Ordered Processing In Apache Beam

Sergei Lilichenko

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Intro

- Solutions Architect at Google Cloud
  - Write Beam pipelines as proof-of-concepts for customers and internal teams
  - Publish blogs and architecture guides
  - Sometimes contribute to the Beam repo
- Enterprise architect and tech lead (FSIs, startups) in my prior life

Agenda

- What is ordered processing and why do we care?
- Why Beam and not [fill in blanks]?
- What is available in Apache Beam?
- … and how does it work?
- Is there more that's coming?
- Lessons learned

BEAM SUMMIT

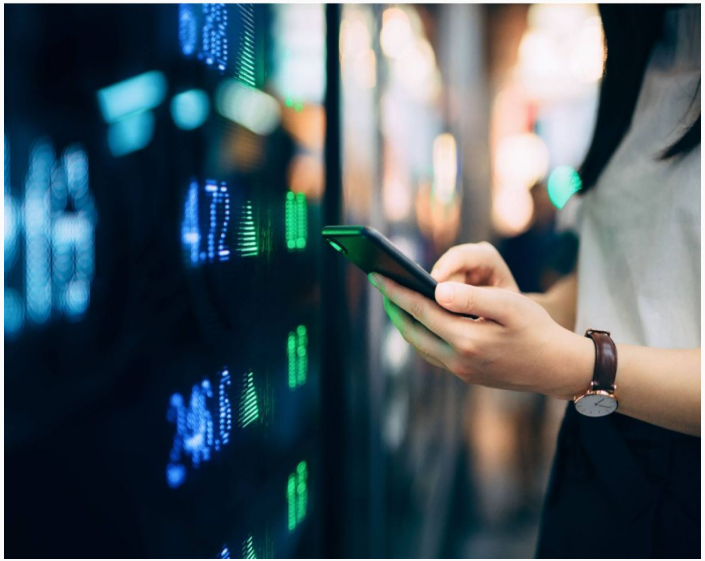
Why ordered processing?

BEAM SUMMIT

Use Case: Order book building

Order Book Processing in Financial Services. “An order book lists the number of shares being bid on or offered at each price point, or market depth.”

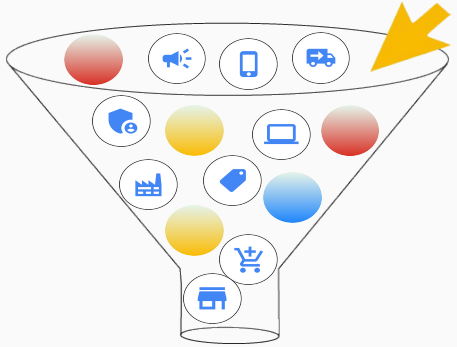
An order book is created per security or financial instrument and must be created in the strict sequence - no gaps in events and events should be applied exactly as they happened in time.

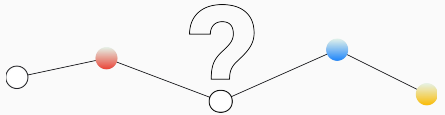


Use Cases: IoT processing, click stream analysis

Processing data from sensors. For the use cases where it's important to process data in the exact sequence, for example to be able to catch an anomaly between two subsequent sensor reads.

Close-to-real-time clickstream analytics. Typically done using time windows to collect all the events for a particular time period, but it requires waiting for the window to close. Ordered processing can reduce processing latencies by guaranteeing that all previous events has been received.





Generic system requirements

- Ability to simultaneously handle ordered processing of thousands or millions of individual ticker symbols, sensors, users, etc.
- Ability to maintain the state of processing
- Ideally, use no additional services for processing besides Beam-provided facilities
- Low latency streaming processing
- Batch processing for historical analysis

BEAM SUMMIT

Ordered processing can be complex

Infrastructure

Not every messaging system supports ordered delivery of the messages to the processing system.

The order of publishing is not always the order of processing.

Code

Coding large scale ordered processing “from scratch” is difficult. Many customers don’t have sufficient expertise in their development teams, or it’s not their core business.

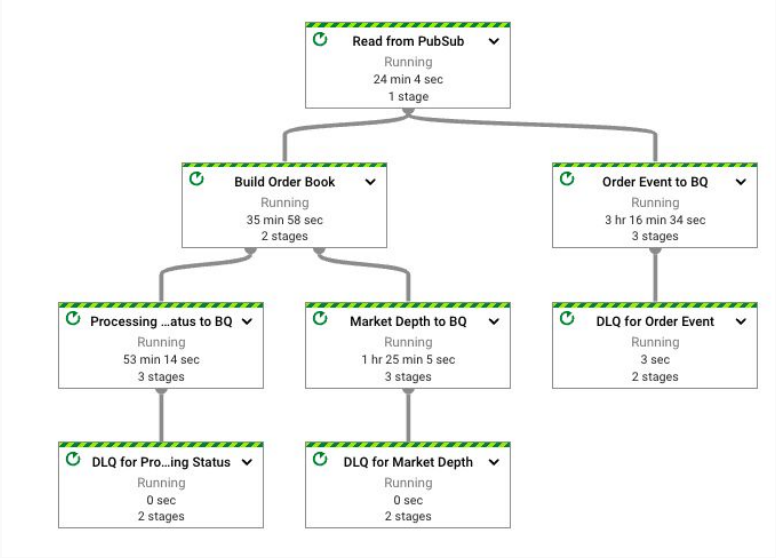
BEAM SUMMIT

Beam pipelines to the rescue…

- Best streaming product on the market
- Works really well with Pub/Sub and Kafka
- Fully managed, great observability, etc.
- Highly scalable and capable of processing millions of events per second
- Automatically manages work distribution and synchronization per key

… but …

- Had limited support for ordered processing out-of-the-box
- Programming model takes time to learn



What is available today?

BEAM SUMMIT

“ordered” extension added in release 2.56.0

- A generic Apache Beam transform that processes elements in order for each key
- Several interfaces customers need to implement. Implementation will not require knowledge of advanced Apache Beam concepts

A reference implementation of a complex customer use case (order book processing) in GitHub’s GoogleCloudPlatform/dataflow-ordered-processing project

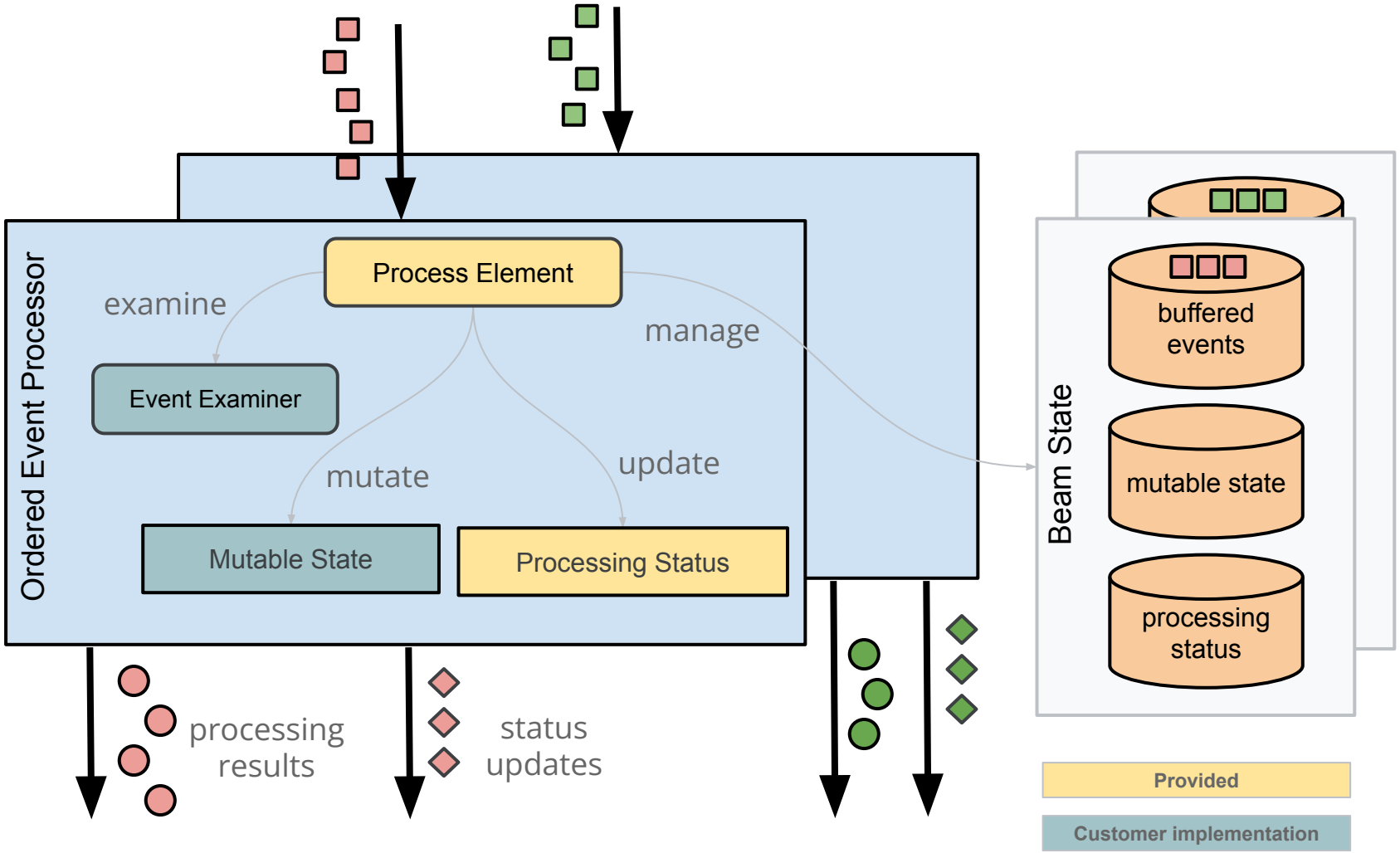
In some cases, DoFn.RequiresTimeSortedInput annotation or “sorter” extension can work.

Can any data be processes in order?

Prerequisites

- Data must be in a KV<Key, KV<Long, Event>> shape, where the Long key is the sequence number
- There should be no gaps in the sequence for a particular key
- Initial sequence can be any number
- Optionally, there is a way to know the end of a sequence
- The pipeline needs to be written using Java SDK (multi-language pipelines can be used to call this transform from Python/Go pipelines)

BEAM SUMMIT


Ordered Event Processor
Process Element
examine
manage
Event Examiner
mutate
update
Mutable State
Processing Status
processing results
status updates
Beam State
buffered events
mutable state
processing status
Provided
Customer implementation

Why does it work?

- Beam guarantees that only a single stateful DoFn will process a PCollection's elements at any time for a pair of key/window
- Buffered events are stored in the OrderedListState, which allows for efficient insertion of the elements and efficiently retrieves the elements sorted by the sequence number.
- There is no excessive memory pressure because the buffered elements are not stored in memory and are not sorted in memory.

BEAM SUMMIT

How hard is it to use?

Isn’t it too complex for most customers?

- All the heavy lifting is done by the generic transform
- Customer only needs to implement several interfaces in plain Java
- No advanced knowledge of Apache Beam is required
- References implementation and documentation provides the jump start

BEAM SUMMIT

How practical is it?

Isn’t it too simplistic for sophisticated cases?

- An FSI customer successfully implemented their order book processing using earlier version of the code
- Reference GitHub project implements a use case of an above-average complexity
- OrderedEventProcessor produces processing status updates which can be used to monitor processing progress per key
- The transform has advanced customization options
- All the code is open sourced. Customers can request FRs or can clone and customize the solution

BEAM SUMMIT

What about the performance?

It works well. This approach will have lower latency than many other solutions which require waiting until all data is received before it can be sorted and processed.

Example: 100M orders for 5,000 securities (⅛ of typical Nasdaq trading daily volume) were processed under 19 minutes.

Ordered processing performance will depend of maximum number of elements for a particular key (e.g., security).

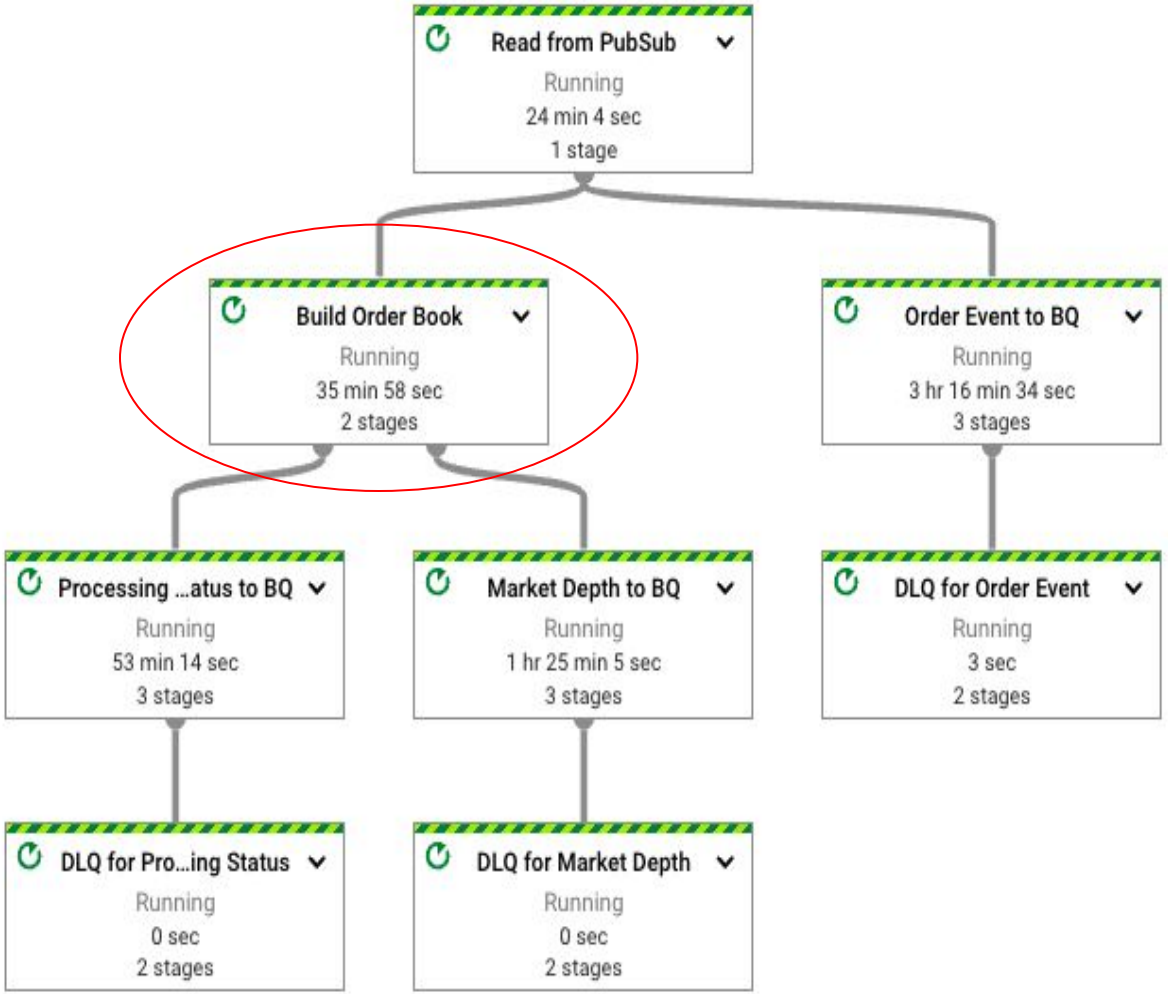
Is it expensive?

No. Under normal circumstances (events arriving mostly in order) the ordered transform performs similarly to many core SDK transforms.

Example: processing 100M orders for 5,000 securities was estimated to cost under $6 when using new Streaming billing model. This doesn't include Pub/Sub and BigQuery costs, just direct Dataflow costs.

	Cost	Adjustments	Net cost
vCPU	$3.05	$0.00	$3.05
Memory	$0.63	$0.00	$0.63
Streaming Engine Compute Units	$2.04	$0.00	$2.04
HDD	$0.04	$0.00	$0.04
SSD	$0.00	$0.00	$0.00
Total	**$5.75**	**$0.00**	**$5.75**

BEAM SUMMIT



End result

This is how a fully functional pipeline looks like. The complexity of the order book building is hidden inside a high level Beam transform.

The developer can focus on how to get the data in and how to persist the results of processing. BigQuery is just one option here, The data can be stored anywhere.

Event Examiner

Called by the generic transform to describe the event. Needs to implement only three methods.

```
@Override
public boolean isInitialEvent(long sequenceNumber, OrderBookEvent event) {
       // Assume all trades for a day will start with 1.
       return sequenceNumber == 1L;
}

@Override
public OrderBookMutableState createStateOnInitialEvent(OrderBookEvent event) {
       OrderBookMutableState orderBookMutableState =
                             new OrderBookMutableState(depth, withTrade);
       orderBookMutableState.mutate(event);
       return orderBookMutableState;
}

@Override
public boolean isLastEvent(long sequenceNumber, OrderBookEvent event) {
       return event.getLastContractMessage();
}
```

Mutable State

Implements the business logic of processing the events.

```
OrderBookMutableState(int depth, boolean withTrade) {
  this.depth = depth;
  this.withTrade = withTrade;
  this.orderBookBuilder = new OrderBookBuilder();
}

@Override
public void mutate(OrderBookEvent event) {
  orderBookBuilder.processEvent(event);
}

@Override
public MarketDepth produceResult() {
  return orderBookBuilder.getCurrentMarketDepth(depth, withTrade);
}
```

Handler

Configuration class. Primary purpose is to provide the event examiner to the generic transform. Parent class provides the default implementations for most of the required methods.

```
public OrderBookOrderedProcessingHandler(int depth, boolean withLastTrade) {
    super(    OrderBookEvent.class,
              SessionContractKey.class,
              OrderBookMutableState.class,
              MarketDepth.class);
    this.depth = depth;
    this.withLastTrade = withLastTrade;
}

@Override
public EventExaminer<OrderBookEvent, OrderBookMutableState> getEventExaminer(){
    return new OrderBookEventExaminer(depth, withLastTrade);
}
```

Ordered Event Processor Transform

Finally, this is how the processing is done.

The input is transformed into the required KV<Key, KV<Long, Event>> shape first, and then the generic ordered event processor does the rest.

```
OrderedEventProcessor<OrderBookEvent, SessionContractKey, MarketDepth,
          OrderBookMutableState> orderedProcessor =
      OrderedEventProcessor.create(handler);

return input
  .apply("Convert to KV", ParDo.of(new ConvertOrderBookEventToKV()))
  .apply("Produce OrderBook", orderedProcessor);
```

Coders

A bit of Beam knowledge is still needed; specifically around specifying coders for classes.

They are used for serialization and deserialization of state and data elements.

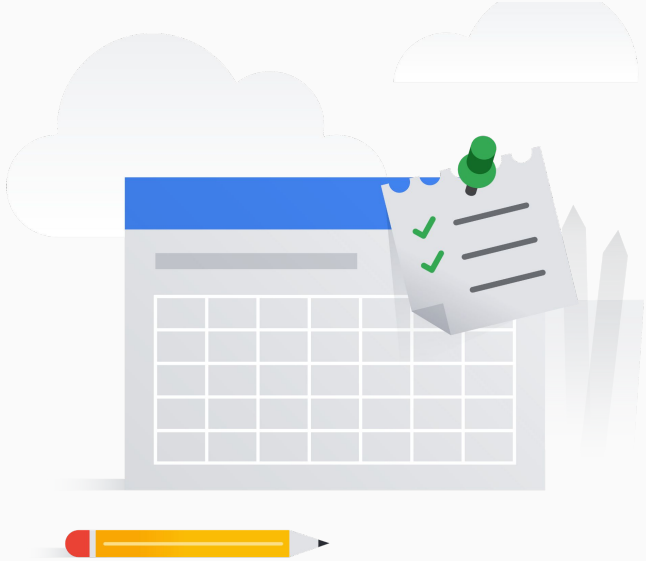
```
@Override
public void encode(OrderBookMutableState state, OutputStream out)
                          throws IOException {
  intCoder.encode(state.getDepth(), out);
  booleanCoder.encode(state.isWithTrade(), out);
  mapCoder.encode(state.getPrices(), out);
  orderBookEventCoder.encode(state.getLastOrderBookEvent(), out);
}

@Override
public OrderBookMutableState decode(InputStream inStream)
                          throws IOException {
  int depth = intCoder.decode(inStream);
  boolean withTrade = booleanCoder.decode(inStream);
  Map<Long, Long> prices = mapCoder.decode(inStream);
  OrderBookEvent lastOrderBookEvent = orderBookEventCoder.decode(inStream);
  OrderBookMutableState orderBookMutableState = new OrderBookMutableState(
           depth, withTrade, prices, lastOrderBookEvent);
  return orderBookMutableState;
}
```

How do I start using this transform?

Check out the unit tests for the “ordered” extension at
https://github.com/apache/beam/tree/master/sdks/java/extensions/ordered

OS GitHub repo
(https://github.com/GoogleCloudPlatform/dataflow-ordered-processing)

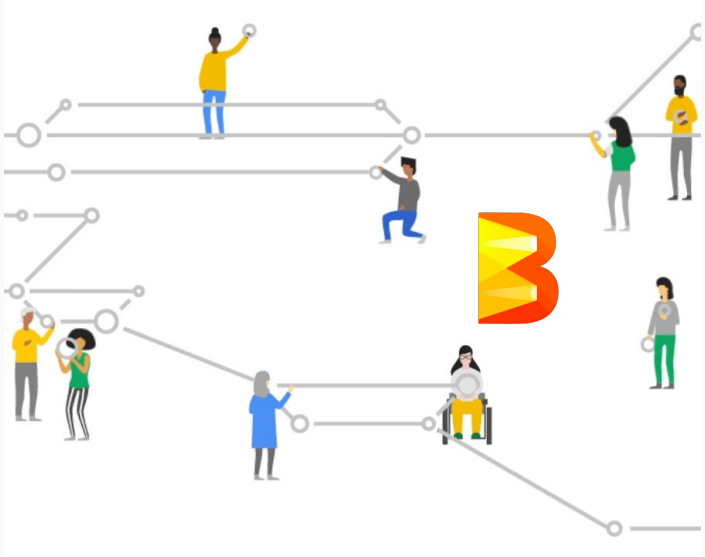
- Reference implementation of the order book processing using a fictitious stock exchange
- Cookbook on how to implement ordered processing
- Simulator to generate desired volume and “shape” of data
- Terraform scripts to create infrastructure
- BigQuery queries to perform data and latency analysis

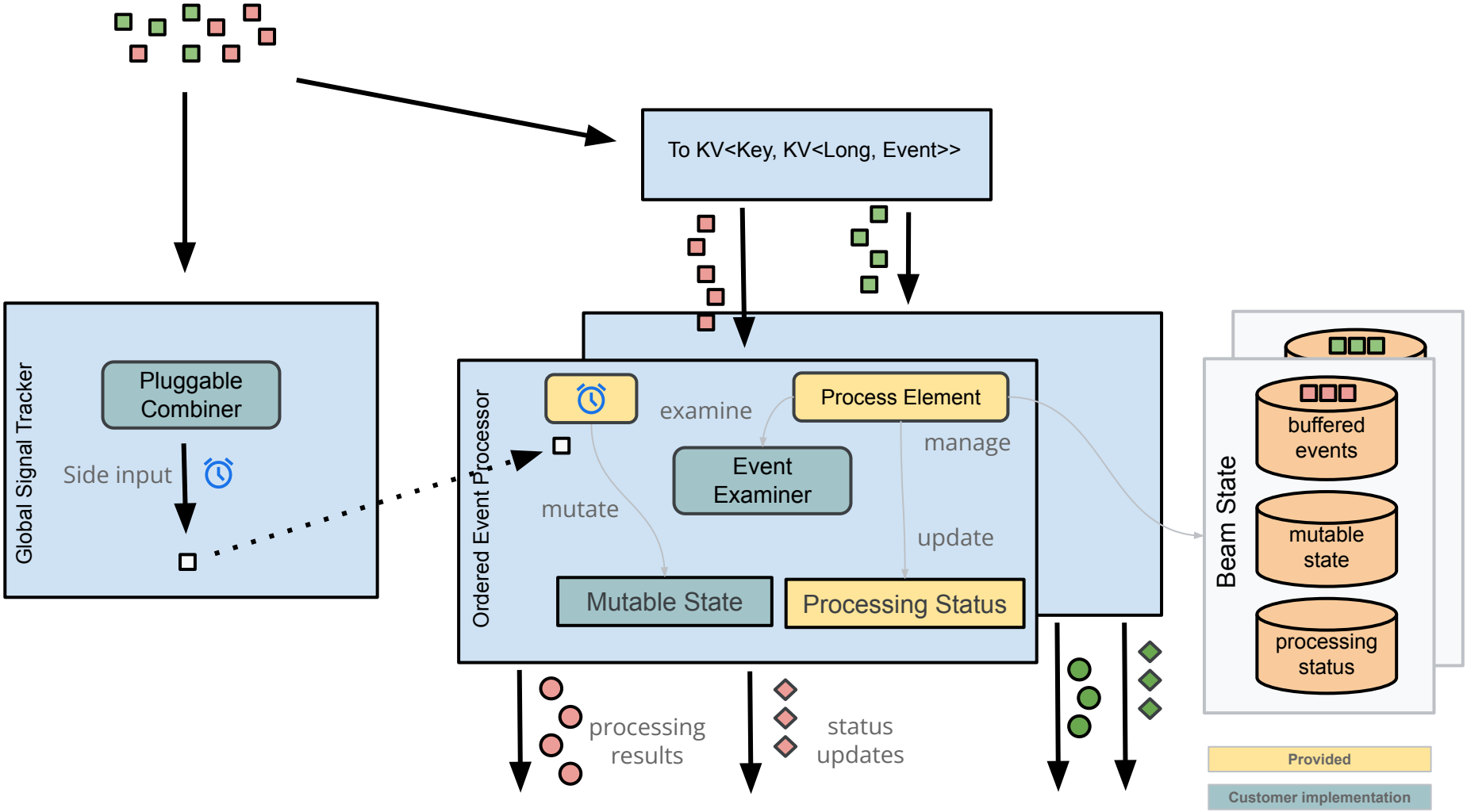


BEAM SUMMIT

Is there more coming?

- Another mode of processing ordered events - when there is a global sequence rather than a per key sequence
- A blog will be published
- An update of the Beam documentation
- Will consider creating wrapper transforms in other SDKs (Python, Go) using multi-language pipelines




To KV<Key, KV<Long, Event>>
Global Signal Tracker
Pluggable Combiner
Side input
Ordered Event Processor
examine
Process Element
manage
Event Examiner
mutate
update
Mutable State
Processing Status
Beam State
buffered events
mutable state
processing status
processing results
status updates
Provided
Customer implementation

Lessons learned

- You, yes - you, can write somewhat complex transforms
- There are a lot of undiscovered gems in Beam States (e.g., OrderedListState)
- Timer concepts are supposed to be trivial, but they are not
- Unit tests are life savers, esp. for streaming pipelines
- Nothing replaces actual integration tests. Every runner has its limitations.
- You have to test at scale and you have to test adverse conditions.

BEAM SUMMIT

Thank you!

Reach out if you have questions:

slilichenko@google.com

https://www.linkedin.com/in/sergei-lilichenko

BEAM SUMMIT