Flink Streaming

Marton Balassi
Flink committer
data Artisans
@MartonBalassi

Stream processing

Stream

Infinite sequence of data
arriving in a continuous fashion.

An example streaming use case

Recommender system

- Based on historic item ratings
- And on the activity of the user
- Provide recommendations
- To tens of millions of users
- From millions of items
- With a 100 msec latency guarantee

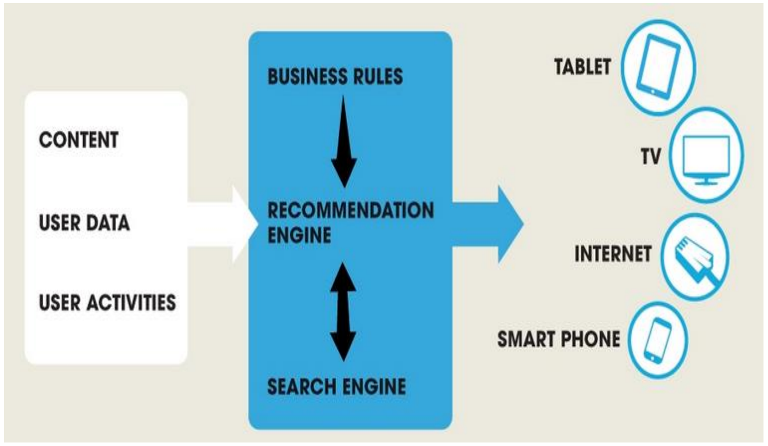


Figure courtesy of Gravity R&D, used with permission.

Many buzzwords, similar concepts

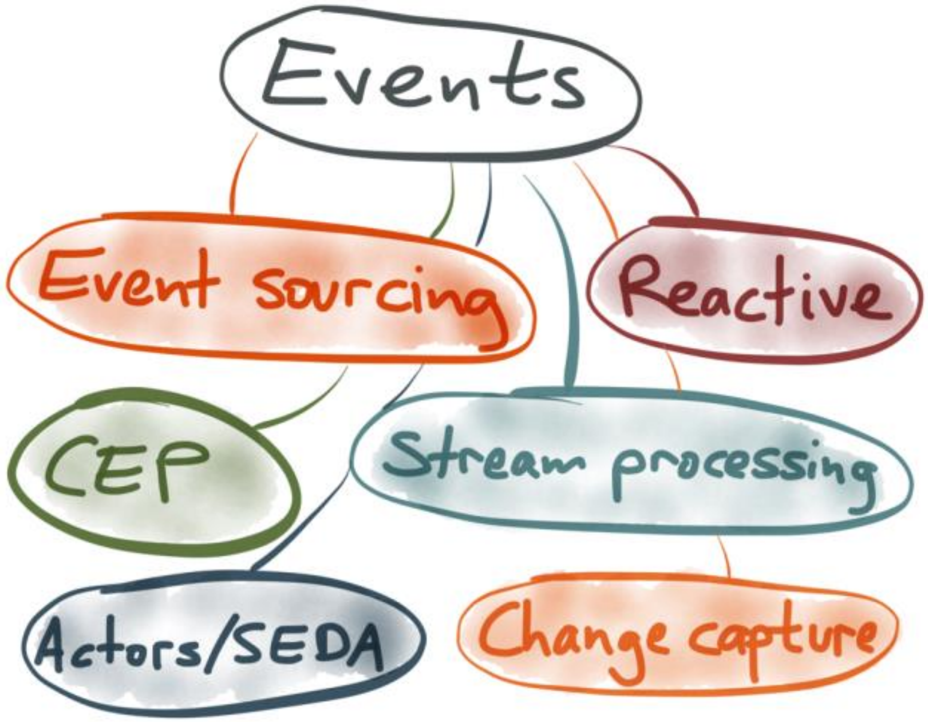


Figure courtesy of Martin Kleppmann, used with permission.

Streaming systems

Apache Storm

- True streaming, low latency - lower throughput
- Low level API (Bolts, Spouts) + Trident

Spark Streaming

- Stream processing on top of batch system, high throughput - higher latency
- Functional API (DStreams), restricted by batch runtime

Apache Samza

- True streaming built on top of Apache Kafka, state is first class citizen
- Slightly different stream notion, low level API

Flink Streaming

- True streaming with adjustable latency-throughput trade-off
- Rich functional API exploiting streaming runtime; e.g. rich windowing semantics

Streaming systems

Apache Storm

- True streaming, low latency - lower throughput
- Low level API (Bolts, Spouts) + Trident

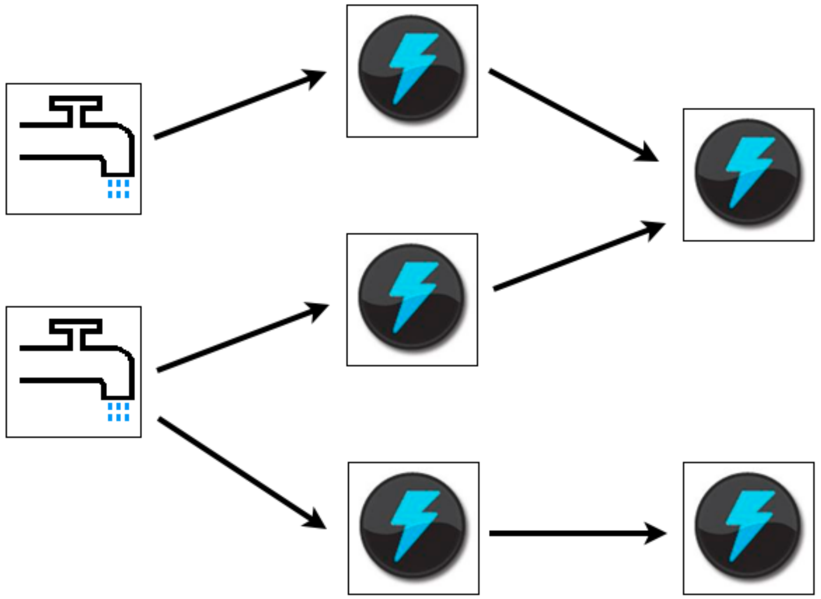


Figure courtesy of Apache Storm, source: http://storm.apache.org/images/topology.png

Streaming systems

Spark Streaming

- Stream processing on top of batch system, high throughput - higher latency
- Functional API (DStreams), restricted by batch runtime

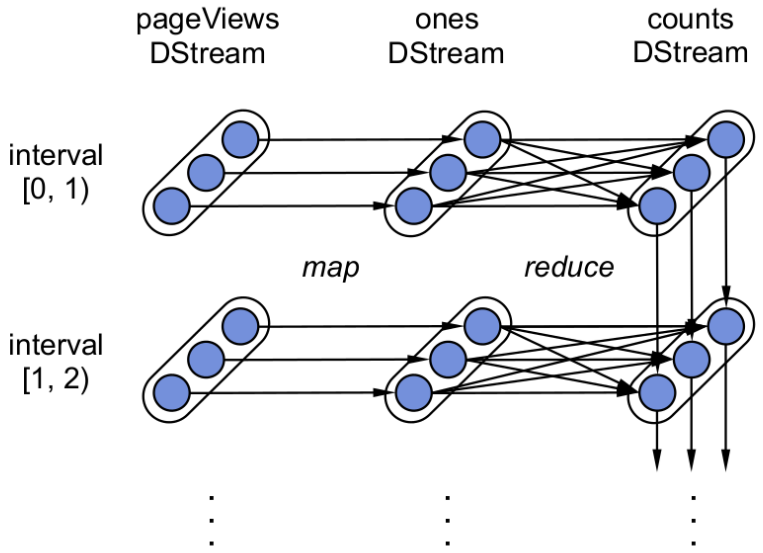


Figure courtesy of Matei Zaharia,
source: http://cs.berkeley.edu/~matei/papers/2013/sosp_spark_streaming.pdf, page 6

Streaming systems

Apache Samza

- True streaming built on top of Apache Kafka, state is first class citizen
- Slightly different stream notion, low level API

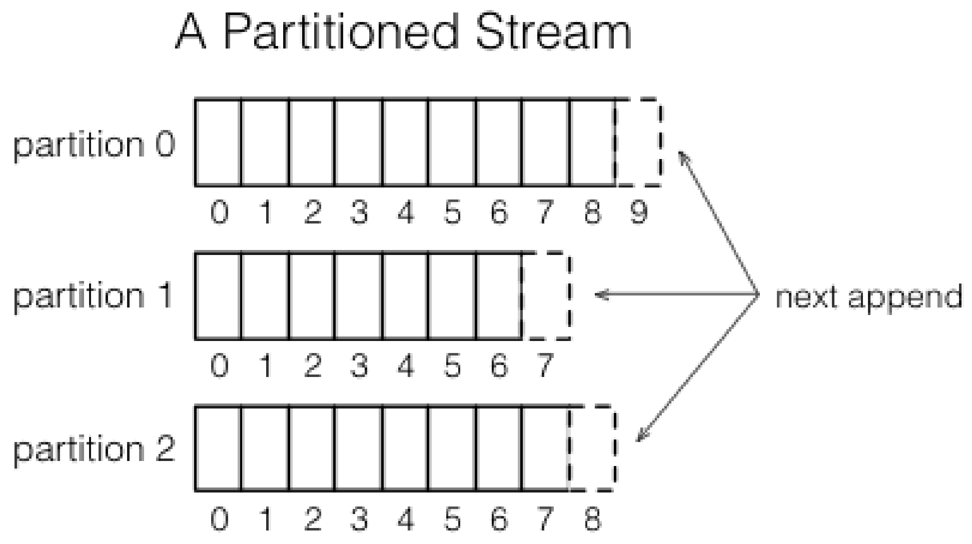


Figure courtesy of Apache Samza,
source: http://samza.apache.org/img/0.8/learn/documentation/introduction/stream.png

Streaming systems

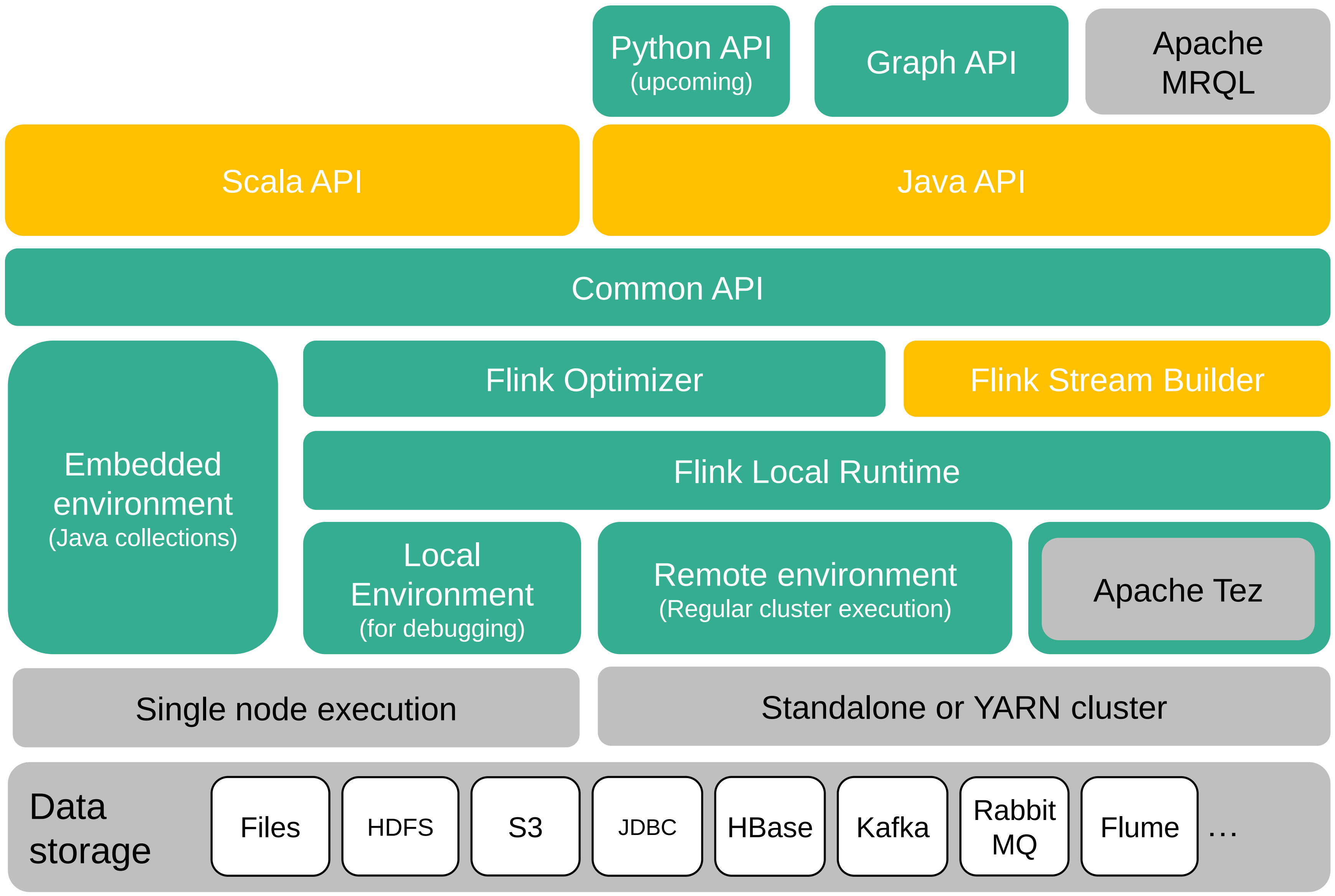
Apache Storm
- True streaming, low latency - lower throughput
- Low level API (Bolts, Spouts) + Trident

Spark Streaming
- Stream processing on top of batch system, high throughput - higher latency
- Functional API (DStreams), restricted by batch runtime

Apache Samza
- True streaming built on top of Apache Kafka, state is first class citizen
- Slightly different stream notion, low level API

Flink Streaming
- True streaming with adjustable latency-throughput trade-off
- Rich functional API exploiting streaming runtime; e.g. rich windowing semantics

Streaming in Flink

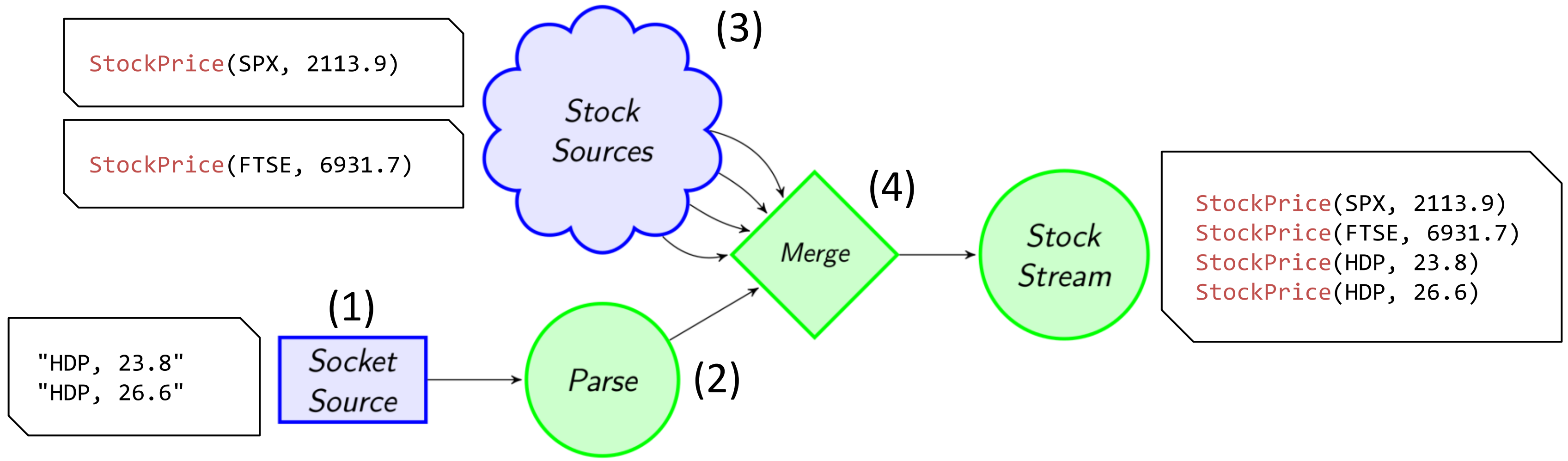


Using Flink Streaming

Example: StockPrices

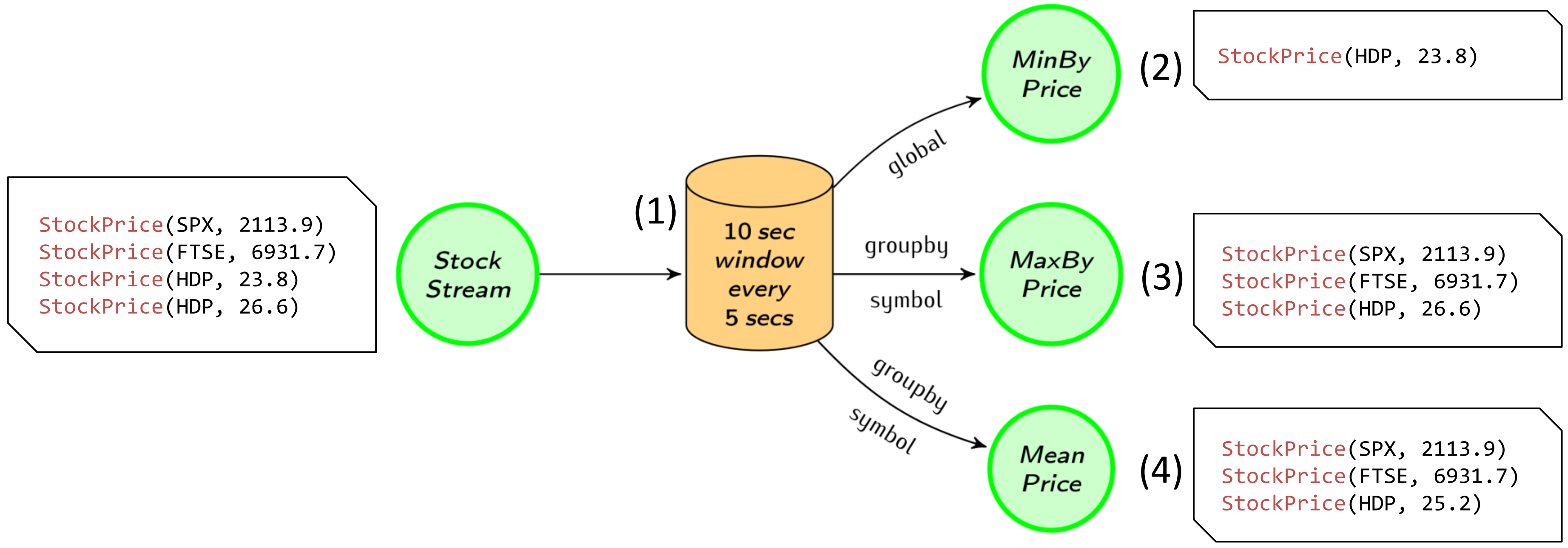
- Reading from multiple inputs
  - Merge stock data from various sources
- Window aggregations
  - Compute simple statistics over windows of data
- Data driven windows
  - Define arbitrary windowing semantics
- Combining with a Twitter stream
  - Enrich your analytics with social media feeds
- Streaming joins
  - Join multiple data streams
- Detailed explanation and source code on our blog
  - http://flink.apache.org/news/2015/02/09/streaming-example.html

Example: Reading from multiple inputs



```
    case class StockPrice(symbol : String, price : Double)
    val env = StreamExecutionEnvironment.getExecutionEnvironment

(1) val socketStockStream = env.socketTextStream("localhost", 9999)
(2)     .map(x => { val split = x.split(",")
            StockPrice(split(0), split(1).toDouble) })

(3) val SPX_Stream = env.addSource(generateStock("SPX")(10) _)
    val FTSE_Stream = env.addSource(generateStock("FTSE")(20) _)
(4) val stockStream = socketStockStream.merge(SPX_Stream, FTSE_STREAM)
```

Example: Window aggregations



```
StockPrice(SPX, 2113.9)
StockPrice(FTSE, 6931.7)
StockPrice(HDP, 23.8)
StockPrice(HDP, 26.6)
```

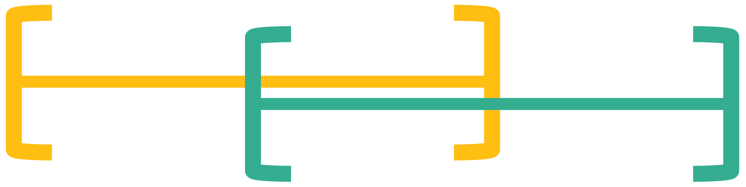
(2)
```
StockPrice(HDP, 23.8)
```

(3)
```
StockPrice(SPX, 2113.9)
StockPrice(FTSE, 6931.7)
StockPrice(HDP, 26.6)
```

(4)
```
StockPrice(SPX, 2113.9)
StockPrice(FTSE, 6931.7)
StockPrice(HDP, 25.2)
```

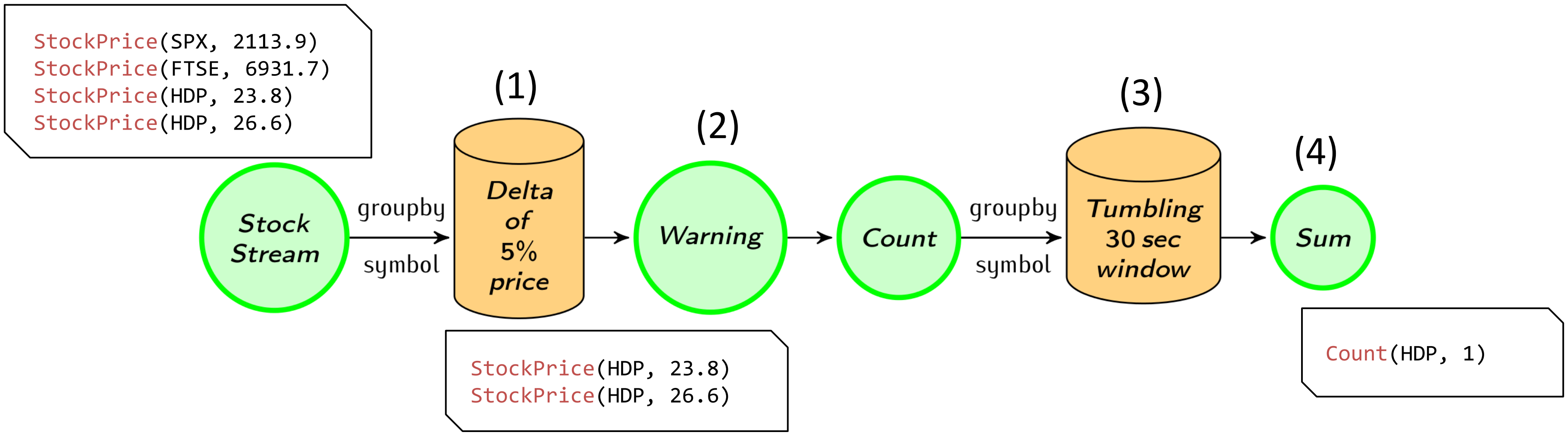
```
    val windowedStream = stockStream
(1)   .window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))

(2) val lowest = windowedStream.minBy("price")
(3) val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4) val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```

Windowing



- Trigger policy
  - When to trigger the computation on current window
- Eviction policy
  - When data points should leave the window
  - Defines window width/size
- E.g., count-based policy
  - evict when #elements > n
  - start a new window every n-th element
- Built-in: Count, Time, Delta policies

Example: Data-driven windows



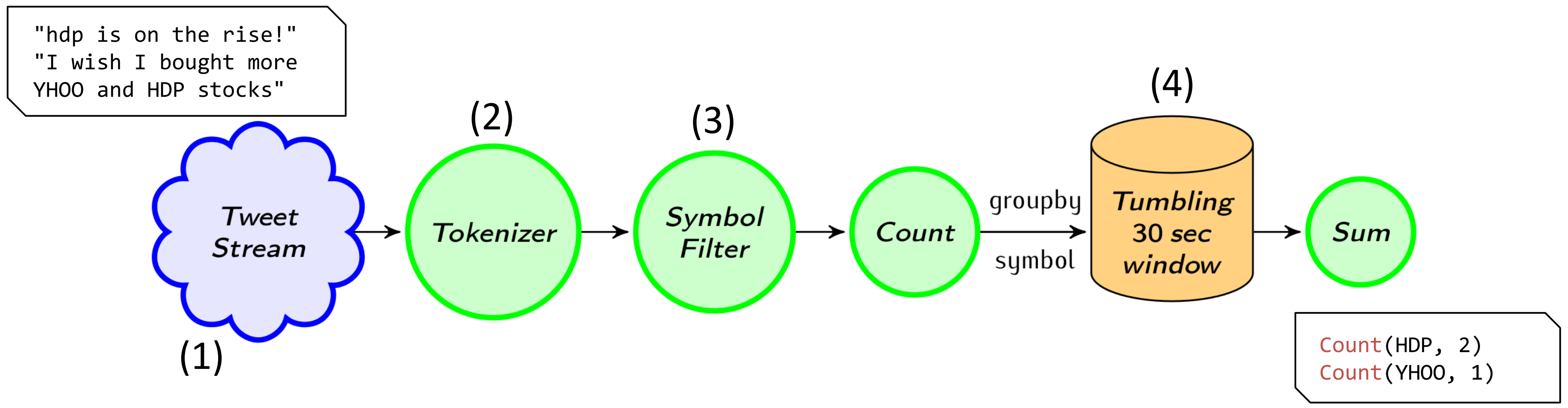
```
StockPrice(SPX, 2113.9)
StockPrice(FTSE, 6931.7)
StockPrice(HDP, 23.8)
StockPrice(HDP, 26.6)
```

```
StockPrice(HDP, 23.8)
StockPrice(HDP, 26.6)
```

```
Count(HDP, 1)
```

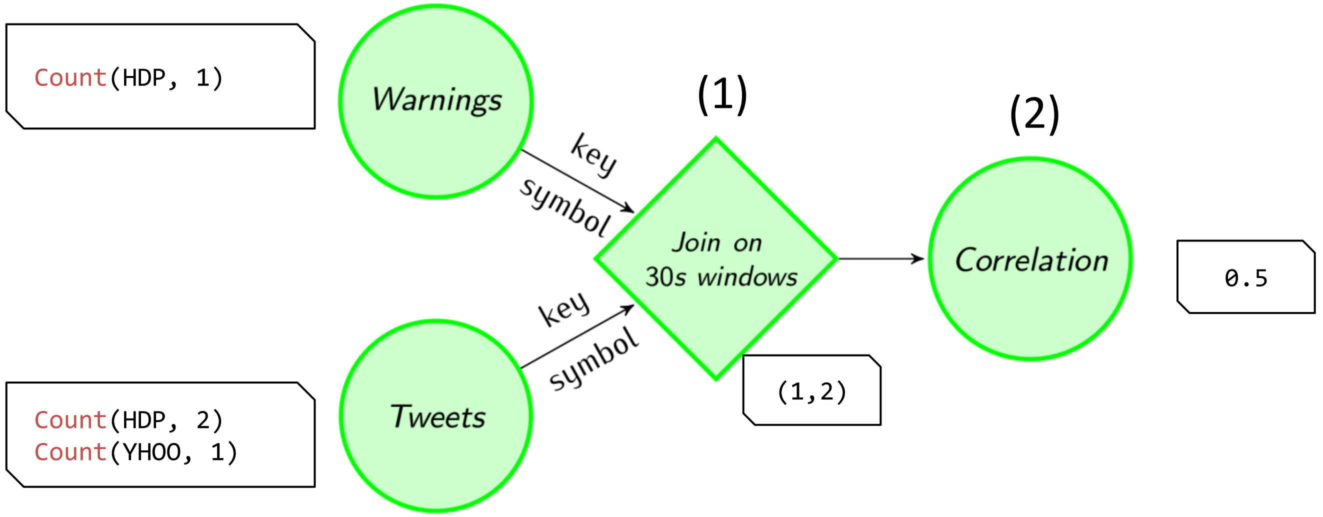
```
    case class Count(symbol : String, count : Int)

    val priceWarnings = stockStream.groupBy("symbol")
(1)   .window(Delta.of(0.05, priceChange, defaultPrice))
(2)   .mapWindow(sendWarning _)

    val warningsPerStock = priceWarnings.map(Count(_, 1))
      .groupBy("symbol")
(3)   .window(Time.of(30, SECONDS))
(4)   .sum("count")
```

Example: Combining with a Twitter stream



```
(1)  val tweetStream = env.addSource(generateTweets _)

(2)  val mentionedSymbols = tweetStream.flatMap(tweet => tweet.split(" "))
       .map(_.toUpperCase())
(3)    .filter(symbols.contains(_))

     val tweetsPerStock = mentionedSymbols.map(Count(_, 1))
       .groupBy("symbol")
(4)    .window(Time.of(30, SECONDS))
       .sum("count")
```

Example: Streaming joins



(1)

```
val tweetsAndWarning = warningsPerStock.join(tweetsPerStock)
    .onWindow(30, SECONDS)
    .where("symbol")
    .equalTo("symbol"){ (c1, c2) => (c1.count, c2.count) }
```

(2)

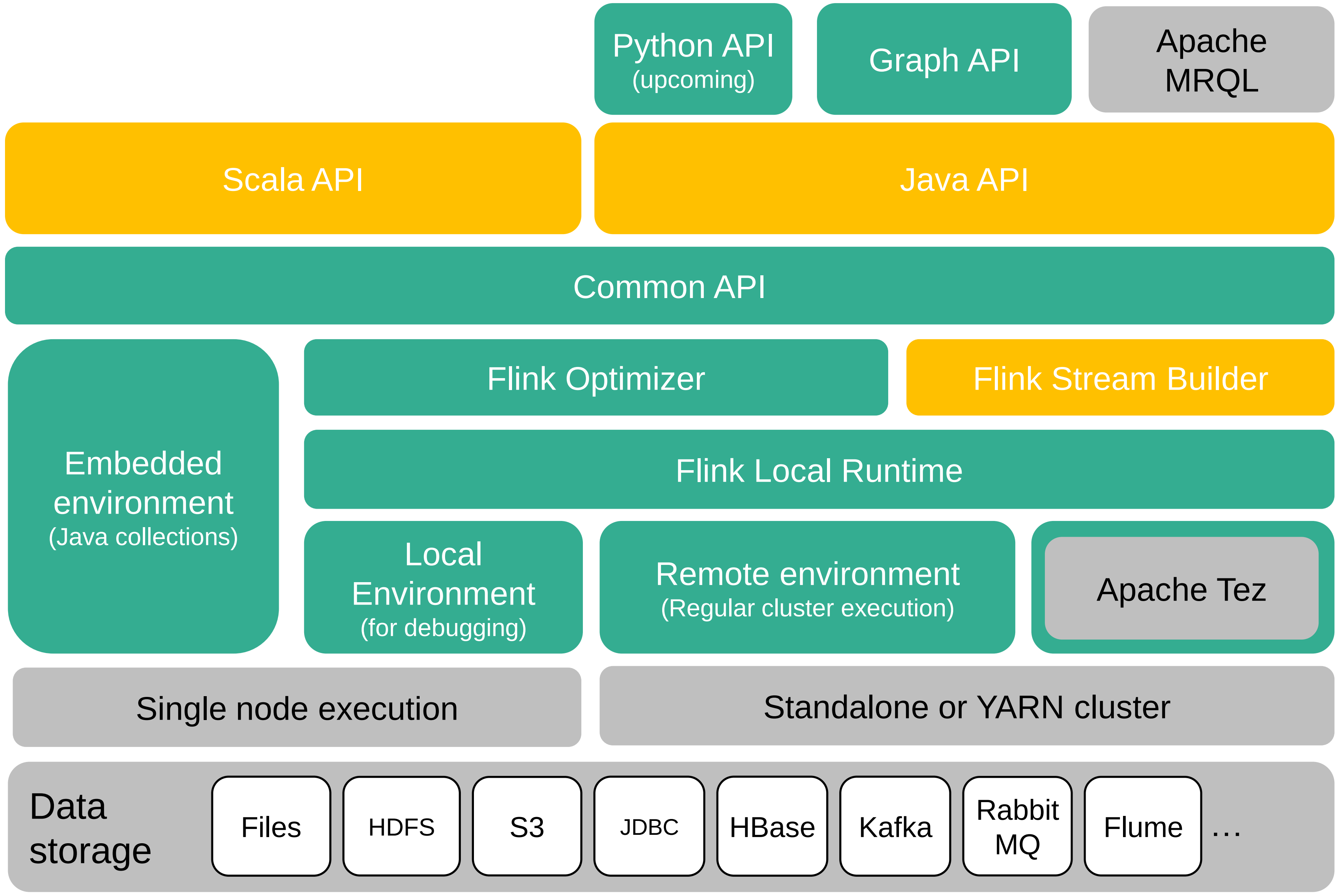
```
val rollingCorrelation = tweetsAndWarning
    .window(Time.of(30, SECONDS))
    .mapWindow(computeCorrelation _)
```

Overview of the API

- Data stream sources
  - File system
  - Message queue connectors
  - Arbitrary source functionality
- Stream transformations
  - Basic transformations: *Map, Reduce, Filter, Aggregations*…
  - Binary stream transformations: *CoMap, CoReduce*…
  - Windowing semantics: *Policy based flexible windowing (Time, Count, Delta…)*
  - Temporal binary stream operators: *Joins, Crosses*…
  - Iterative stream transformations
- Data stream outputs
- For the details please refer to the programming guide:
  - http://flink.apache.org/docs/latest/streaming_guide.html

Internals

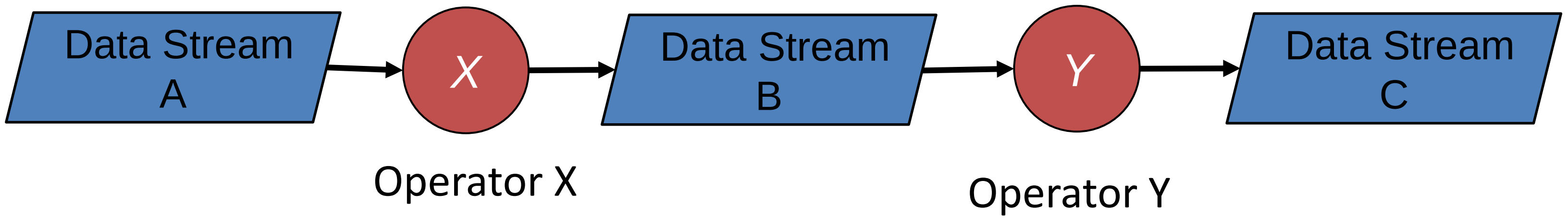
Streaming in Flink



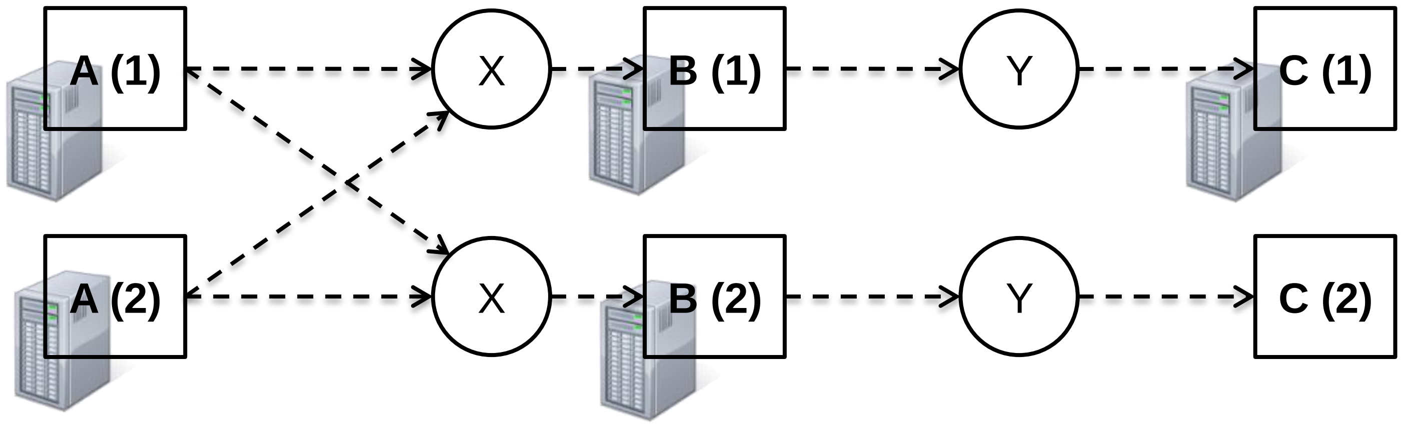
Programming model

Data abstraction: **Data Stream**

Program



Parallel Execution



Fault tolerance

- At-least-once semantics
  - All the records are processed, but maybe multiple times
  - Source level in-memory replication
  - Record acknowledgments
  - In case of failure the records are replayed from the sources
  - Storm supports this approach
  - Currently in alpha version

Fault tolerance

- Exactly once semantics
  - User state is a first class citizen
  - Checkpoint triggers emitted from sources in line with the data
  - When an operator sees a checkpoint it asyncronously checkpoints its state
  - Upstream recovery from last checkpoint
  - Spark and Samza supports this approach
  - Final goal, current challenge

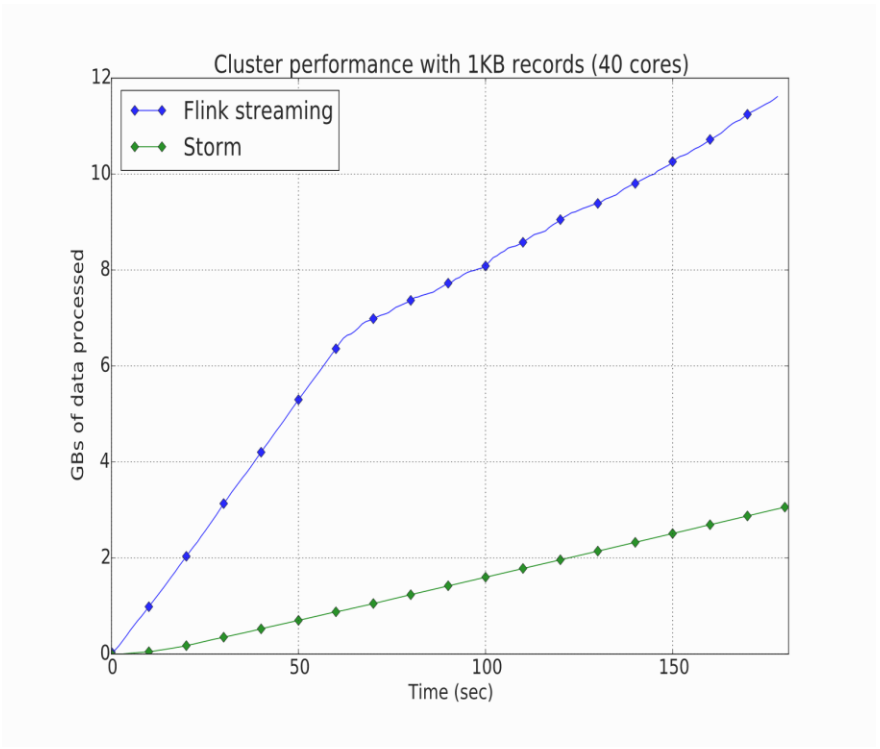
Roadmap

- Fault tolerance – 2015 Q1-2
- Lambda architecture – 2015 Q2
- Runtime Optimisations - 2015 Q2
- Full Scala interoperability – 2015 Q2
- Integration with other frameworks
  - SAMOA – 2015 Q1
  - Zeppelin – 2015 ?
- Machine learning Pipelines library – 2015 Q3
- Streaming graph processing library – 2015 Q3

Performance

Flink Streaming performance

- Current measurements are outdated
- Last measurements showed twice the throughput of Storm
- In a recent specific telecom use case throughput was higher than Spark Streaming’s
- New blogpost on performance measures is coming soon!



Closing

Summary

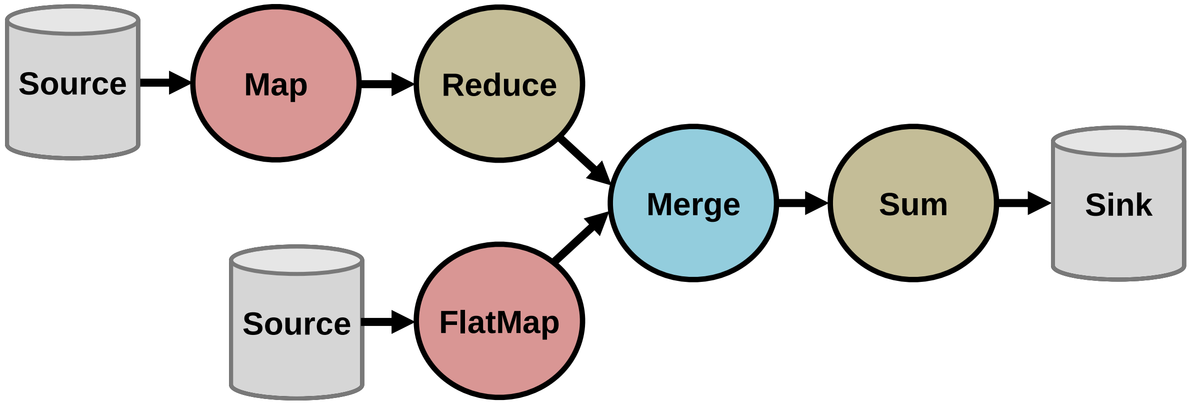
- Flink combines true streaming runtime with expressive high-level APIs for a next-gen stream processing solution
- Flexible windowing semantics
- Iterative processing support opens new horizons in online machine learning
- Competitive performance
- We are just getting started!

flink.apache.org
@ApacheFlink

Appendix

Basic transformations

- Rich set of functional transformations:
  - Map, FlatMap, Reduce, GroupReduce, Filter, Project…
- Aggregations by field name or position
  - Sum, Min, Max, MinBy, MaxBy, Count…

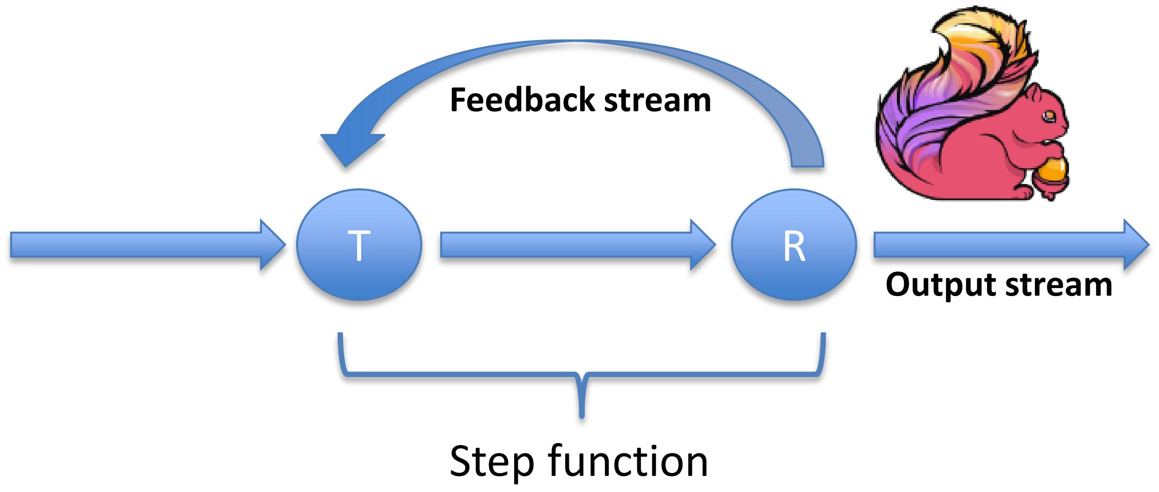


Binary stream transformations

- Apply shared transformations on streams of different types.
- Shared state between transformations
- *CoMap, CoFlatMap, CoReduce…*

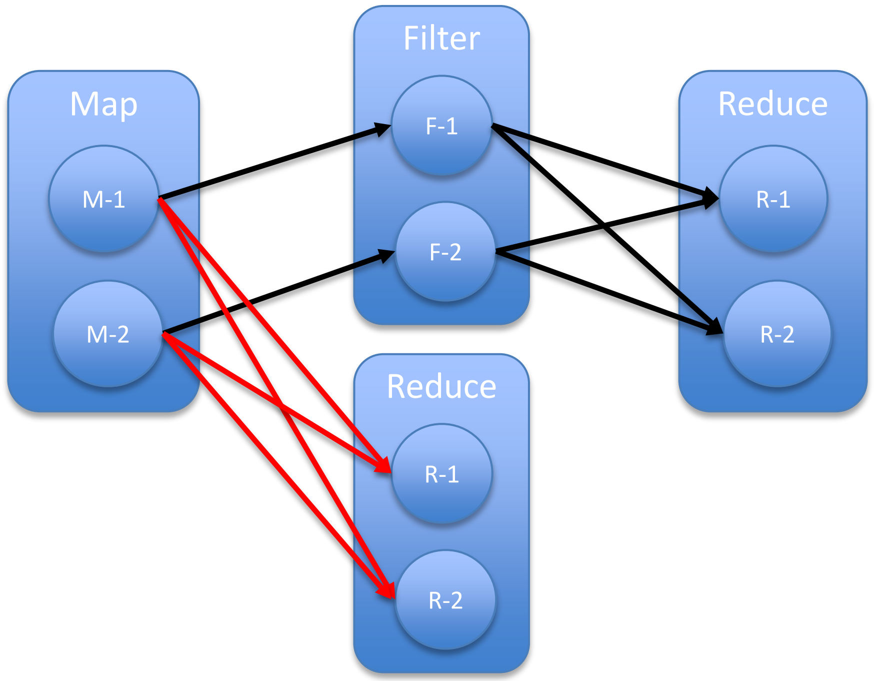
```
public interface CoMapFunction<IN1, IN2, OUT> {

    public OUT map1(IN1 value);
    public OUT map2(IN2 value);

}
```

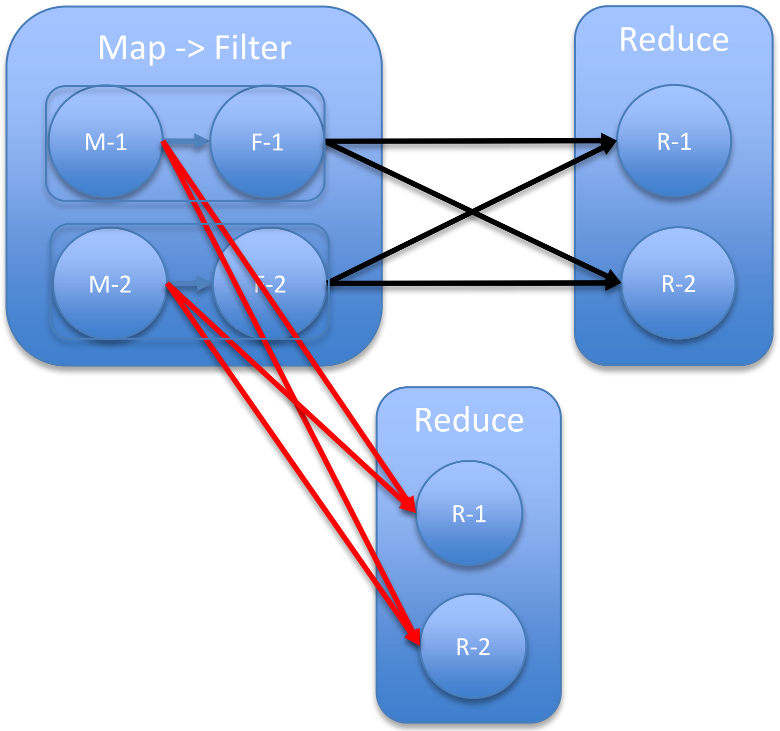
Iterative stream processing



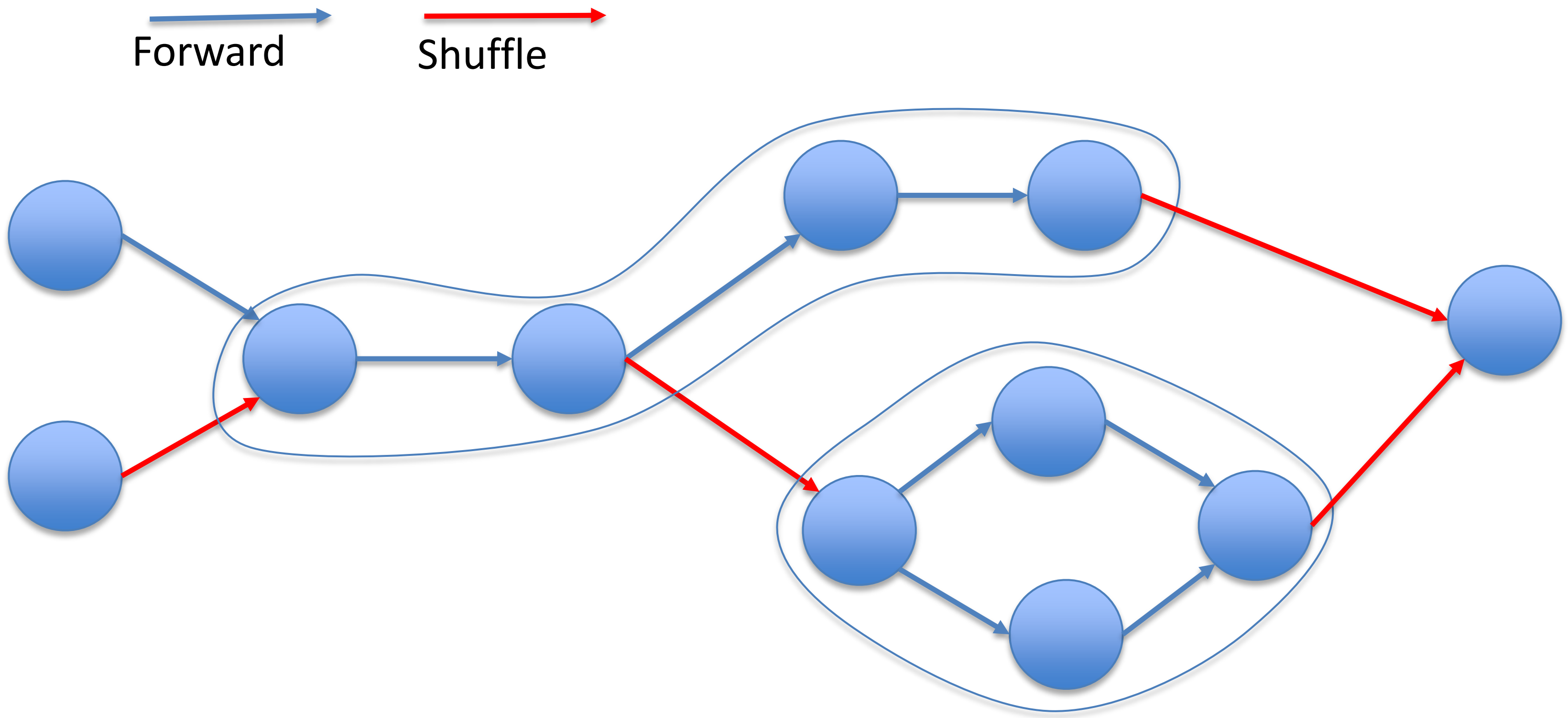
```
def iterate[R](
        stepFunction: DataStream[T] => (DataStream[T], DataStream[R]),
        maxWaitTimeMillis: Long = 0 ): DataStream[R]
```

Operator chaining



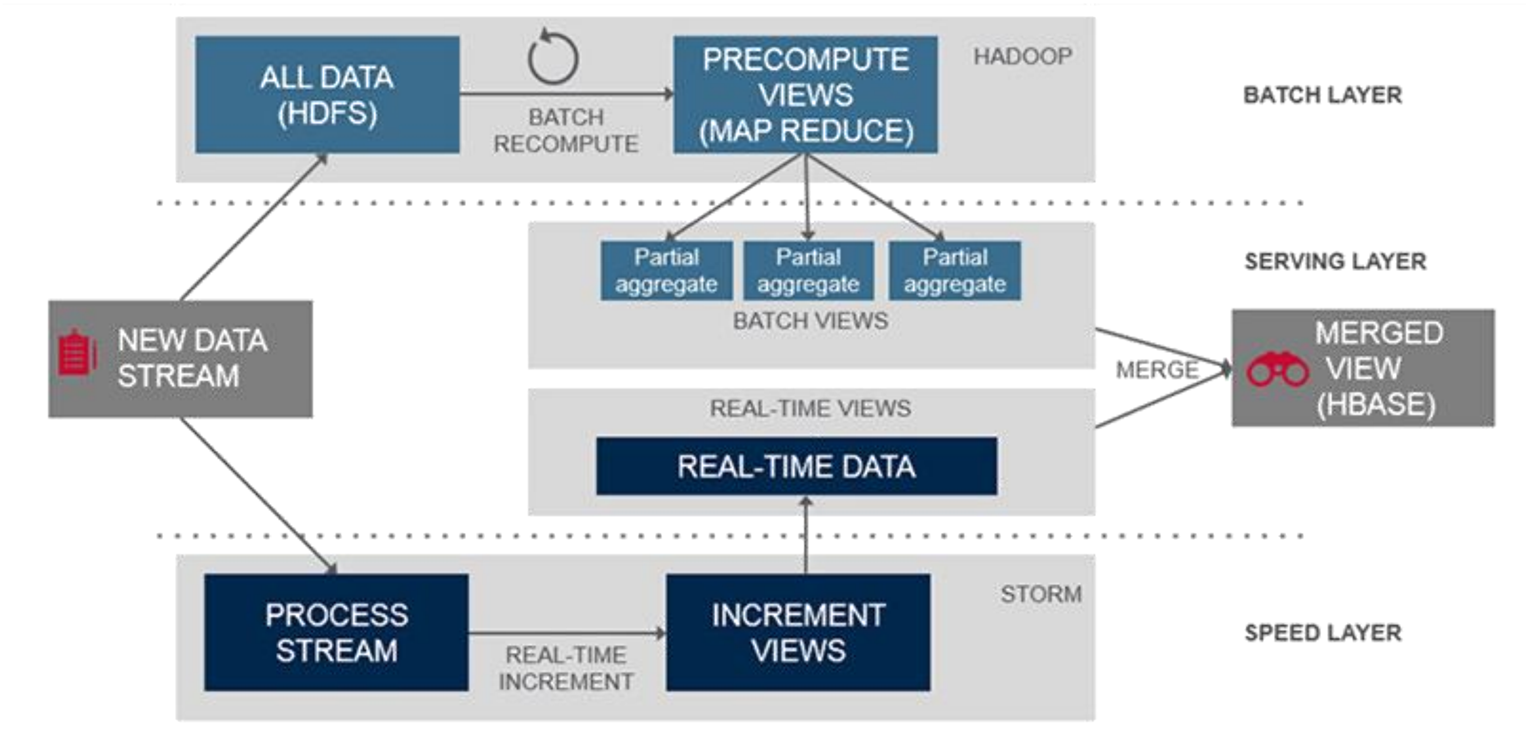


Processing graph with chaining



Lambda architecture

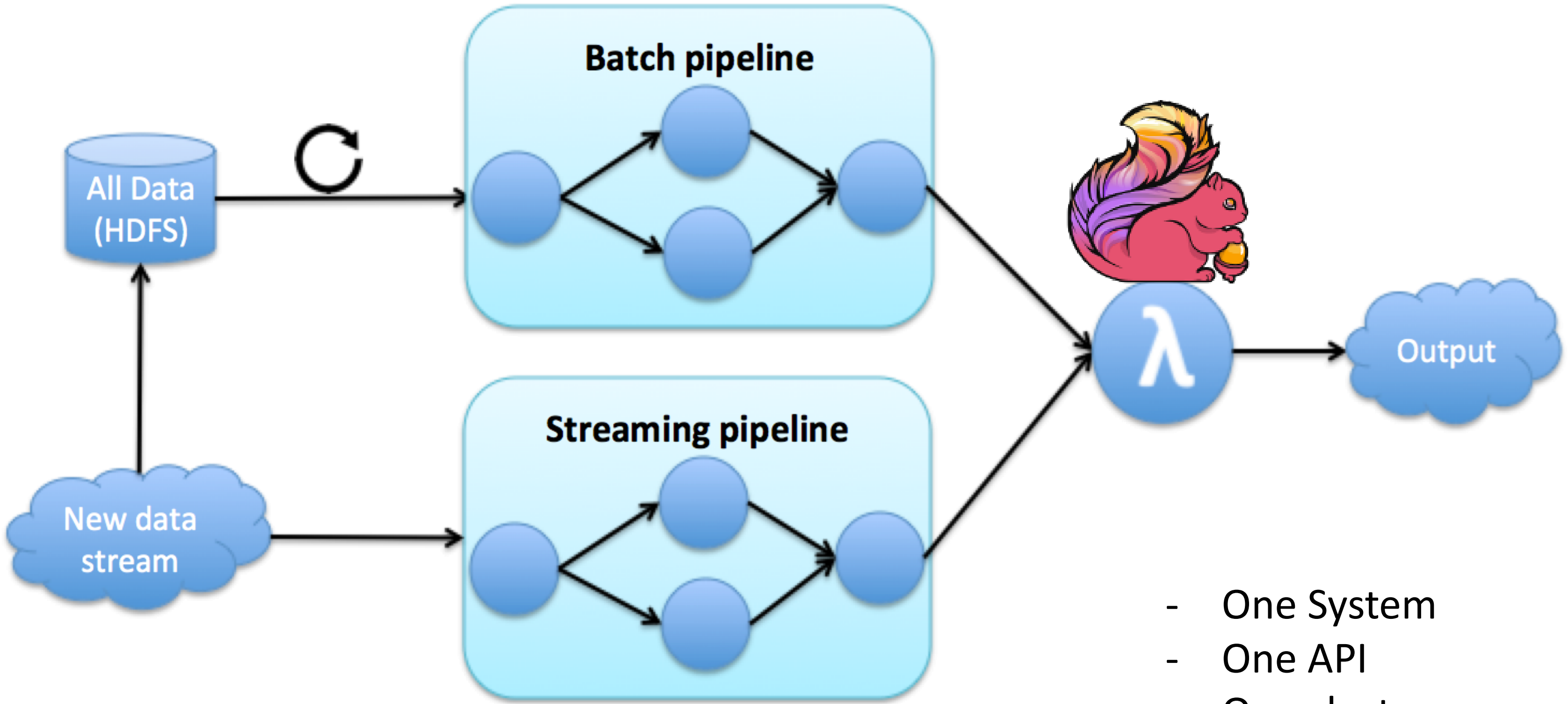
In other systems



Source: https://www.mapr.com/developercentral/lambda-architecture

Lambda architecture

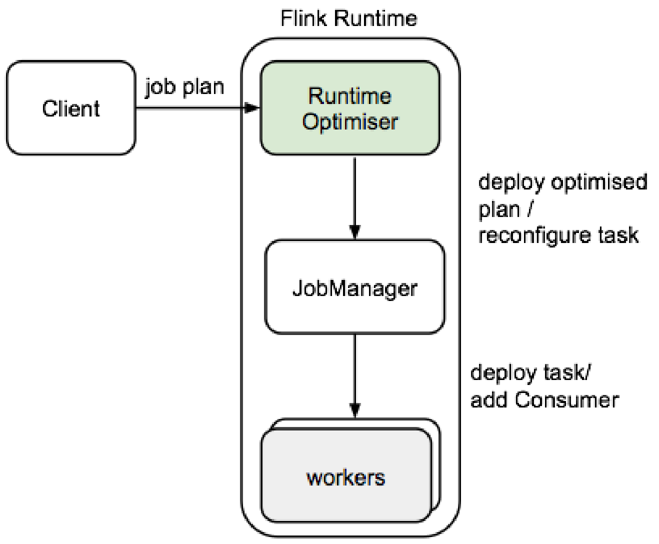
In Apache Flink

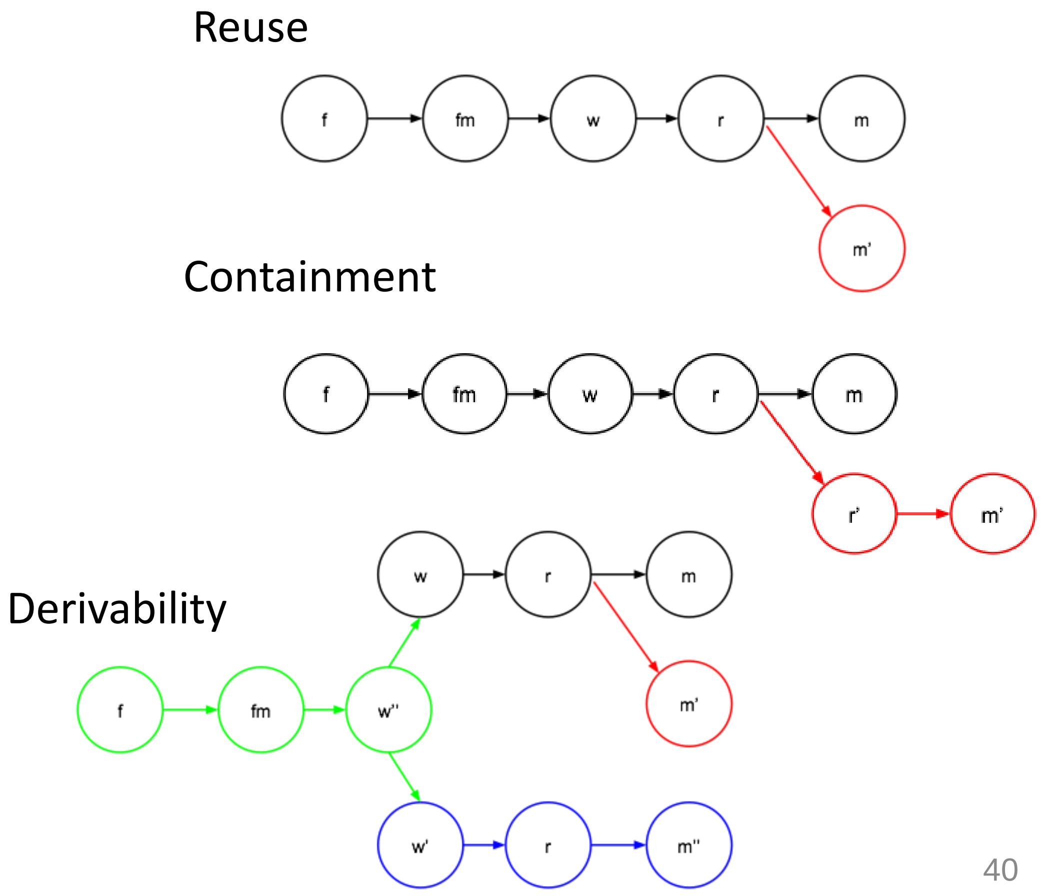


- One System
- One API
- One cluster

Query Optimisations

- Reusing Intermediate Results Between Operators


Flink Runtime
Client
job plan
Runtime Optimiser
deploy optimised plan / reconfigure task
JobManager
deploy task/ add Consumer
workers

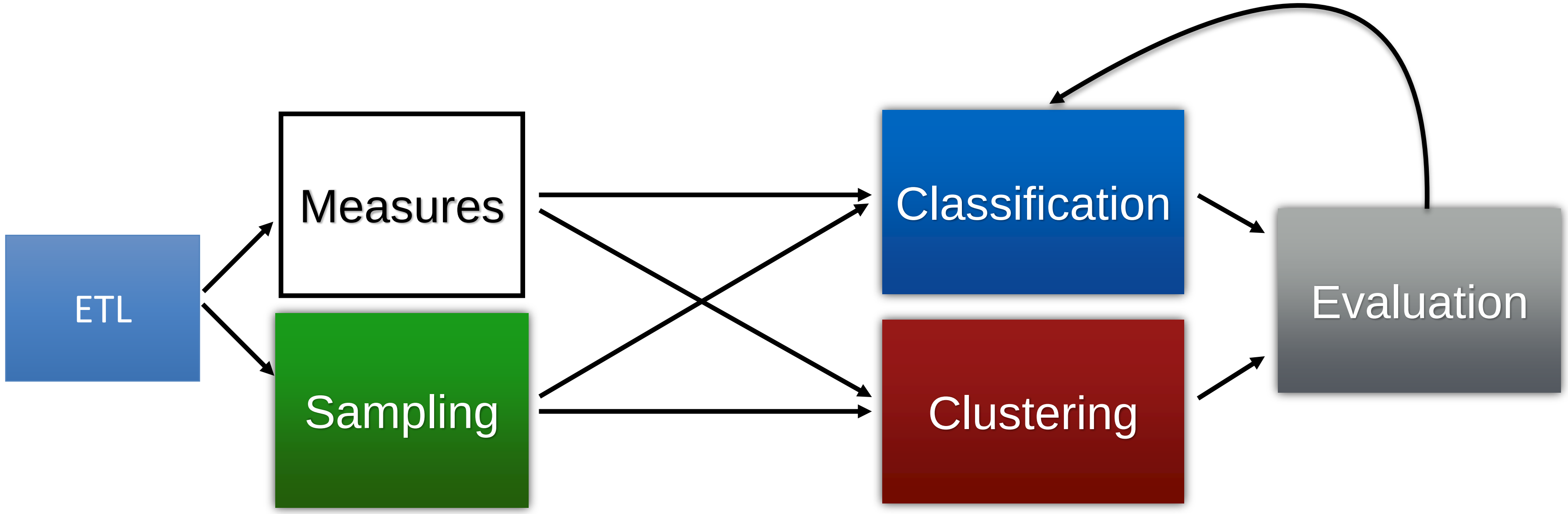


Scala Interoperability

- Seamlessly integrate Flink streaming programs into scala pipelines
- Scala streams implicitly converted to DataStreams
- In the future the output streams will be converted back to Scala streams

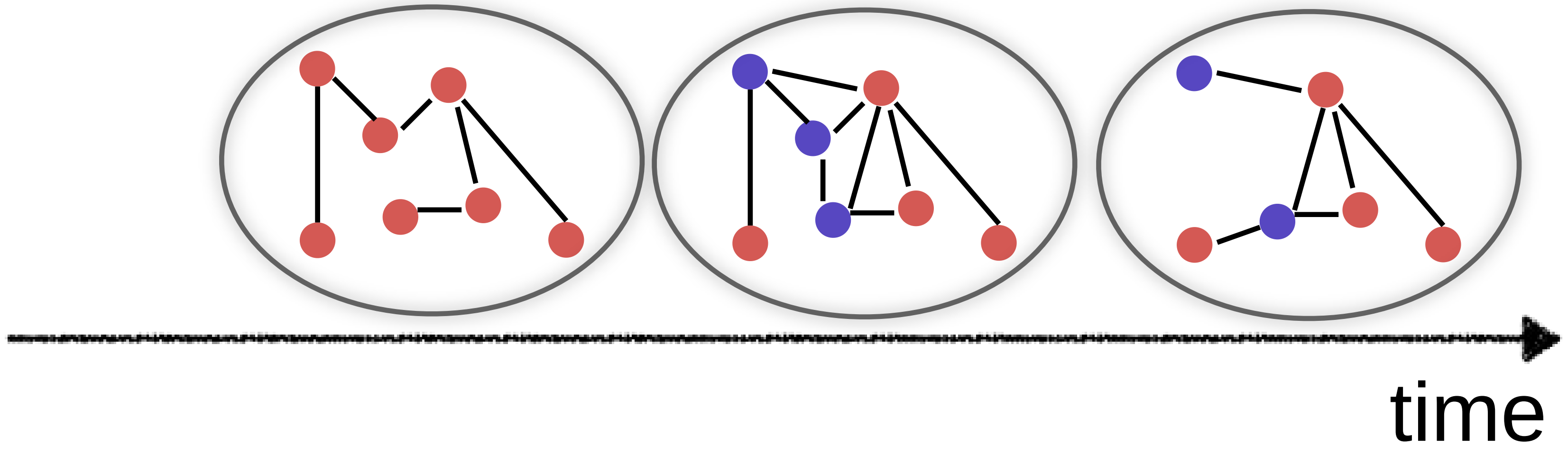
```
fibs.window(Count of 4).reduce((x,y)=>x+y).print

def fibs():Stream[Int] = {0 #::
fibs.getExecutionEnvironment.scanLeft(1)(_ + _)}
```

Machine Learning Pipelines



- Mixing periodic ML batch components with streaming components

Streaming graphs



- Streaming new edges
- Keeping only the fresh state
- Continuous graph analytics