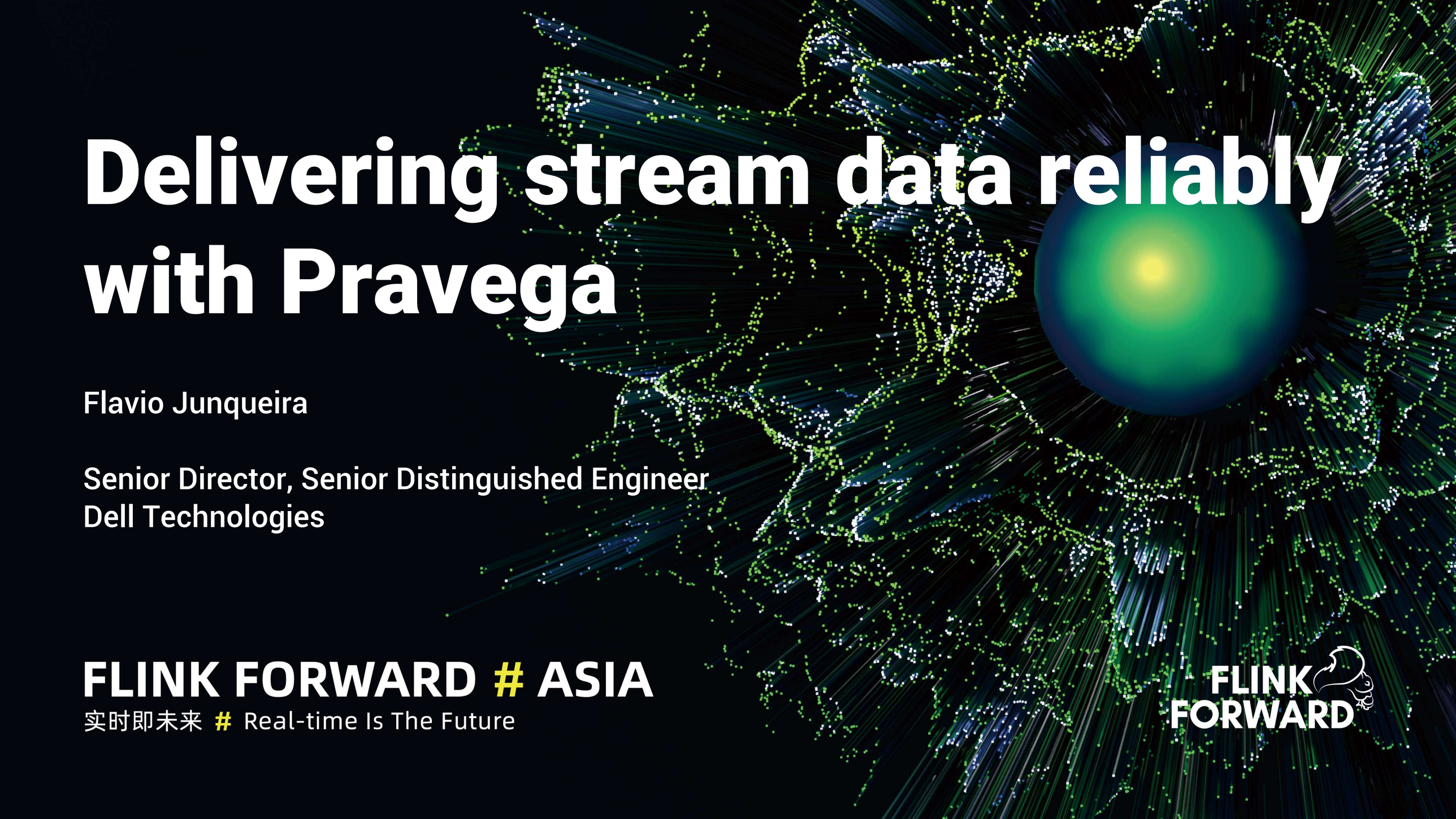


Delivering stream data reliably with Pravega

Flavio Junqueira

Senior Director, Senior Distinguished Engineer
Dell Technologies

FLINK FORWARD # ASIA
实时即未来 # Real-time Is The Future

FLINK FORWARD

Pravega

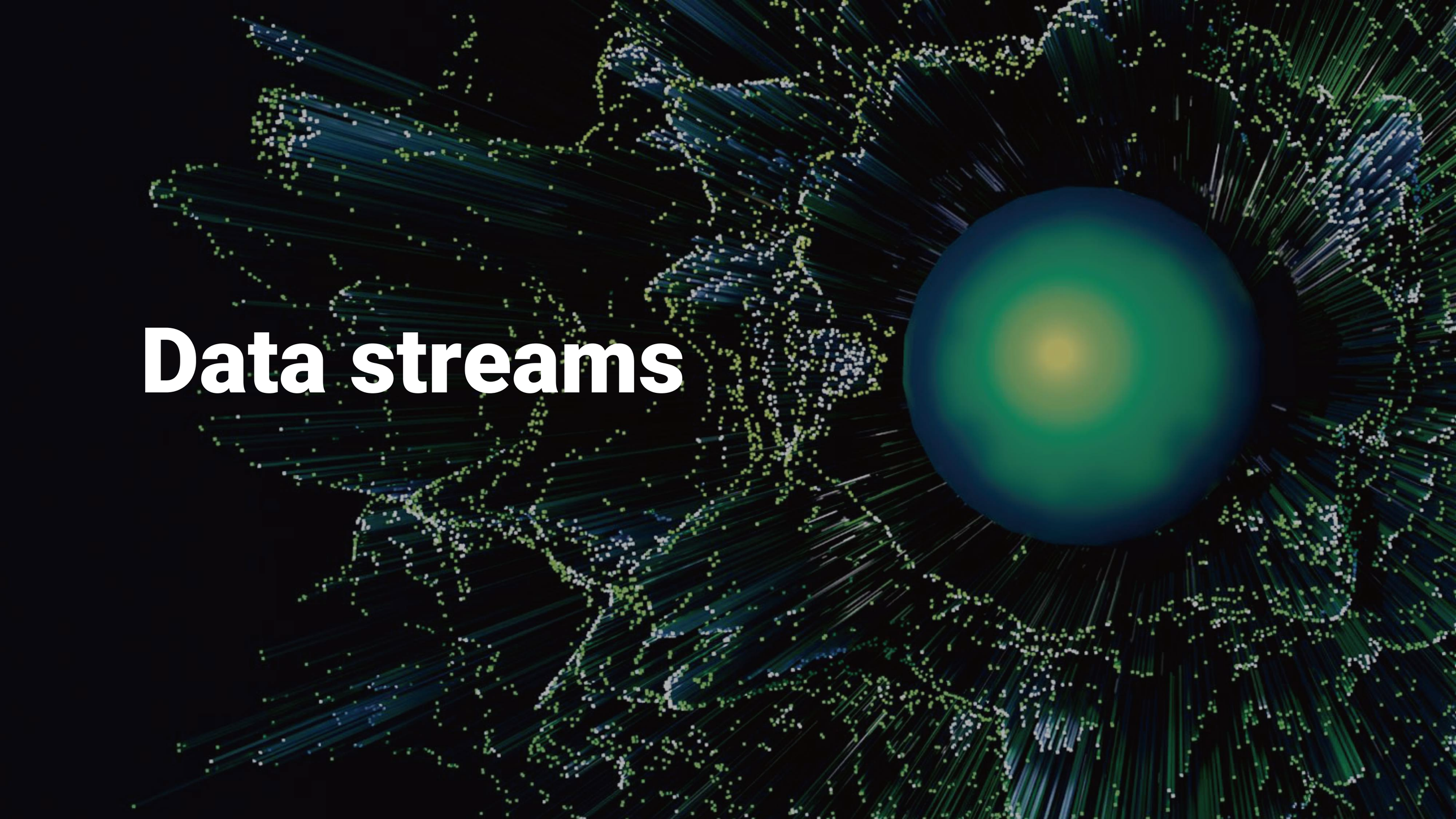
- Pravega is
  - A stream store: stream is the storage primitive
  - The foundation is segments
  - Segments enable a flexible composition of streams
  - Segments enable scaling, transactions, and state replication

- Pravega is open source

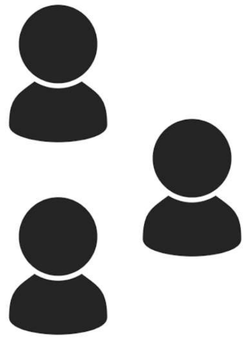
http://pravega.io
https://github.com/pravega/pravega

In this talk

- Data streams
- Introduction to Pravega
- Exactly-once ingestion with Pravega
- Code and demo


Data streams

Social networks
Online shopping



Stream of user events
- Status updates
- Online transactions

Server monitoring

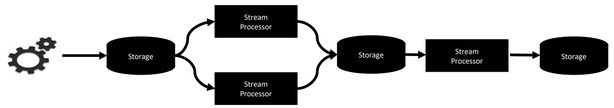
Stream of server events
- CPU, memory, disk utilization

Sensors (IoT)

Stream of sensor events
- Temperature samples
- Samples from radar and image sensors in cars

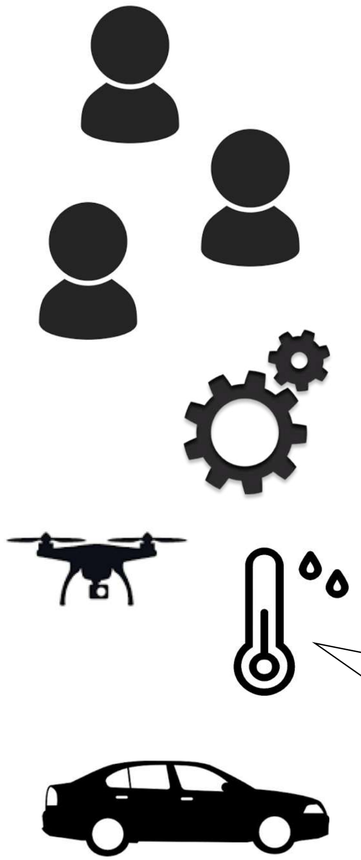
Landscape

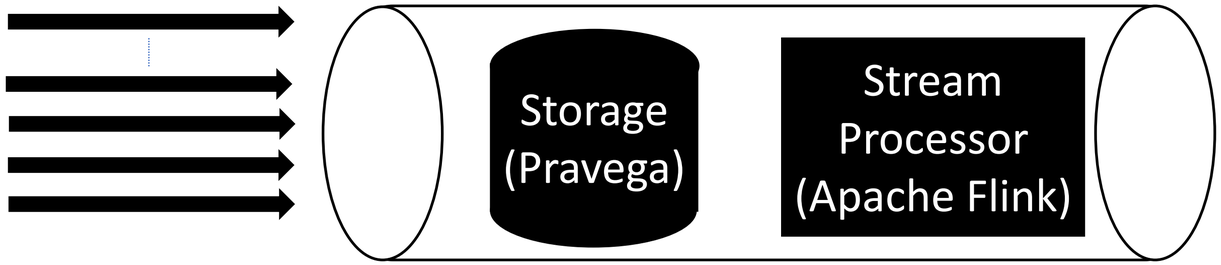
- Core elements
  - Storage (Pravega)
  - Stream processor (Apache Flink)
- Arbitrary Direct Acyclic Processing Graphs

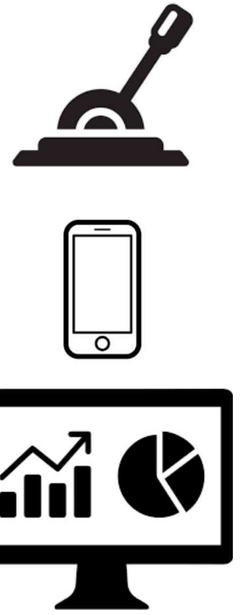


Continuous data flows

Data Pipelines







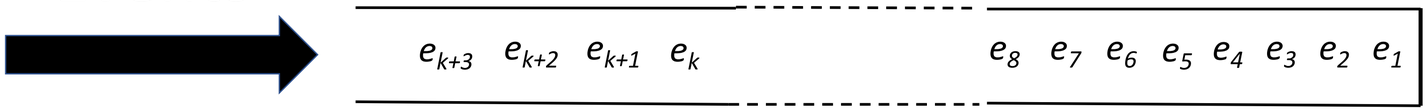
- Sources of continuous data flows:
  - User status
  - Online transactions
  - Server telemetry data
  - Sensor samples
  - Connected cars
  - Drone videos

- Visualization
- Alerts
- Near real-time insights
- Recommendations
- Actionable instructions

Data streams – Sequential

Server,
Sensor,
etc.

Events

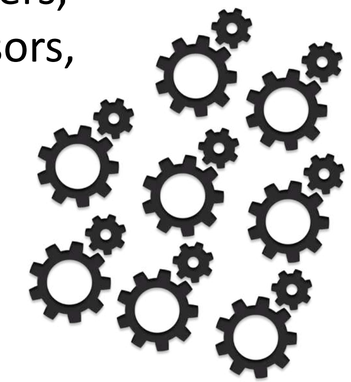


Tail

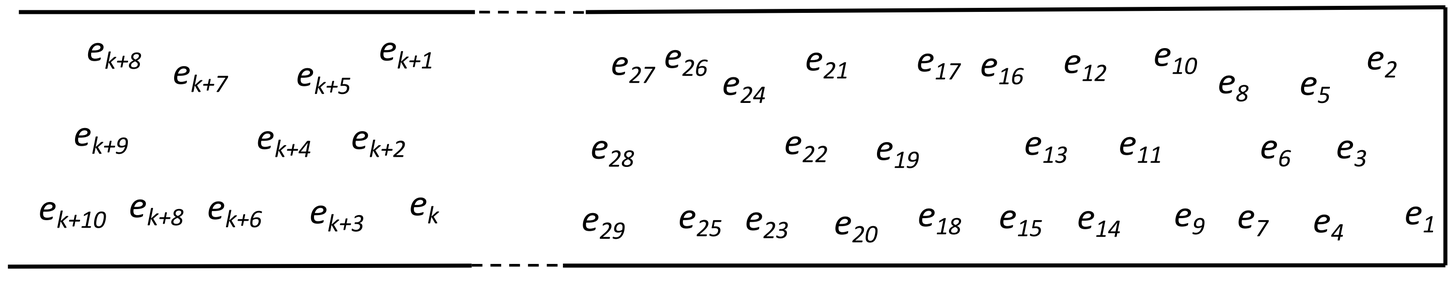
Head

Data streams - Parallelism

Servers,
Sensors,
etc.



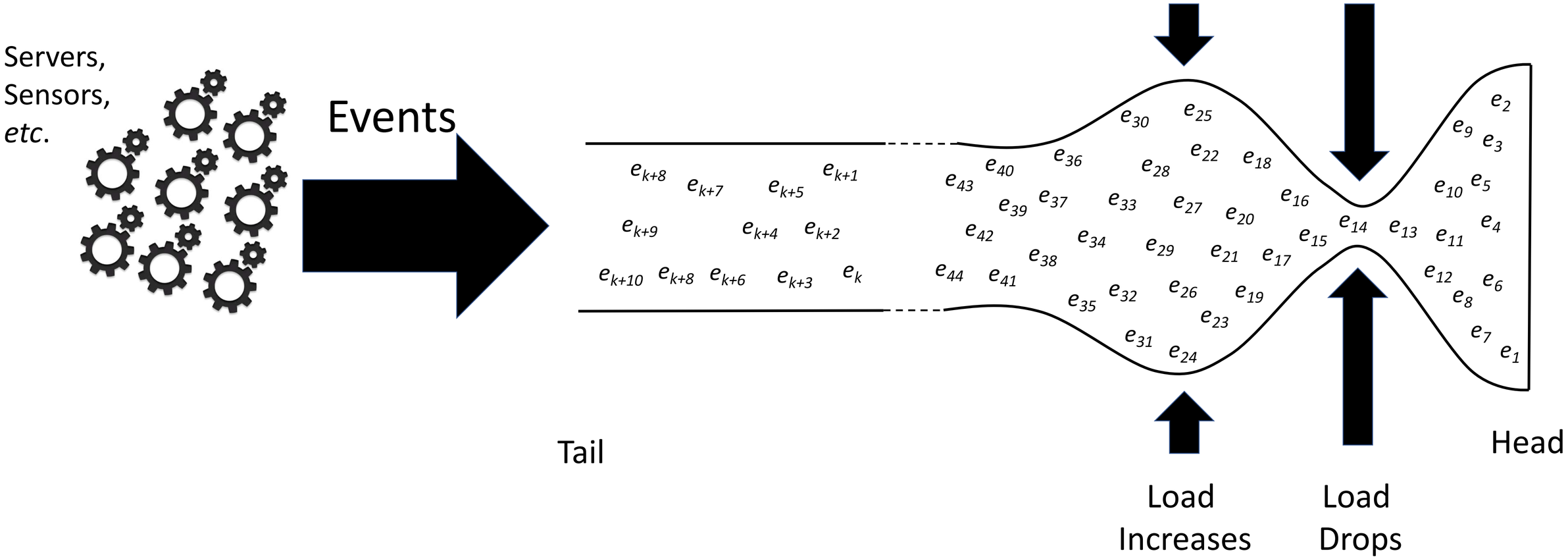
Events



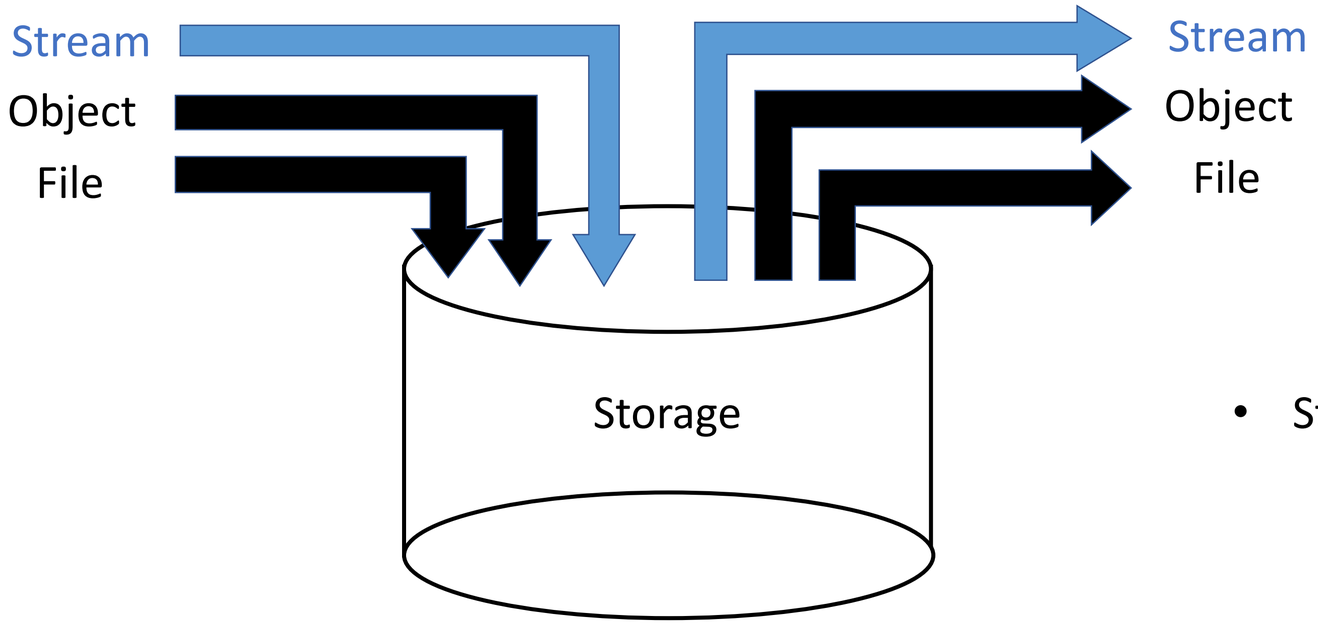
Tail

Head

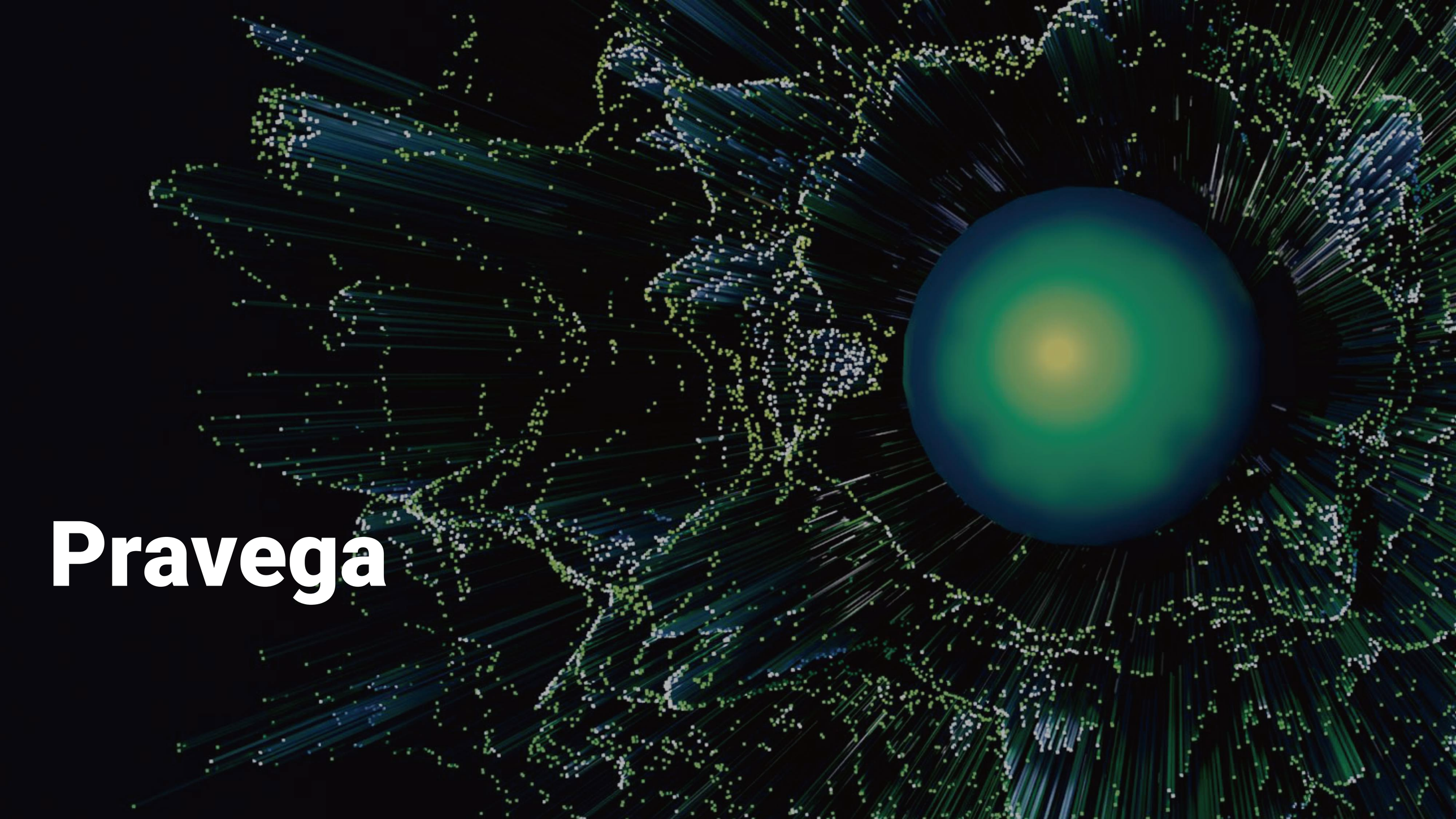
Data streams – Traffic fluctuations



Pravega: Stream as a storage primitive

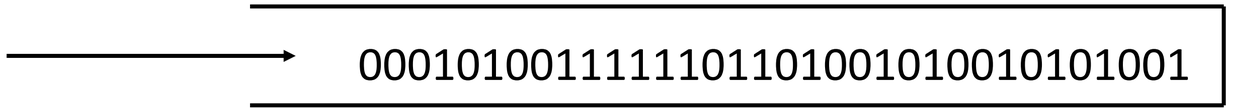


- Stream
  - Continuous flow of data
  - Does not naturally match existing storage abstractions
- Data analytics
  - **Tailing**: low-latency end-to-end
  - **Historical**: some arbitrary time in the future



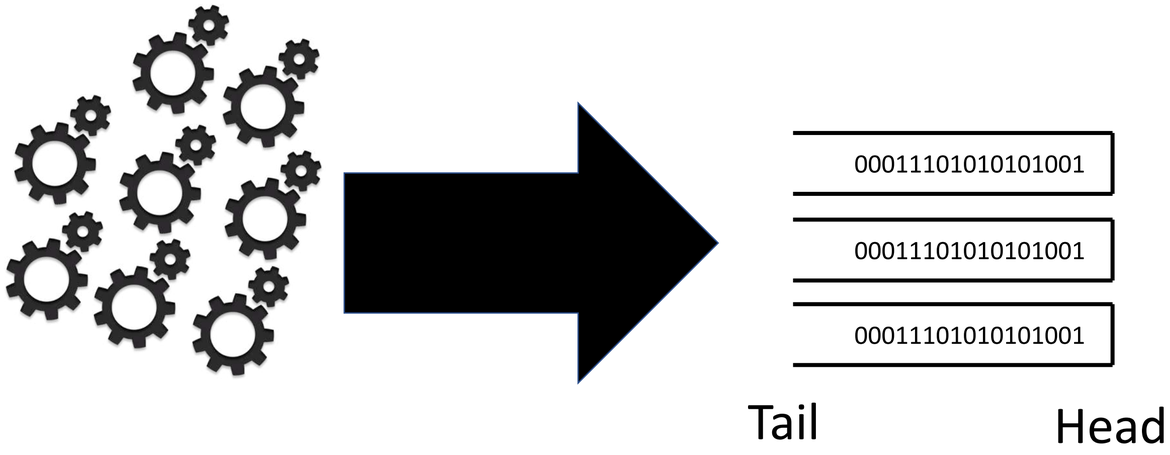
Pravega

Stream Segment



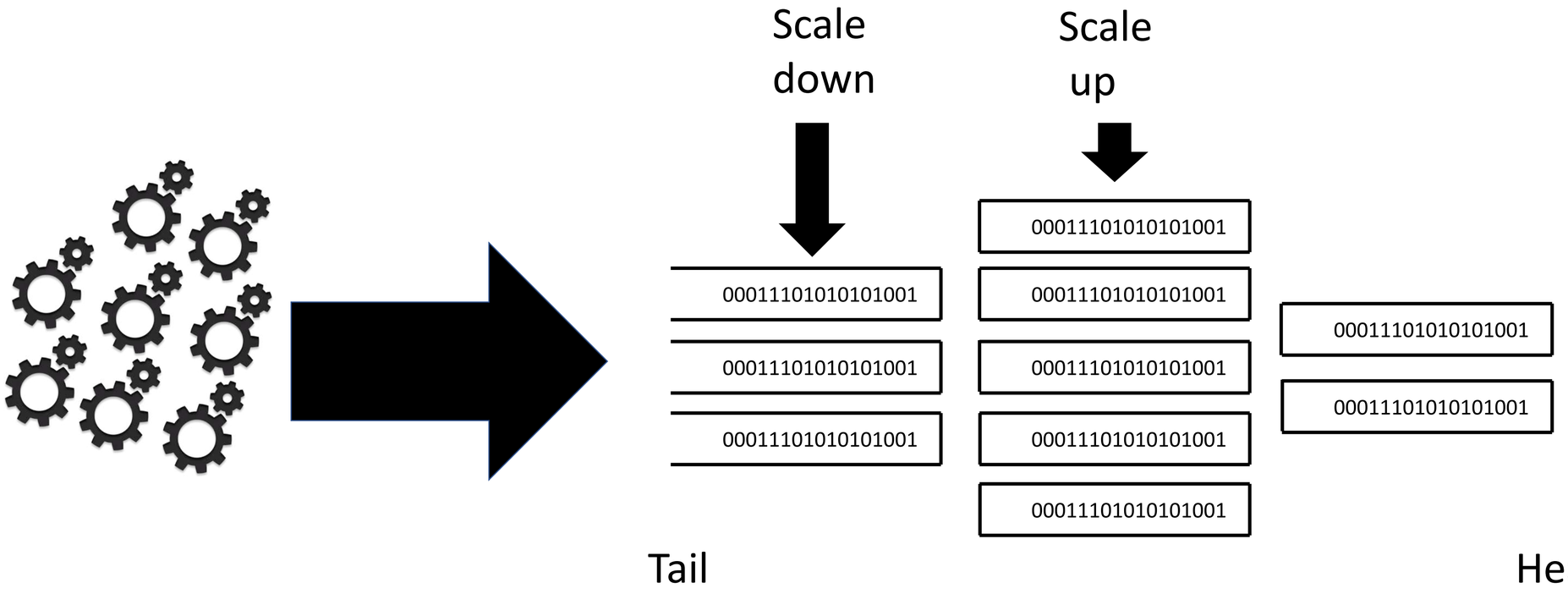
- Storage unit
- Append-only sequences of bytes
- … bytes, not events, messages or records
- Flexible in the formats it can store
- Events: expect serializer implementation

Stream Segment – Parallelism



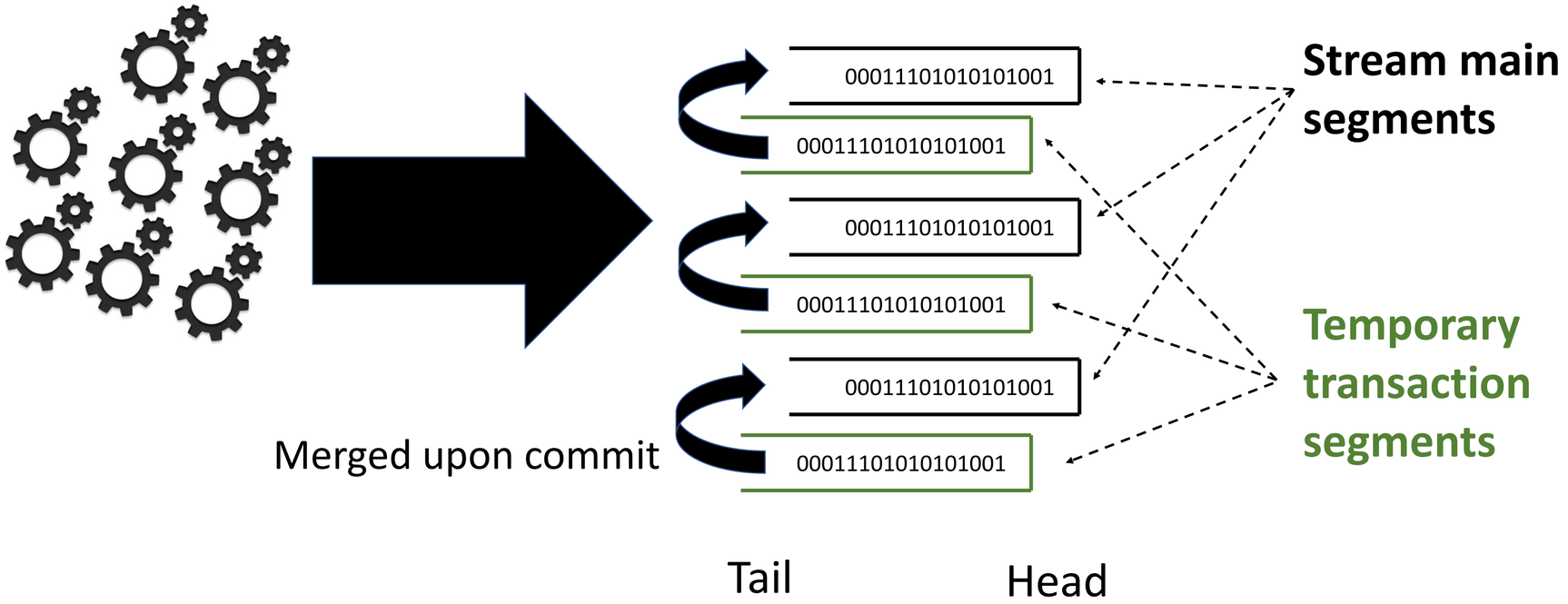
- Sources append to segments in parallel
- Routing keys map appends to segments

Stream Segment – Scaling

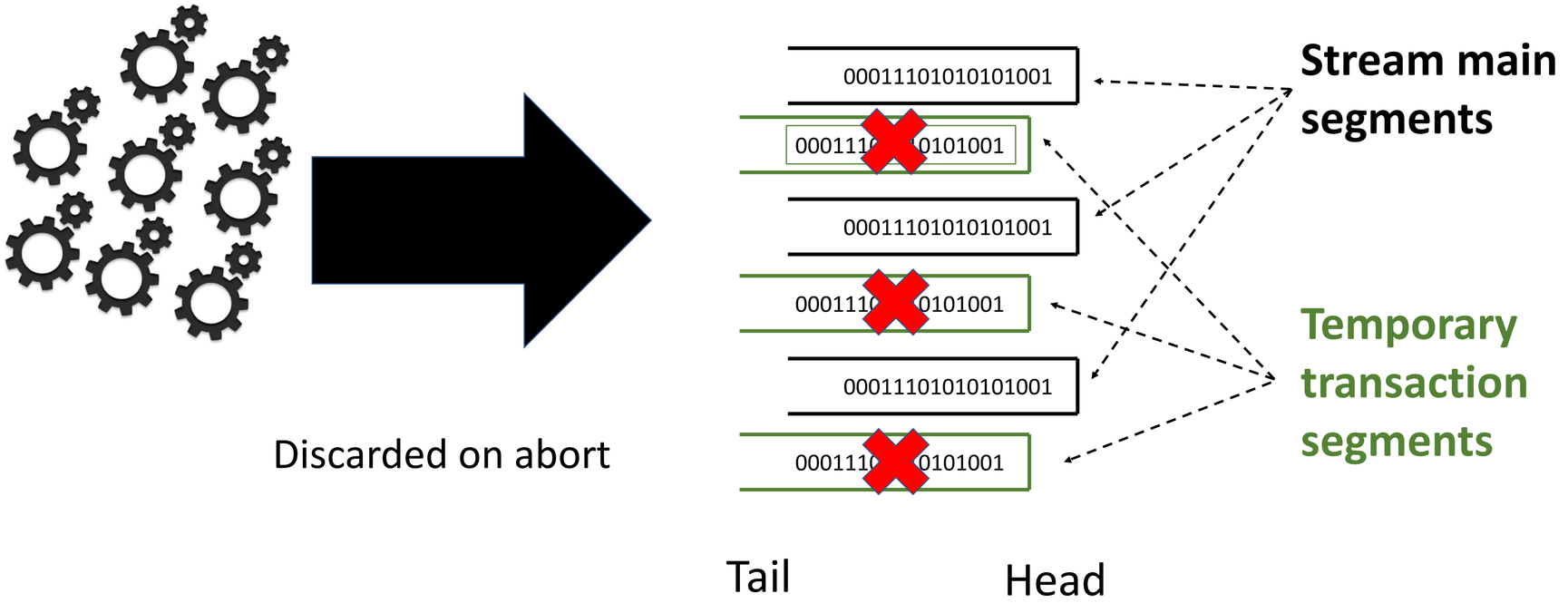


- Degree of parallelism changes dynamically
- Auto-scaling: reacts to workload changes

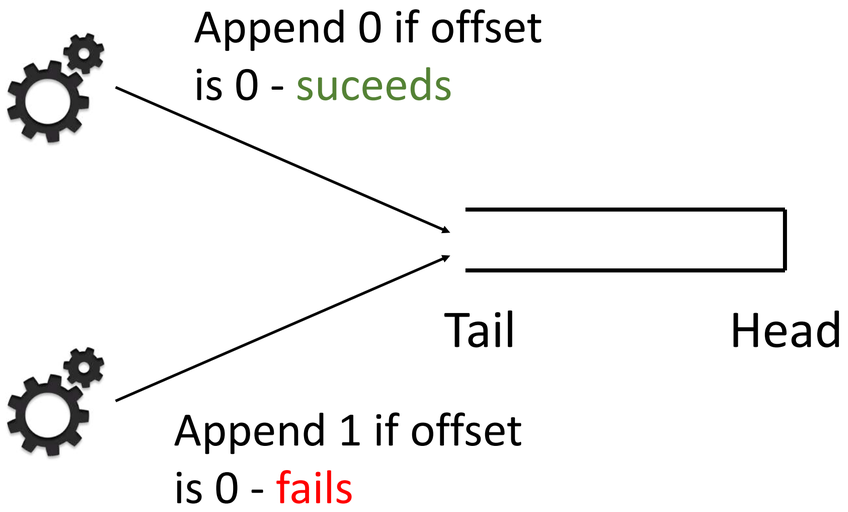
Stream Segment - Transactions



Stream Segment - Transactions



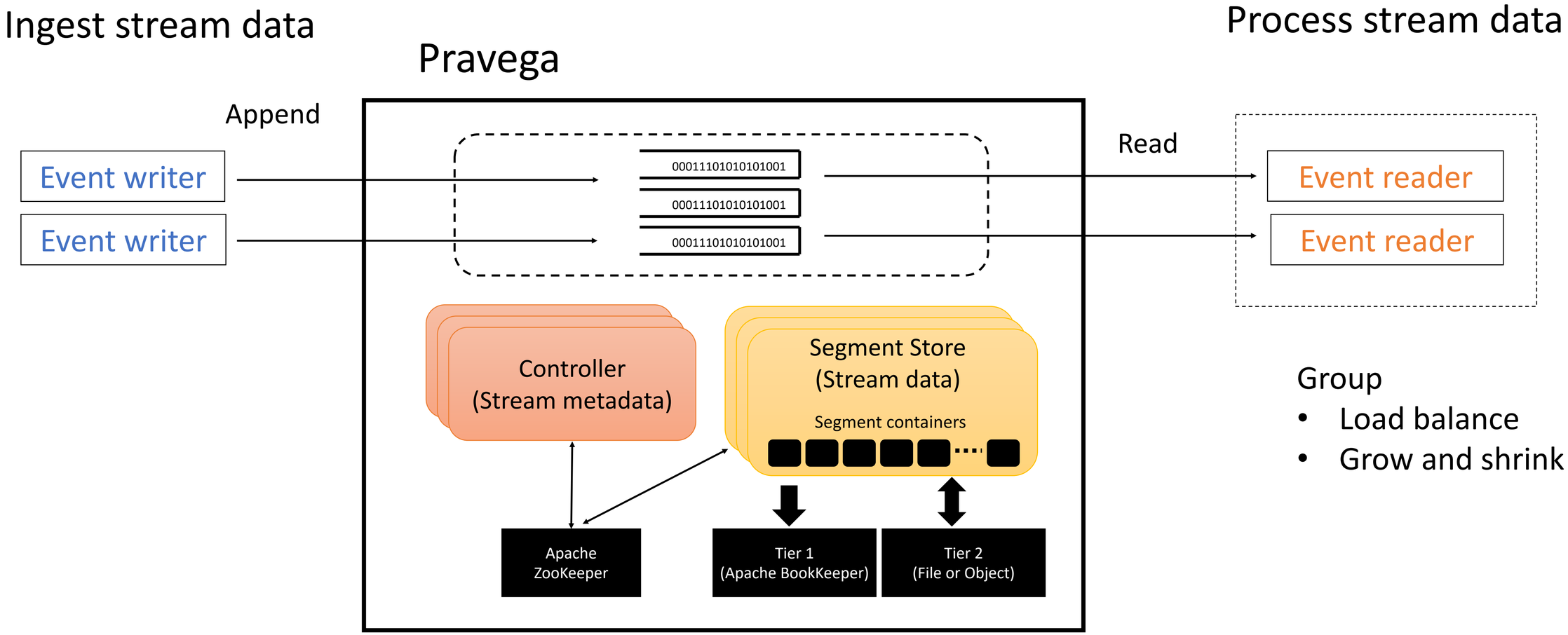
Stream Segment – Appending conditionally



- Revisioned streams
  - Conditional appends
    - Compare expected and current revisions
    - Revision implementation uses segment offset
- State synchronizer
  - Builds on revisioned streams
  - Replicated state machines
  - Optimistic concurrency

Pravega Architecture

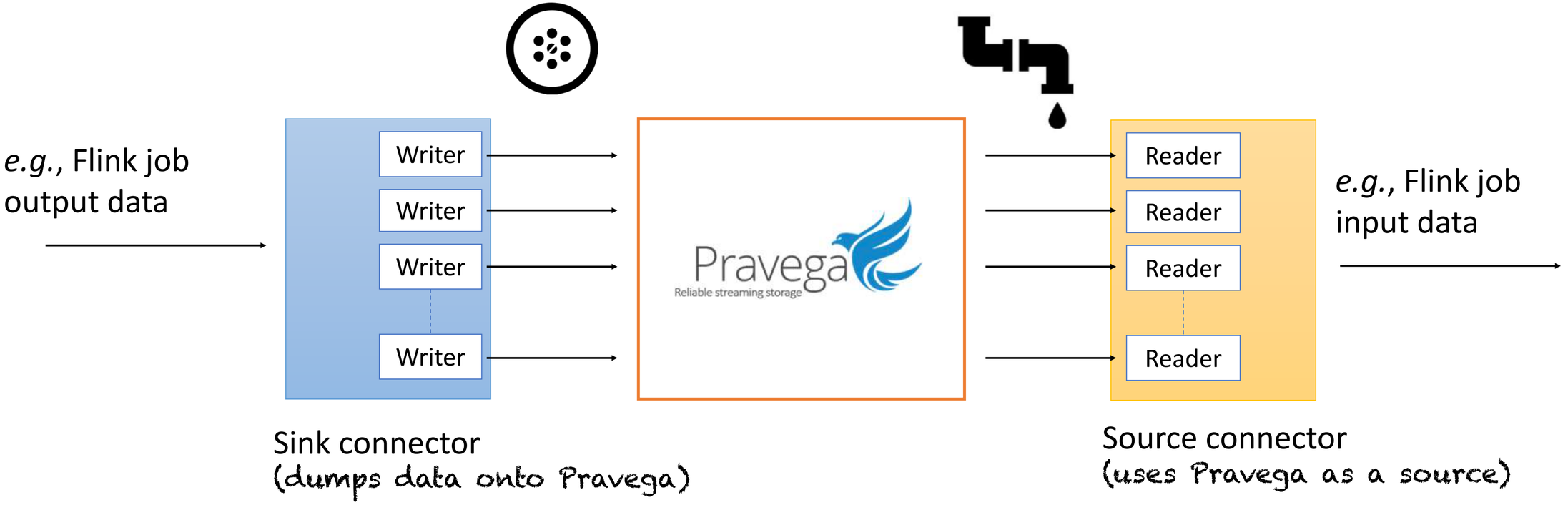
Pravega and Streams



Group
- Load balance
- Grow and shrink

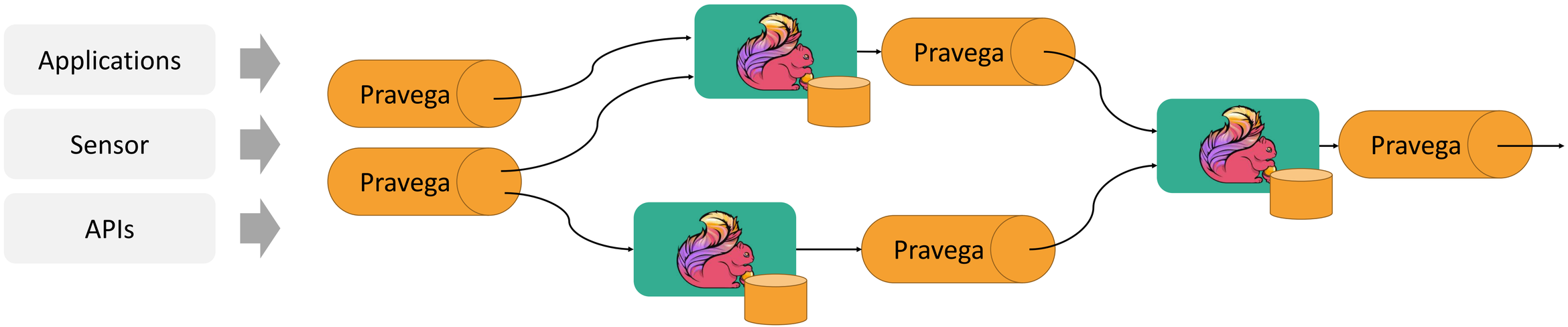
Flink Connector

Connectors



https://github.com/pravega/flink-connectors

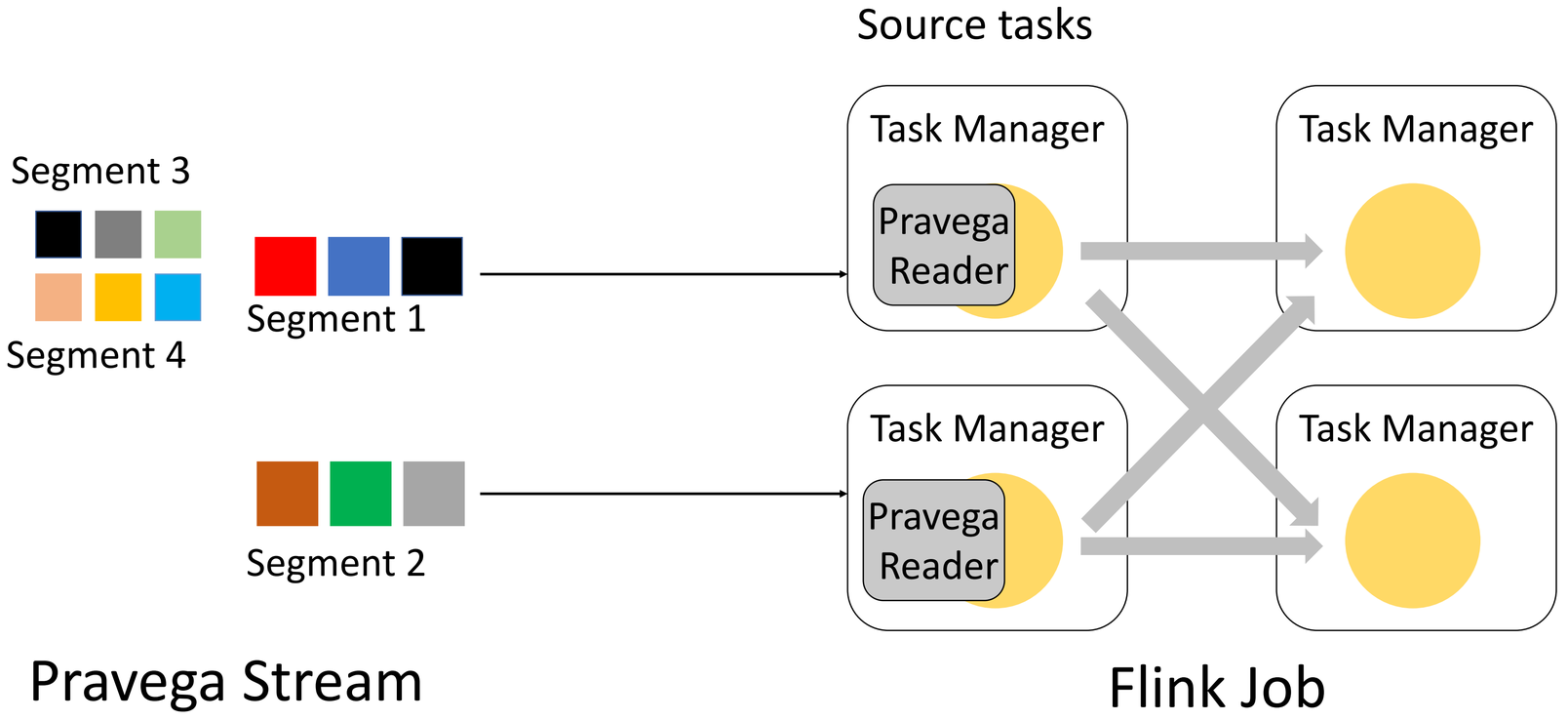
Streaming Pipelines



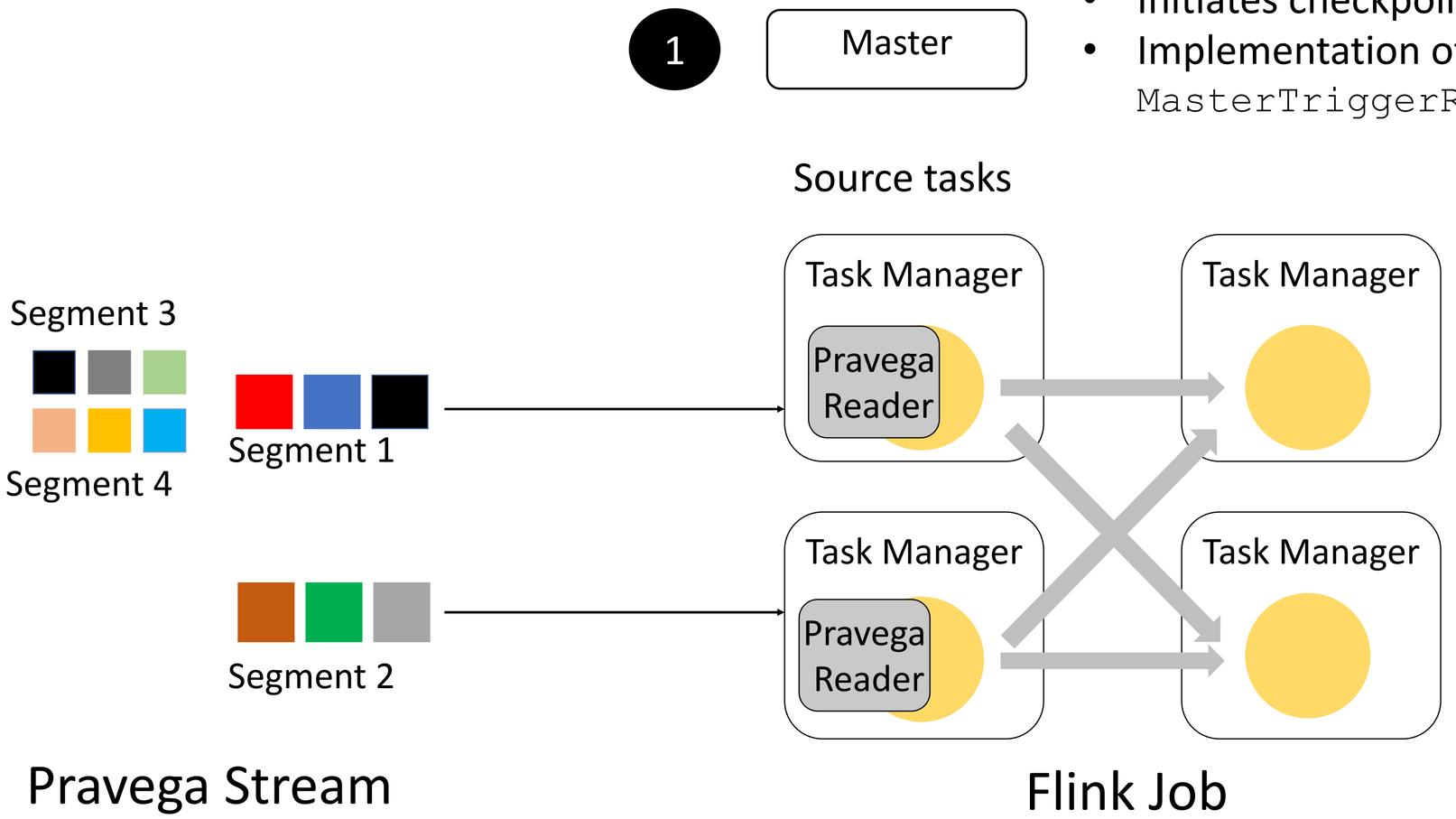
Flink reading from Pravega

Reading via Reader Group

- Reader group automatically assigns and re-balances segments
  - Hides complexity from app
  - Stream scaling



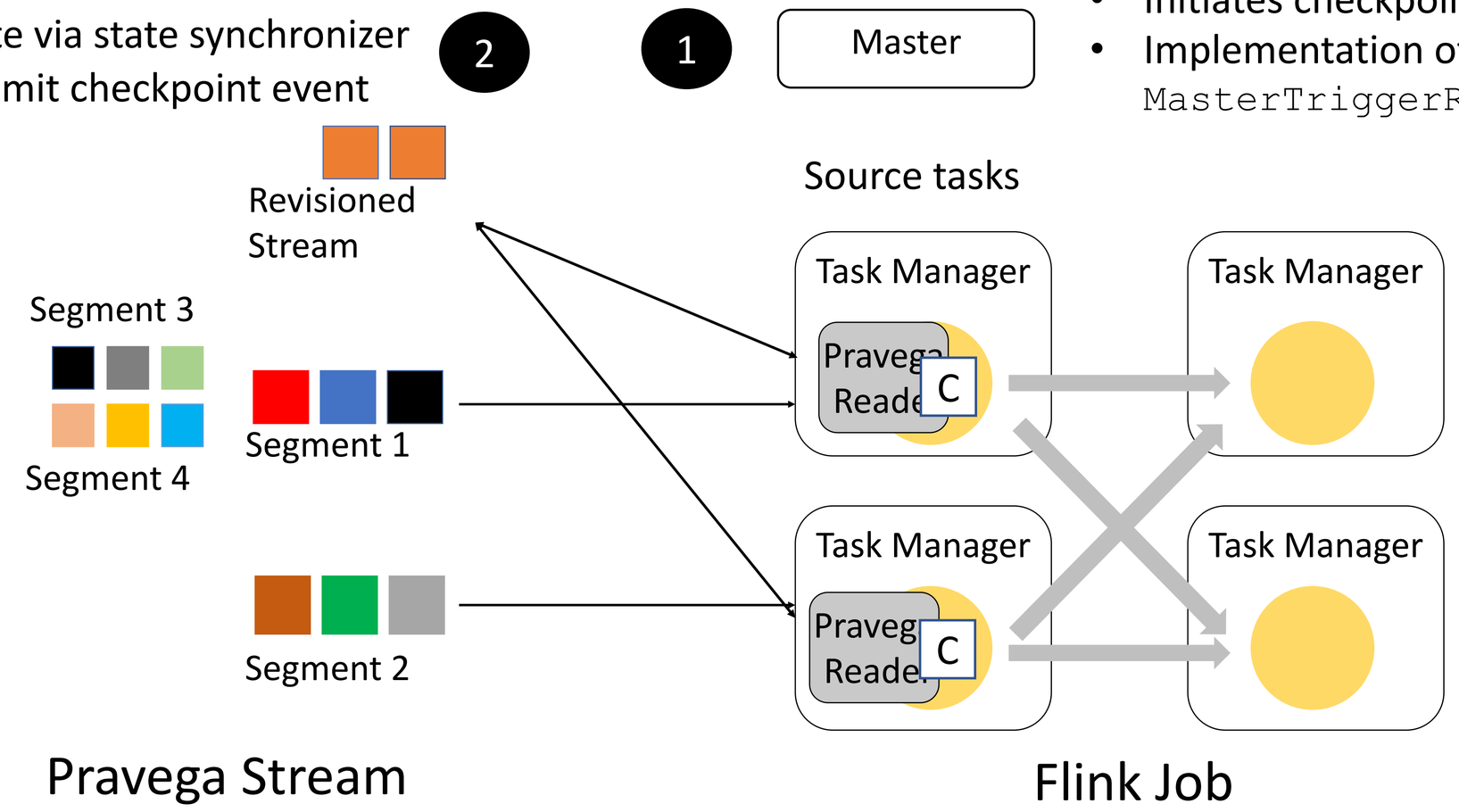
Upon a Flink checkpoint



- Initiates checkpoint
- Implementation of `MasterTriggerRestoreHook`

Upon a Flink checkpoint

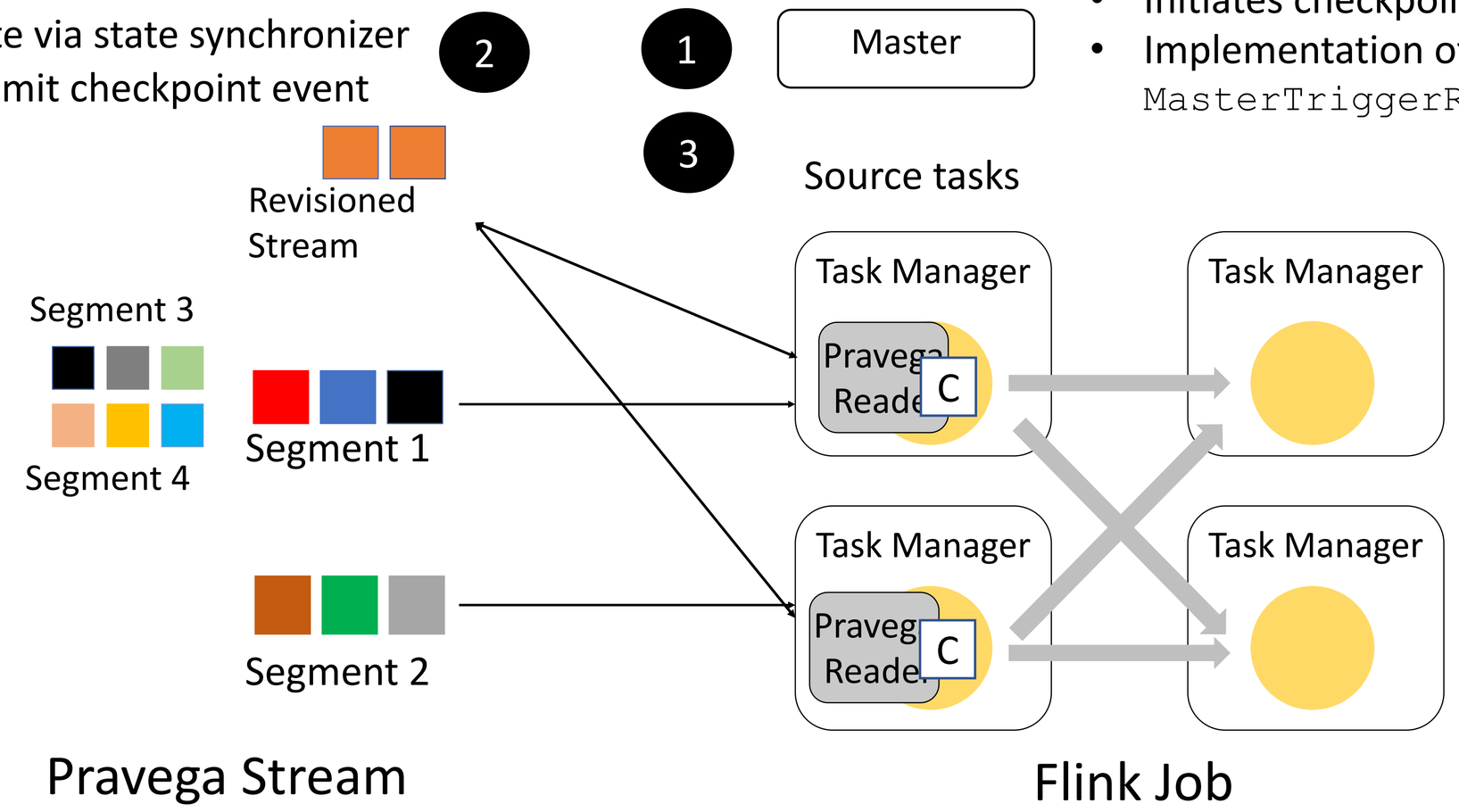
- Coordinate via state synchronizer
- Readers emit checkpoint event



- Initiates checkpoint
- Implementation of `MasterTriggerRestoreHook`

Upon a Flink checkpoint

- Coordinate via state synchronizer
- Readers emit checkpoint event



- Initiates checkpoint
- Implementation of `MasterTriggerRestoreHook`

Creating a Pravega source

```
// create the Pravega source to read a stream of text
FlinkPravegaReader<String> source = FlinkPravegaReader.<String>builder()
                .withPravegaConfig(pravegaConfig)
                .forStream(stream)
                .withDeserializationSchema(PravegaSerialization.deserializationFor(String.class))
                .build();
```

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

Creating a Pravega source (and using it)

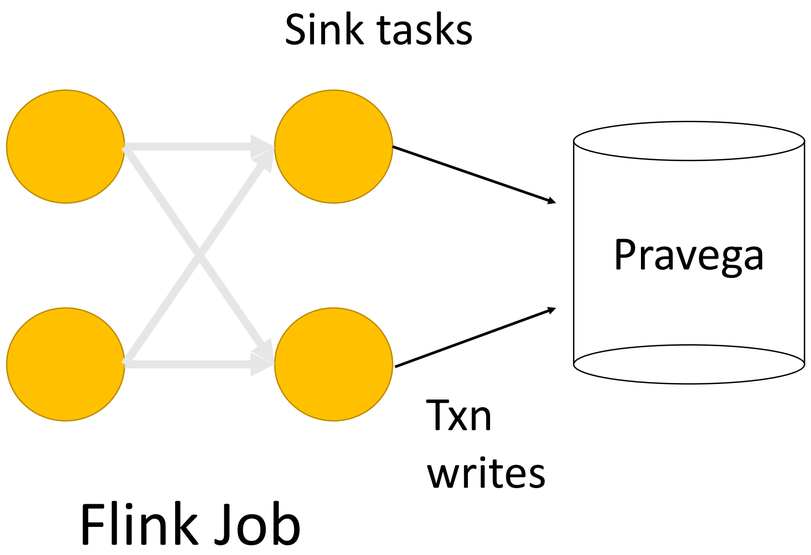
```
// create the Pravega source to read a stream of text
FlinkPravegaReader<String> source = FlinkPravegaReader.<String>builder()
                .withPravegaConfig(pravegaConfig)
                .forStream(stream)
                .withDeserializationSchema(PravegaSerialization.deserializationFor(String.class))
                .build();
```

```
// count each word over a 10 second time period
DataStream<WordCount> dataStream = env.addSource(source).name("Pravega Stream")
                .flatMap(new WordCountReader.Splitter())
                .keyBy("word")
                .timeWindow(Time.seconds(10))
                .sum("count");
```

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

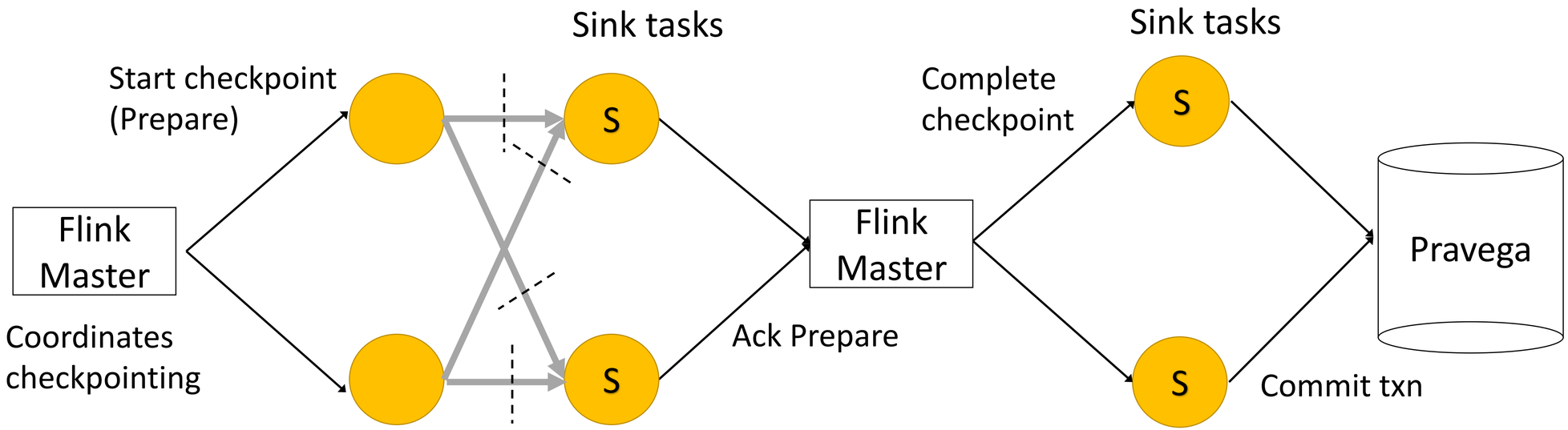
Flink writing to Pravega

Exactly-once with Transactions



- Transactional writes for job output
- Executes a 2PC to commit results
- Option to not use transactions
  - At-least-once semantics

Exactly-once with Transactions



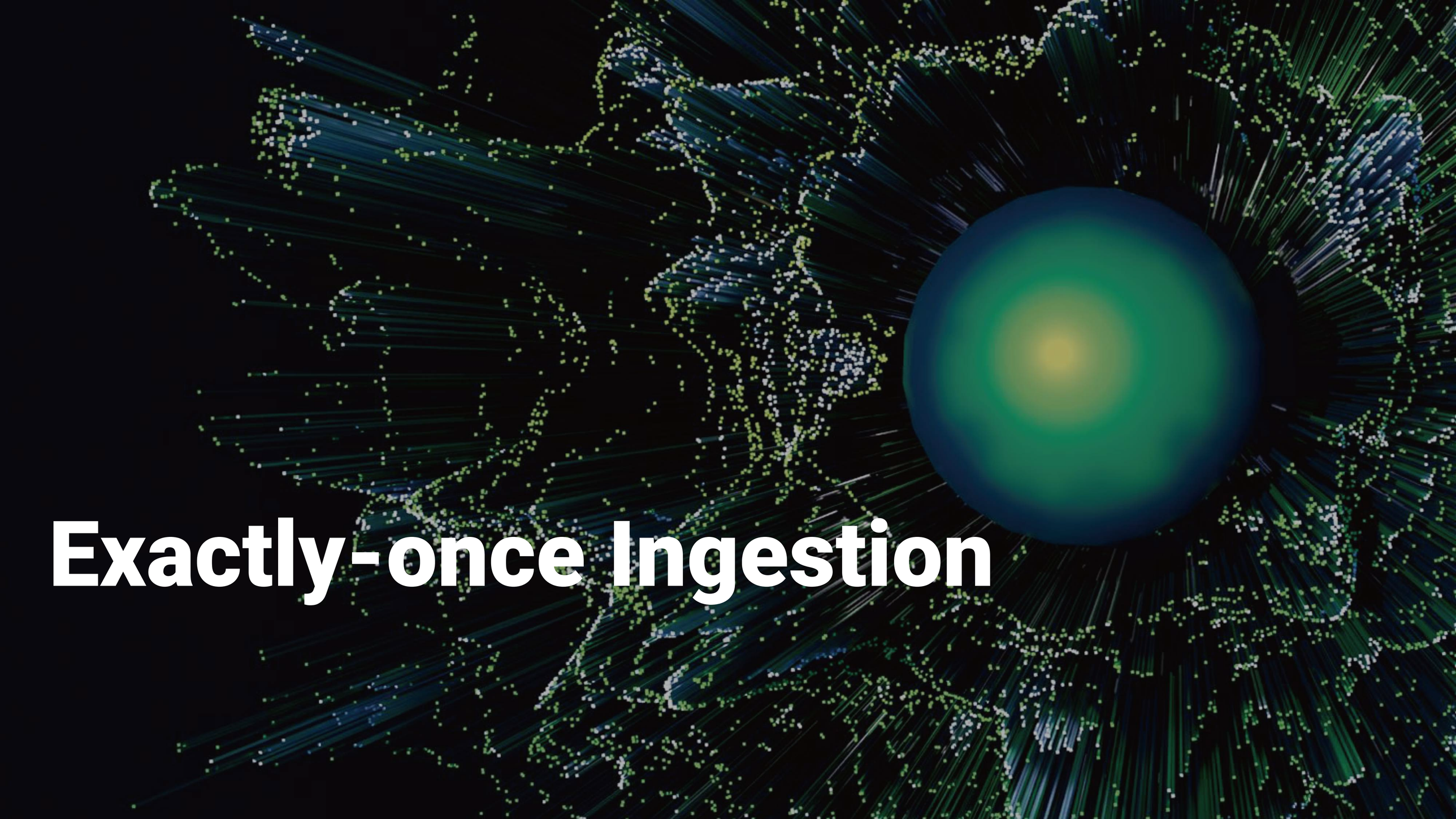
2-Phase commit protocol

Creating a Pravega sink

```
// create the Pravega sink to write a stream of text
FlinkPravegaWriter<String> writer = FlinkPravegaWriter.<String>builder()
                .withPravegaConfig(pravegaConfig)
                .forStream(stream)
                .withEventRouter(new EventRouter())
                .withWriterMode(PravegaWriterMode.EXACTLY_ONCE)
                .withSerializationSchema(PravegaSerialization.serializationFor(String.class))
                .build();
dataStream.addSink(writer).name("Pravega Stream");
```

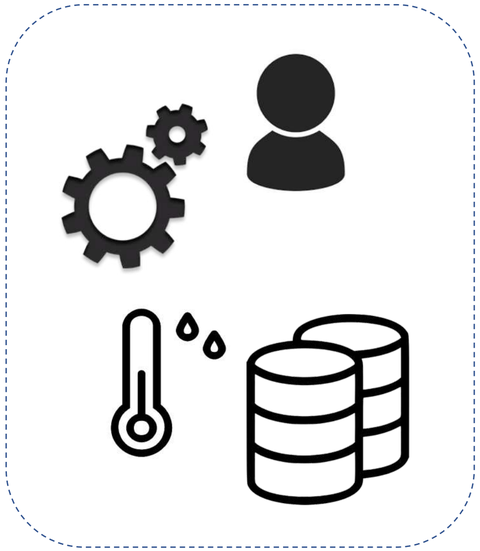
https://github.com/pravega/pravega-samples

Recap time



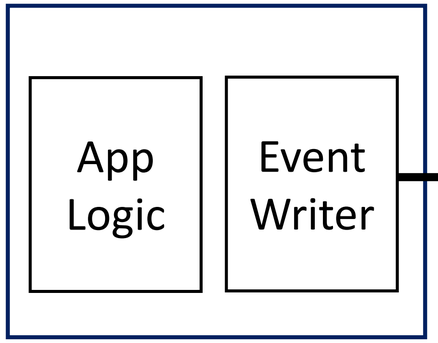
Exactly-once Ingestion

Data source



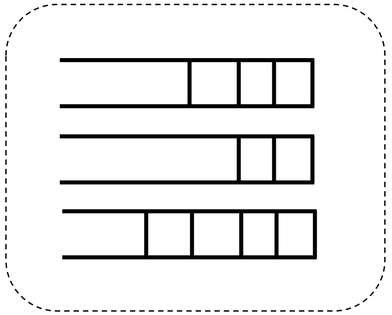
Emit event

Application



Append event to stream

Pravega stream



- A data source emits events to be ingested
- The source sends events to an application that is capable of writing to a Pravega stream using an event writer
- There can be multiple instances of a given data source
- *E.g.*, end users, sensors, servers

- Application receives events from the source and writes them to a Pravega stream
- *E.g.*, IoT Gateway

- Pravega stream ingests data from the data source

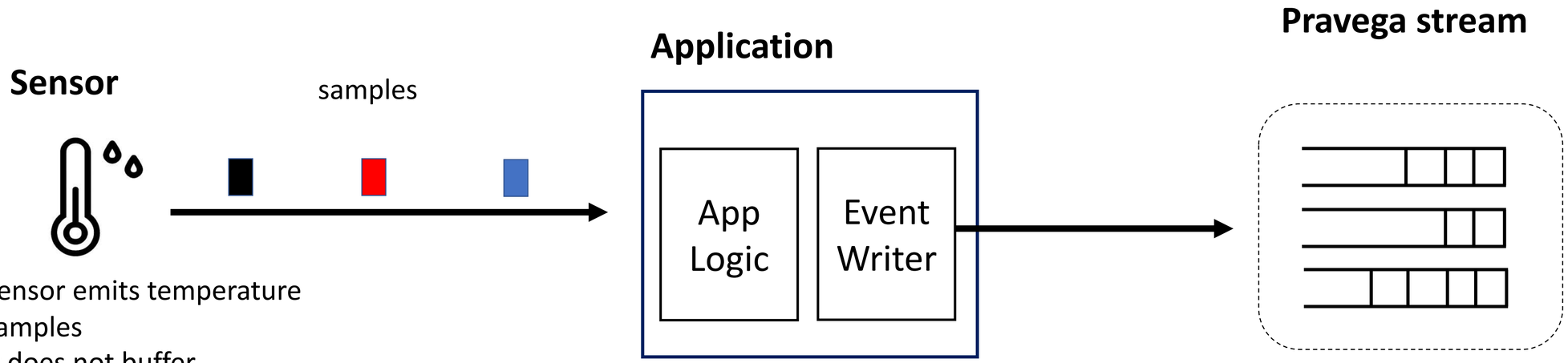
Data source types

Data sources

- Memoryless
  - Data source is not capable of retransmitting events
  - Cannot guarantee exactly-once semantics for such sources in the presence of process crashes
  - *E.g.*, stateless sensors
- Memoryful
  - Data source is capable of retransmitting from any given offset
  - Possibly bounded buffering
  - *E.g.*, files, Flink job

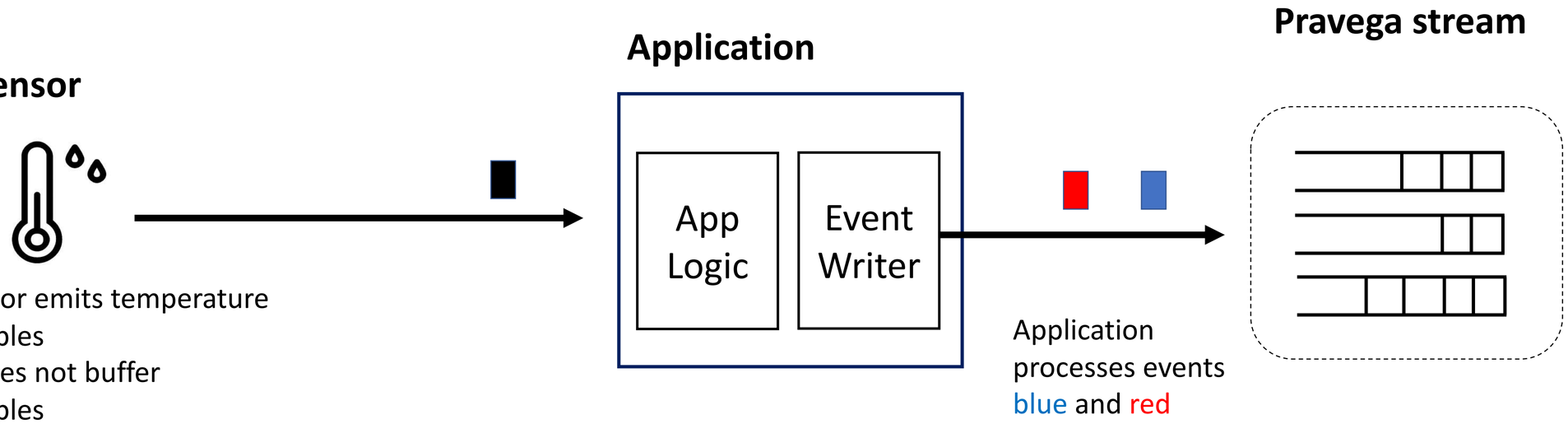
Memoryless sources

Memoryless sources



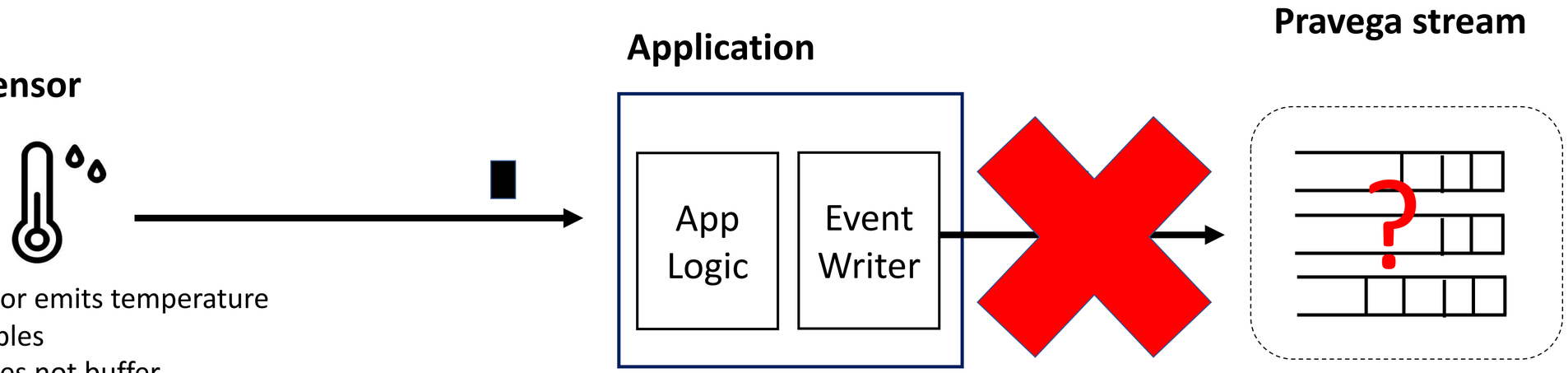
- Sensor emits temperature samples
- It does not buffer samples

Memoryless sources



- Sensor emits temperature samples
- It does not buffer samples

Memoryless sources

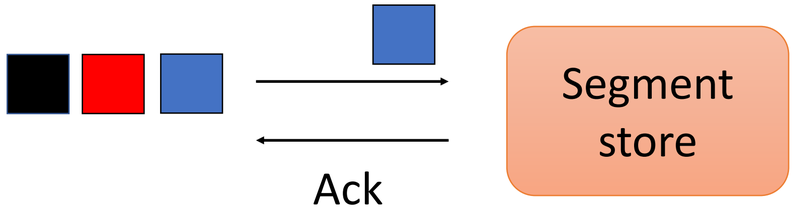


- Sensor emits temperature samples
- It does not buffer samples

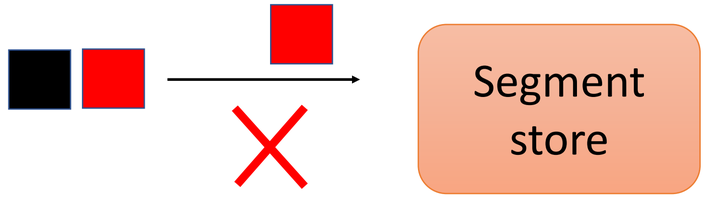
- Connection breaks
- Have blue and red made it?

Under the hood – Avoiding duplicates

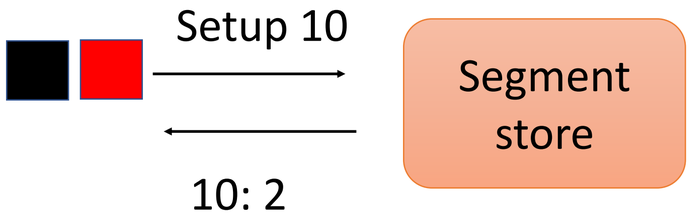
1 Writer ID = 10



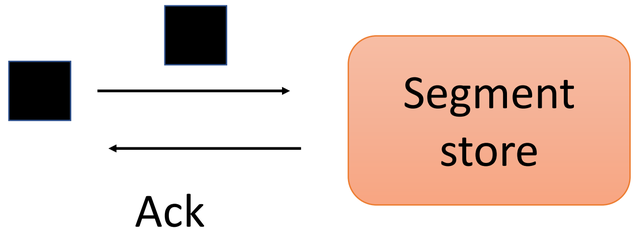
2 Writer ID = 10



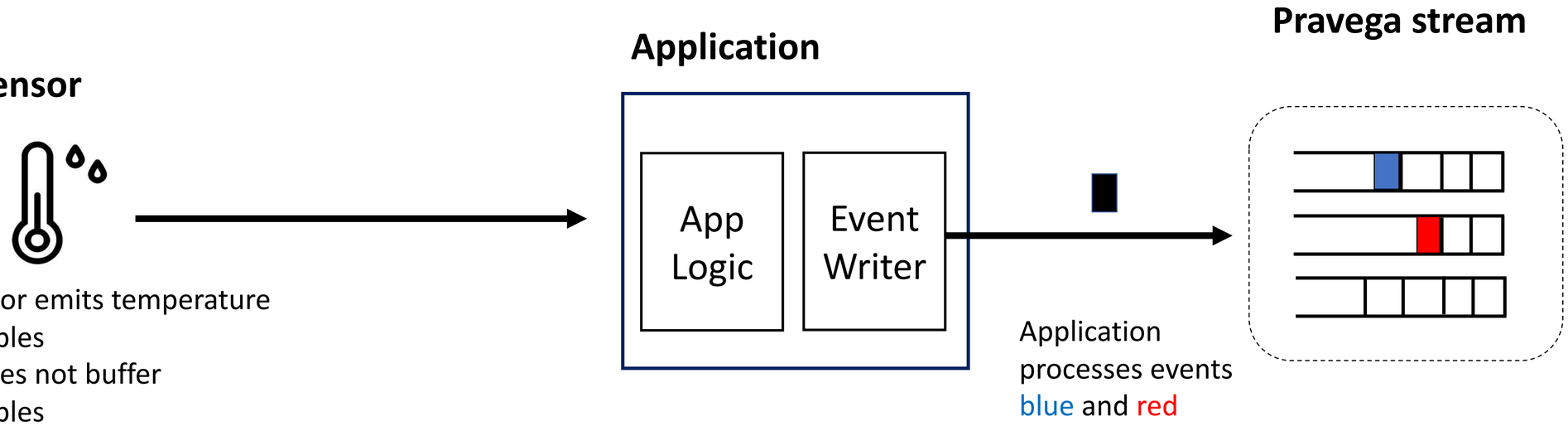
3 Writer ID = 10



4 Writer ID = 10



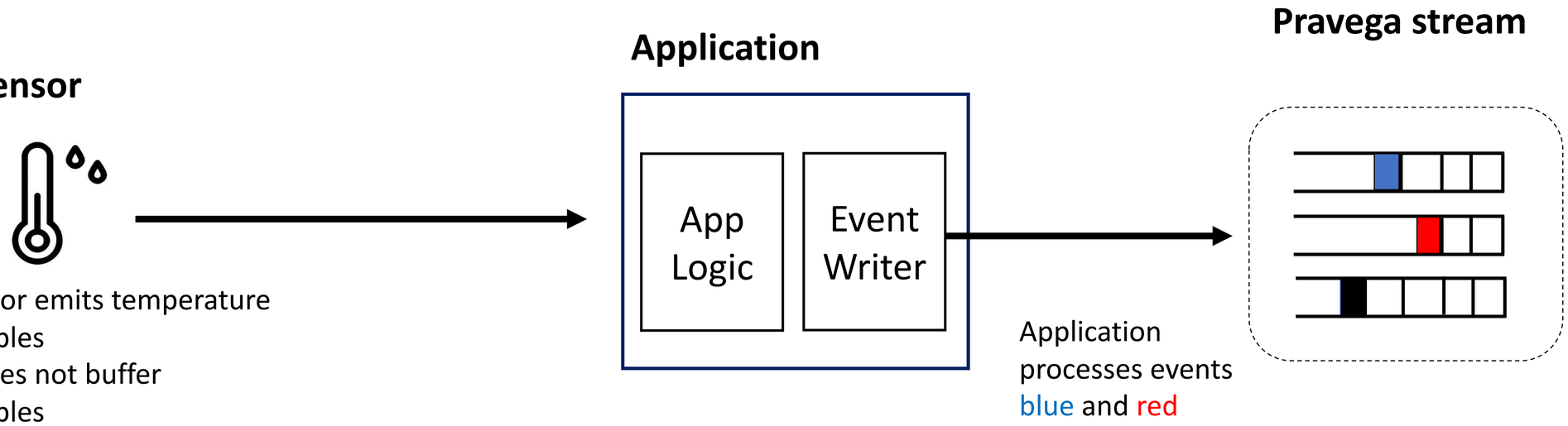
Memoryless sources



- Sensor emits temperature samples
- It does not buffer samples

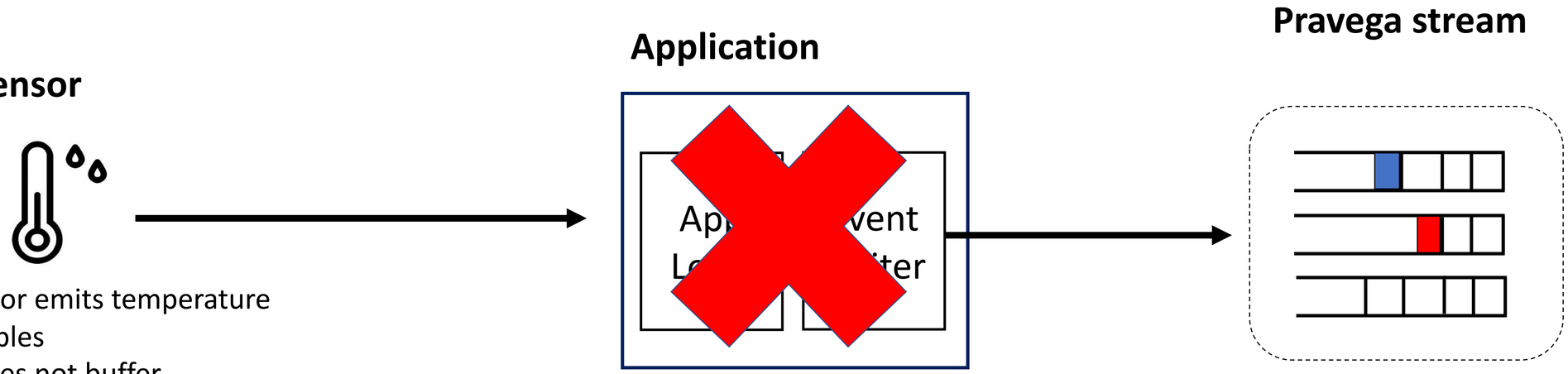
Pravega writer appends the black event

Memoryless sources



- Sensor emits temperature samples
- It does not buffer samples

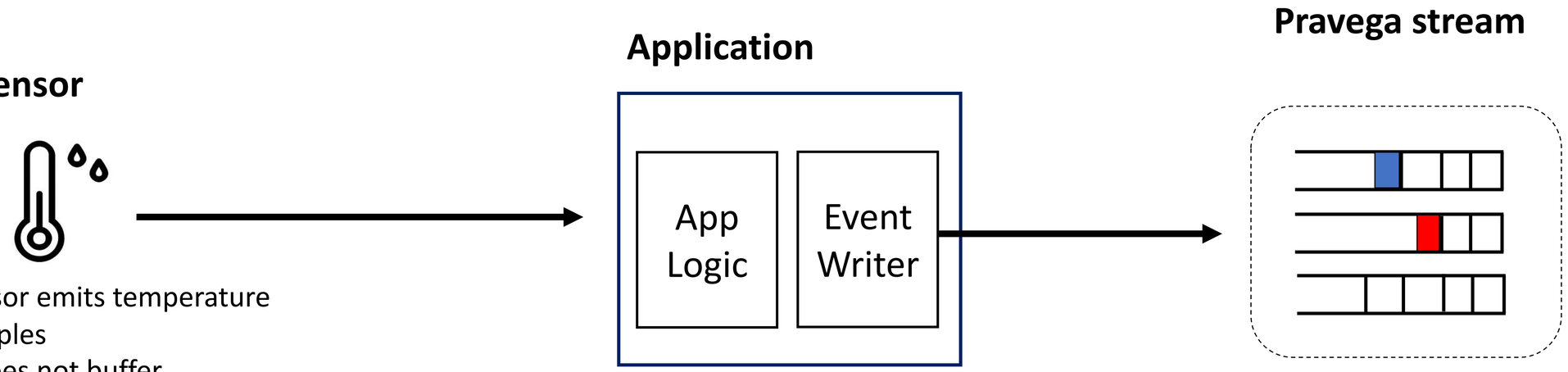
Memoryless sources



- Sensor emits temperature samples
- It does not buffer samples

- Application crashes

Memoryless sources



- Sensor emits temperature samples
- It does not buffer samples

- Application recovers
- Black event has not been processed
- Sensor can’t retransmit it by design

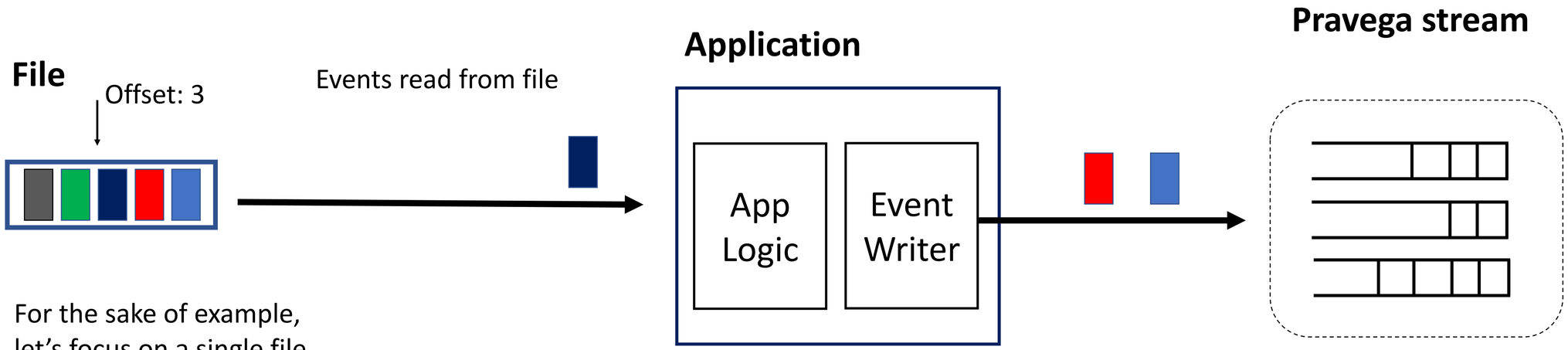
Memoryful sources

Memoryful sources

- Let’s assume that such a source is capable of rewinding
  - Based on some notion of offset
- Simple example of such a source
  - One or more files
  - Offset is a pair ⟨`file name, file offset`⟩
- Complex example of such a source
  - Flink job
  - Retransmitting might require reprocessing data
  - Offset is a job snapshot

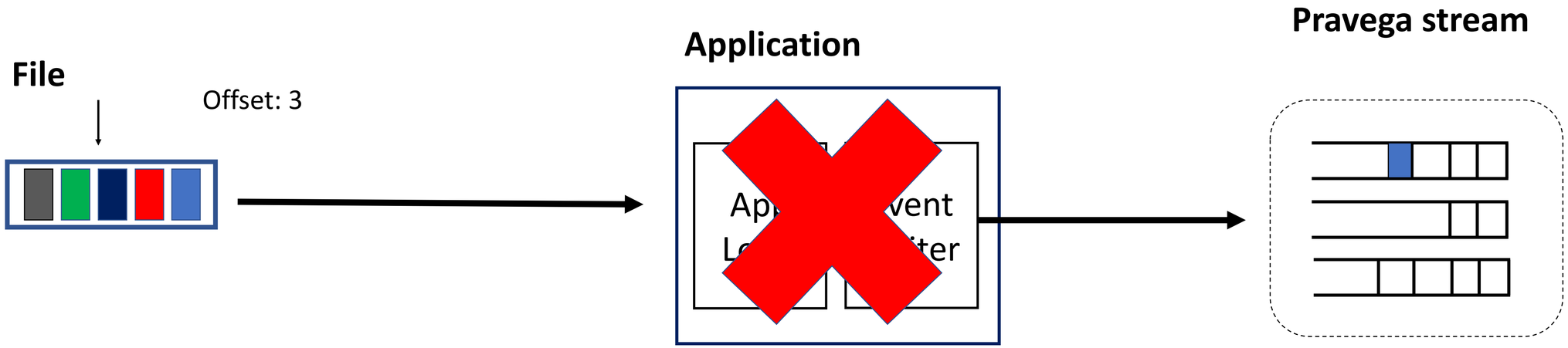
- For the sake of example, let’s focus on files as a data source

File source



- For the sake of example, let’s focus on a single file

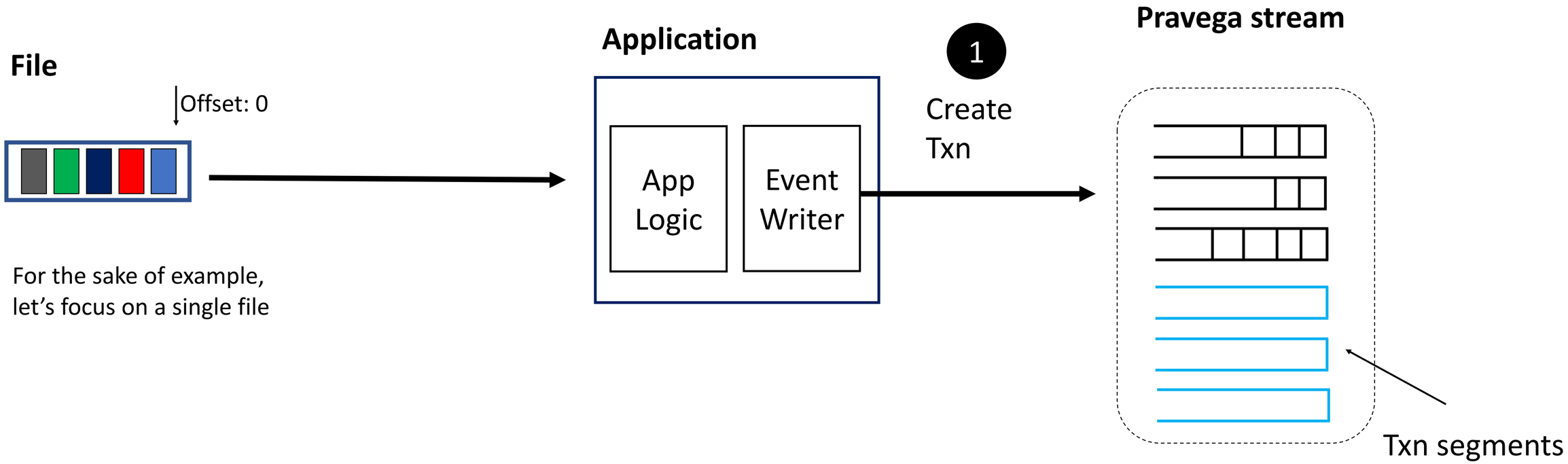
File source



- Application crashes
- **Which events have been successfully appended to the stream?**

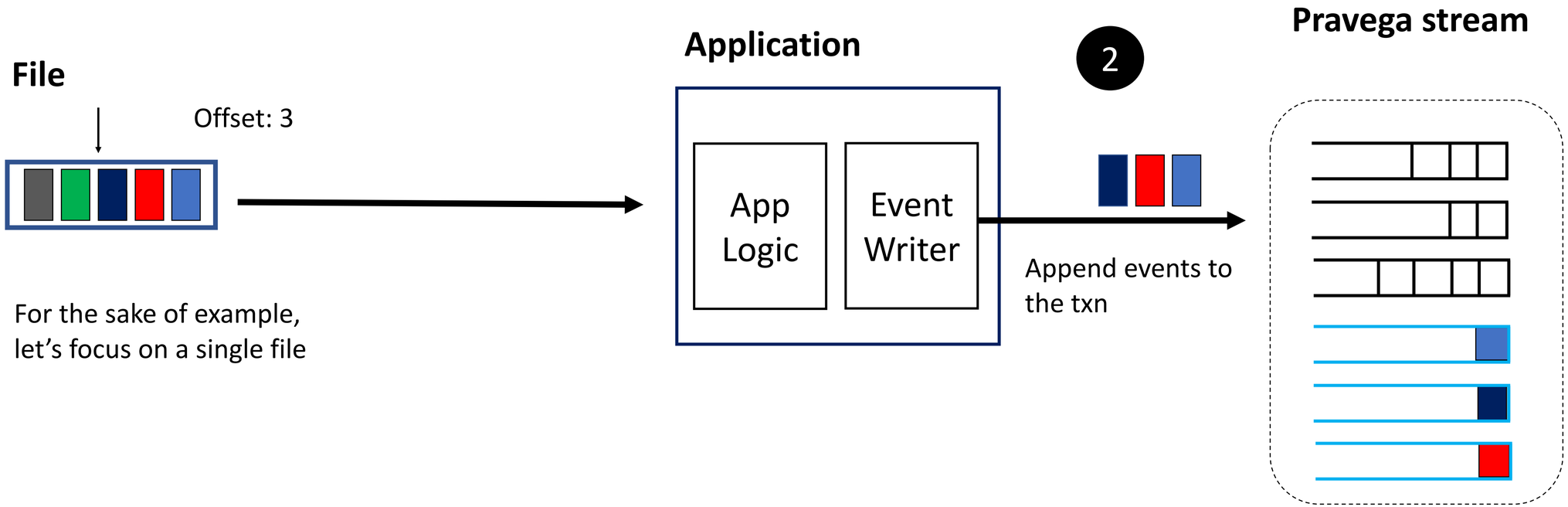
Introducing transactions

File source



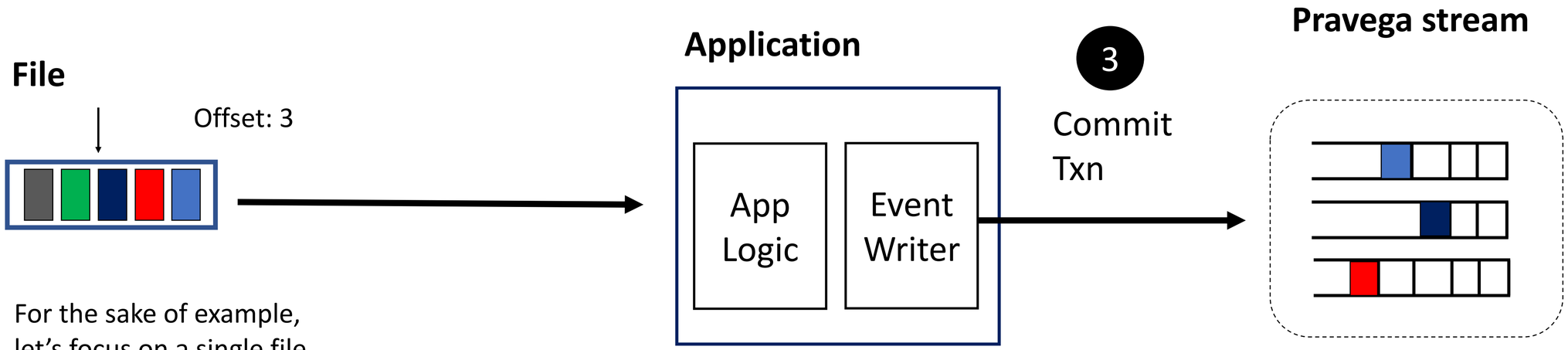
- For the sake of example, let's focus on a single file

File source



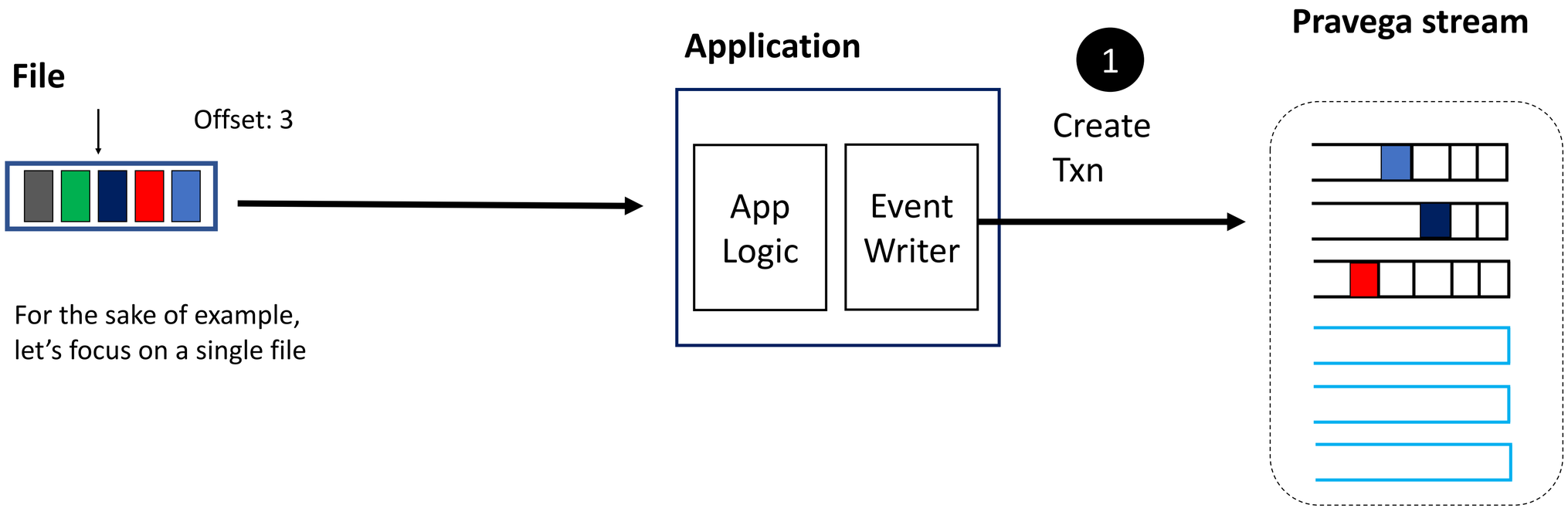
- For the sake of example, let’s focus on a single file

File source



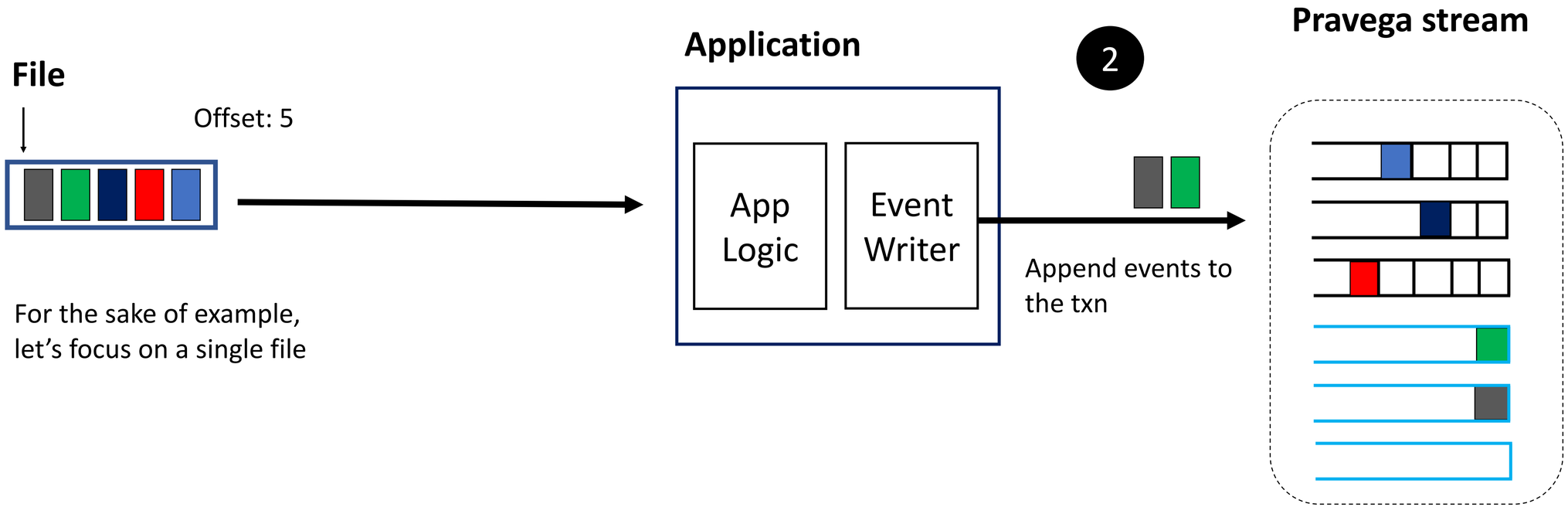
- For the sake of example, let’s focus on a single file

File source



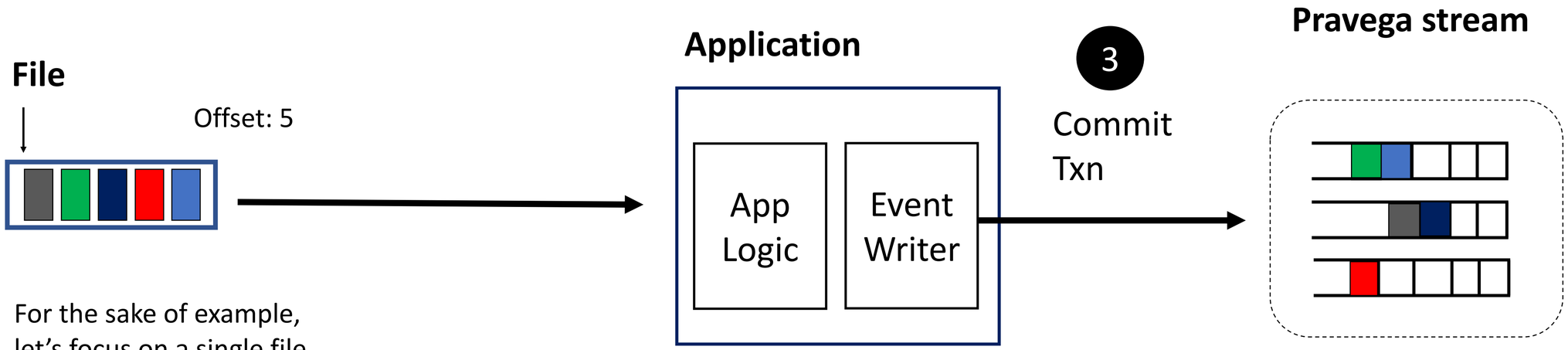
- For the sake of example, let’s focus on a single file

File source



- For the sake of example, let’s focus on a single file

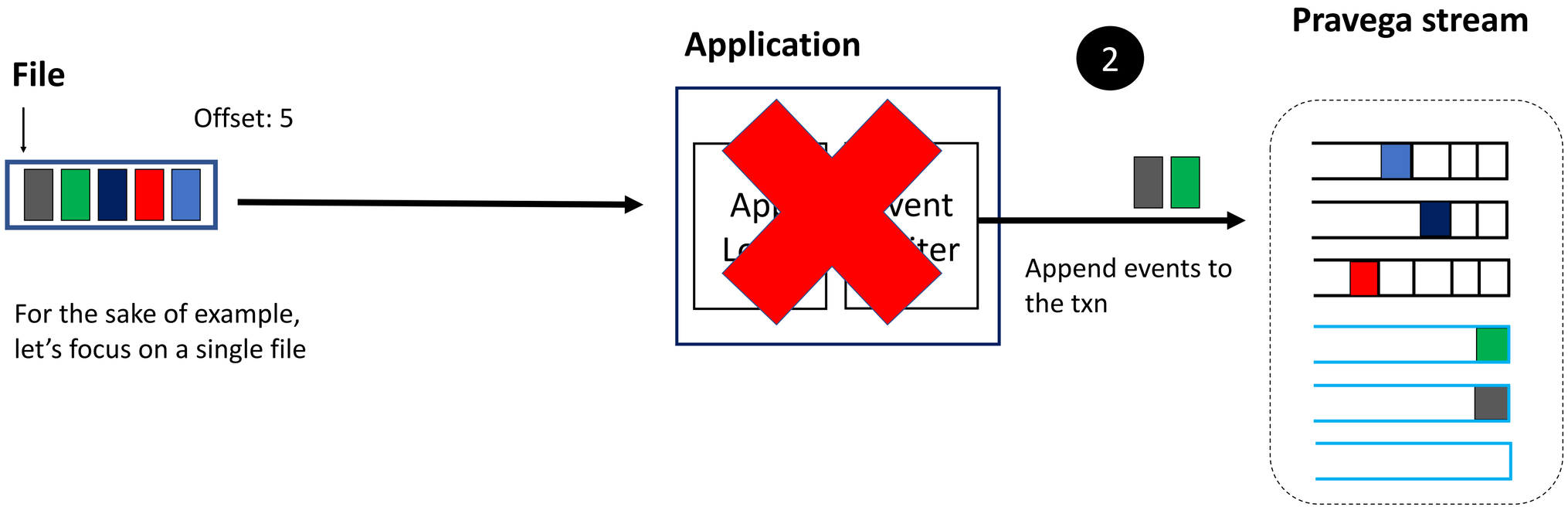
File source



- For the sake of example, let’s focus on a single file

What happens if application crashes in the middle of a transaction?

File source

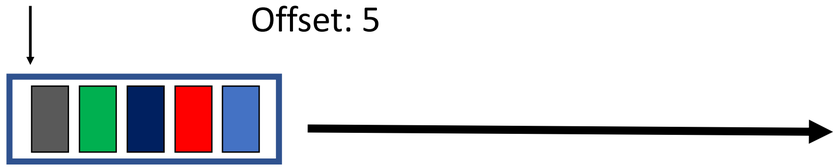


- For the sake of example, let’s focus on a single file

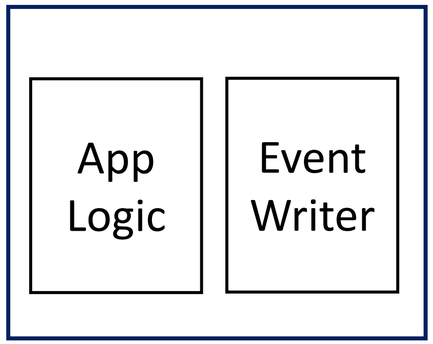
File source

File

Offset: 5

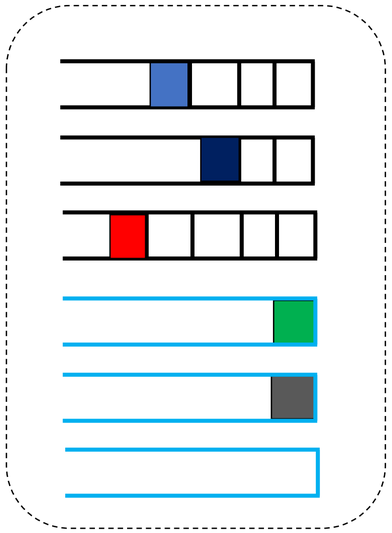


Application recovers



?

Pravega stream



- For the sake of example, let's focus on a single file

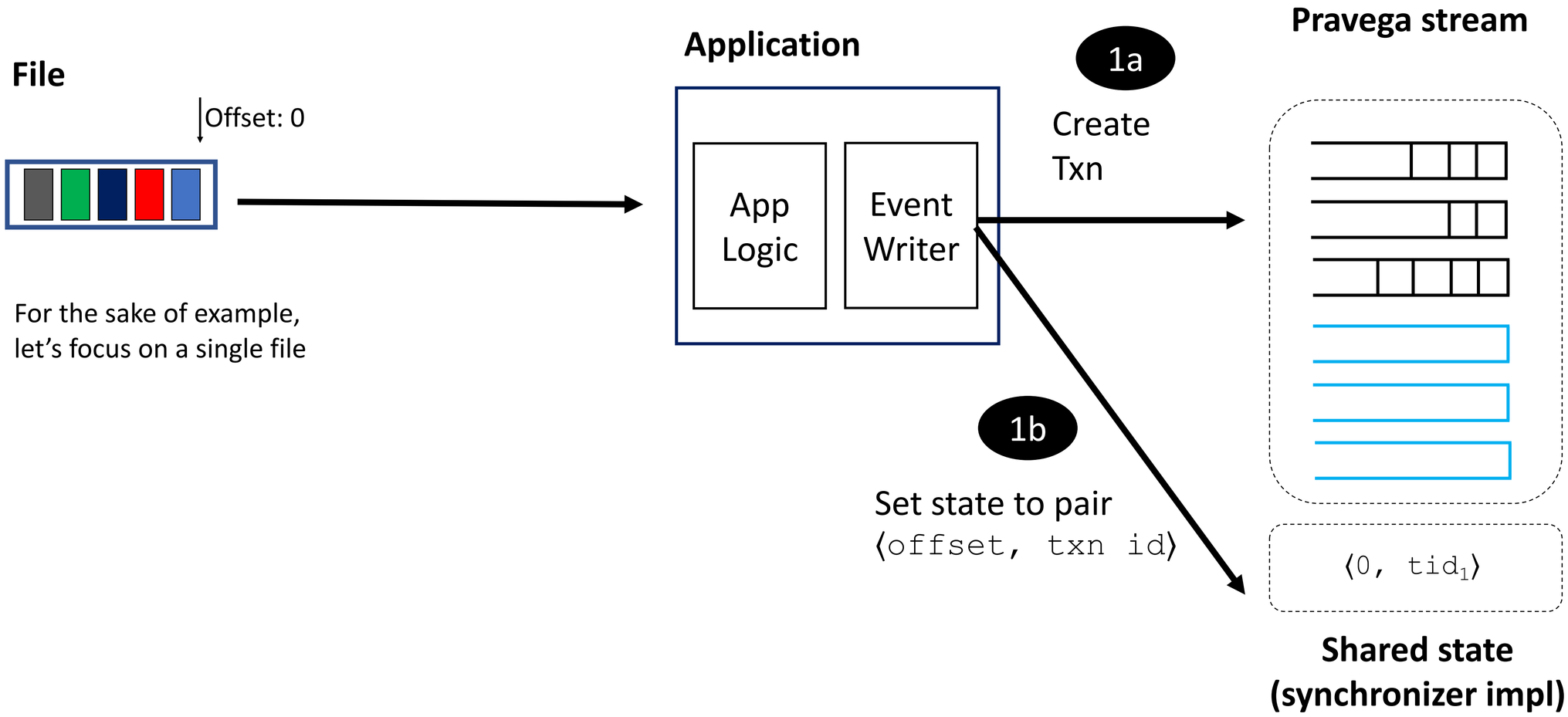
How does it know up to what offset has been successfully committed?

State Synchronizer

State synchronizer

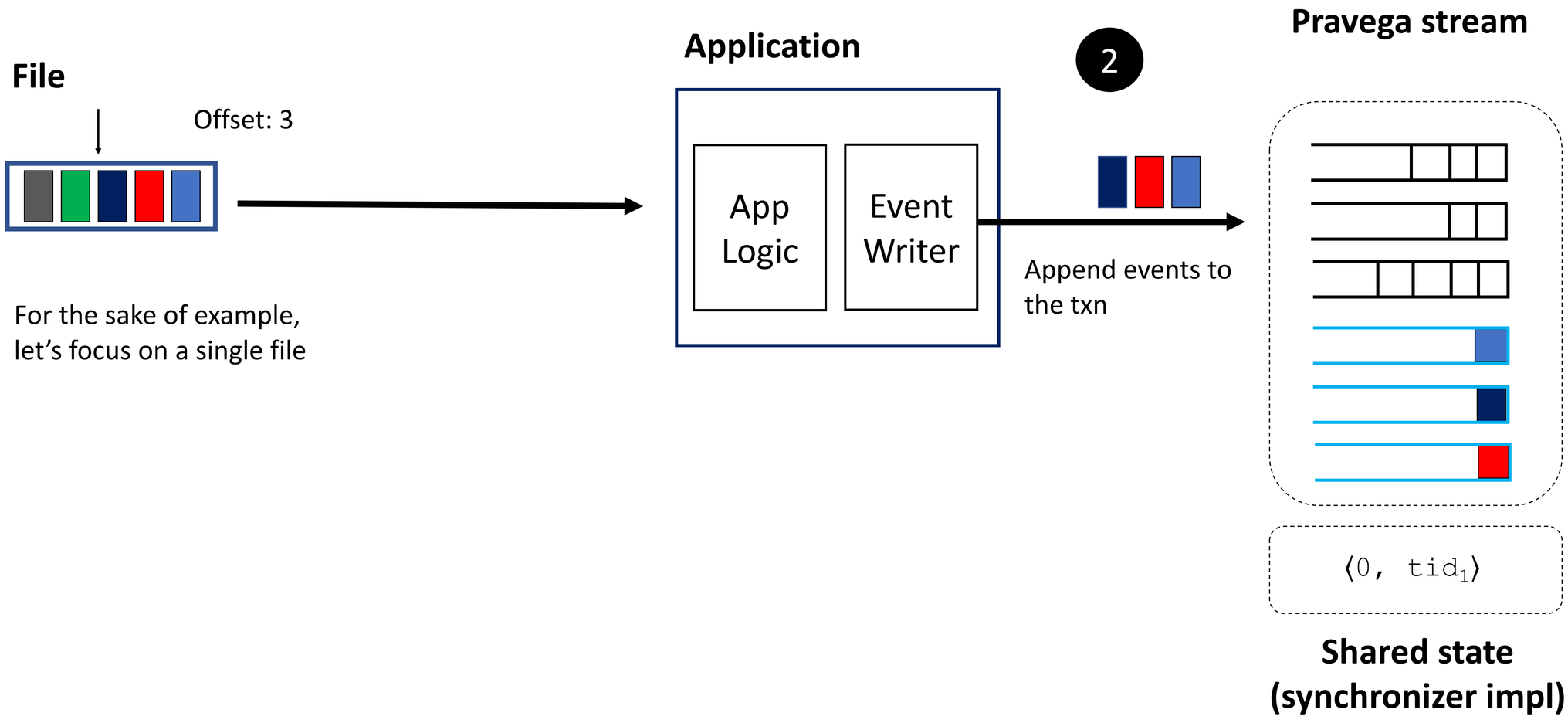
- An abstraction available in the client API
- Coordination of state across processes
  - *E.g.*, application instances
- Processes
  - Update state conditionally
  - Read state updates
- Application (our running example)
  - Shared state using the state synchronizer interface
  - State: a pair ⟨`starting file offset, txn id`⟩

File source



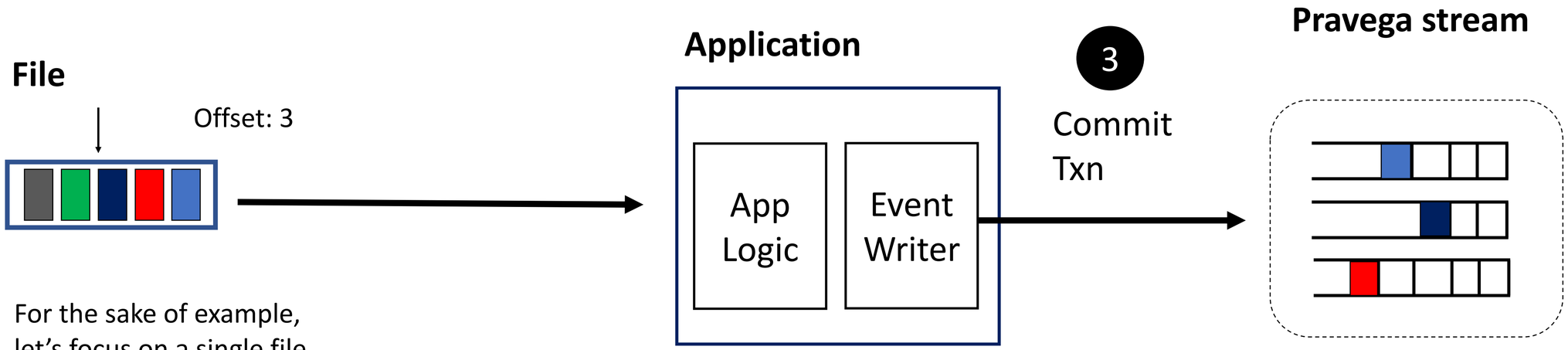
- For the sake of example, let's focus on a single file

File source

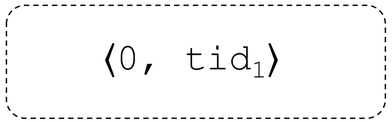


- For the sake of example, let's focus on a single file

File source

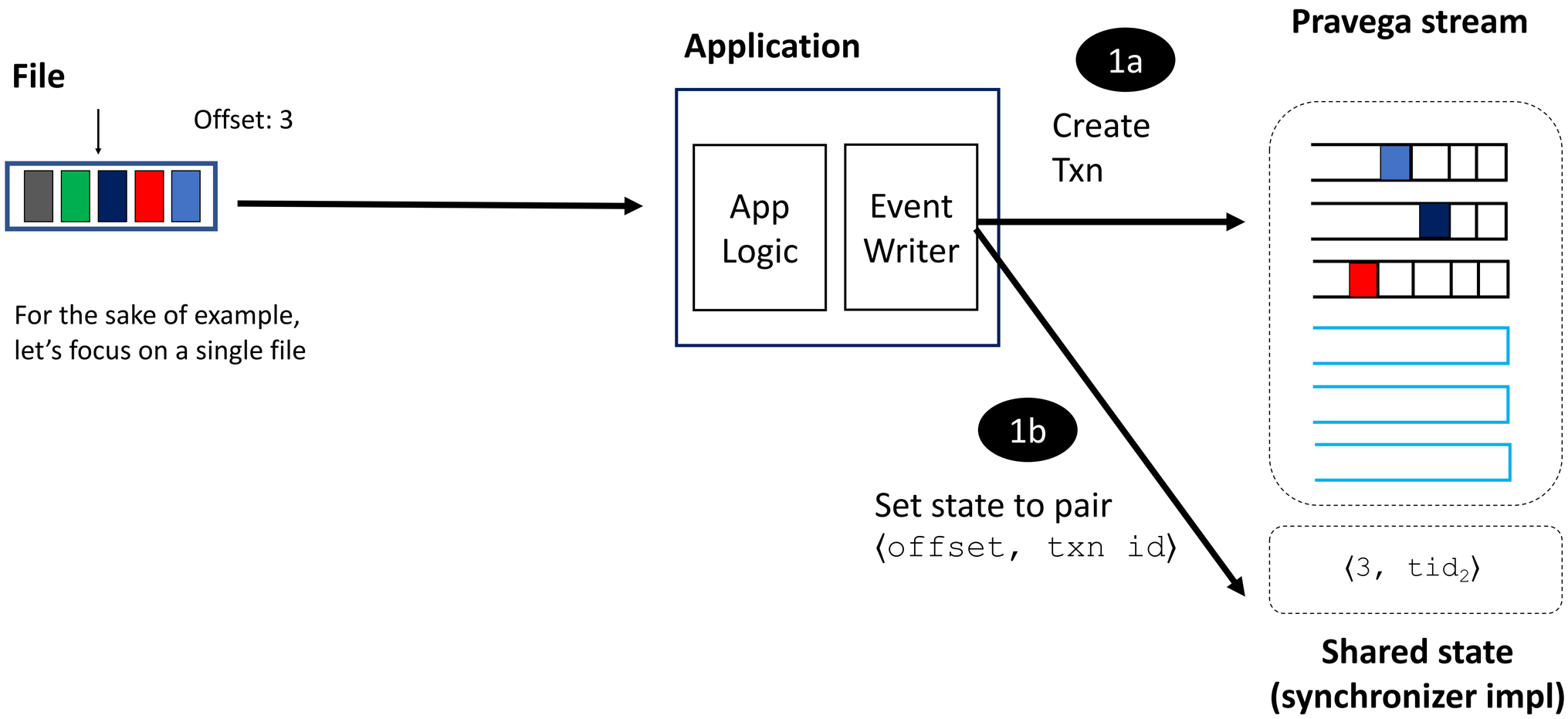


- For the sake of example, let's focus on a single file



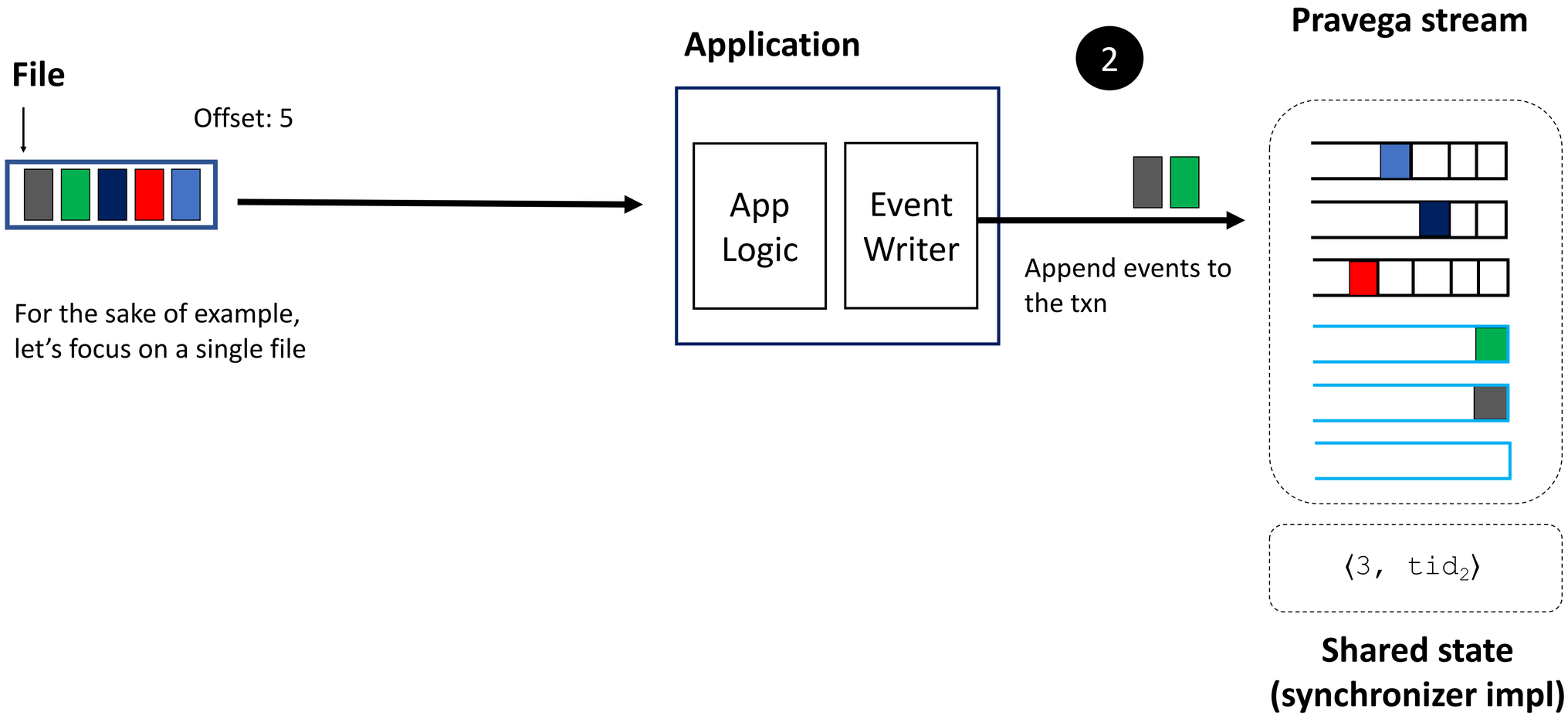
Shared state (synchronizer impl)

File source



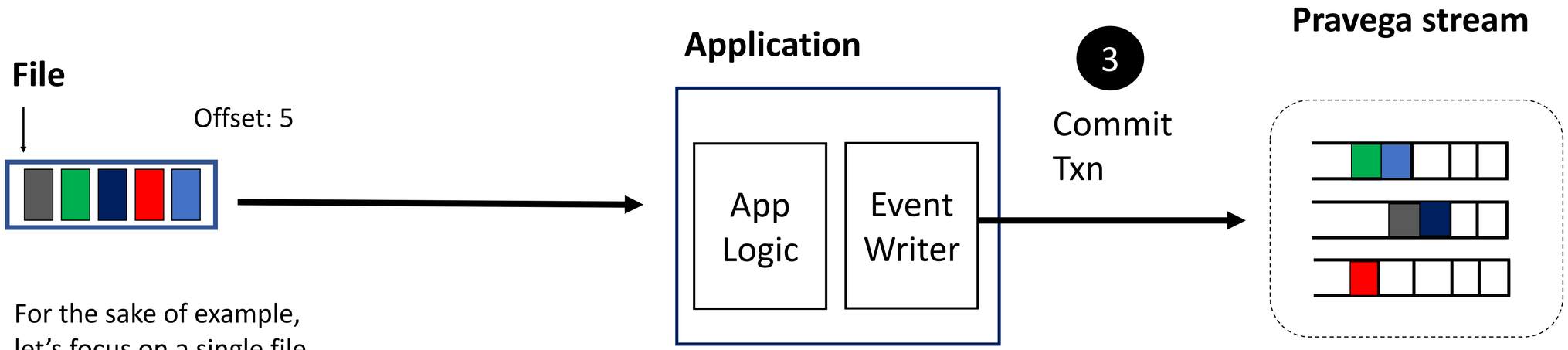
- For the sake of example, let's focus on a single file

File source

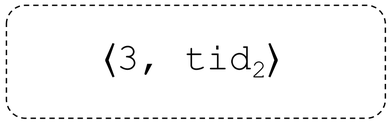


- For the sake of example, let's focus on a single file

File source



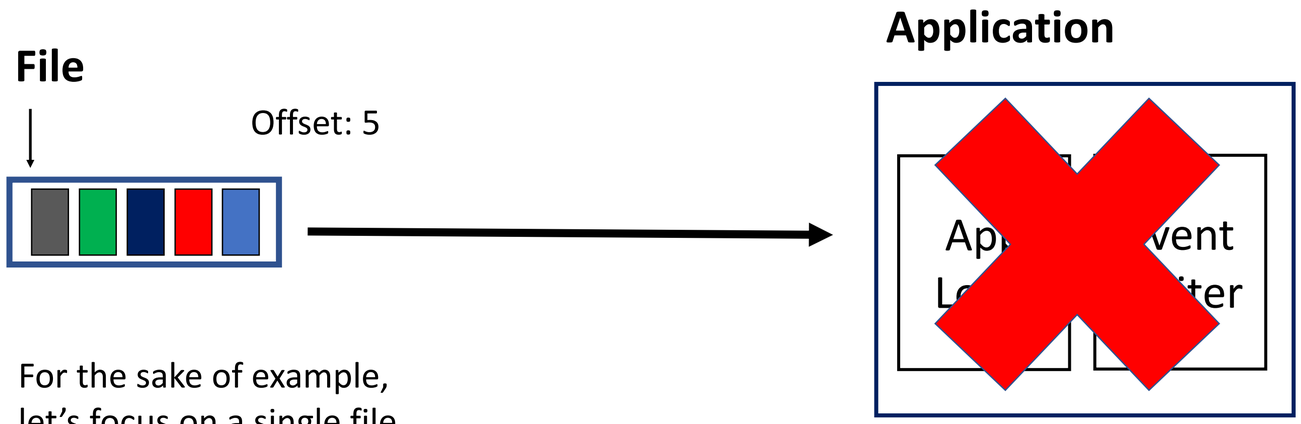
- For the sake of example, let's focus on a single file



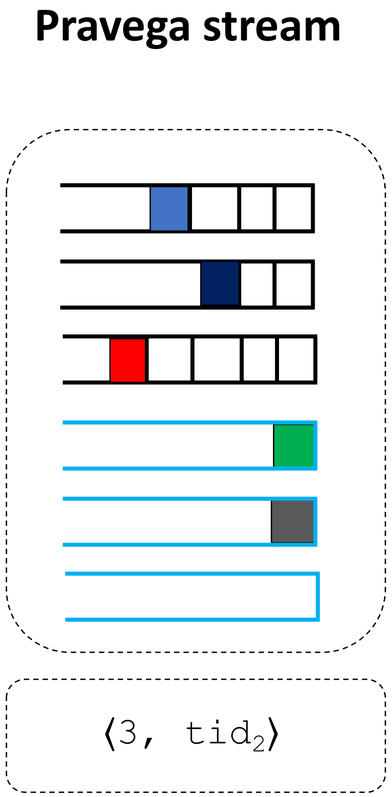
Shared state (synchronizer impl)

Application crashes before committing the transaction

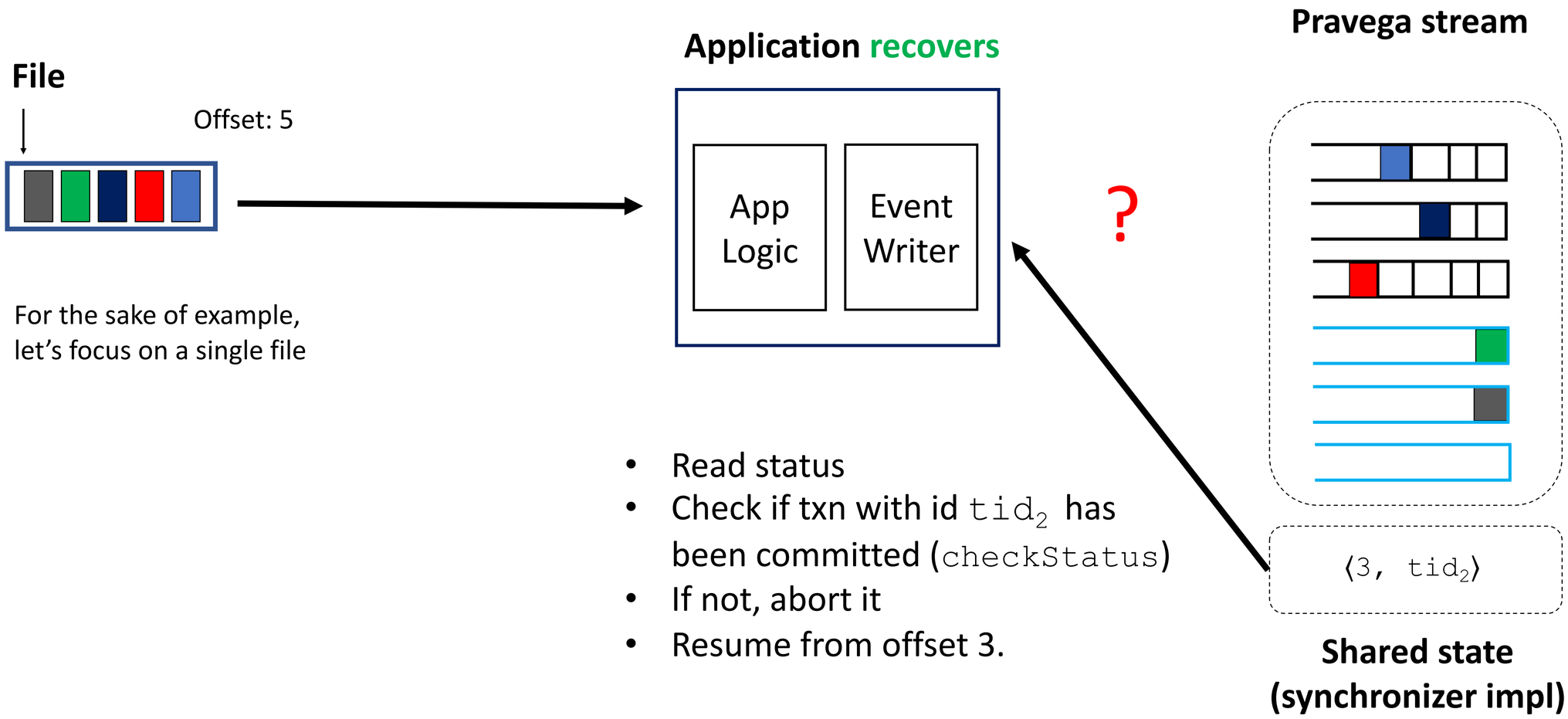
File source



- For the sake of example, let’s focus on a single file



File source



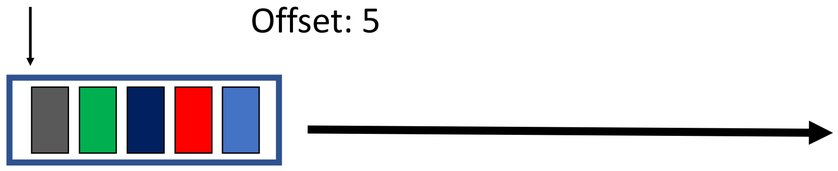
- For the sake of example, let's focus on a single file

- Read status
- Check if txn with id tid_2 has been committed (`checkStatus`)
- If not, abort it
- Resume from offset 3.

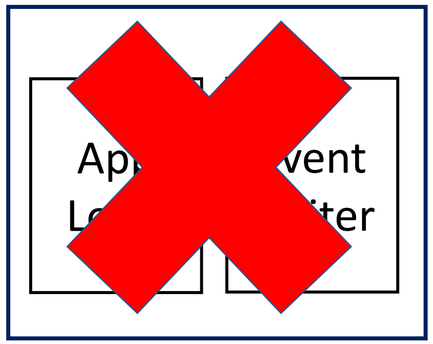
Application crashes after committing the transaction, but before creating a new one

File source

File

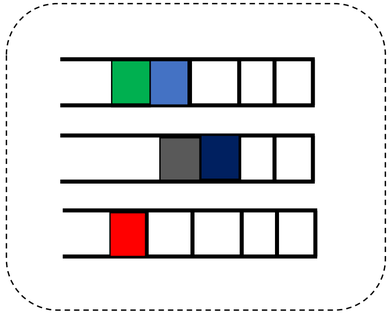


Application



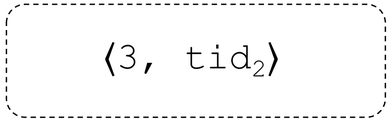
- For the sake of example, let's focus on a single file

Pravega stream



Problem:

- Do not know from which offset to resume
- No record of how much the previous transaction wrote



Shared state (synchronizer impl)

Determining the previous offset

- Deterministic offsets
  - Same number of events in a transaction, except for the last
  - With last recorded offset and transaction length in events, compute new offset
- Pre-compute transaction length
  - Determine transaction length before executing it
  - Record transaction length in the shared state
- Protocol change
  - Add a record for the end of a transaction
  - Record the end of a transaction with its length in the shared state before committing
  - Check status of transaction upon recovery
    - If transaction is not committed, then commit it
    - Proceed with new transaction
- Whole files
  - There is no real limit to the amount of data that can go into a transaction
  - In the special case of files, a whole file can be in a transaction

Code and Demo

Sample code

- Iterates over a set of sample files
  - One transaction per file
  - Commits once all events have been written
- Implements a state synchronizer
  - Upon creating a new txn, update synchronizer state
  - State is a single value: instance of `Status`
  - `Status` data structure: ⟨`file ID, txn ID`⟩
- Upon recovery
  - Fetch the latest status
  - Abort outstanding transaction
  - Set start file

Sources: https://github.com/fpj/eo-ingestion/

Stream initialization

```
// Create sync stream
StreamConfiguration syncStreamConfig = StreamConfiguration.builder()
                        .scalingPolicy(ScalingPolicy.fixed(1))
                        .build();
streamManager.createStream(scope, syncstream, syncStreamConfig);

// Create data stream
StreamConfiguration dataStreamConfig = StreamConfiguration.builder()
                        .scalingPolicy(ScalingPolicy.fixed(10))
                        .build();
streamManager.createStream(scope, datastream, dataStreamConfig);
```

1

2

1- Create a stream for the state synchronizer
2- Create a stream for the ingested data

Iterating over the files

```
Arrays.stream(path.toFile().listFiles()).sorted( (f1, f2) -> { //Comparator}).forEach( f -> {
        // Skip files that have already been committed
        …
        Transaction<Sample> txn = writer.beginTxn();
        // Add txn id and file name to state synchronizer
        Status newStatus = Status.newBuilder().setFileId(fileId).setTxnId(txn.getTxnId().toString()).build();
        this.synchronizer.updateStatus(newStatus, this.currentStatus))
        // Read from file and write to txn
         ….
        txn.writeEvent(sample.getId().toString(), sample);
        // Commit transaction
        txn.commit();
});
```

1
2
3
4
5
6

1- Iterate over the list of sorted files
2- Skip files that have already been committed
3- Create new txn for file to be ingested
4- Write new status to state synchronizer
5- Iteratively, read from file and write to Pravega
6- Once donde, commit txn

State synchronizer implementation

```
public class ExactlyOnceIngestionSynchronizer {
    static class UpdatableStatus implements Revisioned {
    ...
    }
    static class StatusInit implements InitialUpdate<UpdatableStatus>, Serializable {
     ...
    }
    static class StatusUpdate implements Update<UpdatableStatus>, Serializable {
     ...
    }
    private final StateSynchronizer<UpdatableStatus> stateSynchronizer;

    // Access methods
    …
}
```

1

2

3

4

1- Defines the state of this synchronizer
2- Generates the initial state
3- Generates status updates
4- Declares a state synchronizer of type `UpdatableStatus`

Recovery implementation

```
// Obtain txnID from synchronizer status
Transaction<Sample> txn = writer.getTxn(txnId);
Transaction.Status status = txn.checkStatus();
switch(status) {
    case OPEN:
        txn.abort();
        this.startFileId = this.currentStatus.getFileId();
        break;
    case ABORTED: case ABORTING:
        this.startFileId = this.currentStatus.getFileId();
        break;
    case COMMITTED: case COMMITTING:
        this.startFileId = this.currentStatus.getFileId() + 1;
        break;
};
```

1

2

3

4

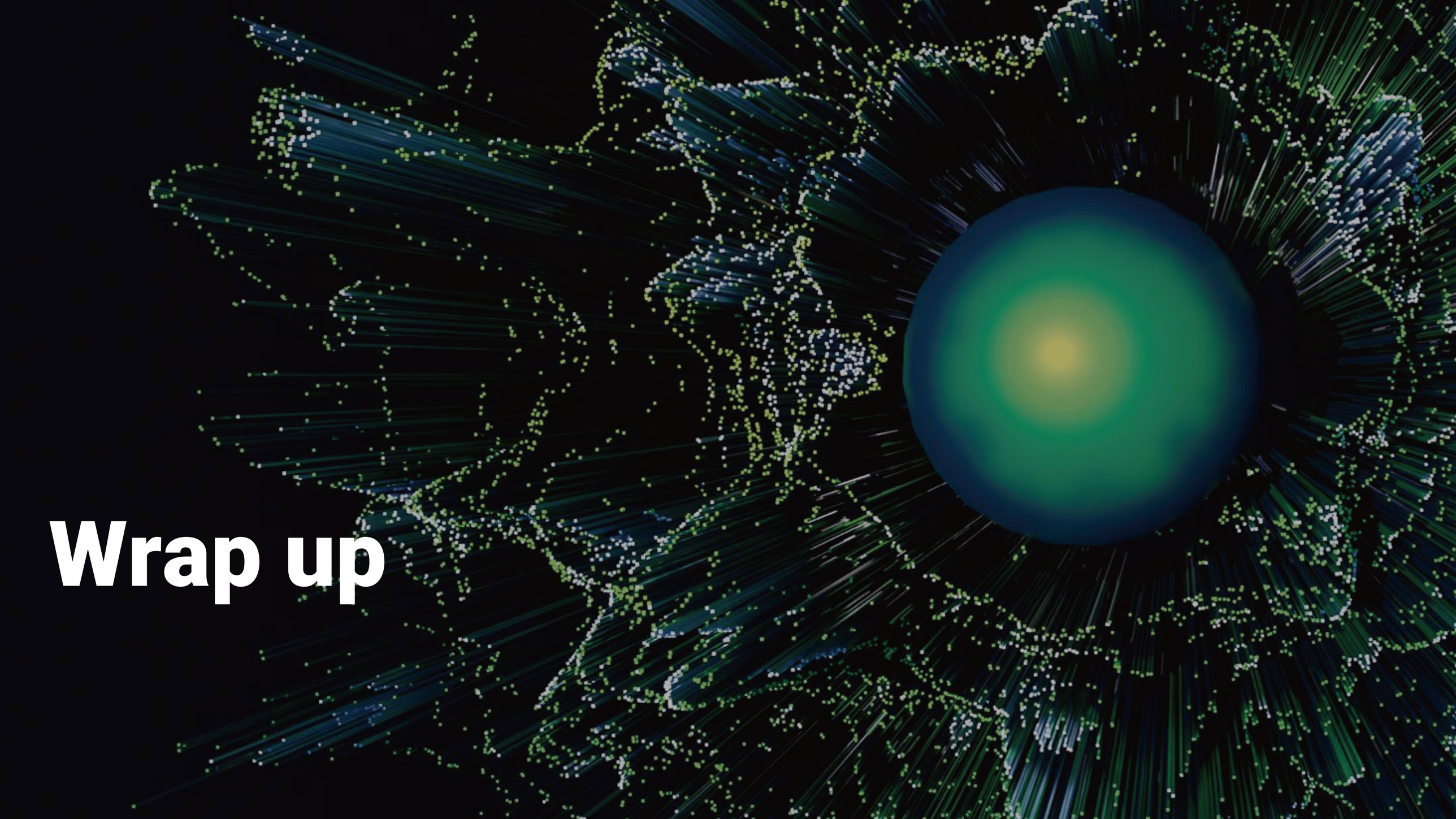
1- Obtain the last written status and get the status of the last txn
2- If txn is open, then abort it and start from the last file
3- If the txn has been aborted, then start from the last file
4- If the txn has been committed, then start from the next file

Demo

Concurrency

Concurrency

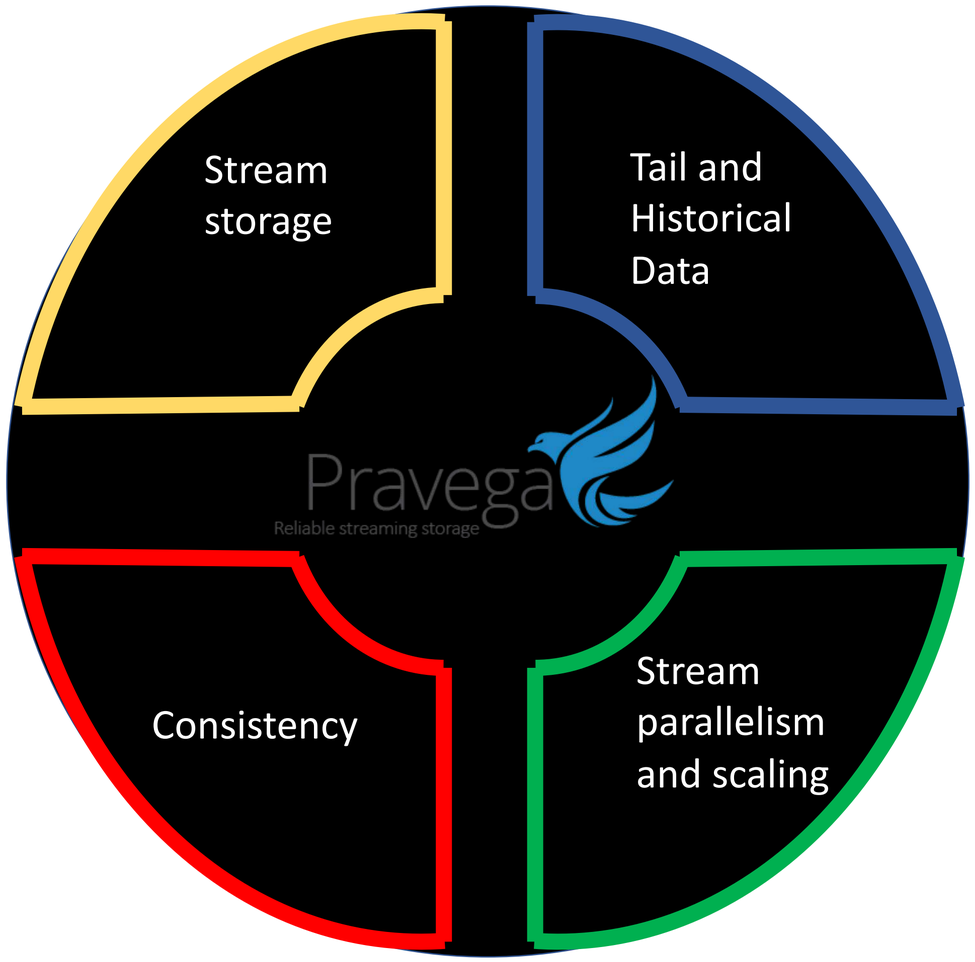
- Writer is falsely suspected
  - No two concurrent transactions should be allowed to proceed
- Correctness
  - Duplicates
    - Writer must update state before writing and committing
    - Conditional update succeeds for one writer
  - Completeness
    - Writer follows the order of sorted files
    - On recovery, starts from the next uncommitted file
- Assumption
  - Some mechanism telling writers to process a given file (leader selection)
  - Such mechanism cannot avoid zombie writers altogether



Wrap up

Conclusion

- Pravega
  - Stream as a storage primitive
  - Low latency and historical processing
  - Built on segments
  - Open source
- Apache Flink
  - Connector enables Flink to read from and write to a Pravega stream
  - Exactly-once end-to-end
    - Source uses checkpoints
    - Sink uses transactions
- Exactly-once ingestion
  - Transaction
  - State synchronizer


Stream storage
Tail and Historical Data
Pravega
Reliable streaming storage
Consistency
Stream parallelism and scaling

Questions?

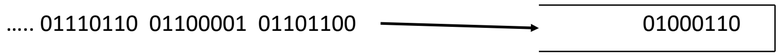
- Email: fpj@pravega.io
- Twitter: @fpjunqueira
- Linkedin: https://www.linkedin.com/in/flavio-junqueira-bab134

- Pravega Web site: http://pravega.io
- GitHub: https://github.com/pravega/pravega
- Flink connector: https://github.com/pravega/flink-connectors
- Ingestion sample: https://github.com/fpj/eo-ingestion

Stream scaling

(a.k.a, changing the degree of parallelism)

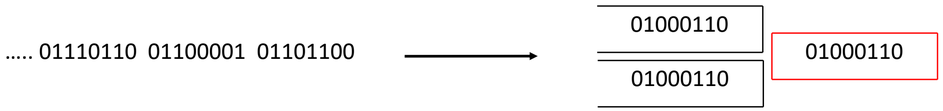
Scaling a stream



- Stream has one segment

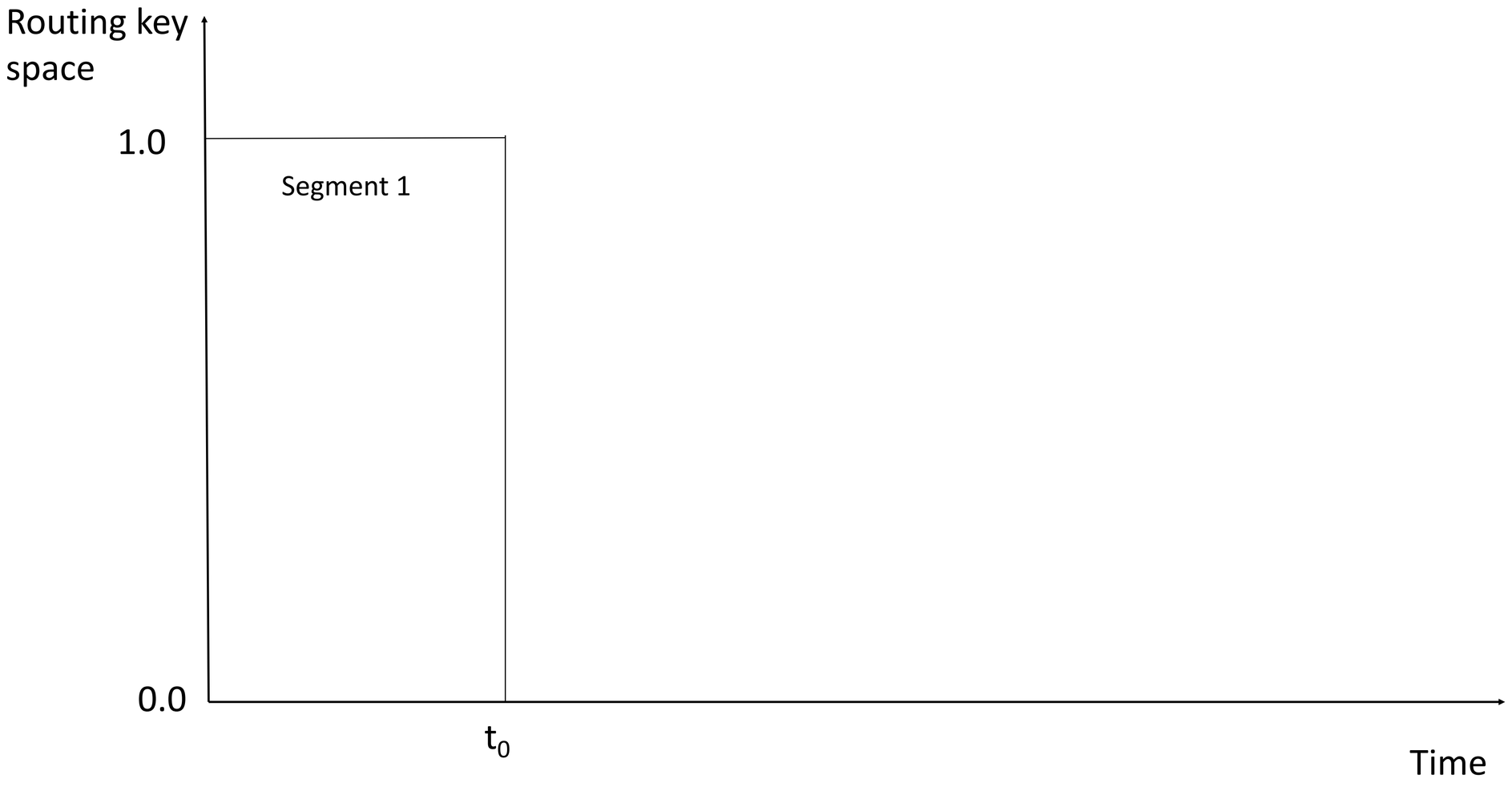
1

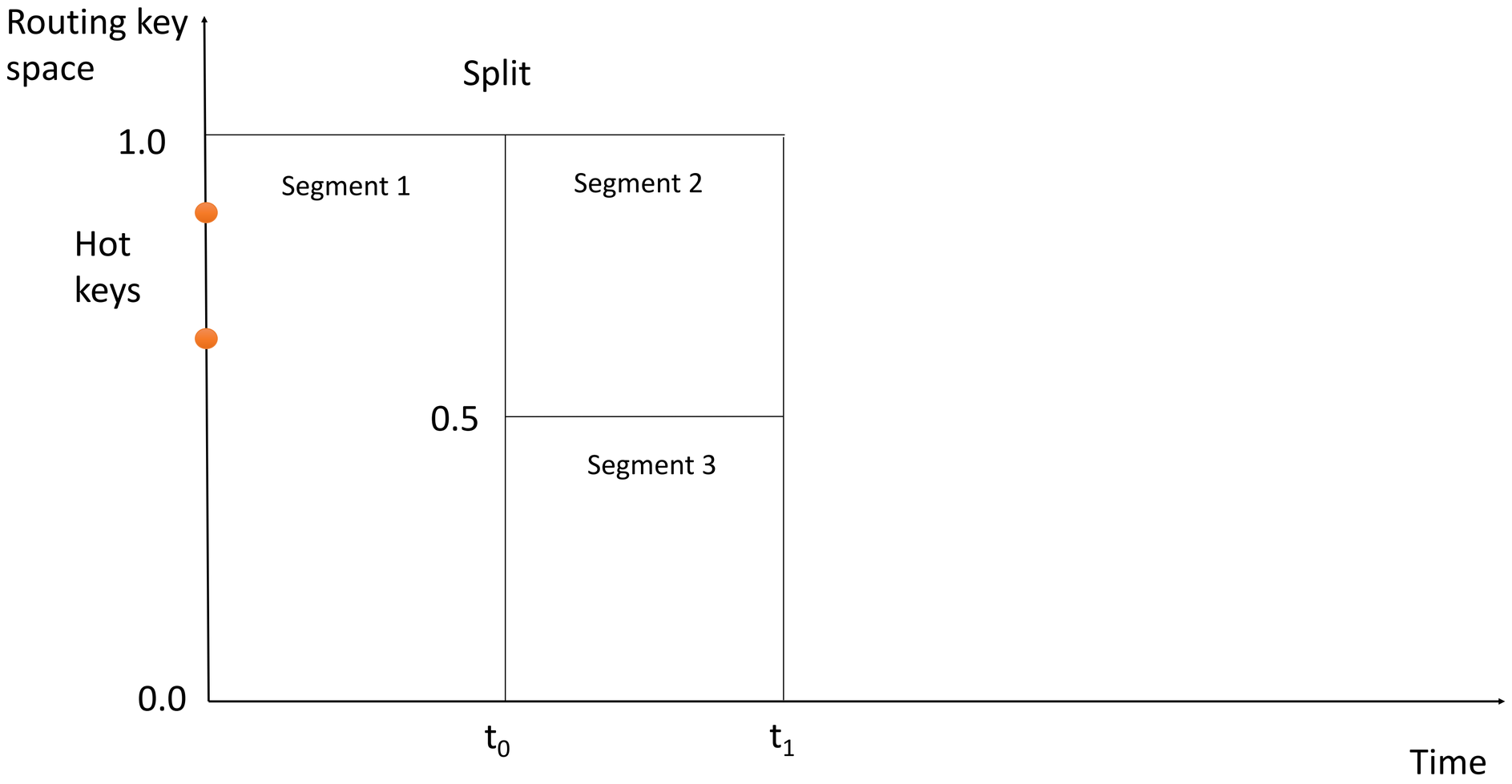
- Auto or manual scaling
- Auto scaling
  - Follows write workload
  - Say input load has increased
  - Increase the degree of parallelism
- Manual scaling
  - API call

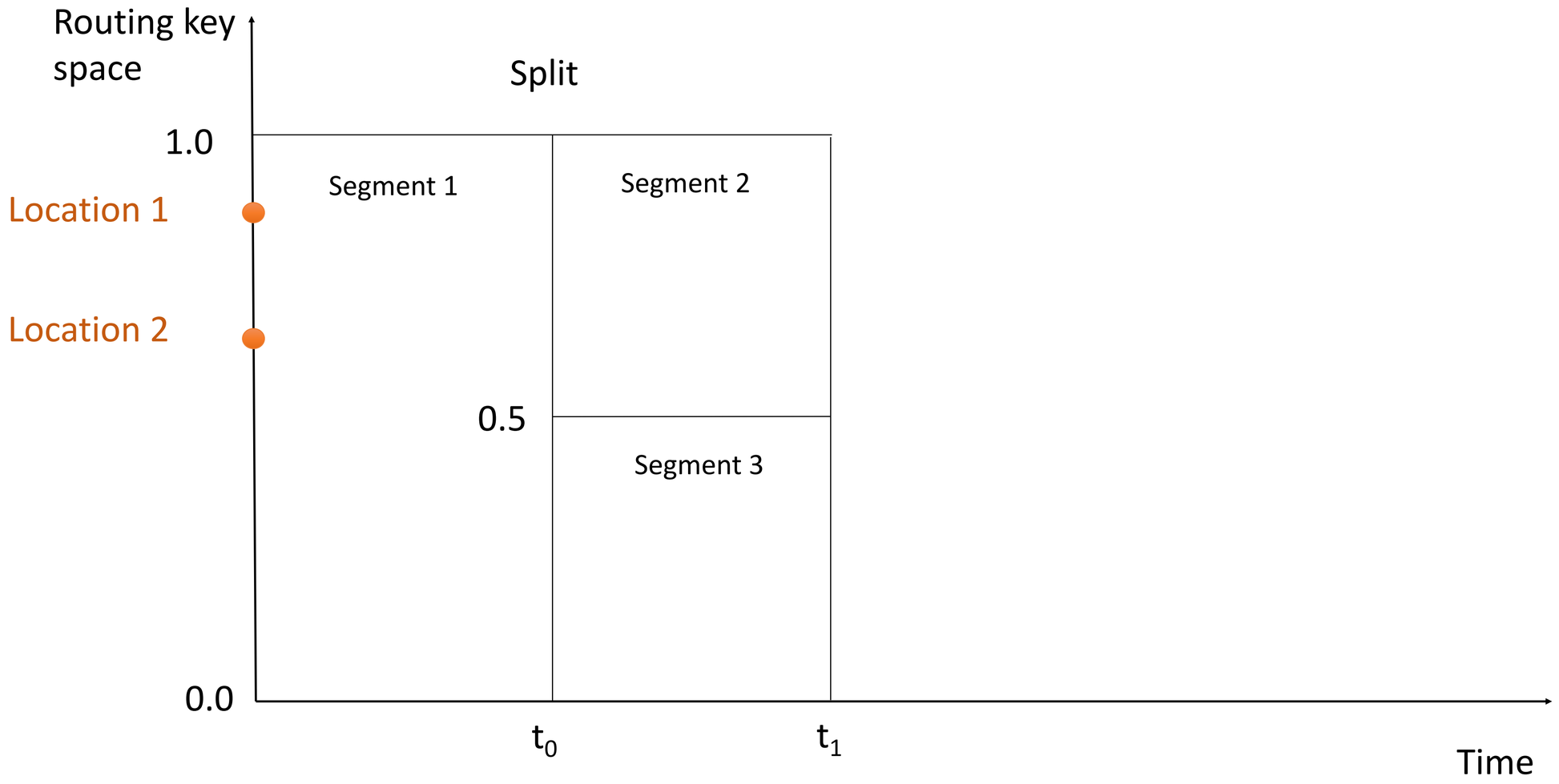


- Seal current segment
- Create new ones

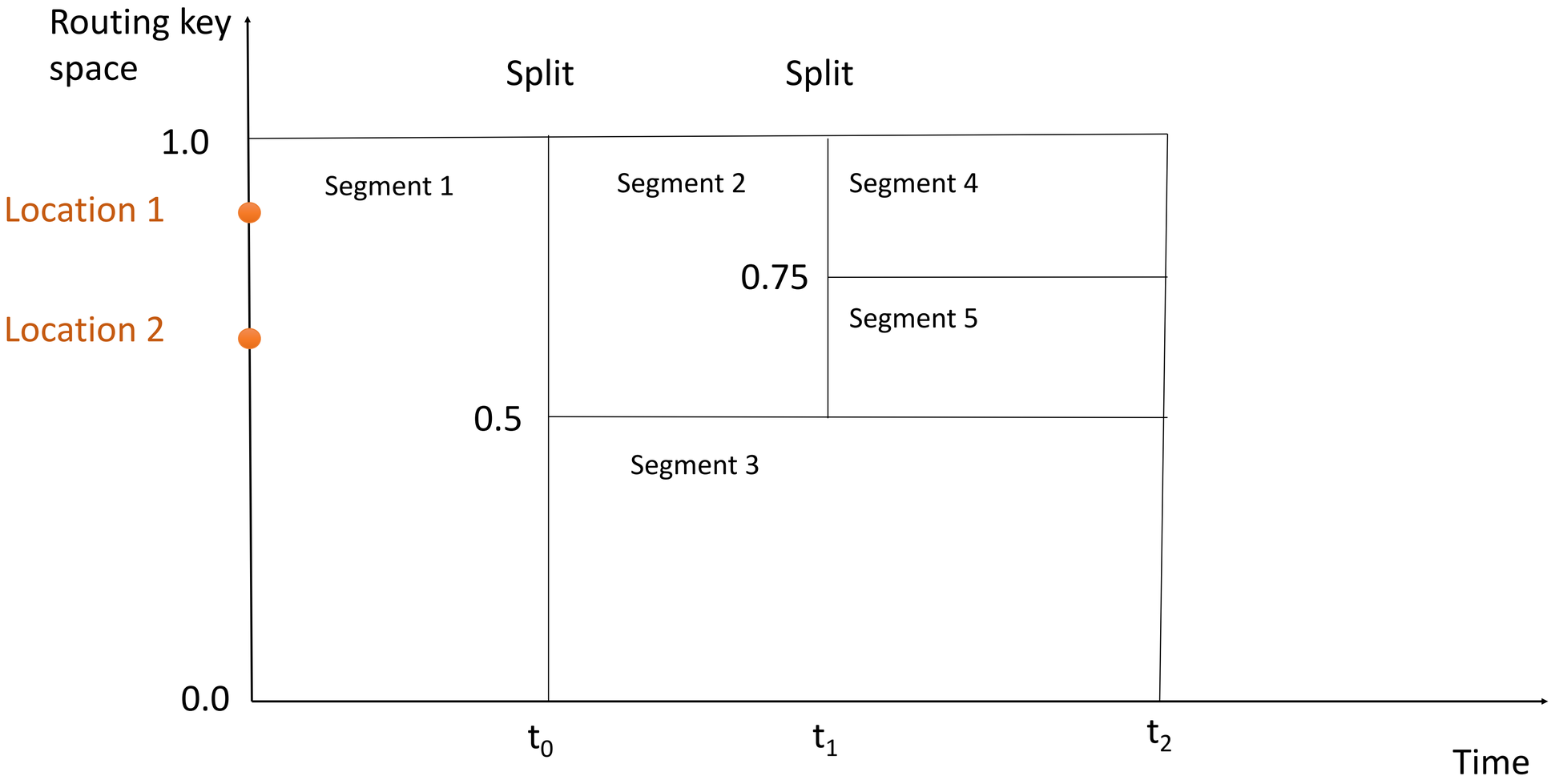
2


Routing key space
1.0
Segment 1
0.0
t_0
Time

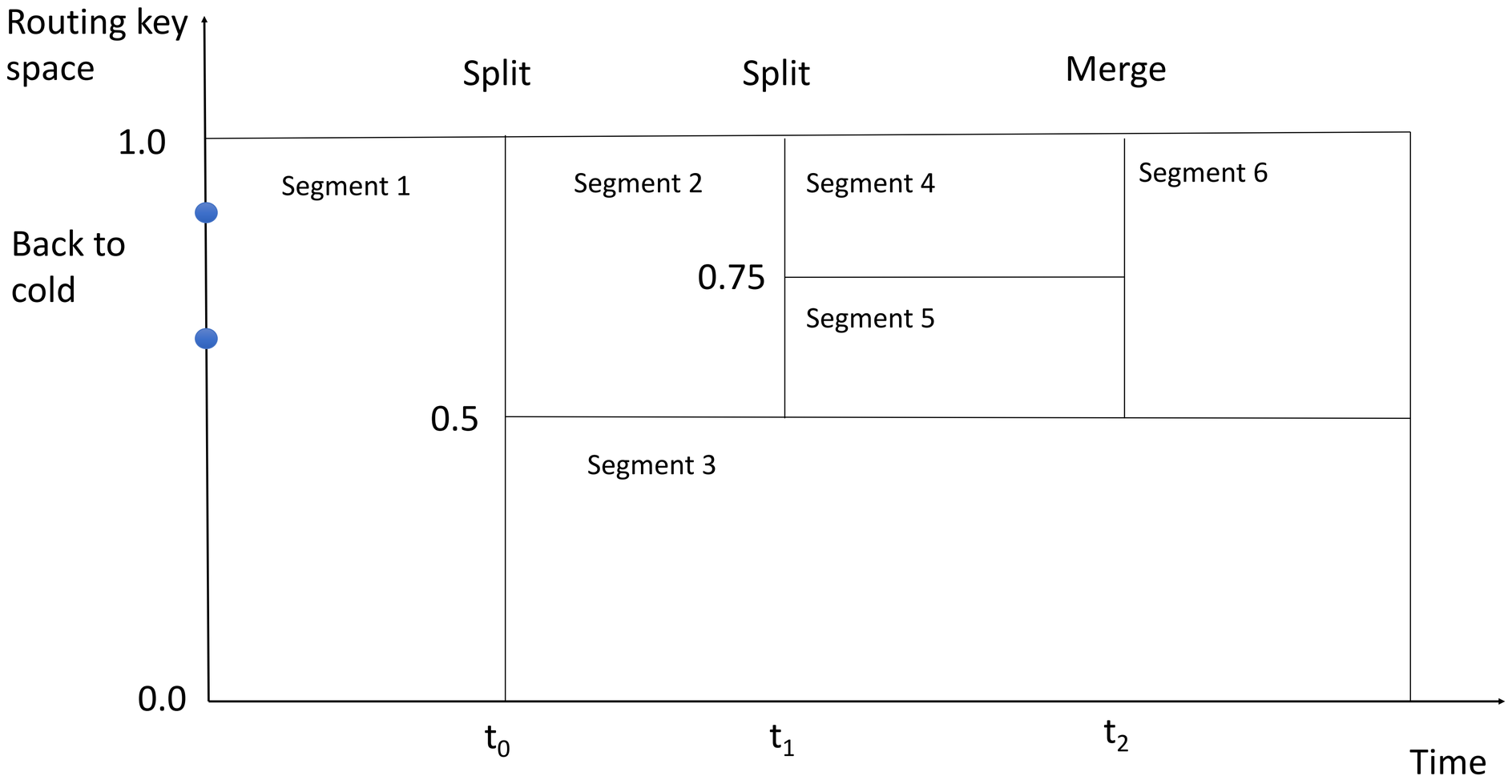

Routing key space
Split
1.0
Segment 1
Segment 2
Hot keys
0.5
Segment 3
0.0
t_0
t_1
Time



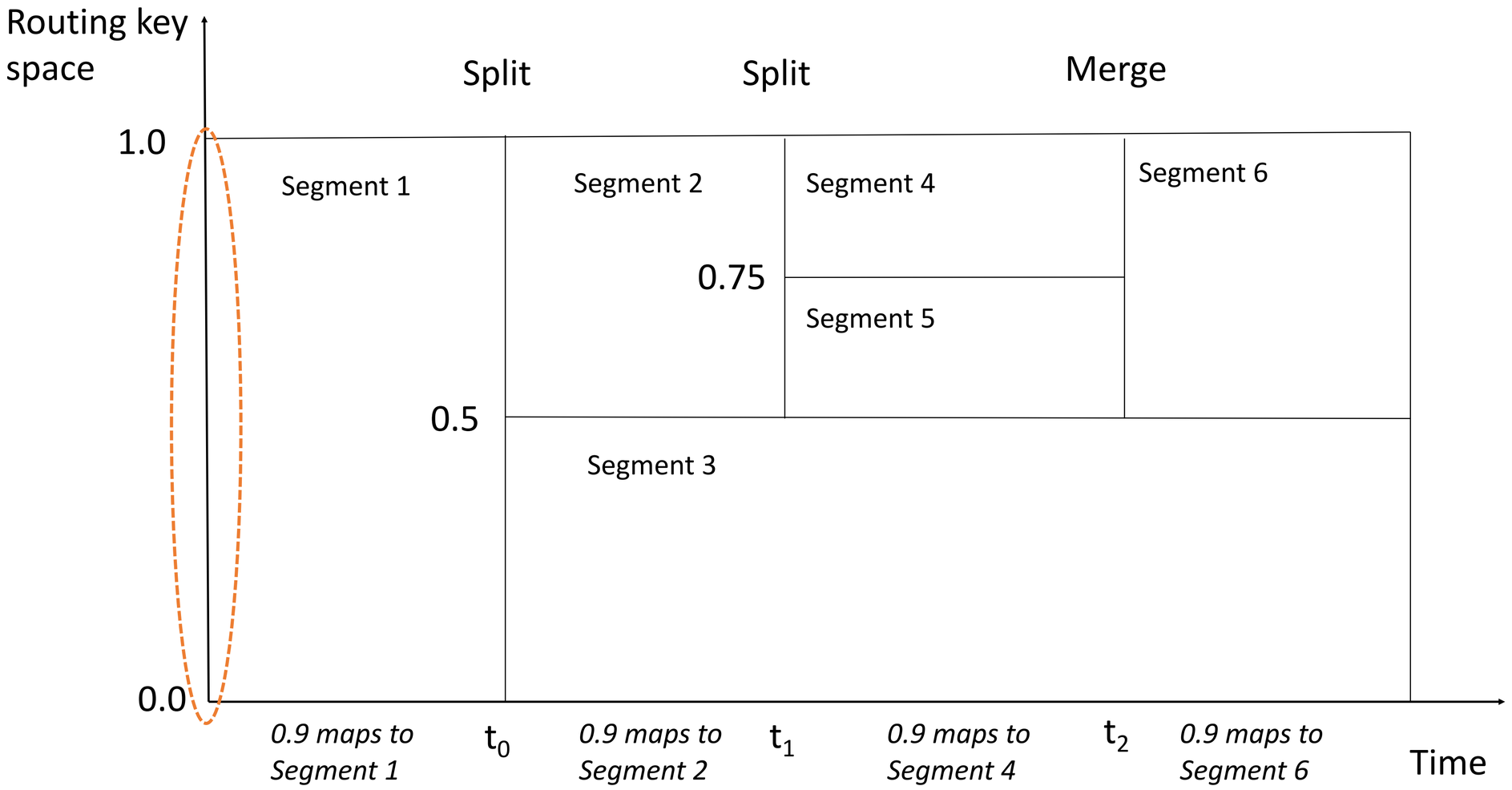
- Keys are coordinates in a geo application
- *E.g.*, taxi rides

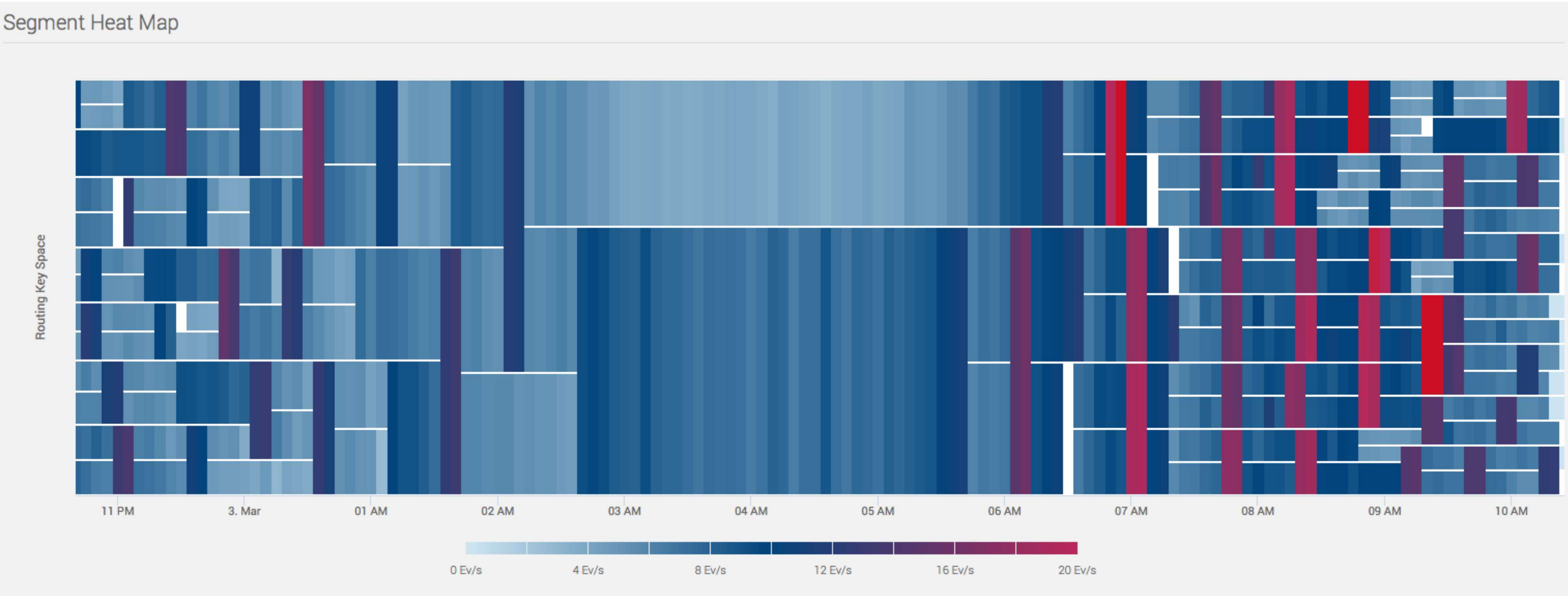


- Keys are coordinates in a geo application
- *E.g.*, taxi rides


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

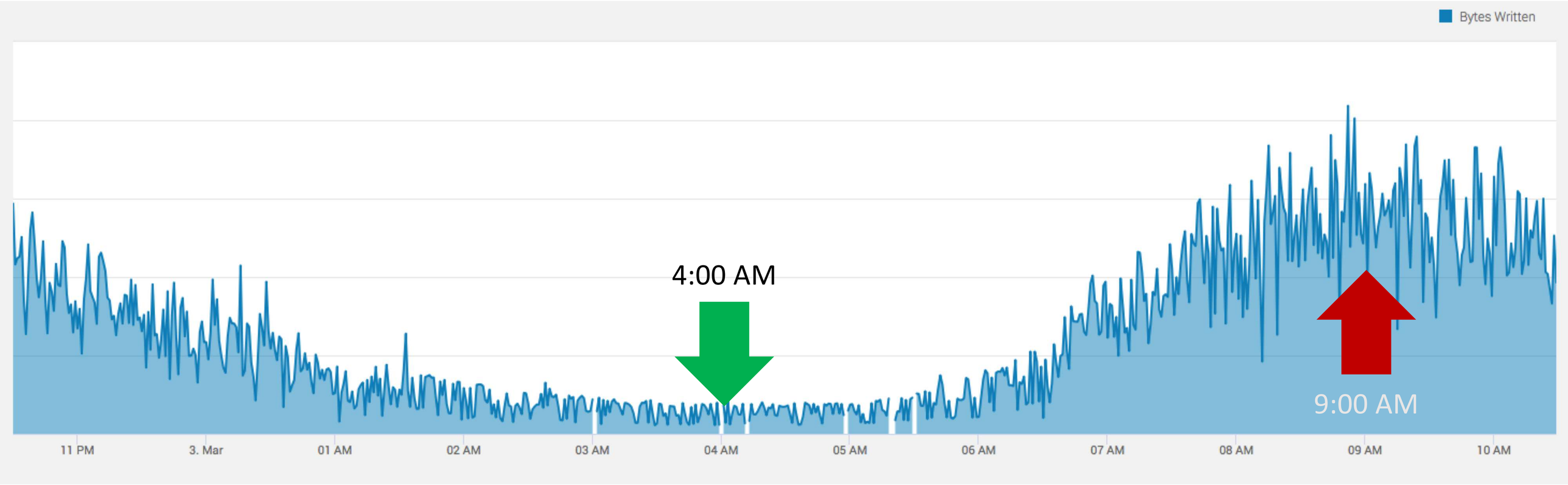
Key ranges are not statically assigned to segments




Segment Heat Map
Routing Key Space
11 PM
3. Mar
01 AM
02 AM
03 AM
04 AM
05 AM
06 AM
07 AM
08 AM
09 AM
10 AM
0 Ev/s
4 Ev/s
8 Ev/s
12 Ev/s
16 Ev/s
20 Ev/s

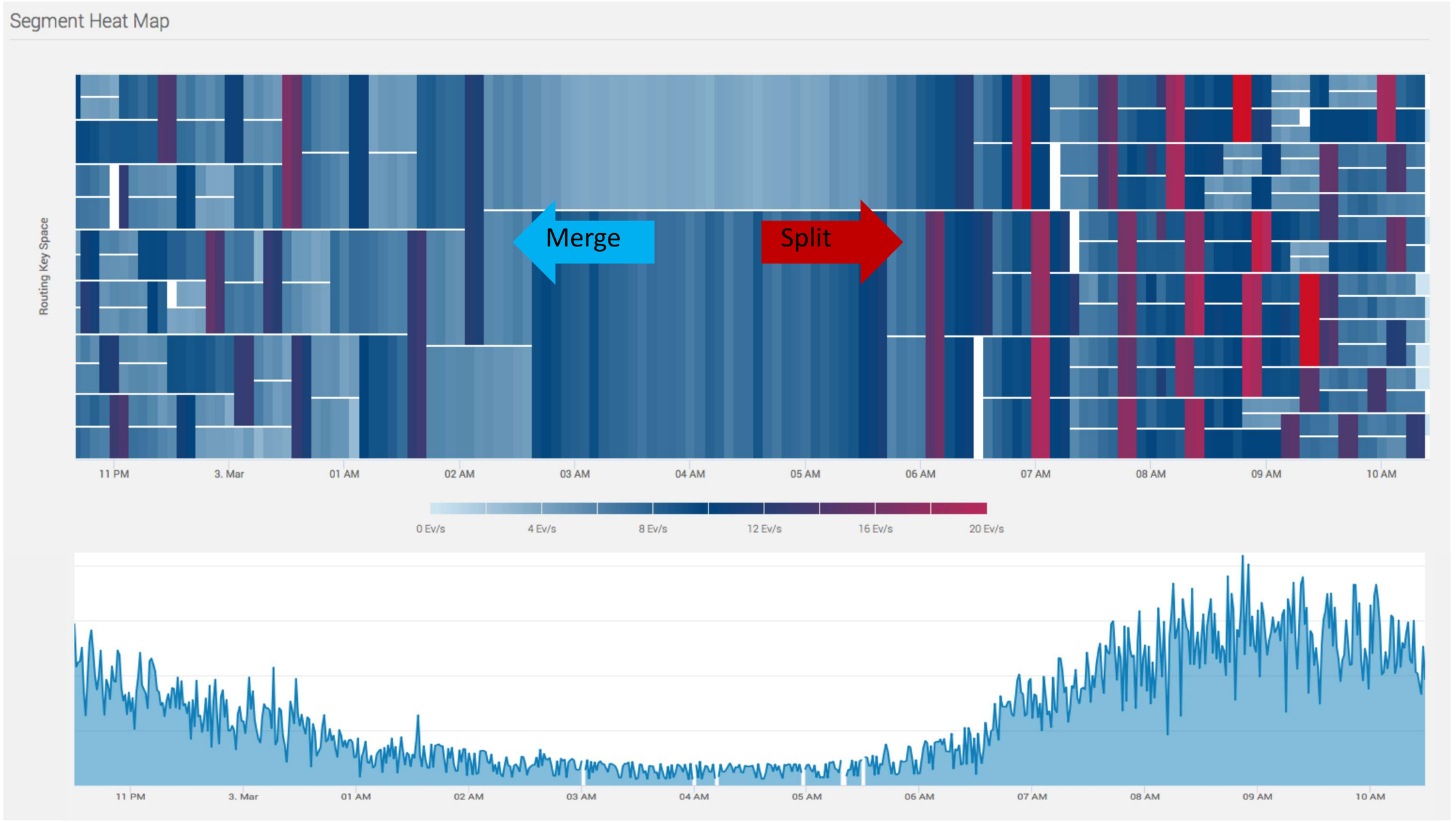
Daily Cycles

Peak rate is 10x higher than lowest rate



NYC Yellow Taxi Trip Records, March 2015
http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

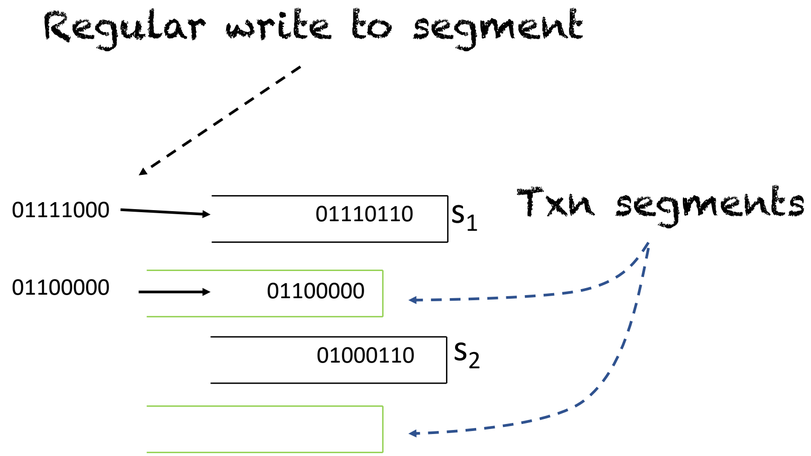
Pravega Auto Scaling

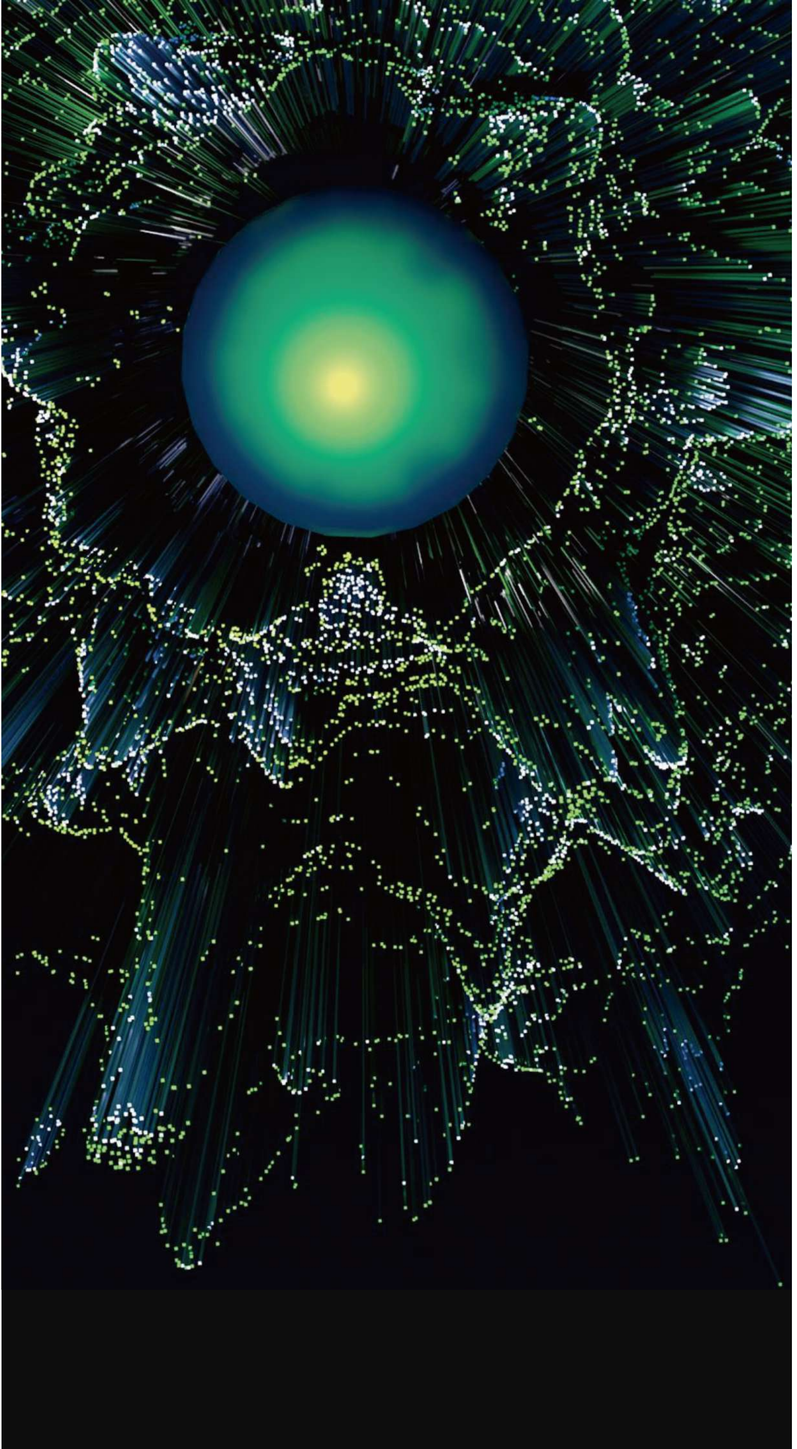


Transactions

Transactions

- Scope is a single stream
- Transactional writes
  - Any open segment of the stream
  - No limitation on the routing key range
  - Interleaved with regular writes
- Important for exactly-once semantics
  - Either all writes become visible or none
  - Aborted manually or via timeout
- Upon scaling
  - Transactions roll
  - Segments preserved until committed or aborted





Pravega
Reliable streaming storage

FLINK FORWARD

Font/Color/logo usage specification

Font： Roboto

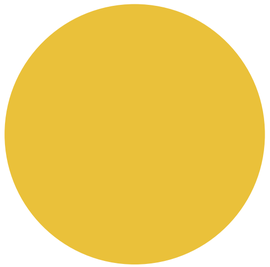
Data analysis platform based on Apache Flink

title- Font size 32

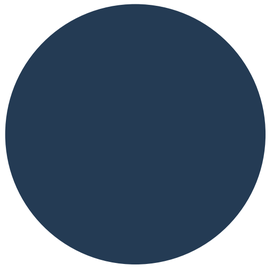
title- Font size 70

title- Font size 120

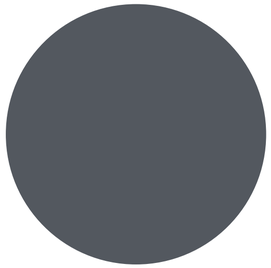
Color selection range



Emphasize color A
eac13a



Emphasize color B
243B54



Content color
53585F

Shadow

Large area background color
e5eaf4

Content security scope

A space should be added English/numerals

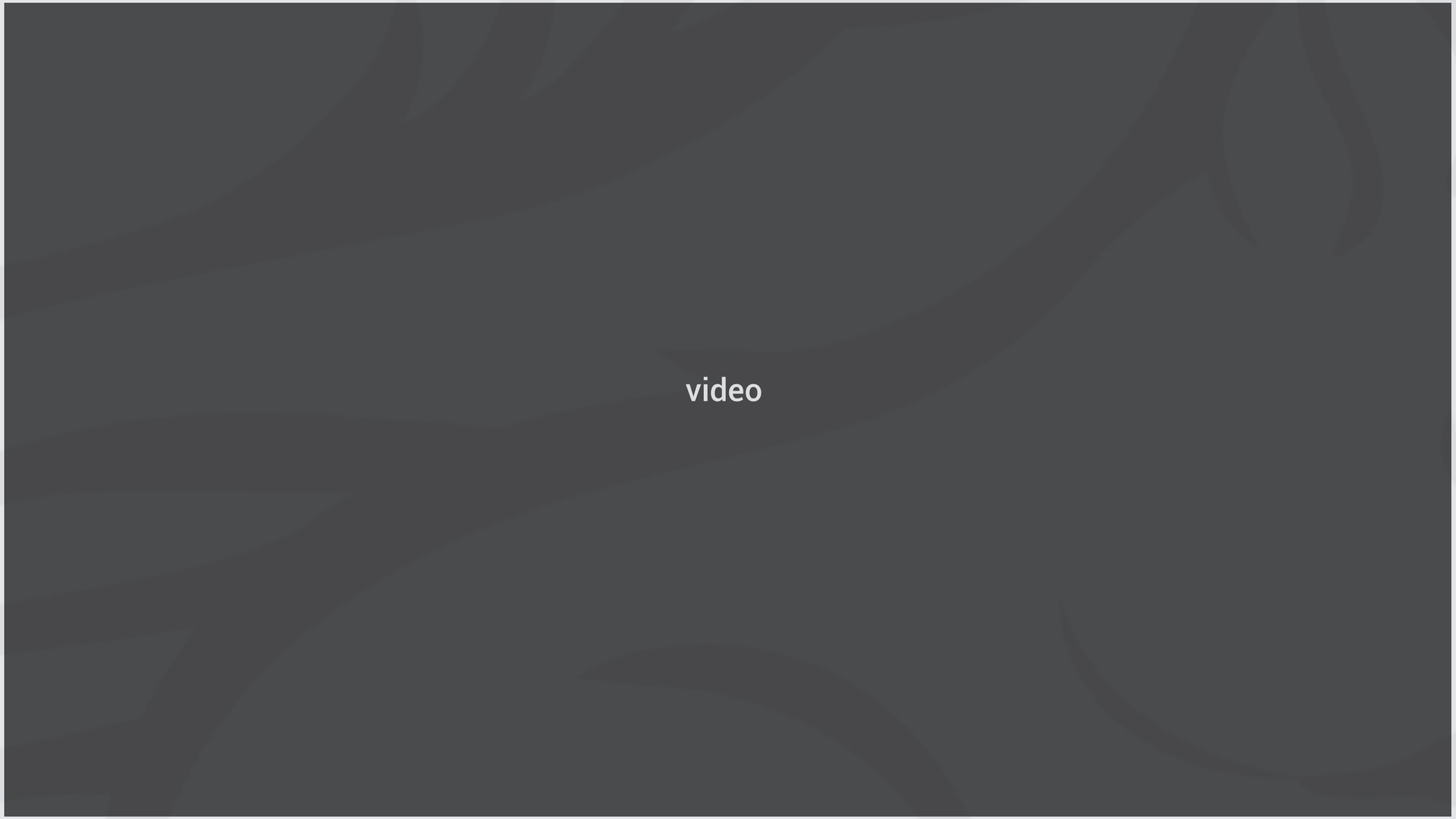
Use left-right or end alignment for multiple lines of text

Multi-line text avoids heading and tail processing by controlling text area

Contents should be kept at a safe distance to ensure that the bottom text is not blocked by the front audience.

Style recommendation of code page

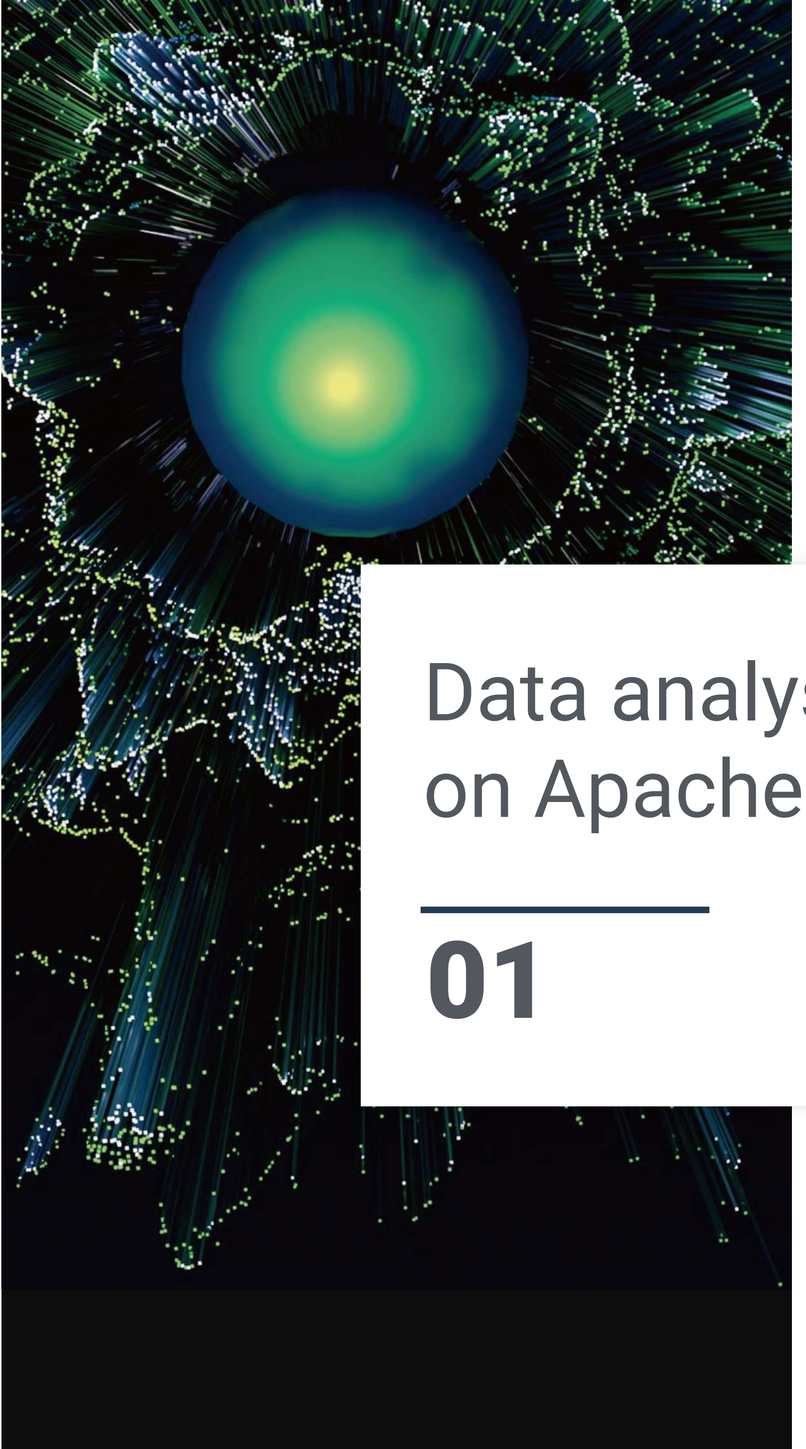
```
MATCH_RECOGNIZE(
        PARTITION BY cellId
        ORDER BY rowTime
        MEASURES
                FIRST(UP.startTime) as rushStart,
                LAST(DOWN.endTime) AS rushEnd,
                SUM(UP.rideCount) + SUM(DOWN.rideCount) AS rideSum
        AFTER MATCH SKIP PAST LAST ROW
        PATTERN (UP{4,} DOWN{2,} E)
        DEFINE
        UP AS UP.rideCount > LAST(UP.rideCount, 1) OR
                LAST(UP.rideCount, 1) IS NULL,

                DOWN AS DOWN.rideCount < LAST(DOWN.rideCount, 1) OR
                        LAST(DOWN.rideCount, 1) IS NULL,

        E AS E.rideCount > LAST(DOWN.rideCount)
)
```


video

PageMaker reference standards

Combination of graphics and text, highlighting key elements, trend chart, architecture chart, form, icons, etc.

Contents



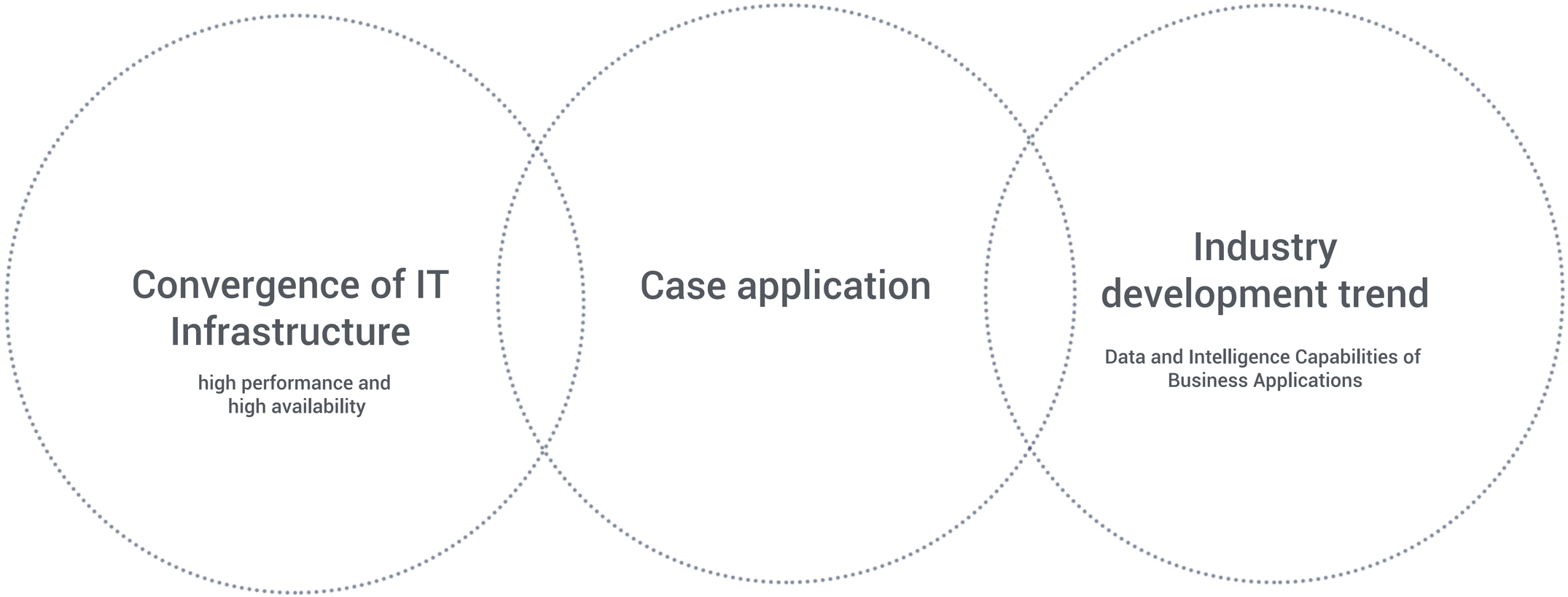
Pravega
Reliable streaming storage

FLINK FORWARD

Data analysis platform based on Apache Flink

01

Data analysis platform based on Apache Flink



Pravega
Reliable streaming storage

FLINK FORWARD

Performance improvement 2,000

FLINK FORWARD

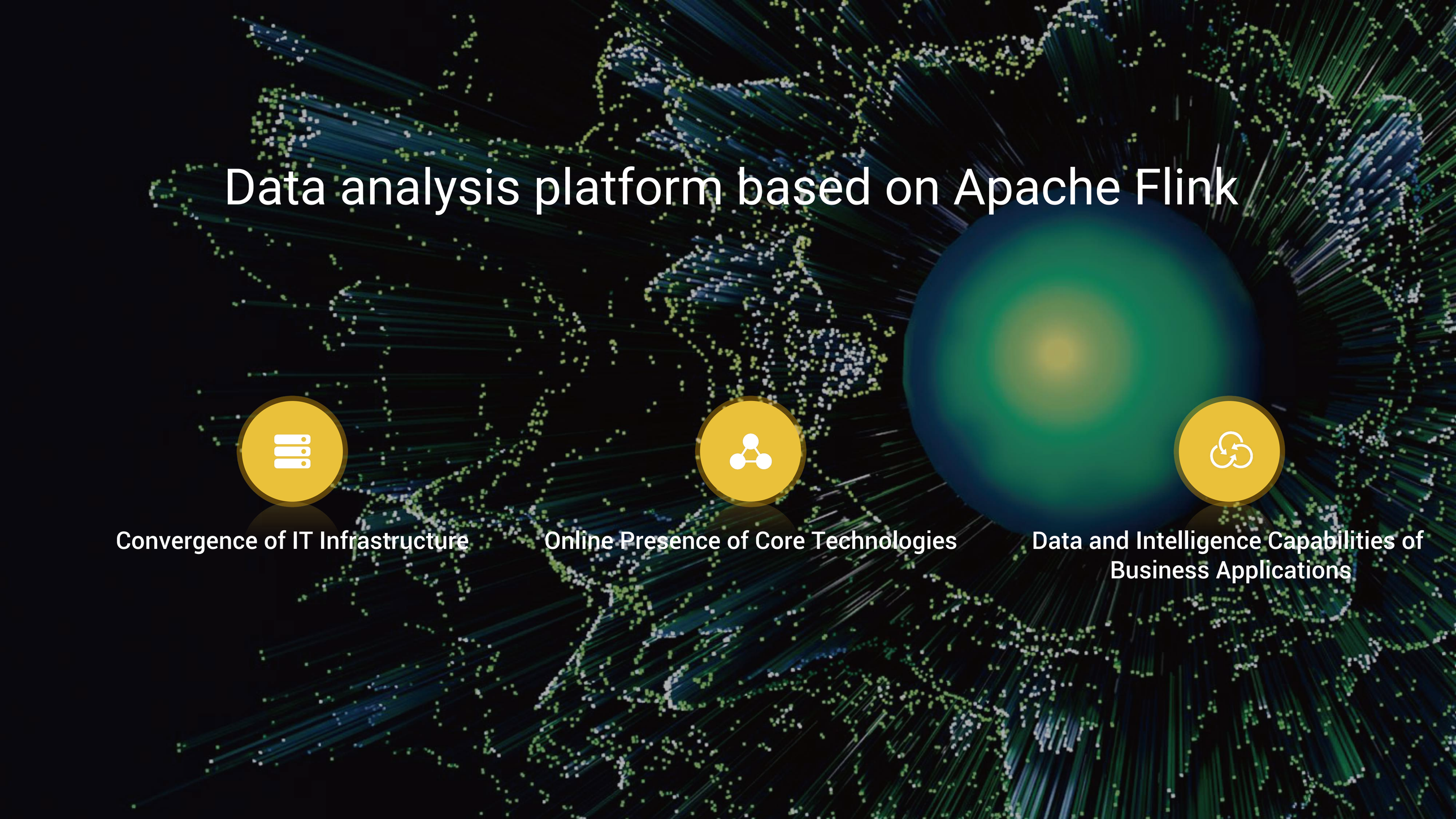
Data analysis platform based on Apache Flink

Convergence of IT Infrastructure

Online Presence of Core Technologies

Data and Intelligence Capabilities of Business Applications

Data analysis platform based on Apache Flink



Convergence of IT Infrastructure

Online Presence of Core Technologies

Data and Intelligence Capabilities of Business Applications

Data analysis platform based on Apache Flink

Chronicle of events

Chronicle of events

2009

Chronicle of events

Chronicle of events

2010

Chronicle of events

Chronicle of events

2011

Chronicle of events

Chronicle of events

2019

Icon recommendation