Pravega: Rethinking storage for streams

Stephan Ewen, *Data Artisans*
Flavio Junqueira, *Pravega*

Flink Forward – Berlin 2017

Outline

- Intro to Pravega
- Flink + Pravega

Pravega

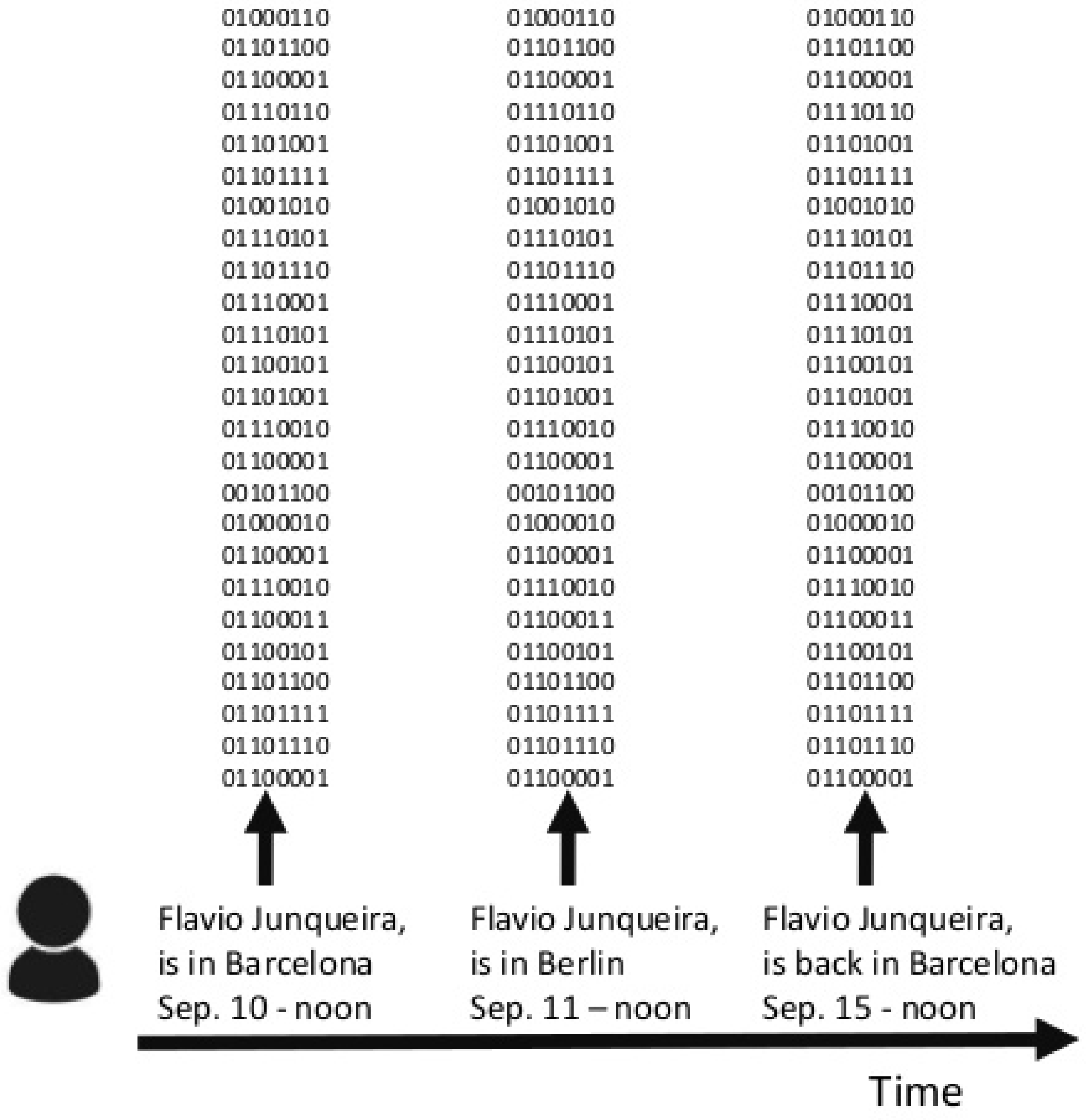
http://pravega.io

Streams

Flavio Junqueira,
Lives in Barcelona →

01000110
01101100
01100001
01110110
01101001
01101111
01001010
01110101
01101110
01110001
01110101
01100101
01101001
01110010
01100001
00101100
01000010
01100001
01110010
01100011
01100101
01101100
01101111
01101110
01100001

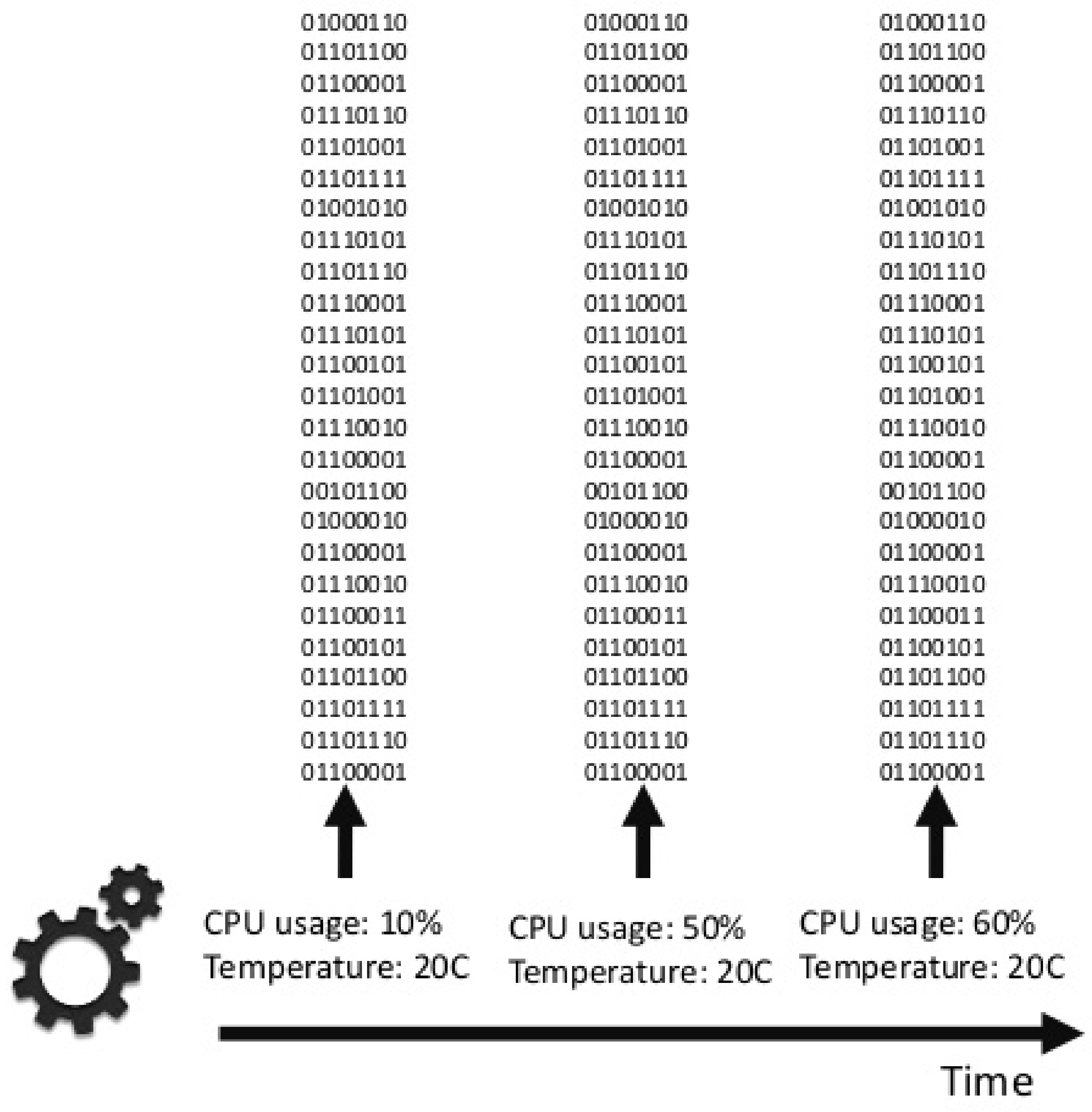
- Process bits
- Store bits
- Transmit bits



```
01000110    01000110    01000110
01101100    01101100    01101100
01100001    01100001    01100001
01110110    01110110    01110110
01101001    01101001    01101001
01101111    01101111    01101111
01001010    01001010    01001010
01110101    01110101    01110101
01101110    01101110    01101110
01110001    01110001    01110001
01110101    01110101    01110101
01100101    01100101    01100101
01101001    01101001    01101001
01110010    01110010    01110010
01100001    01100001    01100001
00101100    00101100    00101100
01000010    01000010    01000010
01100001    01100001    01100001
01110010    01110010    01110010
01100011    01100011    01100011
01100101    01100101    01100101
01101100    01101100    01101100
01101111    01101111    01101111
01101110    01101110    01101110
01100001    01100001    01100001
```

Flavio Junqueira, is in Barcelona Sep. 10 - noon
Flavio Junqueira, is in Berlin Sep. 11 – noon
Flavio Junqueira, is back in Barcelona Sep. 15 - noon
Time

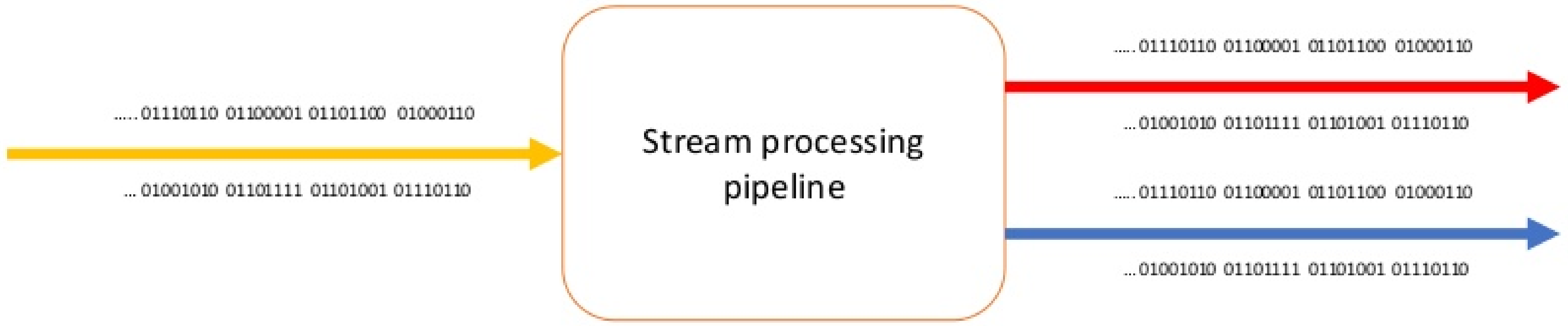
- Order matters
- Correlation between events
- Causality maybe?



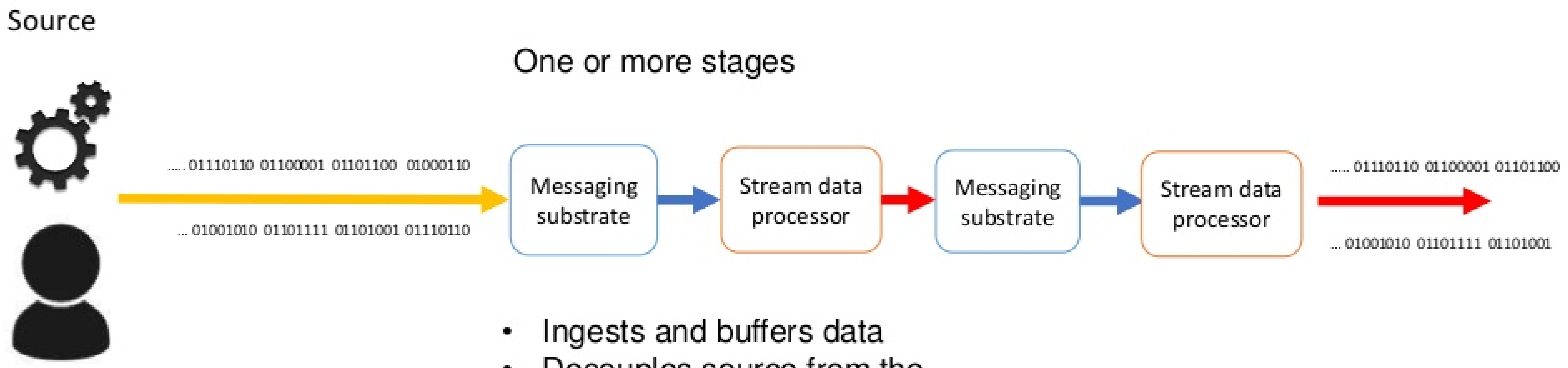
- Number of devices potentially much larger
- Volume per device potentially much higher

Processing data streams

Processing streams



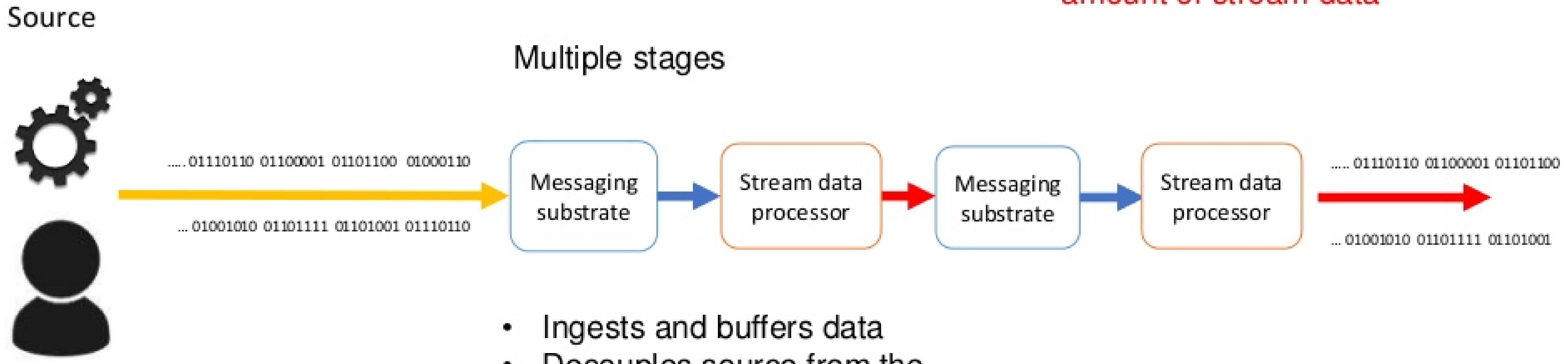
A typical architecture



- Ingests and buffers data
- Decouples source from the engine processing the data

A typical architecture

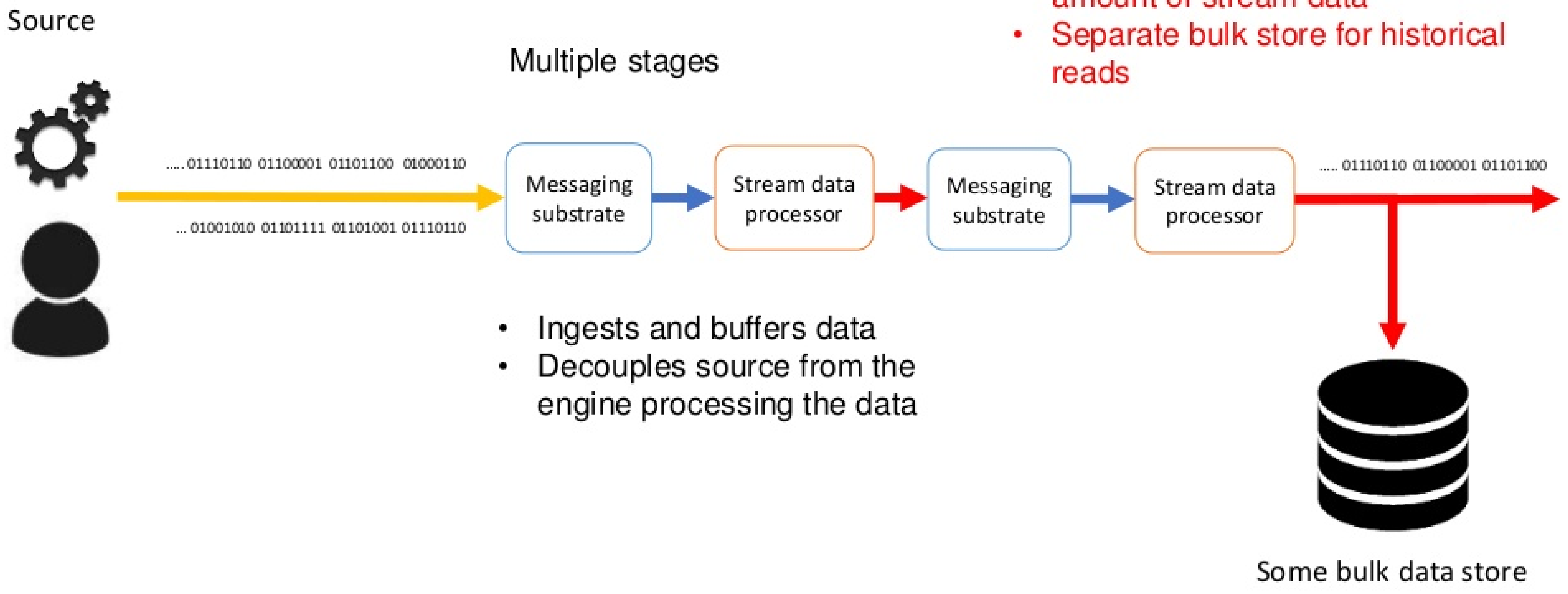
Limitations:
- Data stored temporarily
- Not able to store an unbounded amount of stream data



- Ingests and buffers data
- Decouples source from the engine processing the data

A typical architecture

Limitations:
- Data stored temporarily
- Not able to store an unbounded amount of stream data
- Separate bulk store for historical reads



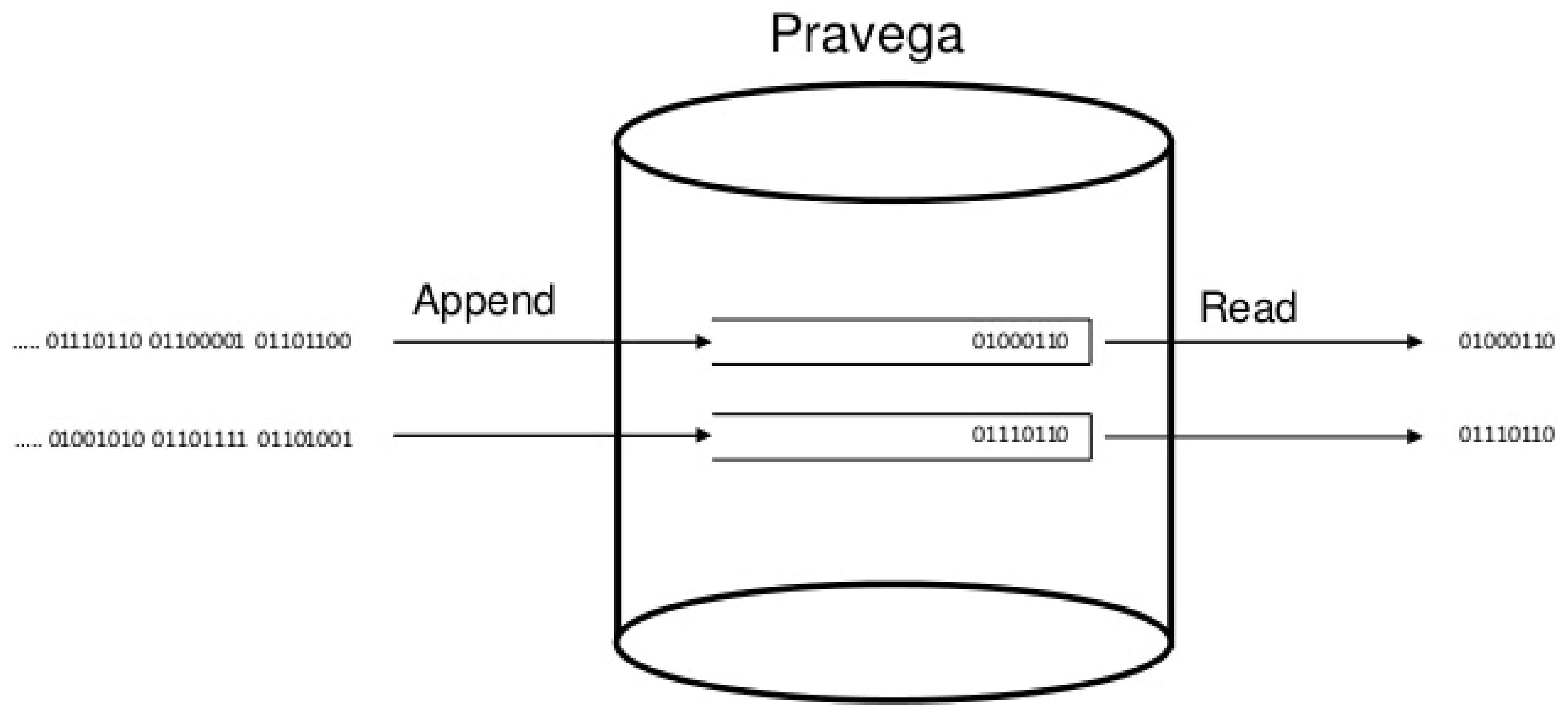
- Ingests and buffers data
- Decouples source from the engine processing the data

Target of Pravega is a stream store able to:

- Store stream data permanently
- Preserve order
- Accommodate unbounded streams

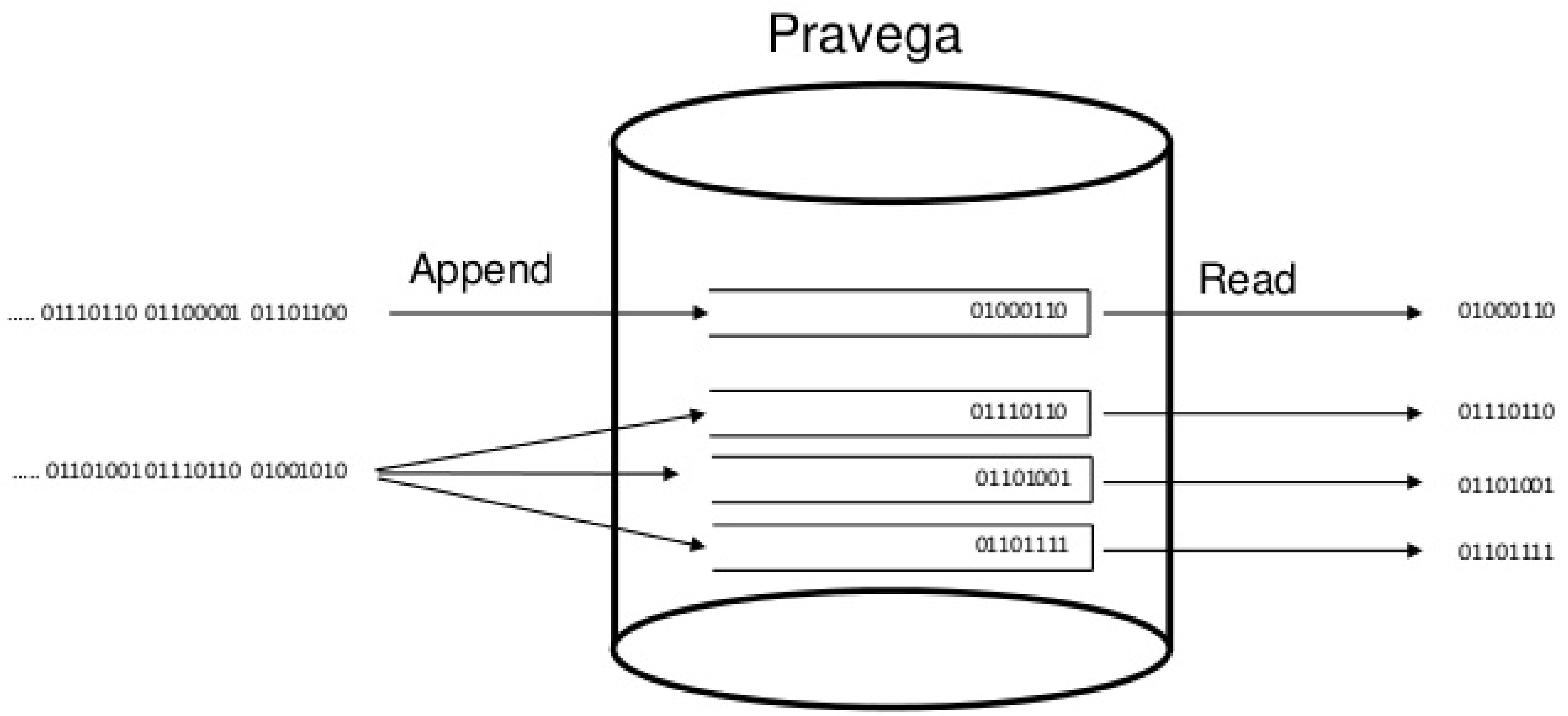
Streams in Pravega

Pravega and Streams

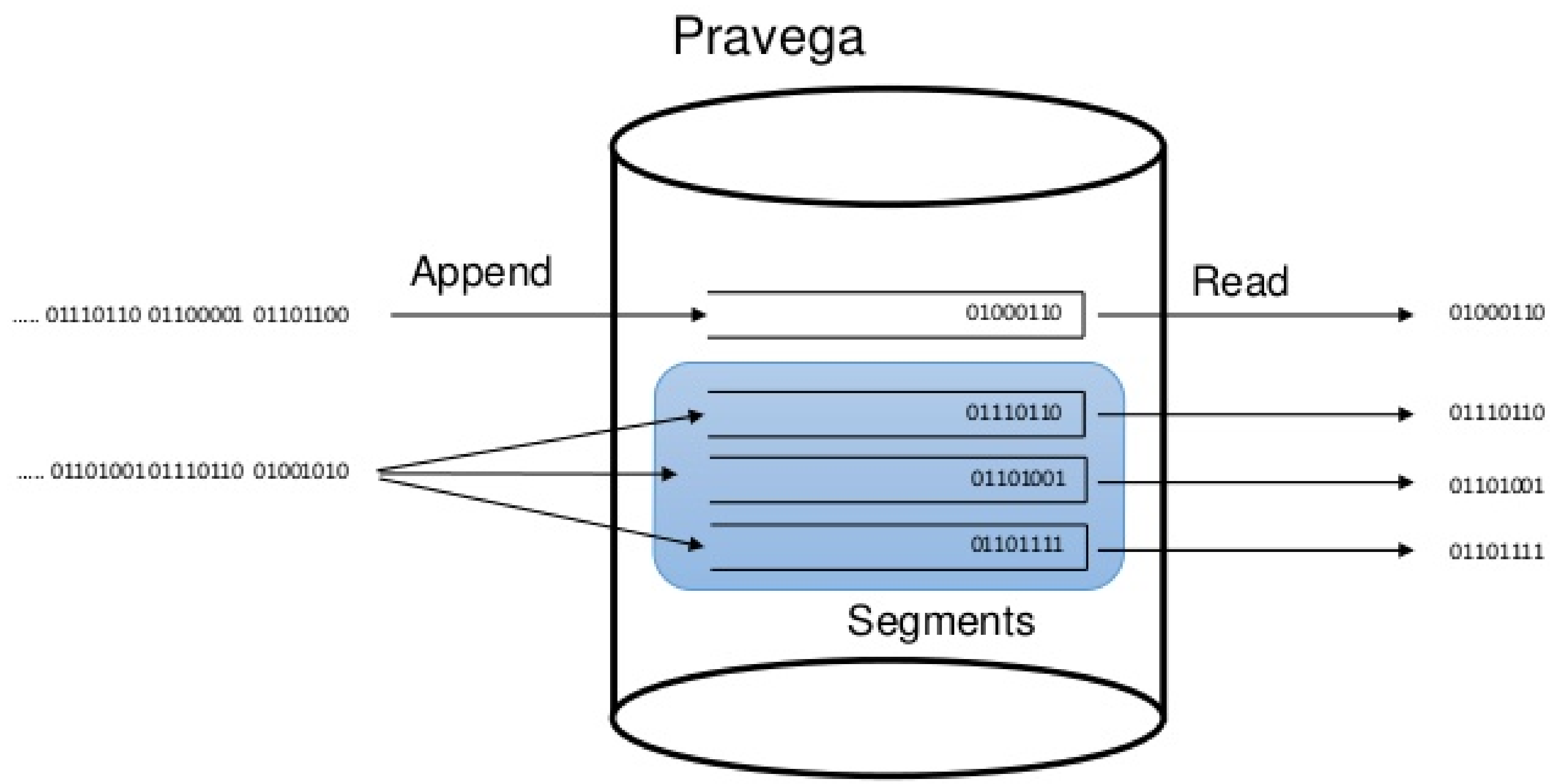


What about parallelism?

Pravega and Streams

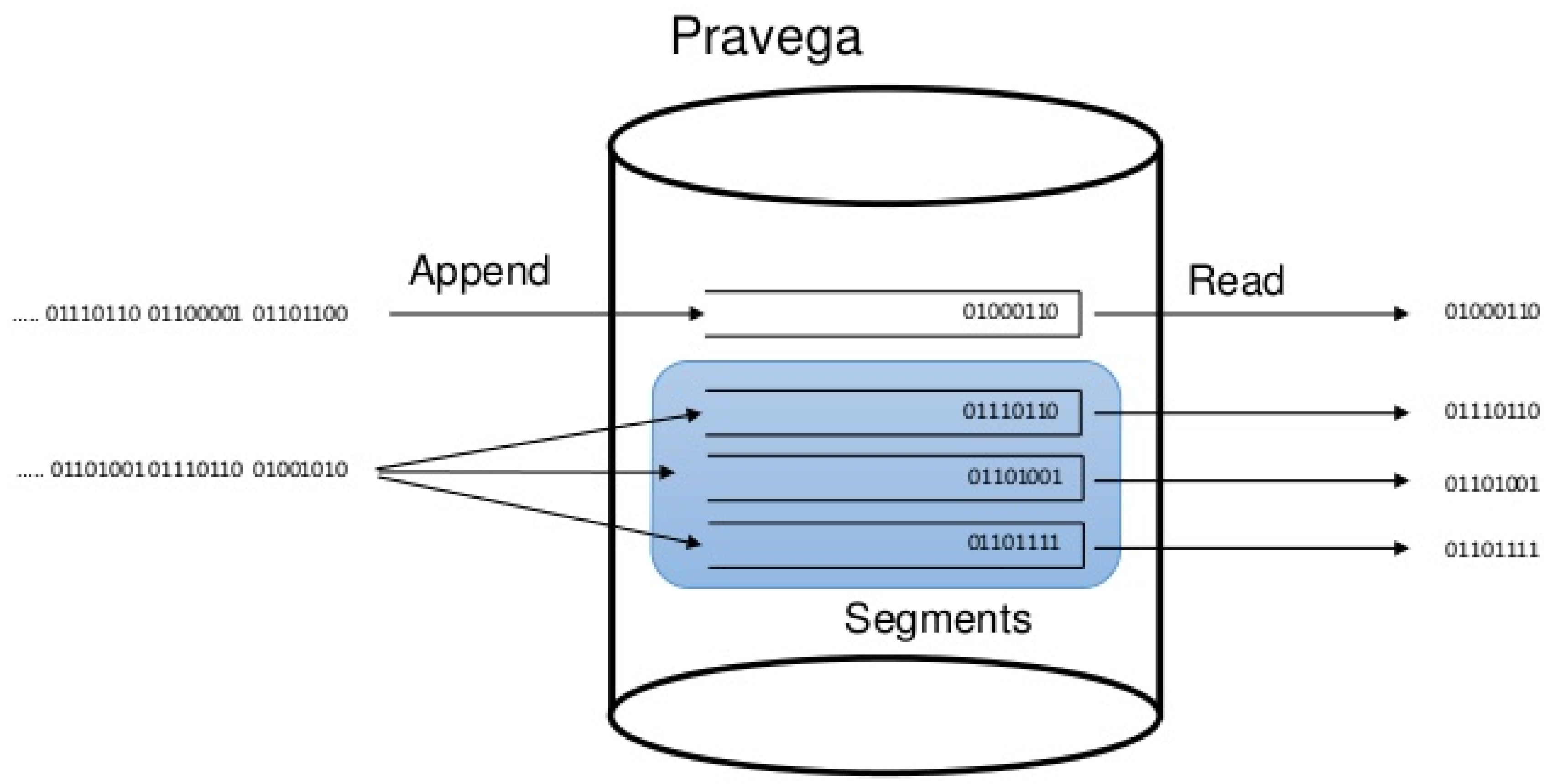


Segments in Pravega



- Segments are sequences of bytes

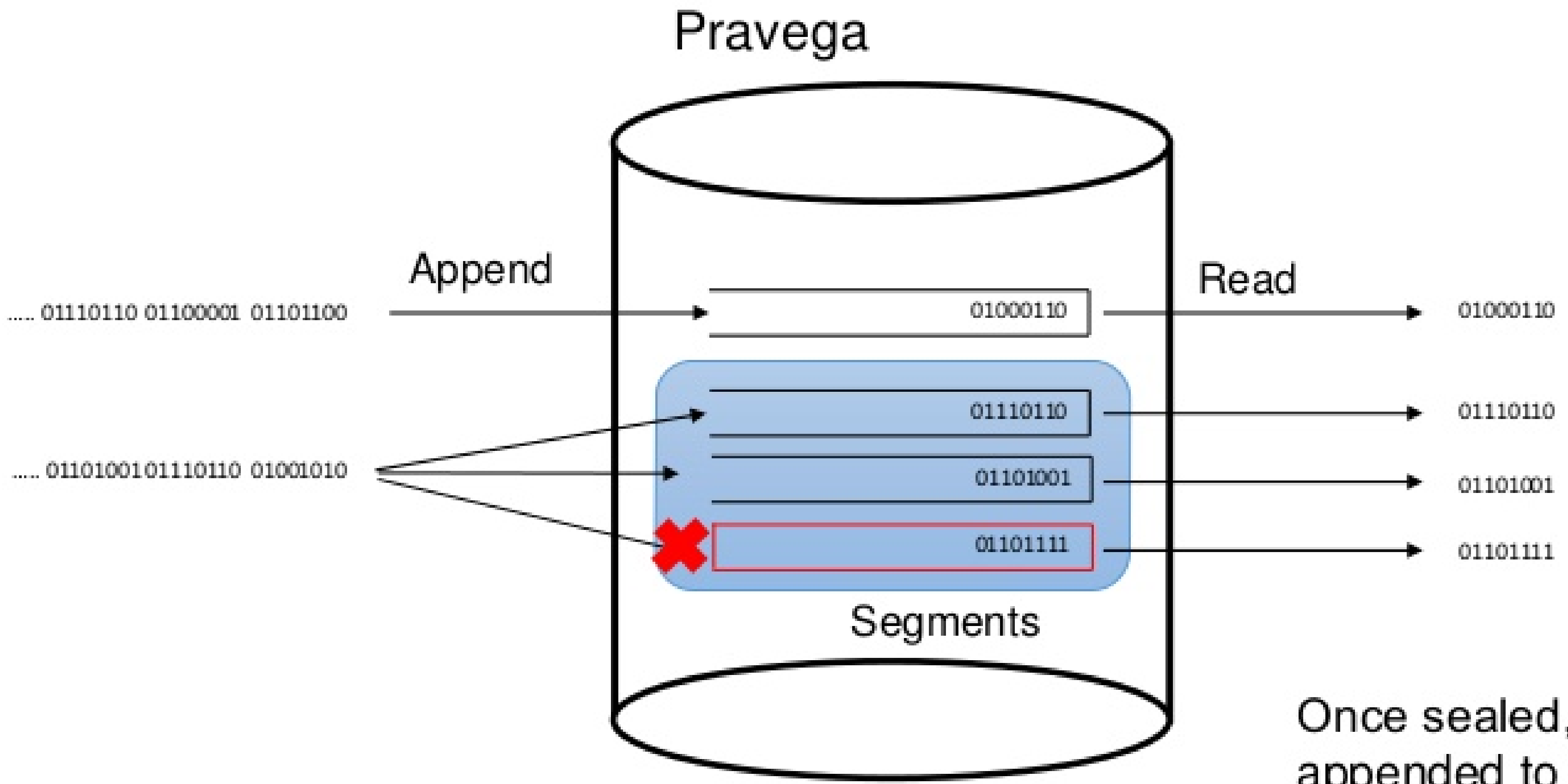
Segments in Pravega



- Segments are sequences of bytes
- Use routing keys to determine segment

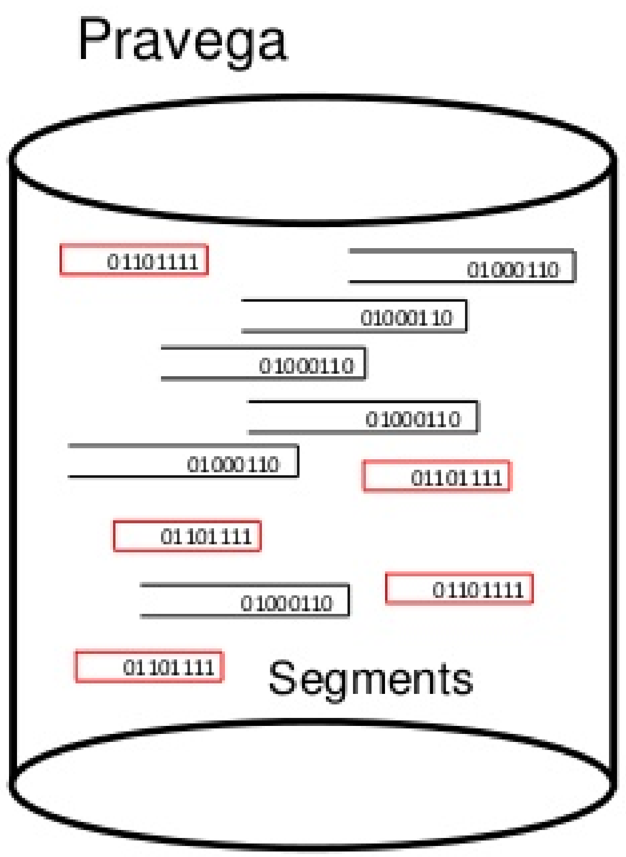
Segments can be sealed

Segments in Pravega



Once sealed, a segment can't be appended to any longer.

Segments in Pravega



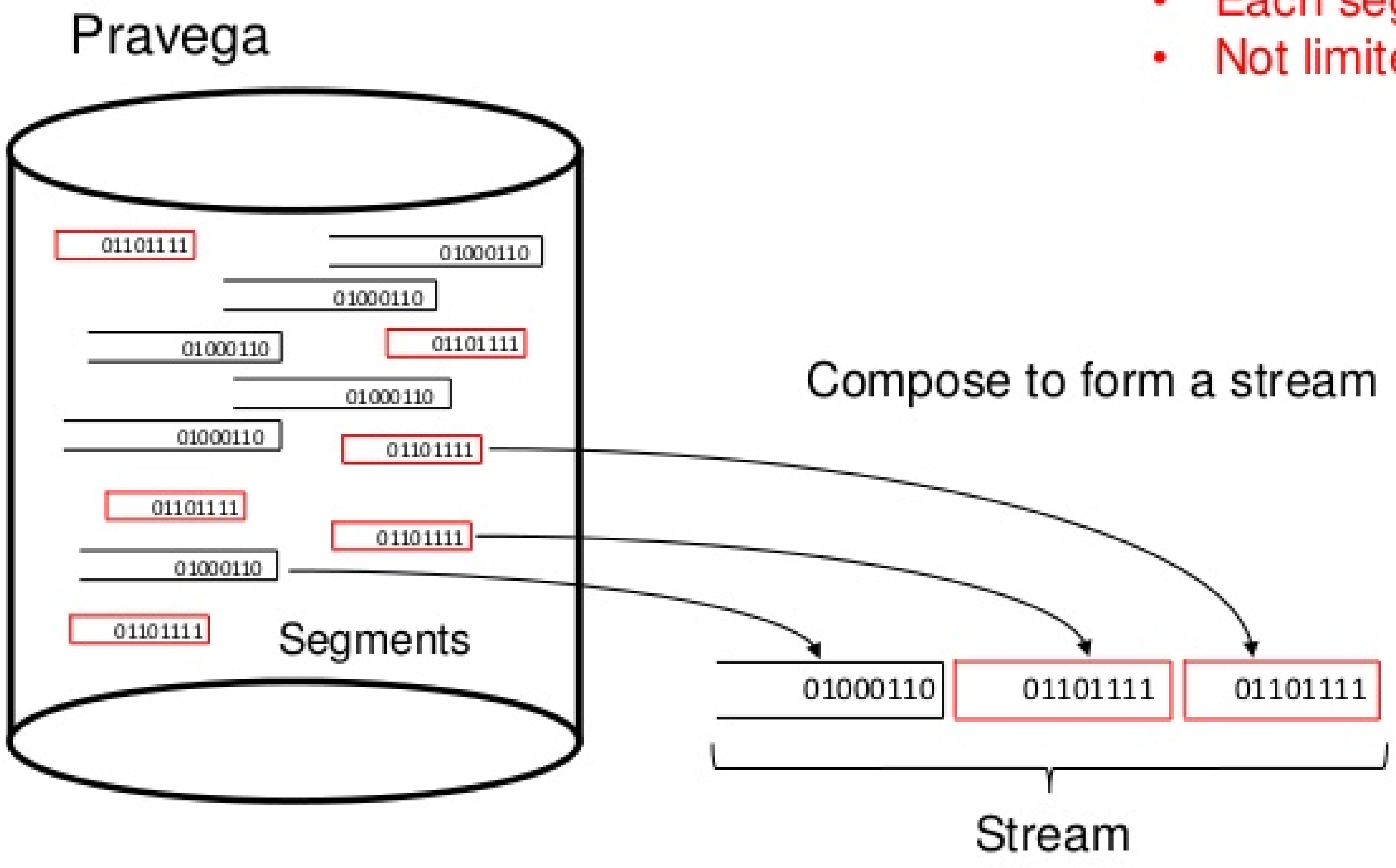
Pravega is primarily:

- A segment store
- Segments sealed or open

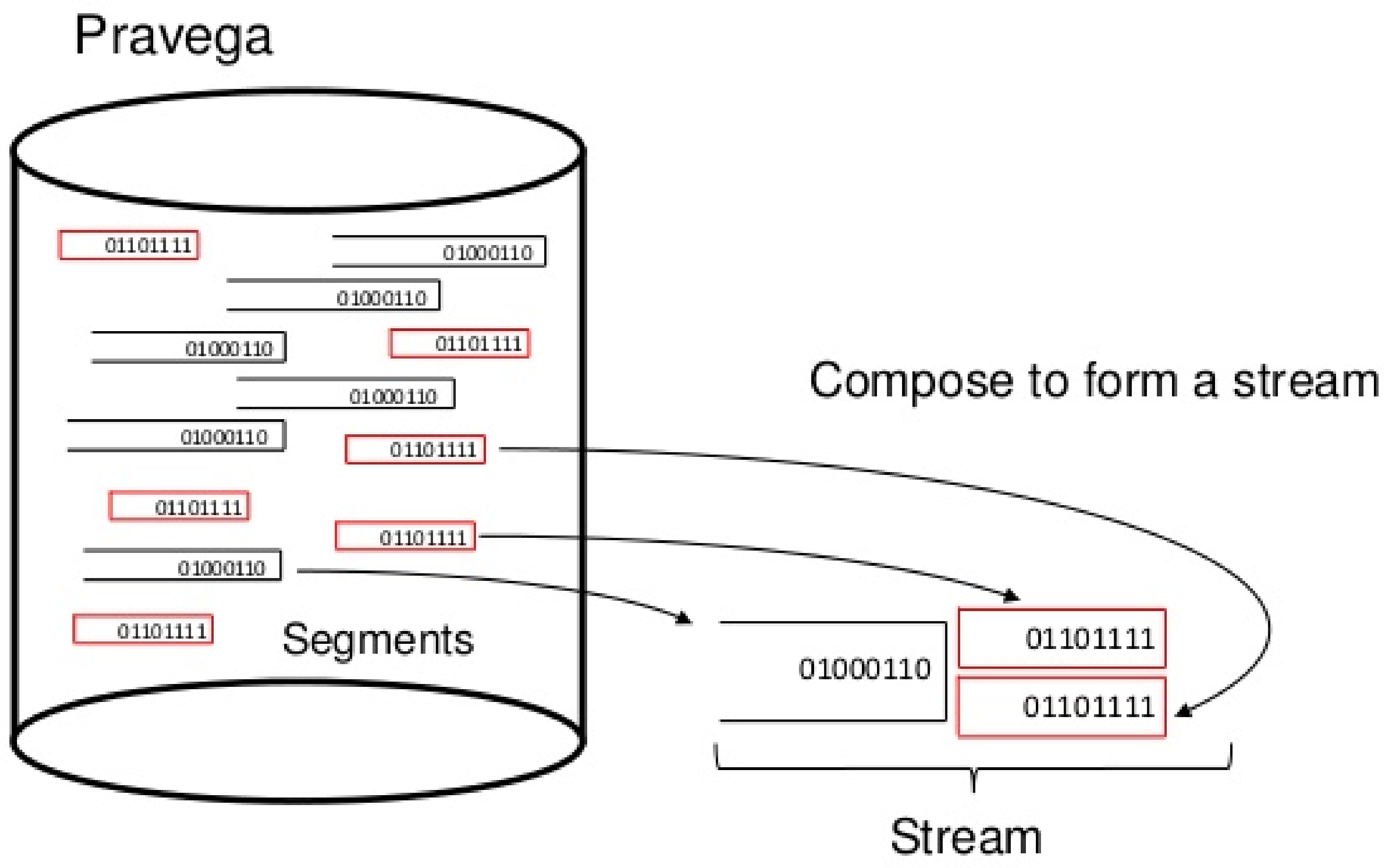
How is sealing segments useful?

Segments in Pravega

- Each segment can live in a different server
- Not limited to the capacity of a single server

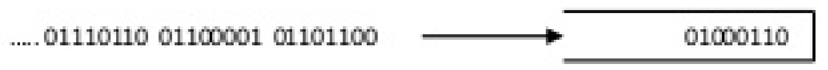


Segments in Pravega



Some useful ways to compose segments

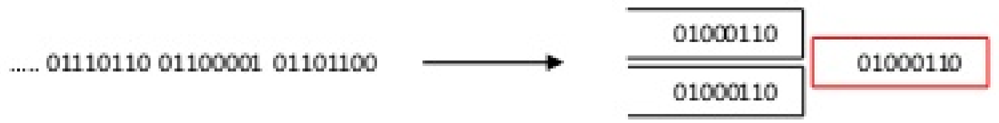
Scaling a stream



Stream has one segment

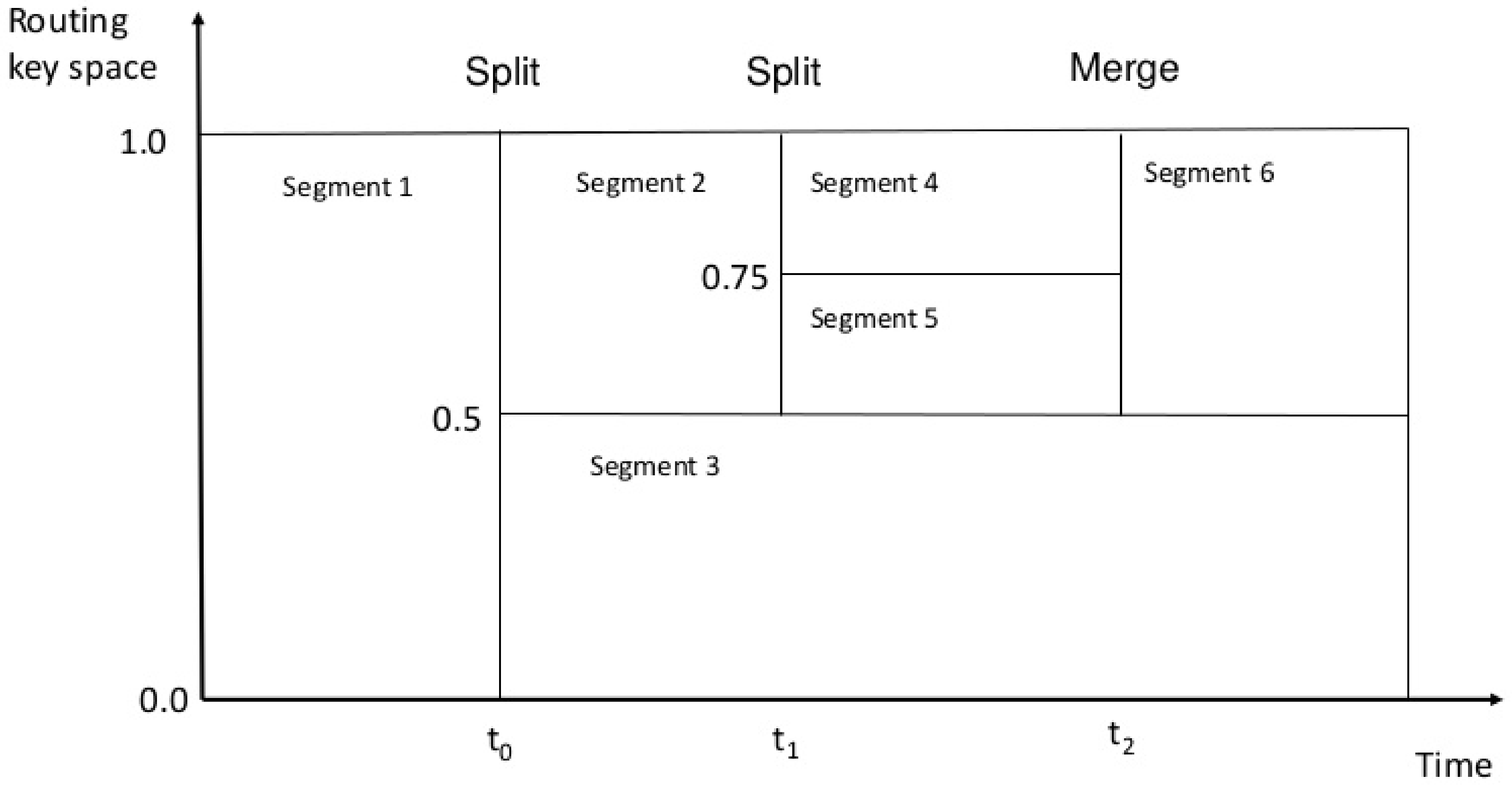
1

- Say input load has increased
- Need more parallelism

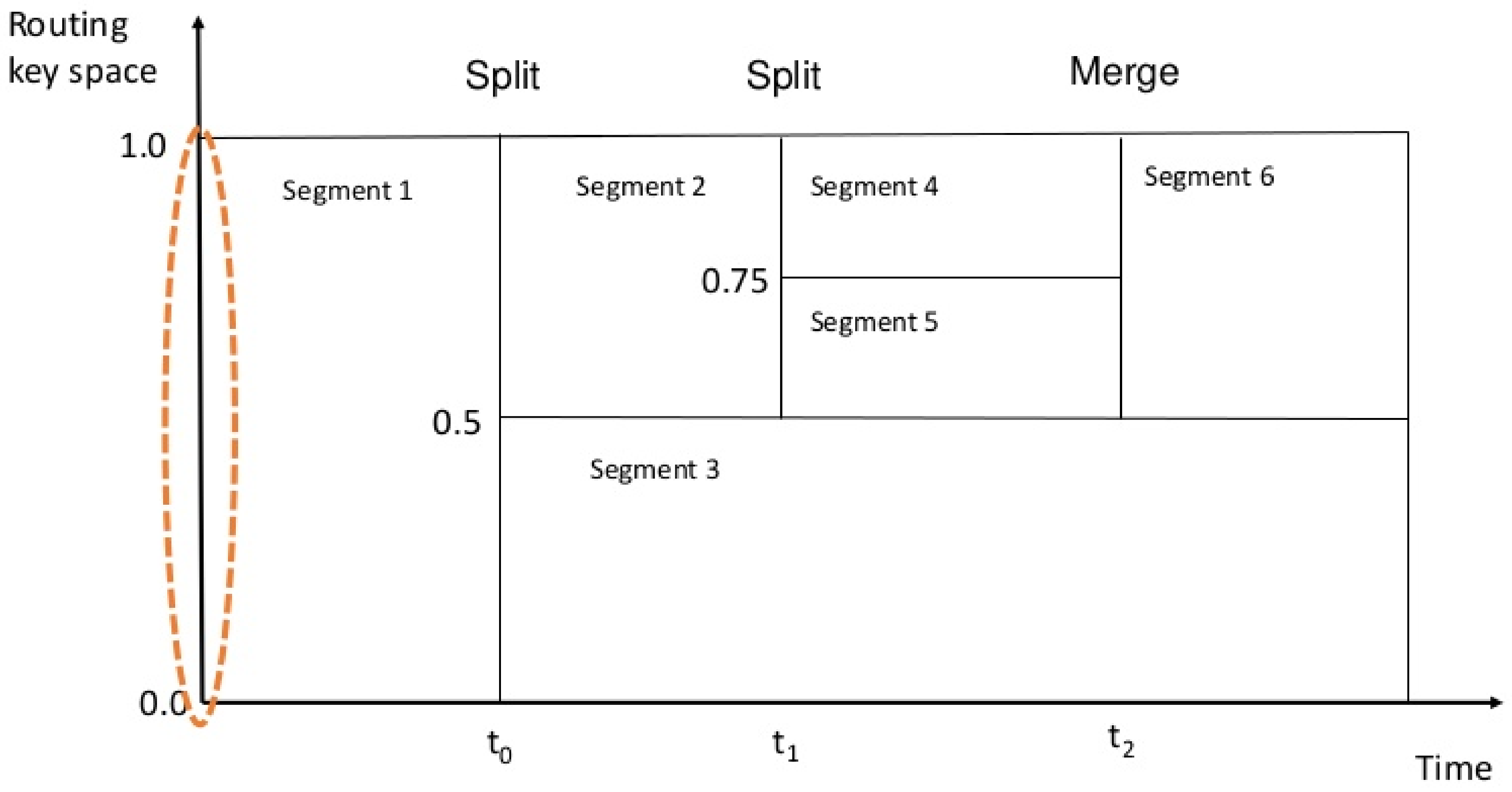


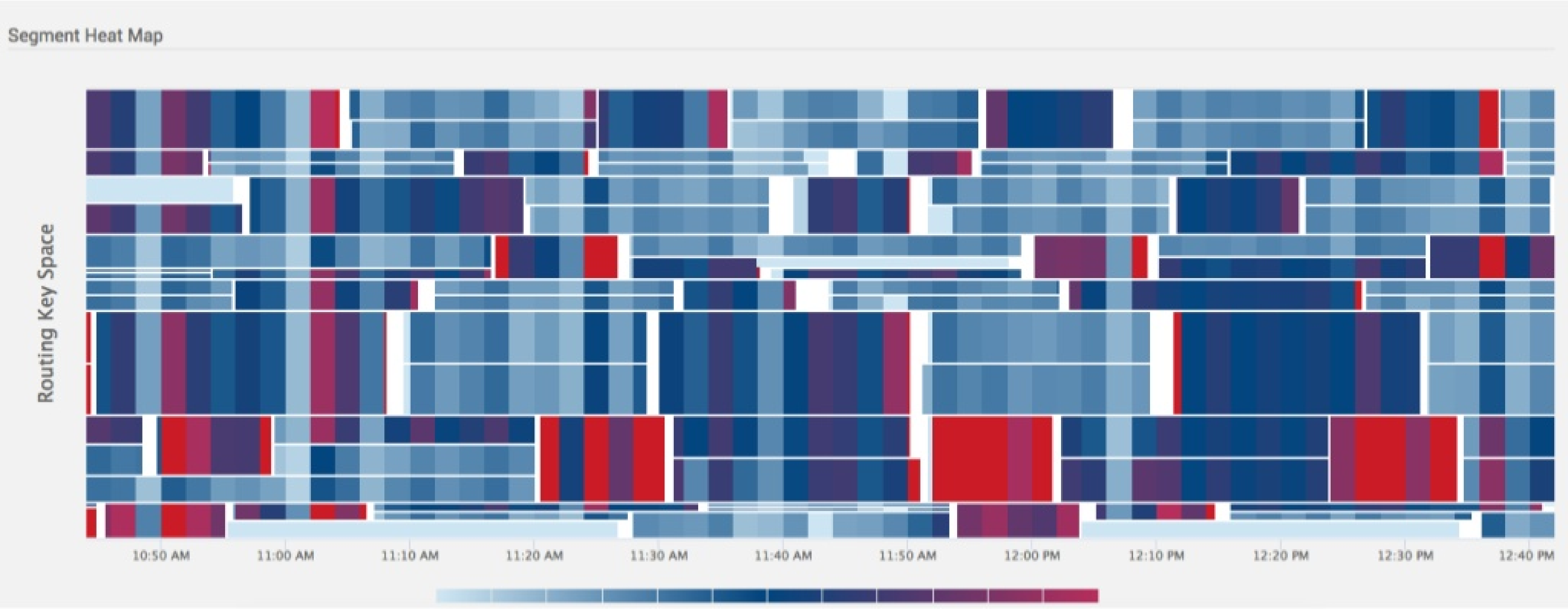
- Seal current segment
- Create new ones

2

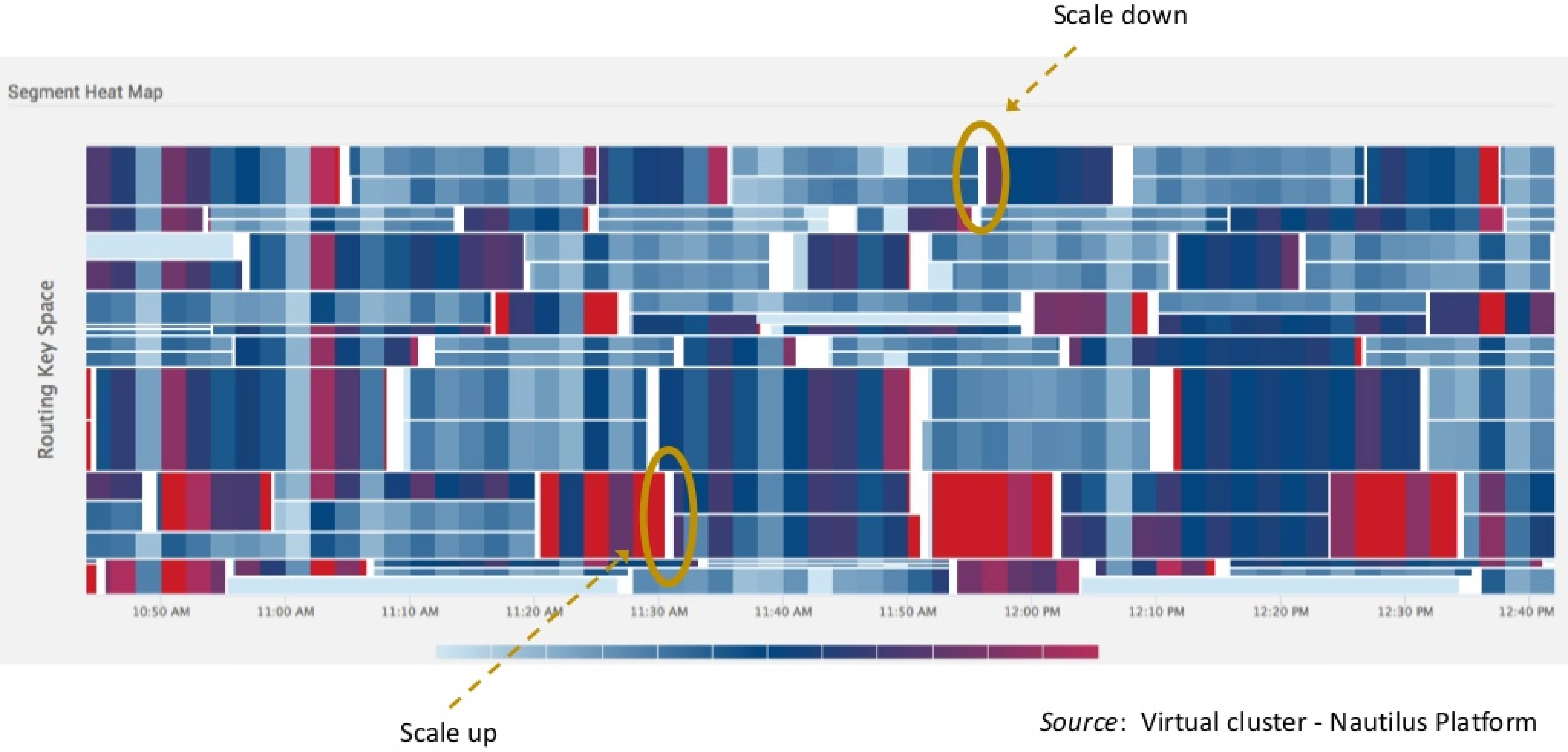

Routing key space
Split
Split
Merge
1.0
Segment 1
Segment 2
Segment 4
Segment 6
0.75
Segment 5
0.5
Segment 3
0.0
t_0
t_1
t_2
Time

Key ranges are not statically assigned to segments



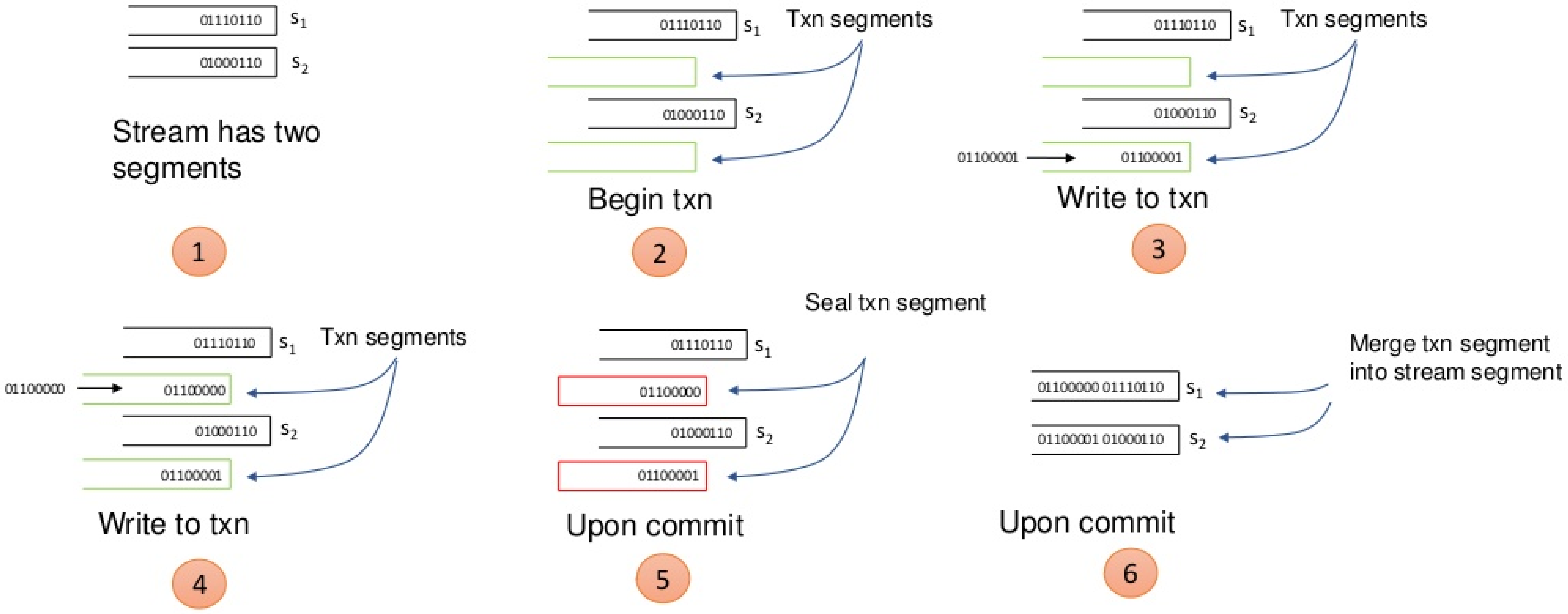


Source: Virtual cluster - Nautilus Platform

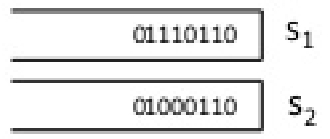


Source: Virtual cluster - Nautilus Platform

Transactions

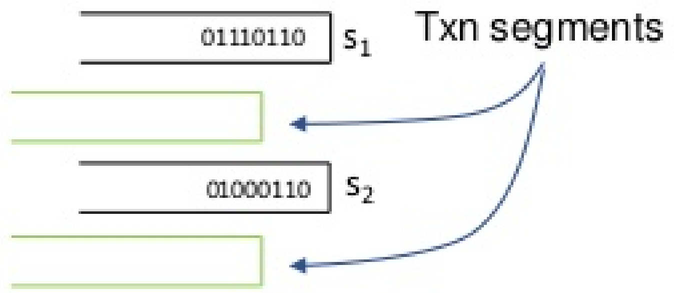


Transactions



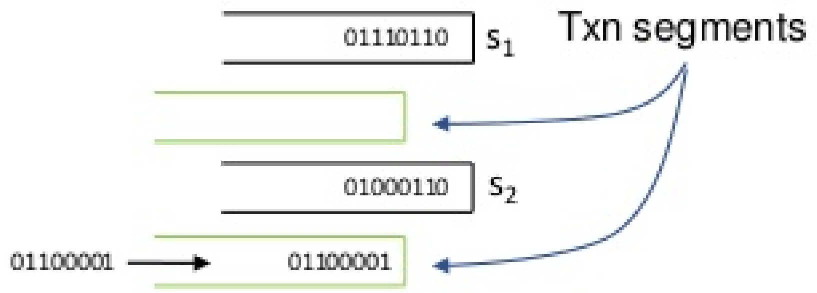
Stream has two segments

1



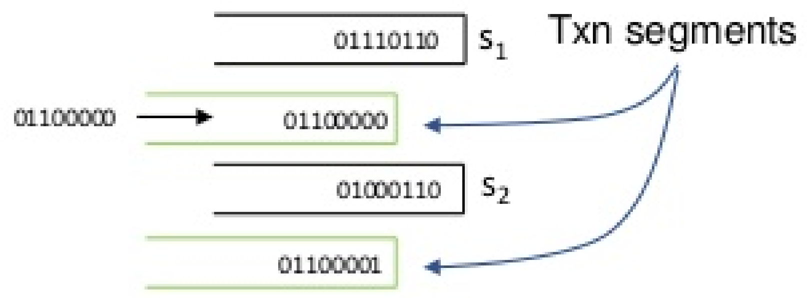
Begin txn

2



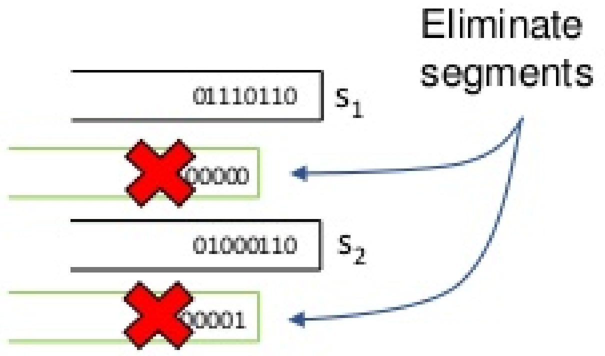
Write to txn

3



Write to txn

4



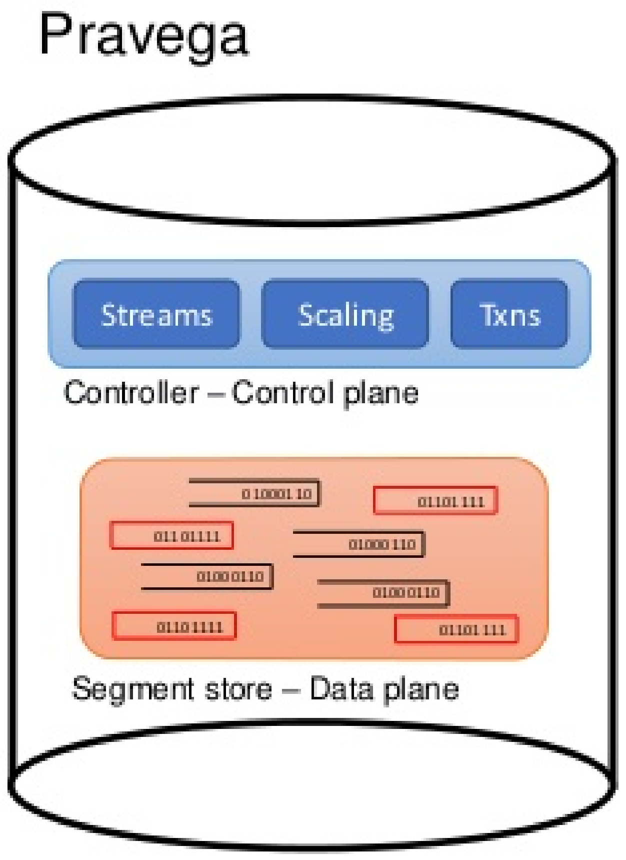
Upon abort

5

Wait, how are segments manipulated?

Controller

- Control plane
- A few of the controller tasks
  - Stream lifecycle
    - Create
    - Delete
    - Scale
  - Txn management
    - Create
    - Commit/Abort



API – *Writer* and *Reader*

Events

- Internally
  - Pravega is all about bytes
- Current API focused on events
  - Some encapsulation of application bytes
  - `Serializer` interface

```
public interface Serializer<T> {
   /**
    * Serializes the given event.
    *
    * @param value The event to be serialized.
    * @return The serialized form of the event.
    * NOTE: buffers returned should not exceed {@link #MAX_EVENT_SIZE}.
    */
   ByteBuffer serialize(T value);

   /**
    * Deserializes the given ByteBuffer into an event.
    *
    * @param serializedValue A event that has been previously serialized.
    * @return The event object.
    */
   T deserialize(ByteBuffer serializedValue);
}
```

EventStreamWriter API

```
String scope = “myScope”;
WriterConfig config = new WriterConfig();
String streamName = “myStream”;

ClientFactory factory = ClientFactory.withScope(scope, new URI(“//demo.pravega.io:3333”));
EventStreamWriter<String> writer = factory.createEventWriter(streamName, serializer, config);

while(!worldEnd) {
    /* E.g., getNewRecord() reads the next line in a file */
    String record = getNewRecord();
    String key = extractKey(record);
    String event = extractEvent(record);
    writer.writeEvent(key, event);
}
```

Using the `EventStreamReader` API

```
String scope = “myScope”;
String myReaderId = “myId”;
String myReadGroup = “myGroup”;
ReaderConfig config = new ReaderConfig(props);
String streamName = “myStream”;

ClientFactory factory = ClientFactory.withScope(scope, new URI(“//127.0.0.1:3333”);
EventStreamReader<String> reader = factory.createEventReader(myReaderId,
                                                            myReadGroup,
                                                            serializer,
                                                            config);

while(!worldEnd) {
  EventRead<String> event = reader.readNextEvent(long timeout);

  // Application consumes event and it is supposed to persist the Position
  // object to guarantee exactly once.
  consumeEvent(event.getEvent(), event.getPosition());
}

reader.close();
```

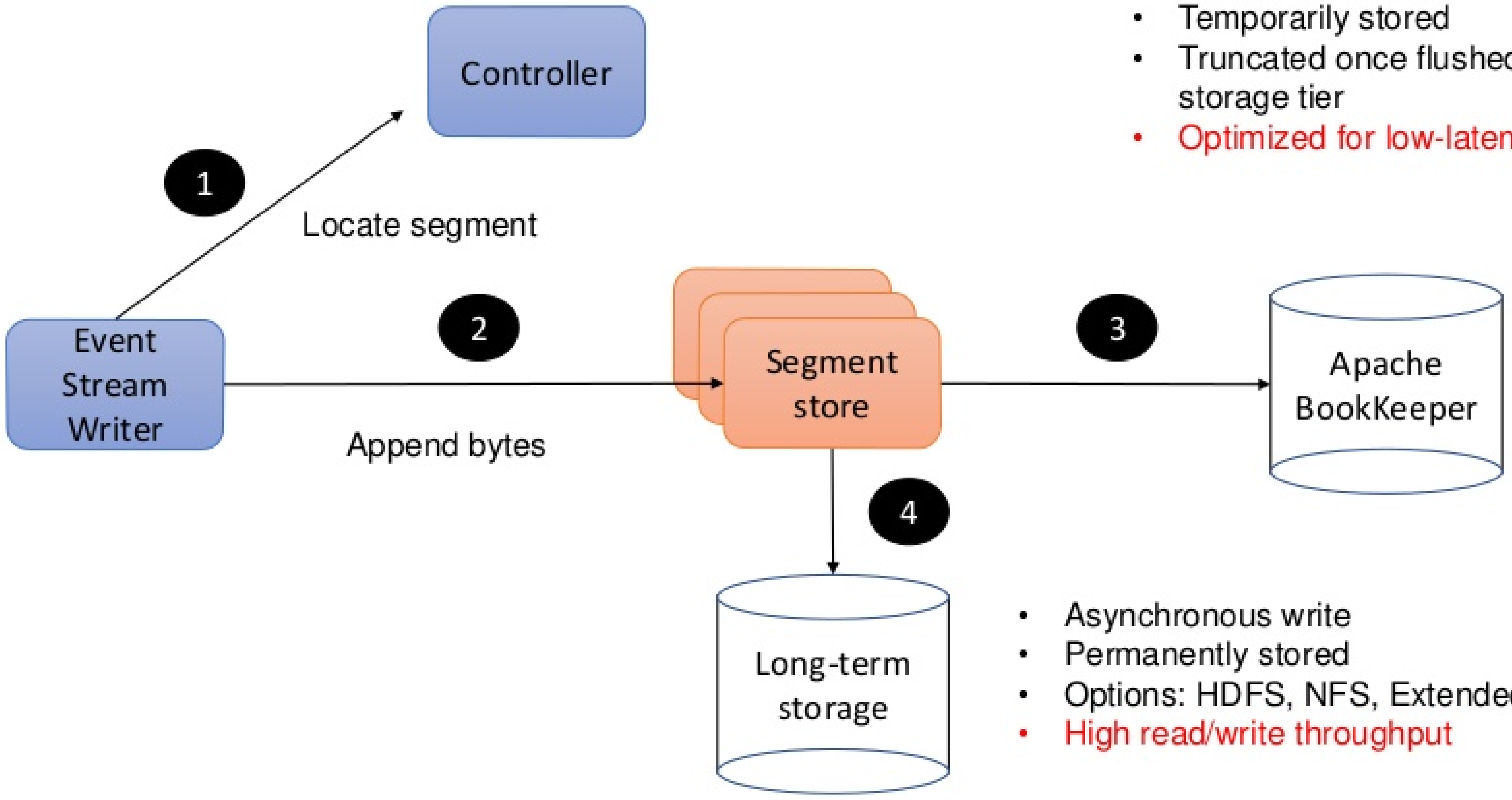
Transactions

```
Txn txn = writer.beginTxn();

txn.writeEvent(getKey("Pravega"), "Pravega");
txn.writeEvent(getKey("is"), "is");
txn.writeEvent(getKey("invading"), "invading");
txn.commit();
```

Pravega semantics

The write path



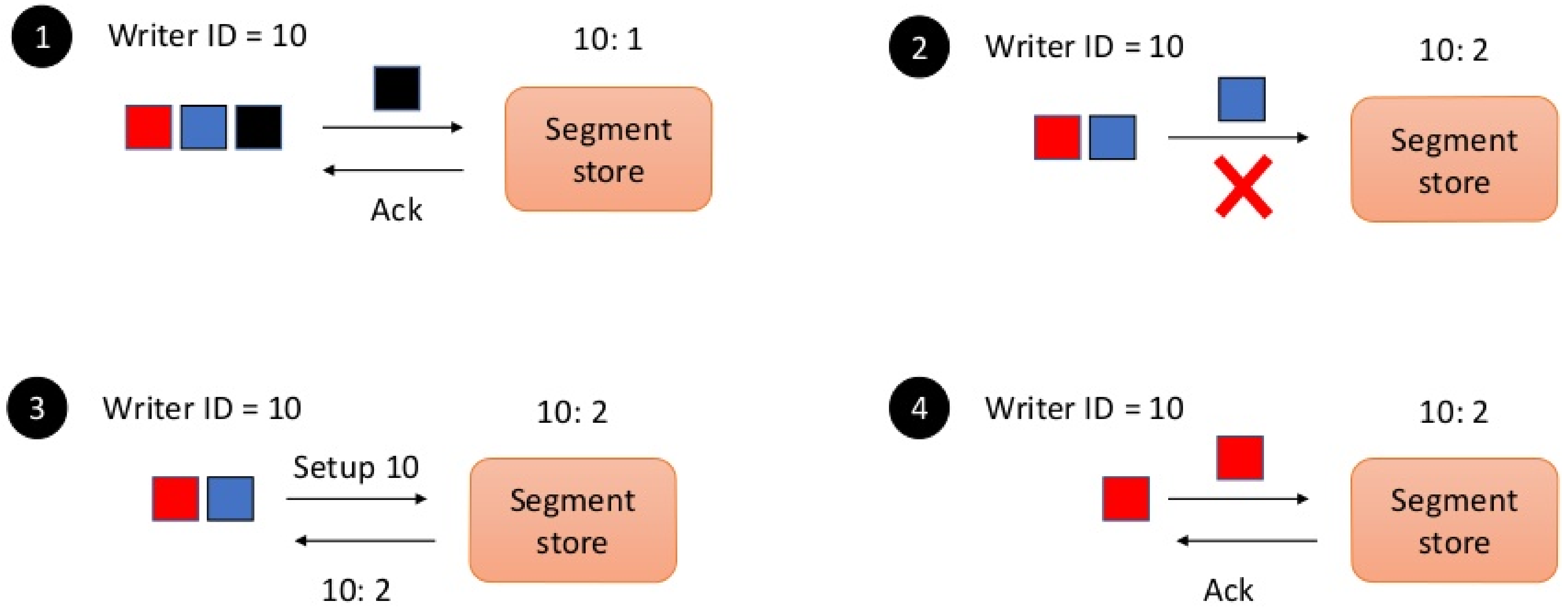
- Synchronous write
- Temporarily stored
- Truncated once flushed to next storage tier
- Optimized for low-latency writes

- Asynchronous write
- Permanently stored
- Options: HDFS, NFS, Extended S3
- High read/write throughput

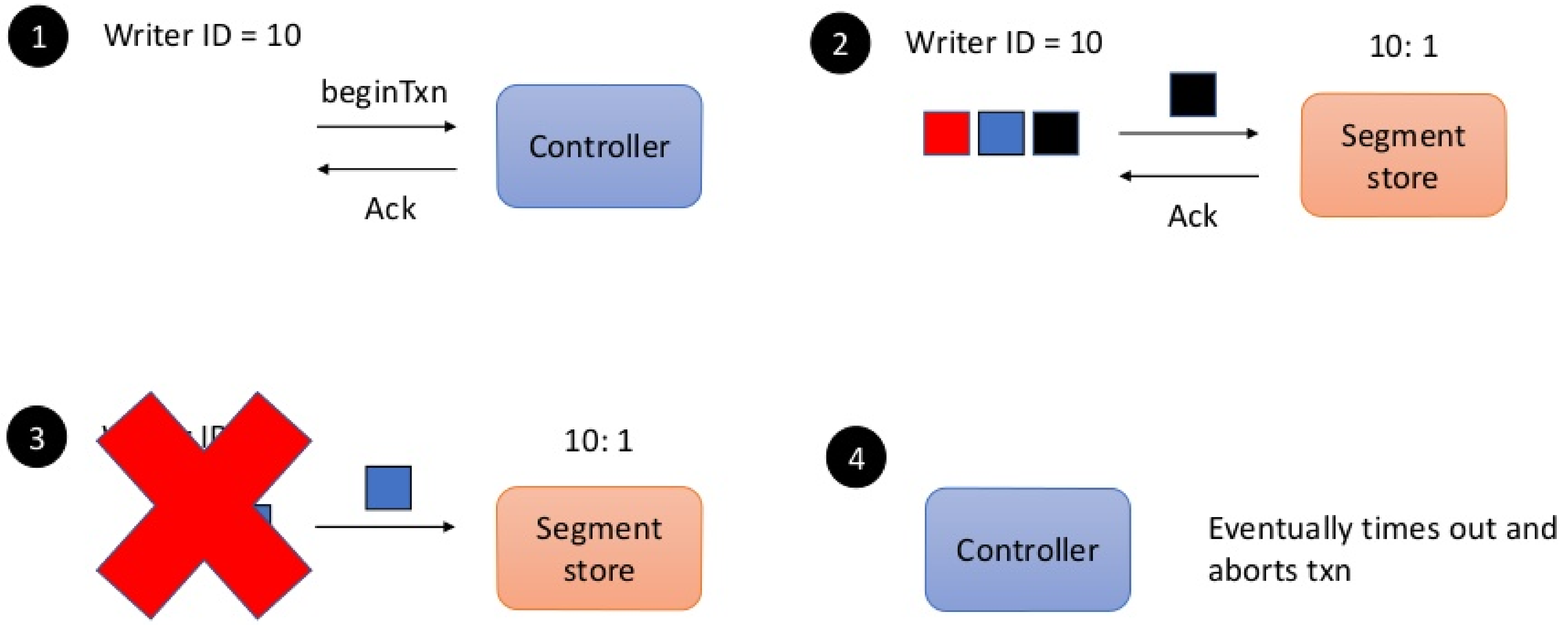
Guarantees on the write path

- Order
  - Writer appends in application order
- Duplicates
  - Writer IDs
  - Maps to last appended data on the segment store
  - Writer does not persist ID to tolerate a crash
- Txns
  - Atomicity at the stream level
  - If anything goes wrong with the writes, either abort or let it time out

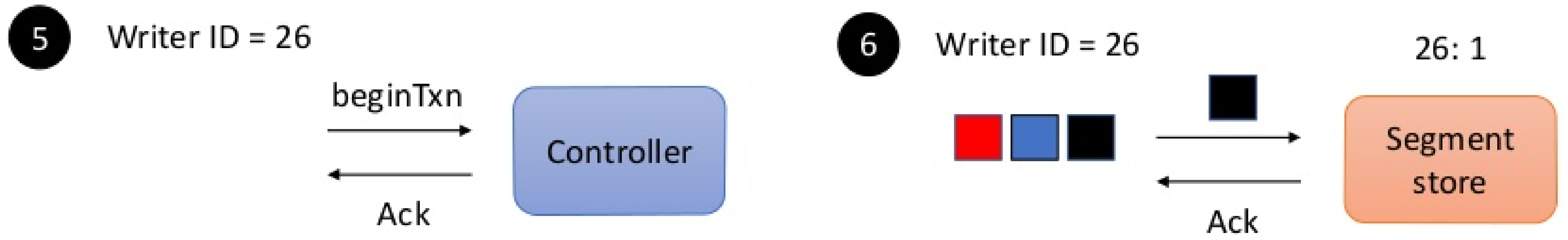
Avoiding duplicates – Reconnect



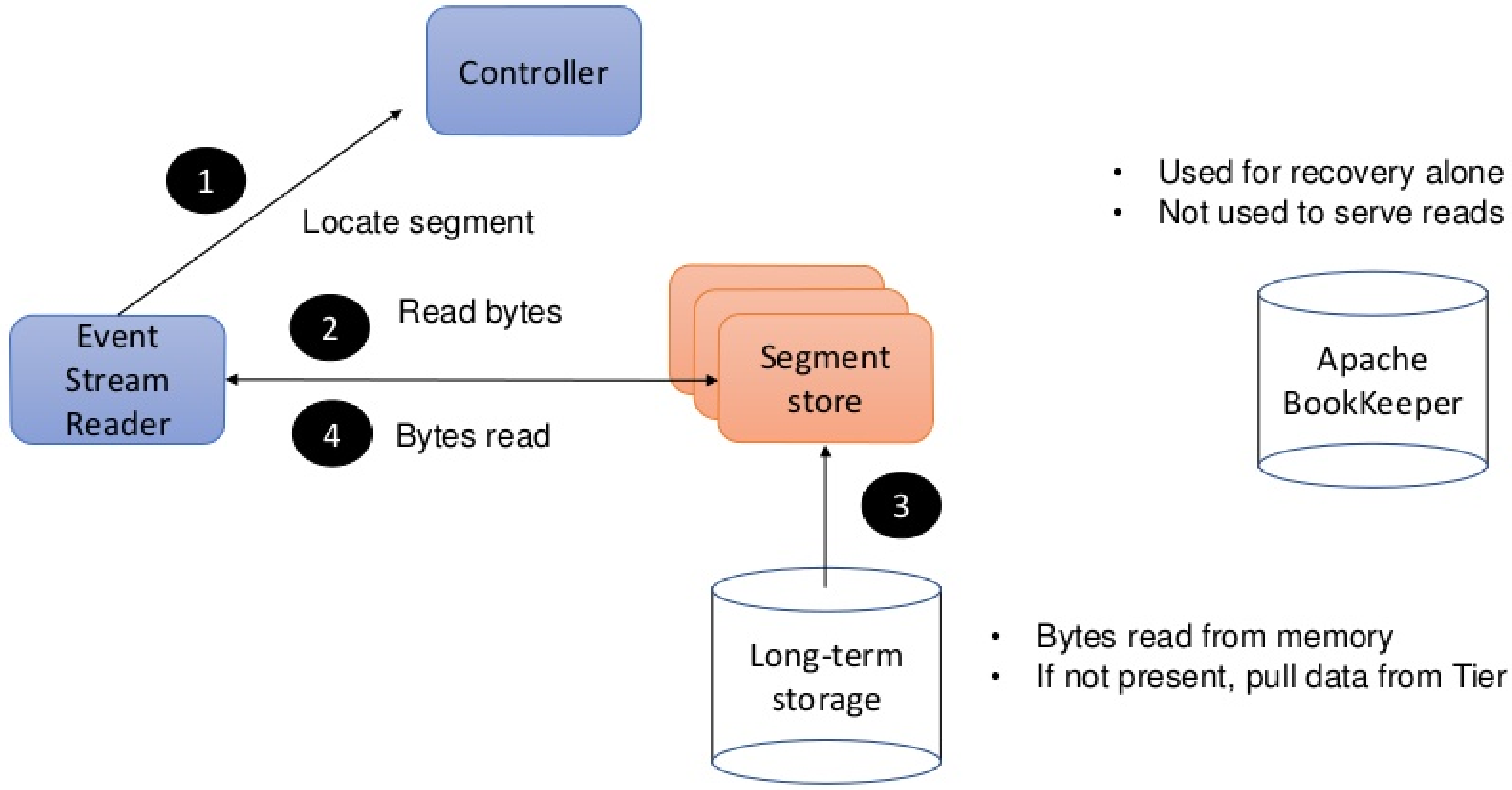
Avoiding duplicates – Transactional writes



Avoiding duplicates – Transactional writes



The read path



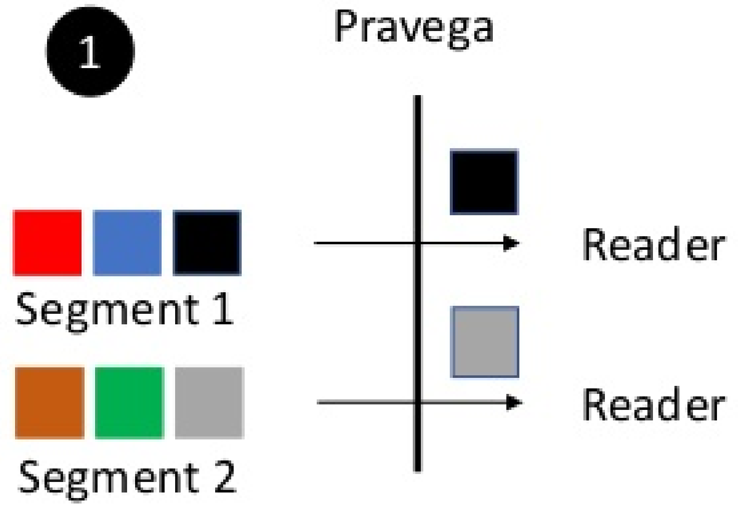
- Used for recovery alone
- Not used to serve reads

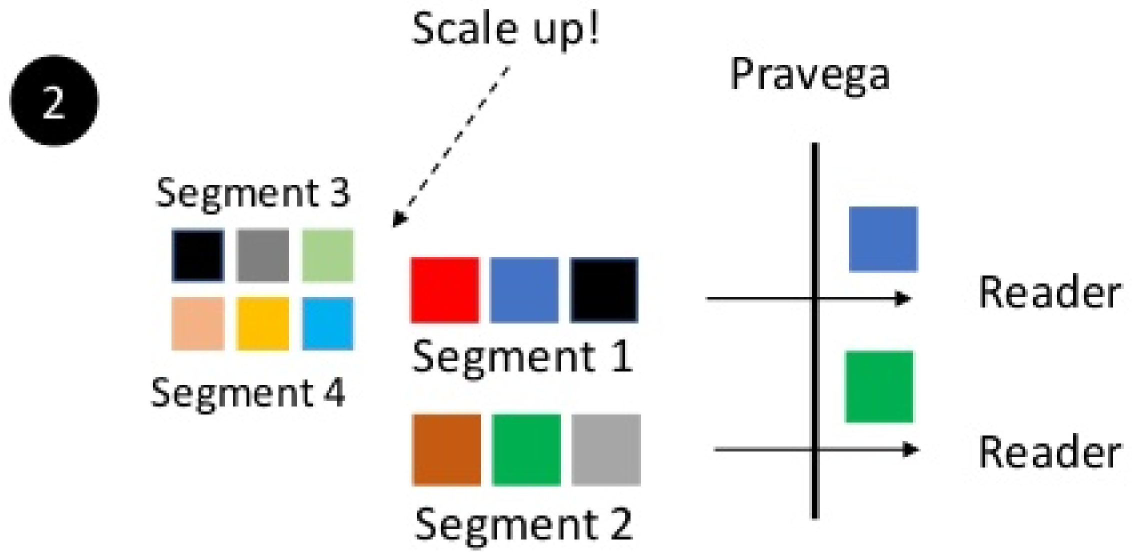
- Bytes read from memory
- If not present, pull data from Tier 2

Reader groups

- Group of event readers
  - Read events from a set of streams
  - Load distributed across readers of the group
- Segments
  - A given reader reads from a set of segments
  - Coordination of segment assignment done via a **state synchronizer**
- State synchronizer
  - General facility for synchronizing state across processes
  - Uses a revisioned Pravega segment
  - Advanced topic: not explained further in this talk

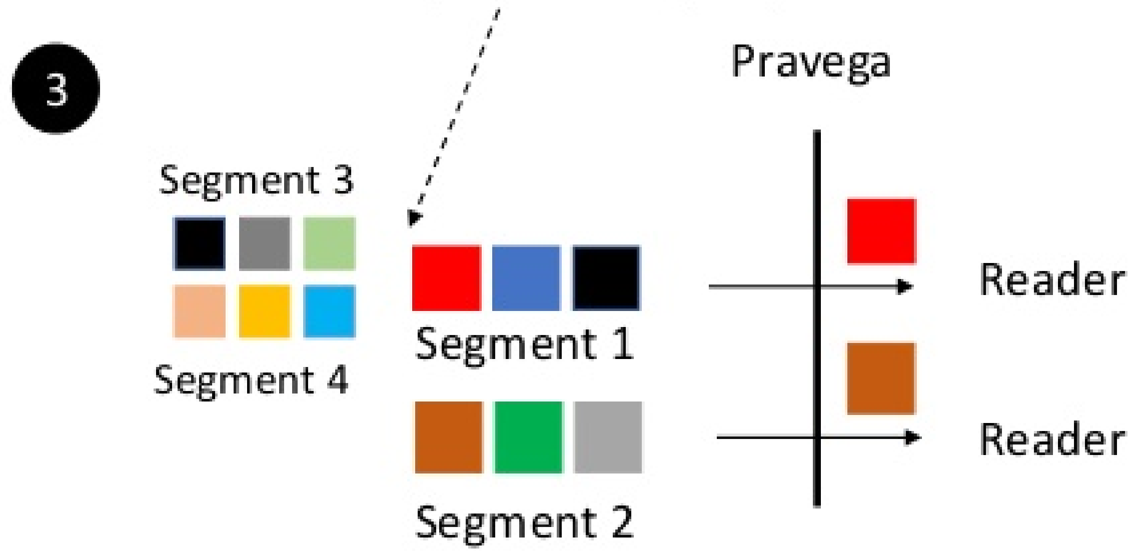
Reader groups + Scaling

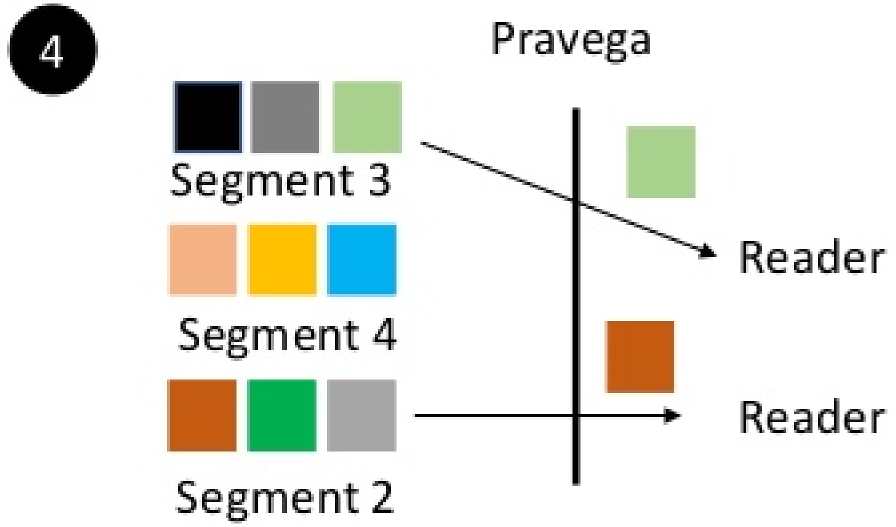


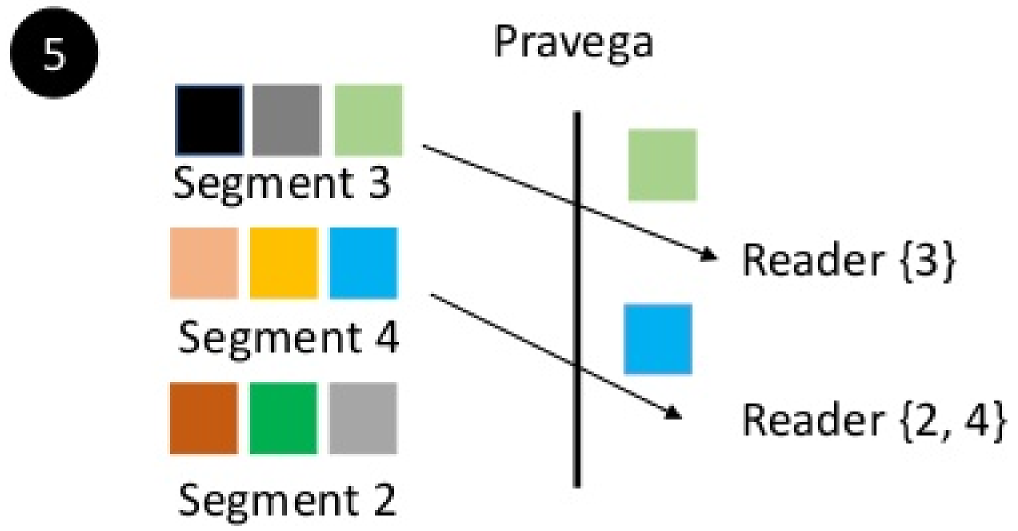


Reader groups + Scaling

- Hit end of segment
- Get successors from controller
- Update reader group state



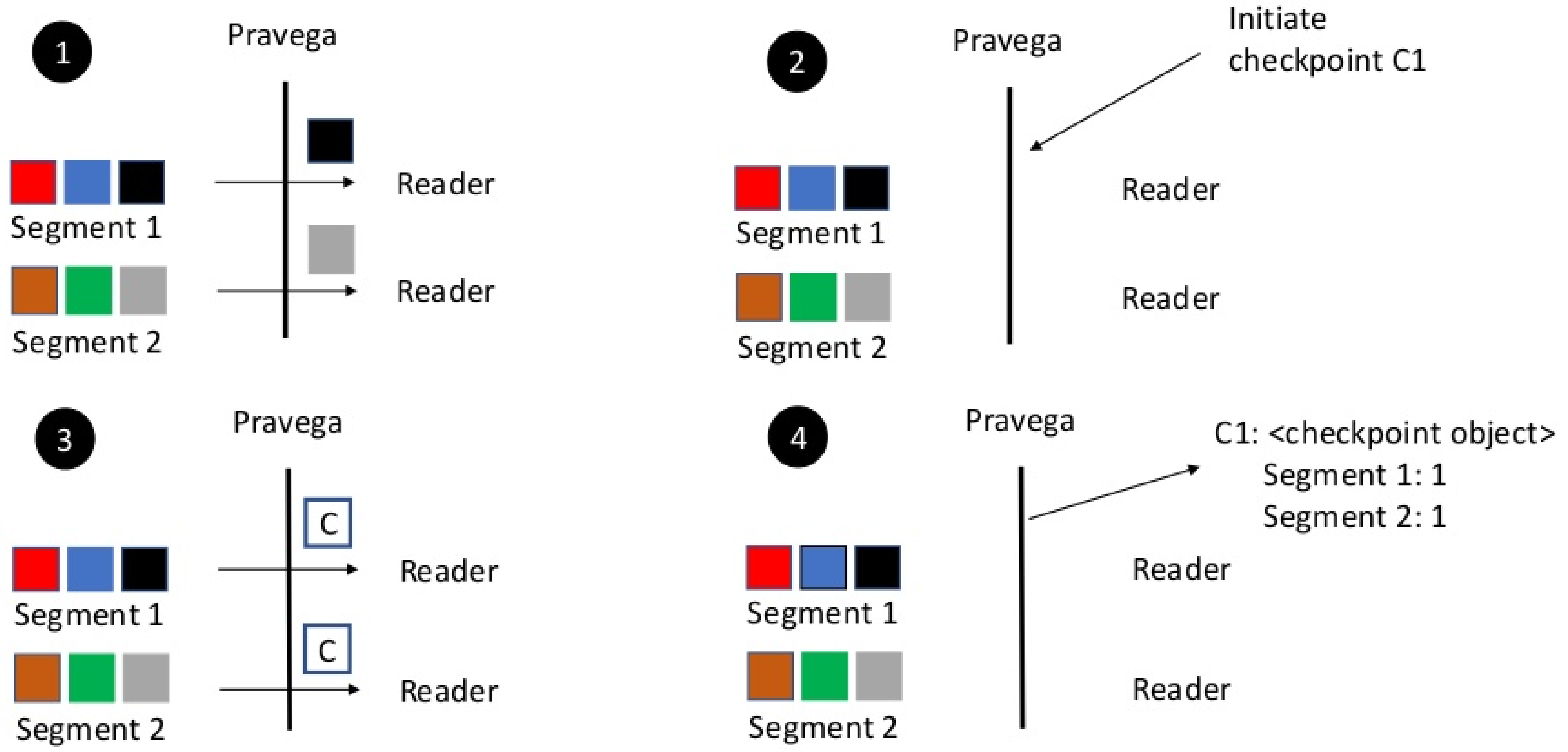




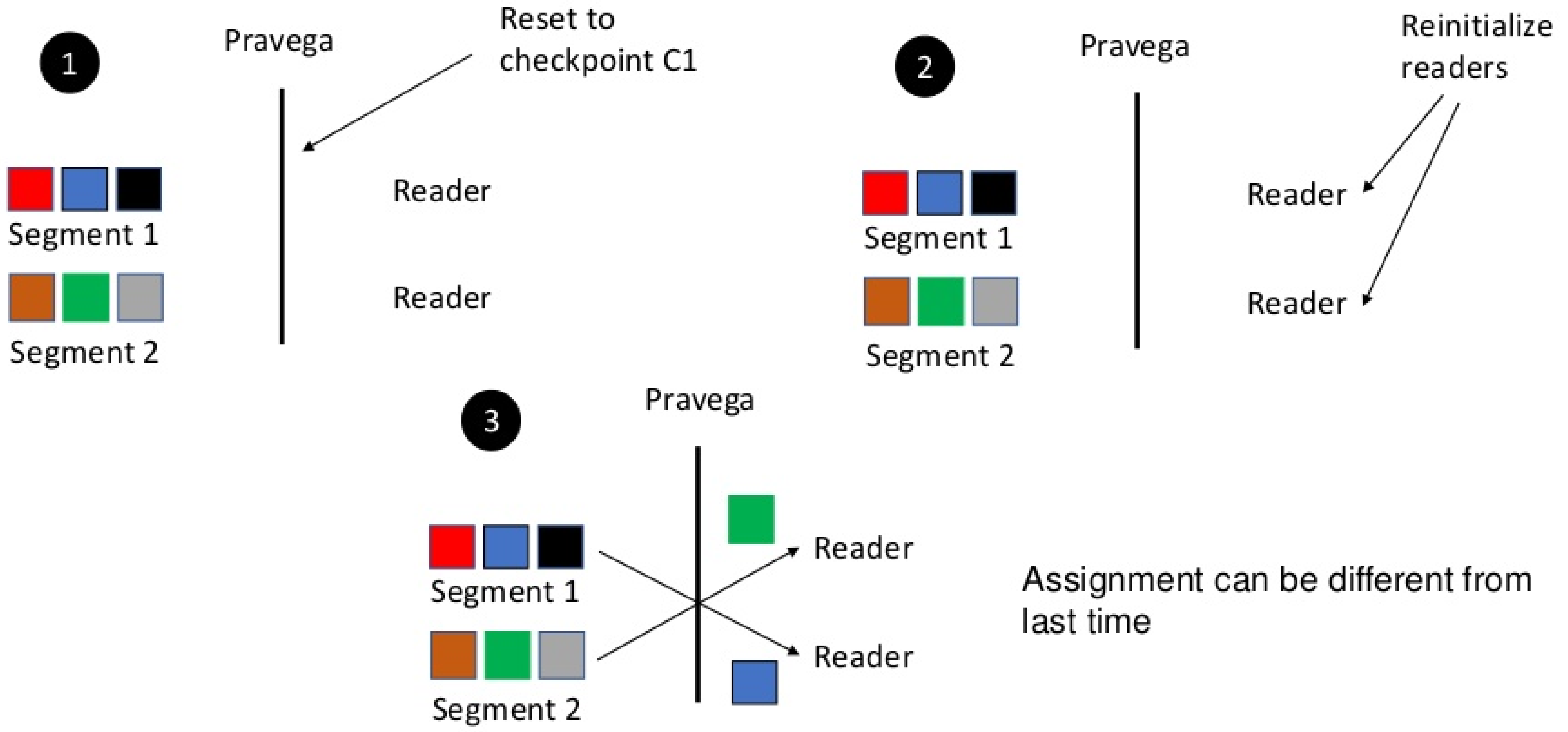
`Checkpoint` object

- Maps segments to corresponding offsets
- Opaque to the application

Getting a checkpoint



Resetting to a checkpoint



Assignment can be different from last time

Wrap up

Take-away messages

- Pravega is all about
  - Unbounded stream data
  - Permanently stored
  - Elasticity for streams
  - Scaling producers and consumers independently
- Under active development
- Looking at first use cases

Ongoing work

- Performance tuning
- Scaling support
- Event-time support
- Geo-distribution
- Security
- … and much more

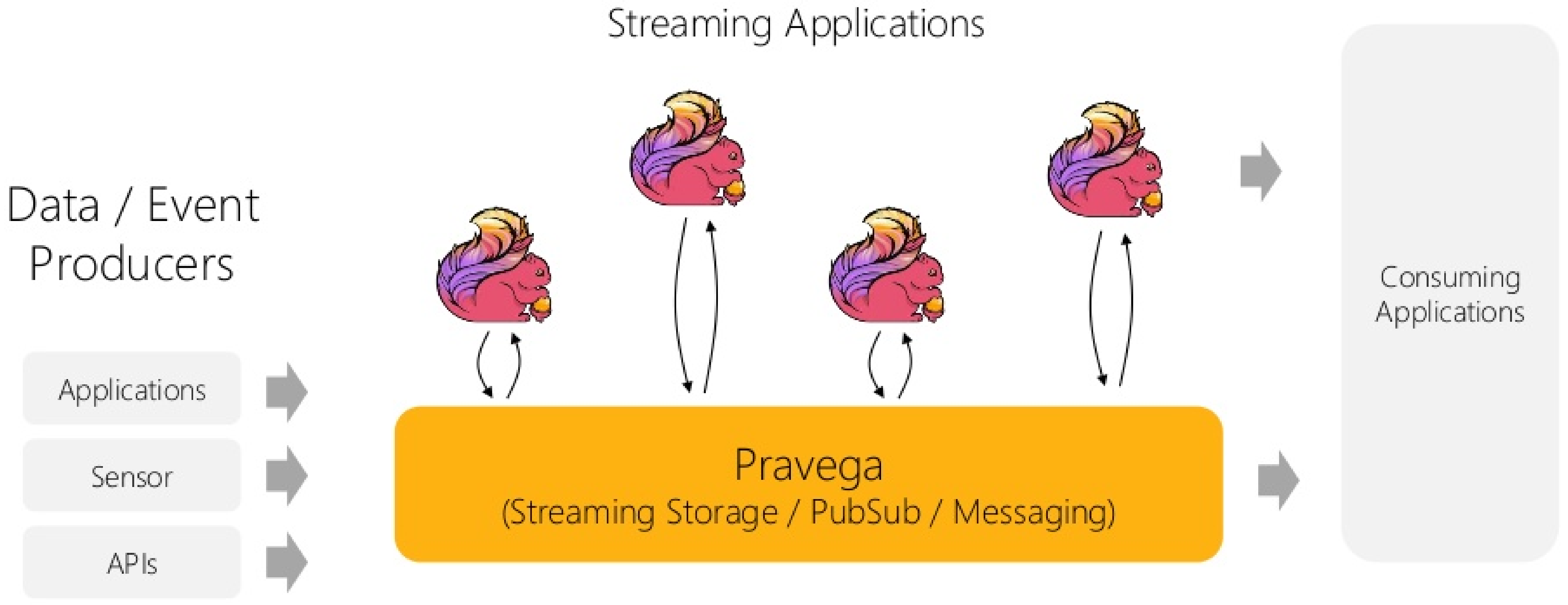
http://pravega.io

E-mail: fpj@apache.org
Twitter: @fpjunqueira

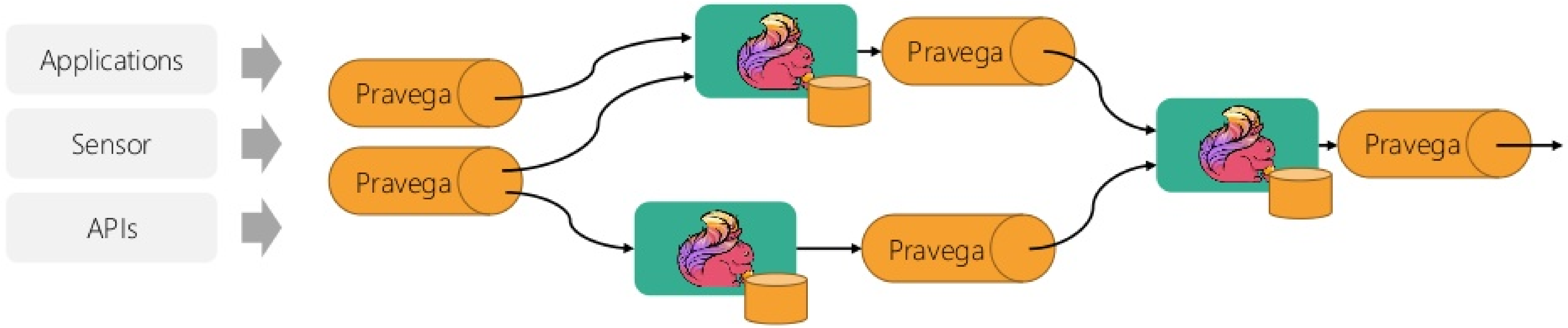
Join the community!

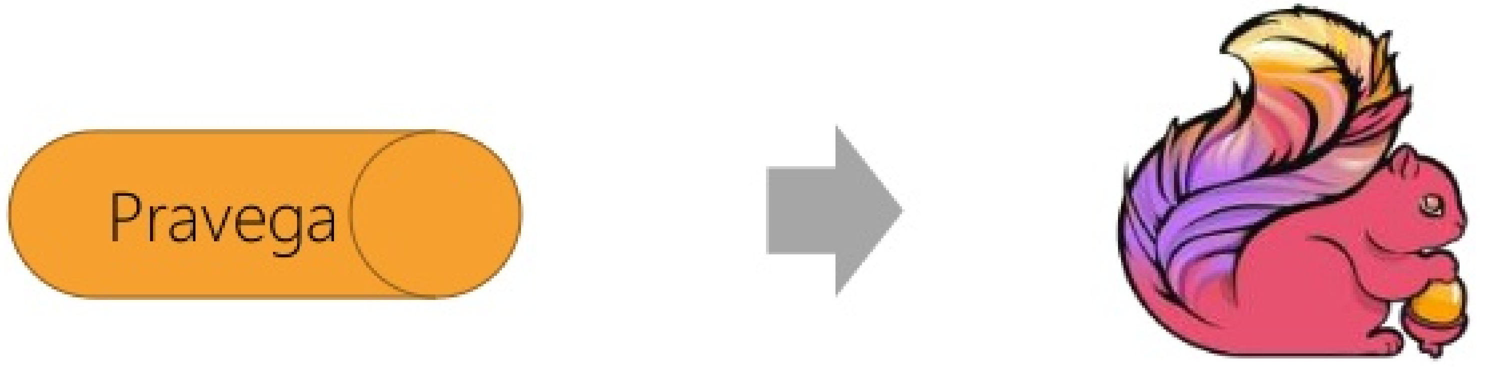
Pravega +

Streaming Storage and Compute



Streaming Pipelines

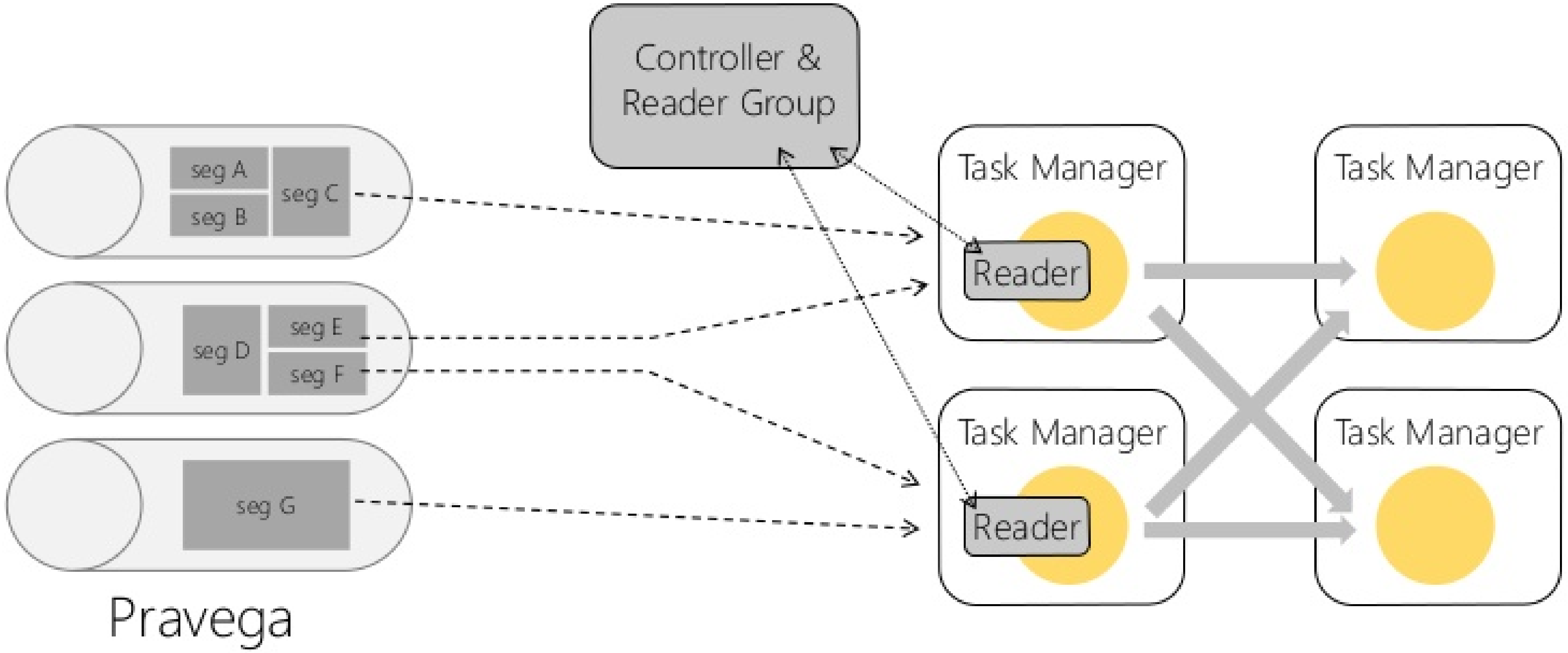




Flink reading from Pravega

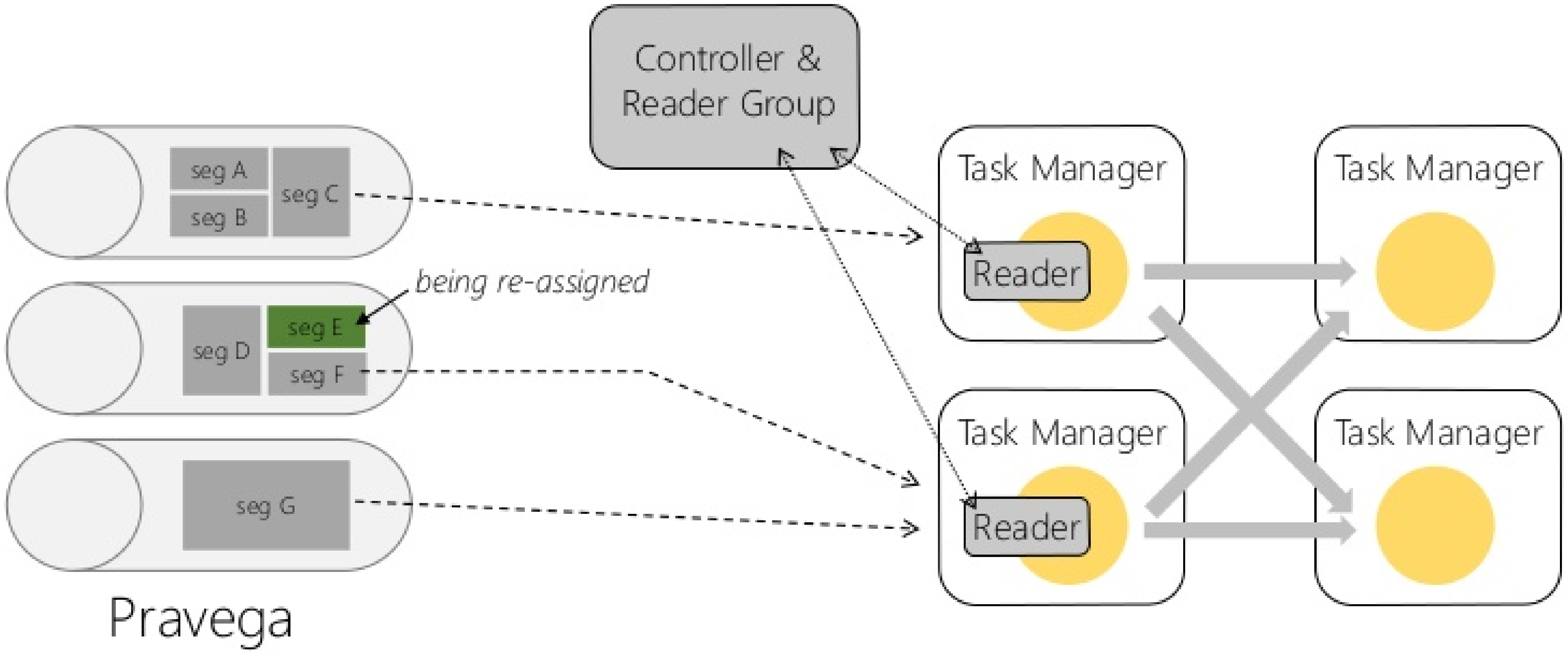
Reading via ReaderGroup

- Readers do not choose their own segments
- ReaderGroup automatically assigns and re-balances segments
- Leaving the ReaderGroup in charge is key to automatic scaling

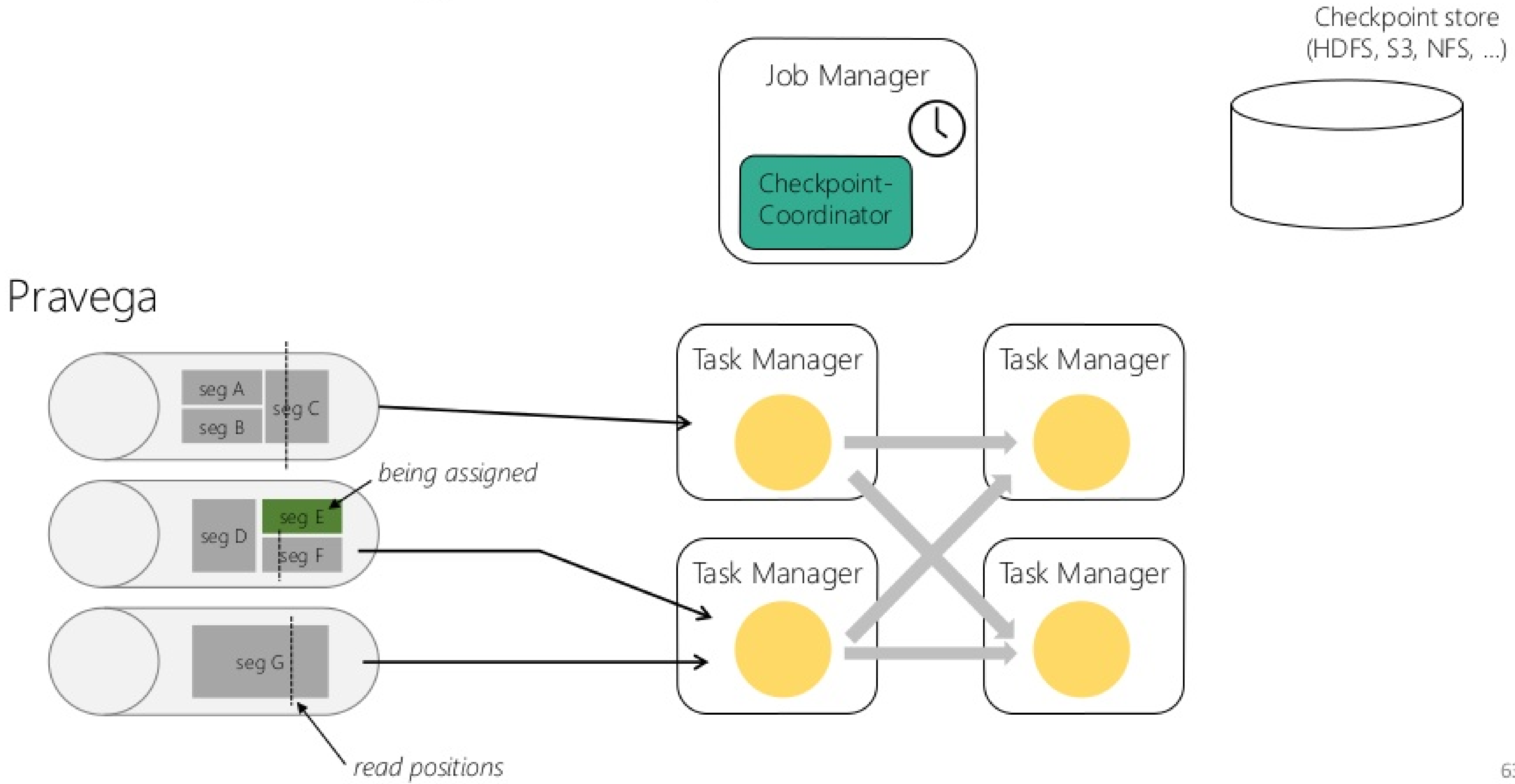


Reading via ReaderGroup

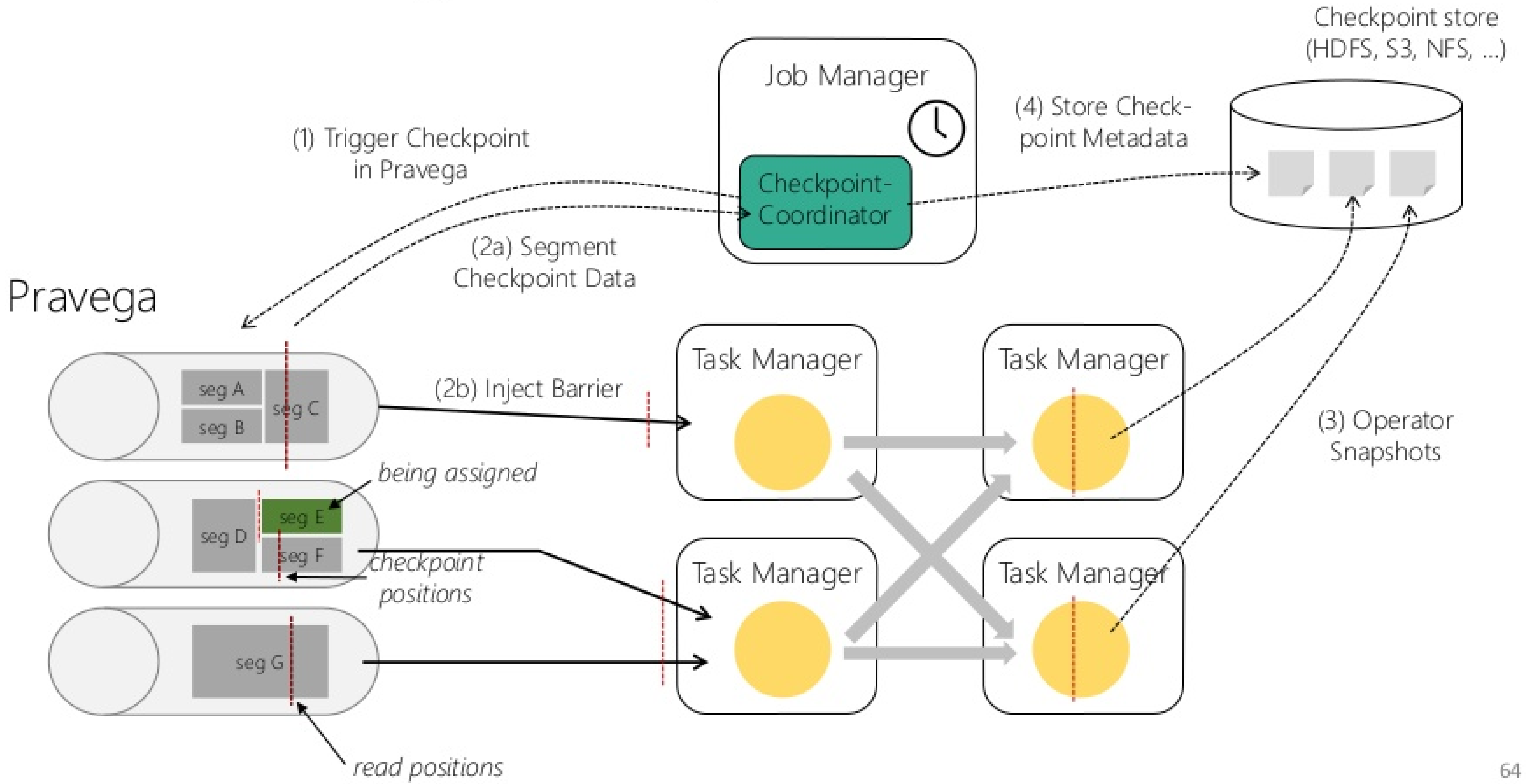
- At points in time, some segments may be not assigned to any reader
- Example: New segments, re-balancing segments, …

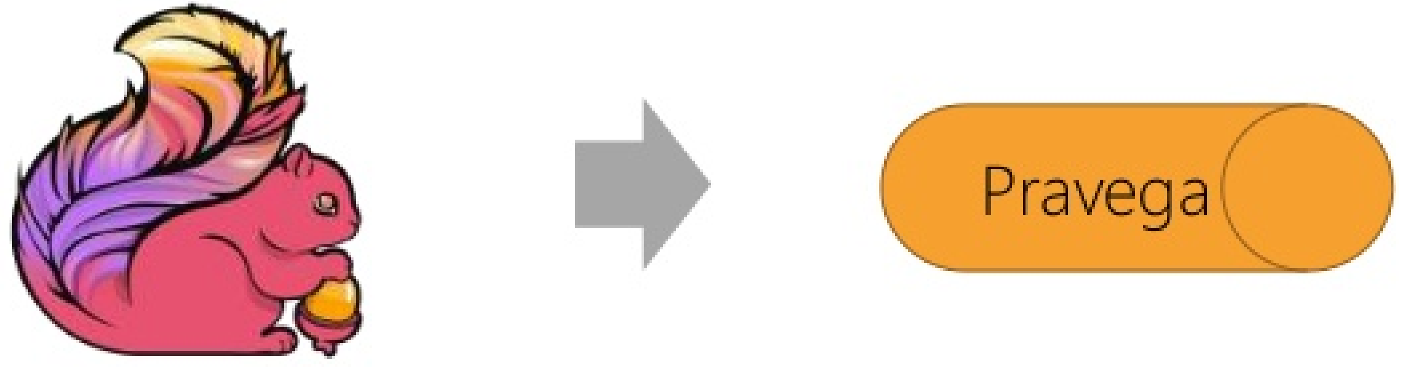


Flink + Pravega Checkpoints



Flink + Pravega Checkpoints

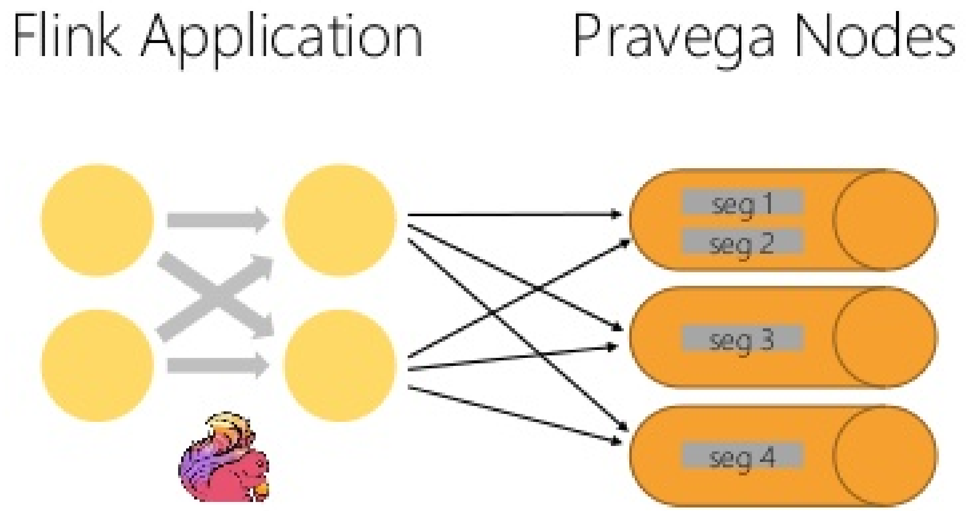




Flink writing to Pravega

The FlinkPravegaWriter

- Regular Flink SinkFunction
- No partitioner, but a "routing key"
- Remember: No partitions in Pravega
  - Just dynamically created segments
- Same key always goes to the same segment
- Order of elements guaranteed per key!

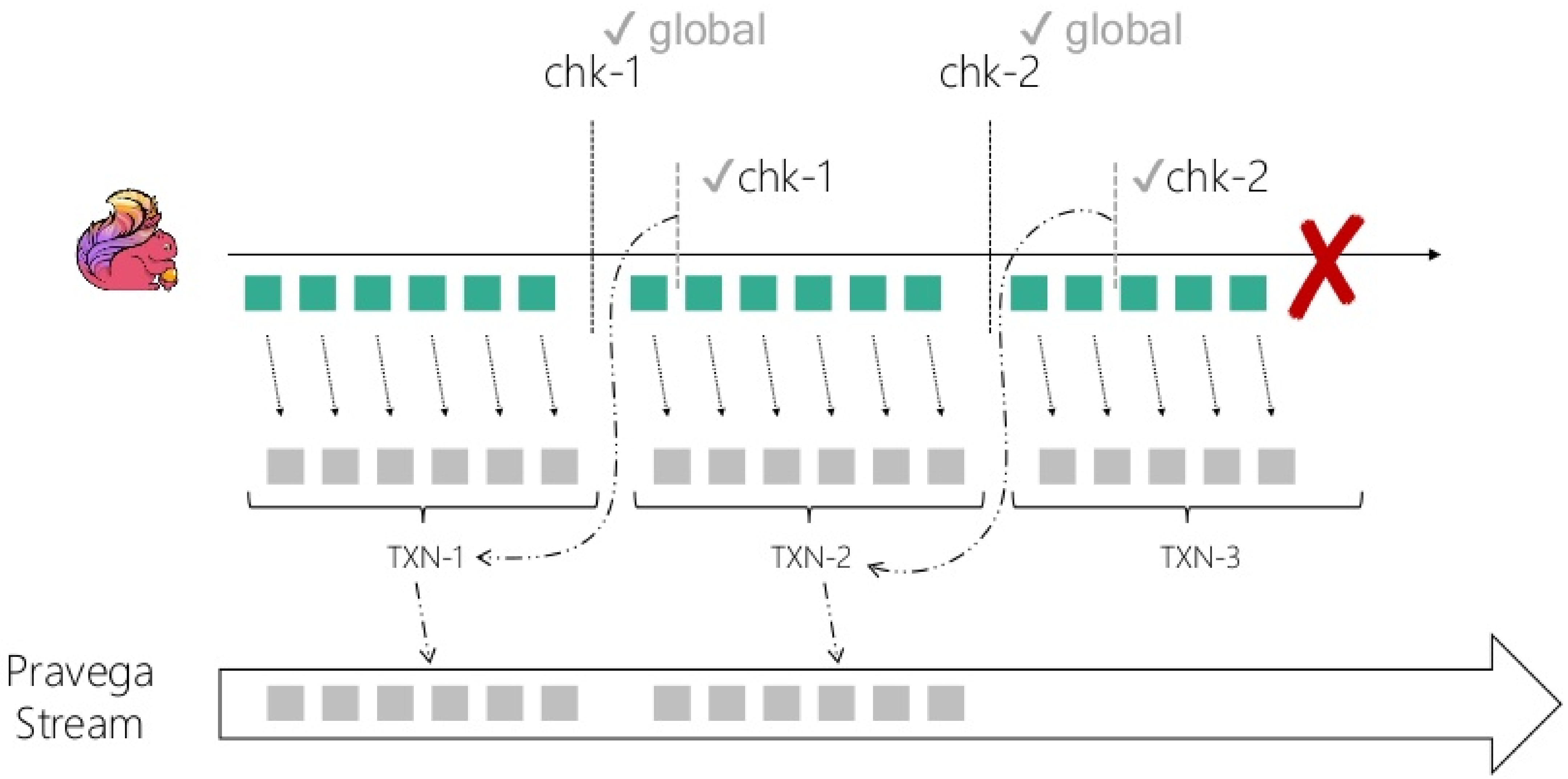


Exactly-once via Transactions

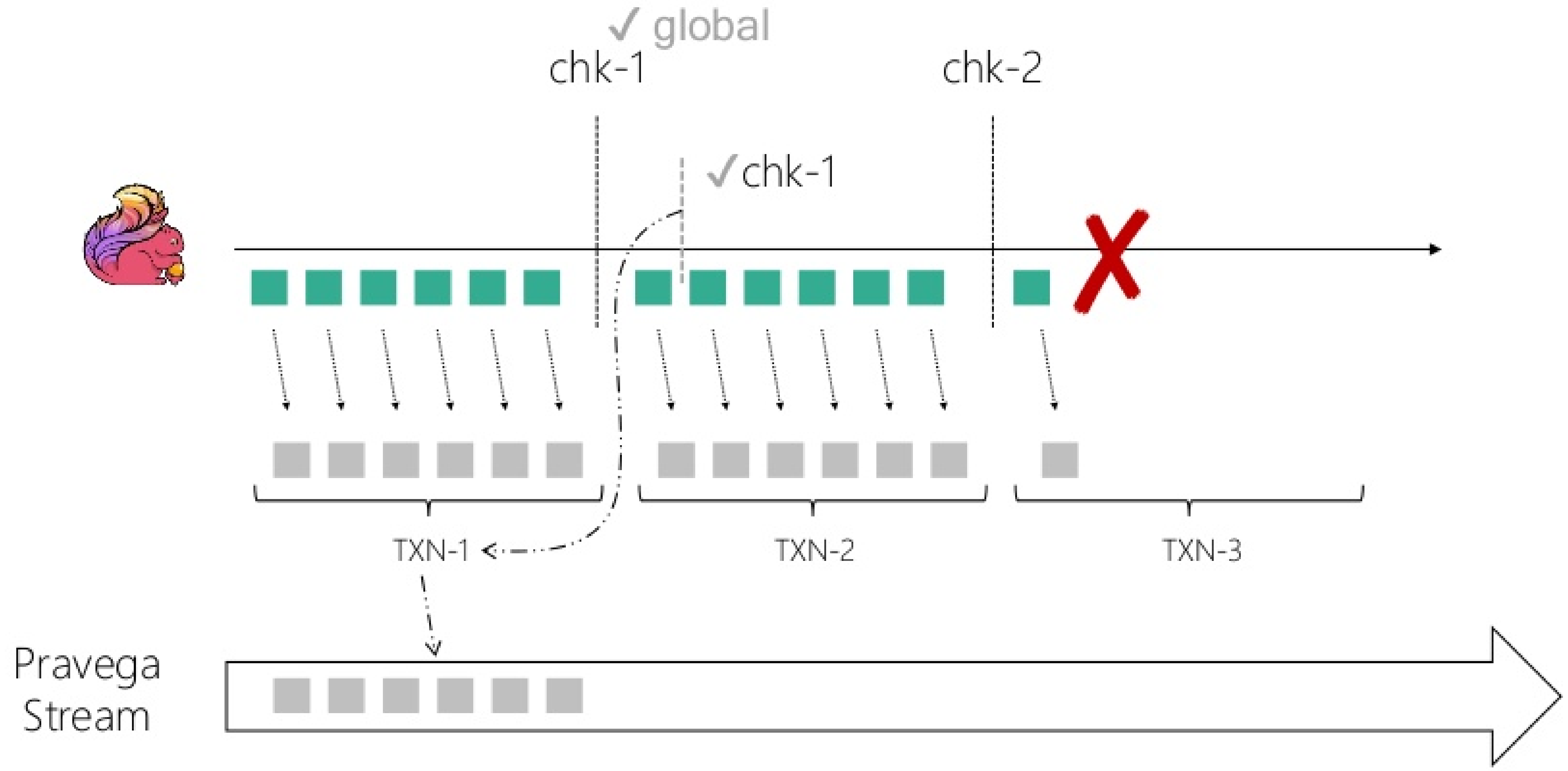
- Similar to a distributed 2-phase commit
- Coordinated by asynchronous checkpoints, no voting delays

- Basic algorithm:
  - Between checkpoints: Produce into transaction
  - On operator snapshot: Flush local transaction *(vote-to-commit)*
  - On checkpoint complete: Commit transactions
  - On recovery: check and commit any pending transactions

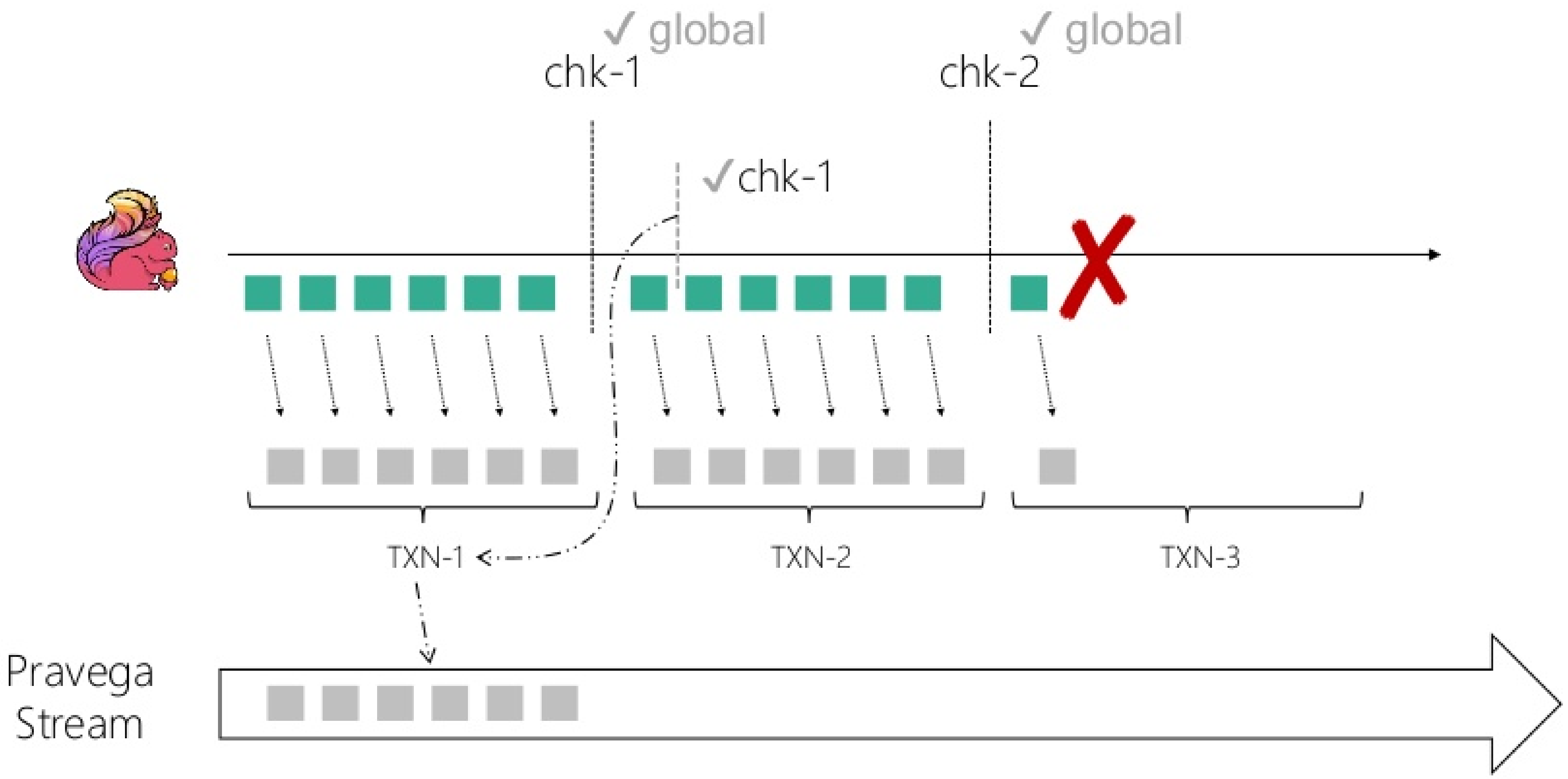
Exactly-once via Transactions



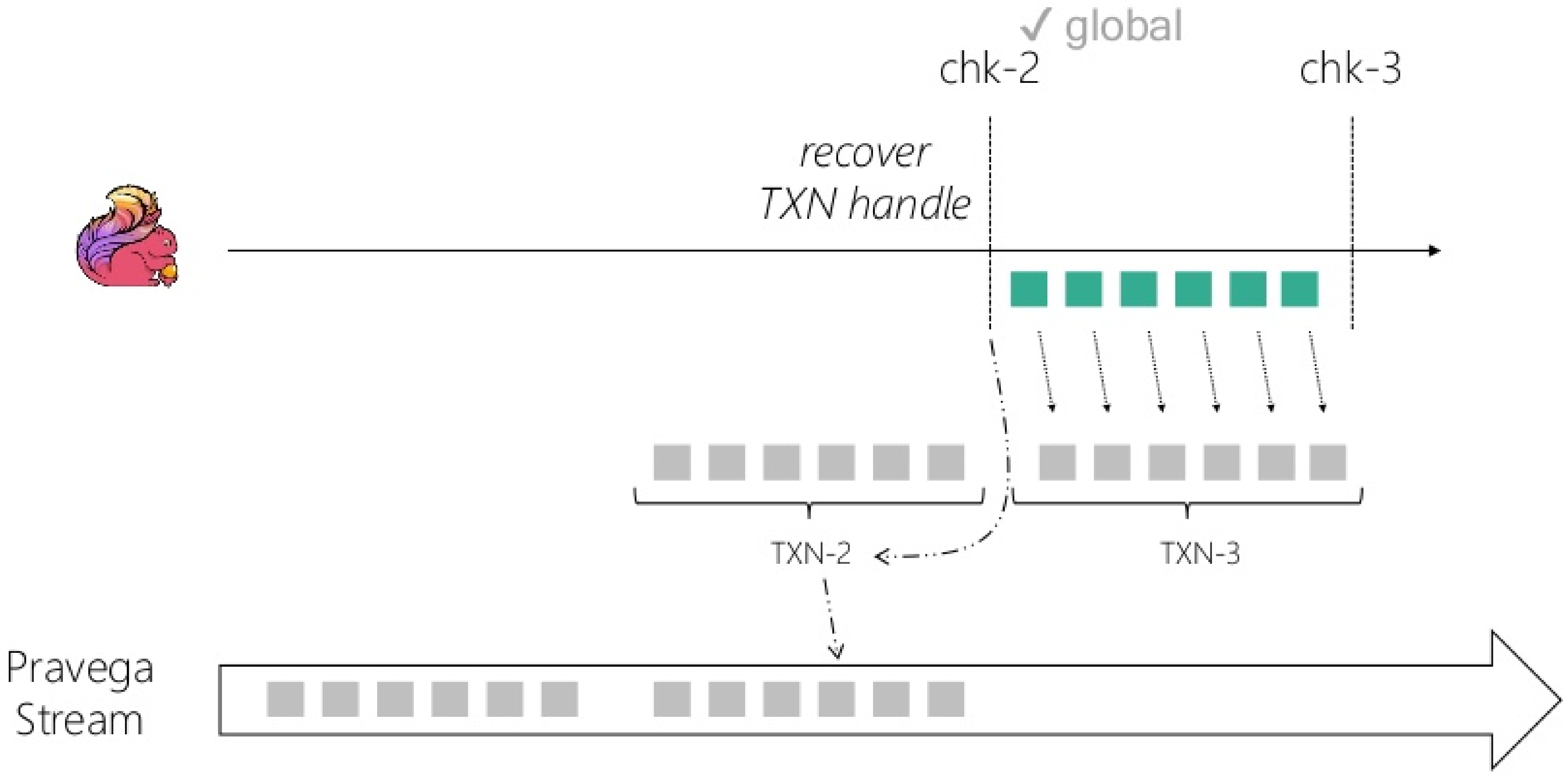
Transaction fails after local snapshot



Transaction fails before commit…



… commit on recovery



Looking ahead…

Looking ahead…

Automatic Scaling

Flink follows Pravega's scaling
(at least the first stage)

High Availability through Pravega

Use synchronizers
instead of ZooKeeper
*(leader election,
distributed atomic counters, …)*

Questions?

http://pravega.io
http://github.com/pravega/pravega
http://github.com/pravega/flink-connectors

E-mail: sewen@apache.org, fpj@apache.org
Twitter: @StephanEwen, @fpjunqueira