Introduction to Stateful Stream Processing with Apache Flink

Robert Metzger
@rmetzger_
robert@ververica.com

ververica

VERVERICA
PLATFORM

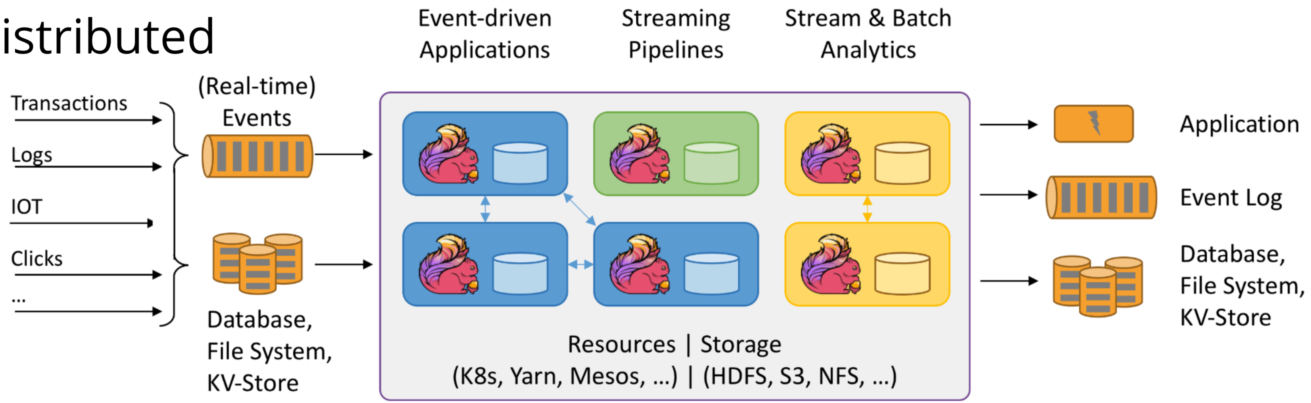
Original creators of
Apache Flink®

Ververica Platform
Open Source Apache Flink
+ Application Manager

Apache Flink 101

Apache Flink

- Apache Flink is an open source stream processing framework
  - Low latency
  - High throughput
  - Stateful
  - Distributed



What is Apache Flink?

Batch Processing

process static and historic data

Data Stream Processing

realtime results from data streams

Event-driven Applications

data-driven actions and services

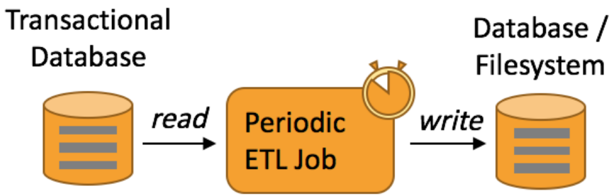
Stateful Computations Over Data Streams

Use Case: Streaming ETL

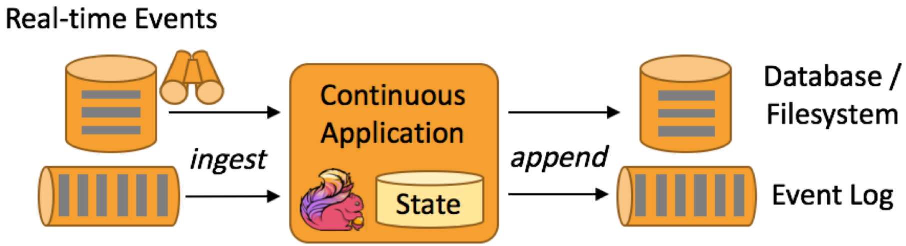
- Periodic ETL is the traditional approach
  - External tool periodically triggers ETL batch job

- Data pipelines continuously move data
  - Ingestion with low latency
  - No artificial data boundaries

Periodic ETL



Data Pipeline / Real-time ETL

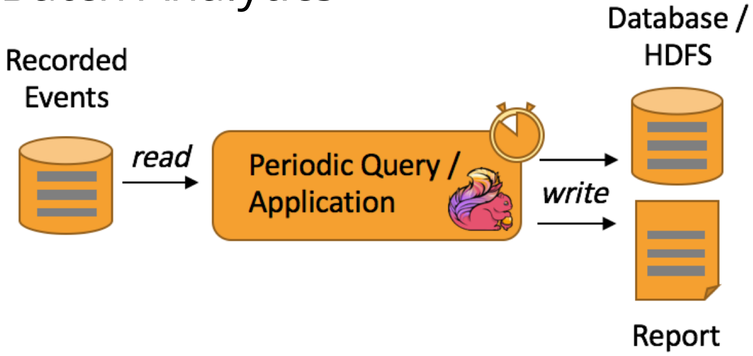


Use Case: Data Analytics

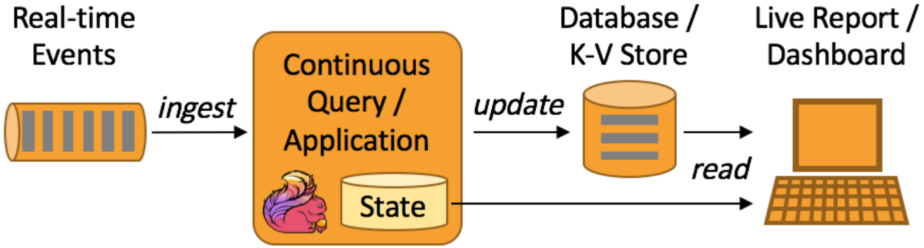
- Batch analytics is great for ad-hoc queries
  - Queries change faster than data
  - Interactive analytics / prototyping

- Stream analytics continuously processes data
  - Data changes faster than queries
  - Live / low latency results
  - No Lambda architecture required!

Batch Analytics



Stream Analytics

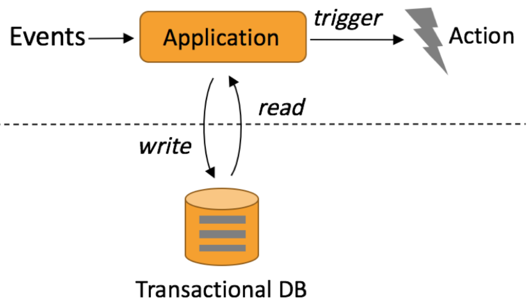


Use Case: Event-driven applications

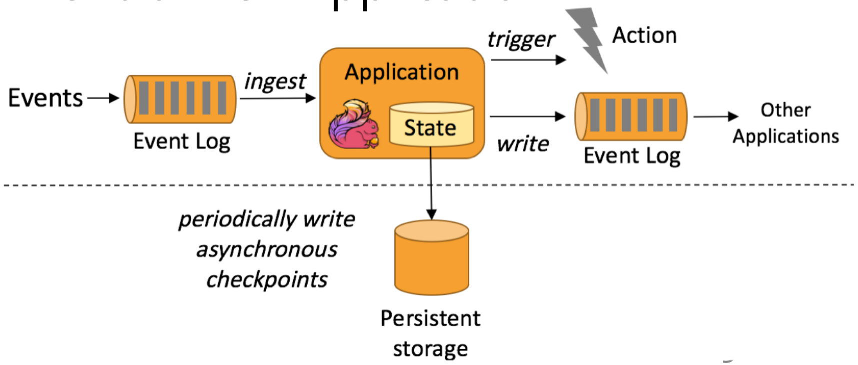
- Traditional application design
  - Compute & data tier architecture
  - React to and process events
  - State is stored in (remote) database

- Event-driven application
  - State is maintained locally
  - Guaranteed consistency by periodic state checkpoints
  - Tight coupling of logic and data (microservice architecture)
  - Highly scalable design

Transactional Application



Event-driven Application



Hardened at scale

Alibaba.com
aws
bouygues TELECOM
Capital One
COMCAST
criteo
D TRB

ebay
ERICSSON
HUAWEI
King
lyft
ING
MUX

otto group
ResearchGate
Telefónica NEXT
Tencent 腾讯
UBER
yelp
zalando

Details about their use cases and more users are listed on Flink's website at https://flink.apache.org/poweredby.html

Case Study: Single’s Day

- Chinese Shopping Festival
- Very high peak load
  - 100s millions records per second
  - 100s thousands payments per second
  - 10 TBs of managed state
  - 10s thousands of cores
- Flink used in various areas in the process incl. payment, shipping, realtime recommendations and the giant dashboard

References

- https://www.ververica.com/blog/singles-day-2018-data-in-a-flink-of-an-eye
- https://medium.com/@alitech_2017/how-to-cope-with-peak-data-traffic-on-11-11-the-secrets-of-alibaba-stream-computing-17d5e807980c

Building blocks

The Core Building Blocks

Event Streams	State	(Event) Time	Snapshots
real-time and hindsight	complex business logic	consistency with out-of-order data and late data	forking / versioning / time-travel

Stateful Event & Stream Processing

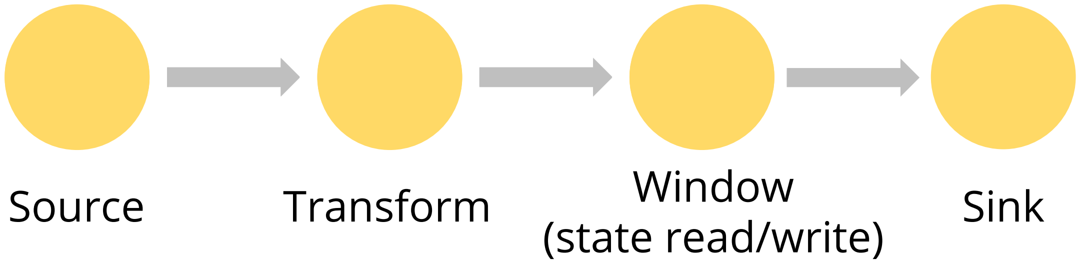
```
val lines: DataStream[String] = env.addSource(new FlinkKafkaConsumer(…))

val events: DataStream[Event] = lines.map((line) => parse(line))

val stats: DataStream[Statistic] = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum(new MyAggregationFunction())

stats.addSink(new RollingSink(path))
```

Source

Transformation

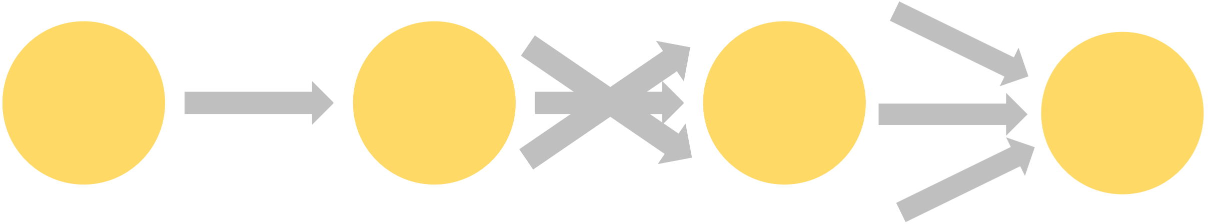
Transformation

Sink



Streaming Dataflow

Stateful Event & Stream Processing



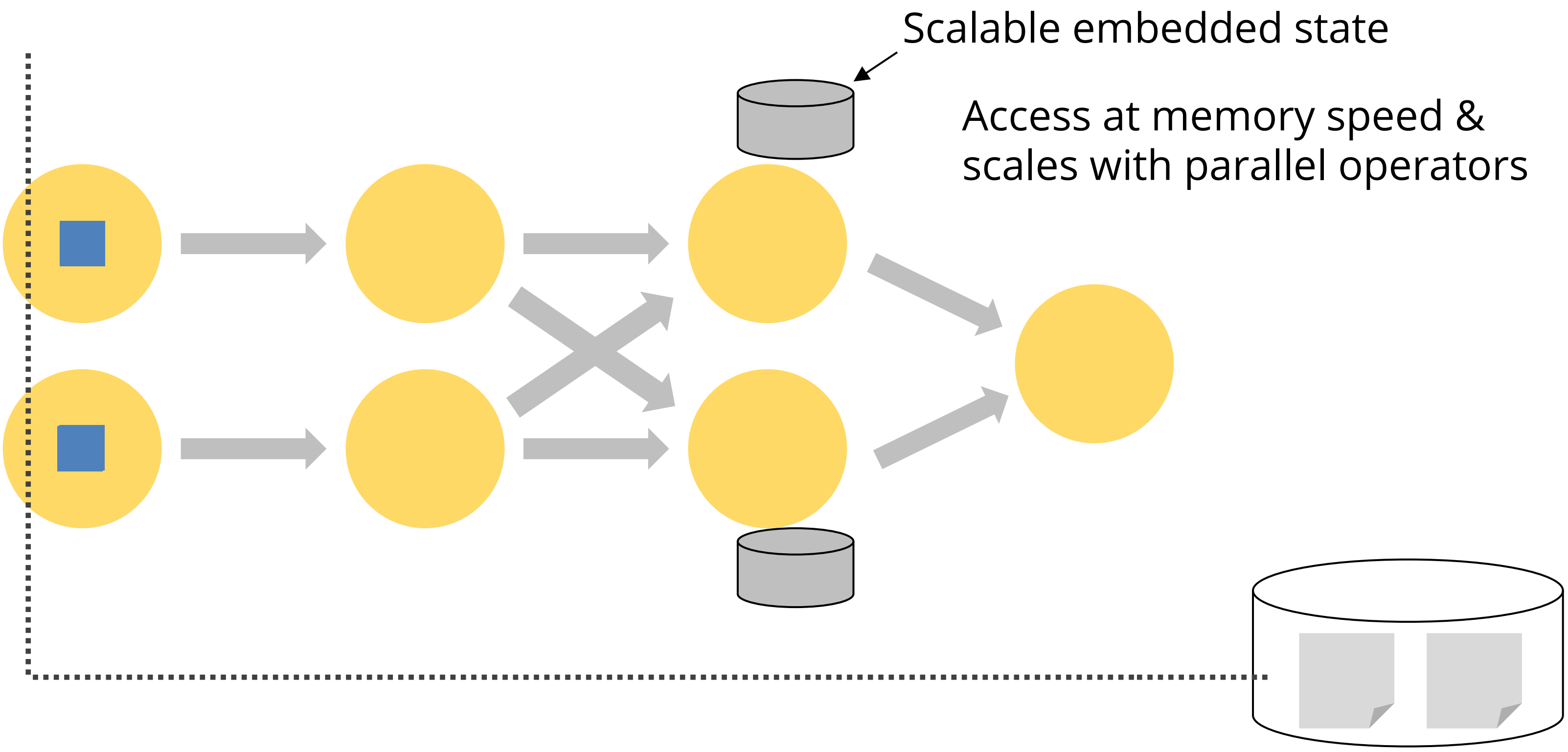
Source

Filter / Transform

State read/write

Sink

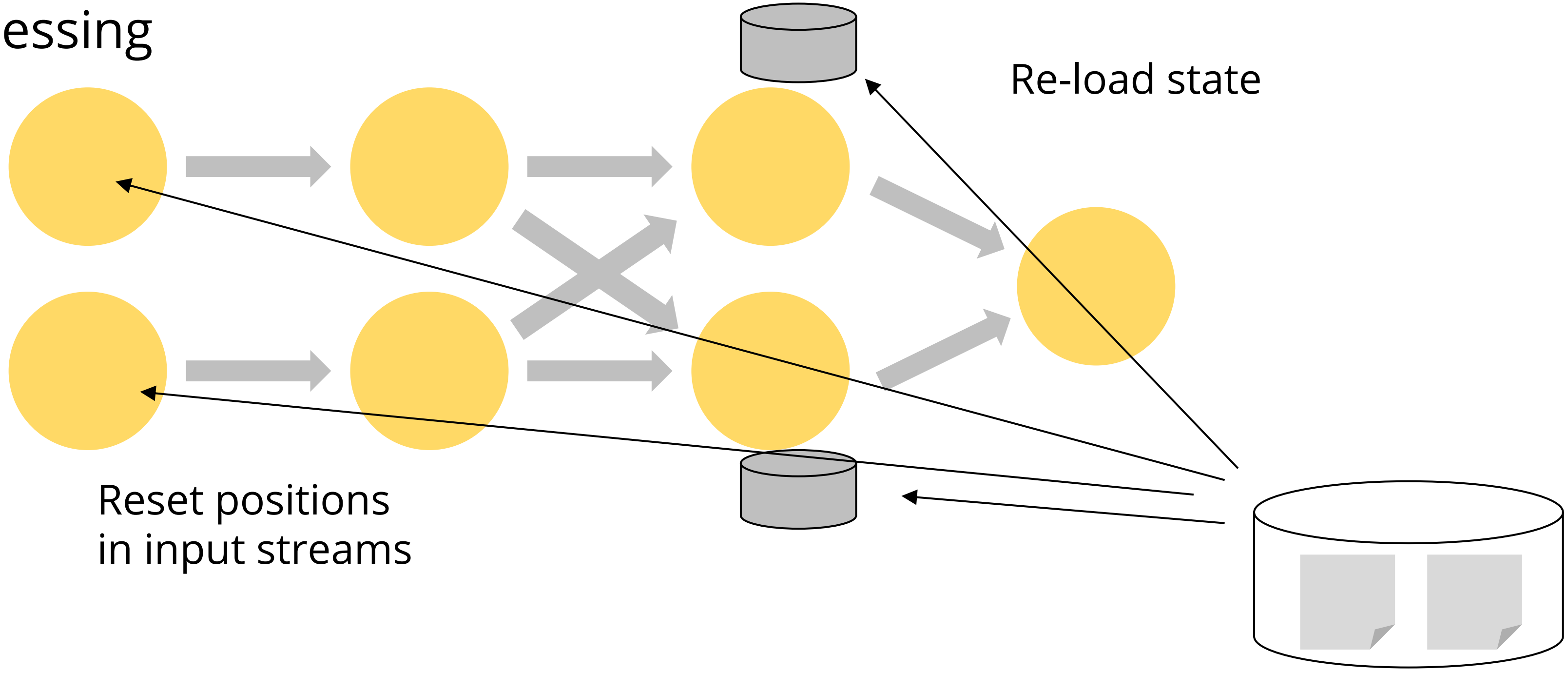
Stateful Event & Stream Processing



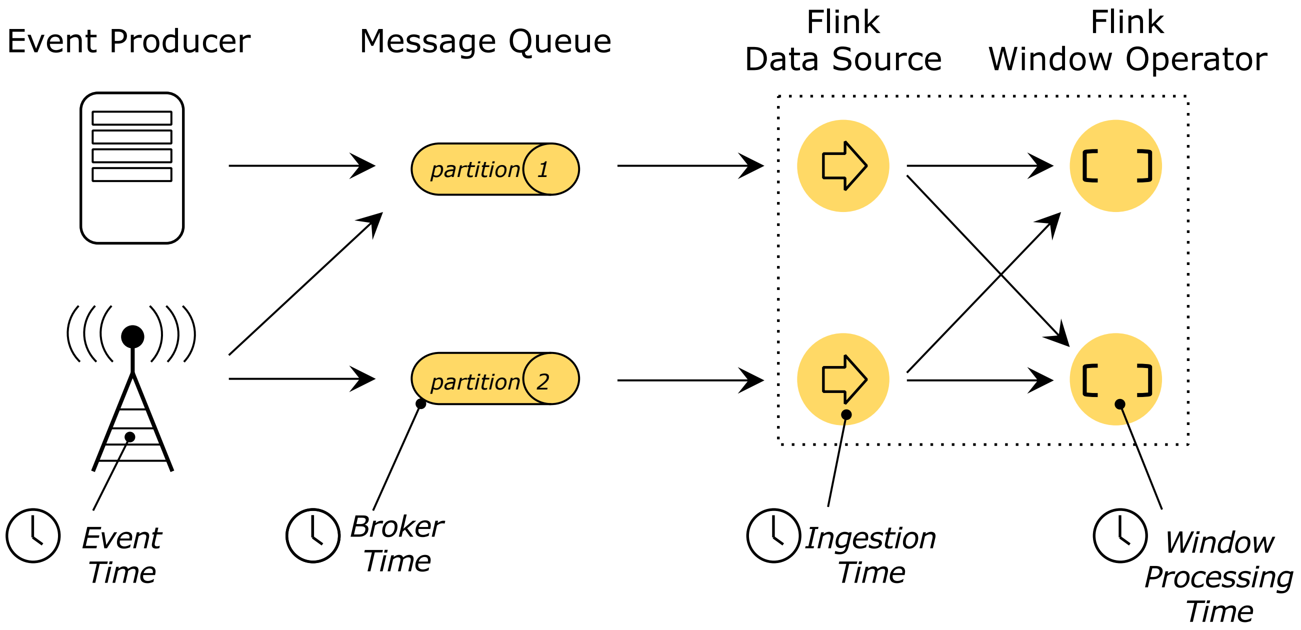
Stateful Event & Stream Processing

Rolling back computation

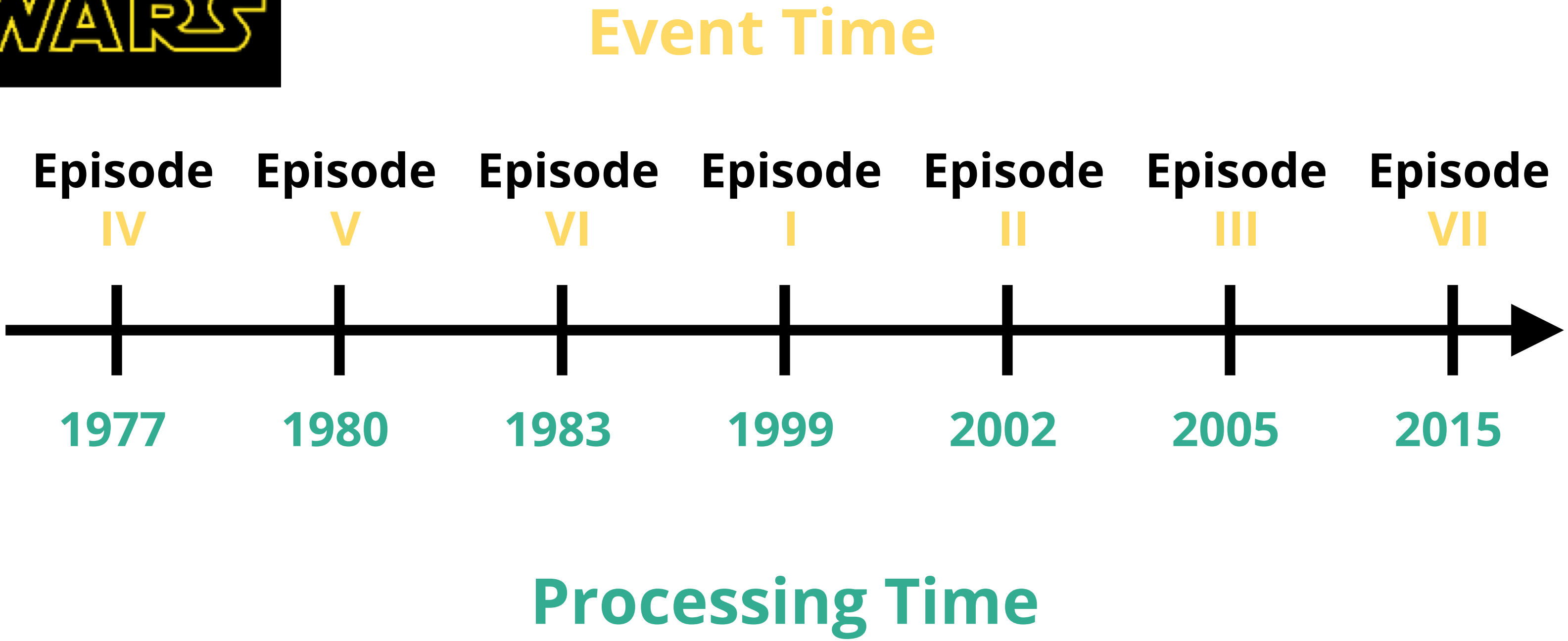
Re-processing



Time: Different Notions of Time



Time: Event Time Example

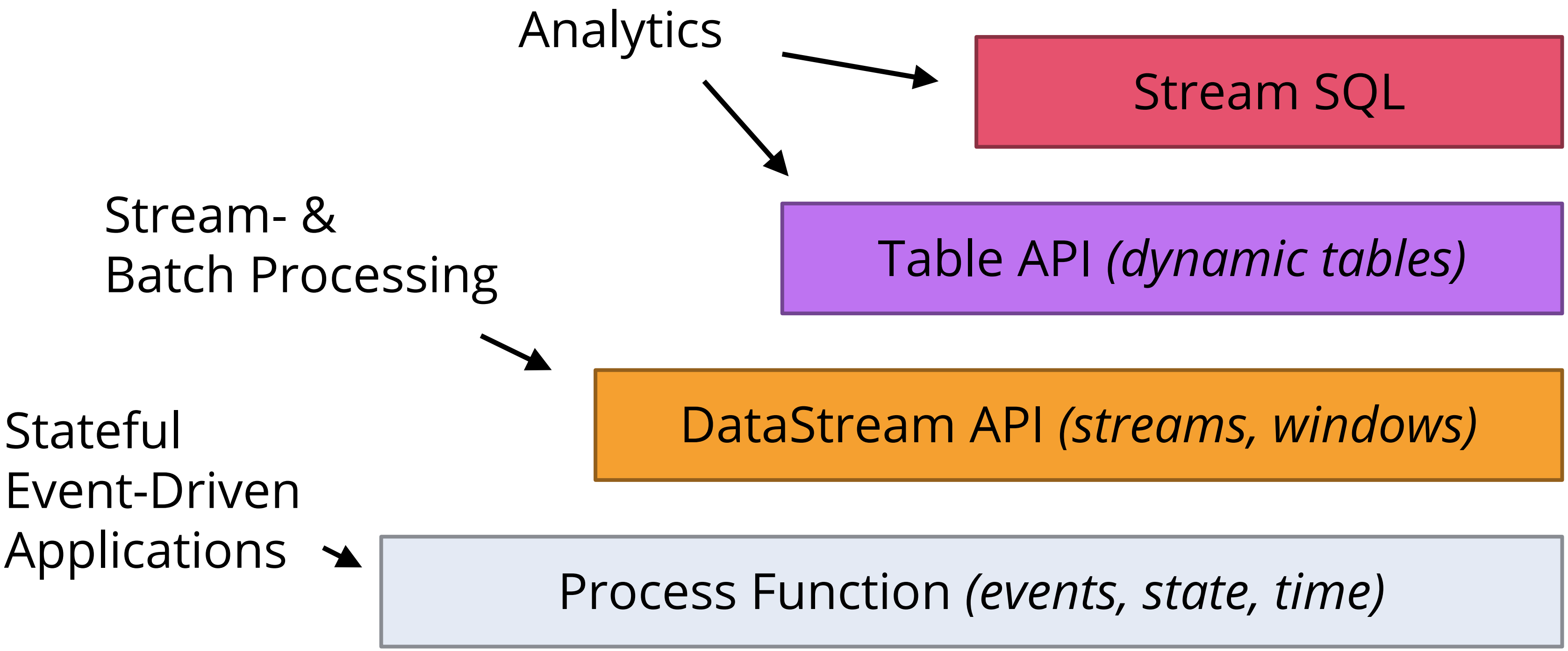


Recap: The Core Building Blocks

Event Streams	State	(Event) Time	Snapshots
real-time and hindsight	complex business logic	consistency with out-of-order data and late data	forking / versioning / time-travel

APIs

The APIs


Analytics
Stream SQL
Stream- &
Batch Processing
Table API *(dynamic tables)*
Stateful
Event-Driven
Applications
DataStream API *(streams, windows)*
Process Function *(events, state, time)*

Process Function

```
class MyFunction extends ProcessFunction[MyEvent, Result] {

    // declare state to use in the program
    lazy val state: ValueState[CountWithTimestamp] = getRuntimeContext().getState(…)

    def processElement(event: MyEvent, ctx: Context, out: Collector[Result]): Unit = {
        // work with event and state
        (event, state.value) match { … }

        out.collect(…) // emit events
        state.update(…) // modify state

        // schedule a timer callback
        ctx.timerService.registerEventTimeTimer(event.timestamp + 500)
    }

    def onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[Result]): Unit = {
        // handle callback when event-/processing- time instant is reached
    }
}
```

Data Stream API

```
val lines: DataStream[String] = env.addSource(
                                    new FlinkKafkaConsumer<>(…))

val events: DataStream[Event] = lines.map((line) => parse(line))

val stats: DataStream[Statistic] = stream
   .keyBy("sensor")
   .timeWindow(Time.seconds(5))
   .sum(new MyAggregationFunction())

stats.addSink(new RollingSink(path))
```

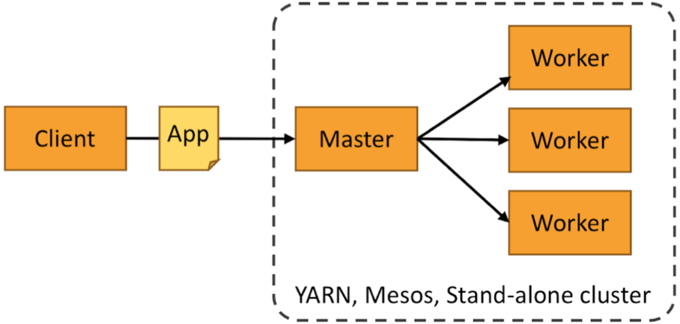
Table API & Stream SQL

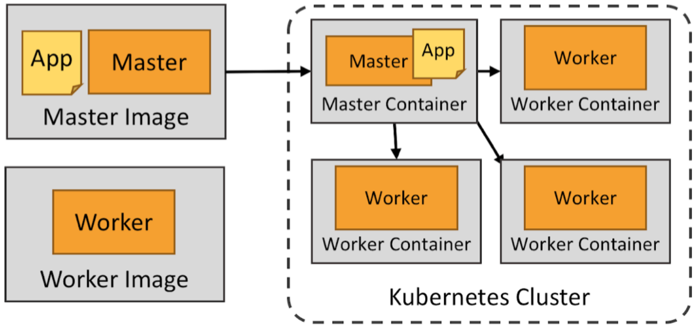
```scala
// Table API
val tapiResult: Table = tEnv.scan("sensors")   // scan sensors table
 .window(Tumble over 1.hour on 'rowtime as 'w) // define 1-hour window
 .groupBy('w, 'room)                           // group by window and room
 .select('room, 'w.end, 'temp.avg as 'avgTemp) // compute average temperature
```

```sql
SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp) AS avgTemp
FROM sensors
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```

Deployment & Integrations

Deployment





kubernetes
hadoop YARN
aws
docker
MESOS
eventador.io

Integrations

- Event logs:
  - Kafka, Kinesis, Pulsar*
- File systems:
  - S3, HDFS, NFS, MapR FS, ...
- Encodings:
  - Avro, JSON, CSV, ORC, Parquet
- Databases:
  - JDBC, HCatalog
- Key-Value Stores
  - Cassandra, Elasticsearch, Redis*

Concluding...

The Apache Flink® Conference

Berlin | October 7-9, 2019

FLINK FORWARD

Organized by ververica

Early Bird ticket sales ends July 15th

flink-forward.org

#flinkforward

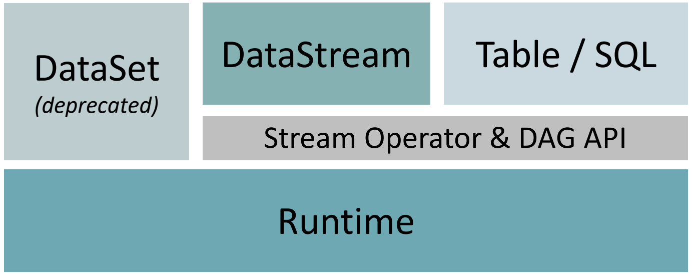
Q & A

Get in touch via eMail:
robert@ververica.com
info@ververica.com

Get in touch via Twitter:
@rmetzger_
@ApacheFlink

What's happening in Flink these days …

- INSERT INTO flink_sql SELECT * FROM blink_sql
  - Turning Table API into an API unified across batch and streaming (FLINK-11439)
  - Integration with Hive ecosystem (FLINK-10556)

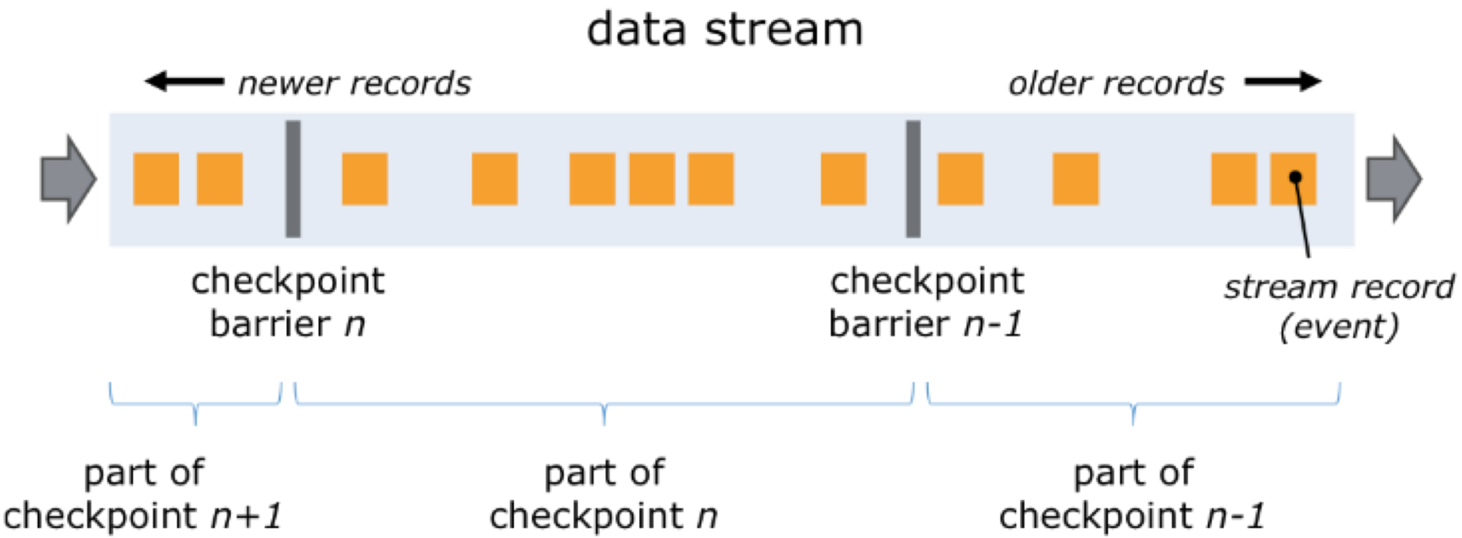


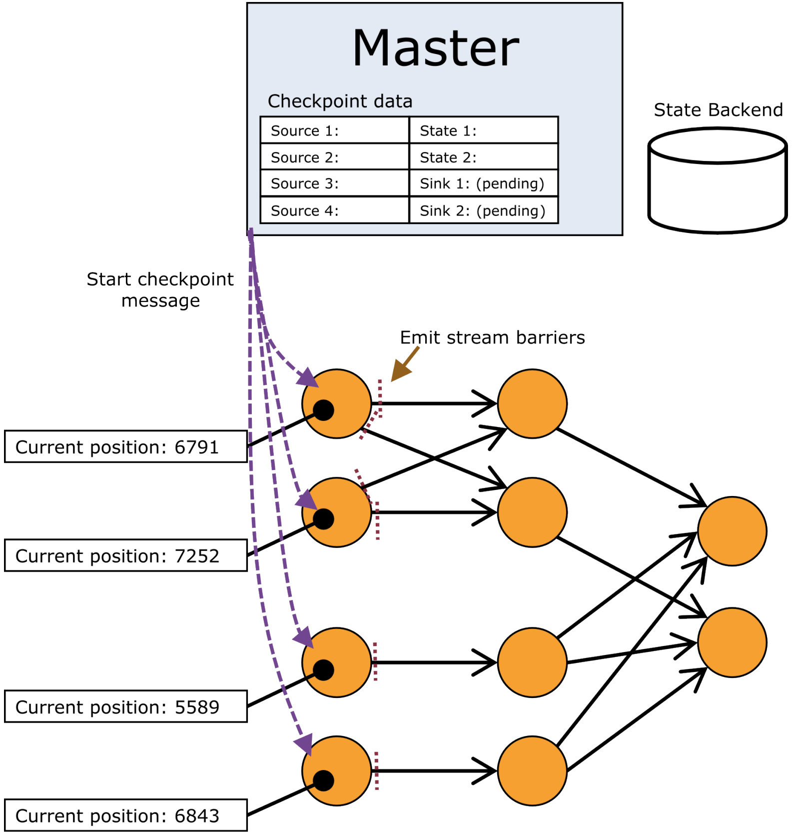
- Batch runtime improvements: Fine-grained recovery (FLINK-4256), more schedulers (FLINK-10429), pluggable shuffle service (FLINK-10653)
- Machine Learning Pipelines on Table API (FLIP-39)
- Table API: Caching of intermediate results (.cache() API) (FLINK-11199)
- Table API: Python support (FLINK-12308)

Implementation: State Checkpointing

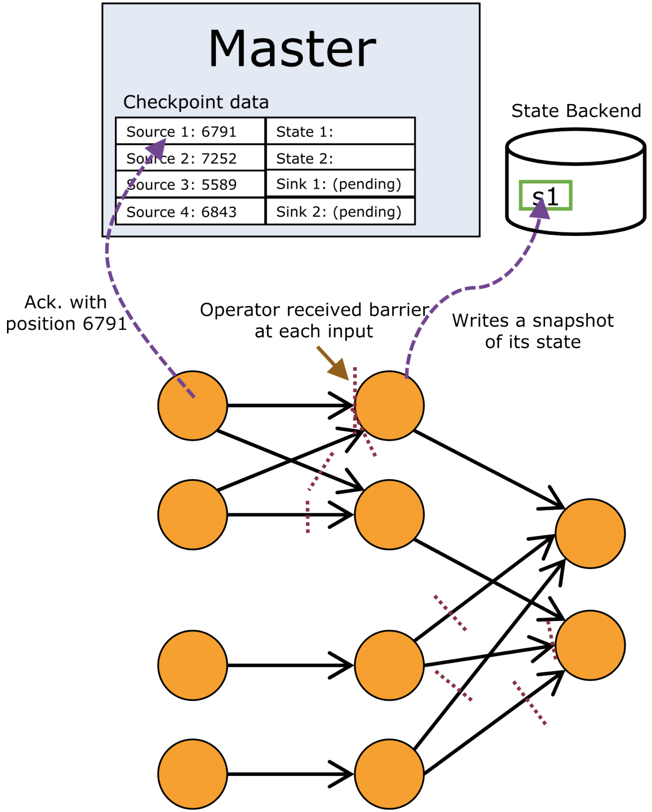
State, Snapshots, Recovery

Coordination via markers, injected into the streams

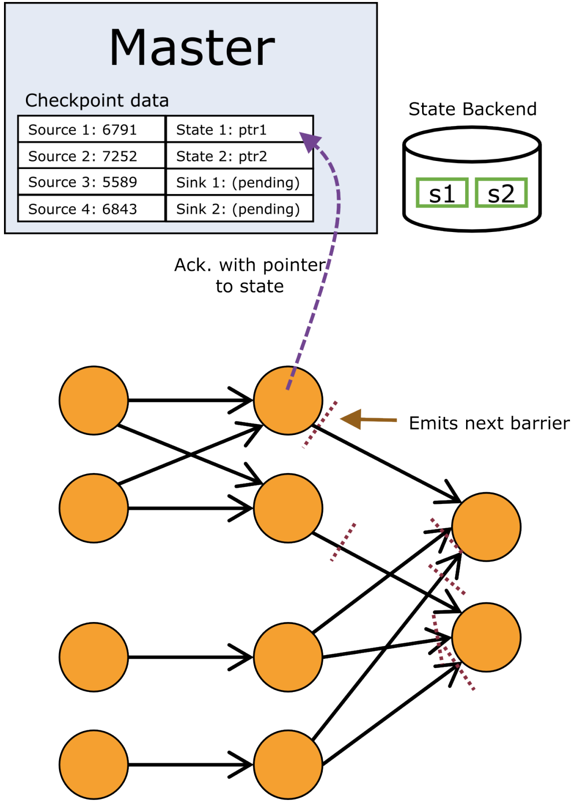




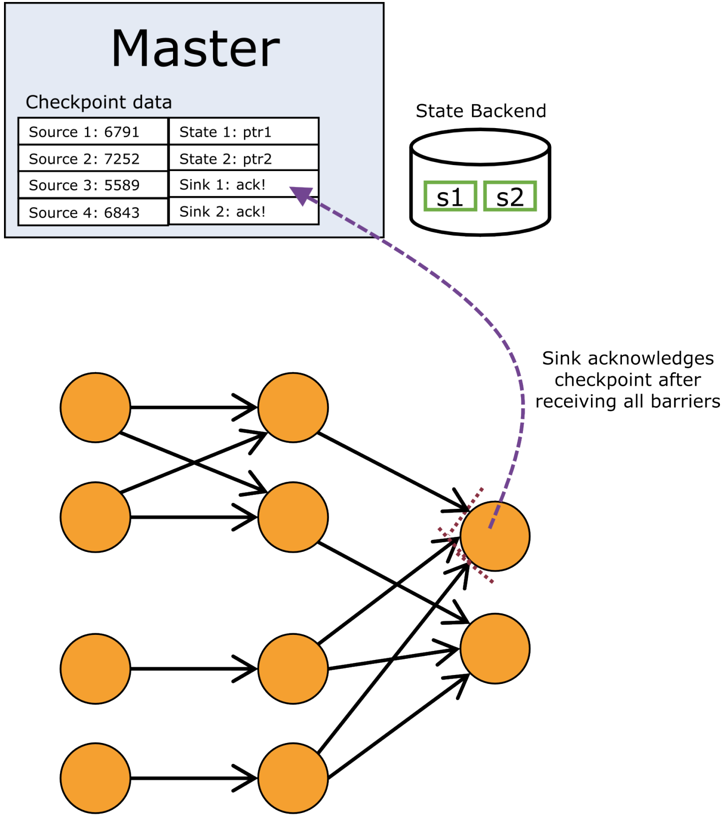
Starting Checkpoint



Checkpoint in Progress



Checkpoint in Progress



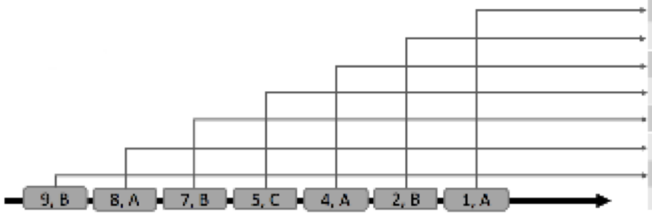
Checkpoint Completed

Implementation: Stream SQL

Stream SQL: Intuition

Stream / Table Duality

Table without Primary Key

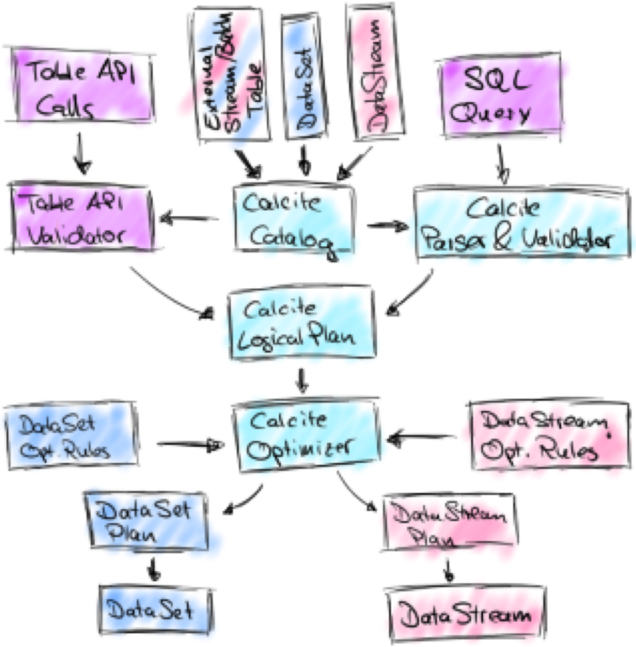


time	k
1	A
2	B
4	A
5	C
7	B
8	A
9	B
..	..

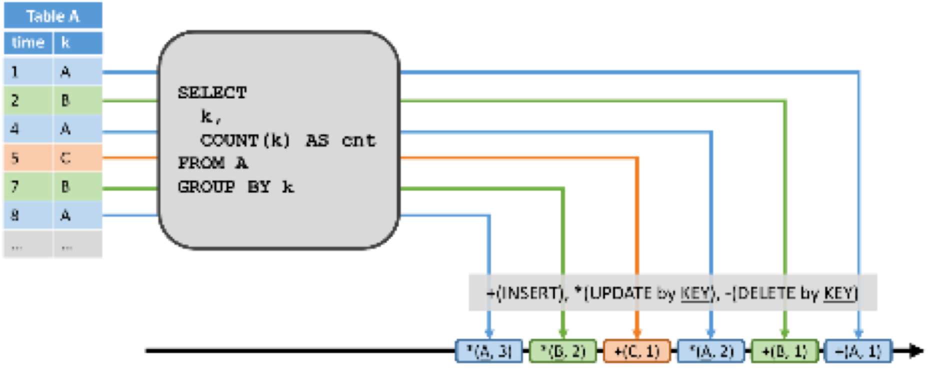
Table with Primary Key

time	k
8	A
9	B
5	C
...	...

Stream SQL: Implementation



Query Compilation



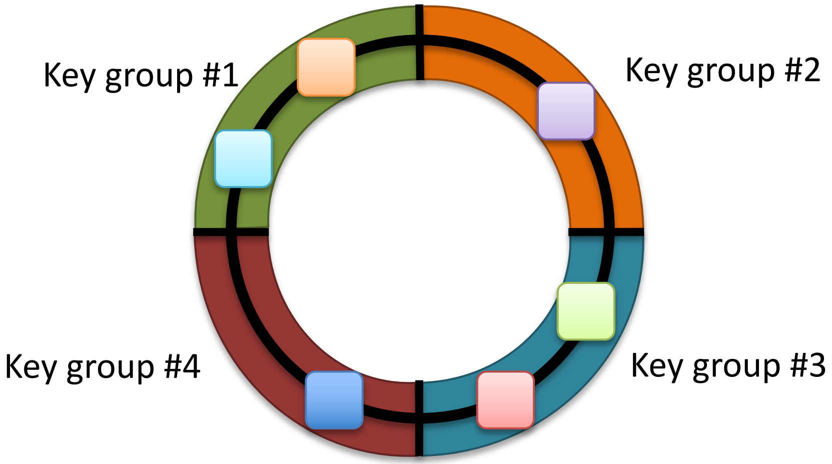
Differential Computation
(add/mod/del)

Implementation: Rescaling

Rescaling State / Elasticity

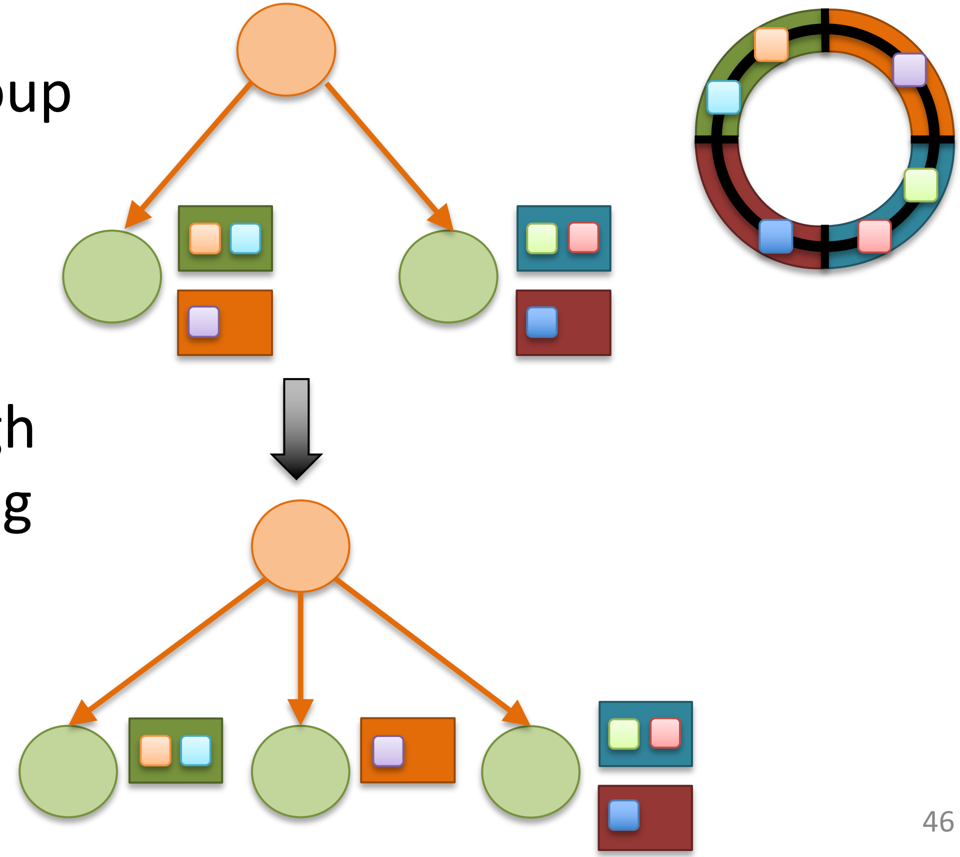
- Similar to consistent hashing
- Split key space into key groups
- Assign key groups to tasks

Key space



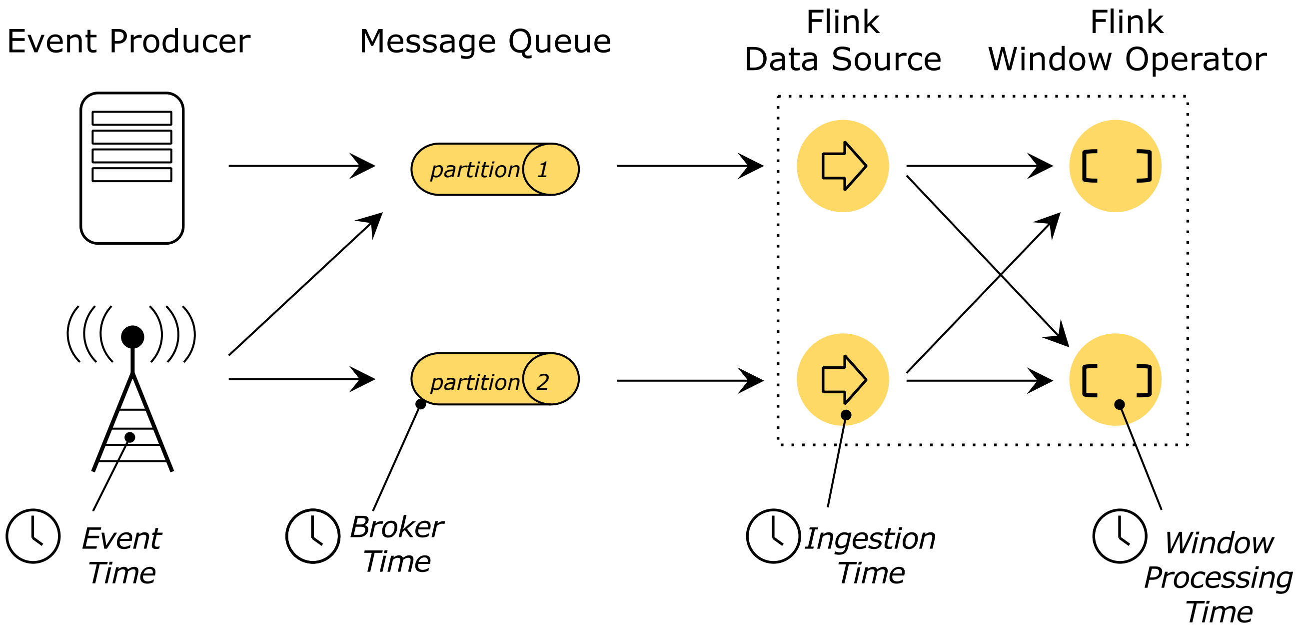
Rescaling State / Elasticity

- Rescaling changes key group assignment
- Maximum parallelism defined by #key groups
- Rescaling happens through restoring a savepoint using the new parallism

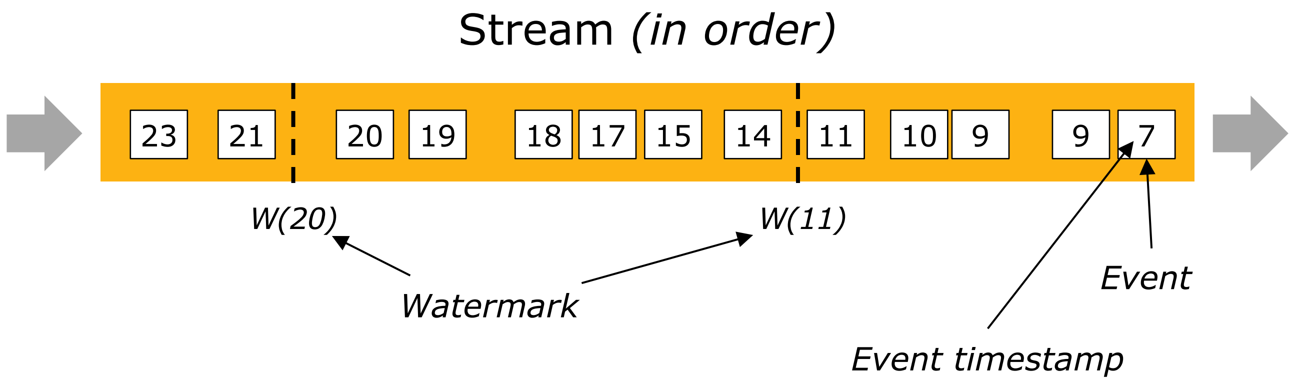


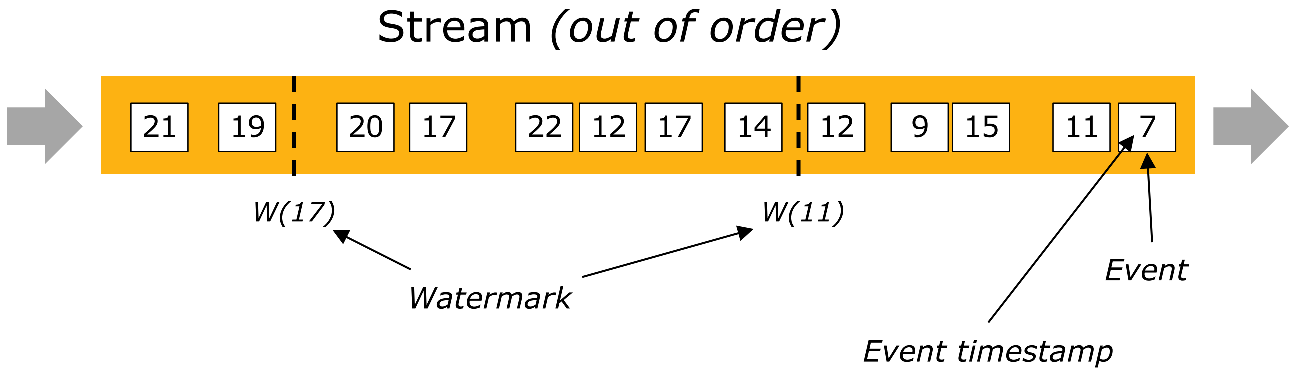
Implementation: Time-handling

Time: Different Notions of Time

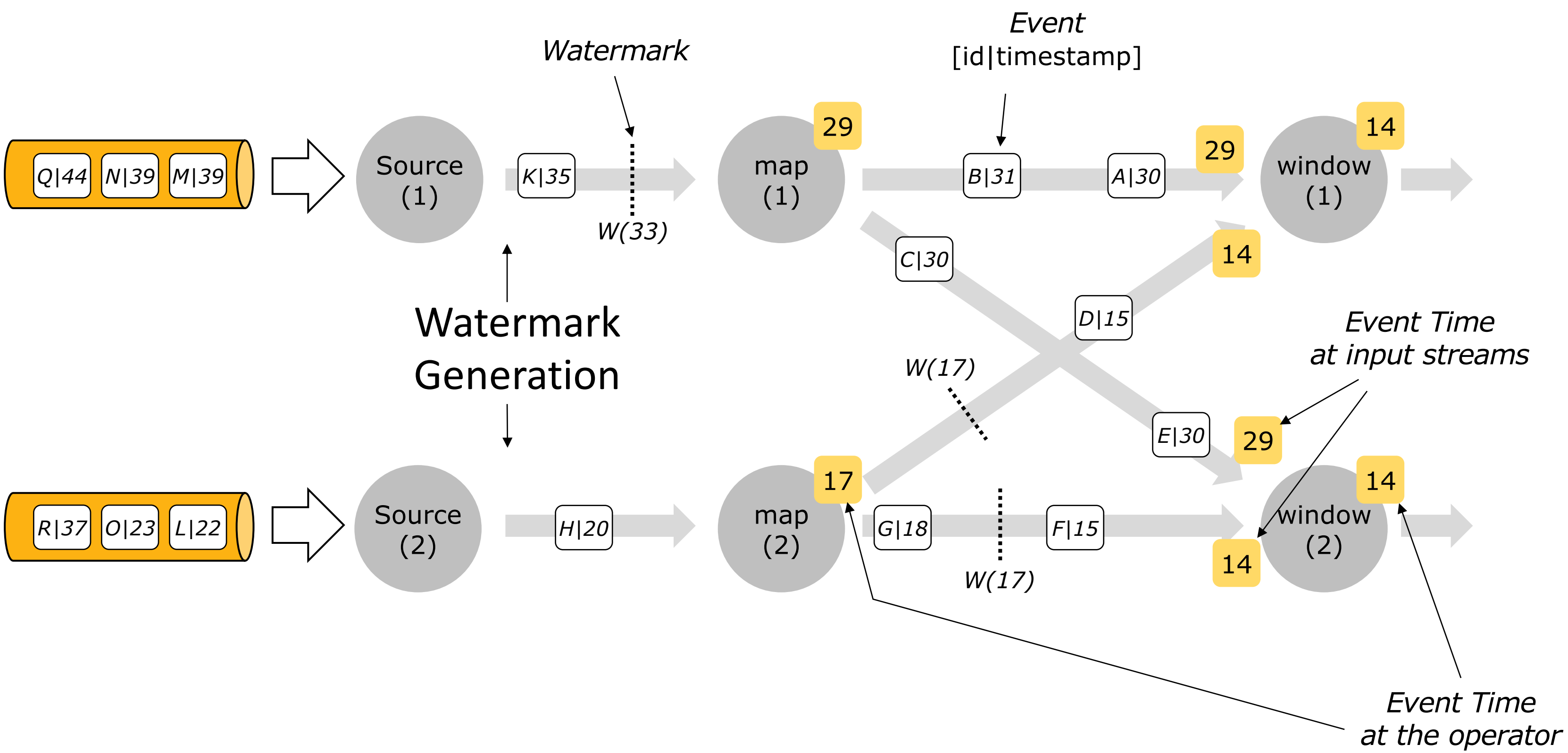


Time: Watermarks



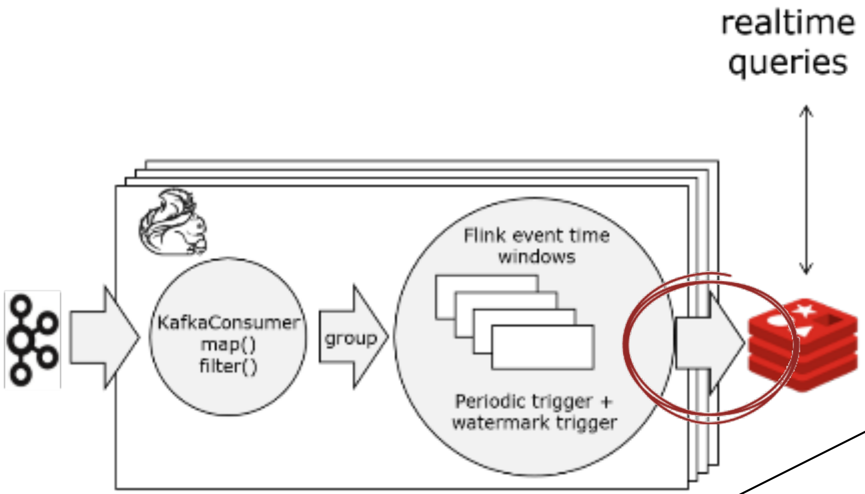


Time: Watermarks in Parallel

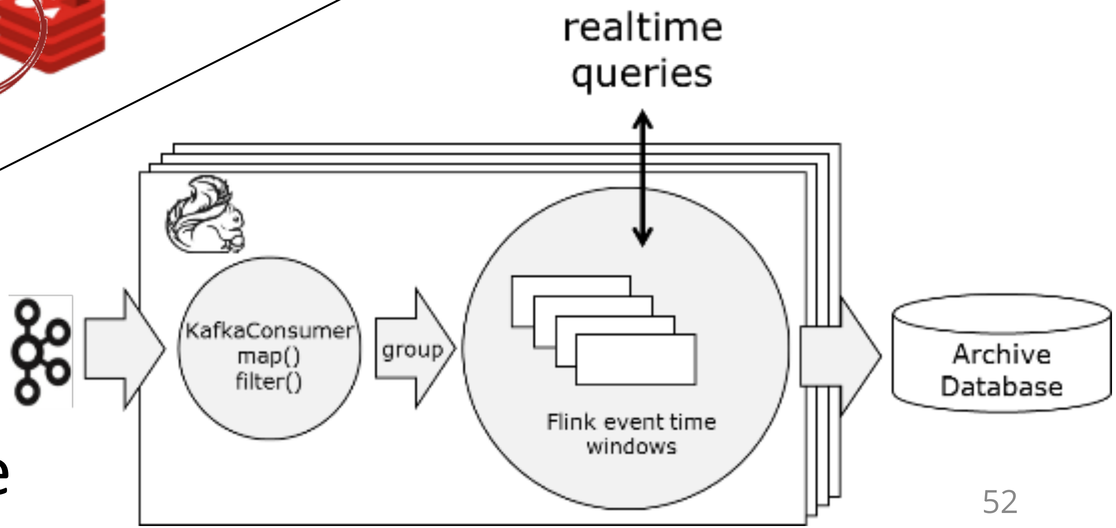


Implementation: Queryable State

Queryable State



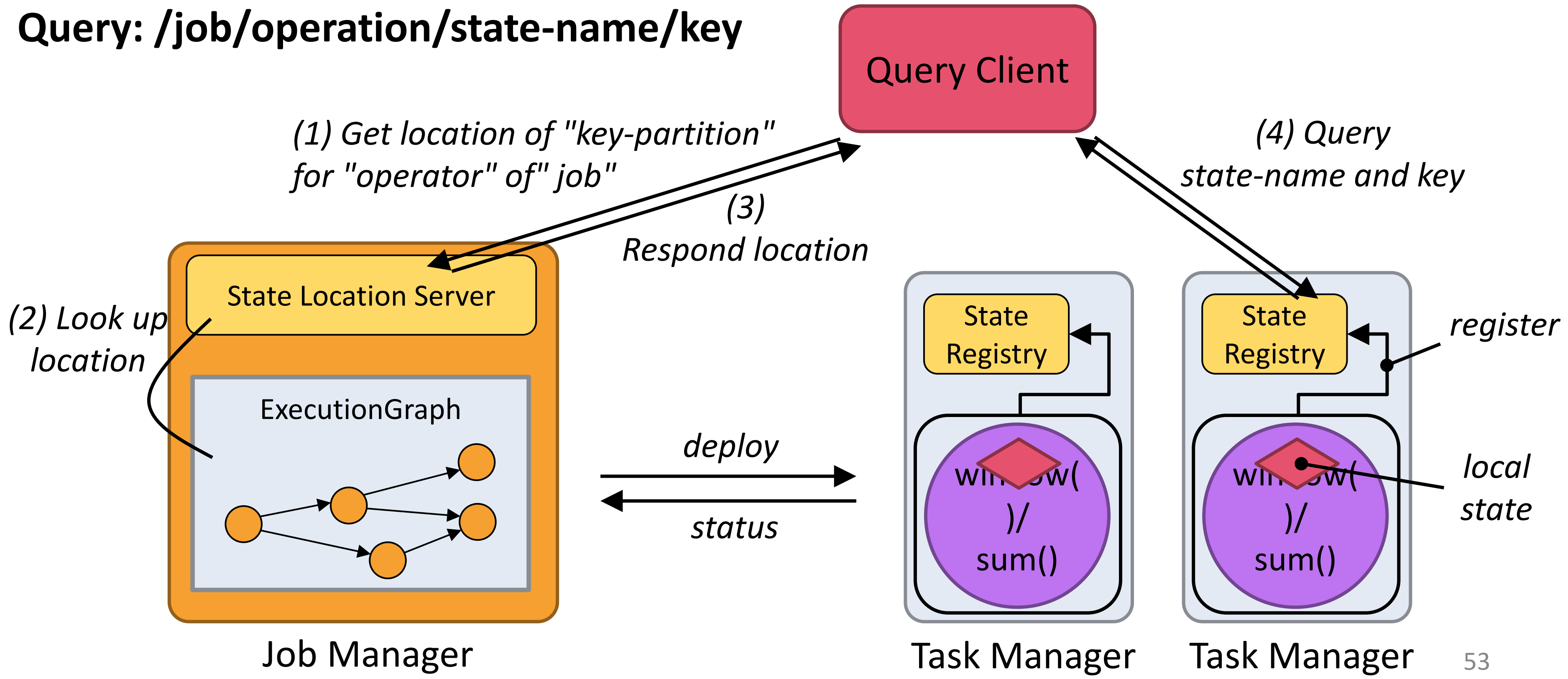
Traditional



Queryable State

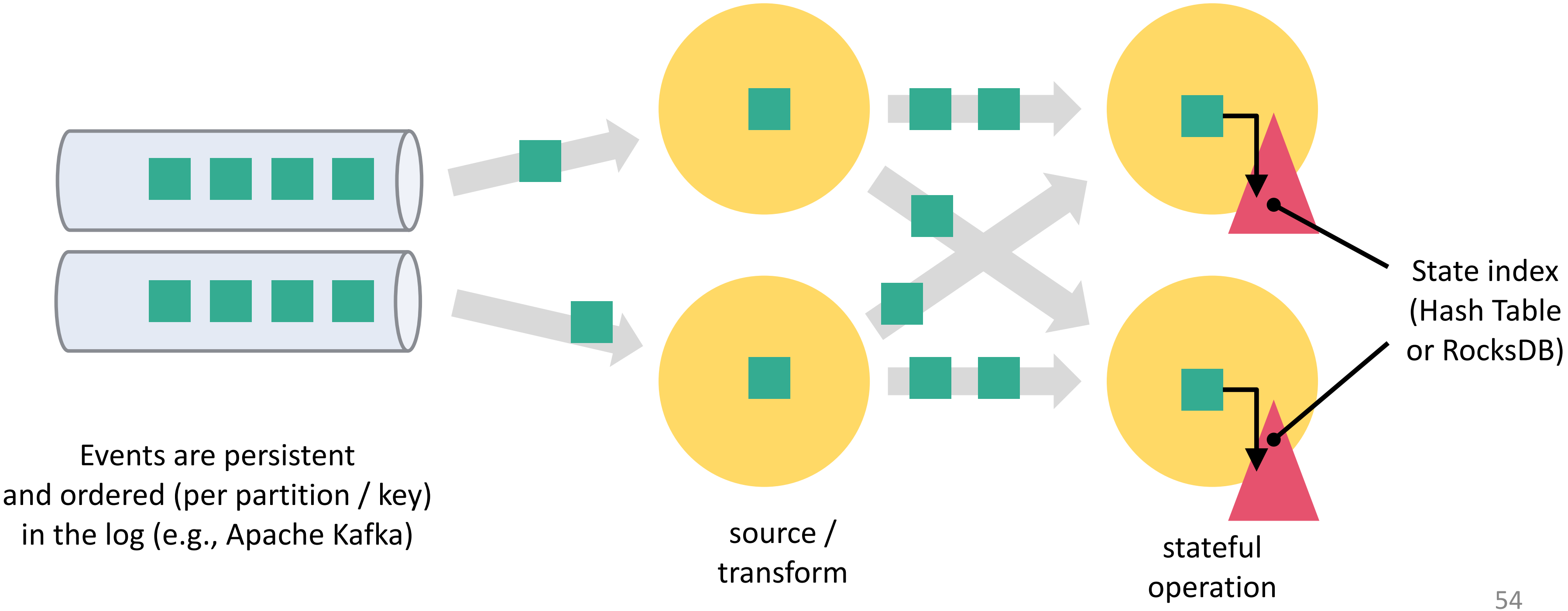
Queryable State: Implementation

Query: /job/operation/state-name/key

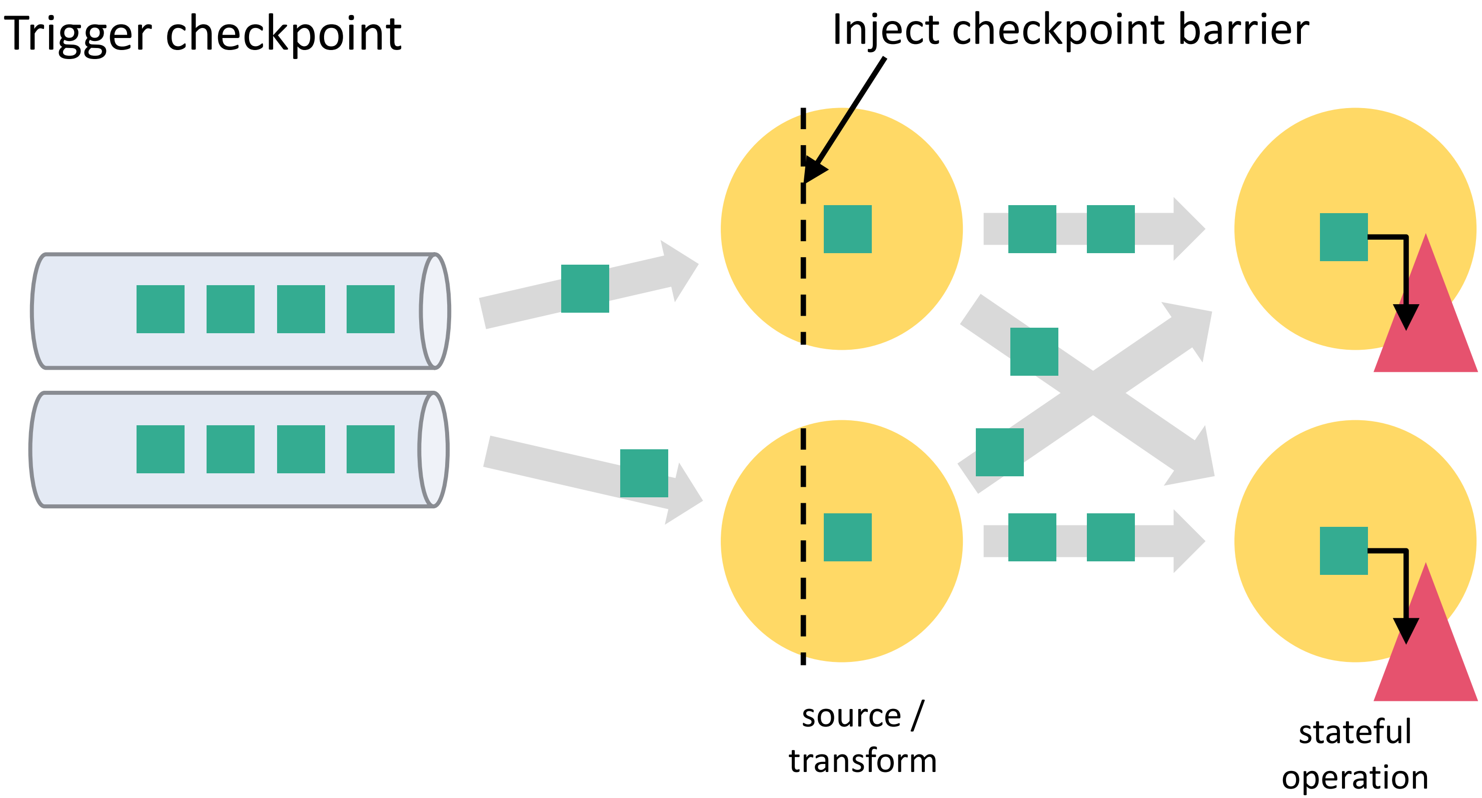


State, Snapshots, Recovery

Events flow without replication or synchronous writes

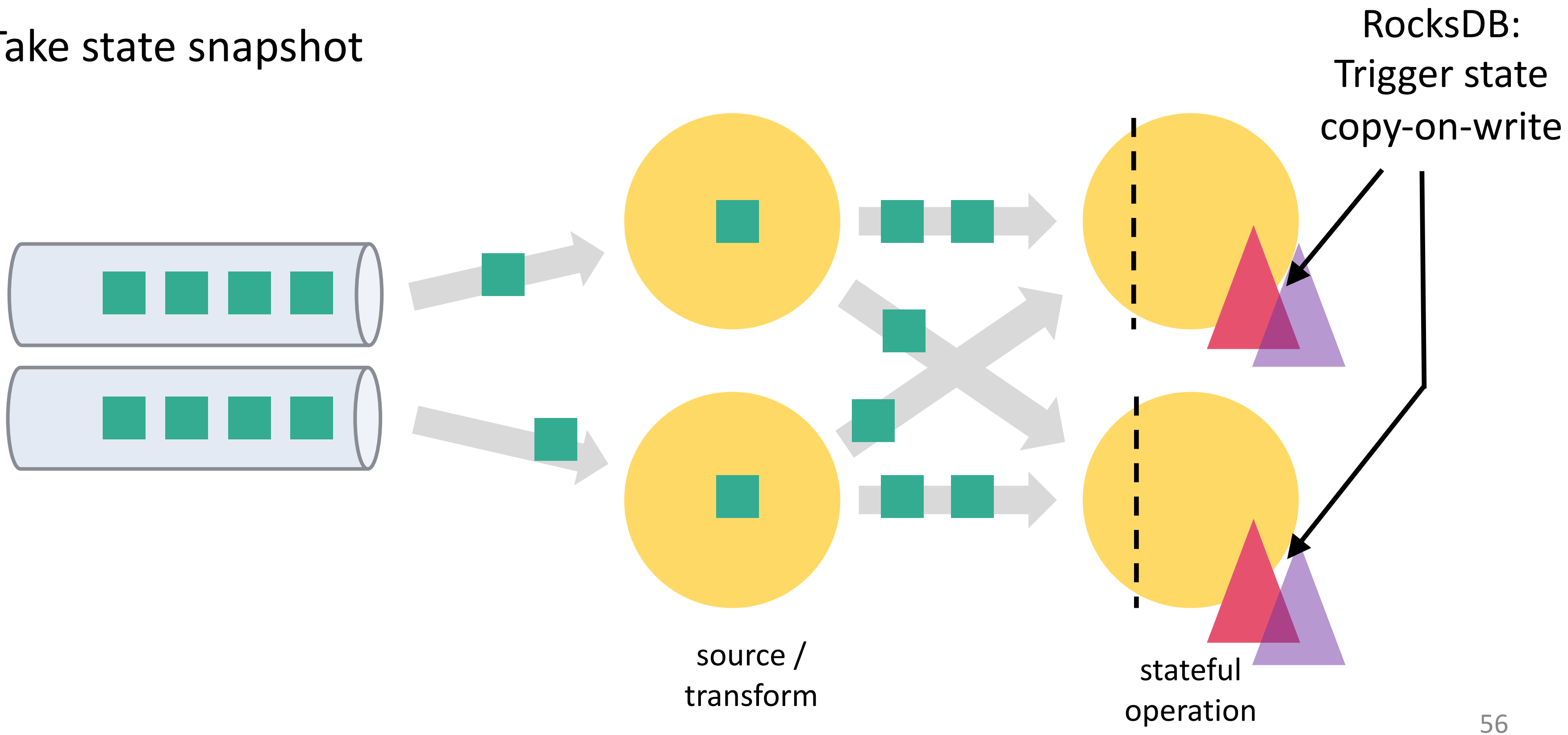


State, Snapshots, Recovery



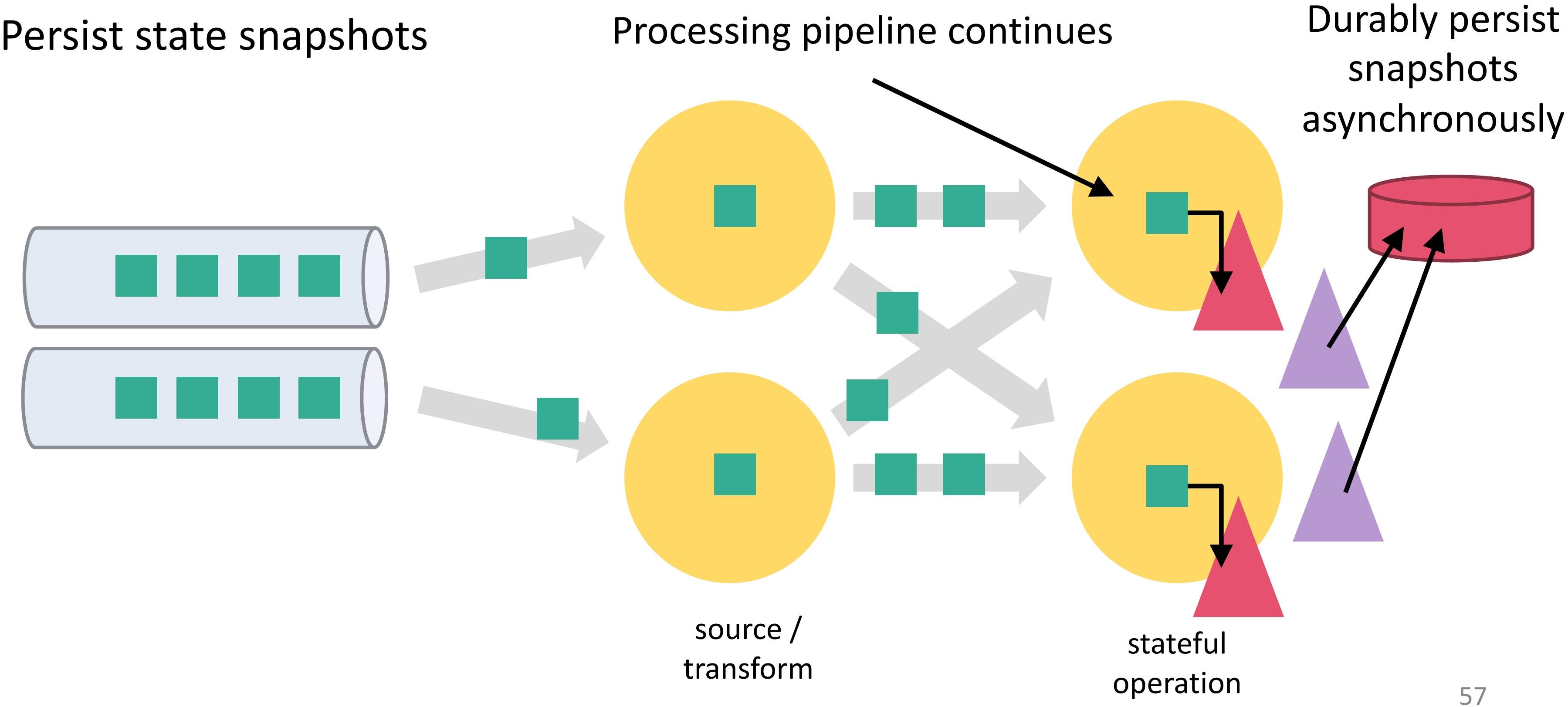
State, Snapshots, Recovery

Take state snapshot



State, Snapshots, Recovery

Persist state snapshots



Powerful Abstractions

Layered abstractions to
navigate simple to complex use cases

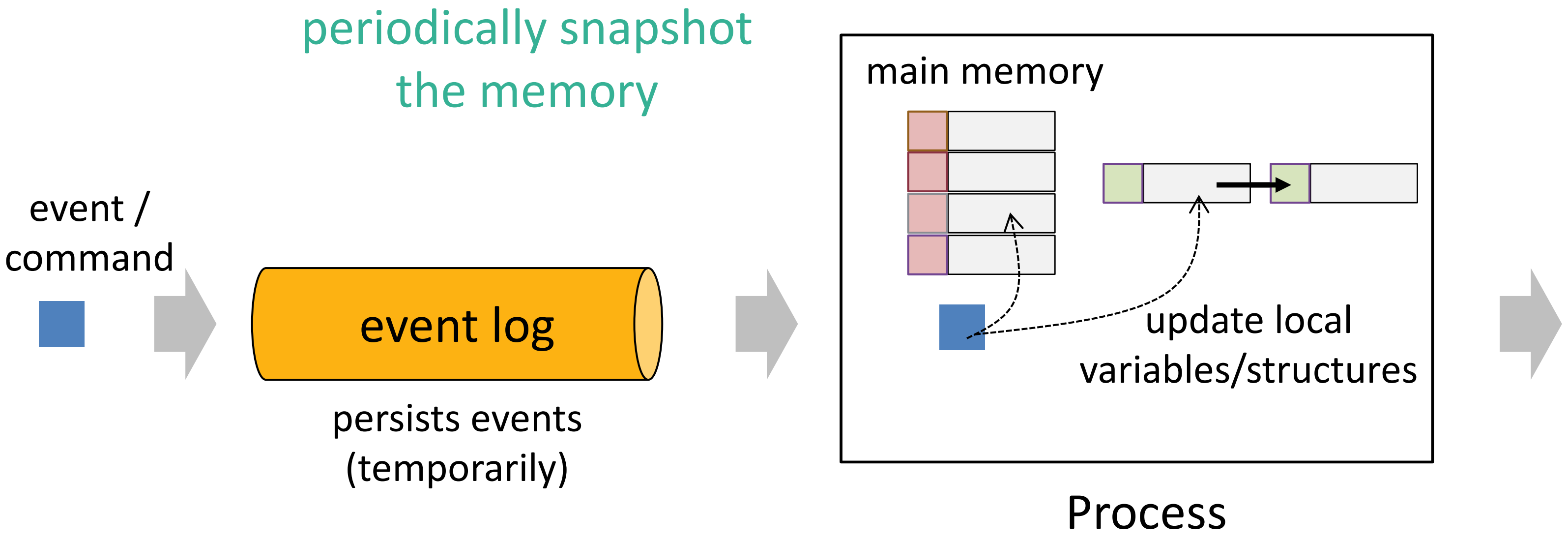
```
SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp)
FROM sensors
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```

High-level Analytics API	Stream SQL / Tables *(dynamic tables)*
Stream- & Batch Data Processing	DataStream API *(streams, windows)*
Stateful Event-Driven Applications	Process Function *(events, state, time)*

```
val stats = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum((a, b) -> a.add(b))
```

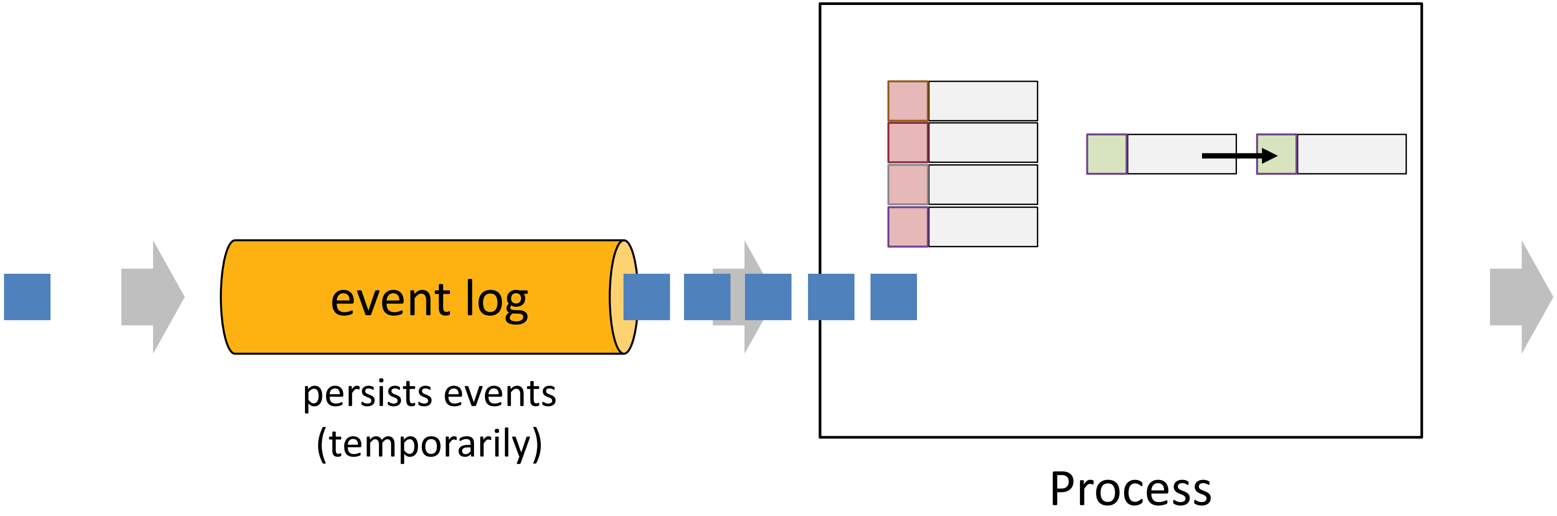
```
def processElement(event: MyEvent, ctx: Context, out: Collector[Result]) = {
  // work with event and state
  (event, state.value) match { … }

  out.collect(…) // emit events
  state.update(…) // modify state

  // schedule a timer callback
  ctx.timerService.registerEventTimeTimer(event.timestamp + 500)
}
```

Event Sourcing + Memory Image



Event Sourcing + Memory Image

Recovery: Restore snapshot and replay events since snapshot



Distributed Memory Image

Distributed application, many memory images.
Snapshots are all consistent together.