Future framework directions

An LDRD approach called Meld

Kyle J. Knoepfel

5 June 2023

Fermilab Frameworks Workshop

Setting the stage

- Experiments use computing frameworks (like art) to process detector and simulated data for physics analysis.

- Existing computing frameworks are based on collider-physics concepts.

  Their processing constructs assume localized accelerator bunch crossings

  Events are treated as statistically independent

- DUNE, Fermilab's flagship experiment, will analyze interaction regions and data-collection time windows that sometimes overlap or vary in size.

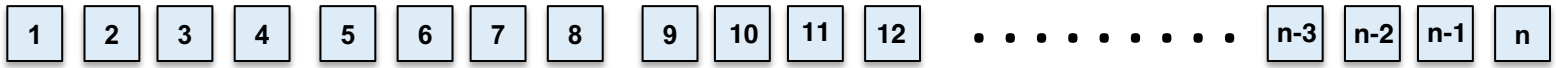
  No framework solution exists that readily supports this

  Neutrino physicists must work around current framework limitations

Processing complications (1)

- The collider physics approach to data processing

  Existing computing frameworks process data in rigid ways

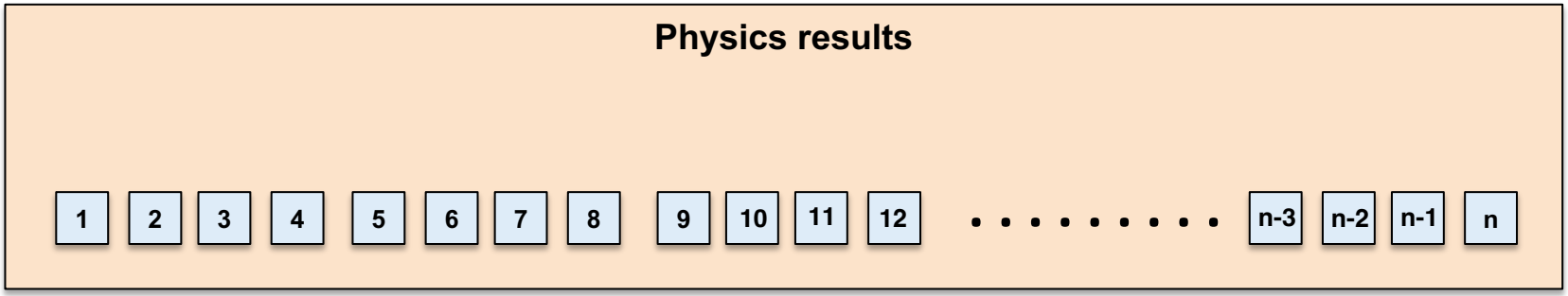


Fermilab

Processing complications (1)

- The collider physics approach to data processing

Existing computing frameworks process data in rigid ways

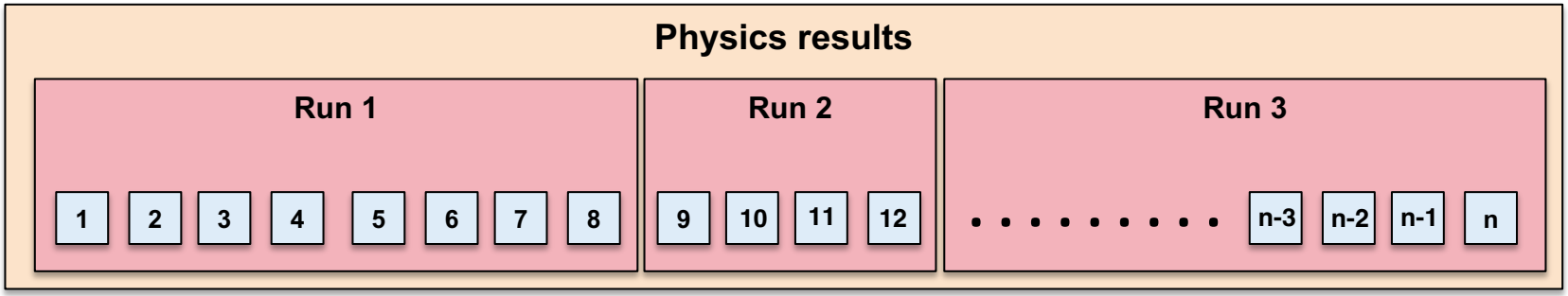


Physics results are obtained by analyzing the data as a whole

Processing complications (1)

- The collider physics approach to data processing

  Existing computing frameworks process data in rigid ways



  Physics results are obtained by analyzing the data as a whole

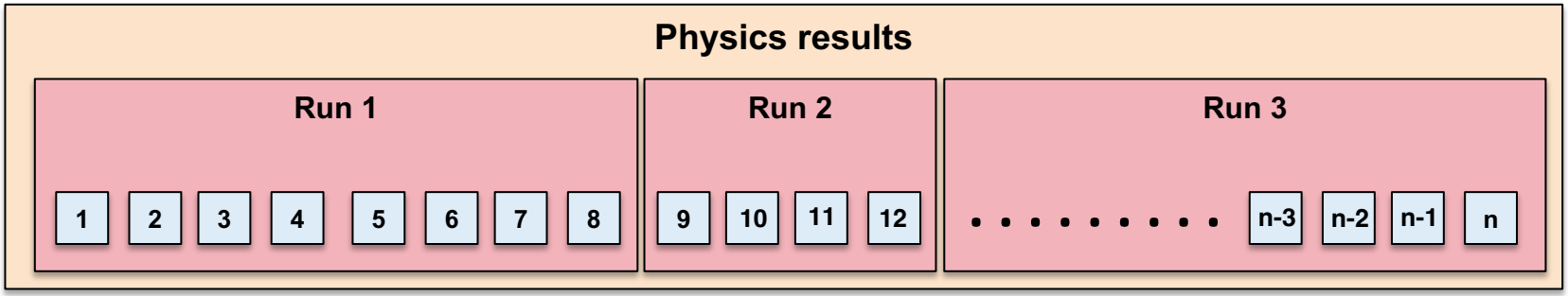
  Data is grouped into runs to reflect changes in (e.g.) beam or detector conditions

Fermilab

Processing complications (1)

- The collider physics approach to data processing

  Existing computing frameworks process data in rigid ways



  Physics results are obtained by analyzing the data as a whole

  Data is grouped into runs to reflect changes in (e.g.) beam or detector conditions

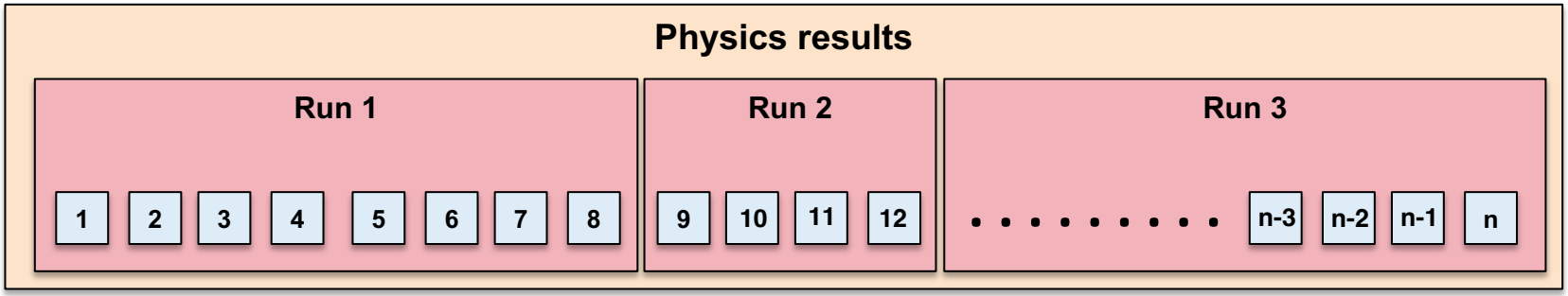
  Data-processing frameworks set these hierarchies “in stone”

Fermilab

Processing complications (1)

- The collider physics approach to data processing

  Existing computing frameworks process data in rigid ways



  Physics results are obtained by analyzing the data as a whole

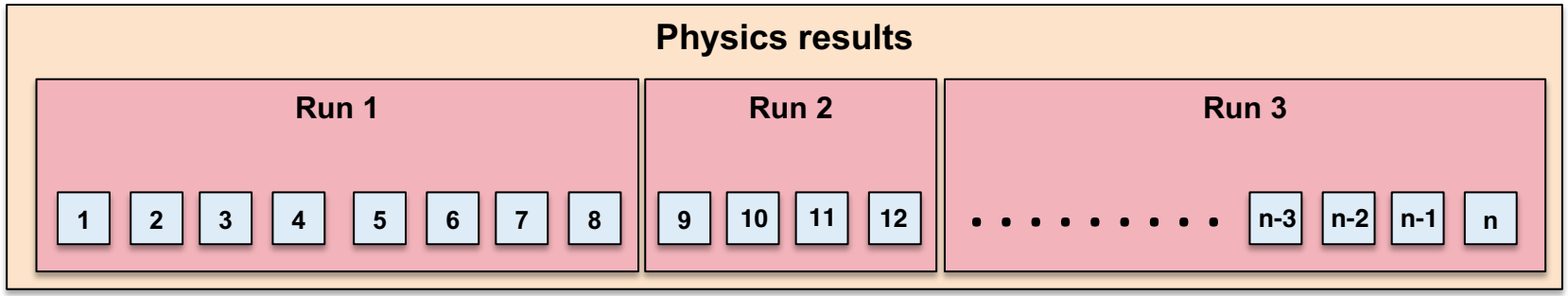
  Data is grouped into runs to reflect changes in (e.g.) beam or detector conditions

  Data-processing frameworks set these hierarchies “in stone”

- **This approach does not work well for DUNE.**

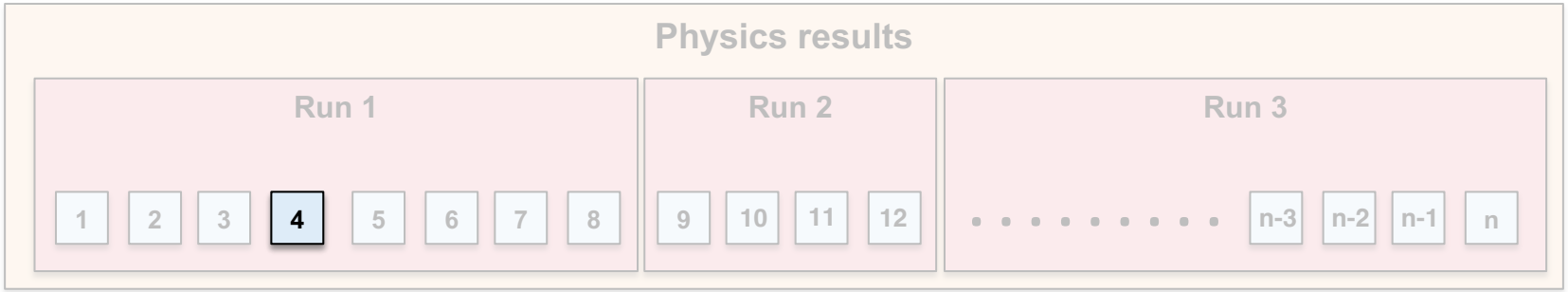
Fermilab

Processing complications (2)



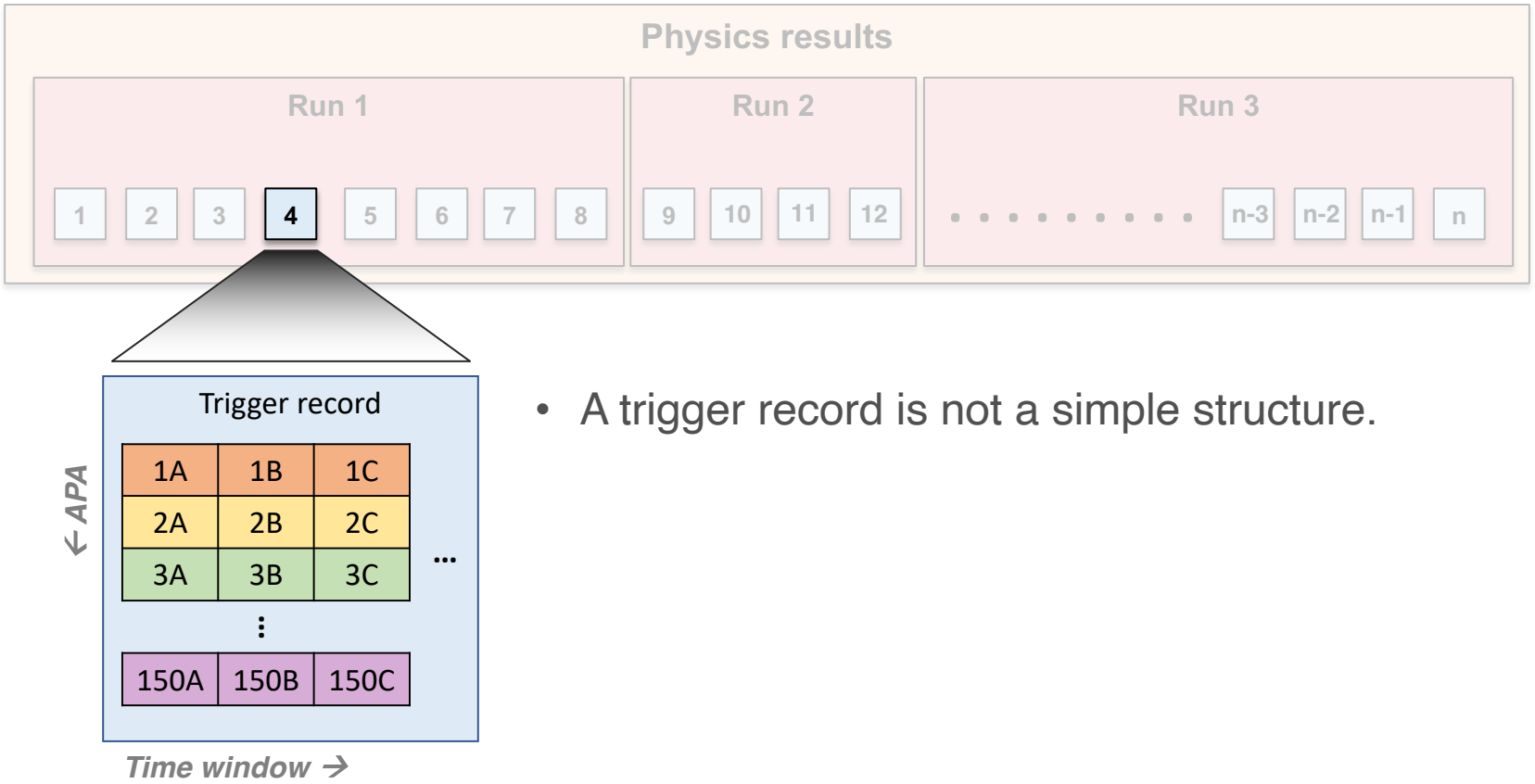
Fermilab

Processing complications (2)



Fermilab

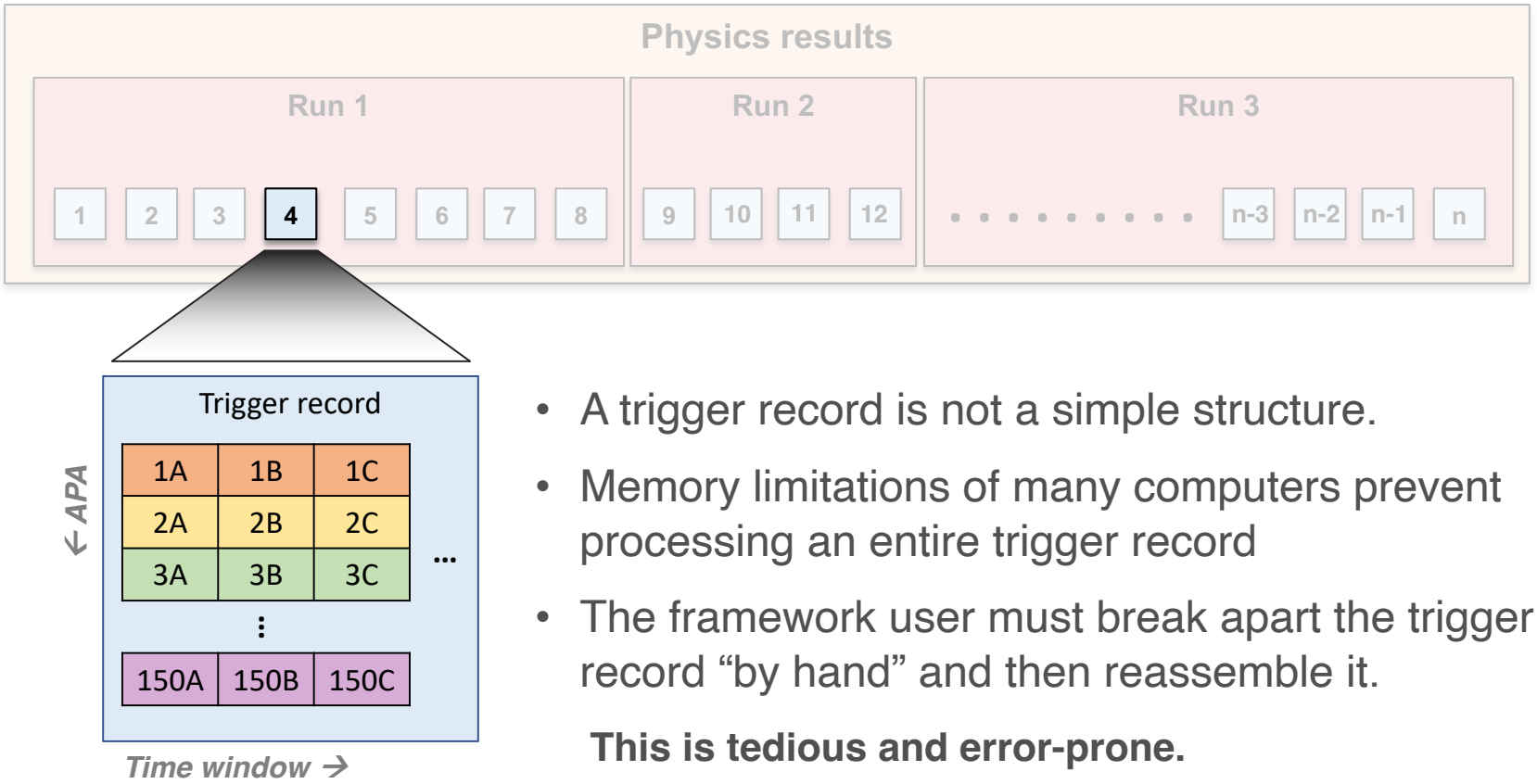
Processing complications (2)



- A trigger record is not a simple structure.

Fermilab

Processing complications (2)

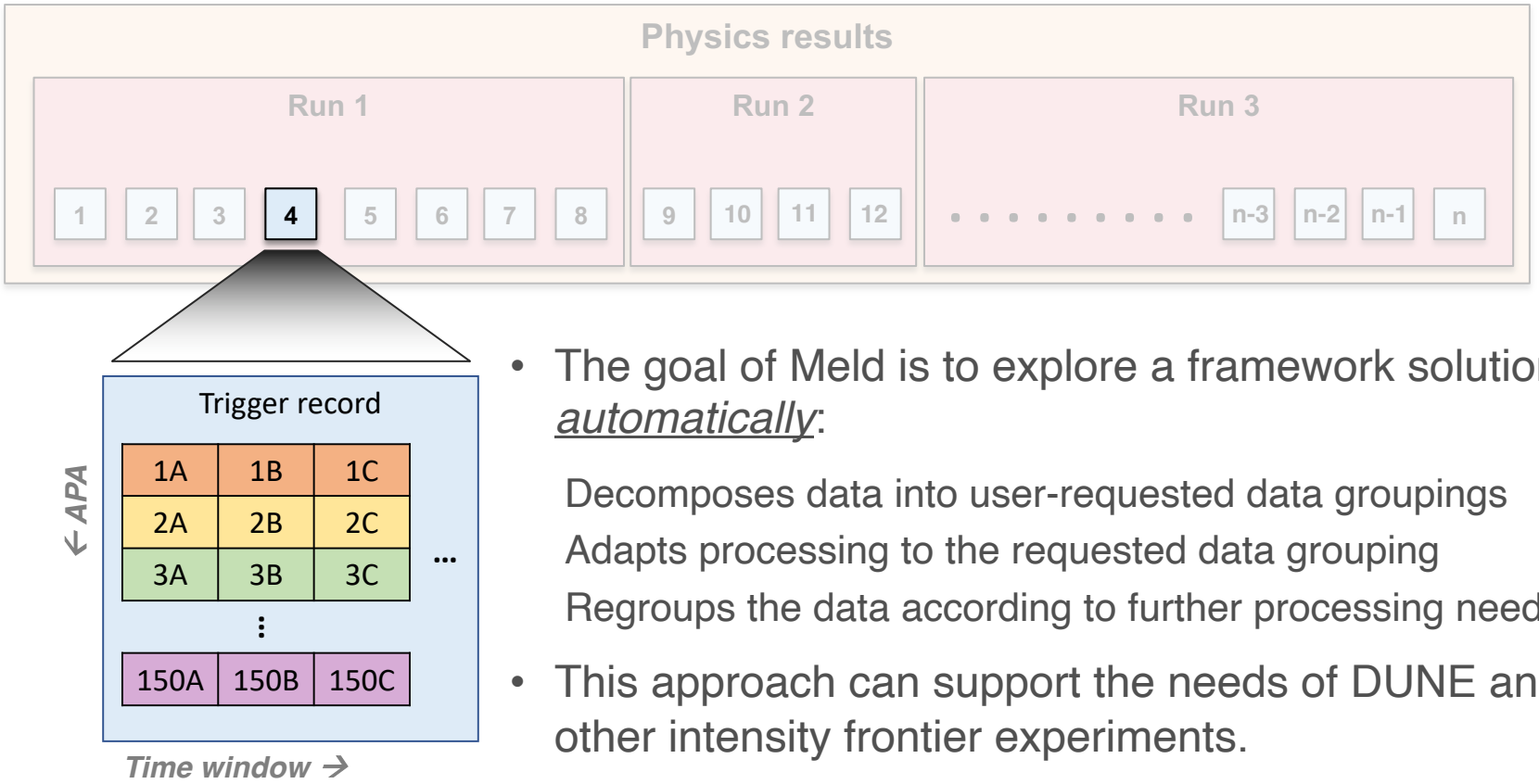


- A trigger record is not a simple structure.
- Memory limitations of many computers prevent processing an entire trigger record
- The framework user must break apart the trigger record “by hand” and then reassemble it.

This is tedious and error-prone.

Fermilab

Processing complications (2)



- The goal of Meld is to explore a framework solution that *automatically*:
  - Decomposes data into user-requested data groupings
  - Adapts processing to the requested data grouping
  - Regroups the data according to further processing needs.
- This approach can support the needs of DUNE and other intensity frontier experiments.

Fermilab

Meld

- A 2-year, laboratory-directed R&D project based at Fermilab.
- The goal is to explore options, not necessarily to provide software.

Meld

- A 2-year, laboratory-directed R&D project based at Fermilab.
- The goal is to explore options, not necessarily to provide software.

- Meld has been heavily influenced by:
  - Regular discussions with DUNE experts
  - Existing framework capabilities and limitations
  - Functional programming (e.g. Haskell)
  - Mathematics (set, graph, and category theory)

Meld

- A 2-year, laboratory-directed R&D project based at Fermilab.
- The goal is to explore options, not necessarily to provide software.

- Meld has been heavily influenced by:
  - Regular discussions with DUNE experts
  - Existing framework capabilities and limitations
  - Functional programming (e.g. Haskell)
  - Mathematics (set, graph, and category theory)

Prerequisites

Support user-provided algorithms written in C++20 or newer

Design for concurrency

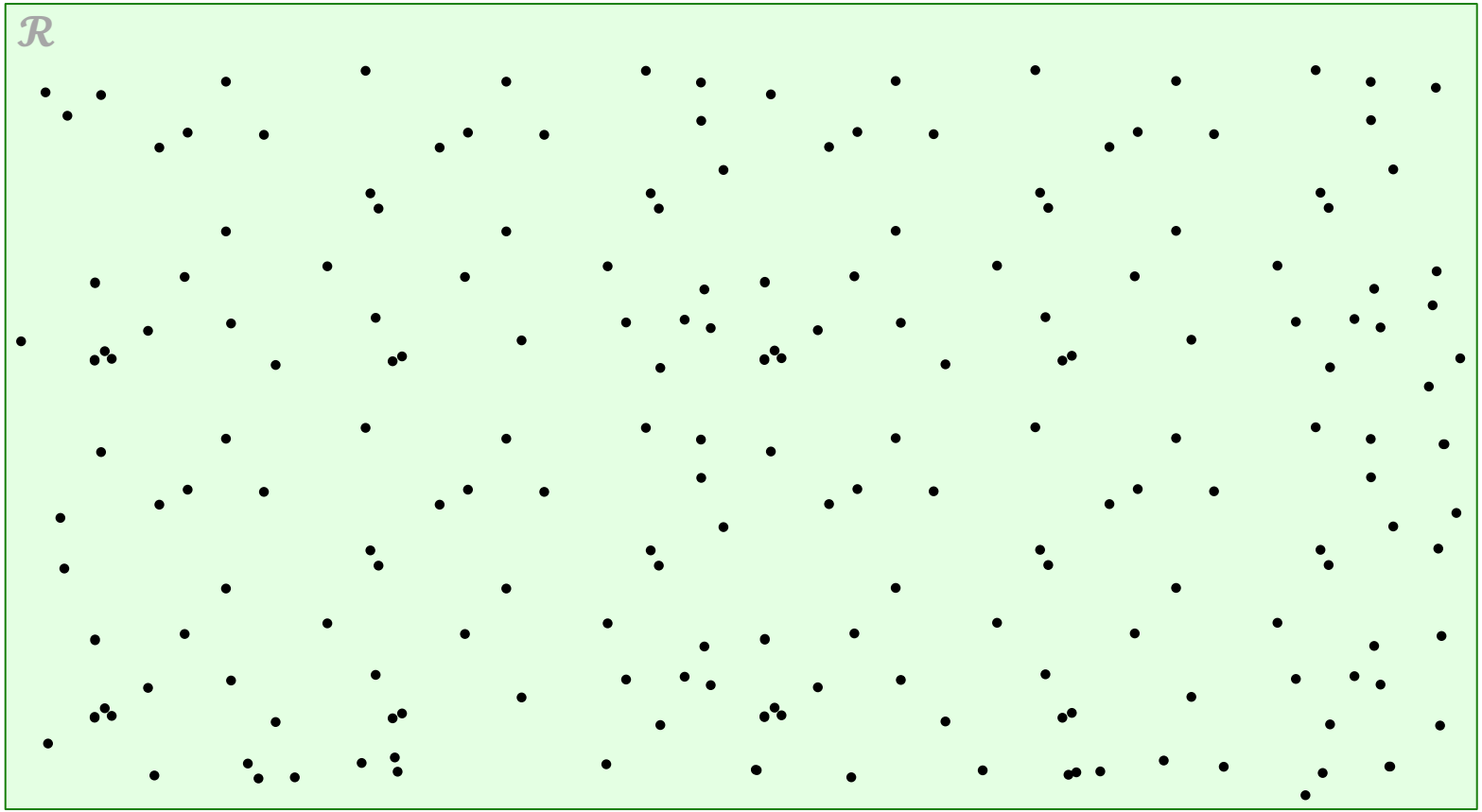
Favor community-provided software

Looking at the data

The following discussion describes a logical organization of data.

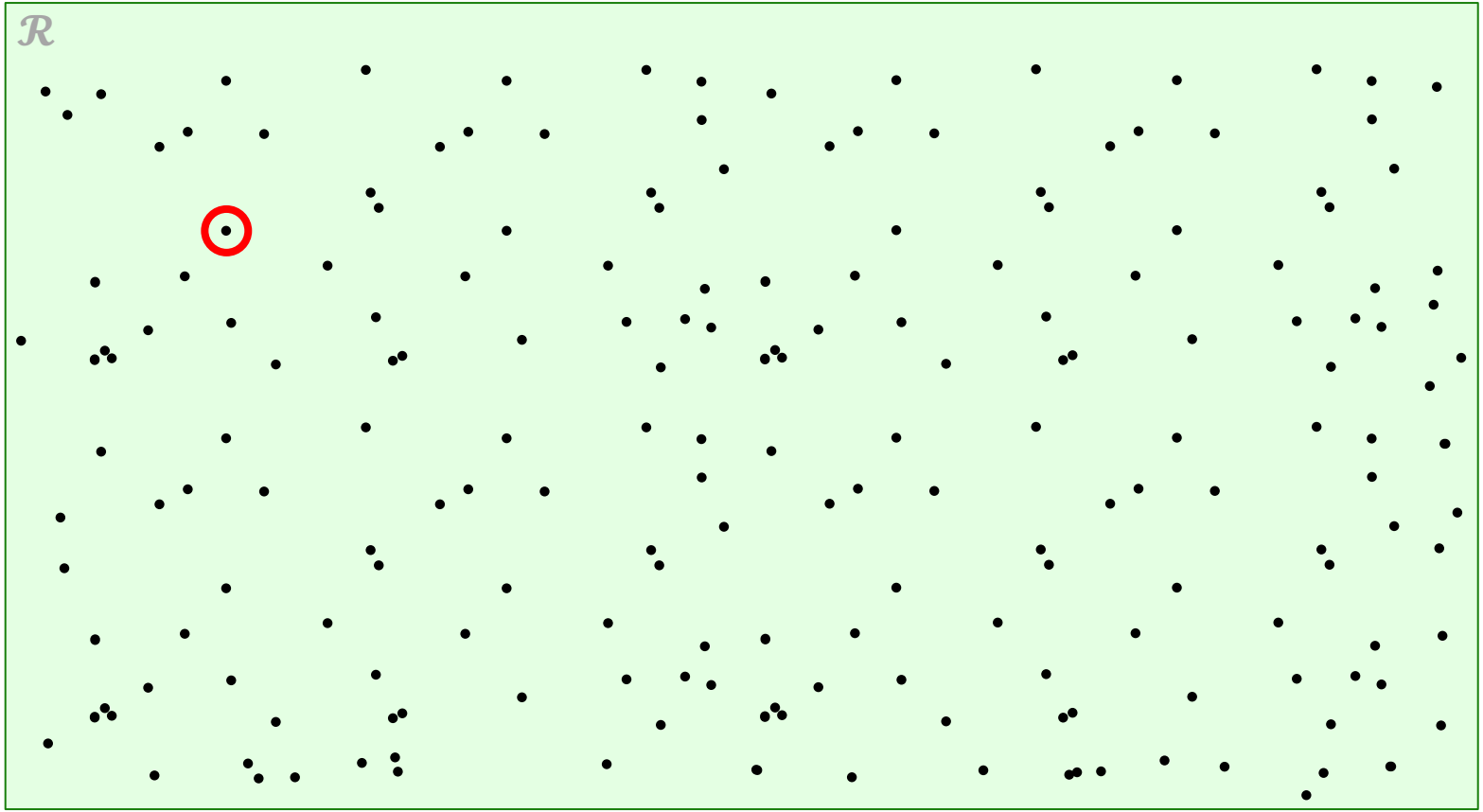
It does not imply a specific in-memory representation of data.

Looking at the data (set)



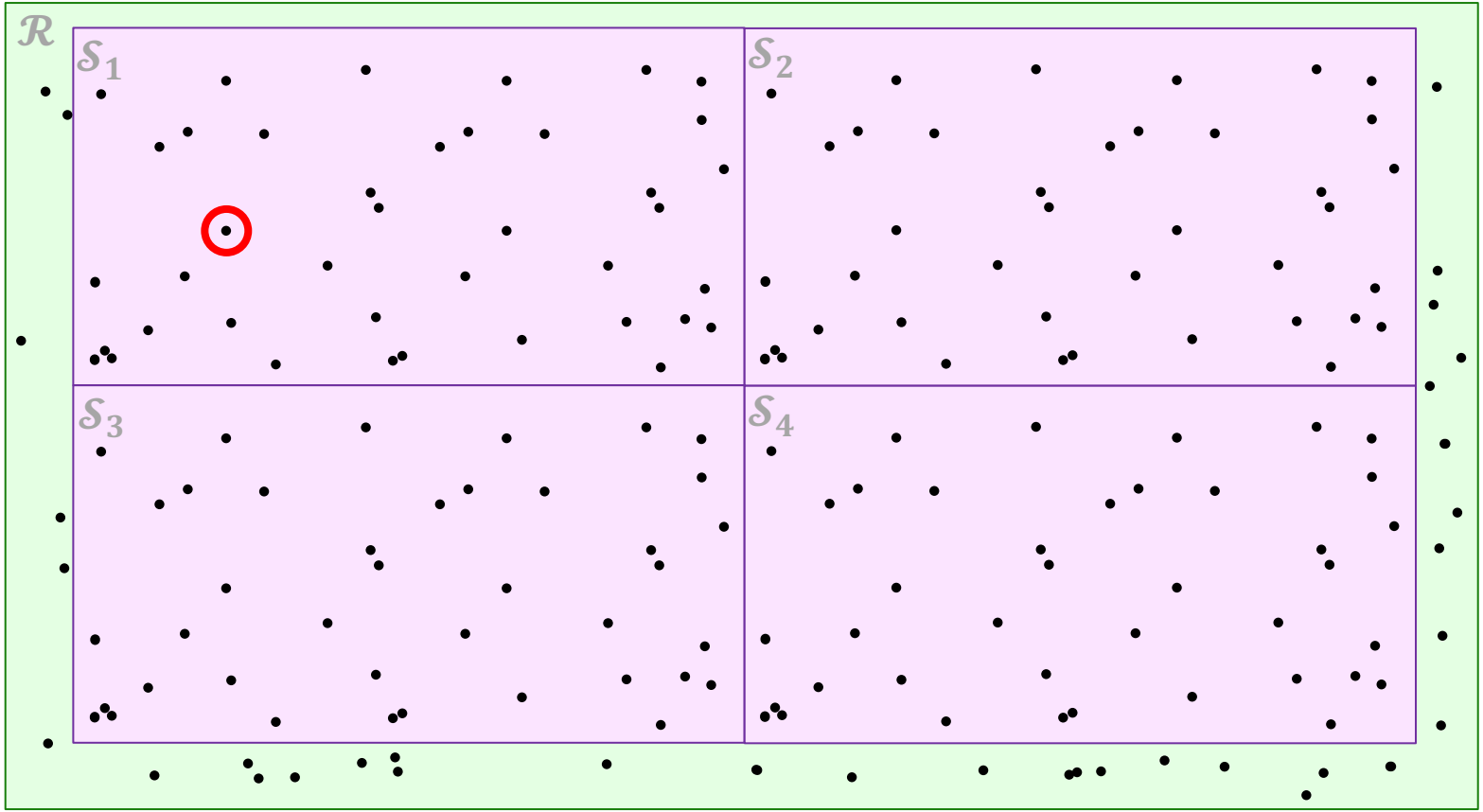
Fermilab

Looking at the data (products)



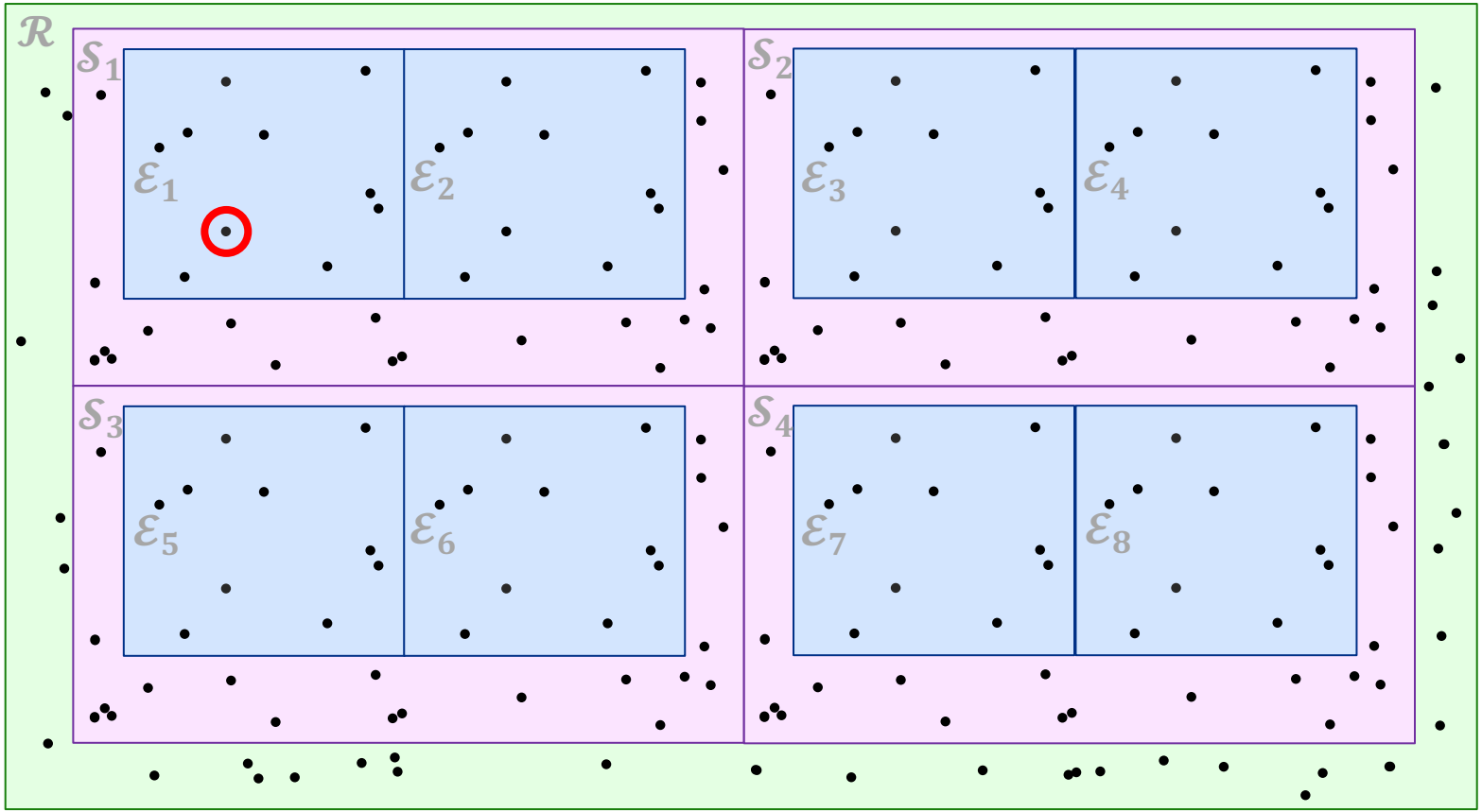
Fermilab

Looking at the data (products)



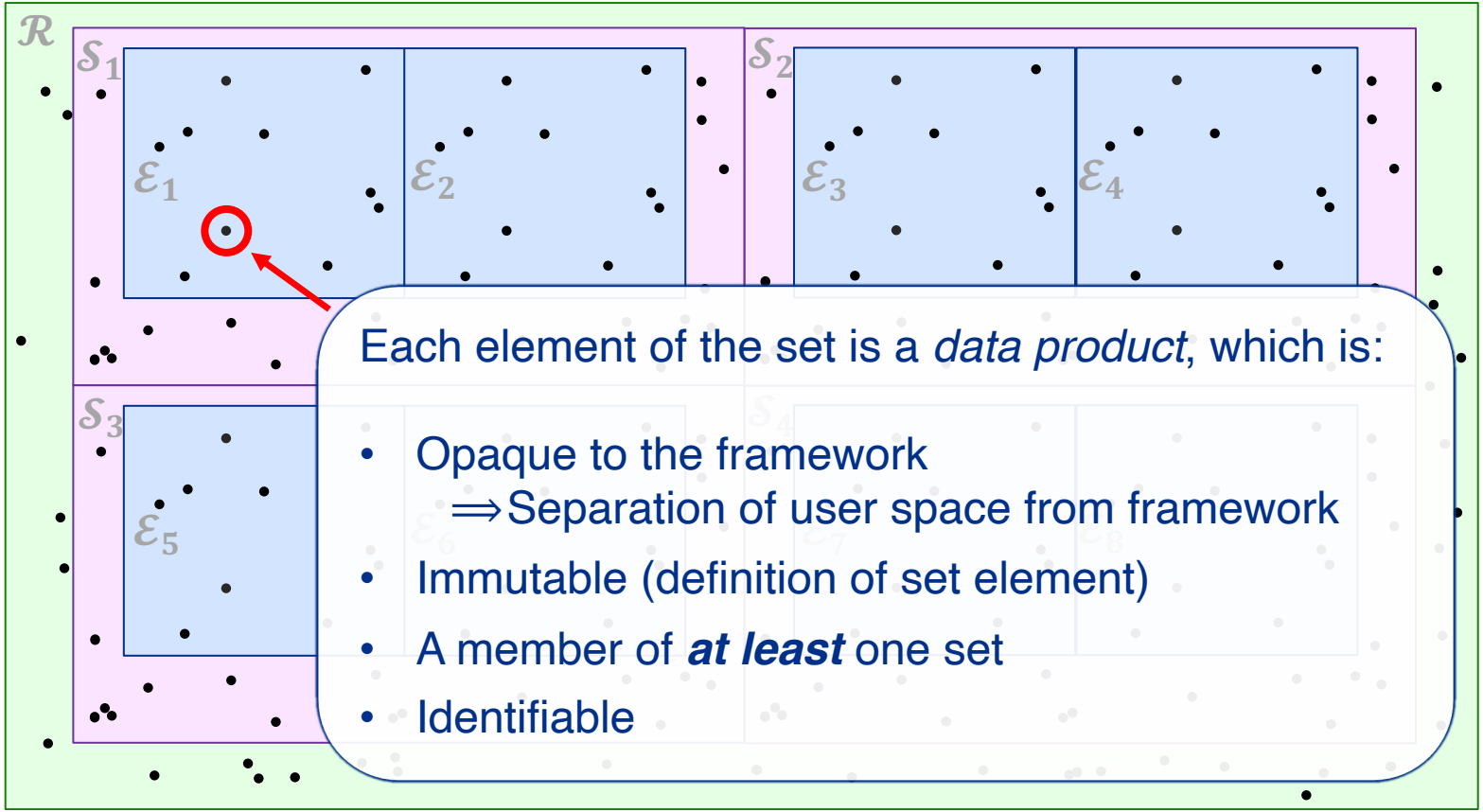
Fermilab

Looking at the data (products)



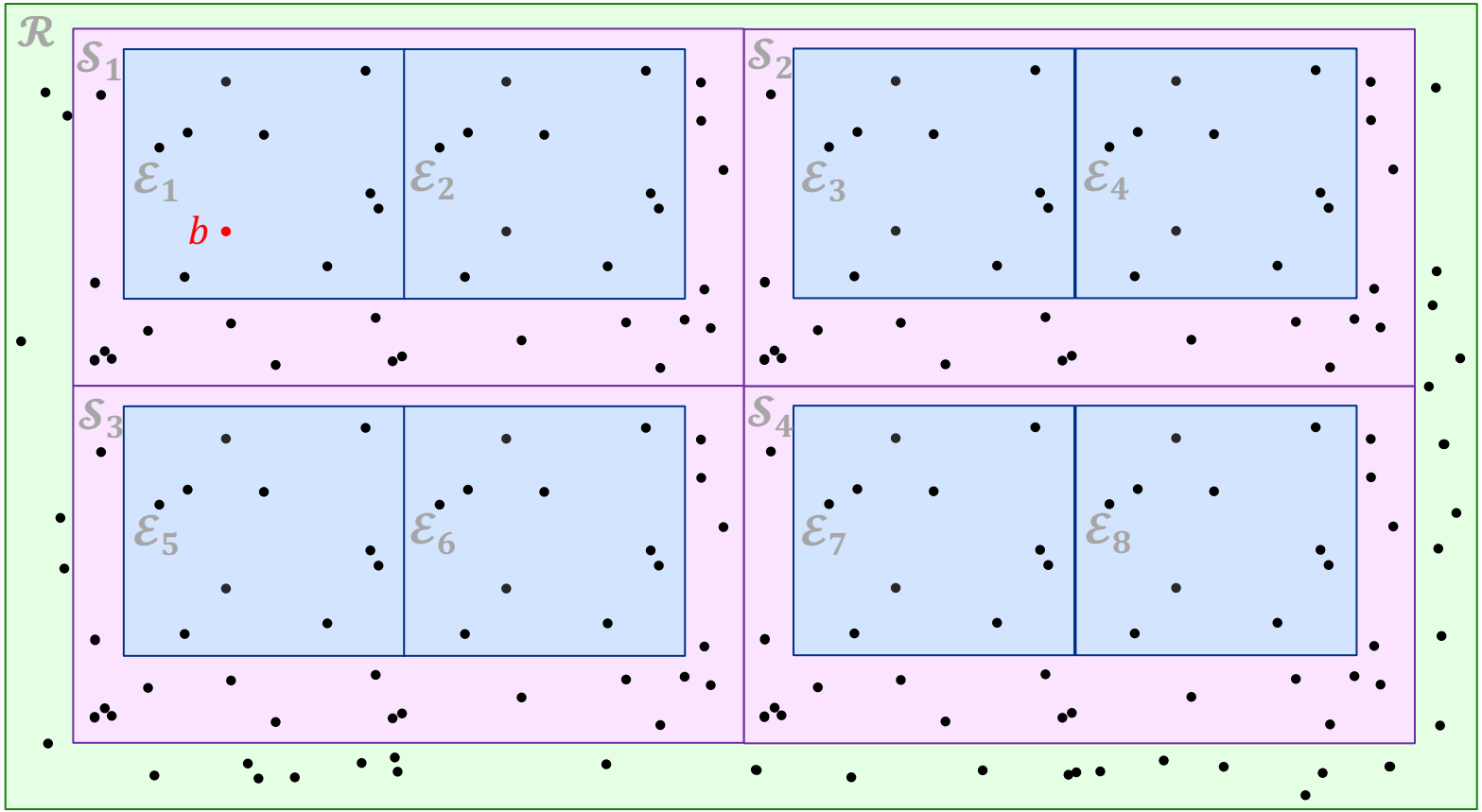
Fermilab

Looking at the data (products)



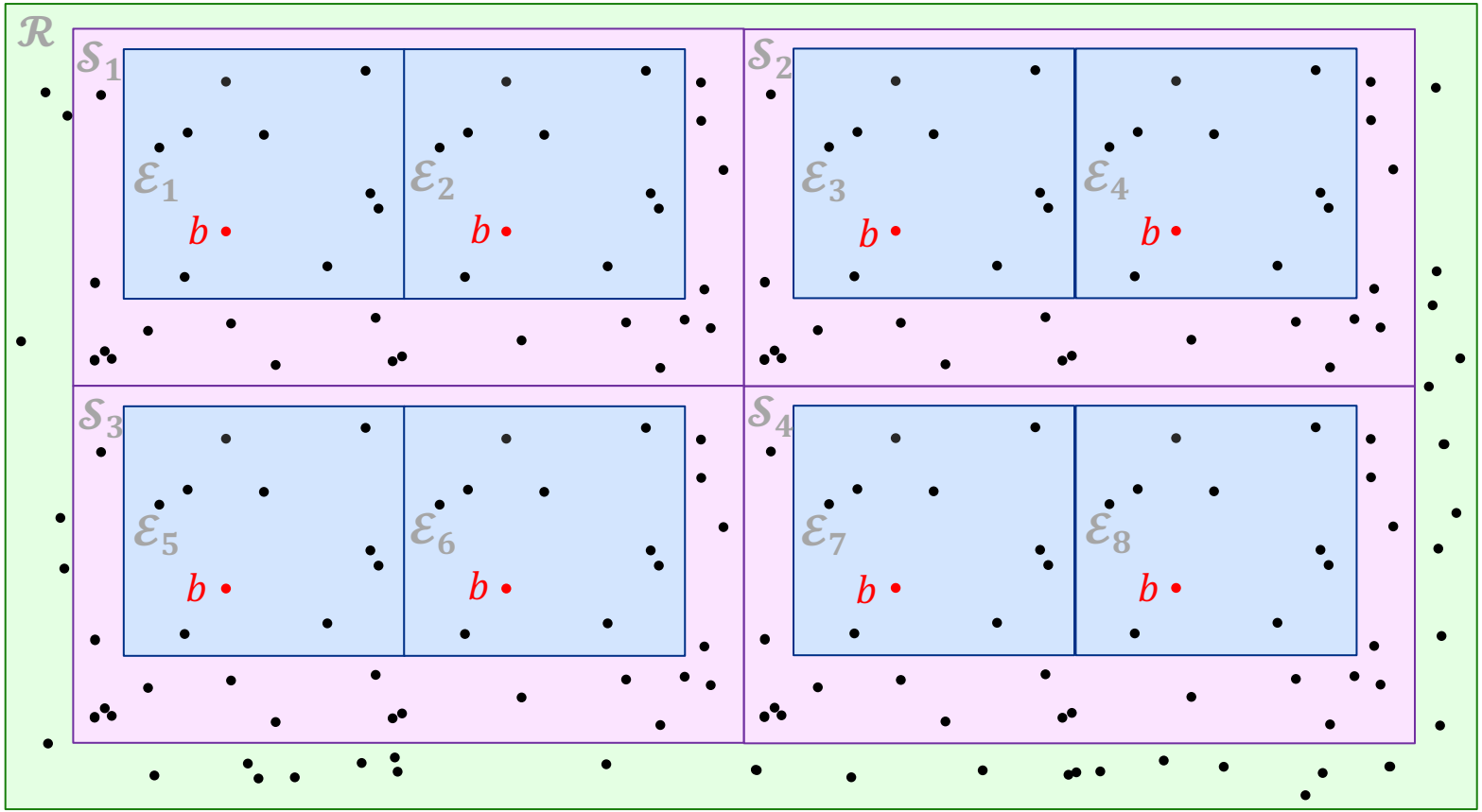
Fermilab

Looking at the data (products)



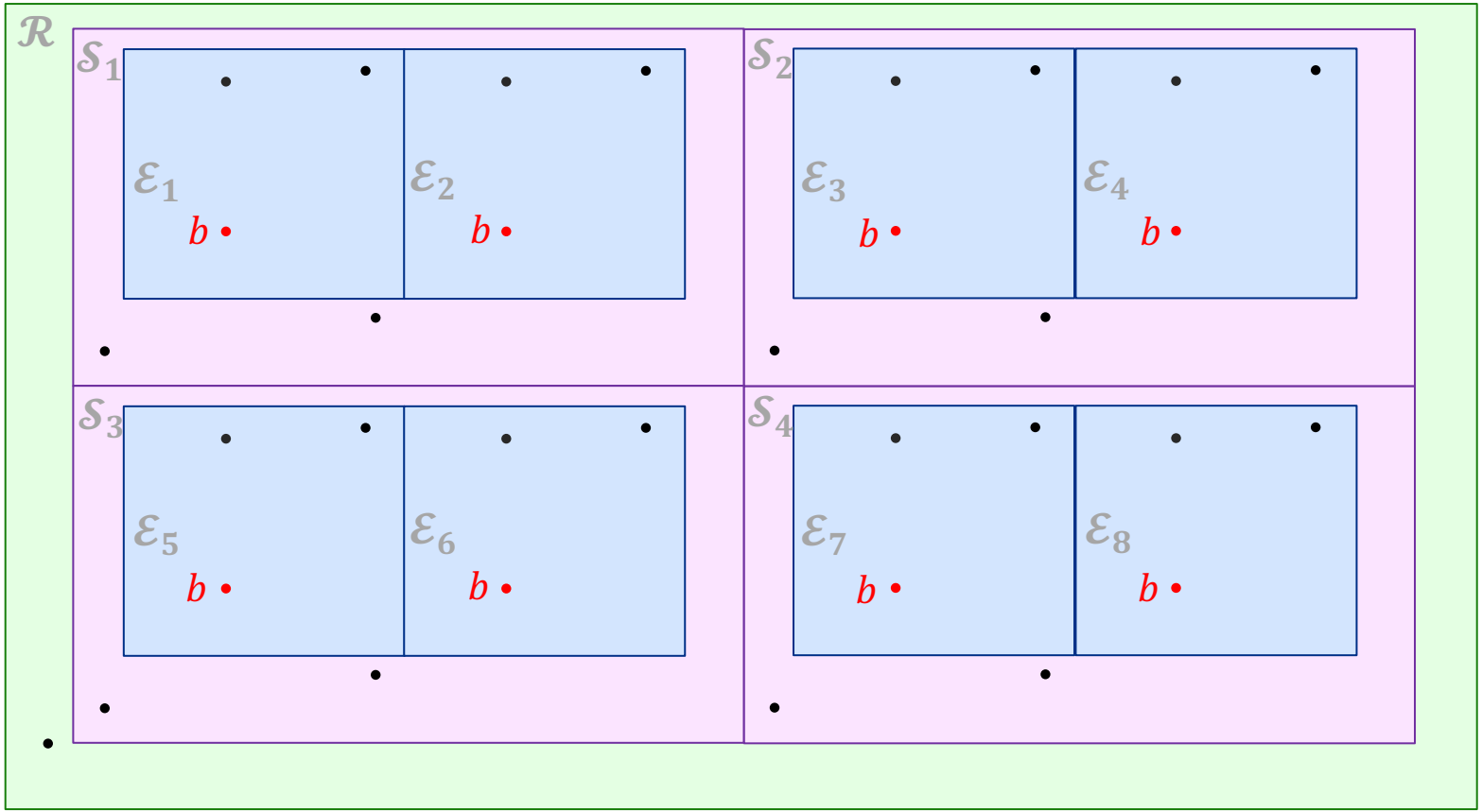
Fermilab

Looking at the data (products)



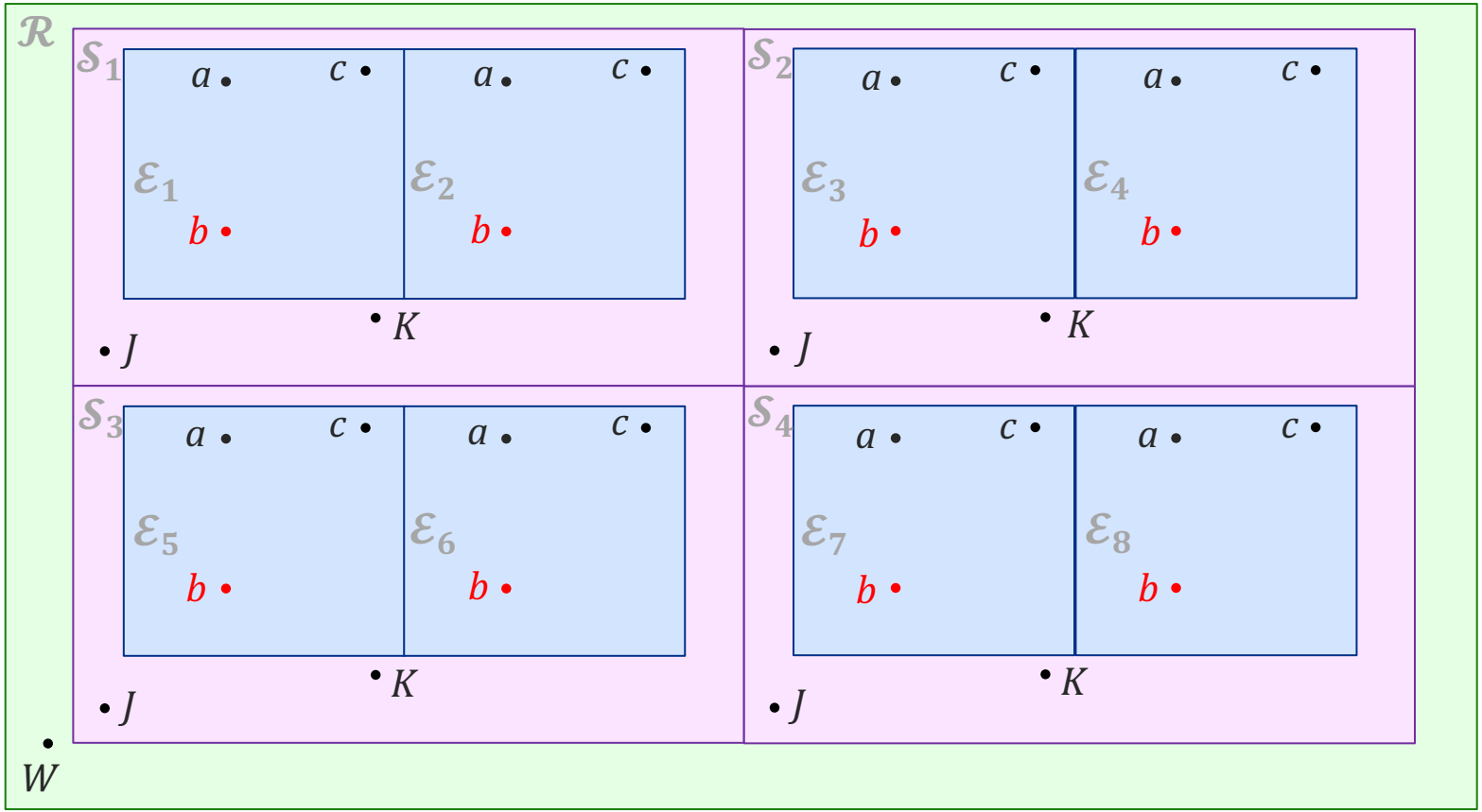
Fermilab

Looking at the data (products)

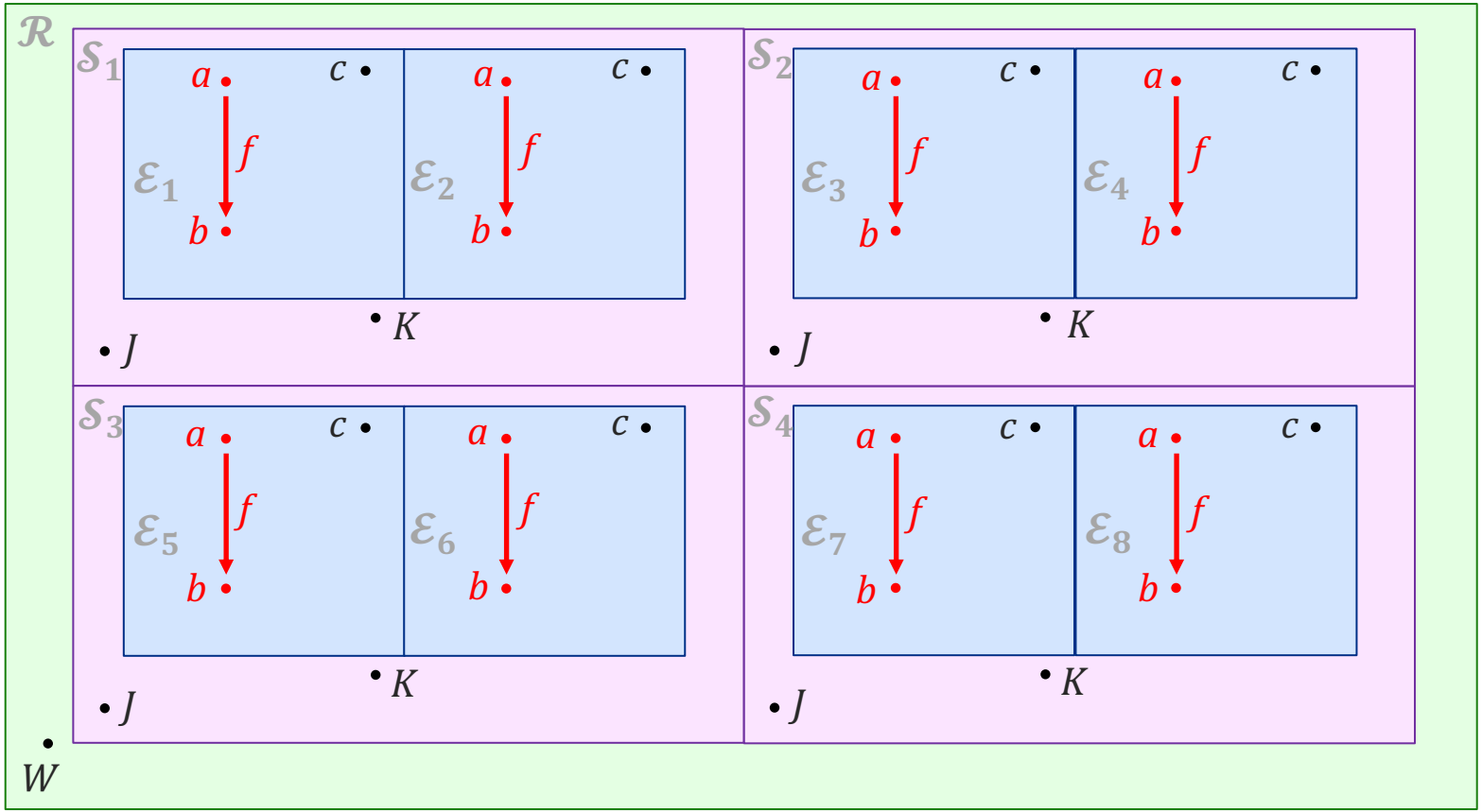


Fermilab

Looking at the data (products)

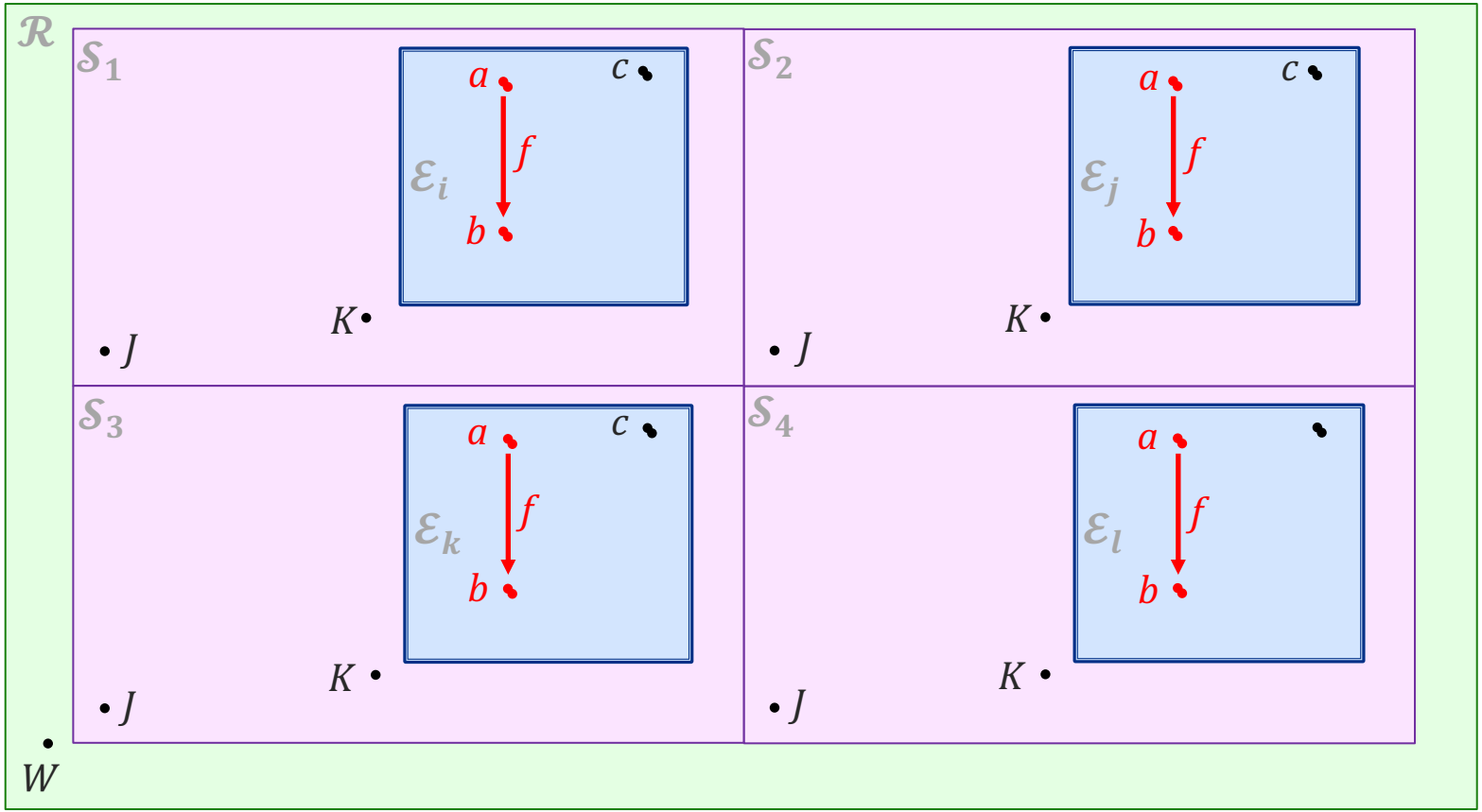


Looking at the data (product mappings)



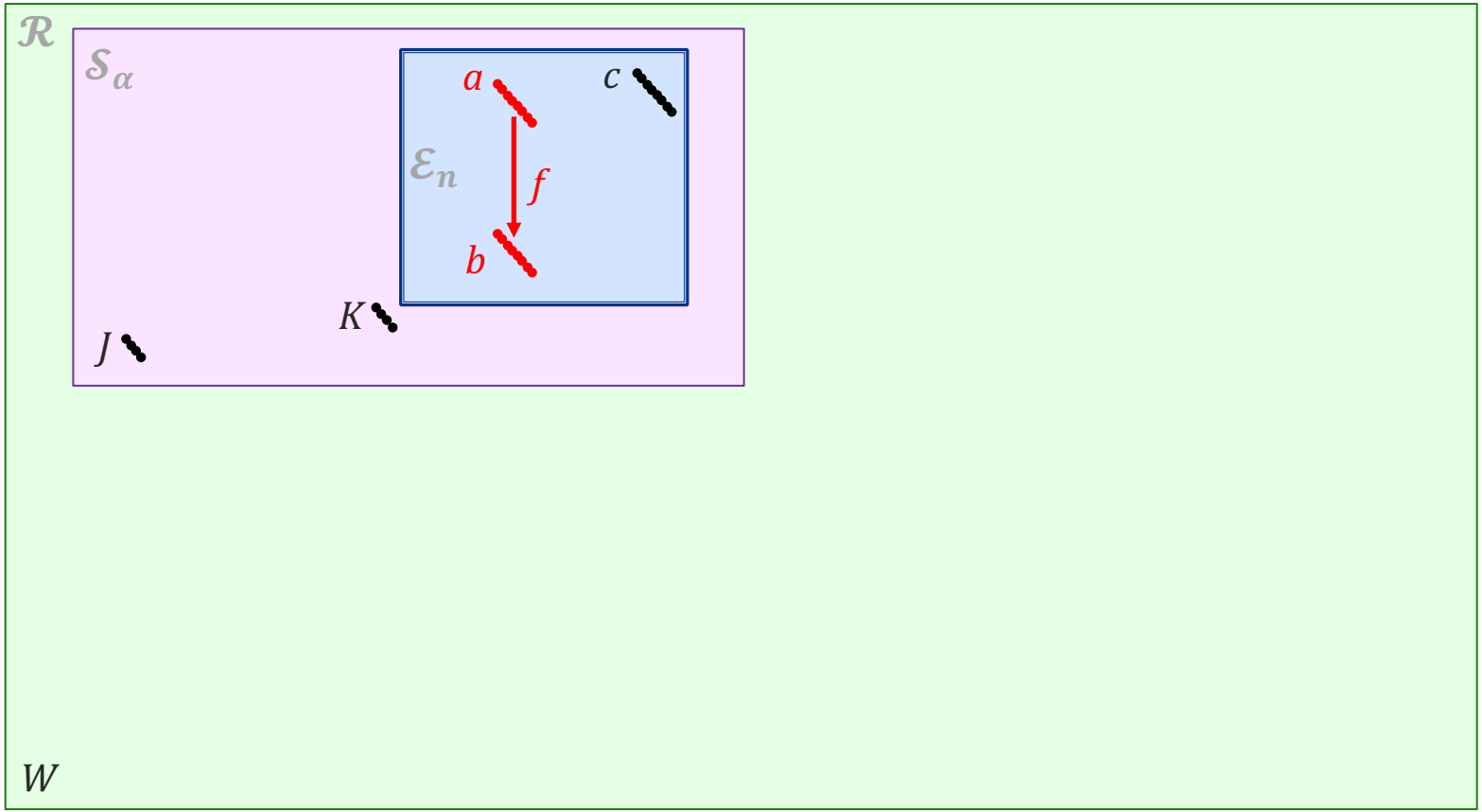
Fermilab

Looking at the data (product sequences)

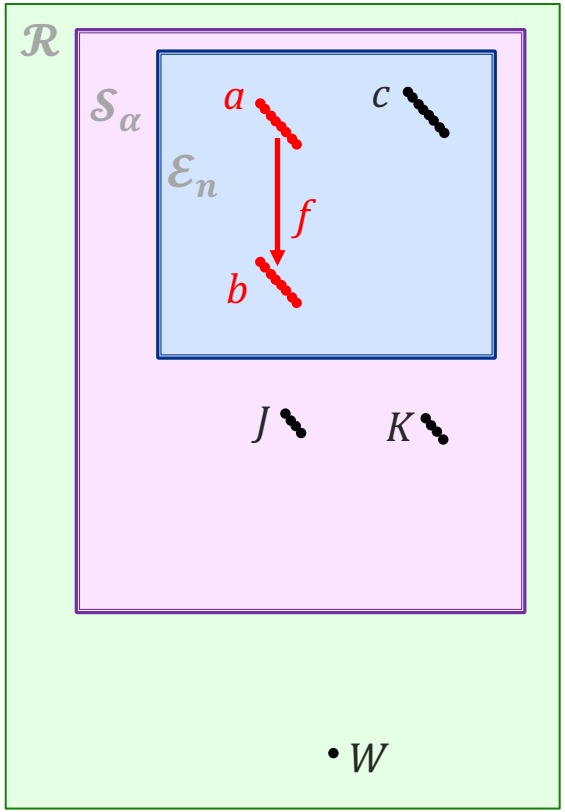


Fermilab

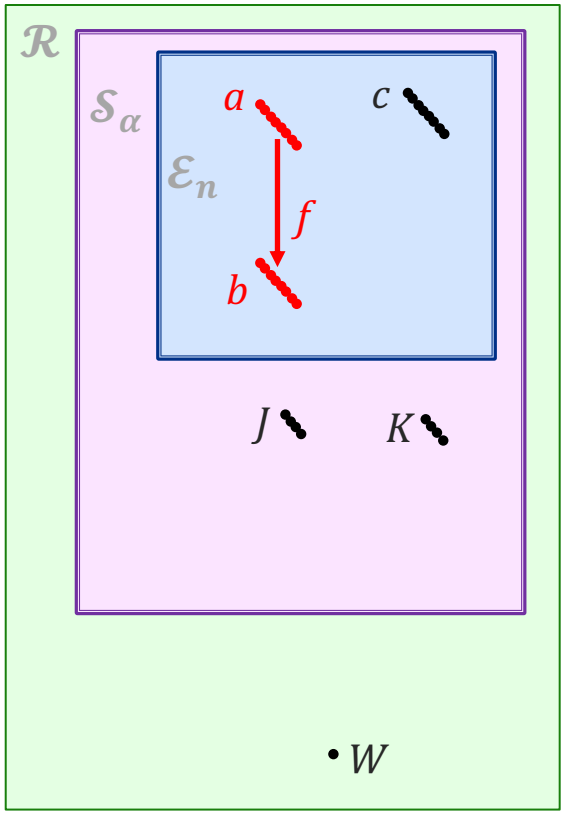
Looking at the data (product sequences)



Looking at the data (product sequences)



Looking at the data (product sequences)



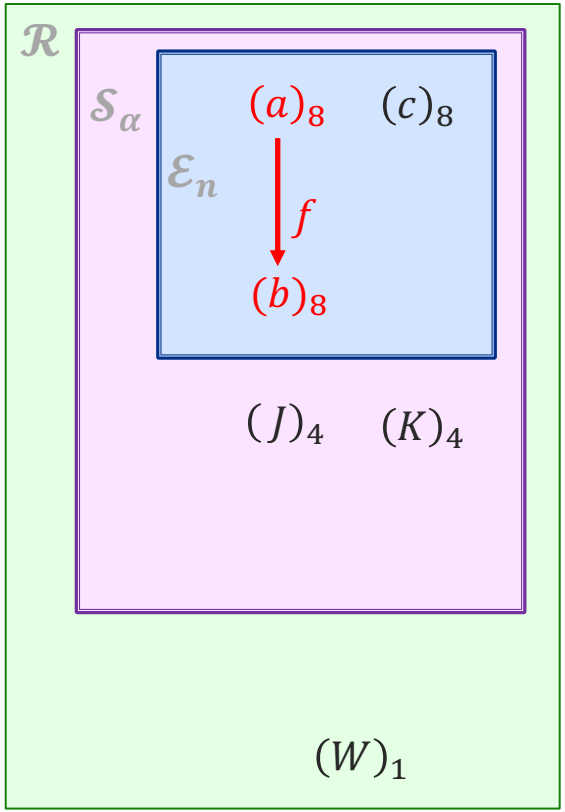
We can make the following replacement (e.g.):

$$c \quad = (c)_8$$

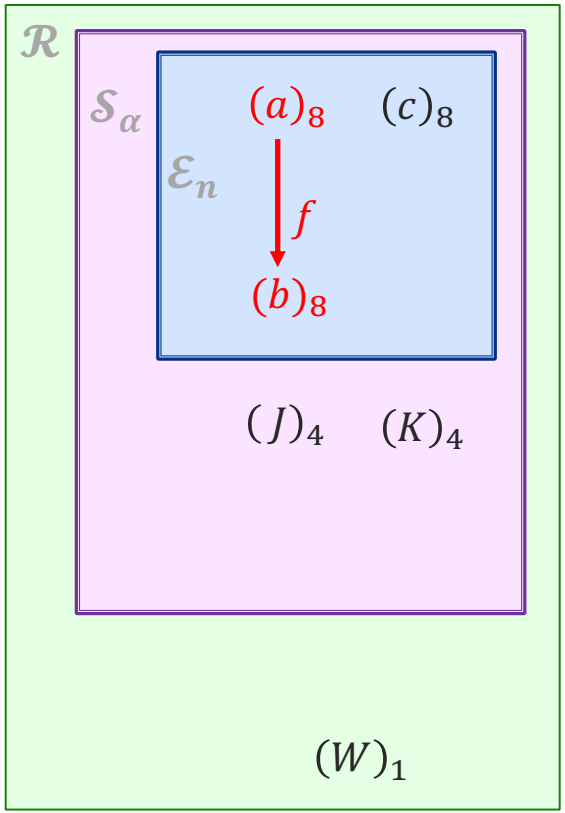
depicting the data products labeled c from 8 events as a sequence.

Fermilab

Looking at the data (product sequences)

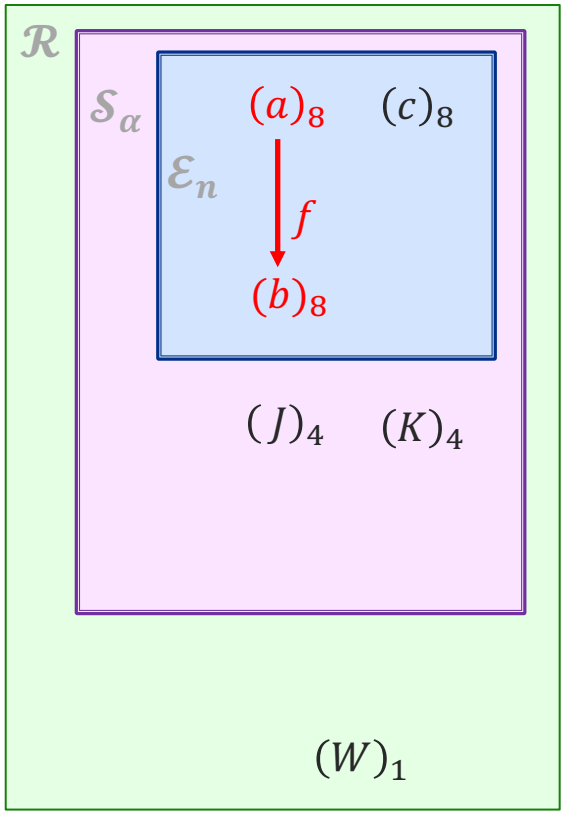


What type of things are we dealing with?



- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

What type of things are we dealing with?

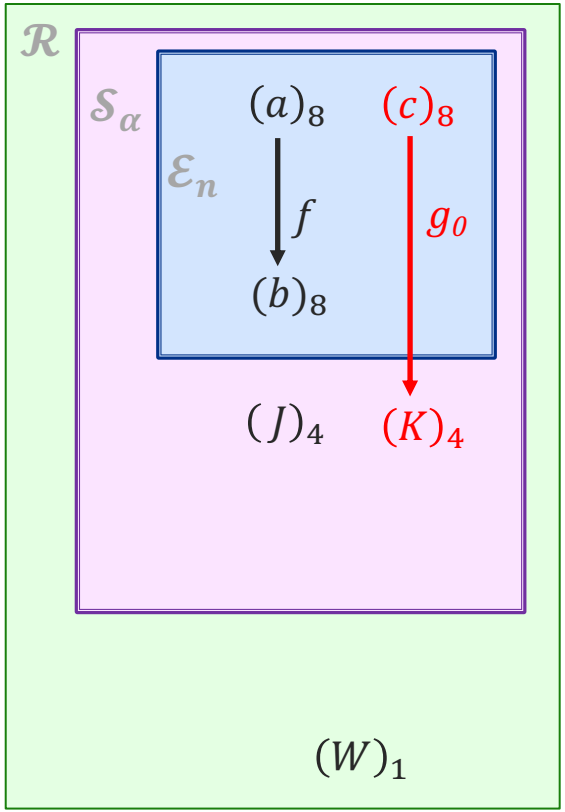


- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

  This is a map or transform.

Fermilab

What type of things are we dealing with?



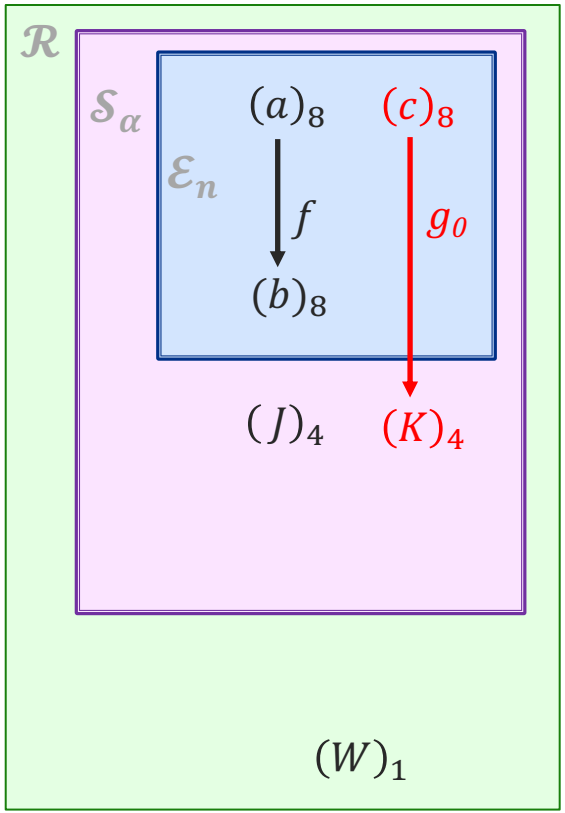
- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

  This is a map or transform.

- An operation that converts a sequence of elements $(c)_8$ to a shorter sequence of elements $(K)_4$ at a higher level of nesting, using a function g_0:

Fermilab

What type of things are we dealing with?



- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

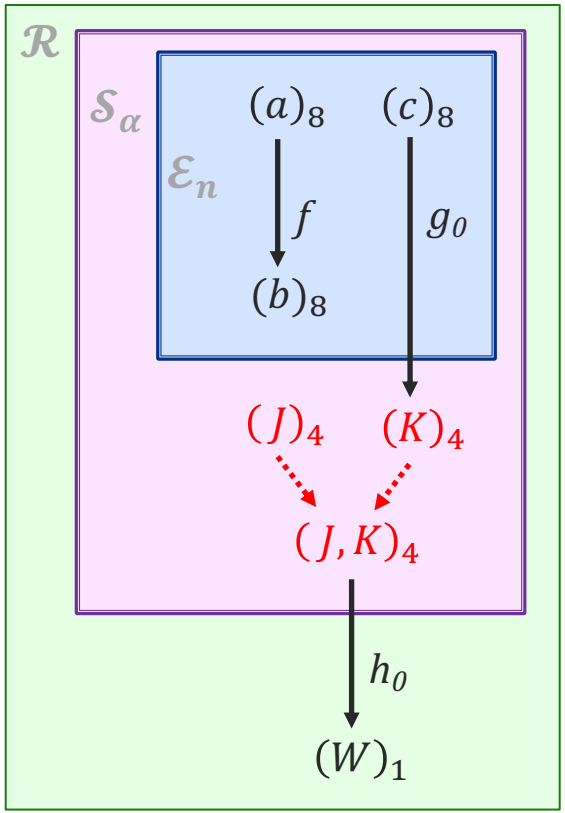
  This is a map or transform.

- An operation that converts a sequence of elements $(c)_8$ to a shorter sequence of elements $(K)_4$ at a higher level of nesting, using a function g_0:

  This is a fold or reduction.

Fermilab

What type of things are we dealing with?



- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

  This is a map or transform.

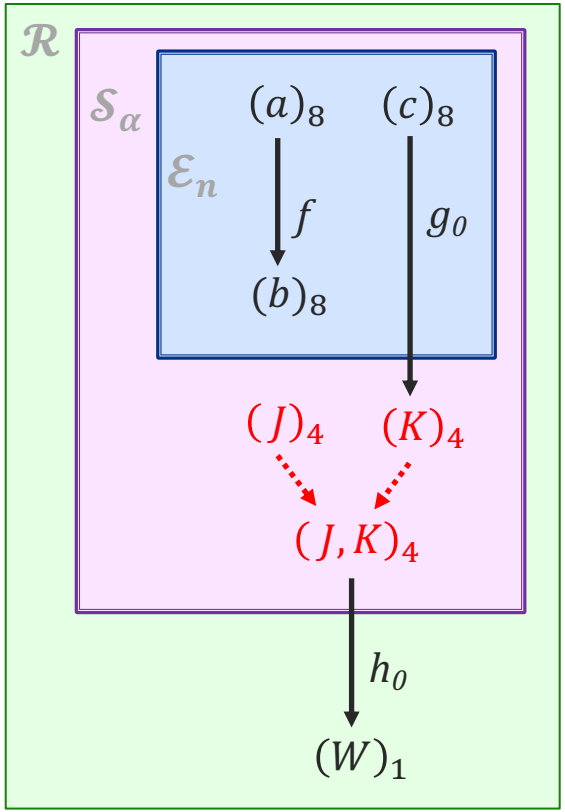
- An operation that converts a sequence of elements $(c)_8$ to a shorter sequence of elements $(K)_4$ at a higher level of nesting, using a function g_0:

  This is a fold or reduction.

- An operation that pairs element of two sequences $(J)_4$ and $(K)_4$ into one sequence $(J,K)_4$:

Fermilab

What type of things are we dealing with?



- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

  This is a map or transform.

- An operation that converts a sequence of elements $(c)_8$ to a shorter sequence of elements $(K)_4$ at a higher level of nesting, using a function g_0:

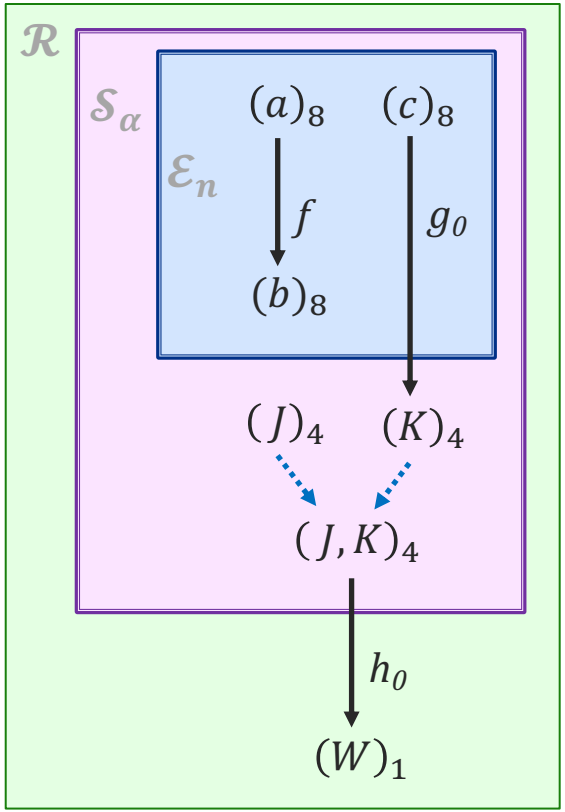
  This is a fold or reduction.

- An operation that pairs element of two sequences $(J)_4$ and $(K)_4$ into one sequence $(J,K)_4$:

  This is a zip.

Fermilab

What type of things are we dealing with?



- An operation that converts a sequence of elements $(a)_8$ to a sequence of elements $(b)_8$ *of the same length* using a function f:

  This is a map or transform.

- An o… elements $(c)_8$ … $_4$ at a highe…

  This is a fold or reduction.

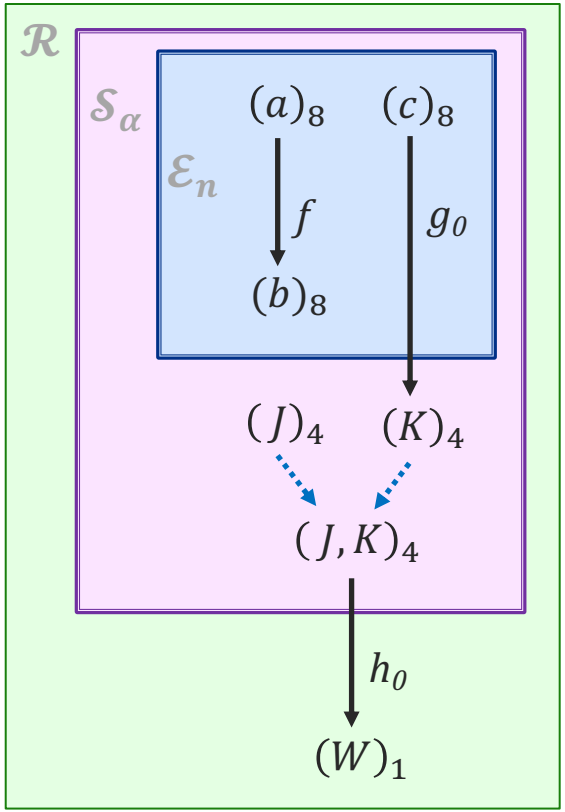
- An operation that pairs element of two sequences $(J)_4$ and $(K)_4$ into one sequence $(J,K)_4$:

  This is a zip.

These have to do with higher-order functions.

Fermilab

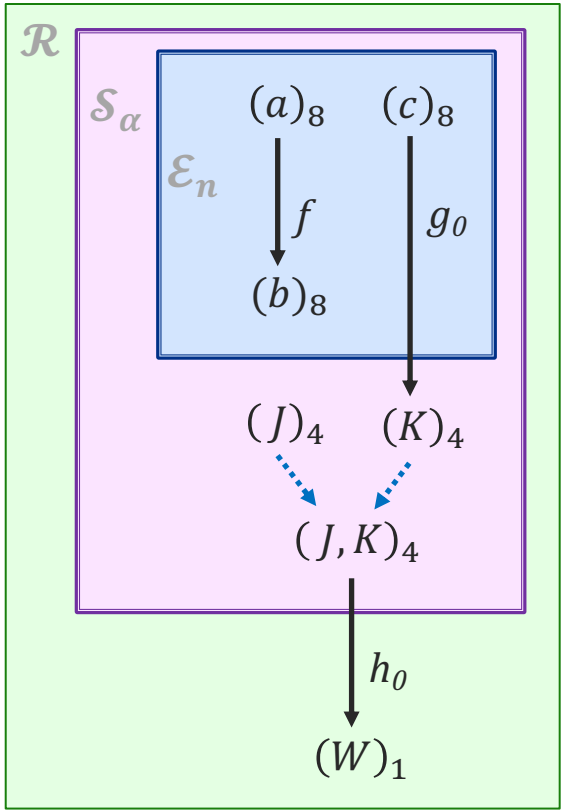
Graph of data-product sequences



View	Nodes	Edges	
Data-centric	**Data products**	**Mappings**	*This work*

Fermilab

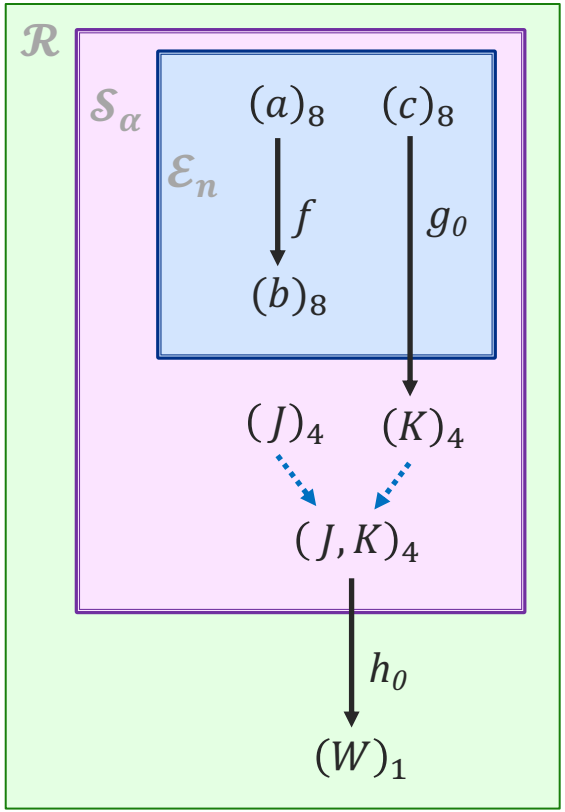
Graph of data-product sequences



View	Nodes	Edges	
Data-centric	**Data products**	**Mappings**	*This work*
Map-centric	Mappings	Data products	*More common*

Fermilab

Graph of data-product sequences



View	Nodes	Edges	
Data-centric	**Data products**	**Mappings**	*This work*
Map-centric	Mappings	Data products	*More common*

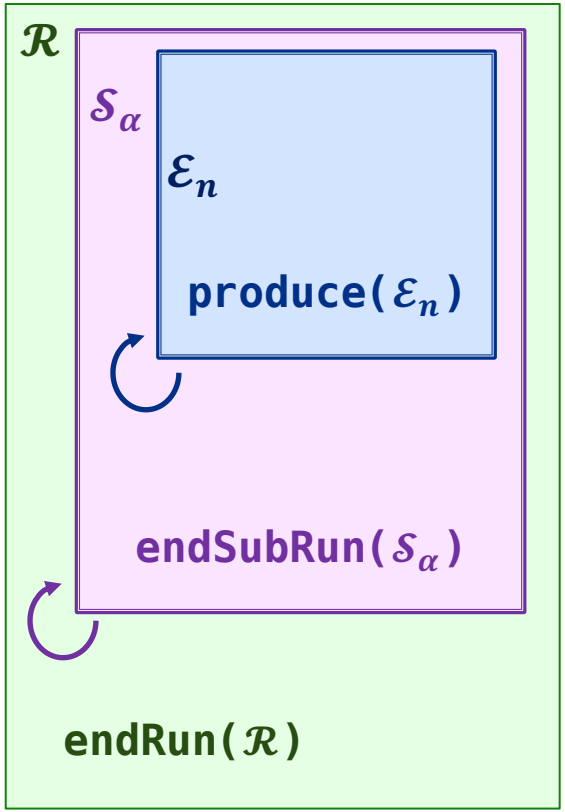
The user specifications are the same with either view:

- Which data products to process
- The data set(s) that contain those products (event, etc.)
- Which higher-order function to use (transform, etc.)
- Which user-defined function to serve as the operation to the higher-order function.
- Allowed concurrency of each function.

The focus is just different.

Fermilab

How are data products and their mappings supported now?

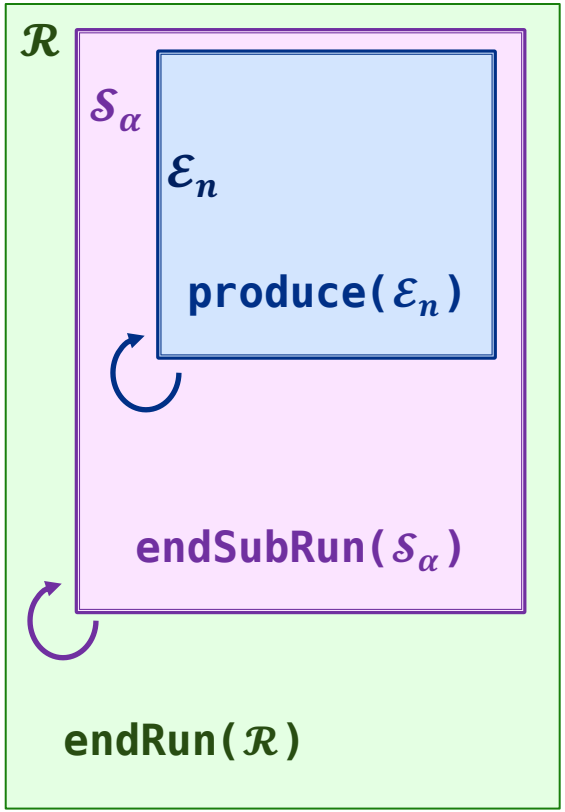


With art, users do not transparently interact with data products. They instead:

- Implement functions based on datasets (e.g. event)
- "Open" the dataset to retrieve and insert products

Fermilab

How are data products and their mappings supported now?



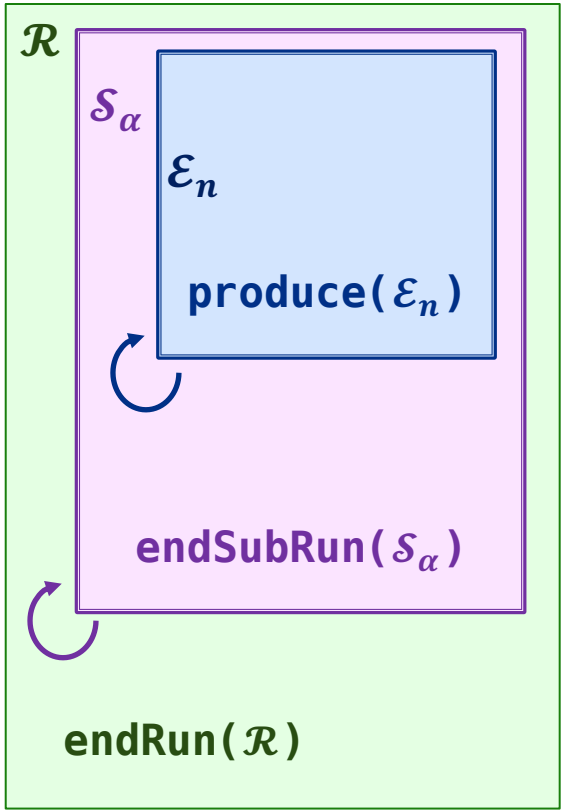
With art, users do not transparently interact with data products. They instead:

- Implement functions based on datasets (e.g. event)
- "Open" the dataset to retrieve and insert products

Some of this is historical and due to:

- The object-oriented nature of the framework.
- Technical limitations of C++ whenever the framework was designed.

How are data products and their mappings supported now?



With art, users do not transparently interact with data products. They instead:

- Implement functions based on datasets (e.g. event)
- "Open" the dataset to retrieve and insert products

Some of this is historical and due to:

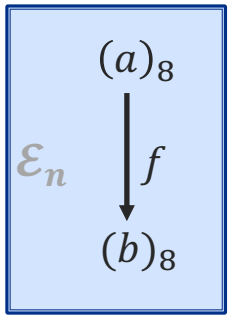
- The object-oriented nature of the framework.
- Technical limitations of C++ whenever the framework was designed.

Results in a lot of software mechanics...

Fermilab

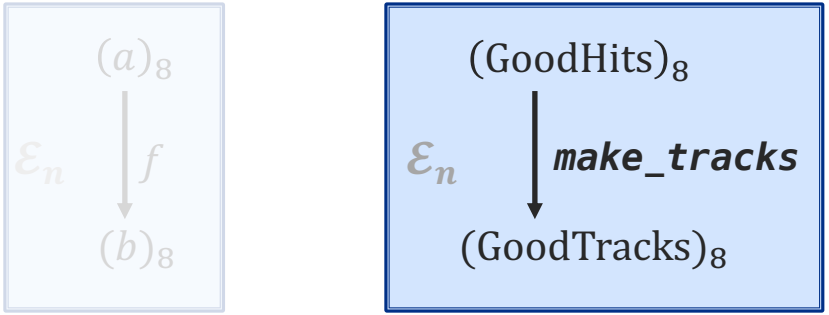
Example: Simple transform

- Create tracks from hits for each event.



Example: Simple transform

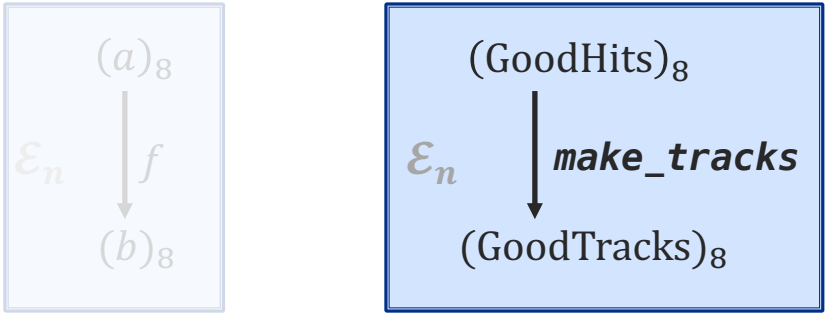
- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

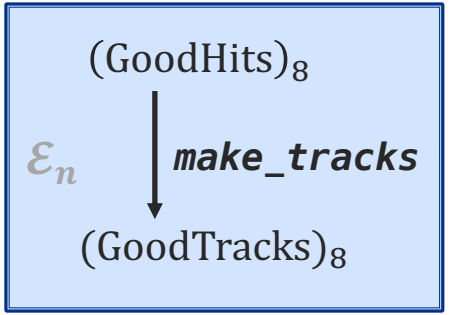
art

```
#include "art/Framework/Core/SharedProducer.h"
#include "art/Framework/Principal/Event.h"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

namespace expt {
  class TrackMaker : public art::SharedProducer {
  public:
    TrackMaker(fhicl::ParameterSet const&) :
    {
      consumes<Hits, art::InEvent>("GoodHits");
      produces<Tracks, art::InEvent>("GoodTracks");

      async<art::InEvent>();
    }

    void produce(art::Event& e,
                 art::ProcessingFrame const&) override
    {
      auto const& hits = e.getProduct<Hits>("GoodHits");
      auto tracks = make_tracks(hits);
      e.put(std::make_unique<Tracks>(std::move(tracks)),
            "GoodTracks");
    }
  };
}

DEFINE_ART_MODULE(expt::TrackMaker)
```


Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

This is just a transform? 😬

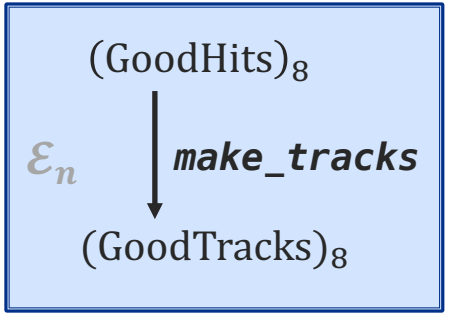
art

```
#include "art/Framework/Core/SharedProducer.h"
#include "art/Framework/Principal/Event.h"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

namespace expt {
  class TrackMaker : public art::SharedProducer {
  public:
    TrackMaker(fhicl::ParameterSet const&) :
    {
      consumes<Hits, art::InEvent>("GoodHits");
      produces<Tracks, art::InEvent>("GoodTracks");

      async<art::InEvent>();
    }

    void produce(art::Event& e,
                 art::ProcessingFrame const&) override
    {
      auto const& hits = e.getProduct<Hits>("GoodHits");
      auto tracks = make_tracks(hits);
      e.put(std::make_unique<Tracks>(std::move(tracks)),
            "GoodTracks");
    }
  };
}

DEFINE_ART_MODULE(expt::TrackMaker)
```


Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

This is just a transform? 😬

art

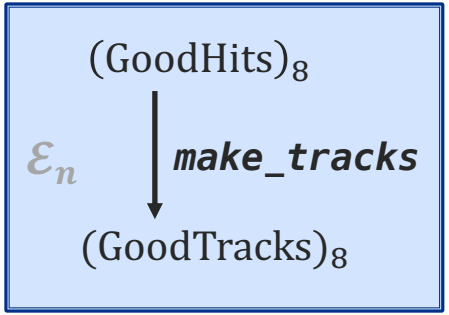
```
#include "art/Framework/Core/SharedProducer.h"
#include "art/Framework/Principal/Event.h"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

namespace expt {
  class TrackMaker : public art::SharedProducer {
  public:
    TrackMaker(fhicl::ParameterSet const&) :
    {
      consumes<Hits, art::InEvent>("GoodHits");
      produces<Tracks, art::InEvent>("GoodTracks");

      async<art::InEvent>();
    }

    void produce(art::Event& e,
                 art::ProcessingFrame const&) override
    {
      auto const& hits = e.getProduct<Hits>("GoodHits");
      auto tracks = make_tracks(hits);
      e.put(std::make_unique<Tracks>(std::move(tracks)),
            "GoodTracks");
    }
  };
}

DEFINE_ART_MODULE(expt::TrackMaker)
```


Fermilab

Example: Simple transform

art

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

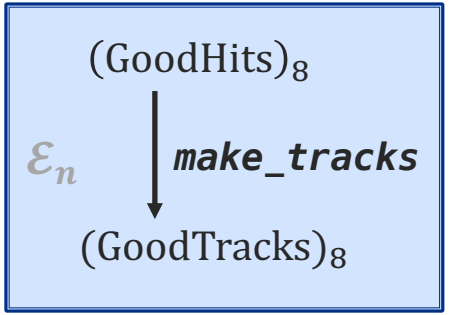
This is just a transform? 😬
Nobody wants this.

```
#include "art/Framework/Core/SharedProducer.h"
#include "art/Framework/Principal/Event.h"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

namespace expt {
  class TrackMaker : public art::SharedProducer {
  public:
    TrackMaker(fhicl::ParameterSet const&) :
    {
      consumes<Hits, art::InEvent>("GoodHits");
      produces<Tracks, art::InEvent>("GoodTracks");

      async<art::InEvent>();
    }

    void produce(art::Event& e,
                 art::ProcessingFrame const&) override
    {
      auto const& hits = e.getProduct<Hits>("GoodHits");
      auto tracks = make_tracks(hits);
      e.put(std::make_unique<Tracks>(std::move(tracks)),
            "GoodTracks");
    }
  };
}

DEFINE_ART_MODULE(expt::TrackMaker)
```


Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

Meld

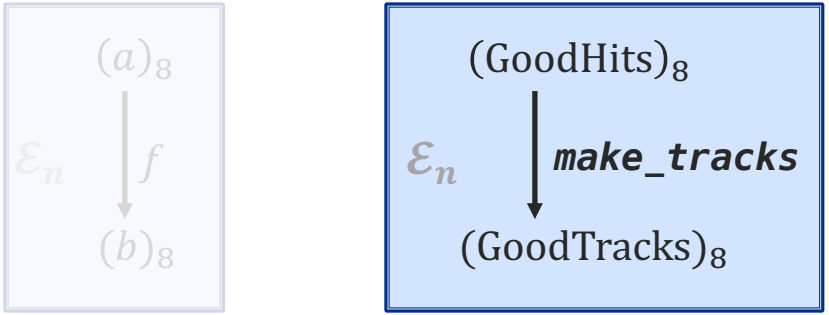
```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits").in_each("Event")
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```

A better way…

Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

Meld

```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits").in_each("Event")
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```

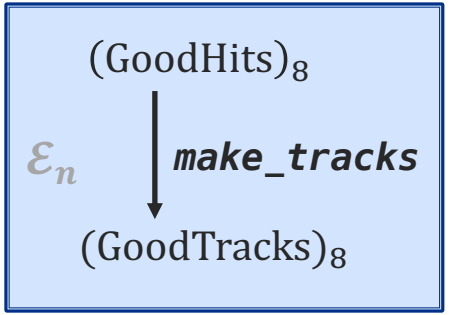
- Minimal boilerplate.

A better way…

Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

A better way…

Meld

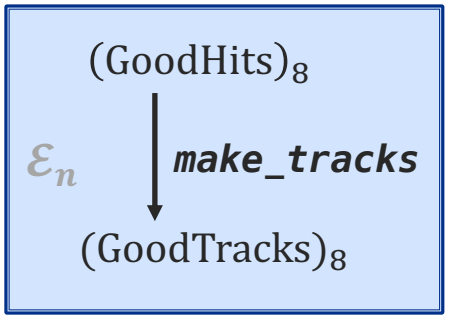
```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits").in_each("Event")
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```

- Minimal boilerplate.
- `Event` is now a label.

Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

A better way…

Meld

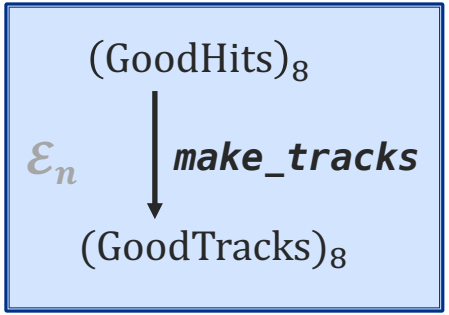
```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits").in_each("Event")
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```

- Minimal boilerplate.
- `Event` is now a label.
- Higher-order function is now explicit.

Fermilab

Example: Simple transform

- Create tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits) { ... }
```

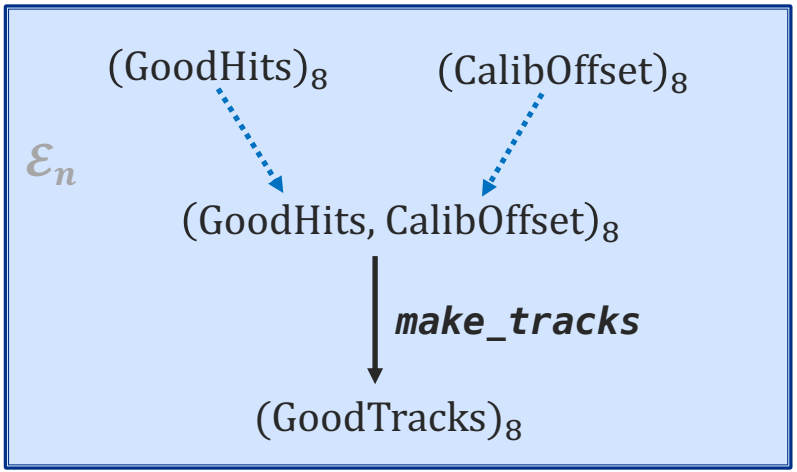
Meld

```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits").in_each("Event")
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```

"That's a nice, clean, toy problem. I'd like to see what reality looks like."
—A member of the ATLAS experiment

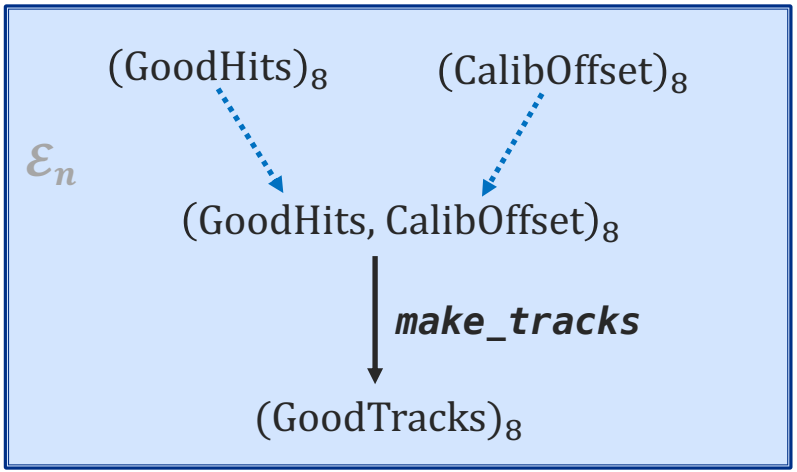
Example: Transform with two arguments

- Create calibrated tracks from hits for each event.



Example: Transform with two arguments

- Create calibrated tracks from hits for each event.

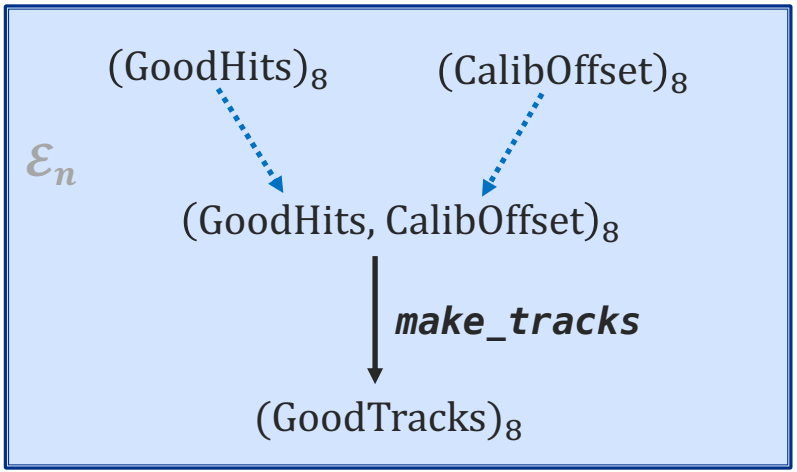



```
Tracks make_tracks(Hits const& hits,
                   ScalarOffset const& offset)
{ ... }
```


Fermilab

Example: Transform with two arguments

- Create calibrated tracks from hits for each event.



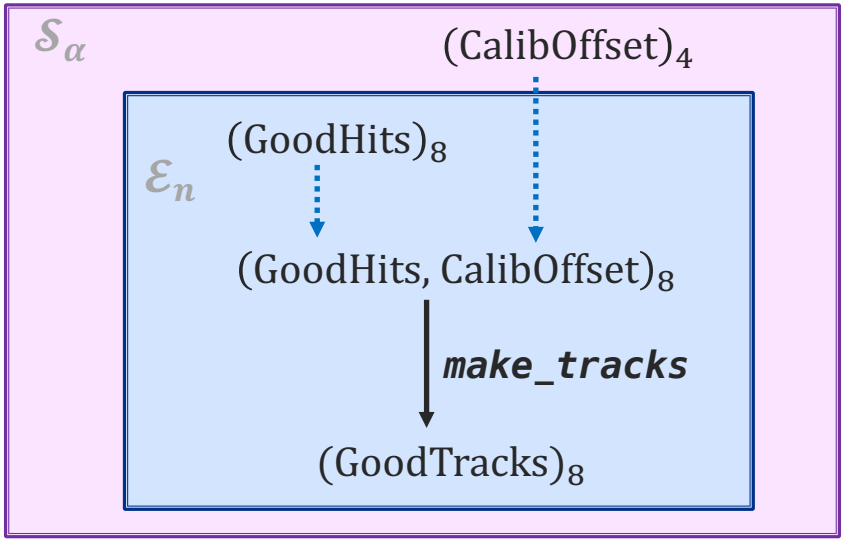

```
Tracks make_tracks(Hits const& hits,
                   ScalarOffset const& offset)
{ ... }
```

```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits,
                     ScalarOffset const& offset)
  { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits"_p.in_each("Event"),
              "CalibOffset"_p.in_each("Event"))
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```


Fermilab

Example: Transform with two arguments (different domains)

- Create calibrated tracks from hits for each event.



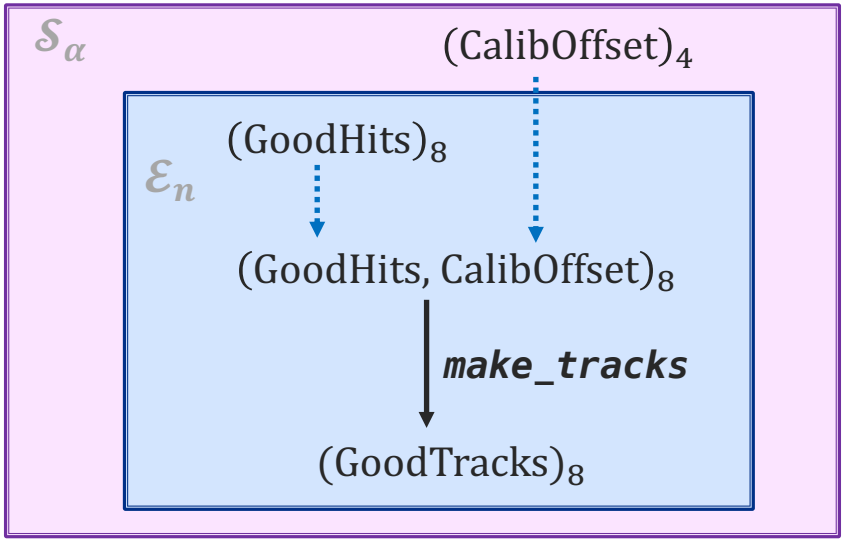

```
Tracks make_tracks(Hits const& hits,
                   ScalarOffset const& offset)
{ ... }
```

```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits,
                     ScalarOffset const& offset)
  { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits"_p.in_each("Event"),
              "CalibOffset"_p.in_each("SubRun"))
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```


Fermilab

Example: Transform with two arguments (different domains)

- Create calibrated tracks from hits for each event.




```
Tracks make_tracks(Hits const& hits,
                   ScalarOffset const& offset)
{ ... }
```

```
#include "meld/module.hpp"

namespace {
  Tracks make_tracks(Hits const& hits,
                     ScalarOffset const& offset)
  { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform(config.get<input_tag>("HitsProduct"),
              config.get<input_Tag>("OffsetProduct"))
   .to(config.get<input_tag>("TracksProduct"))
   .using_concurrency(
      config.get<unsigned int>("Concurrency"));
}
```

```
# Configuration for the above user-registered function
{
  HitsProduct: { name: "GoodHits" domain: "Event"}
  OffsetProduct: { name: "CalibOffset" domain: "SubRun"}
  TracksProduct: "GoodTracks"
  Concurrency: 4
}
```


Fermilab

Meld implementation

- https://github.com/knoepfel/meld (not even alpha release)
- Implemented using oneTBB's flow graph 

Supported construct	User function	
Transform (Map)	$f(\boldsymbol{a}) \rightarrow \boldsymbol{b}$	*Standard data-processing idioms*
Filter	$f(\boldsymbol{a}) \rightarrow \text{Boolean}$	
Monitor	$f(\boldsymbol{a}) \rightarrow \text{Void}$	
Reduction (Fold)	$f_c(\boldsymbol{a}) \rightarrow \boldsymbol{c}$	*For splitting and then combining events*
Splitter (Unfold)	$f_n(\boldsymbol{a}) \rightarrow (\boldsymbol{d})_n$	
Zip	—	*For combining arguments to user functions*
Sliding window	—	*To do:* ***For sliding over adjacent events***

Sample hierarchies tested by Meld

```
[info] Number of worker threads: 12
[info] Processed levels:

  job
   │
   └ run: 1
      │
      └ subrun: 2
         │
         └ event: 10

[info] CPU efficiency: 259.55%
[info] Max. RSS: 6.205 MB
```

art-based hierarchy

Performance numbers are preliminary

Sample hierarchies tested by Meld

```
[info] Number of worker threads: 12
[info] Processed levels:

  job
   │
   └ run: 1
      │
      └ subrun: 2
         │
         └ event: 10

[info] CPU efficiency: 259
[info] Max. RSS: 6.205 MB
```

art-based hierarc

Performance numbers are preliminary

```
[info] Number of worker threads: 12
[info] Processed levels:

  job
   │
   ├ trigger primitive: 10
   │
   └ run: 2
      │
      └ event: 10

[info] CPU efficiency: 230.81%
[info] Max. RSS: 6.136 MB
```

Non-trivial hierarchy

Sample hierarchies tested by Meld

Performance numbers are preliminary

```
[info] Number of worker threads: 12
[info] Processed levels:

  job
   │
   └ run: 1
      │
      └ subrun: 2
         │
         └ event: 10

[info] CPU efficiency: 259
[info] Max. RSS: 6.205 MB
```

art-based hierarc

```
[info] Number of worker threads: 12
[info] Processed levels:

  job
   │
   ├ trigger primitive: 10
   │
   └ run: 2
      │
      └ event: 10

[info] CPU efficiency: 230.81%
[info] Max. RSS: 6.136 MB
```

Non-trivial hierarchy

```
[info] Number of worker threads: 12
[info] Processed levels:

  job
   │
   └ event: 100000

[info] CPU efficiency: 882.50%
[info] Max. RSS: 16.527 MB
```

Flat hierarchy

But I have all this art code…

- There are a few million lines of art-aware code out there.
- Is it possible for meld to be backwards compatible?
- Conceptually not difficult.

But I have all this art code…

- There are a few million lines of art-aware code out there.
- Is it possible for meld to be backwards compatible?
- Conceptually not difficult.

```
#include "art/Framework/Core/EDProducer.h"

namespace expt {
  class MyProducer : public art::EDProducer {
  public:
    MyProducer(fhicl::ParameterSet const& pset);
    void beginRun(art::Run& r) override;
    void beginSubRun(art::SubRun& sr) override;
    void produce(art::Event&) override;
    void endSubRun(art::SubRun& sr) override;
    void endRun(art::Run& r) override;
  };

  // Implementation ...
}

DEFINE_ART_MODULE(expt::MyProducer)
```

But I have all this art code…

- There are a few million lines of art-aware code out there.
- Is it possible for meld to be backwards compatible?
- Conceptually not difficult.

```
#include "art/Framework/Core/EDProducer.h"

namespace expt {
  class MyProducer : public art::EDProducer {
  public:
    MyProducer(fhicl::ParameterSet const& pset);
    void beginRun(art::Run& r) override;
    void beginSubRun(art::SubRun& sr) override;
    void produce(art::Event&) override;
    void endSubRun(art::SubRun& sr) override;
    void endRun(art::Run& r) override;
  };

  // Implementation ...
}

DEFINE_ART_MODULE(expt::MyProducer)
```

But I have all this art code…

- There are a few million lines of art-aware code out there.
- Is it possible for meld to be backwards compatible?
- Conceptually not difficult.

```
#include "art/Framework/Core/EDProducer.h"

namespace expt {
  class MyProducer : public art::EDProducer {
  public:
    MyProducer(fhicl::ParameterSet const& pset);
    void beginRun(art::Run& r) override;
    void beginSubRun(art::SubRun& sr) override;
    void produce(art::Event&) override;
    void endSubRun(art::SubRun& sr) override;
    void endRun(art::Run& r) override;
  };

  // Implementation ...
}

DEFINE_ART_MODULE(expt::MyProducer)
```

Can expand to

```
DEFINE_MODULE(m, pset)
{
  auto bound_obj = m.make<expt::MyProducer>(pset);
  bound_obj.with_legacy(&expt::MyProducer::beginRun)
           .process<Run>();
  bound_obj.with_legacy(&expt::MyProducer::beginSubRun)
           .process<SubRun>().following<SubRun>();
  bound_obj.with_legacy(&expt::MyProducer::produce)
           .reduce<Event>().to<SubRun>();
  bound_obj.with_legacy(&expt::MyProducer::endSubRun)
           .reduce<SubRun>().to<Run>();
  bound_obj.with_legacy(&expt::MyProducer::beginRun)
           .reduce<Run>();
}
```

Summary

> ***"Ways change, Stil."*** —Paul from *Dune* by Frank Herbert

- Supporting DUNE's framework needs suggests rethinking framework concepts.

Summary

> ***"Ways change, Stil."*** —Paul from *Dune* by Frank Herbert

- Supporting DUNE's framework needs suggests rethinking framework concepts.
- Meld seeks to address these needs by considering a framework job as a

(1) graph of data products *connected by*
(2) user-provided operations *of*
(3) higher-order functions.

- Preliminary work indicates this is a productive avenue to pursue.
- Conceptually possible to be backwards compatible with existing art modules.

Summary

> ***"Ways change, Stil."*** —Paul from *Dune* by Frank Herbert

- Supporting DUNE's framework needs suggests rethinking framework concepts.
- Meld seeks to address these needs by considering a framework job as a

  (1) graph of data products *connected by*
  (2) user-provided operations *of*
  (3) higher-order functions.

- Preliminary work indicates this is a productive avenue to pursue.
- Conceptually possible to be backwards compatible with existing art modules.

Thank you for your time and attention.

Backup slides

Accessing provenance information

```
#include "meld/module.hpp"

namespace {
-  Tracks make_tracks(Hits const& hits) { ... }
+  Tracks make_tracks(meld::handle<Hits> hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(make_tracks)
   .transform("GoodHits").in_each("Event")
   .to("GoodTracks")
   .using_concurrency(unlimited);
}
```

Class example using lambda expression

```
#include "meld/module.hpp"

DEFINE_MODULE(m, config)
{
  auto threshold = config.get<unsigned int>("threshold");
  m.with([threshold](Hits const& hits) { return hits.size() > threshold; })
   .filter("GoodHits").in_each("Event")
   .using_concurrency(unlimited);
}
```

Class example registering two member functions

```
#include "meld/module.hpp"

class Selector {
public:
  Selector(unsigned int n) : threshold{n} {}
  bool gt(Hits const& hits) const { return hits.size() > threshold; }
  bool le(Hits const& hits) const { return !gt(hits); }

private:
  unsigned int threshold;
};

DEFINE_MODULE(m, config)
{
  auto threshold = config.get<unsigned int>("threshold");
  auto bound_m = m.make<Selector>(threshold);
  bound_m.with(&Selector::gt).filter("GoodHits").in_each("Event");
  bound_m.with(&Selector::le).filter("GoodHits").in_each("Event");
}
```

Reduction example

```
class MyAccumulator : public art::EDProducer {
public:
  MyAccumulator(ParameterSet const&)
  {
    produces<int, art::InSubRun>("sum");
  }

  void produce(art::Event&) override
  {
    ++counter_;
  }

  void endSubRun(art::SubRun& sr) override
  {
    sr.put(std::make_unique<int>(counter_), "sum");
    counter_ = 0;
  }

private:
  int counter_ = 0;
};

DEFINE_ART_MODULE(MyAccumulator)
```

```
void accumulate(int& counter,
                meld::level_id const&)
{
  ++counter;
}


DEFINE_MODULE(m) {
  m.with(accumulate, 0).for_each("SubRun")
   .reduce("id").in_each("Event")
   .to("sum");
}
```

Higher-order functions

- We are interested in the mappings of the form:

$$\left\{(\boldsymbol{a})_n \xrightarrow{f} (\boldsymbol{b})_m\right\} \in \mathcal{D}$$

- Each object $\boldsymbol{a}$ corresponds to a tuple of arguments passed to f.
- The signature of f and the value $f(\boldsymbol{a})$, depends on the higher-order function.
- The above mapping happens within a domain $\mathcal{D}$ (e.g. job, run, event).
- Each object $\boldsymbol{a}$ is an element of a subset of the domain $\mathcal{D}$.

Supported higher-order functions

Meld term	CS term	Mathematical description		Domain
Transform	Map	$(\boldsymbol{a})_n \xrightarrow{f} (\boldsymbol{b})_n$	where $f(\boldsymbol{a}) \to \boldsymbol{b}$	Same as $(\boldsymbol{a})_n$
Filter	Filter	$(\boldsymbol{a})_n \xrightarrow{f} (\boldsymbol{a})_m$ where $m \leq n$	where $f(\boldsymbol{a}) \to \mathrm{Boolean}$	Same as $(\boldsymbol{a})_n$
Monitor	—	$(\boldsymbol{a})_n \xrightarrow{f} (\ )_0$	where $f(\boldsymbol{a}) \to \mathrm{Void}$	Same as $(\boldsymbol{a})_n$
Reduction	Fold	$(\boldsymbol{a})_n \xrightarrow{f_c} (\boldsymbol{c})_1$	where $f_c(\boldsymbol{a}) \to \boldsymbol{c}$	**Above** $(\boldsymbol{a})_n$
Splitter	Unfold	$(\boldsymbol{a})_1 \xrightarrow{f_n} (\boldsymbol{d})_m$	where $f_n(\boldsymbol{a}) \to (\boldsymbol{d})_n$	**Below** $(\boldsymbol{a})_n$
Zip	Zip	$\left((a)_n, (b)_n\right) \to (a, b)_n$		More nested domain