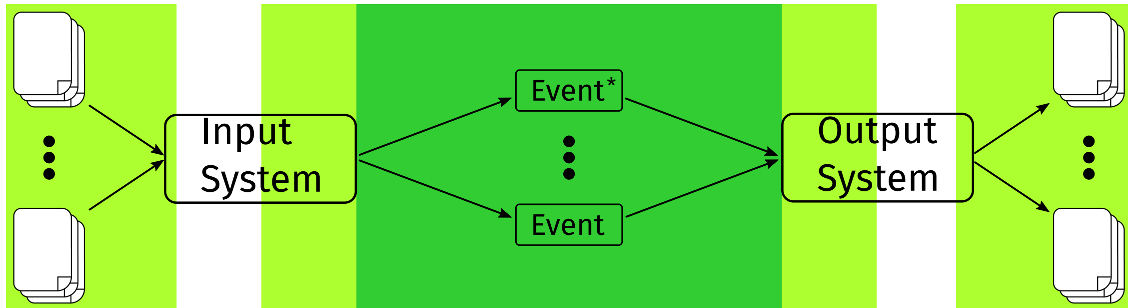
podio I/O re-design

Some diagrams and thoughts

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

(Very) high-level overview



- **"Thread safe by default"** (Assuming one event per thread!)
  - Users do not need to worry about threads at all, Event is fully self-contained
- **Potentially thread safe**
  - Up to the implementation to do necessary internal synchronization
  - Potential to have some of this in the "thread safe by default" category

Event

Event

Requirements

- The “Event” is not necessarily a user facing class!
  - In framework usage it could also just be a simple container of collections (or possibly even just something that acts like this)
- If it is a user facing class it gives access to the collections and manages their “access rights” and resources
  - Getting already stored collections
  - Storing new collections
  - A “mini framework” could potentially built around this class, where it is (to first order) the only thing that needs to be passed around
- Access to metadata
  - Definitely for event-level metadata
  - Also for run metadata and collection metadata?

Event Interface

Conceptually

```
struct Event {
// get a collection via an id
template<class CollectionT, class IdT>
CollectionT const& get(IdT id) const;

// put a collection into the event
template<class CollectionT, class IdT>
void put(CollectionT&& coll, IdT id);

// create a new, empty collection
template<class CollectionT, class IdT>
CollectionT& create(IdT id);

// ... more convenience functionality
};
```

- Event owns all the collections / data
- Access only const references to already stored collections
- `put`ting a collection into the Event makes it “invalid” for further outside usage.
  - At compile time enforce that the calling site reflects this, e.g. `event.put(std::move(collection))`[1]
- Only `create` gives access to a new, empty and mutable collection

[1]Potentially make collections a move only type?

NOTE: `IdT` is a placeholder for any type that can be used for uniquely identifying a collection.

Event Interface

Some concrete questions

- Support different policies for adding / creating collections?
  - E.g. only possible to `put` or `create` collections with a pre-defined (registered) set of ids?
  - Essentially a restriction imposed on the event by the output system.
  - Compile time enforcement possible? What is the earliest point we can fail in case of a mismatch at runtime?
- Define an abstract `IEvent` interface that can be implemented by different event classes?
  - Makes certain compile time checks impossible
  - Unified user interface, independent of the details of the actually used event without having to template functions
  - Would allow to hide a lot of implementation details
- How to best enforce “invalidation” of collections that are `put` into the event?
- Event level metadata? Metadata implementation in general.
- ...

Input System

Input System

Requirements

- Support in principle arbitrary I(/O) backends
  - `root`, `sio`, generator files?, ...
- Support reading from different sources "simultaneously" and combining them into events
  - Pile-up mixing, background overlay, ...
- Schema evolution when reading files using a different version of the EDM
- Potentially produce different user facing Event classes
  - lazily (self-)unpacking, ...
- Make as much as possible thread safe
  - Do not foresee multithreaded access to a single file, but potentially read several files in different threads?

Which are the ones we need/want to support for standalone podio? Which are better left for frameworks?

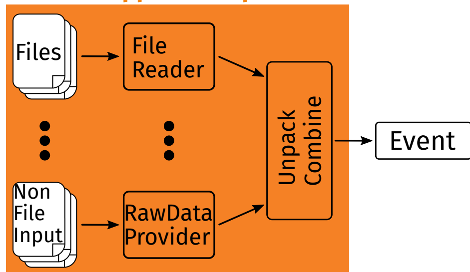
Input System

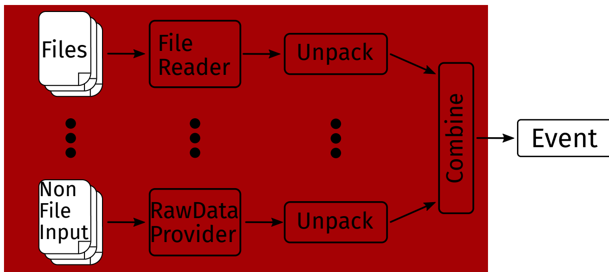
General remarks

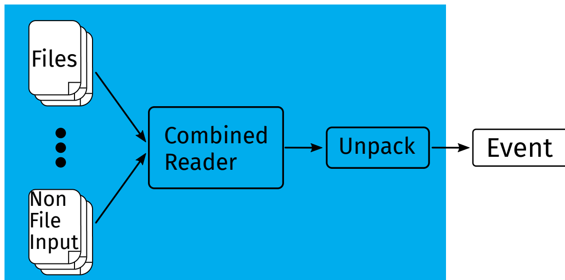
- “Unpacking” comprises several steps in the following even if they are ideally more or less free-standing, re-entrant functions that are just applied one after the other
  - Decompressing raw input data blobs / buffers
  - Schema evolution either on decompressed raw buffers or on final data types
  - Building and preparing collections from the decompressed buffers
- Ownership is always with the component where the data (in any form) currently resides
  - “Dataflow” programming (sort of)
  - If a framework wants to re-use Event slots, this can be achieved, e.g. by providing a dedicated Unpacker that can fill pre-allocated Events instead of allocating / creating new ones
- Ideally once the reader has read all necessary data for a given event, the rest of the chain can be run on several events simultaneously (and independently)
- “Non File Input” refers, e.g. to pre-mixing library / catalogue

Input System

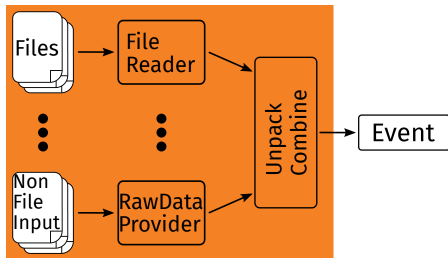
Different approaches possible





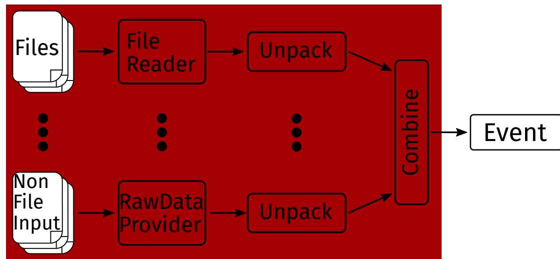


Variant 1



- Readers provide arbitrary data format and unpacker that knows how to unpack its data
- Different options for combining events
  - 1. unpack, 2. combine (almost) final collections
  - 1. combine data blobs, 2. unpack (on demand?)
  - Until a collection is unpacked it has to know how it can be unpacked!
- Can be very flexible approach that can handle new readers and arbitrary numbers of input sources
- Can get away with one synchronization point when combining the different sources into one self contained "Event"
- Threading all the necessary information through the combination step is probably the hardest part

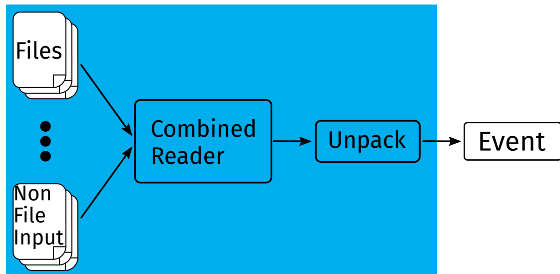
Variant 2



- Readers provide arbitrary data format and unpacker that immediately unpacks this data
- Combination is done on (almost) final collections
- Similar to Variant 1 with unpacking before combination

- Can be flexible approach that can handle new readers and arbitrary numbers of input sources
- Parallelization before combination could be achieved by having each reader + unpacker on its own thread
- Only combination would need explicit synchronization
- No lazy unpacking possible (major conceptual difference to Variant 1)

Variant 3



- Reader reads from several sources and provides a fully combined arbitrary data blob and a way to unpack it
- Unpacking can be done immediately or lazily on demand
- Easiest approach from combination point of view

- Rather inflexible. Potentially each combination of input sources would need a dedicated reader
- Readers become very large as they have to potentially hold a lot of state
- Lazy unpacking would be rather easy to implement on top of this, since the reader can normalize the different input source raw data formats

Input System

Orchestration

- Not yet addressed orchestration of all the components
- Orchestration really should just be orchestration and not having to deal with anything else rather than controlling that the data flow from / to the different components is as expected (important for framework use cases)
- Preference to have the components work standalone independently of how they are orchestrated (as long as that is done correctly)
  - E.g. Simply chaining all the things in a single threaded application should work with the same components as having them consume from / produce to different queues (or any other means of flow control)
  - Incidentally also allows for easier testing and benchmarking
- User access to input system will be provided by this orchestration layer
  - EventQueue / EventStore in standalone mode
  - Framework probably already has a dedicated way

Output System

Output System

Requirements

- Write to several different output files
  - Different contents (subset of all available content)
  - Different formats (different (I/)O backends)
- Transform from structured format to POD-buffers ("packing"), potentially compressing them before writing
- Allow for asynchronous operation wrt the rest of the event processing
- Also responsible for eventually "cleaning-up" the Event that is passed in
  - Free all resources
  - Hand back to framework after producing a valid "empty" slot state in case of slot re-usage
- Intermediate writing?
  - Equivalent results can be achieved with filtering and writing at the end

Output System
Requirements (ctd.)

- No access to the same file from multiple threads, but potentially have multipe threads handle multiple files
- Potentially has to handle different Event classes
  - Might have the same interface (`IEvent`) - should facilitate things
  - Differences at writing end probably smaller than at reading end

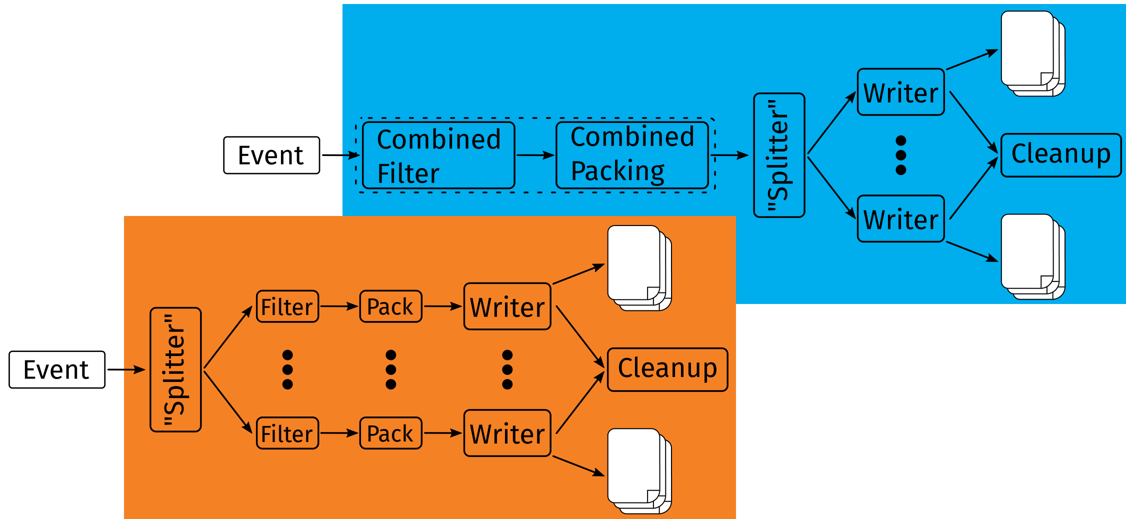
Output System

General Remarks

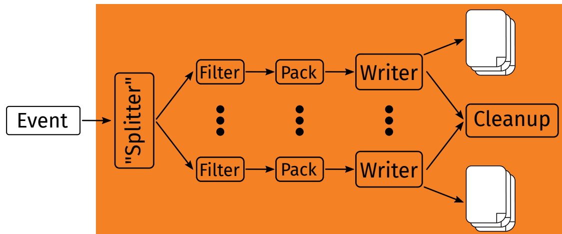
- “Packing” refers to collecting all the collection data into POD buffers and potentially additionally compressing them
  - Ideally free-standing, re-entrant functions, that leave their input untouched
  - Produce newly allocated buffers, for which ownership is handed over to the writer after returning (?)
- The writer only frees the resources of the “packed” data it writes
- The cleanup of the Event is left to a dedicated component
  - By default completely frees all resources
  - In framework usage can also just clear internal buffers and return an “empty” Event back to the arena
- Filter components can be held very general, doesn’t alter the Event at all and basically just returns a list of booleans, one per collection to indicate whether to write this collection or not
  - Maybe invoke on collections individually and decide on collection-by-collection basis instead of “event-level” (?)

Output System

Different approaches possible



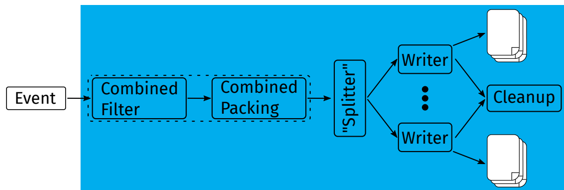
Variant 1



- Each “writer chain” operates on the complete Event
- Filtering to only include the desired collections before packing (and compressing) them before writing them to file

- Event is cleaned up once all chains have finished
- “Splitter” essentially an intermediate orchestration component that launches the writer chains (can in principle act as the “Input Queue” to the Output system)
  - Individual threads, async tasks, working through the list in one thread, …
  - Invokes cleanup at the end?
- Easy to extend approach, but some work can be duplicated
  - E.g. some collections will be packed twice, if they are written to file by two chains

Variant 2



- In a first step only the desired collections are packed
  - Possible that one collection is packed in several different ways if written into different formats
- Packed collections are passed to “Splitter” which distributes them to different writers and also controls cleanup of the whole Event in the end
  - Needs to distribute the correct data to the corresponding writer
- This approach can avoid packing the same collection multiple times
- Writers need to register their filter and packing function to the combined filter / packer
- Slightly less opportunity for running on multiple threads, but overall possibly less work

Output System

Orchestration

- The previous variants only partially address the orchestration of the components
- Components should work independently of the orchestration
- Orchestration just ties them together in the correct order and ensures the proper data flow
- User access through the orchestration layer
  - Possible for all the configuration, e.g. filtering?

A few final remarks

My preferences and final thoughts

- This overview / first shot probably misses a lot of the details that make the implementation hard in the end
  - Should help to define the goals of the design (also non-goals)
- I think that for both **Input** and **Output** System **Variant 1** is currently the most promising
  - Most flexible for Input System
  - Probably easiest to implement for Output System (even if work is potentially duplicated)
- I personally prefer composition at compile time over composition at runtime
  - Allows for checking compatibility very early
  - Potentially more efficient
- It might be necessary to also define an abstract interface for the Input / Output system
- "Copying" a subset of all collections from one file to another without fully unpacking events, should in principle be possible with custom unpacker / packer pairs
  - Might need to define an explicit "intermediate raw data container" format