Apache Flink™ Training

FlinkCEP and Table API / SQL

Tzu-Li (Gordon) Tai

tzulitai@apache.org

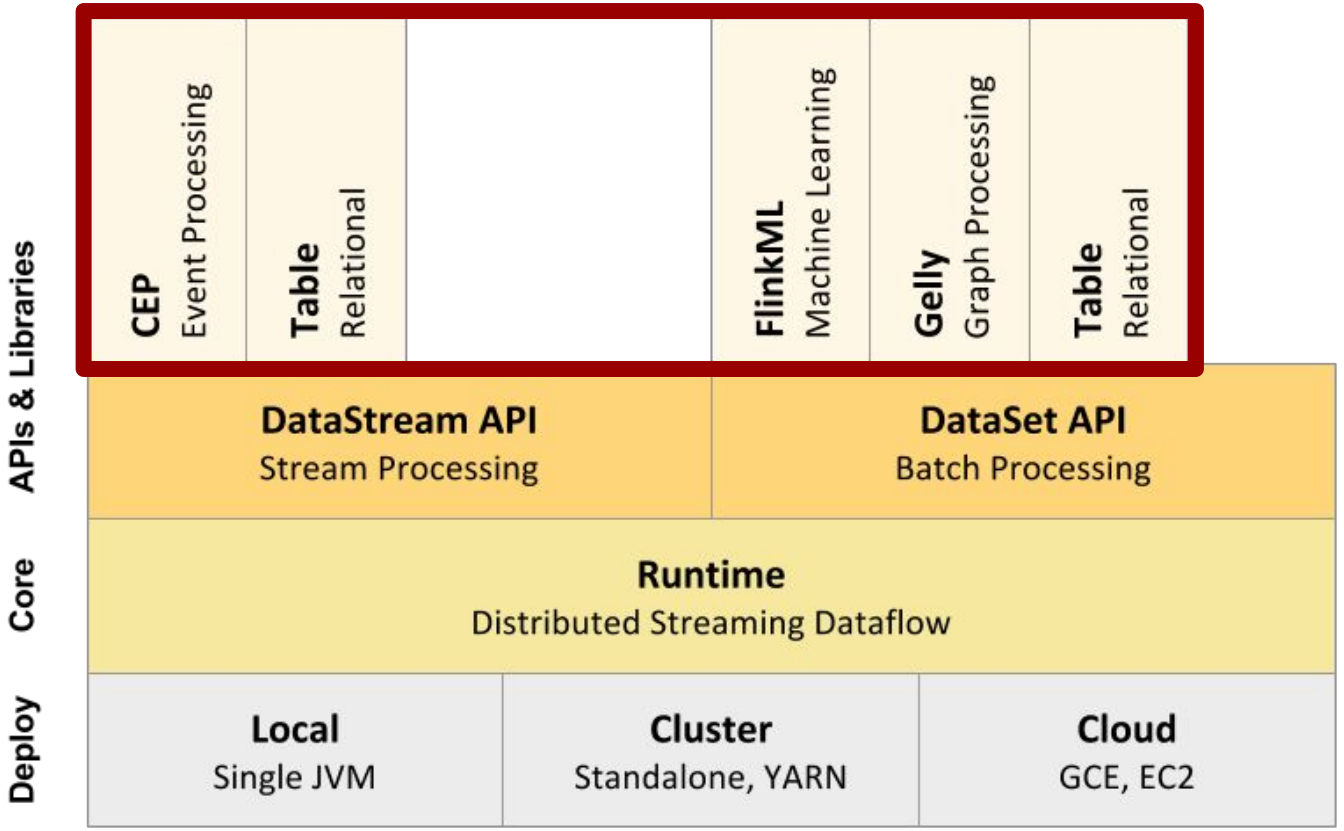
@tzulitai

Flink.tw

Apache Flink Taiwan User Group

Sept 2016 @ HadoopCon

00 This session will be about ...



00 This session will be about ...

- The Flink libraries, built on DataSet / DataStream API

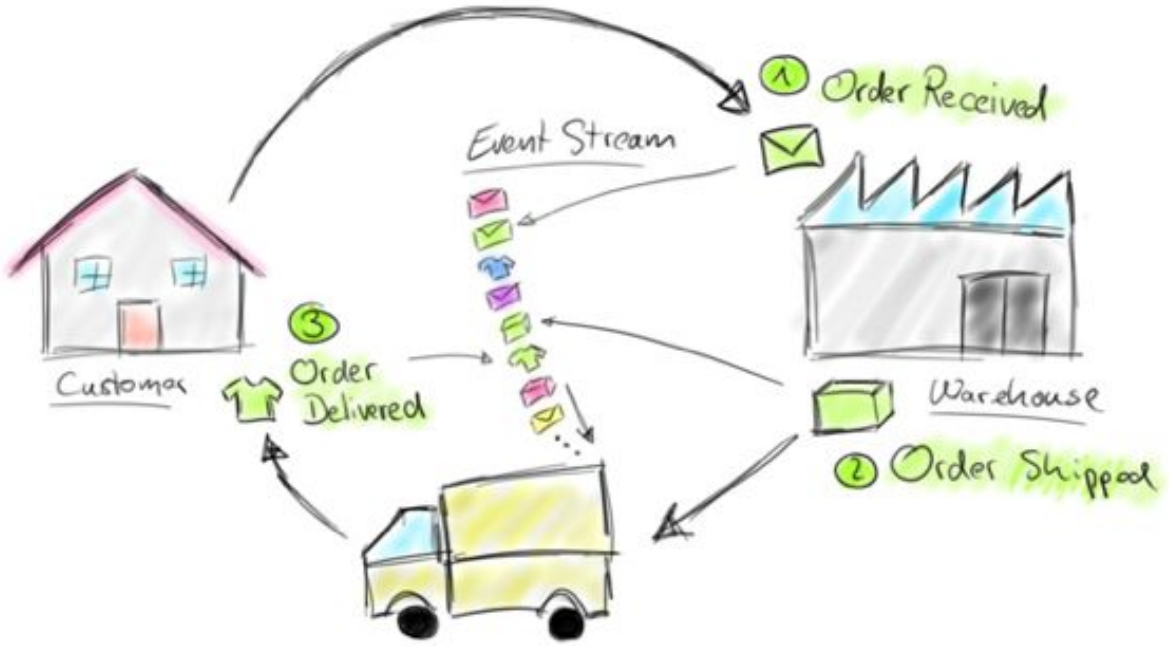
- FlinkCEP *
- Table API *
- **x FlinkML**
- **x Gelly**

* The examples used are from http://www.slideshare.net/tillrohrmann/streaming-analytics-cep-two-sides-of-the-same-coin

01 FlinkCEP

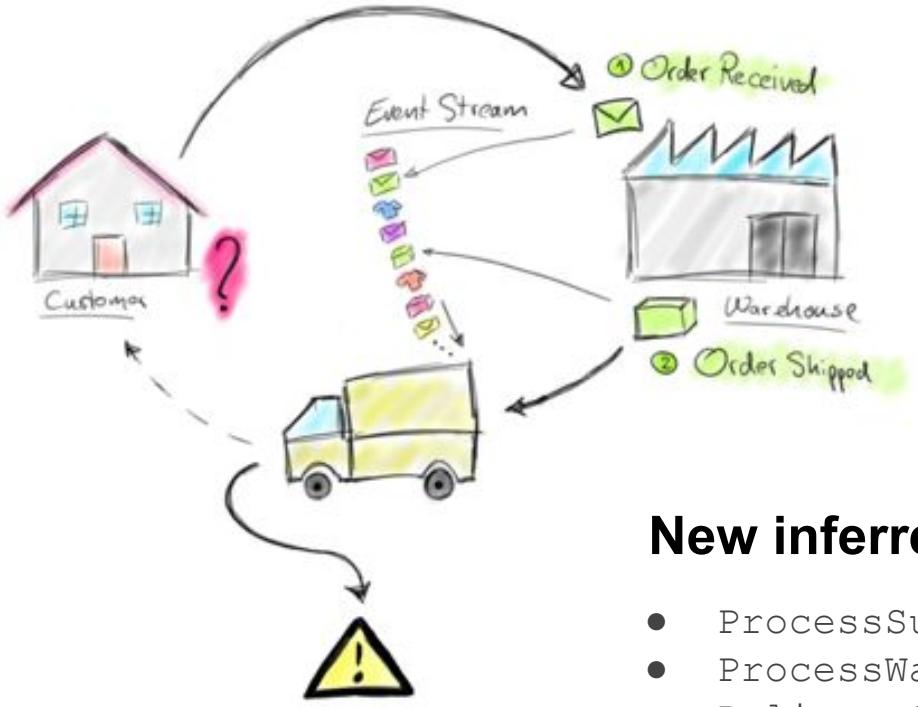
- CEP: Complex Event Processing
- Generate *derived events* when a specified pattern on raw events occur in a data stream
  - if `A` and then `B` → infer complex event `C`
- Goal: identify meaningful event patterns and respond to them as quickly as possible

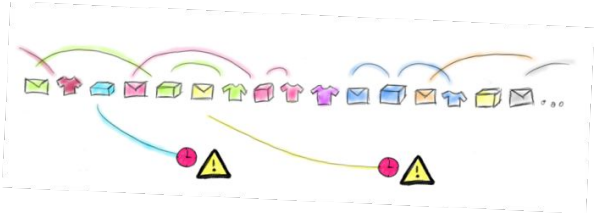
01 FlinkCEP



- `Order(orderId, tStamp, "received") extends Event`
- `Shipment(orderId, tStamp, "shipped") extends Event`
- `Delivery(orderId, tStamp, "delivered") extends Event`

01 FlinkCEP

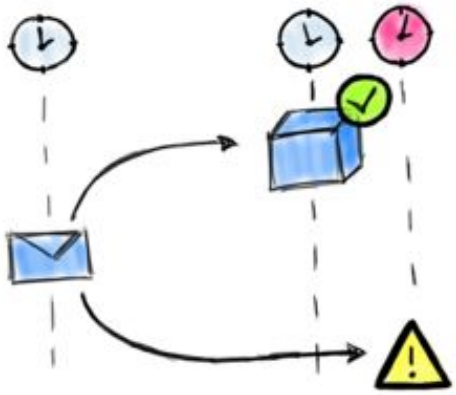




New inferred events:

- `ProcessSucc(orderId, tStamp, duration)`
- `ProcessWarn(orderId, tStamp)`
- `DeliverySucc(orderId, tStamp, duration)`
- `DeliveryWarn(orderId, tStamp)`

01 FlinkCEP



```
val processingPattern = Pattern
  .begin[Event]("orderReceived").subtype(classOf[Order])
  .followedBy("orderShipped").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("orderReceived").orderId, timestamp)
  } {
    fP => // Select function
      ProcessSucc(
        fP("orderReceived").orderId, fP("orderShipped").tStamp,
        fP("orderShipped").tStamp - fP("orderReceived").tStamp)
  }
```

01 FlinkCEP

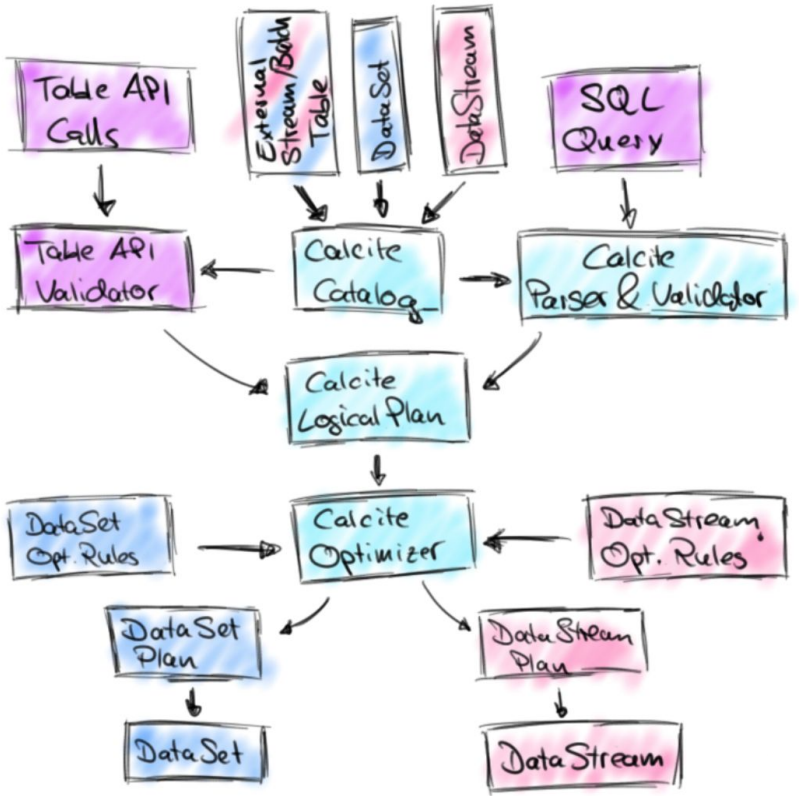
```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment

val input: DataStream[Event] = env.addSource(new FlinkKafkaConsumer09(...))

val processingPattern  = Pattern.begin(...)...

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

val procResult : DataStream[Either[ProcessWarn, ProcessSucc]] = processingPatternStream.select(...)

procResult.addSink(new RedisSink(...))
          // .addSink(new FlinkKafkaProducer09(...))
          // .addSink(new ElasticsearchSink(...))
          // .map(new MapFunction{...})
          // … anything you'd like to continue to do with the inferred event stream

env.execute()
```

01 FlinkCEP

```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic( TimeCharacteristic.EventTime)

val input: DataStream[Event] = env
     .addSource(new FlinkKafkaConsumer09(...))
     .assignTimestampsAndWatermarks(new CustomExtractor)

val processingPattern = Pattern.begin(...)...

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

val procResult : DataStream[Either[ProcessWarn, ProcessSucc]] = processingPatternStream.select(...)

procResult.addSink(new RedisSink(...))
          // .addSink(new FlinkKafkaProducer09(...))
          // .addSink(new ElasticsearchSink(...))
          // .map(new MapFunction{...})

env.execute()
```

02 Table API / SQL

- Relational processing on DataStreams / DataSets / External Sources
- Parsing and optimization by Apache Calcite
- SQL queries are translated into native Flink programs



02 Table API / SQL

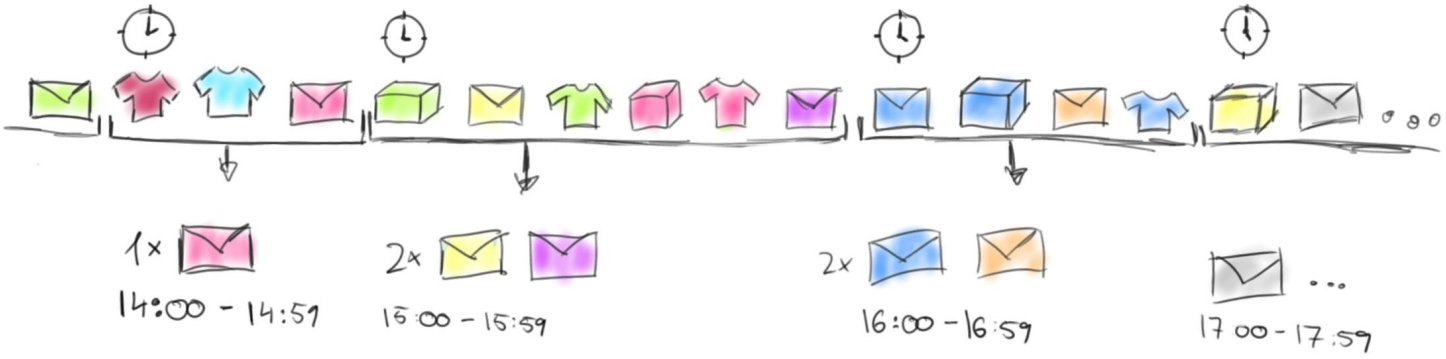
Batch Table

```
val env = ExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// register the DataSet cust as table "Customers" with fields derived from the dataset
tableEnv.registerDataSet("Customers", cust)

// register the DataSet ord as table "Orders" with fields user, product, and amount
tableEnv.registerDataSet("Orders", ord, 'user, 'product, 'amount)
```

Streaming Table

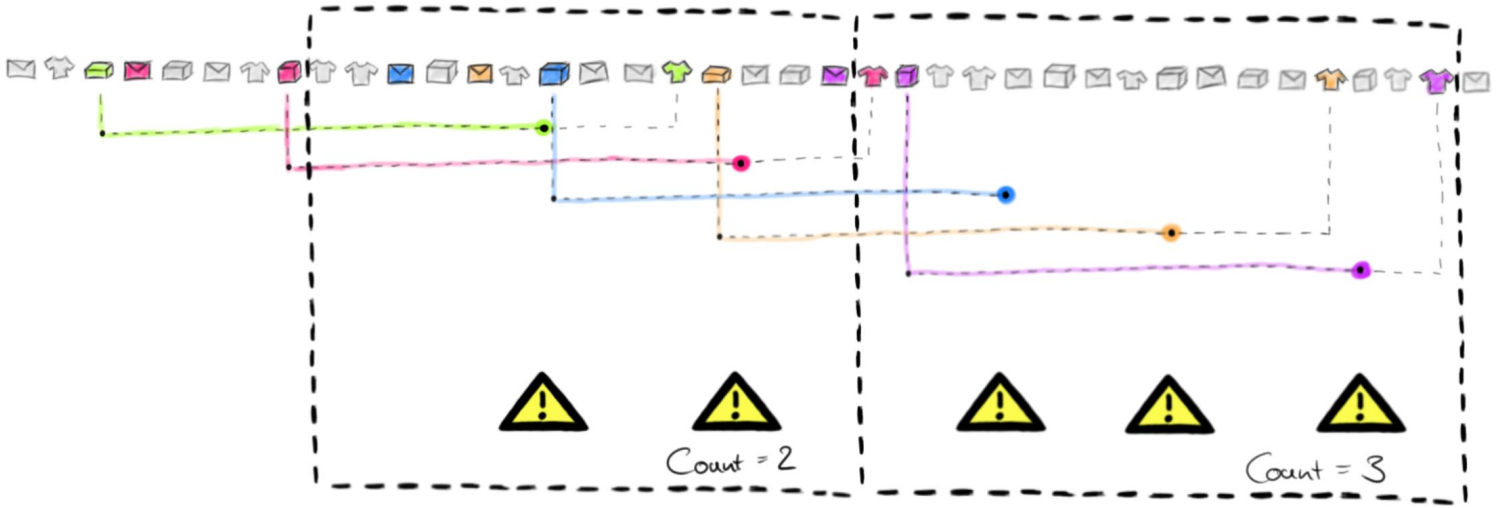
```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// register the DataStream cust as table "Customers" with fields derived from the datastream
tableEnv.registerDataStream("Customers", cust)

// register the DataStream ord as table "Orders" with fields user, product, and amount
tableEnv.registerDataStream("Orders", ord, 'user, 'product, 'amount)
```

02 Table API / SQL




```
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)
```

03 CEP and SQL combined



- Streaming analytics with CEP and SQL at the same time!
- Count the number of delayed deliveries per hour

03 CEP and SQL combined

```
// 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)
```