




Fabian Hueske
fhueske@apache.org

 @fhueske

Till Rohrmann
trohrmann@apache.or

 @stsffap

Streaming Analytics & CEP

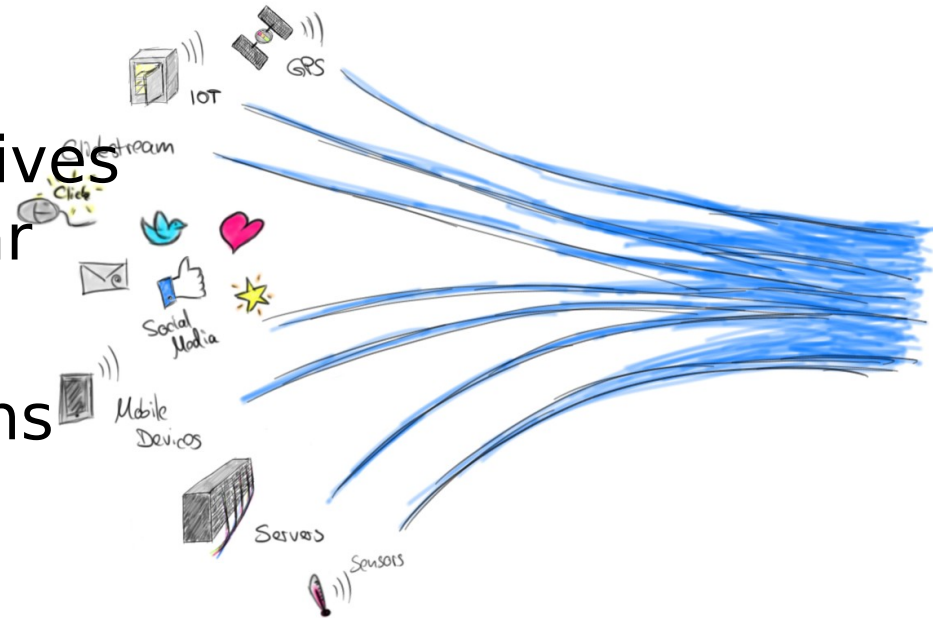
Two sides of the same coin?

dataArtisans

Streams are Everywhere



- Most data is continuously produced as stream
- Processing data as it arrives is becoming very popular
- Many diverse applications and use cases



Complex Event Processing

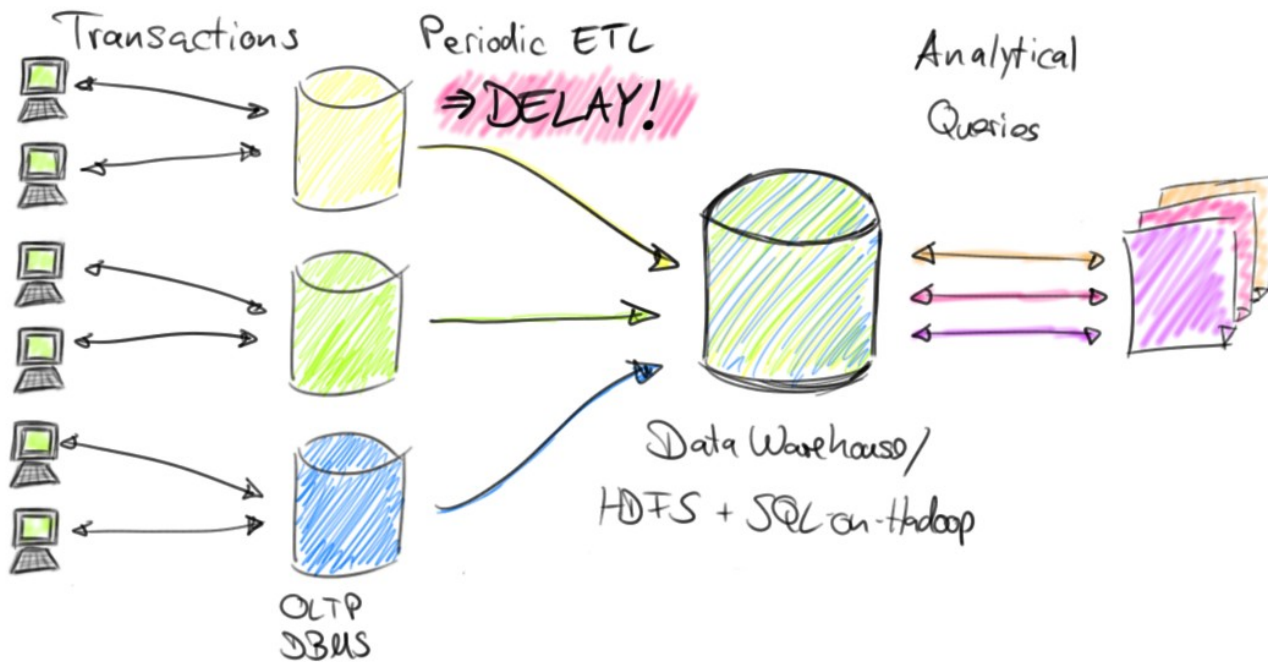


- Analyzing a stream of events and drawing conclusions
 - Detect patterns and assemble new events
- Applications
 - Network intrusion
 - Process monitoring
 - Algorithmic trading
- Demanding requirements on stream processor
 - Low latency!
 - Exactly-once semantics & event-time support

Batch Analytics



- The batch approach to data analytics



Streaming Analytics



- Online aggregation of streams
 - No delay - Continuous results
- Stream analytics subsumes batch analytics
 - Batch is a finite stream
- Demanding requirements on stream processor
 - High throughput
 - Exactly-once semantics
 - Event-time & advanced window support

Apache Flink™



- Platform for scalable stream processing
- Meets requirements of CEP and stream analytics
 - Low latency and high throughput
 - Exactly-once semantics
 - Event-time & advanced windowing
- Core DataStream API available for Java & Scala

This Talk is About



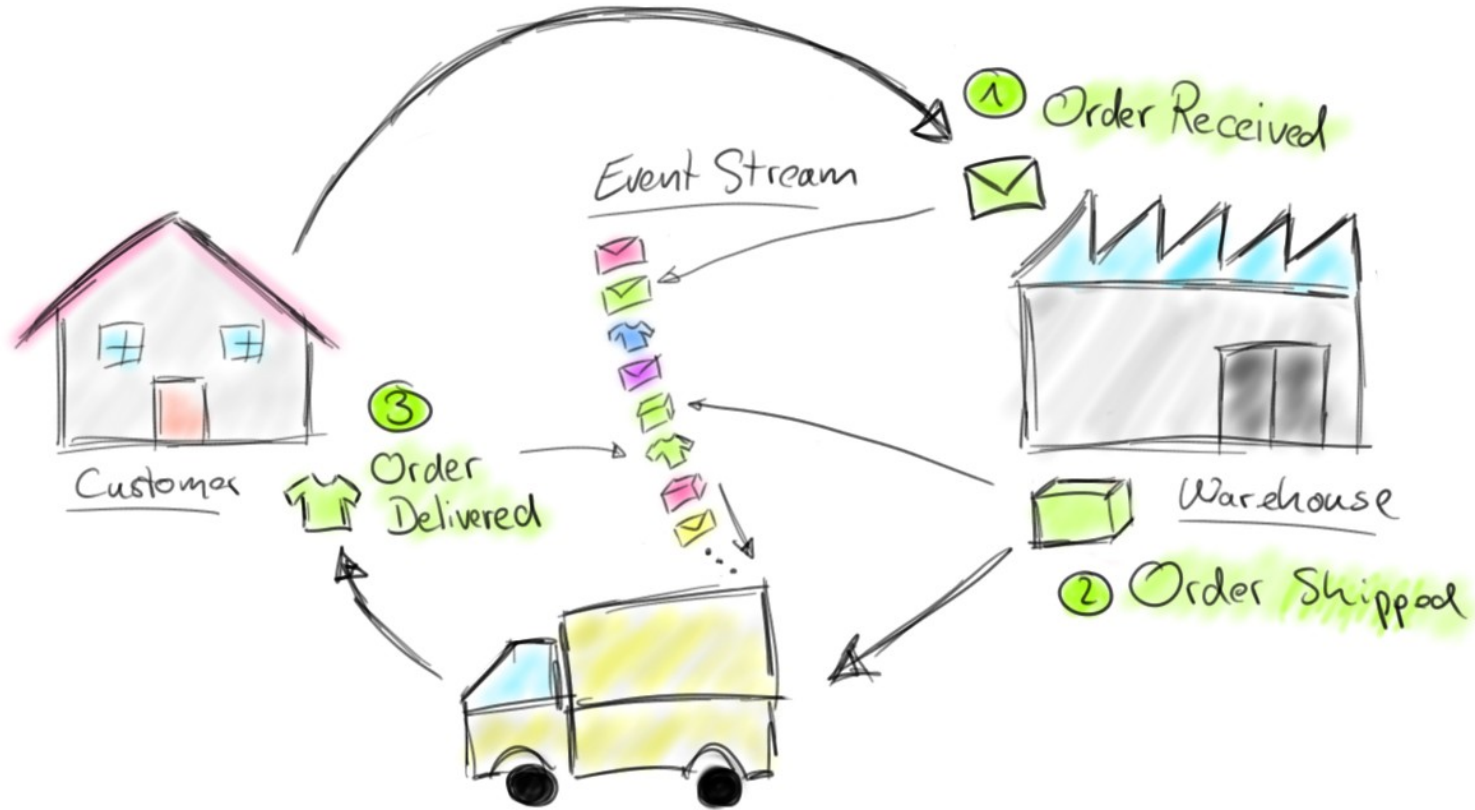
- Flink's new APIs for CEP and Stream Analytics
 - DSL to define CEP patterns and actions
 - Stream SQL to define queries on streams
- Integration of CEP and Stream SQL
- Early stage □ Work in progress



Use Case

Tracking an Order Process

Order Fulfillment Scenario



Order Events



- Process is reflected in a stream of order events
- `Order(orderId, timeStamp, "received")`
- `Shipment(orderId, timeStamp, "shipped")`
- `Delivery(orderId, timeStamp, "delivered")`

- `orderId`: Identifies the order
- `timeStamp`: Time at which the event happened



Stream Analytics

Aggregating Massive Streams

Stream Analytics



- Traditional batch analytics
 - Repeated queries on finite and changing data sets
 - Queries join and aggregate large data sets
- Stream analytics
 - “Standing” query produces continuous results from infinite input stream
 - Query computes aggregates on high-volume streams
- How to compute aggregates on infinite streams?

Compute Aggregates on Streams



- Split infinite stream into finite “windows”

- Split usually by time

Sender → 9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →

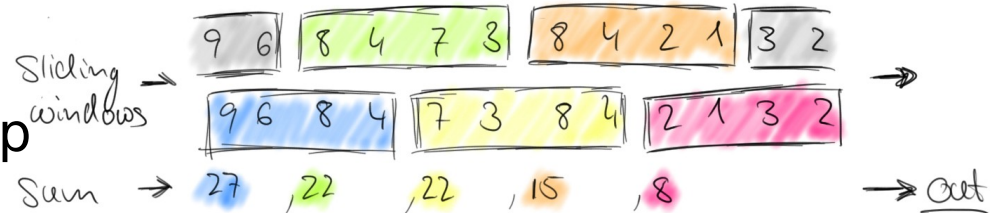
- Tumbling windows

- Fixed size & consecutive



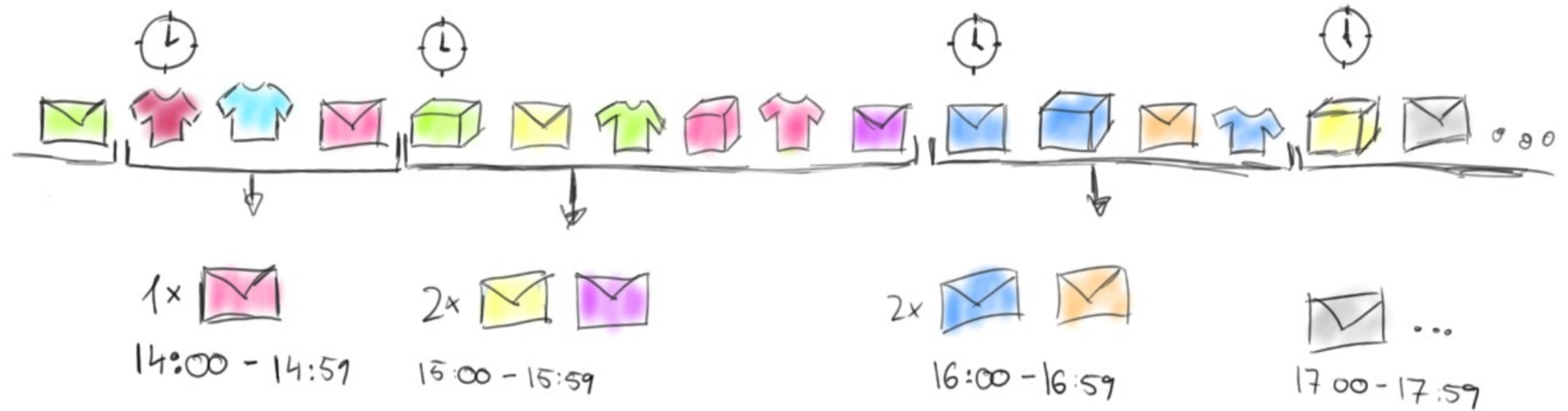
- Sliding windows

- Fixed size & may overlap



- Event time mandatory for correct & consistent results!

Example: Count Orders by Hour



Example: Count Orders by Hour

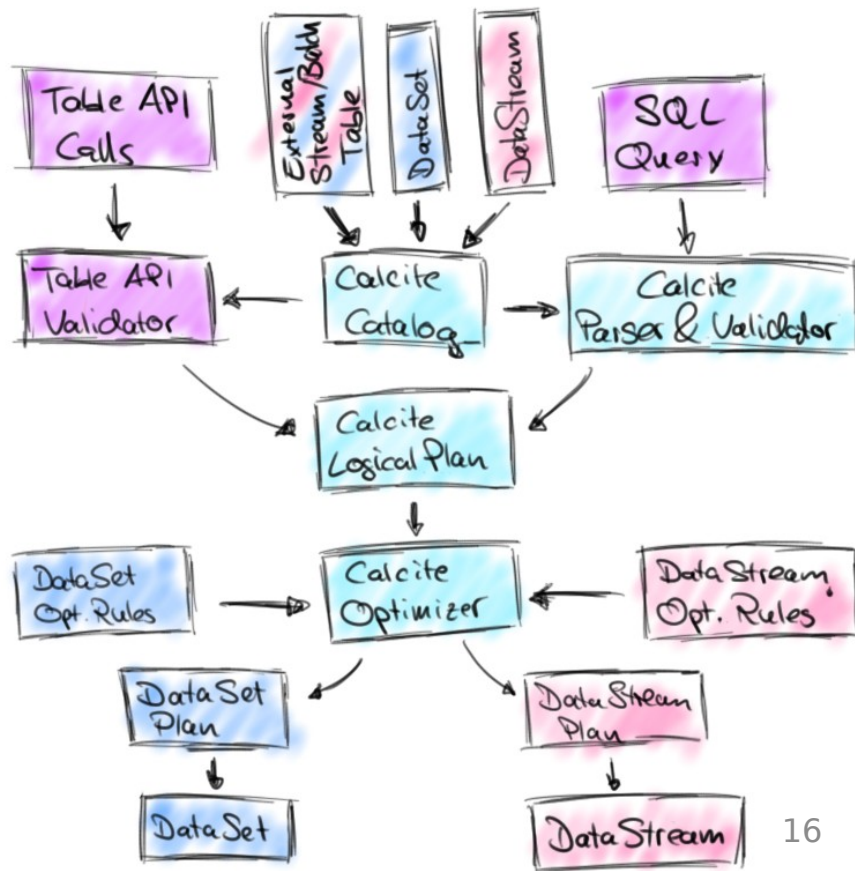


```
SELECT STREAM
  TUMBLE_START(tStamp, INTERVAL '1' HOUR) AS hour,
  COUNT(*) AS cnt
FROM events
WHERE
  status = 'received'
GROUP BY
  TUMBLE(tStamp, INTERVAL '1' HOUR)
```

Stream SQL Architecture



- Flink features SQL on static and streaming tables
- Parsing and optimization by Apache Calcite
- SQL queries are translated into native Flink programs

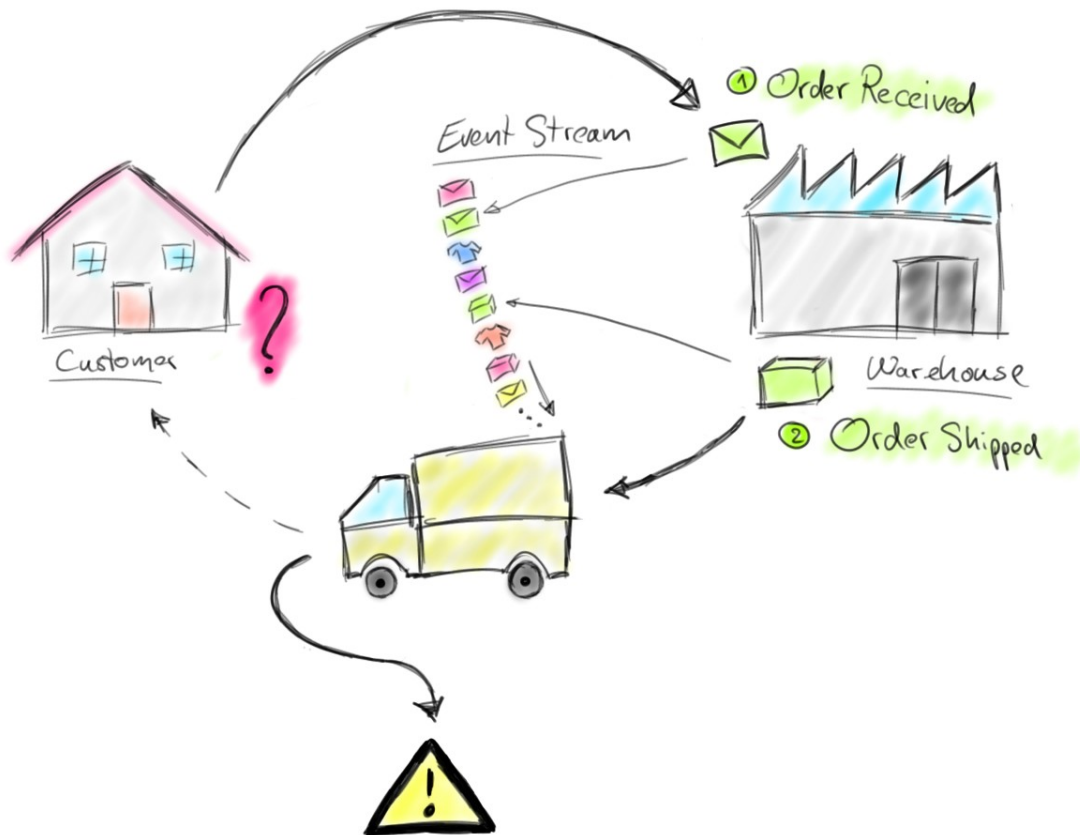




Complex Event Processing

Pattern Matching on Streams

Real-time Warnings



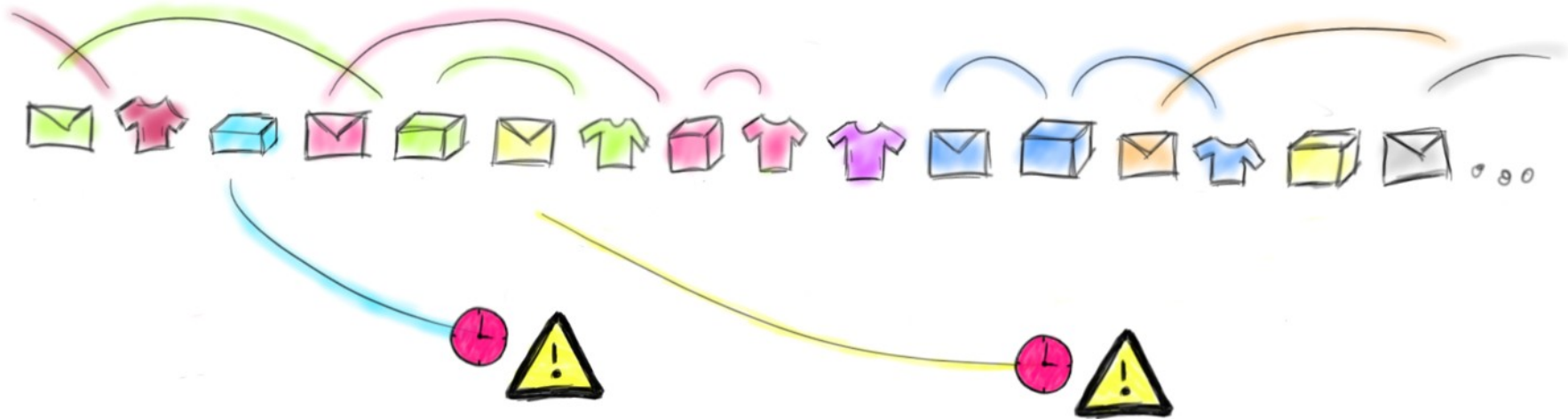
CEP to the Rescue



- Define processing and delivery intervals (SLAs)
- `ProcessSucc(orderId, timeStamp, duration)`
- `ProcessWam(orderId, timeStamp)`
- `DeliverySucc(orderId, timeStamp, duration)`
- `DeliveryWam(orderId, timeStamp)`

- `orderId`: Identifies the order
- `timeStamp`: Time when the event happened
- `duration`: Duration of the processing/delivery

CEP Example



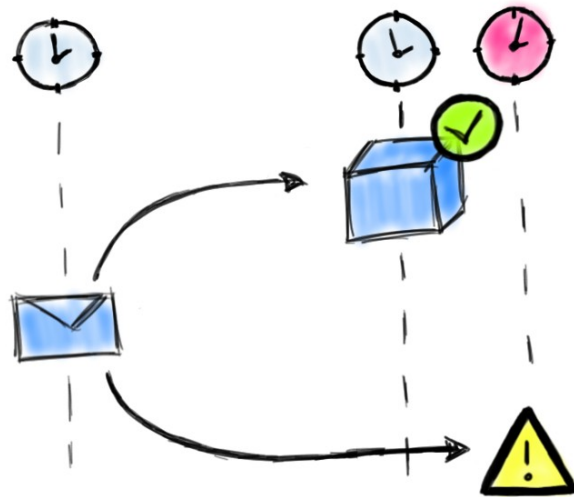
Processing: Order → Shipment



```
val processingPattern = Pattern
    .begin[Event]("received").subtype(classOf[Order])
    .followedBy("shipped").where(_.status == "shipped")
    .within(Time.hours(1))

val processingPatternStream = CEP.pattern(
    input.keyBy("orderId"),
    processingPattern)

val procResult: DataStream[Either[ProcessWarn, ProcessSucc]] =
    processingPatternStream.select {
        (pP, timestamp) => // Timeout handler
            ProcessWarn(pP("received").orderId, timestamp)
    } {
        fP => // Select function
            ProcessSucc(
                fP("received").orderId, fP("shipped").tStamp,
                fP("shipped").tStamp - fP("received").tStamp)
    }
```

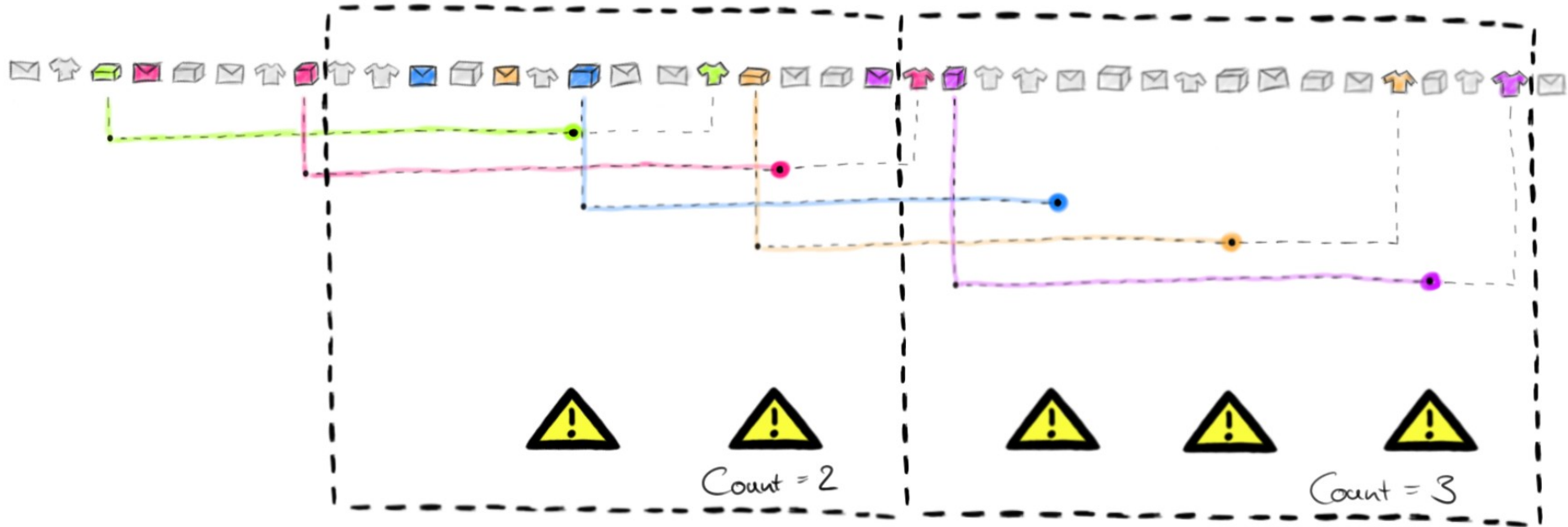




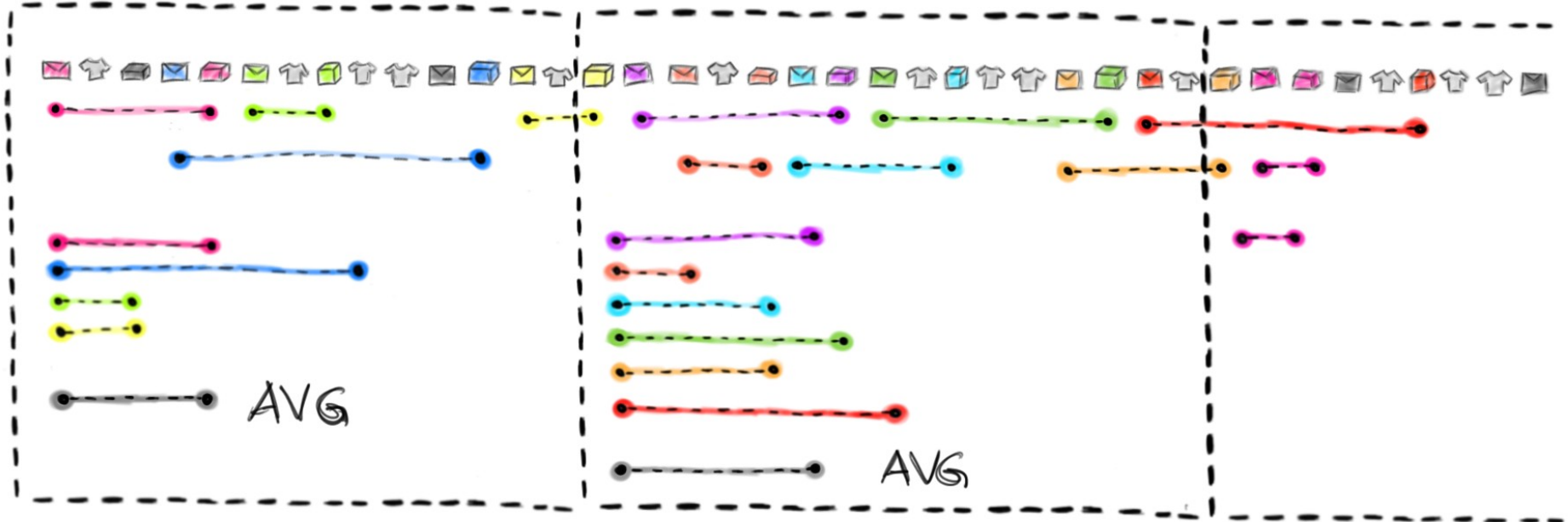
Integrated Stream Analytics with CEP

... and both at the same time!

Count Delayed Shipments



Compute Avg Processing Time



CEP + Stream SQL



```
// complex event processing result
val delResult: DataStream[Either[DeliveryWarn, DeliverySucc]] = ...

val delWarn: DataStream[DeliveryWarn] = delResult.flatMap(_.left.toOption)

val deliveryWarningTable: Table = delWarn.toTable(tableEnv)
tableEnv.registerTable("deliveryWarnings", deliveryWarningTable)

// calculate the delayed deliveries per day
val delayedDeliveriesPerDay = tableEnv.sql(
  """SELECT STREAM
    | TUMBLE_START(tStamp, INTERVAL `1` DAY) AS day,
    | COUNT(*) AS cnt
    |FROM deliveryWarnings
    |GROUP BY TUMBLE(tStamp, INTERVAL `1` DAY)""".stripMargin)
```

CEP-enriched Stream SQL



```
SELECT
  TUMBLE_START(tStamp, INTERVAL '1' DAY) as day,
  AVG(duration) as avgDuration
FROM (

  // CEP pattern
  SELECT (b.tStamp - a.tStamp) as duration, b.tStamp as tStamp
  FROM inputs
  PATTERN
    a FOLLOW BY b PARTITION BY orderId ORDER BY tStamp
  WITHIN INTERVAL '1' HOUR
  WHERE
    a.status = 'received' AND b.status = 'shipped'
)
GROUP BY
  TUMBLE(tStamp, INTERVAL '1' DAY)
```

Conclusion



- Apache Flink handles CEP and analytical workloads
- Apache Flink offers intuitive APIs
- New class of applications by CEP and Stream SQL integration □

12-14 SEP 2016

BERLIN

CALL FOR SUBMISSIONS

PALAIS >

Flink Forward 2016, Berlin
Submission deadline: June 30, 2016
Early bird deadline: July 15, 2016
www.flink-forward.org



dataArtisans

We are hiring!
data-artisans.com/careers