Do try this at home

declarative stream data processing framework

whoami: Cyrill

Luxeria Talks
08.04.2020

Disclaimer

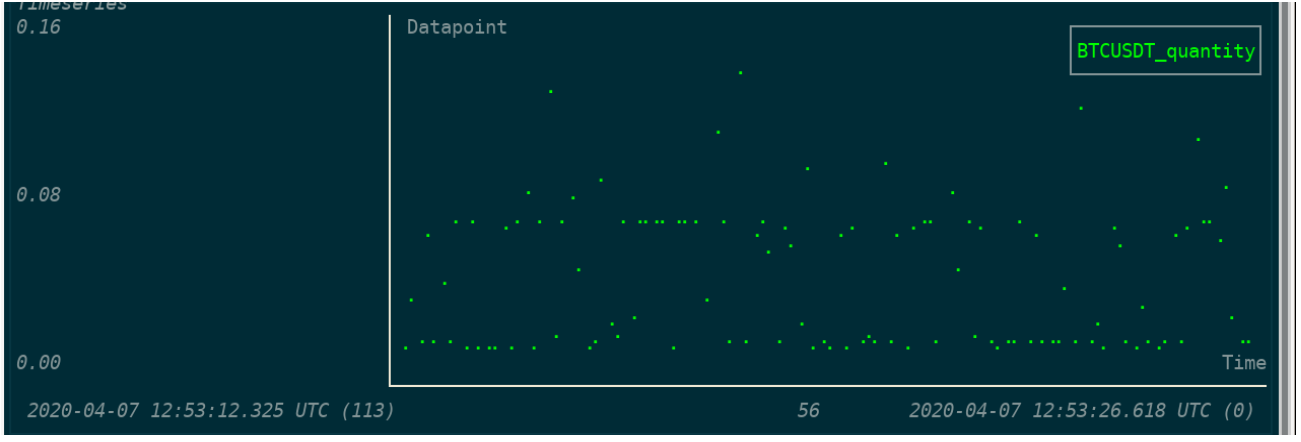
- This is just a hobby project
- I’m not a data scientist at all

  → If you are, I welcome discussion after the talk :-)

How this got started

- Background: Trading Robots
- *“… a data stream is a sequence of digitally encoded coherent signals…”*

  https://en.wikipedia.org/wiki/Data_stream



Motivation

- *“Whenever 1000 bitcoins were traded, how many Ethereum trades took place during this time period? I want this information in real time.”*

→ Given a stream of trade data, I want to…

- Create an event whenever 1000 btc were traded
- Count how much Ethereum was traded since last event
- Yield this event to a sink for further processing

Example problem

timestamp	btc_volume	eth_trades
1586266495	1000.0678547757	1325
1586266526	1000.8439159264	1284
1586266532	1018.89622194	764
1586266538	1121.7621517515	767

This problem is already solved!?

- Bothers entire academic fields
  - Stream data processing (ask @bio)
  - High Performance Computing
  - Statistics, machine learning, “big data”, whatnot
- Promising existing solutions:
  - Apache: **Spark**, **Flink**, **Hadoop**, **Storm**, … (mostly Java)
  - **Onyx** (http://onyxplatform.org, clojure)
  - **Timely**/**Differential Data** (rust)
  - **Wallaroo** (https://github.com/WallarooLabs/wallaroo, python + C)
- Many books to read about data processing in general

  → It is complicated

  → It is easy to get wrong
- Why do I still bother?

Issue #1: I’m no (data) scientist

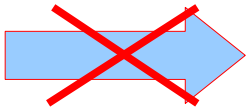
- Sophisticated APIs
- Bound to specific ecosystems
- Best fit in academic deployments
  - Trouble deploying to kubernetes
  - Probably requires a datacenter

Spark reduce Example Using Java 8

```
package com.backtobazics.sparkexamples;

import java.util.Arrays;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function2;

public class ReduceExample {
    public static void main(String[] args) throws Exceptio
        JavaSparkContext sc = new JavaSparkContext();

        //Reduce Function for cumulative sum
        Function2<Integer, Integer, Integer> reduceSumFunc

        //Reduce Function for cumulative multiplication
        Function2<Integer, Integer, Integer> reduceMulFunc

        // Parallelized with 2 partitions
        JavaRDD<Integer> rddX = sc.parallelize(
                Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 1
                2);

        // cumulative sum
        Integer cSum = rddX.reduce(reduceSumFunc);
        // another way to write
        Integer cSumInline = rddX.reduce((accum, n) -> (ac

        // cumulative multiplication
        Integer cMul = rddX.reduce(reduceMulFunc);
        // another way to write
        Integer cMulInline = rddX.reduce((accum, n) -> (ac

        System.out.println("cSum: " + cSum + ", cSumInline
```

Issue #2: Not exactly declarative

- Writing code (Java or python) as configuration
- Doesn’t sound ideal to me
- I better like “declarative” things. Describe **what** state you want instead of **how** to get to it



Spark reduce Example Using Java 8

```
package com.backtobazics.sparkexamples;

import java.util.Arrays;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function2;

public class ReduceExample {
    public static void main(String[] args) throws Exceptio
        JavaSparkContext sc = new JavaSparkContext();

        //Reduce Function for cumulative sum
        Function2<Integer, Integer, Integer> reduceSumFunc

        //Reduce Function for cumulative multiplication
        Function2<Integer, Integer, Integer> reduceMulFunc

        // Parallelized with 2 partitions
        JavaRDD<Integer> rddX = sc.parallelize(
                Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 1
                2);

        // cumulative sum
        Integer cSum = rddX.reduce(reduceSumFunc);
        // another way to write
        Integer cSumInline = rddX.reduce((accum, n) -> (ac

        // cumulative multiplication
        Integer cMul = rddX.reduce(reduceMulFunc);
        // another way to write
        Integer cMulInline = rddX.reduce((accum, n) -> (ac

        System.out.println("cSum: " + cSum + ", cSumInline
```

Issue #3: I need it more dynamic

- The example use case, again:

  Whenever 1000 bitcoins were traded, how many Ethereum trades took place during this time period?

- What about a parameter search?

  *Whenever **N** bitcoins were traded, how many **Altcoins** trades took place during this time period?*

  *Where **N*** = [50..5000]

  *and **Altcoins*** = [ETH, XRP, XMR, BNB, XLM, LTC, XTZ]

Issue #3: I need it more dynamic

- After some very vague research:
  - None of the state-of-the-art solution fits all my wishlist
  - Everything could probably be done, somehow, probably…
  - … this would most certainly need significant amounts of Java, Rust or Python coding efforts

“… significant amounts of … coding efforts”

Tubes

- Decided to DIY
  - Blackjack ✓
  - Hookers ✓
  - Unicorns ✓
- Build it on the shoulders of async rust ecosystem
  - Should be reasonably fast on commodity hardware ✓
  - Design my own insane API ✓
  - Compiles to single binary ✓
    - Docker ✓
    - Kubernetes ✓
- Main reason: Lots of fun ✓

Core concepts

- Pipeline
  - Datapoint
    - Timestamp
    - Fields (values)
  - Tubes
    - Generator
    - Throughput
    - Sink
- Events
- ~~Black magic~~

Pipeline and Datapoints

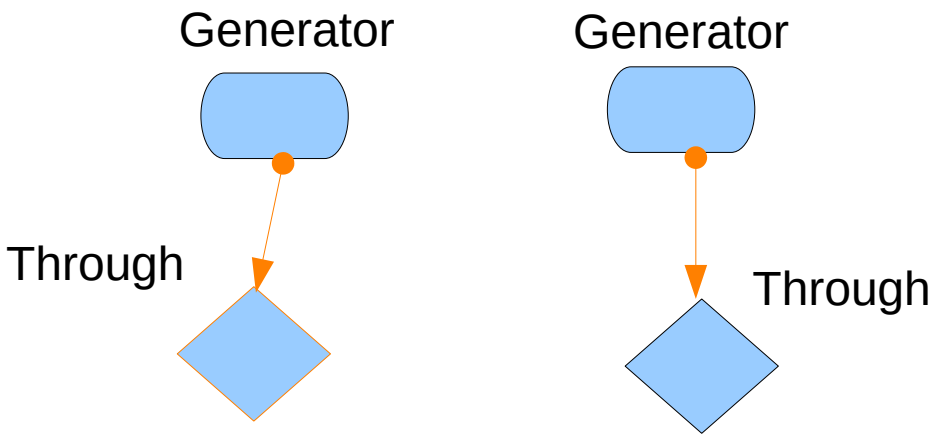
- Pipeline
  - Datapoint
    - Timestamp
    - Fields (values)

Generator tube

- Generators
  - CSV Files
  - Databases (TSDB)
  - WebSockets
  - Message Queue
  - ...

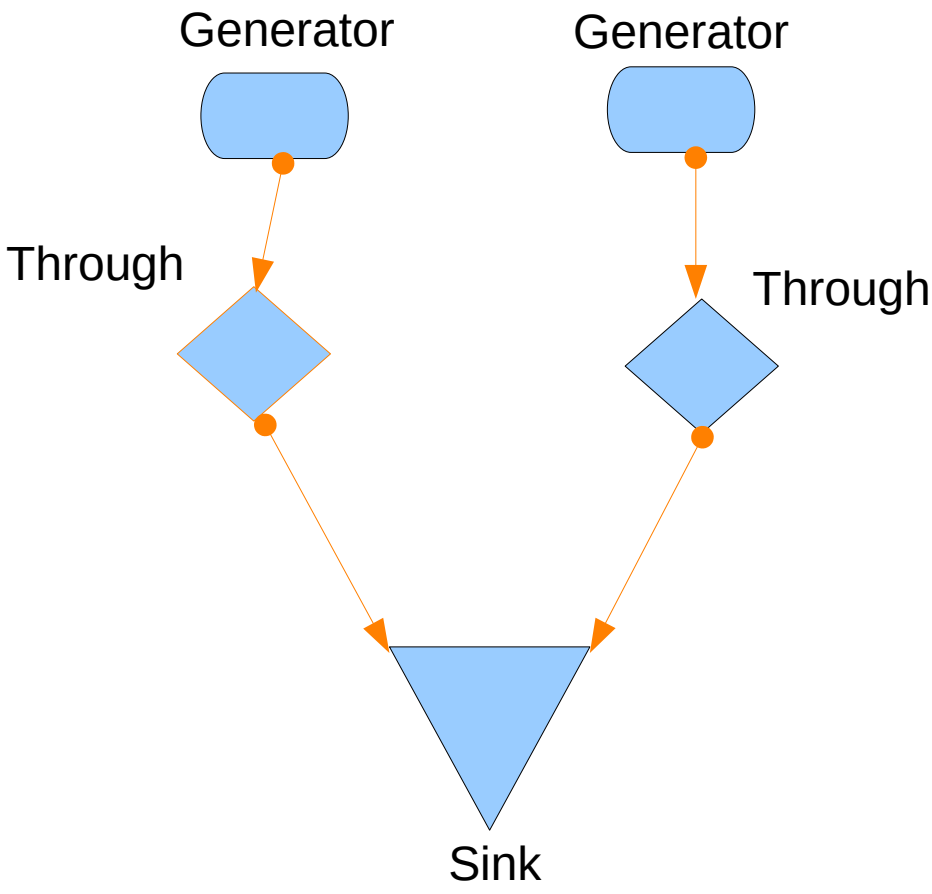
Throgh tube

- Through
  - SUM
  - COUNT
  - FIRST/LAST/MAX/MIN
  - …



Sink tube

- Sink
  - CSV Files
  - Databases
  - Message Queue
  - STDOUT
  - ...

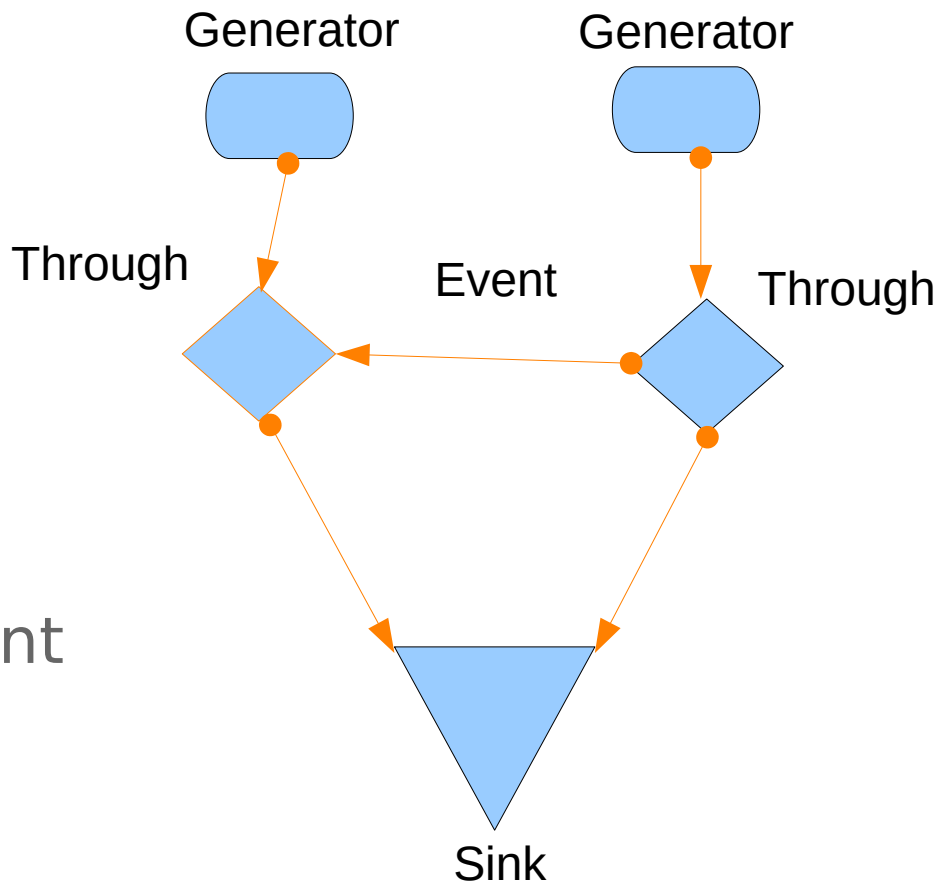


Events

- Events
  - Communication between tubes
  - Reflects “business logic”

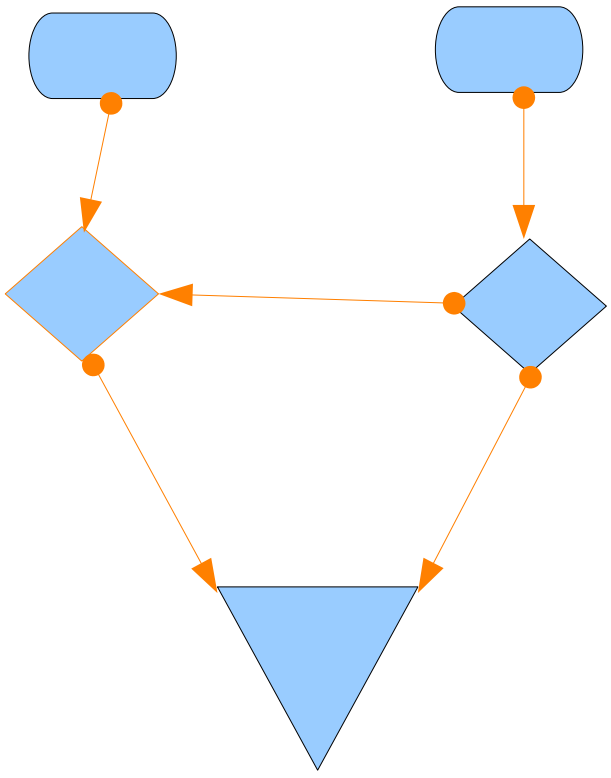
→ *1000 bitcoins traded*

Would be an example event



Design

- Inspired by Actor pattern
- Communication via channels
- “Processing nodes”

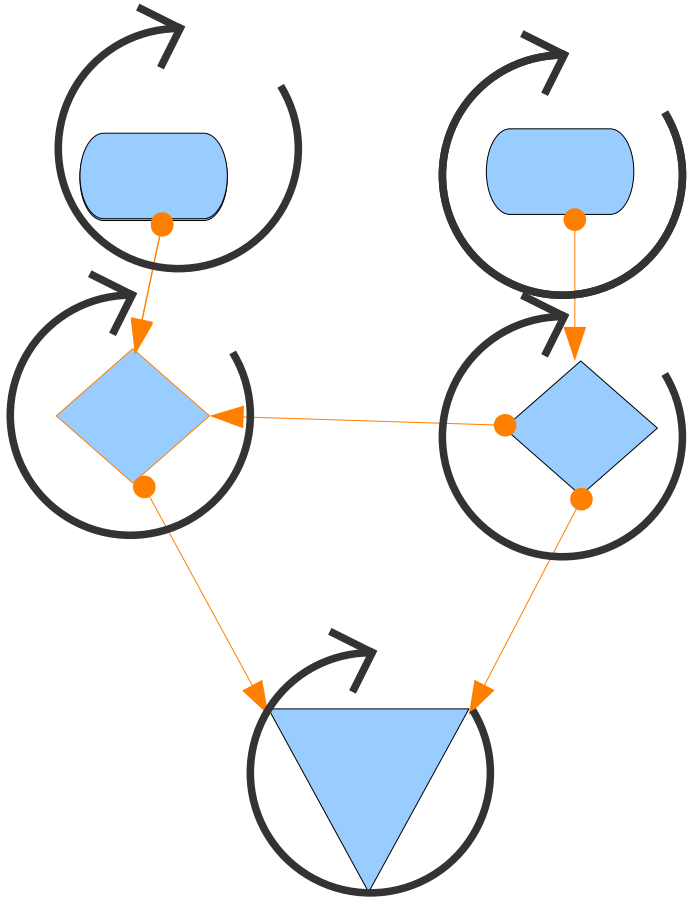


Design

- Inspired by Actor pattern
- Communication via channels
- “Processing nodes”

- Each node is a ***future***

  → Poll to completion
- Running on a threadpool
- Scales *good enough*



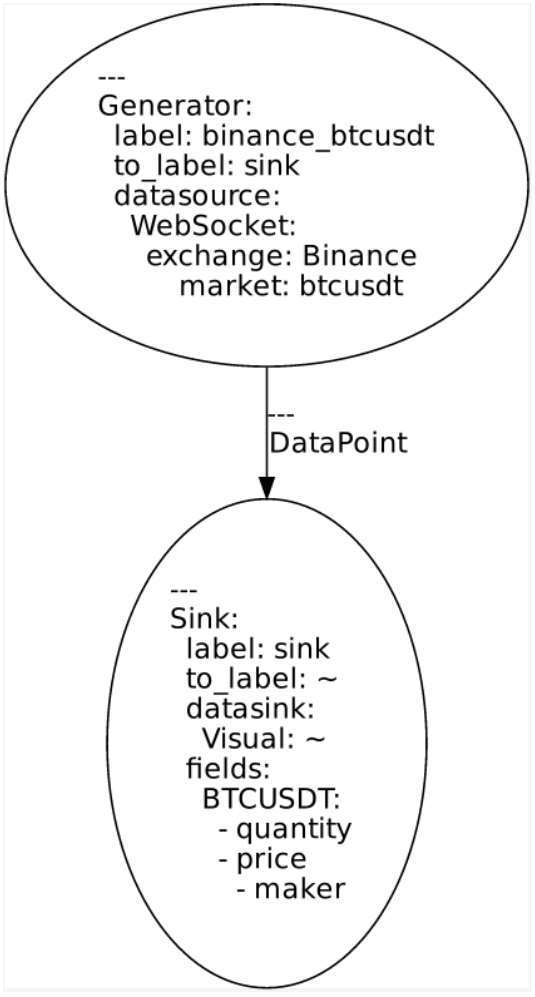
Configuration

- YAML
- Send via gRPC
- Or use the local command line interface

```
---
pipeline_type: Stream

tubes:

- Generator:
    label: binance_btcusdt
    datasource:
        WebSocket:
            exchange: Binance
            market: btcusdt
    to_label: sink

- Sink:
    label: sink
    datasink:
      Visual:
    fields:
      BTCUSDT:
      - quantity
      - price
      - maker
```

Demo time!

- Straightforward example
- Only Generator → Sink
- Streams trade events from a cryptocurrency exchange to SDTOUT



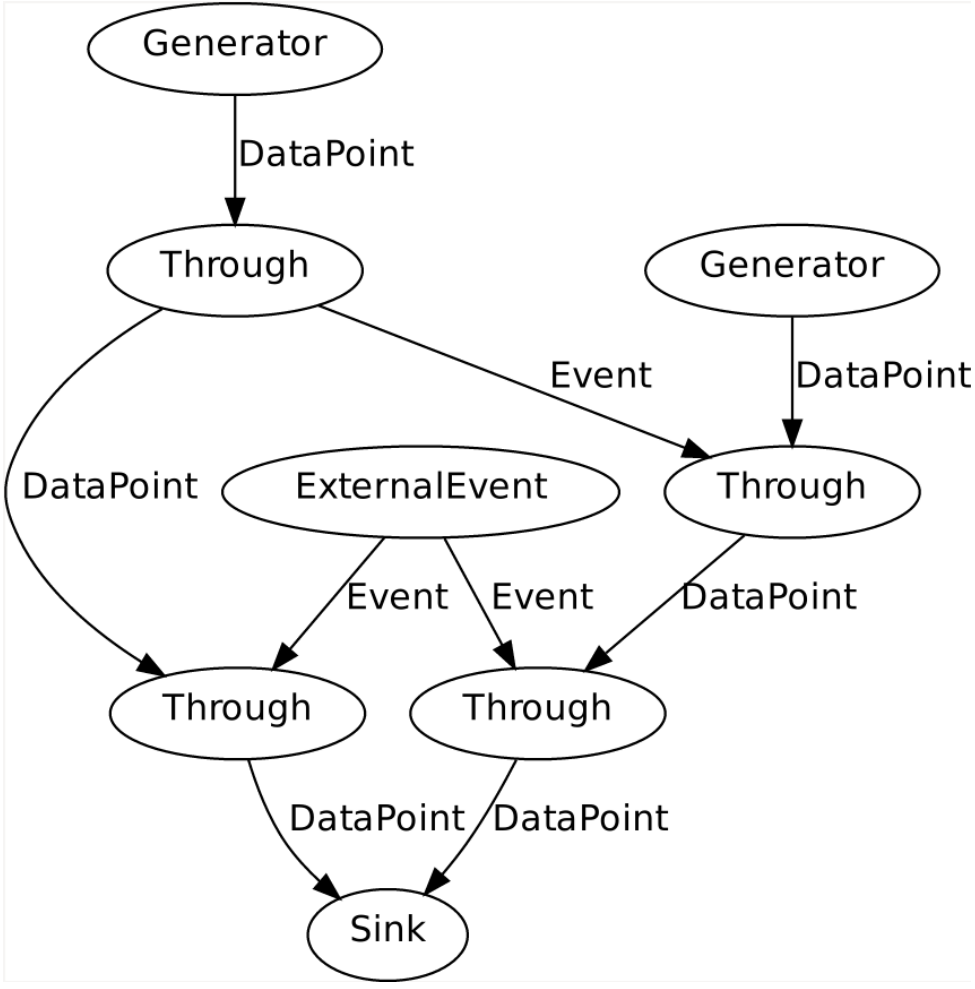
A more involved example

- *Whenever 1000 bitcoins were traded, how many Ethereum trades took place during this time period?*
- Chain tubes arbitrarily

```
---
pipeline_type:
  Batch:
    - "2017-01-01T00:00:00Z"
    - "2018-01-01T00:00:00Z"

tubes:

- Generator:
    label: btc_data
    datasource:
      CSV:
testdata/gemini_BTCUSD_2017_1min.csv
    to_label: btc_vol_sum

- Generator:
    label: eth_data
    datasource:
      CSV:
testdata/gemini_ETHUSD_2017_1min.csv
    to_label: eth_counter

- Through:
    label: btc_vol_sum
    to_label: btc_static_timer_ticker
    reducer:
      SUM:
        field: Volume
    event:
      Produce:
        GREATER:
          field: sum_Volume
          threshold: 1000

- Through:
    label: btc_static_timer_ticker
    to_label: sink
    reducer: NOP
    event:
      Consume: extevt

- Through:
    label: eth_counter
    to_label: eth_static_time_ticker
    reducer: COUNT
    event:
      Consume: btc_vol_sum

- Through:
    label: eth_static_time_ticker
    to_label: sink
    reducer: NOP
    event:
      Consume: extevt

- Sink:
    label: sink
    datasink:
      #Visual:
      JSON:
        /tmp/pipeline_sink_only.json
    fields:
      btc_data:
      - sum_Volume
      eth_data:
      - count

- ExternalEvent:
    label: extevt
    condition:
      StaticTimeTick:
        from: "2017-01-01T00:00:00Z"
        to: "2018-01-01T00:00:00Z"
        interval:
          secs: 60
          nanos: 0
```

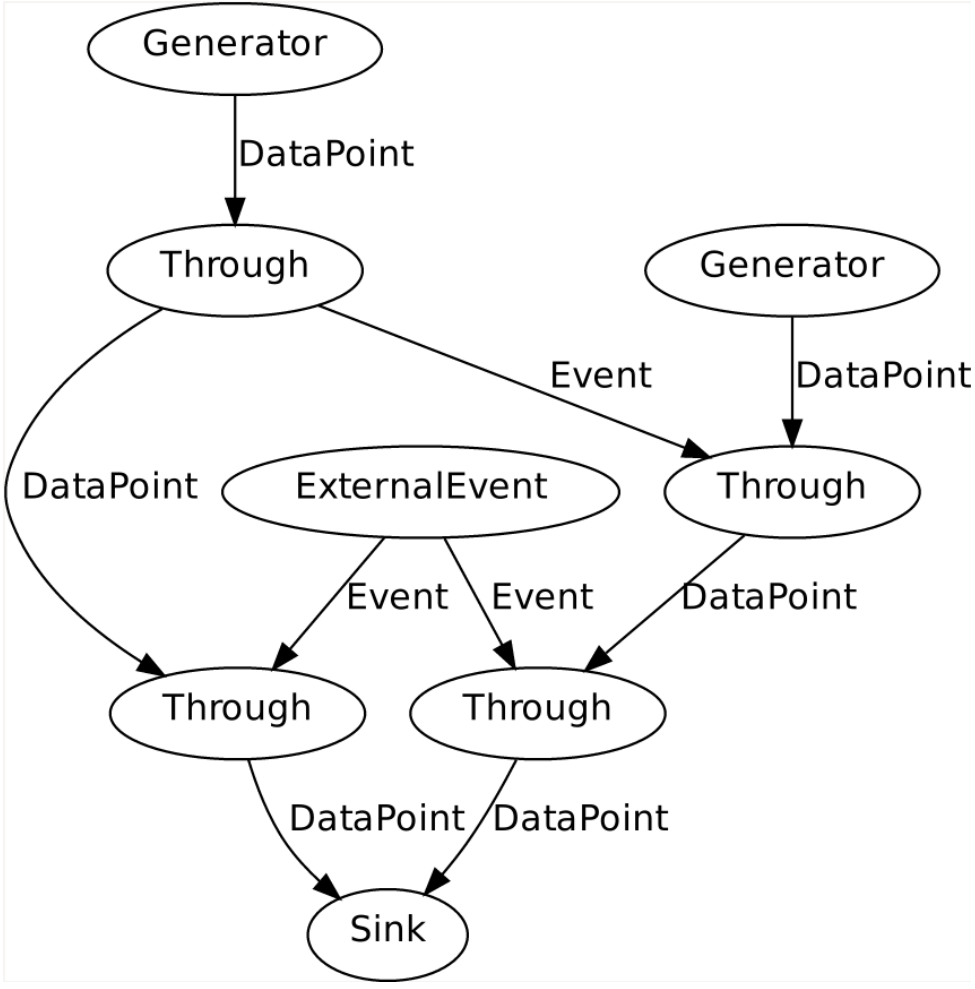
Directed Acyclic Graph

- Config is parsed into Graph
- *Some* sanity checks
- *RUST_LOG=DEBUG*
  → print graphviz xdot



Demo time!

- Lets see it in action!



Closing thoughts

- Not limited to crypto currencies!
- Not open source yet :-(
  - Released a tiny helper crate
  - https://crates.io/crates/poolparty
- Ideally, it will be FOSS in the future :-)
  - Not stable or even feature complete yet

    → Need to refactor some rather hacky things first
  - Some trivial performance optimization opportunities TBD
  - For now I love the freedom to break things quickly

Questions