particularly machine learning tasks, onto heterogeneous hardware such as GPUs.

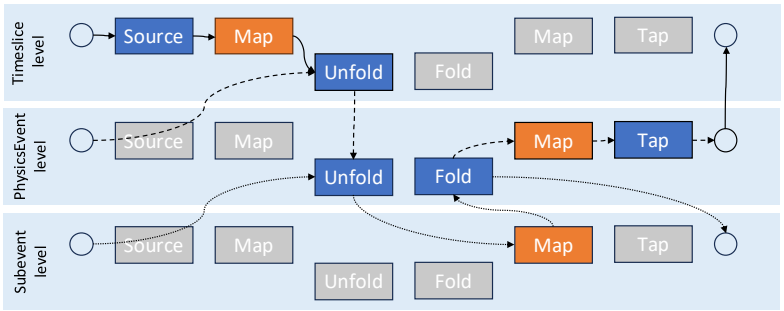

Figure 4. An example of a subevent topology. Here, the user provided a Timeslice-level JEventSource, a Timeslice-to-PhysicsEvent JEventUnfolder, a PhysicsEvent-to-Subevent JEventUnfolder, a Subevent-to-PhysicsEvent JEventFolder, and a PhysicsEvent-level JEventProcessor. JANA2 was able to activate the correct arrows and wire the topology automatically. The greyed-out boxes indicate components that were *not* provided by the user, but form the abstract grid used by the wiring algorithm.

9 Acknowledgements

This work was produced in part by Jefferson Science Associates, LLC under Contract No. AC05-06OR23177 with the U.S. Department of Energy. Publisher acknowledges the U.S. Government license and provide public access under the DOE Public Access Plan [8].

References

[1] R. Abdul Khalek, A. Accardi, J. Adam, D. Adamiak, W. Akers, M. Albaladejo, A. Albataineh, M. Alexeev, F. Ameli, P. Antonioli et al., Science requirements and detector concepts for the electron-ion collider: Eic yellow report, Nuclear Physics A **1026**, 122447 (2022). https://doi.org/10.1016/j.nuclphysa.2022.122447

[2] Brookhaven National Laboratory, The epic collaboration, accessed: February 27, 2025, https://www.bnl.gov/eic/epic.php

[3] Lawrence, David, Eic software overview, EPJ Web of Conf. **295**, 03011 (2024). 10.1051/epjconf/202429503011

[4] Lawrence, David, Boehnlein, Amber, Brei, Nathan, Jana2 framework for event based and triggerless data processing, EPJ Web Conf. **245**, 01022 (2020). 10.1051/epjconf/202024501022

[5] G. Kahn, The Semantics of a Simple Language for Parallel Programming, in *Proc. IFIP Congress on Information Processing*, edited by J.L. Rosenfeld (North-Holland, 1974), ISBN 0-7204-2803-3

[6] F. Ameli, M. Battaglieri, V.V. Berdnikov, M. Bondí, S. Boyarinov, N. Brei, A. Celentano, L. Cappelli, T. Chiarusi, R. De Vita et al., Streaming readout for next generation electron scattering experiments, The European Physical Journal Plus **137**, 958 (2022). 10.1140/epjp/s13360-022-03146-z

[7] Knoepfel, Kyle, Meld exploring the feasibility of a framework-less framework, EPJ Web of Conf. **295**, 05014 (2024). 10.1051/epjconf/202429505014

[8] http://energy.gov/downloads/doe-public-access-plan