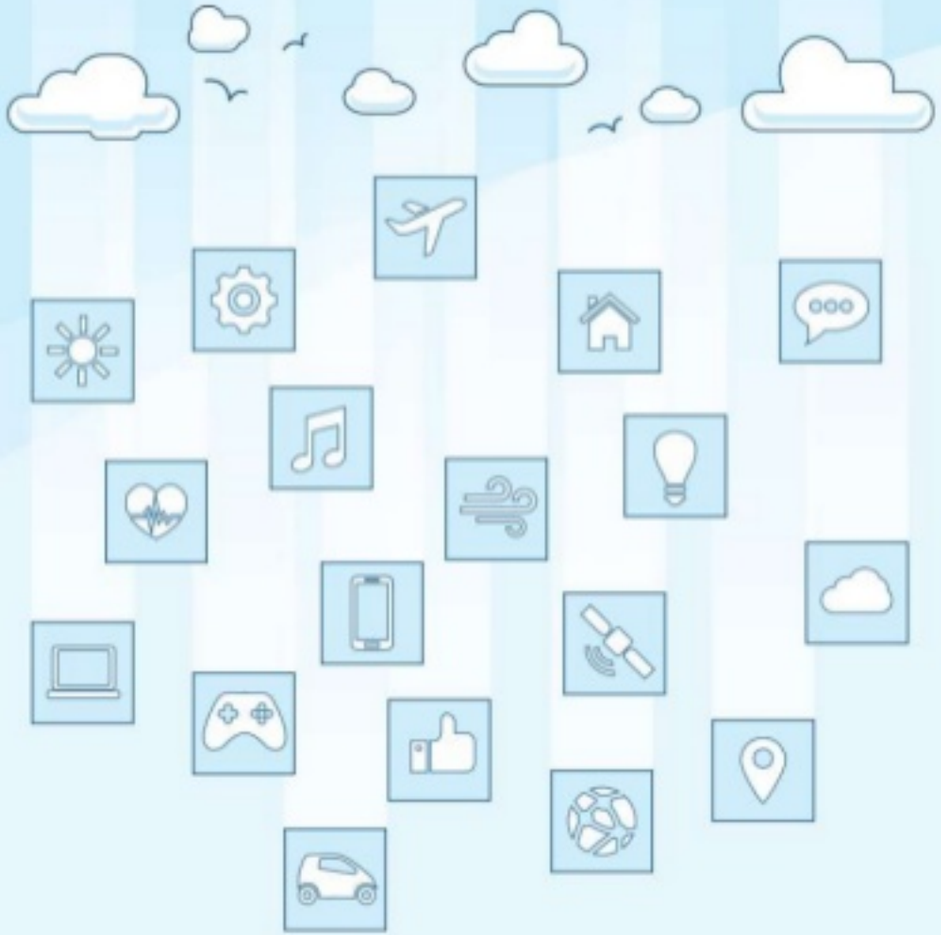


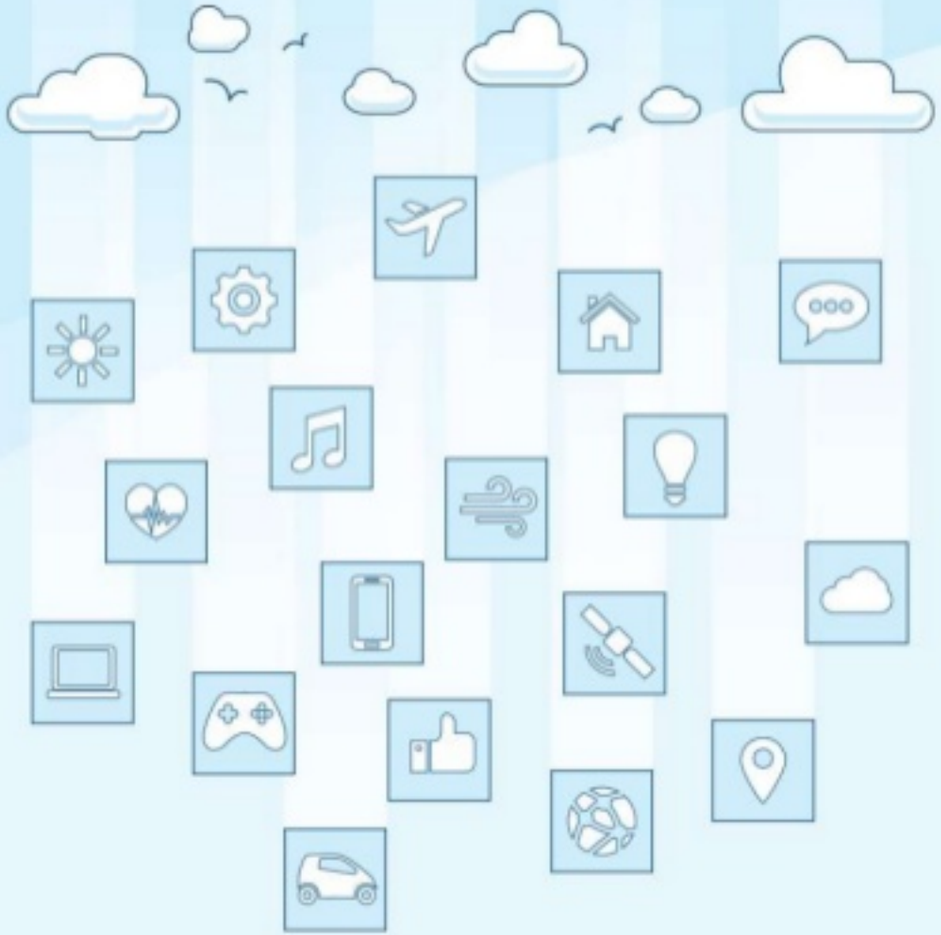
“Play it again, Sam": Bookmarking, Slicing, and Replaying Unbounded Data Streams for Analytics Applications

Raúl Gracia, Pravega by DellEMC

Flink Forward – Berlin 2018

Outline

- Introduction & Motivation
- Pravega: A Storage System for Unbounded Data Streams
- Pravega: Simplifying Processing of Unbounded Data Streams
- StreamCuts in Action: Sample Application
- Next Steps & Conclusions



Introduction & Motivation

The Age of Data Streams

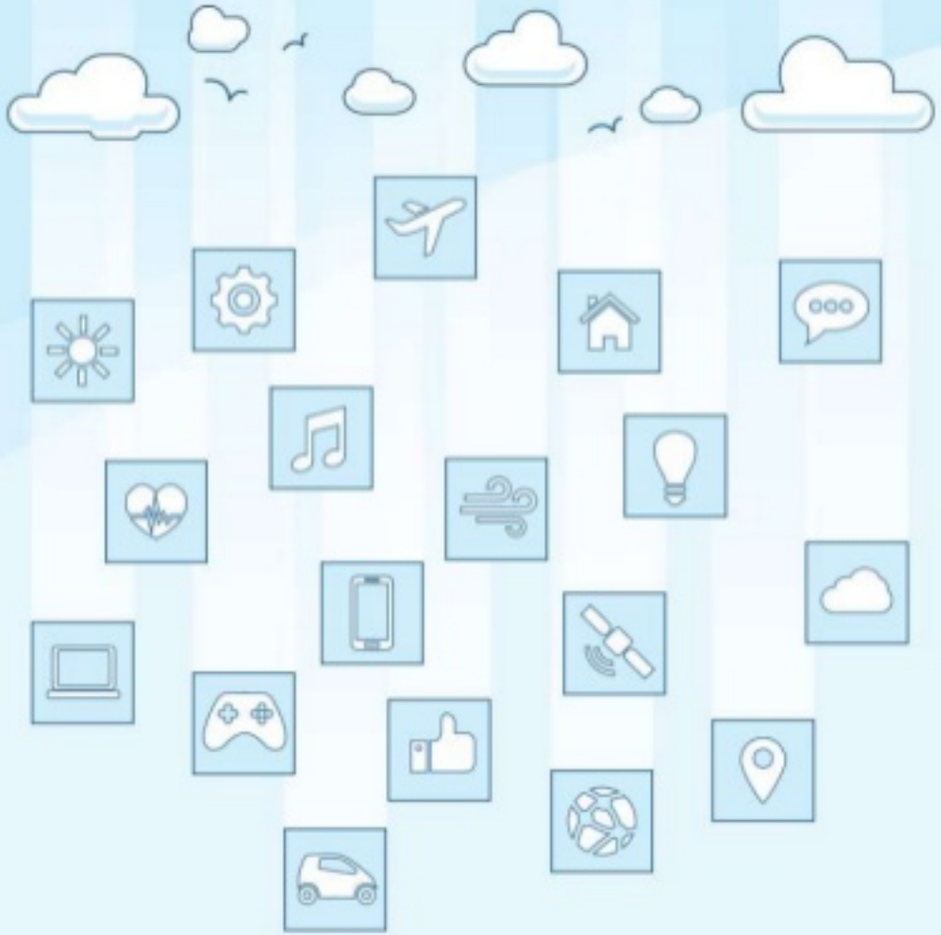
- *Data stream*: first-class citizen to manage and process data.
  - Unbounded sequence of data events.
- *Increasing* number of use-cases that produce data streams.
- Data streams have value both at *real time* and at *rest*.

But Things Are Not Always Easy

Store a stream of consumption values from smart energy meters.

Execute algorithm X in a stream fashion and compare the result with a specific week of last year.

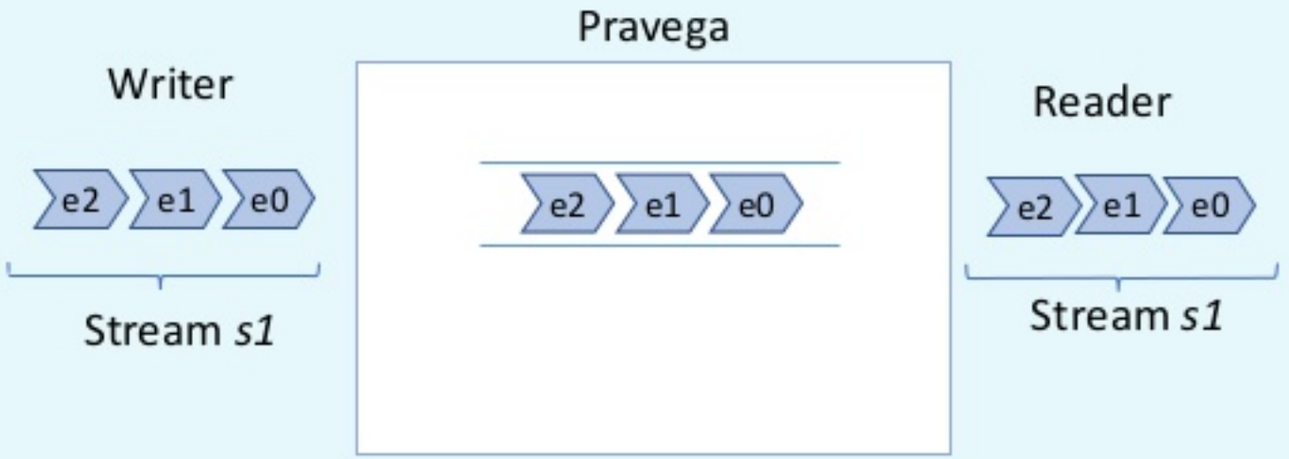
- *Storage-related problems*:
  - How do we store "unbounded" data streams?
  - Latency? Throughput?
  - Which read/write guarantees can we expect?
- *Processing-related problems*:
  - Unified storage solution for both real-time and batch analytics?
  - How do we represent "references" or "slices" in an unbounded data stream?
  - Can we make it easier for developers to process unbounded data streams?
- Our solution: *Pravega (by DellEMC)*.



Pravega: A Storage System for Unbounded Data Streams

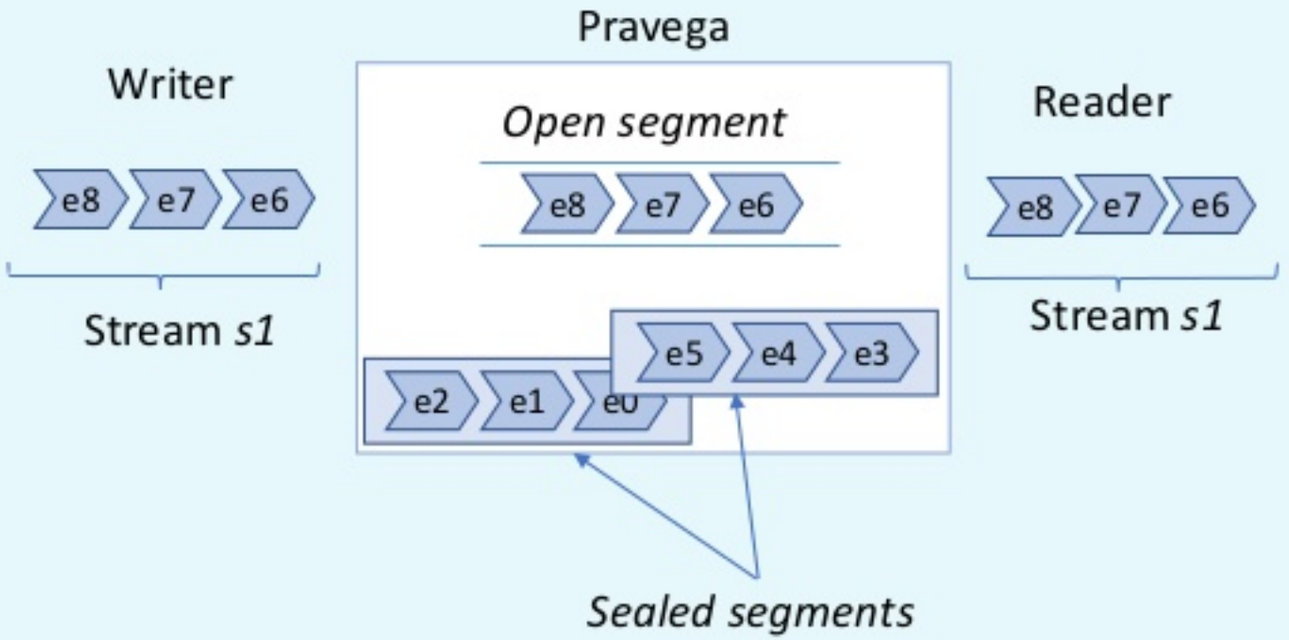
Pravega Concepts I: Streams & Clients

- Pravega is an **open-source storage system** to store/serve **unbounded data streams**.
- **Stream**: Unbounded sequence of bytes.
  - Append-only abstraction (but can be truncated).
- *Clients*: Operate on Streams.
  - **Writer**: `writer.writeEvent(message)`
  - **Reader**: `reader.readNextEvent(timeout)`



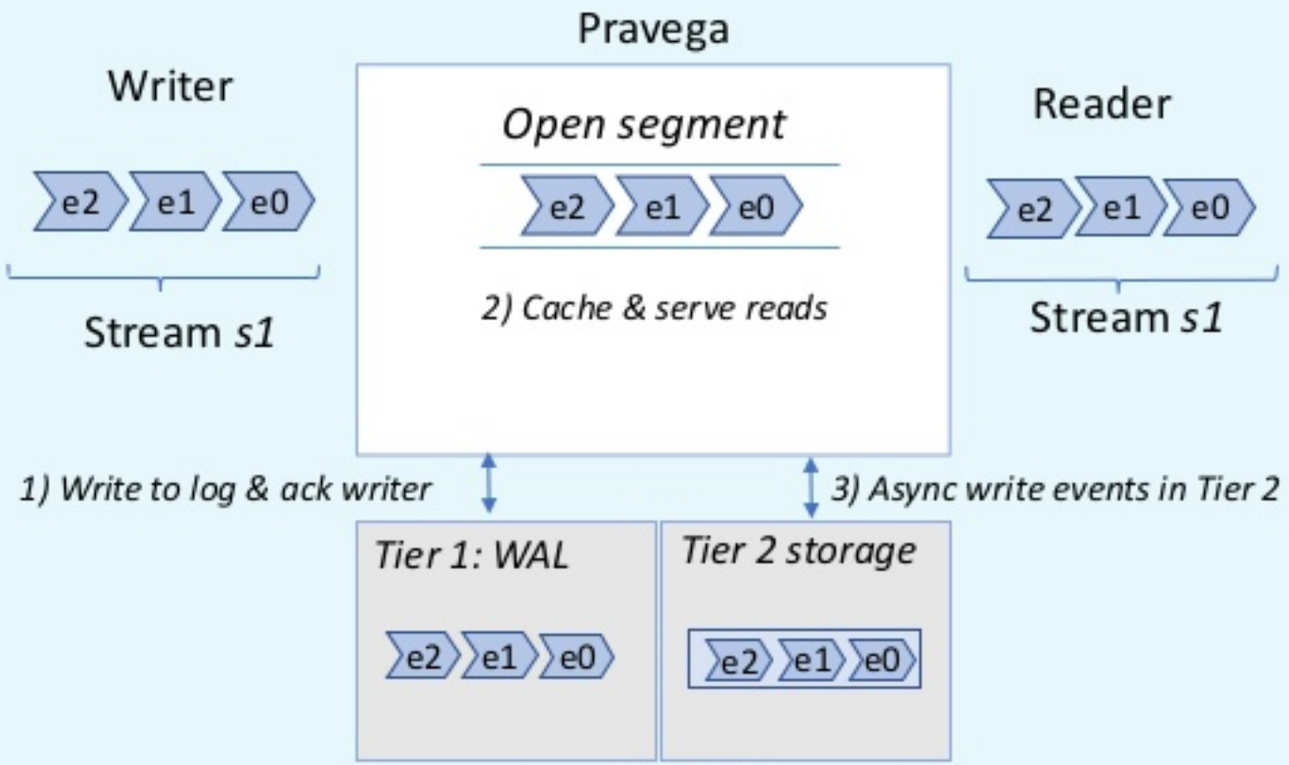
Pravega Concepts II: Stream Segments

- Pravega splits Streams into **segments**:
  - Basic *unit of storage* for Pravega.
- A Stream can be seen as a *sequence of segments*.
- State of segments:
  - *Open* segment: Events are being appended.
  - *Sealed* segment: Read-only.



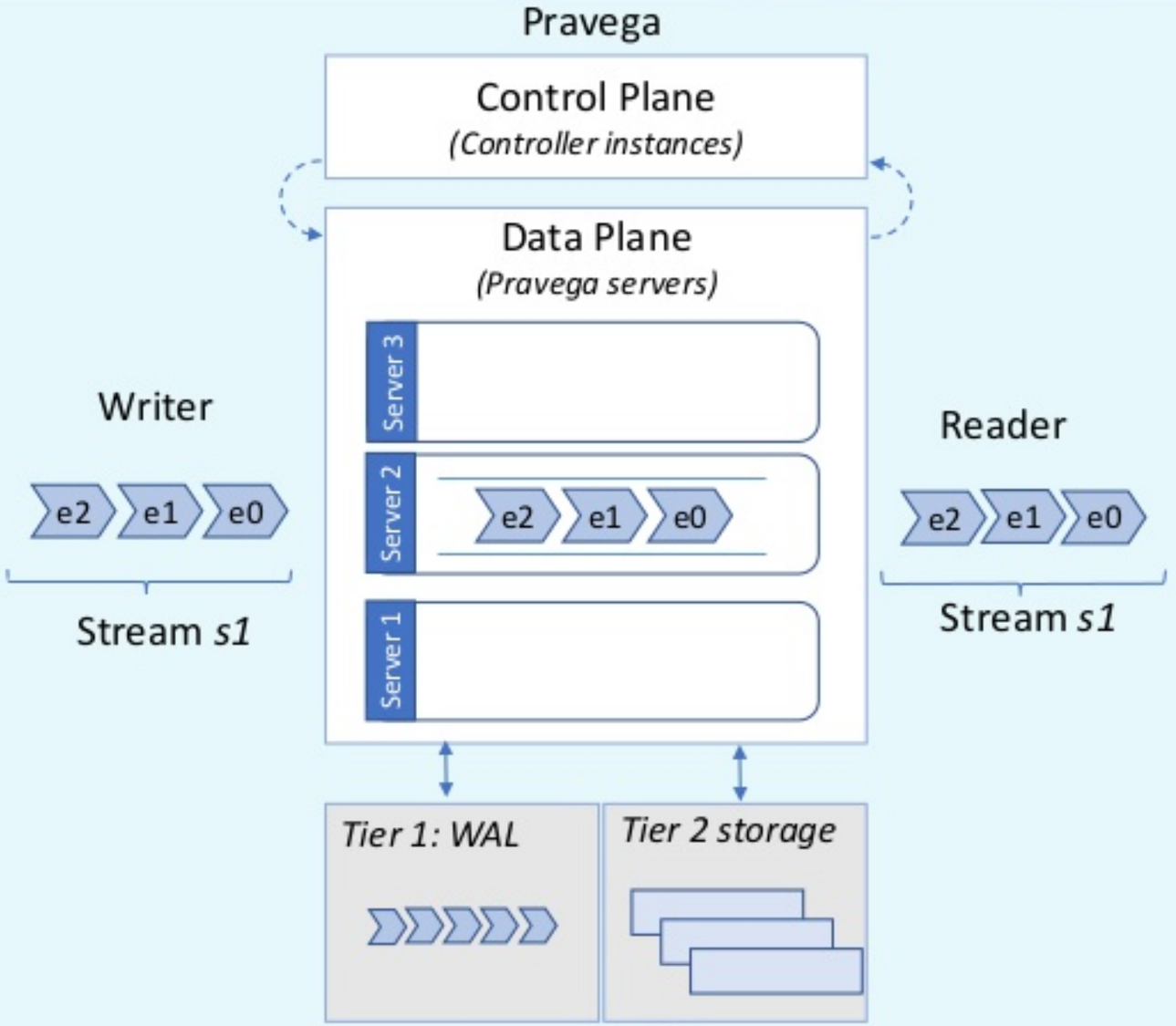
Pravega Tiered Storage

- Open segments are durably written to Tier 1:
  - Low write-to-read latency (*real time analytics*).
  - Write Ahead Log (e.g., Apache Bookkeeper).
  - WAL is only read to recover from failures.
- Sealed segments live in Tier 2 :
  - High throughput (*batch analytics*).
  - Pluggable: HDFS, Amazon S3, DellEMC ECS/Isilon.
- Sweet spot in *latency vs throughput trade-off*.



Pravega's Architecture

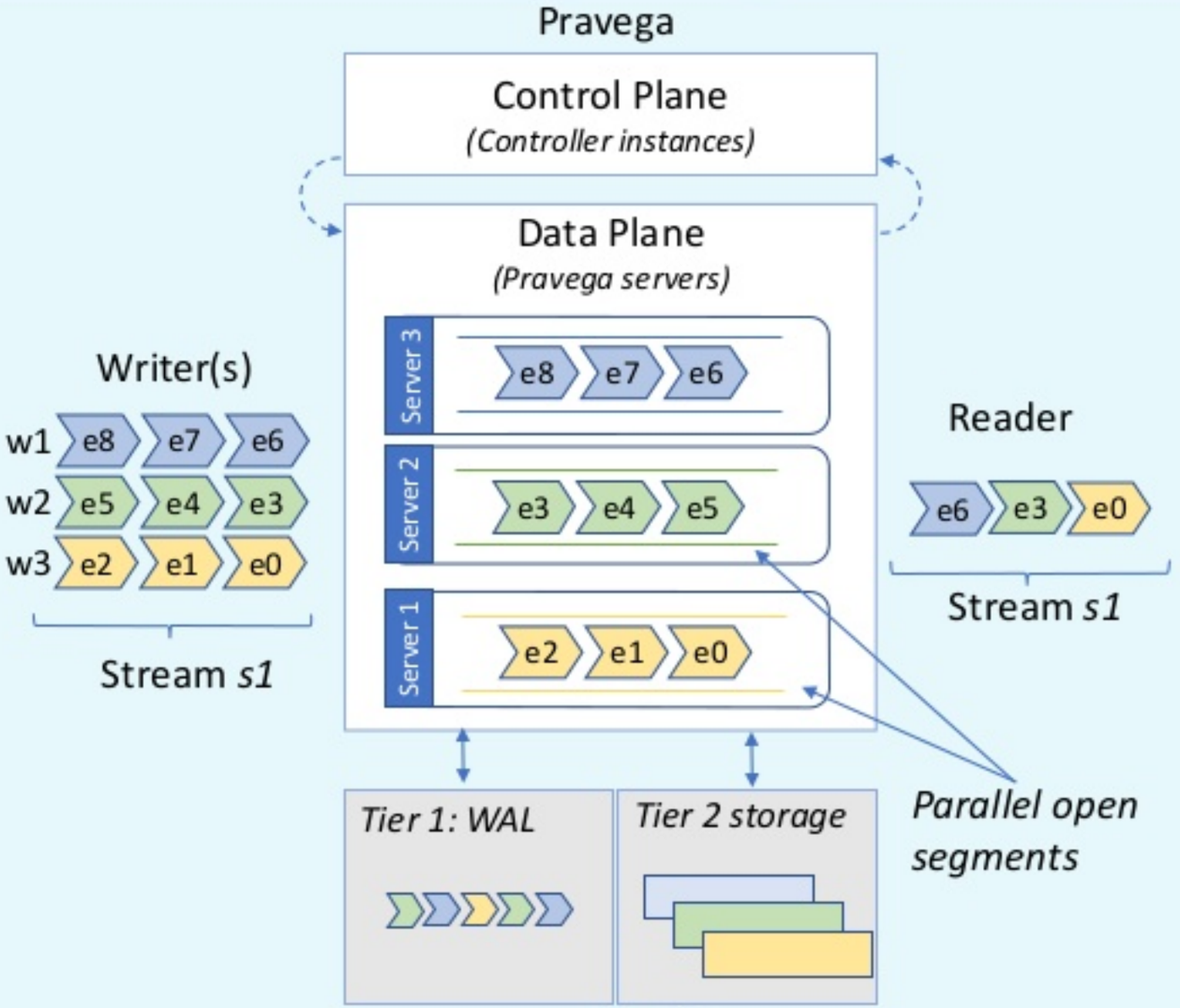
- **Software-defined architecture:** Control/Data planes, policies & feedback loop.
- **Control plane**:
  - Controller instances manage metadata.
  - Notion of what a "Stream" is.
  - Enforce policies at the Stream level (retention, scaling).
- **Data plane**:
  - Formed by Pravega servers that store and serve data.
  - Only understand the notion of "Segment".



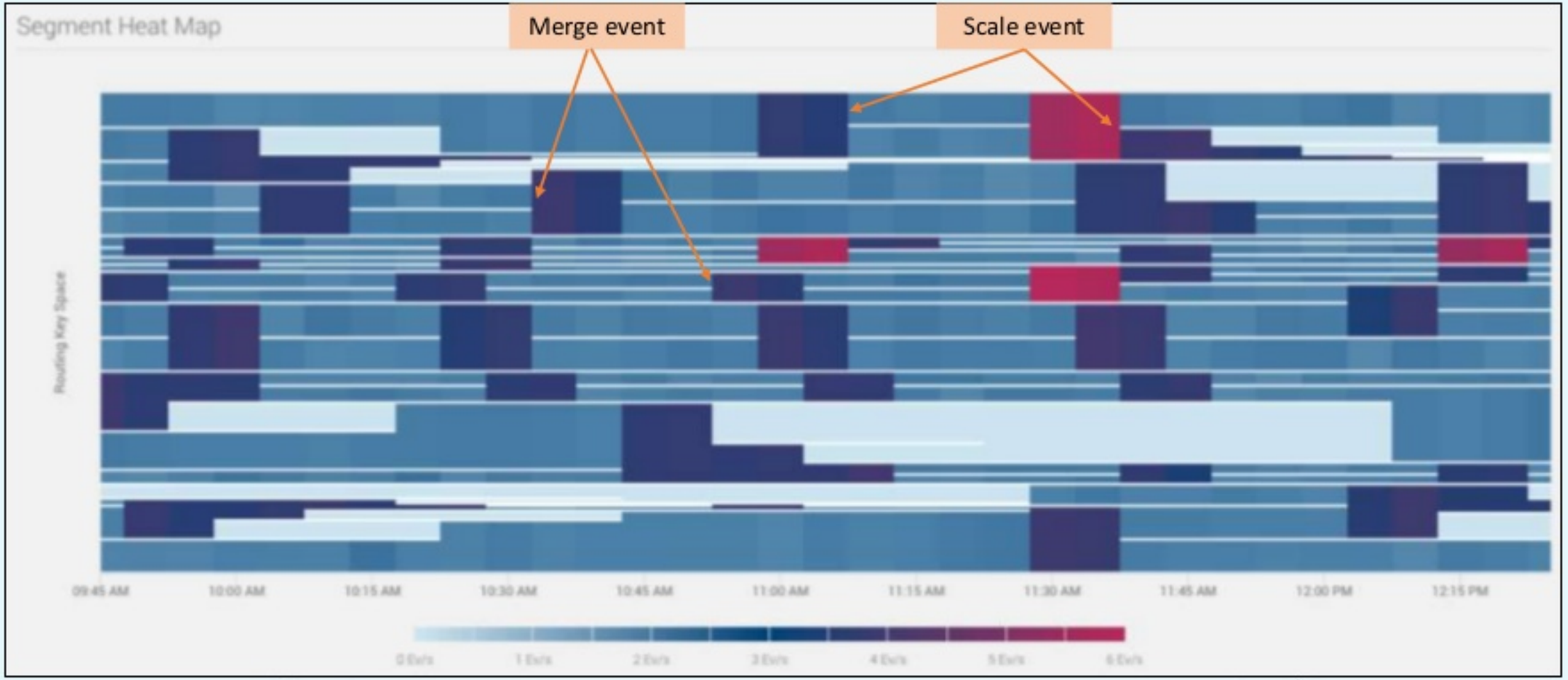
Write Parallelism

- A Stream may have **multiple open segments**:
  - Each segment can be placed in a different Pravega server for higher throughput.
- Writers can **write events in parallel** to a Stream.
- **Write guarantees**:
  - *Exactly-once*: No event duplicates (e.g., on reconnections).
  - All events written to a **routing key** will be read in the *same order as they were written*.

```
writer.writeEvent(routingKey, message)
```



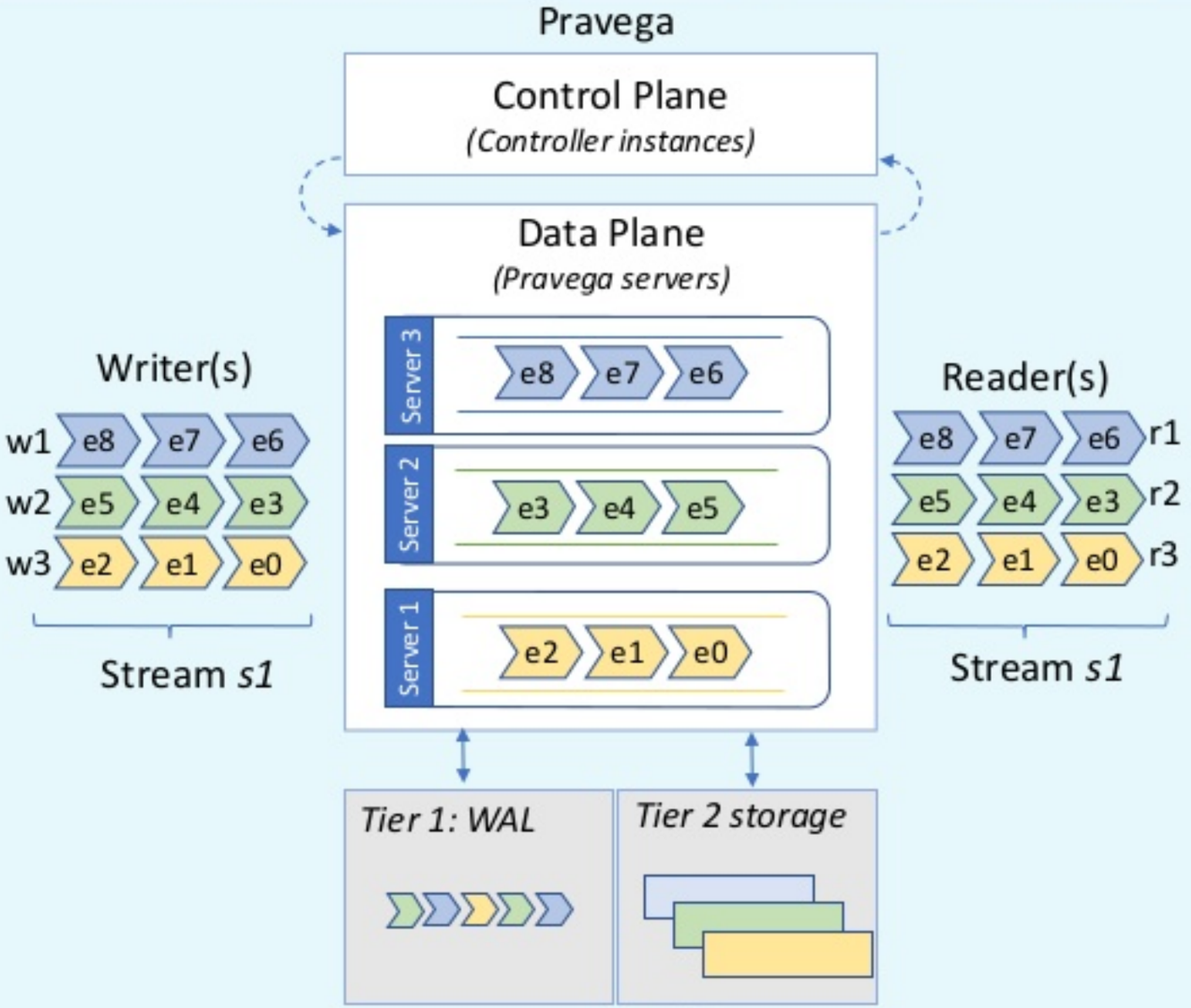
Auto-Scaling



Nautilus platform experiment (virtual cluster)

Read Parallelism

- Readers can **read events in parallel** from Streams:
  - *Tail reads* (Pravega cache), *catch-up reads* (Tier2 via Pravega cache).
- **Read guarantees**:
  - All the events from a set of Streams will be read by *only one reader in a group of readers.*
  - *Application support for reader recovery*: Consistent information of reader positions.

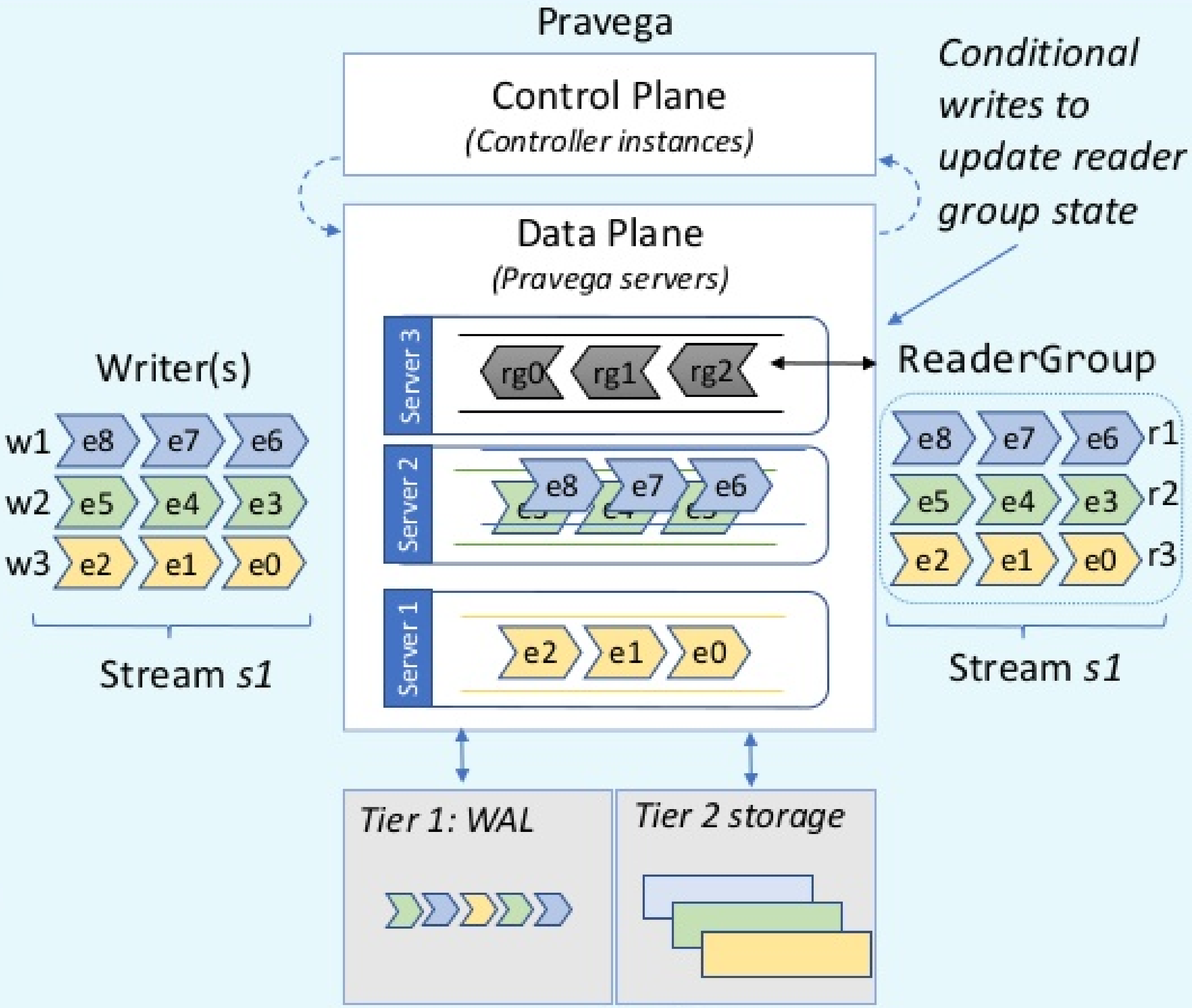


The ReaderGroup

- **ReaderGroup**: Abstraction to coordinate a set of readers to read from one or many streams.

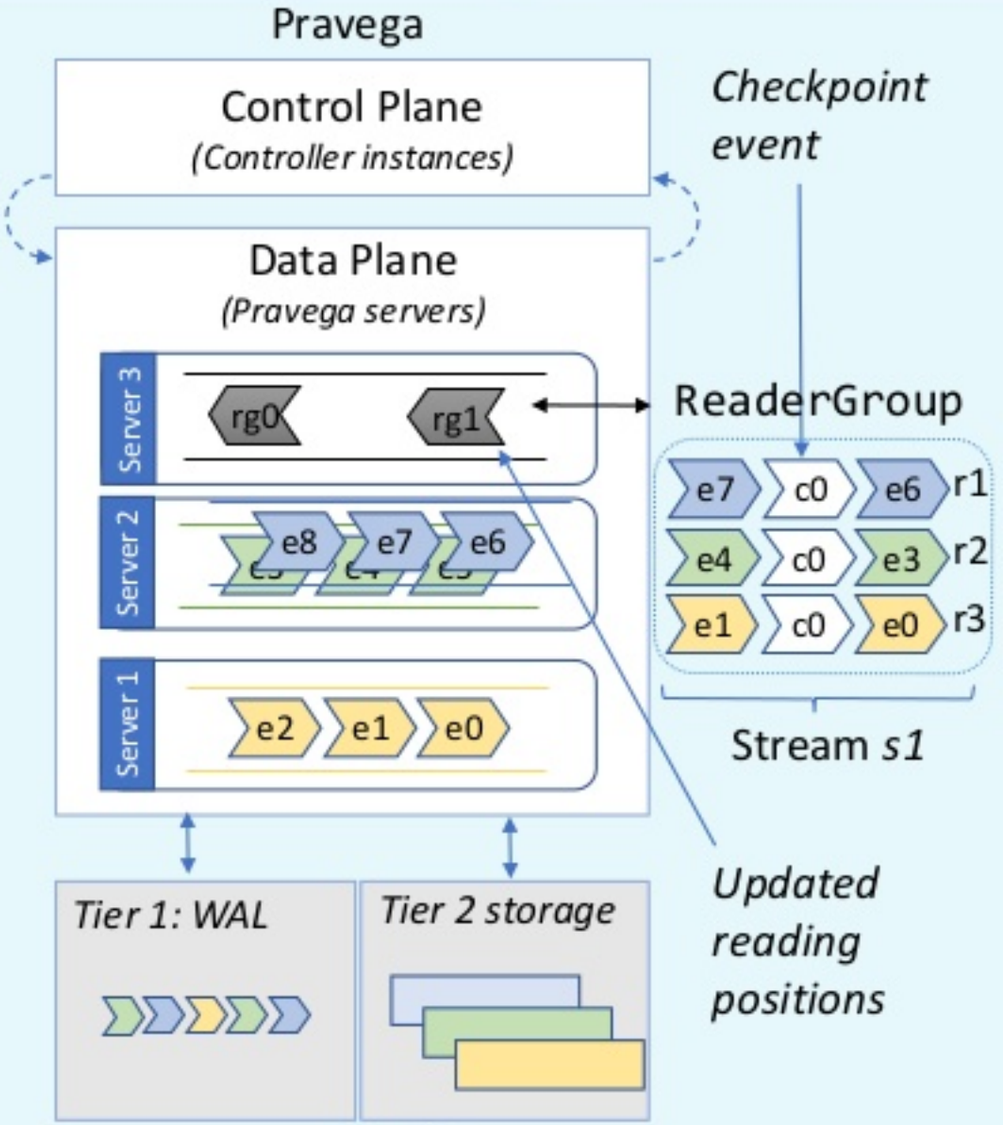
- Shared ReaderGroup **state** among readers:
  - ReaderGroupState contains **reader positions**: <reader: <segment:offset>>
  - *Persistent state* (internally stored in a Pravega Stream).
  - *Synchronization:* Optimistic concurrency via conditional writes.

- The ReaderGroup state is **updated** by:
  - *Change in the group* (e.g., closing a reader, finished segment).
  - Pravega Checkpoints.



Pravega Checkpoints

- **Checkpoint**: Special event that signals all readers within a ReaderGroup to persist their state.
- Checkpoints can be **invoked** via ReaderGroup:
  - `CompletableFuture<Checkpoint> initiateCheckpoint(id, executor)`
- **Automatic checkpointing** on group creation:
  - Set in ReaderGroup configuration (ReaderGroupConfig).
  - `automaticCheckpointIntervalMillis(millis);`



Benefits of Pravega as a Storage System

- **Unlimited retention**:
  - Stream segments can be stored in Tier 2 forever.
  - Clients are agnostic of segments: They only work with Streams.

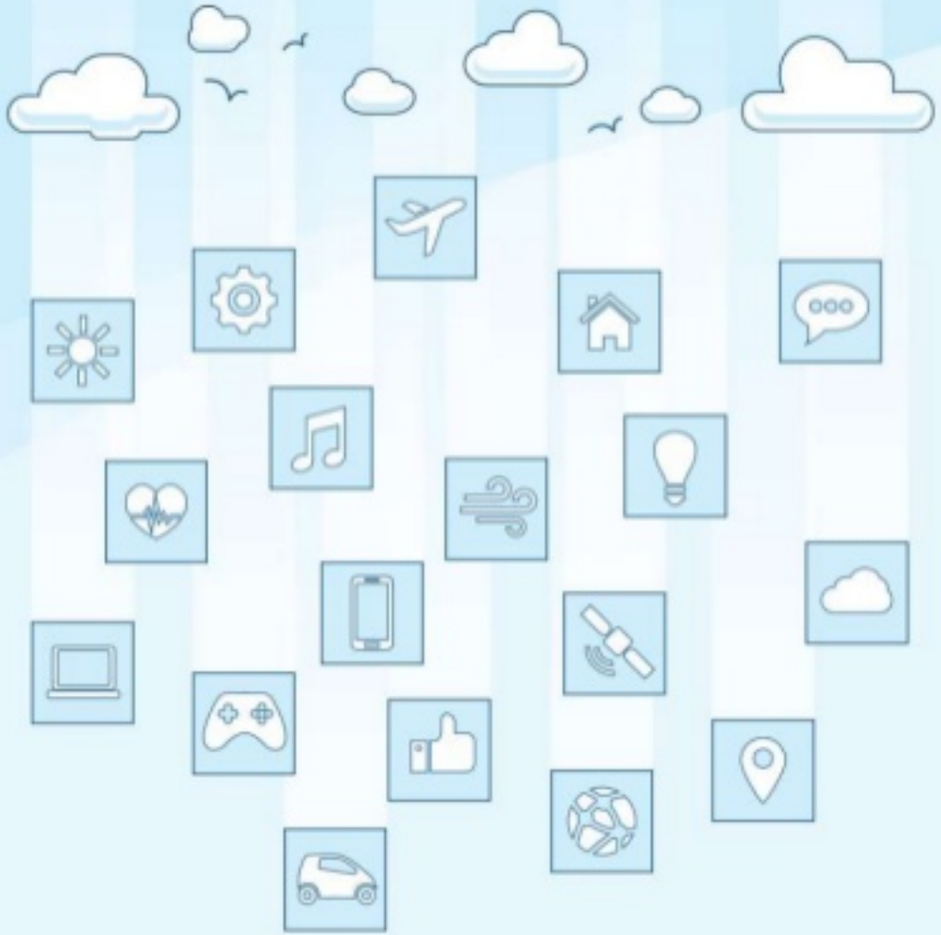
- **Unified storage primitive**:
  - Sweet spot in latency vs throughput trade-off: copes with both real-time/batch analytics.
  - No need to maintain dual analytics pipelines.

- **Data durability:**
  - Data is durably stored in both tiers.

Benefits of Pravega as a Storage System

- **Parallelism**:
  - Multiple readers and writers may read/write on the same stream in parallel.

- **Guarantees for data processing**:
  - Exactly-once semantics.
  - Consistent event ordering (enforced via writer routing key).



Pravega: Simplifying Processing of Unbounded Data Streams

DELL EMC

http://pravega.io

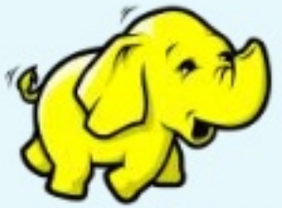
Pravega Connectors

- Pravega has been designed to **be a storage substrate for data processing engines**.

- Connectors for Pravega:
  - **Flink Connector**.
  - Hadoop Connector.
  - Logstash (experimental stage).

- Many proposals from the community (Spark, NodeJS, …)

Pravega Flink Connector
https://github.com/pravega/flink-connectors



Pravega Hadoop Connector
https://github.com/pravega/hadoop-connectors

Flink Data Access: Streaming & Batch

- Pravega allows *stream-based* (ordered) and *parallel* (unordered) access to events.

- Connector-level implementation (Flink):
  - `FlinkPravegaReader` exploits low latency tail reads in Pravega.
  - `FlinkPravegaInputFormat` exploits high throughput parallel catch-up reads (Tier 2).

Stream-based data access to Pravega (Flink connector)

```
SourceFunction<Double> myReader =
    FlinkPravegaReader.<Double>builder()
        .withPravegaConfig(pravegaConfig)
        .forStream(pravegaStreamName)
        .withDeserializationSchema(deserializationSchema))
        .build();
```

Batch data access to Pravega (Flink connector)

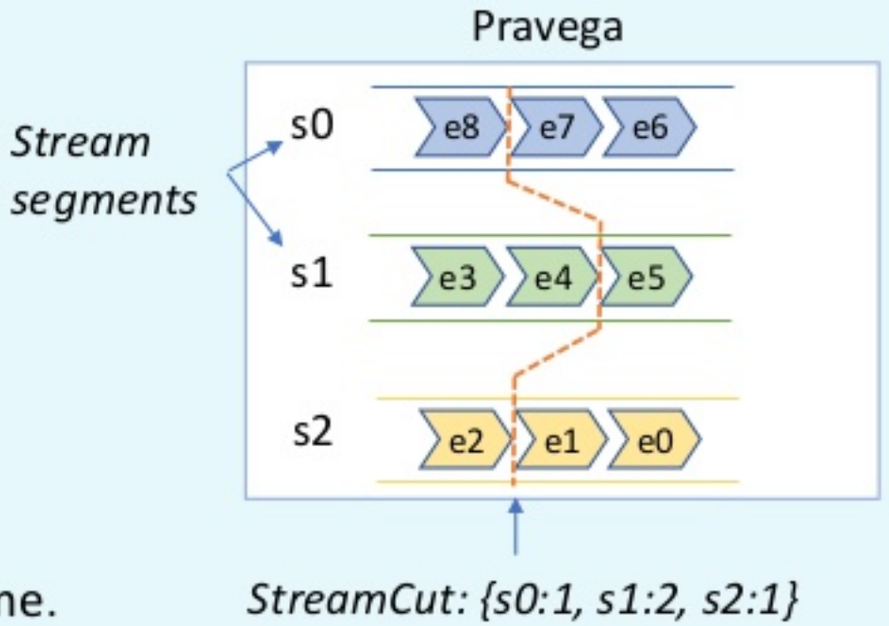
```
DataSet<Double> myData = env.createInput(
    FlinkPravegaInputFormat.<Double>builder()
        .forStream(pravegaStreamName)
        .withPravegaConfig(pravegaConfig)
        .withDeserializationSchema(deserializationSchema)
        .build(),
     BasicTypeInfo.DOUBLE_TYPE_INFO);
```

References in a Stream: The StreamCut

- *Need for a simple* abstraction that allows us to:
  - Bookmark/reference a specific point in the Stream.
  - Define slices in a Stream.

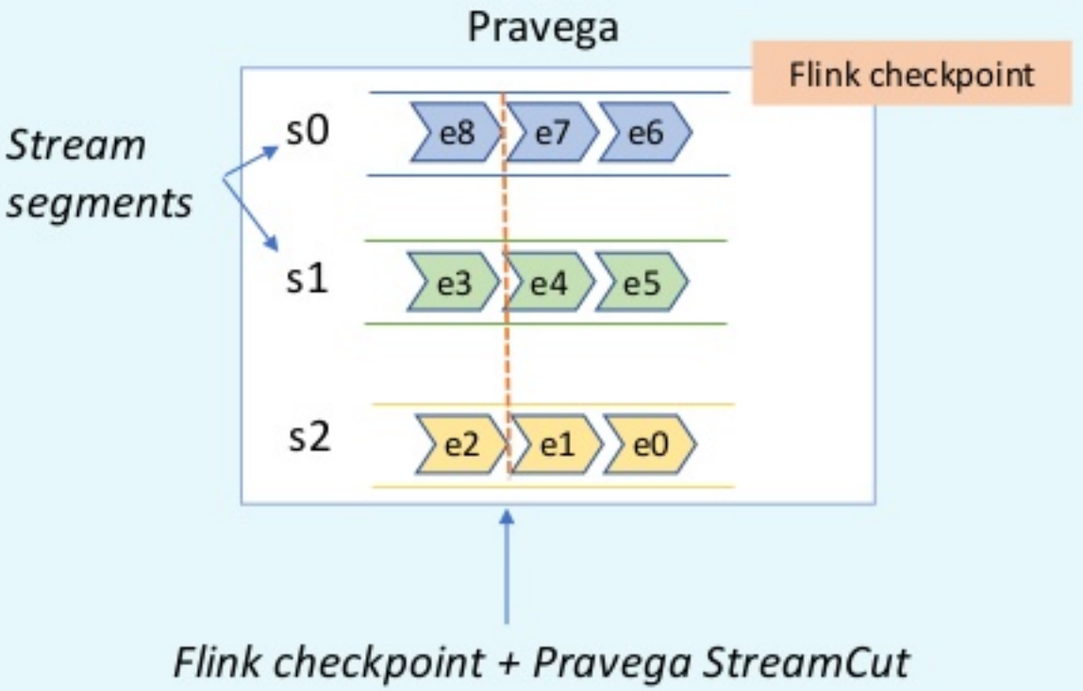
- **StreamCut**: Compact data structure that represents a *consistent event boundary* in a Stream.
  - Formed by reading positions in the ReaderGroup.
  - Encompasses all the segments at a given point in time.
  - Incomplete events are not allowed.

- Works for both *open and sealed segments*.



StreamCuts in Flink: End-to-end Checkpointing

- Flink connector for Pravega *is compatible with Flink checkpoints and savepoints*.
  - Implements `ExternallyInducedSource<T, Checkpoint>`
- Flink checkpoints trigger Pravega `Checkpoints`:
  - Readers in the ReaderGroup persist their positions.
- *At-least-once* read guarantee: Readers recover from a failure by *rewinding to the checkpointed position* in the Stream.
  - The checkpointed position is represented via a `StreamCut`.
- Pravega *is an exactly-once* storage sink:
  - Flink connector exploits Pravega `Transactions`.



StreamCuts in Flink: Bounded Processing

- `StreamCuts` define *read boundaries* in a `Stream`.

- Using `StreamCuts` in the Flink connector we can:
  1. Read from a StreamCut up to the tail of the Stream.
  2. Read from the head of the Stream up to a StreamCut.
  3. Read a Stream "slice" between two StreamCuts.

- Special `StreamCut`: `StreamCut.UNBOUNDED`.
  - Represents either the head or the tail of the Stream.

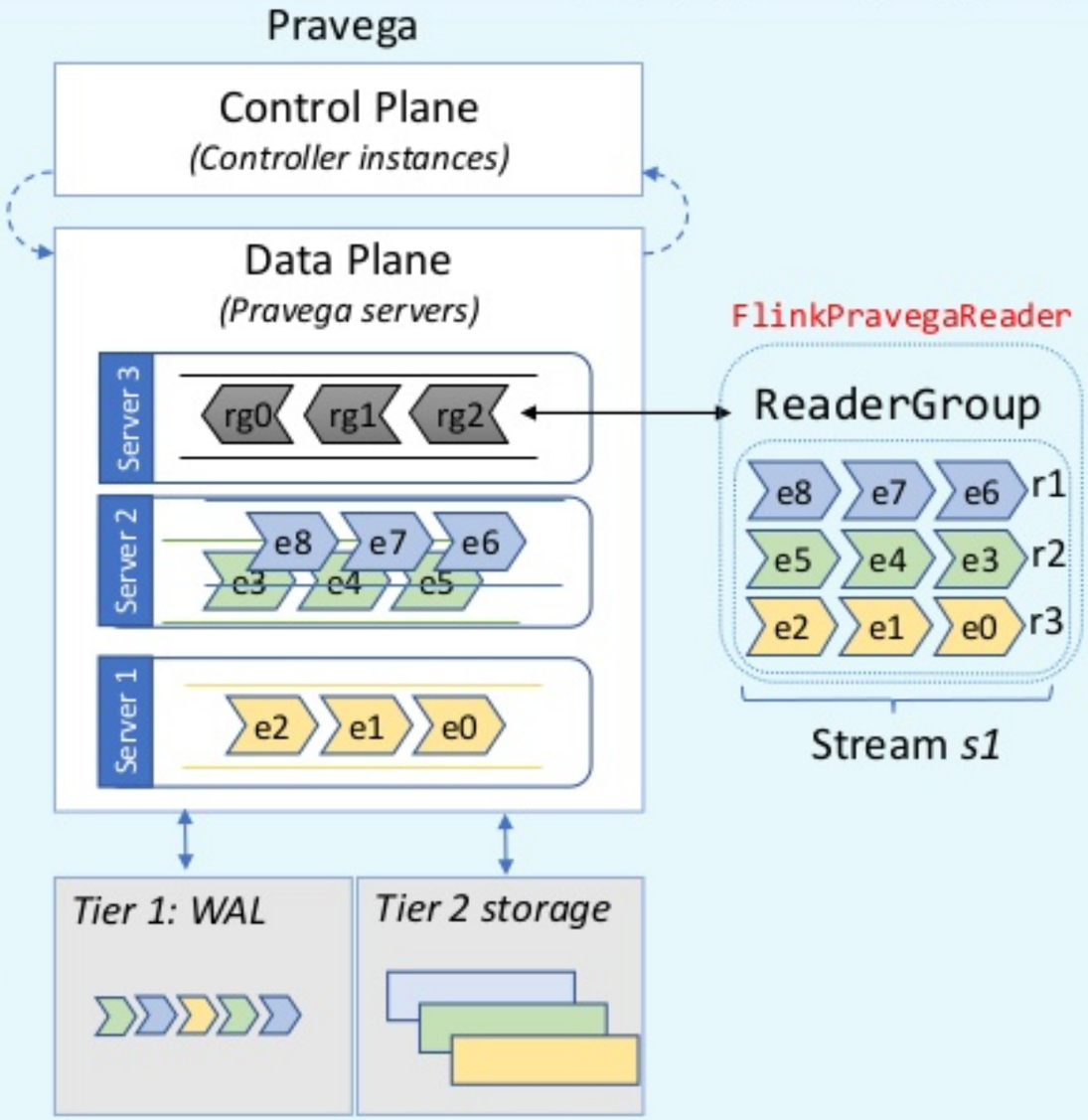
```
FlinkPravegaReader.builder()
 .forStream(pravegaStreamName, startStreamCut)
 …
```

```
FlinkPravegaReader.builder()
 .forStream(streamName, StreamCut.UNBOUNDED, endStreamCut)
 …
```

```
FlinkPravegaReader.builder()
 .forStream(streamName, startStreamCut, endStreamCut)
 …
```

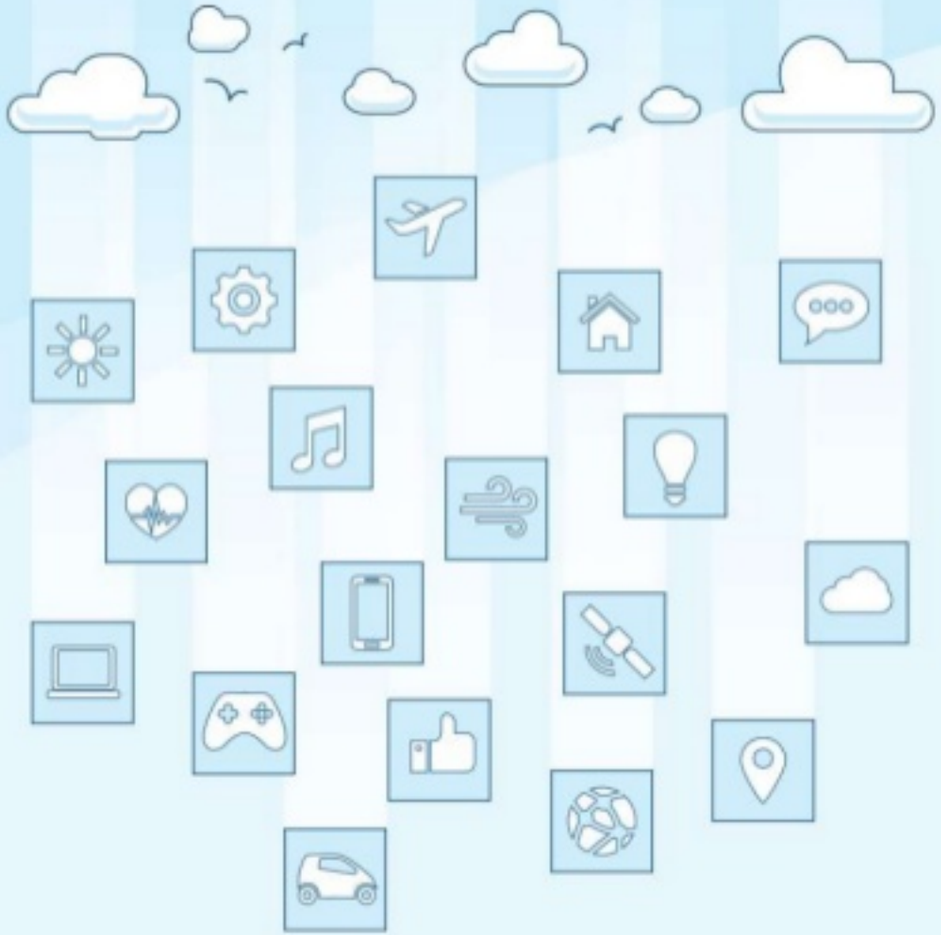
Getting StreamCuts

- Flink connector internally uses a `ReaderGroup`.
- Users can specify the "name" of this `ReaderGroup`:
  - Read access to `ReaderGroup` state used by the connector.
  - `readerGroupManager.getReaderGroup("name")`.
- Get StreamCuts from the ReaderGroup :
  - ReaderGroup API: `Map<Stream, StreamCut> getStreamCuts();`
- When `StreamCuts` are updated in `FlinkPravegaReader`?
  - Upon *special events* (e.g., reader offline, finished segment).
  - Upon a *Flink Checkpoint*: They are associated with Pravega `Checkpoints`.



StreamCuts with Flink: Wrap up

- **`StreamCut` abstraction**:
  - Pravega abstraction to define a *global, consistent position in the Stream*.
- **`StreamCuts` for end-to-end checkpoint support**:
  - `StreamCuts` are used as Stream references to *rewind readers* to a Flink checkpoint for recovery.
- **Bounded processing in Flink with `StreamCuts`**:
  - We can use `StreamCuts` with the Pravega Flink connector to *process slices of a Stream*.
- **Getting `StreamCuts` in our applications:**
  - We can access to the *most updated* `StreamCut` in a `ReaderGroup` used by Flink connector.

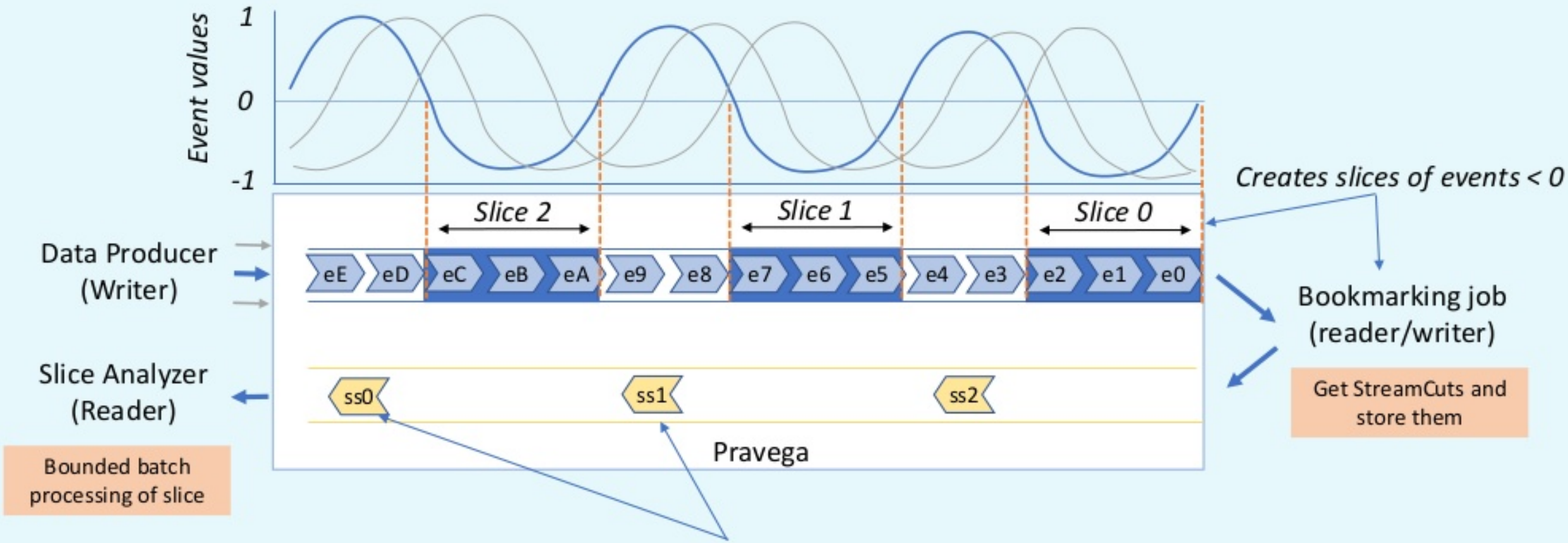


StreamCuts in Action: Sample Application

Objectives of this Example

- Write/read data in **parallel with consistent order**.
- **Bookmarking**: Get `StreamCuts` while reading from a Pravega Stream.
- **Slicing**: Store pairs of `StreamCuts` containing “events of interest”.
- **Replaying**: Bounded processing of data events based on `StreamCut` pairs.

Our Sample App



Code: https://github.com/pravega/pravega-samples

Bookmarking Job: Set Up In/Out Streams

- Read sensor events *in order*.
- We set a known READER_GROUP_NAME:
  - Easy access to ReaderGroup state.
  - Allows us to access the StreamCuts.
- Create a sink to store SensorStreamSlices:
  - SensorStreamSlice: StremCut pair + sensorId.
  - Stream slices where events for a sensor are < 0.
- Set up the processing pipeline:
  - Flink keyed stream by sensorId.

Reader for sensor data

```
SourceFunction<Tuple2<Integer, Double>> reader =
  FlinkPravegaReader.<Tuple2<Integer, Double>>builder()
    .withPravegaConfig(pravegaConfig)
    .forStream(sensorEvents)
    .withReaderGroupName(READER_GROUP_NAME)
    .withDeserializationSchema(new Tuple2DeserializationSchema())
    .build();
```

Write results: stream slices

```
SinkFunction<SensorStreamSlice> writer =
  FlinkPravegaWriter.<SensorStreamSlice>builder()
    .withPravegaConfig(pravegaConfig)
    .forStream(streamCutsStream)
    .withSerializationSchema(new SensorStreamSliceSerializer()))
    .withEventRouter(new EventRouter())
    .build();
```

Processing pipeline

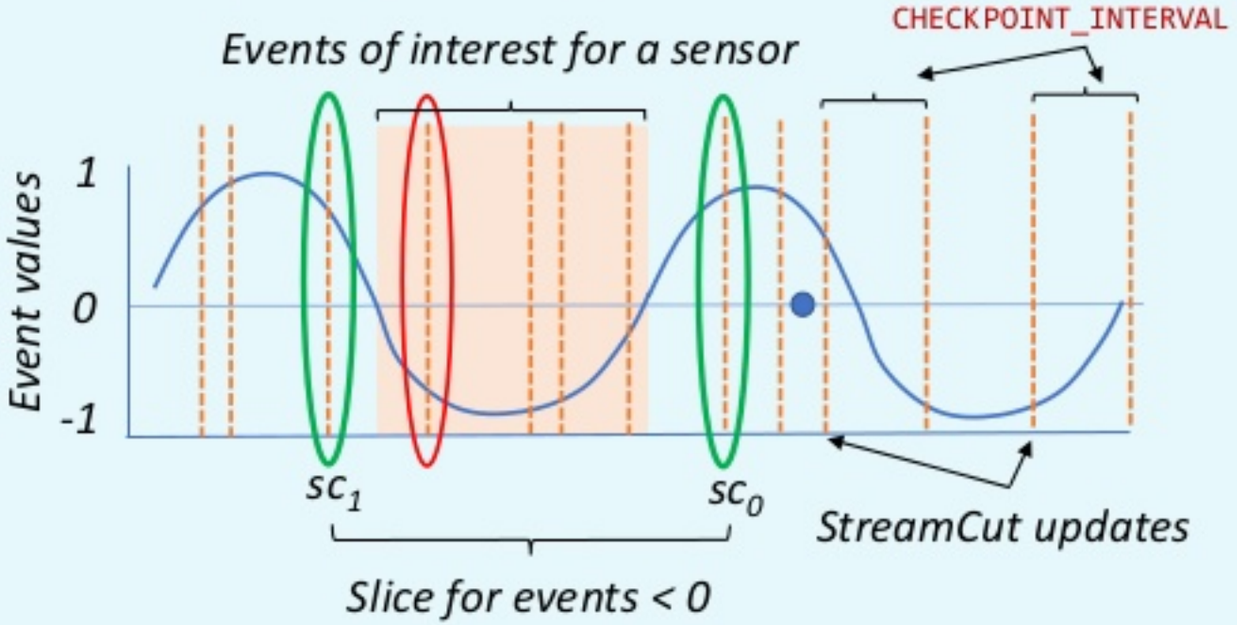
```
DataStreamSink<SensorStreamSlice> sliceSink = env.addSource(reader)
    .setParallelism(numSensors)
    .keyBy(0)
    .process(new Bookmarker(pravegaControllerURI))
    .addSink(writer);
```

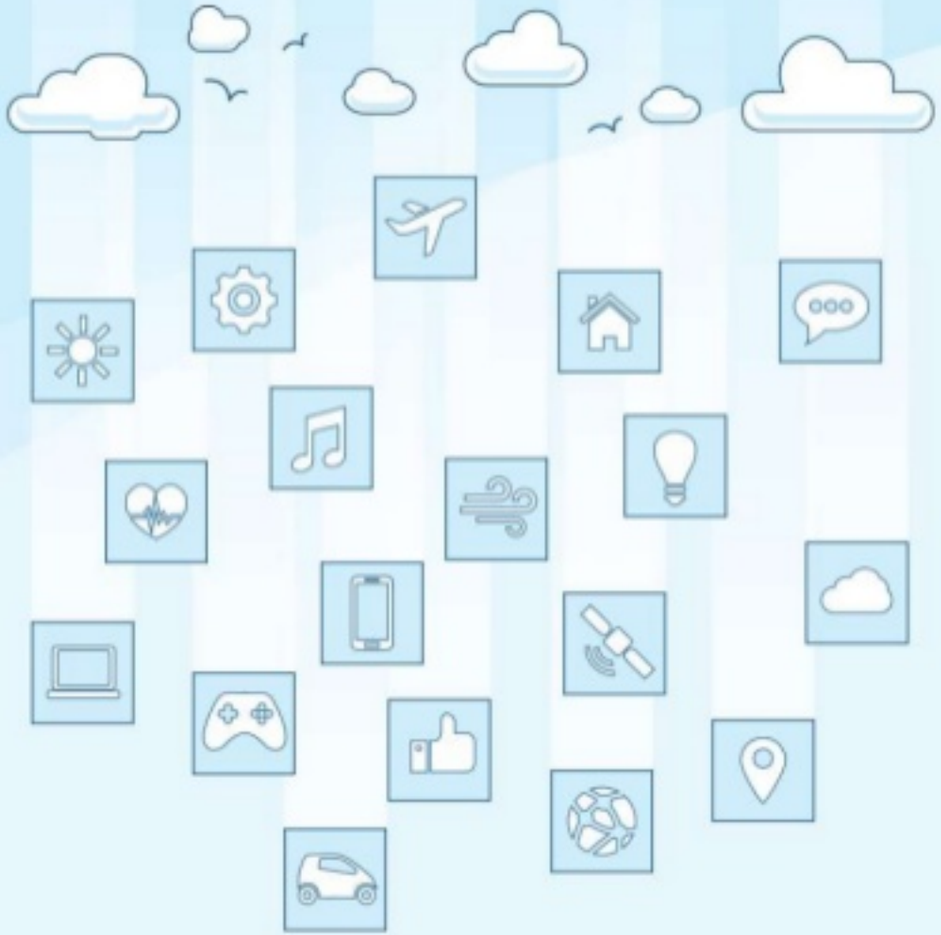
Bookmarking Job: Creating Stream Slices

- The Flink checkpoint interval sets the max `StreamCut` update interval:
  - *Precision granularity* to create Stream slices.
- First event < 0 seen for a `sensorId`:
  - Persistently store the current `StreamCut`.
- If events > 0, get current `StreamCut`:
  - It may not contain all the events of interest.
- Check for the next `StreamCut` to ensure that the slice contains "at least" all the events of interest.

Set Flink checkpointing interval

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment
    .getExecutionEnvironment()
    .enableCheckpointing(CHECKPOINT_INTERVAL);
```





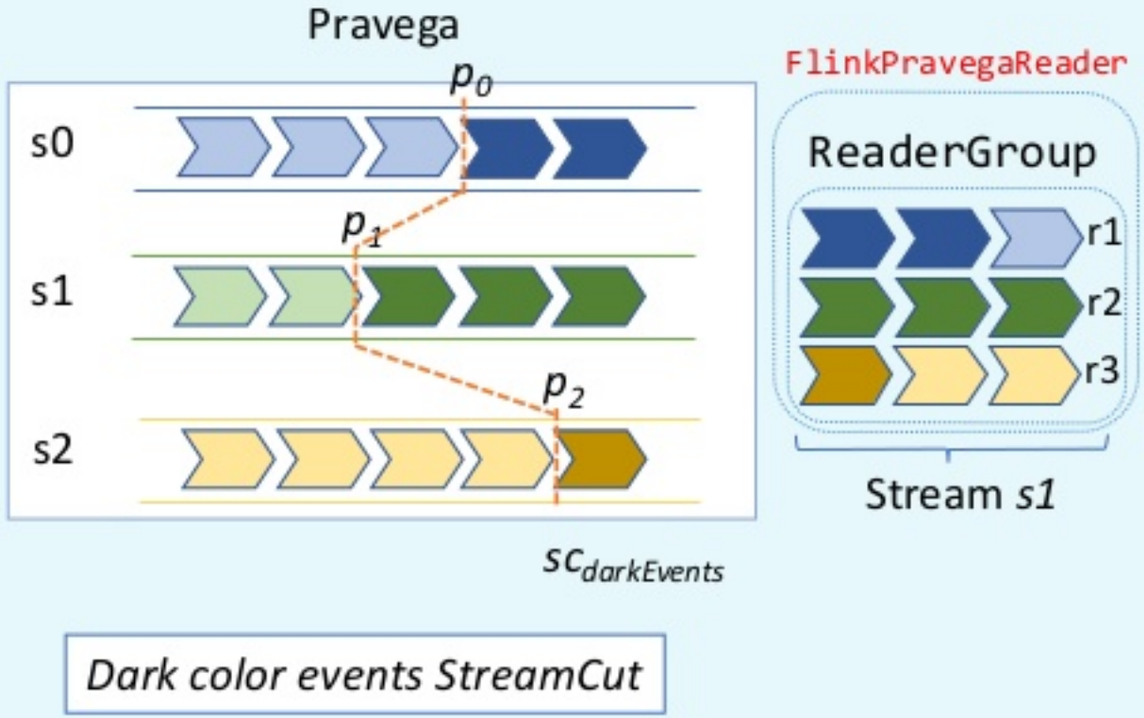
Next Steps & Conclusions

What's Next: Future for StreamCuts

- `StreamCut` API in `ReaderGroup` is a recent addition to Pravega.
- Focus on making it easier for users to work with `StreamCuts`:
  - Important improvements expected in Pravega Beta version (end of 2018)!
- Some interesting points under discussion…

Conditional StreamCuts

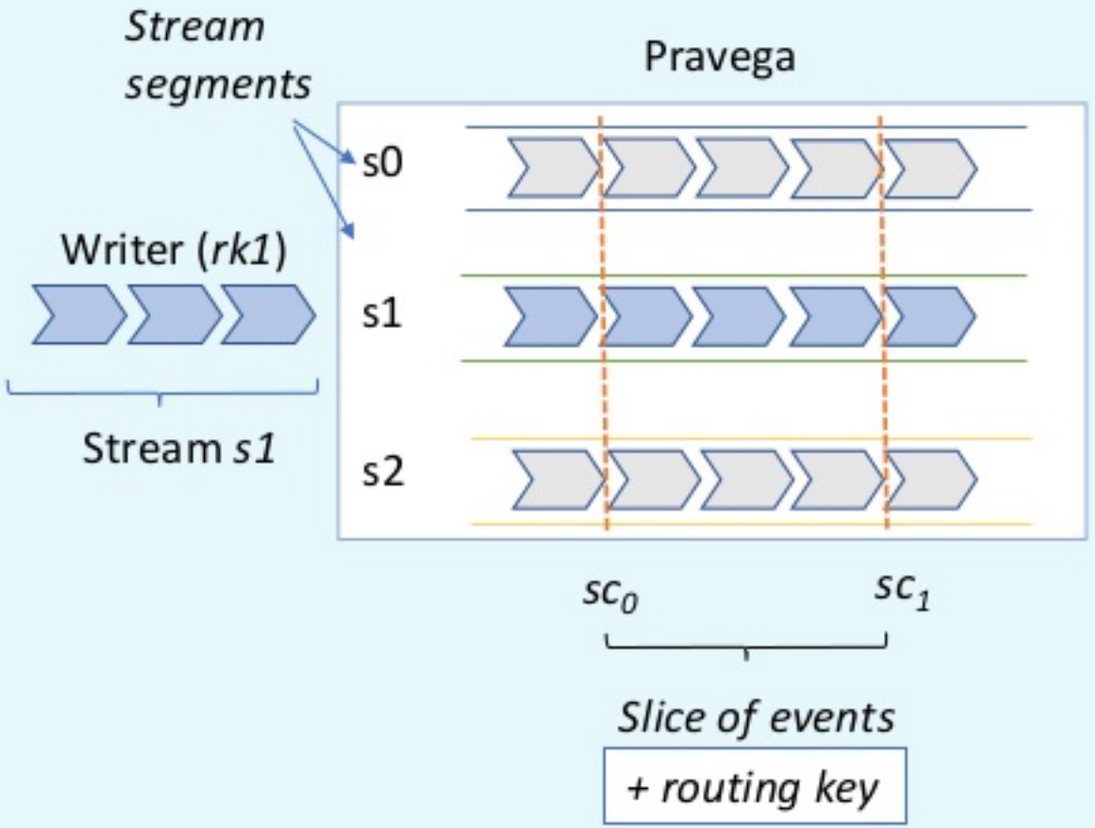
- *Now*: A `StreamCut` represents the most recent persisted reading positions of a set of readers.
- *Idea*: Allow a `ReaderGroup` to create `StreamCuts` based on a condition satisfied by readers in the group.



Dark color events StreamCut

Augmenting Slices with Routing Keys

- *Now*: `StreamCuts` are coarse-grained, global references in a Stream.
- *Idea*: Combine `StreamCuts` and routing keys at the reader side to only get segments for a routing key.



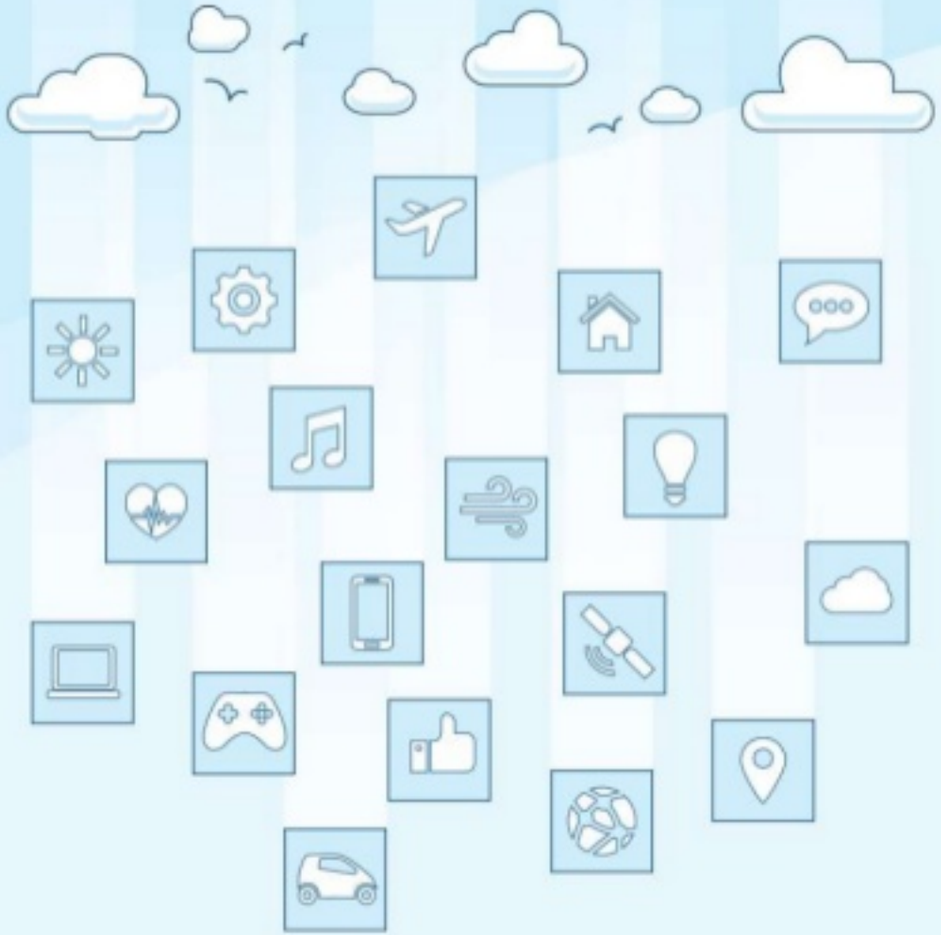
Conclusions

- Pravega is a *storage system for unbounded data streams*:
  - Parallelism, sweet spot in latency vs throughput trade-off.
  - Exactly-once read/write guarantees, consistent event order (based on routing key).

- Pravega as a *storage substrate for data processing engines*:
  - Ready-to-use connectors (e.g., Flink, Hadoop).
  - Supports both stream (event order) and batch (parallel segment reads) processing modes.

- Pravega provides *abstractions to make it easier writing analytics applications*:
  - `StreamCuts` allow us to create references or bookmarks in a Stream.
  - We can execute batch/stream jobs on stream slices defined by pairs of `StreamCuts`.

- Stay tuned for next `StreamCuts` API in Pravega Beta release (end of 2018)!



Thanks for your attention!
Q&A