Systems and Architectures for Big Data AY 2017/2018
University of Rome Tor Vergata

Data stream processing for social network data with Apache Flink, Storm and Kafka Streams on Google Cloud Platform

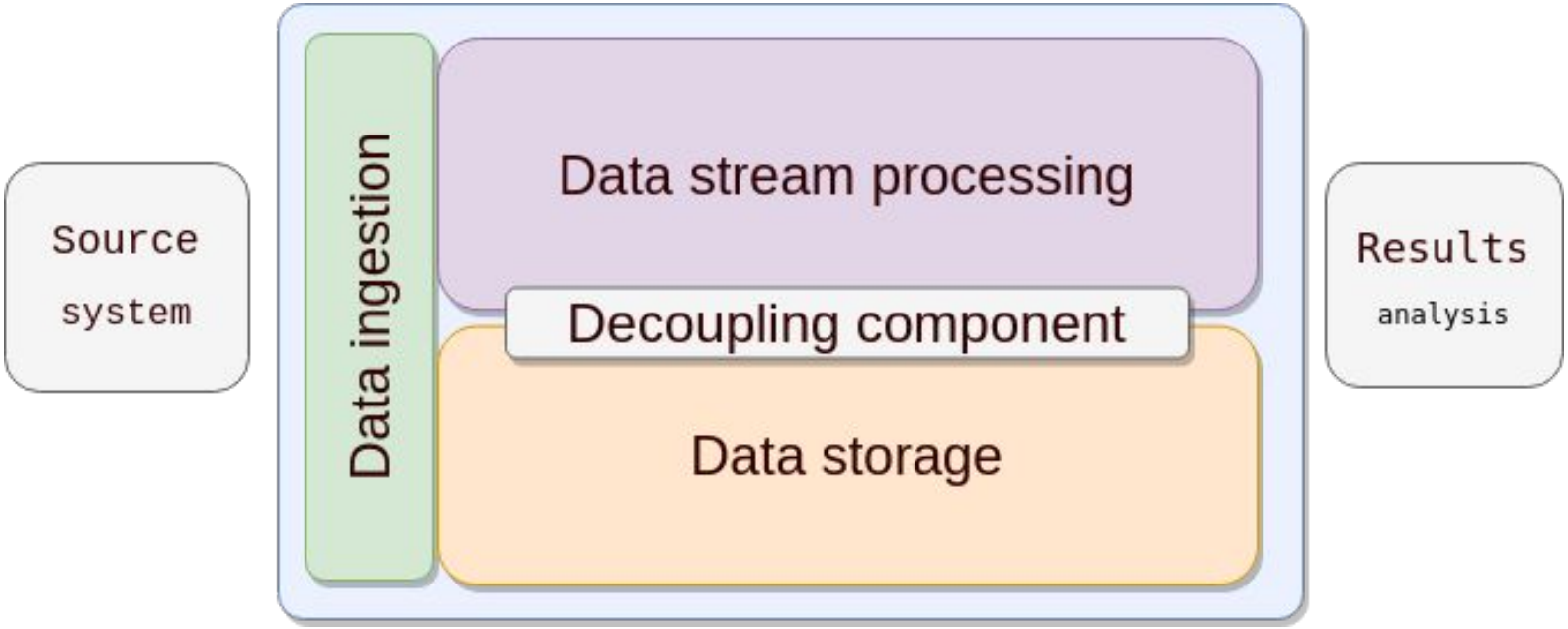
Ovidiu Daniel Barba
Laura Trivelloni
Emanuele Vannacci

1.

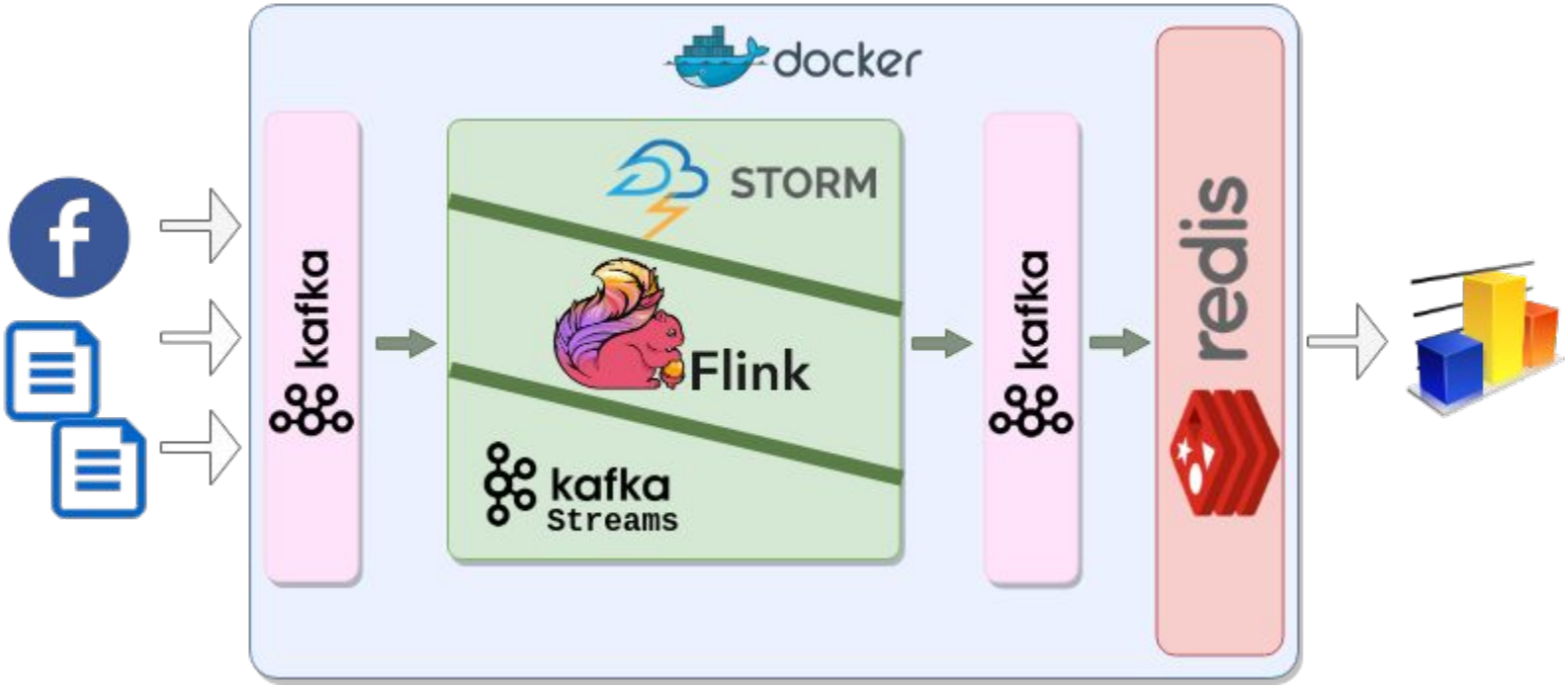
System Architecture

High Level Overview

System Layers



System Layers



Data Processing Layer

- Flink DataStream API
- Kafka Streams DSL API & Processor API
- Storm API
- Topologies composes a multistage stream computation

Data Ingestion & Decoupling

- Inject data from external sources to Kafka input topics
- Results stream published on Kafka output topics

Data Producer

- Read from local file
- Parametrized emission frequency proportional to the real one
- Apache Avro as serialization format

Data Storage

redis

- Redis is an in-memory NoSQL key-value database
- Stream results consumed from Kafka
- Results grouped in Sorted Sets keyed by Avro schema name sorted by statistics timestamp

Results Collector

redis

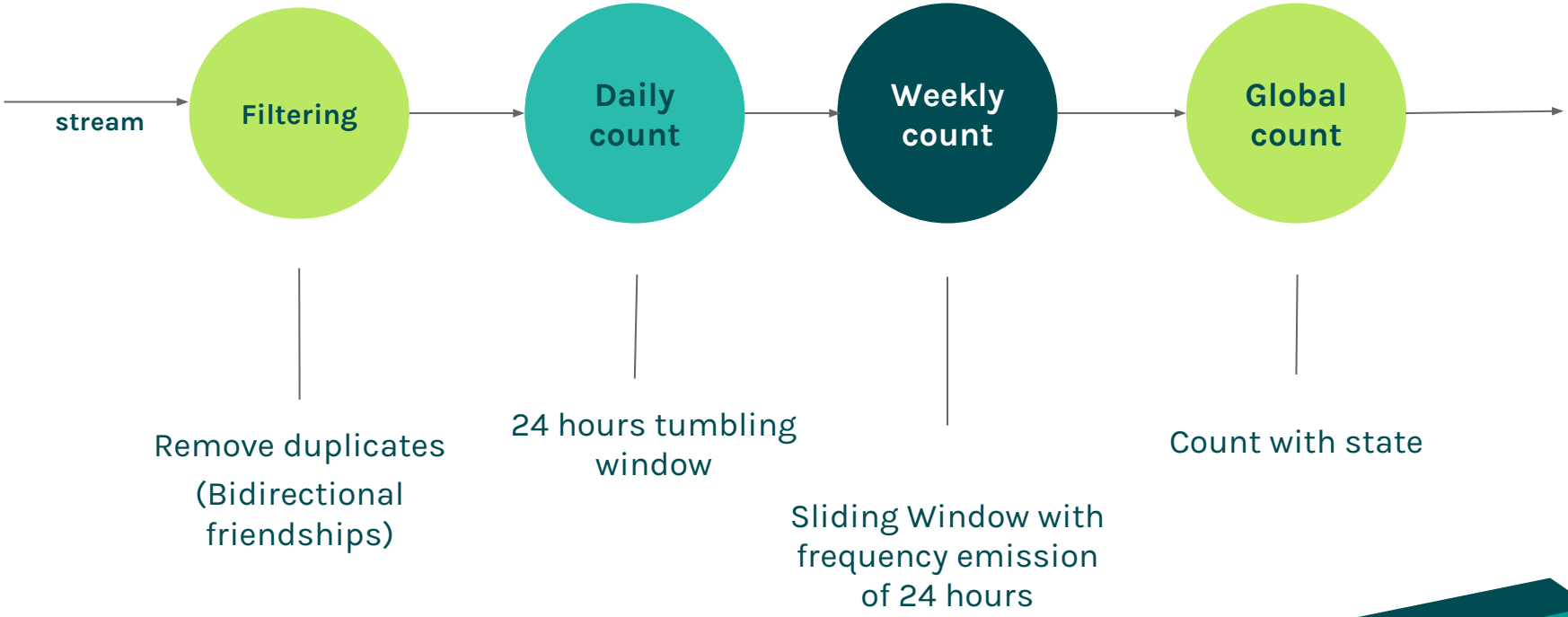
- Consume all new results published on output topics
- Write asynchronously records into Redis

2.

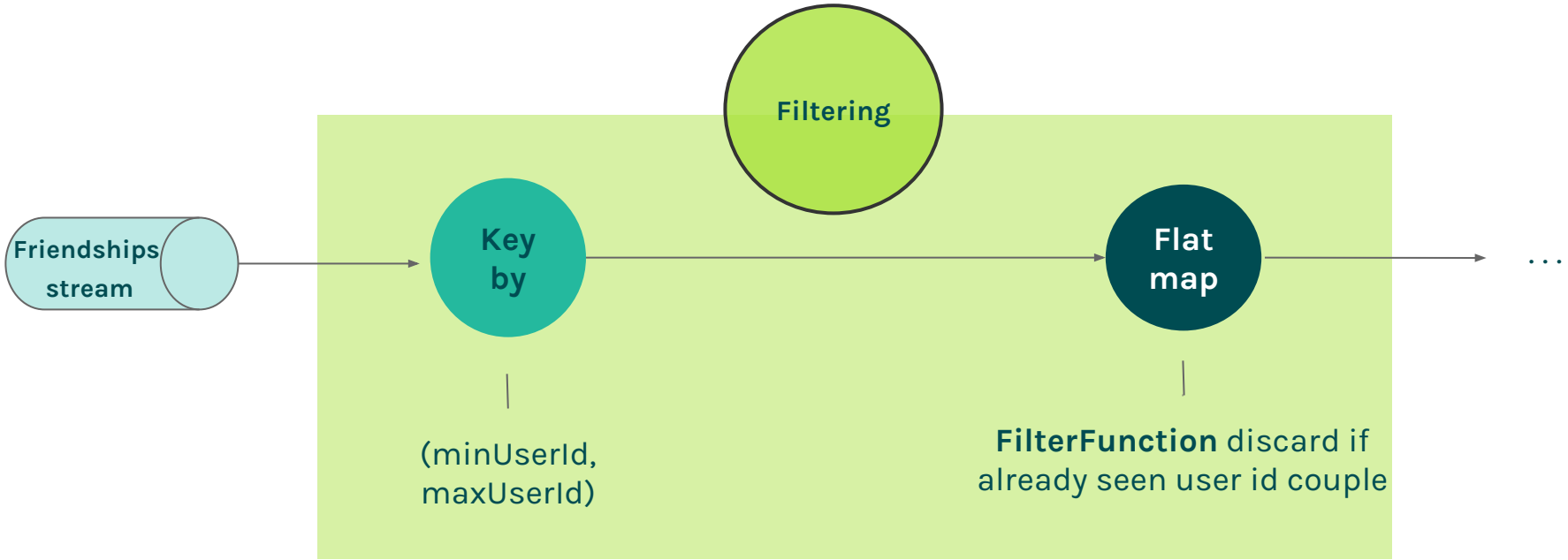
Queries

Detailed queries description

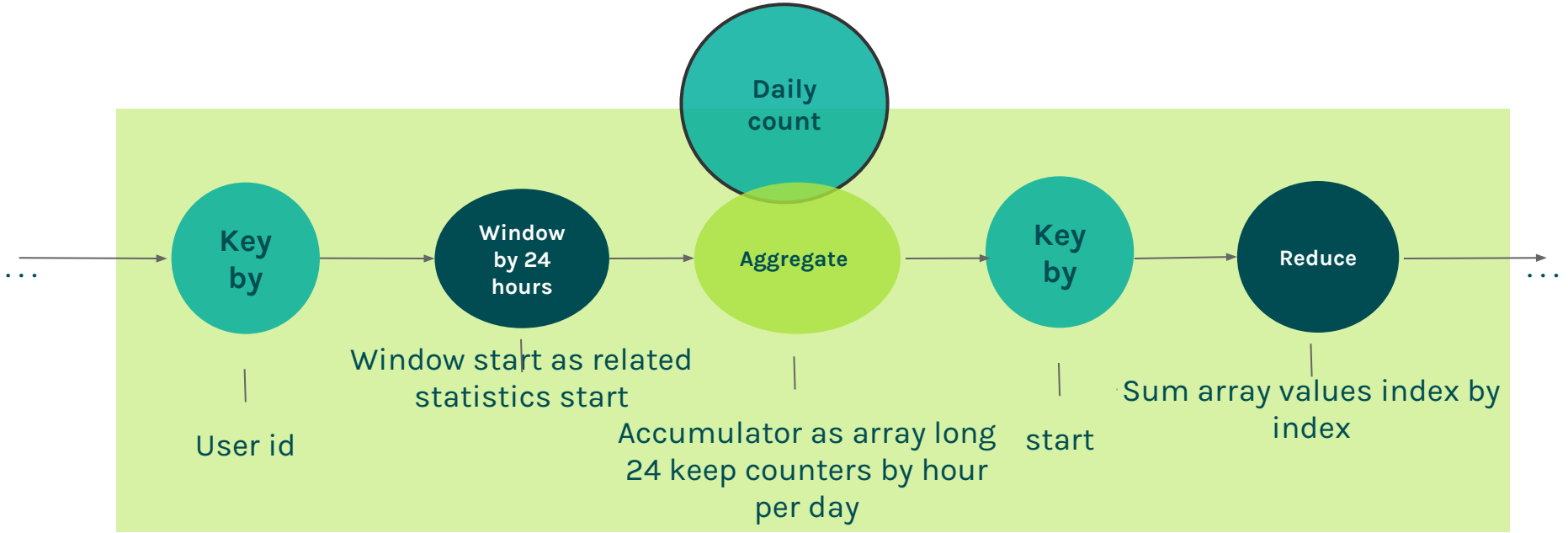
Query 1


stream
Filtering
Daily count
Weekly count
Global count
Remove duplicates (Bidirectional friendships)
24 hours tumbling window
Sliding Window with frequency emission of 24 hours
Count with state

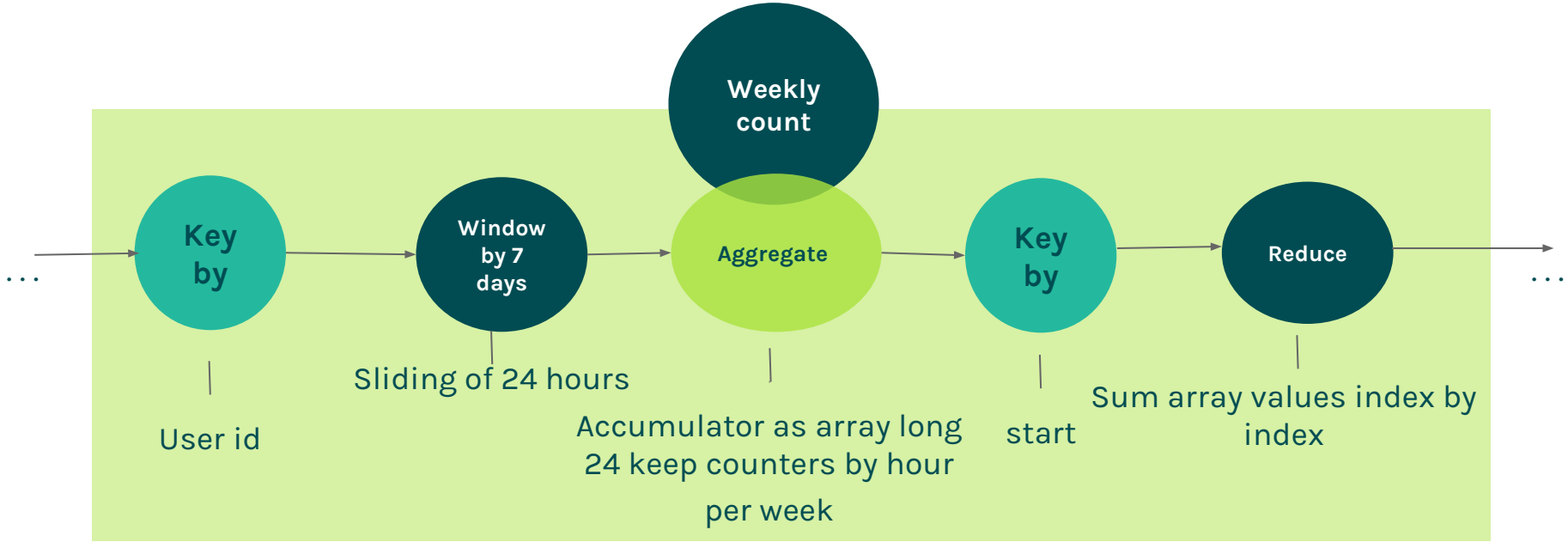
Query 1 - Flink & Kafka Streams



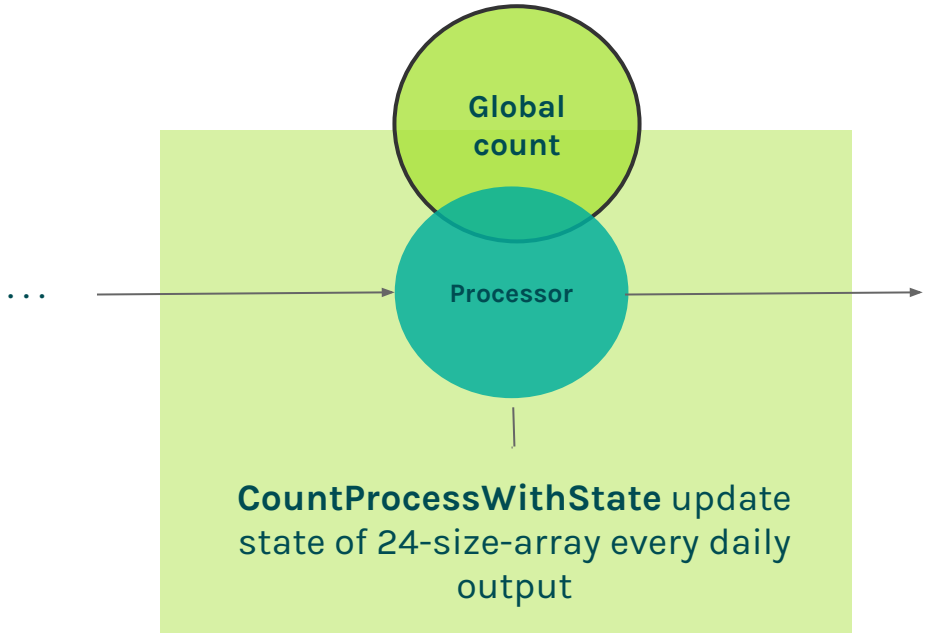
Query 1 - Flink & Kafka Streams



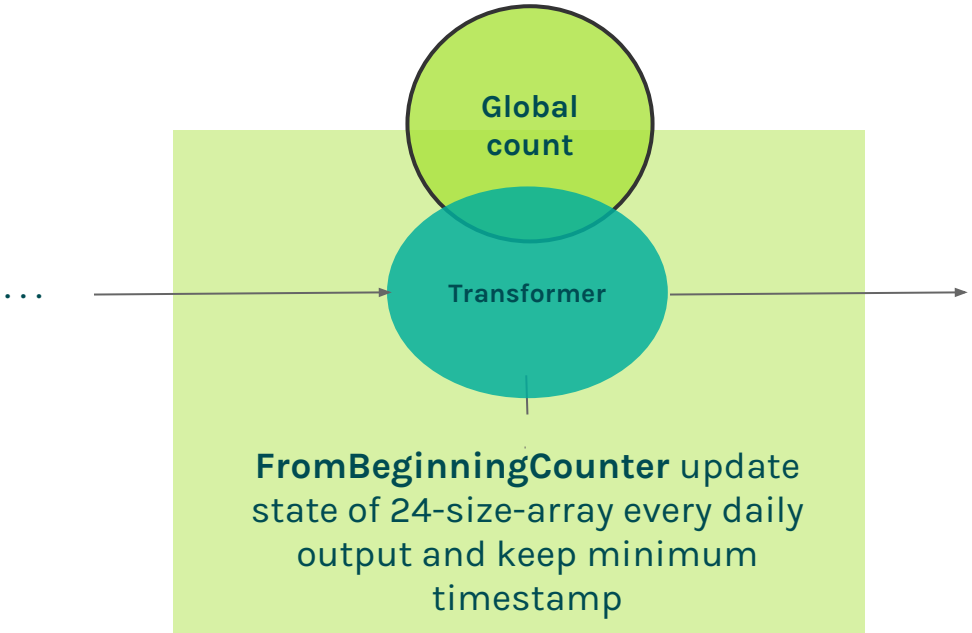
Query 1 - Flink & Kafka Streams



Query 1 - Flink



Query 1 - Kafka Streams



Ranking Queries in Flink

- Goal of Query 2 and 3 is the **Top-10** ranking of most commented posts and most active active user
- Both queries must provide rankings in hourly, daily and weekly event-time windows
- Since they have the same goal, a **general ranking query structure** has been defined and applied to the specific query and related data stream(s)

Ranking Board

- Generic data structure for maintaining the current score of ranked elements (in a *HashMap*)
- Caches and updates the Top-K (implemented with a ListBuffer of size K) so to retrieve it in **constant time**
- Ranked elements are of type *GenericRankElement[A](id: A, score: Score)* where A is data type of ID
- *Score* is a Scala **trait** with methods for adding two scores and returning a “global” score to compare it with others
- Examples : *SimpleScore(val: Int)*, *UserScore(a,b,c: Int)*

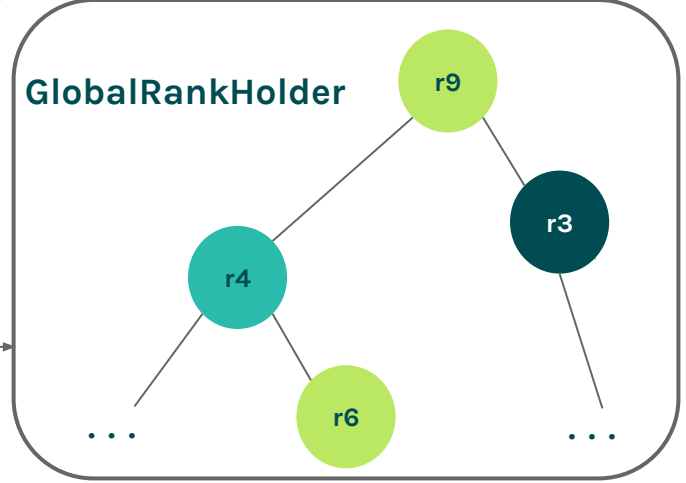
Partial Rankings

- Each **Top-K** returned from the *Board* must be assigned a event timestamp from which the computation begun, becoming a *RankingResult*
- Since **Top-K**s can be partial rankings, a merge method is defined
- Given two partial rankings (r_1 and r_2), the merged ranking r_3 (useful for global ranking) is defined by:
  - $timestamp(r_3) = min(\ timestamp(r_1), timestamp(r_2)\ )$
  - $K(r_3) = min(\ K(r_1), K(r_2)\ )$
  - $rank(r_3) = (\ rank(r_1) + rank(r_2)\ ).topK(\ K(r_3)\ )$

Global Ranking Holder

- Data structure used for holding and updating global rankings
- Ranks are kept in a *TreeMap* ordered by event timestamp
- Offers methods for:
  - Clearing rankings older than delta ms from current event timestamp (save memory)
  - Computing **global ranking** with given partial ranking by checking the *TreeMap* for older matching rankings and merging it with them while also updating the map

Ranking Overview



Partial Ranker Operator

Global Ranker Operator

Board

Top-K

RankElement(id1, s1)

RankElement(id2, s2)

RankElement(id6, s6)

RankElement(id10, s10)

RankElement(id21, s21)

RankElement(id42, s42)

RankElement(id1, s1)

RankElement(id42, s42)

RankElement(id6, s6)

RankElement(id10, s10)

event

timestamp

r1=(t1,rank1,k1)

Merge

r3=(t3,rank3,k3)

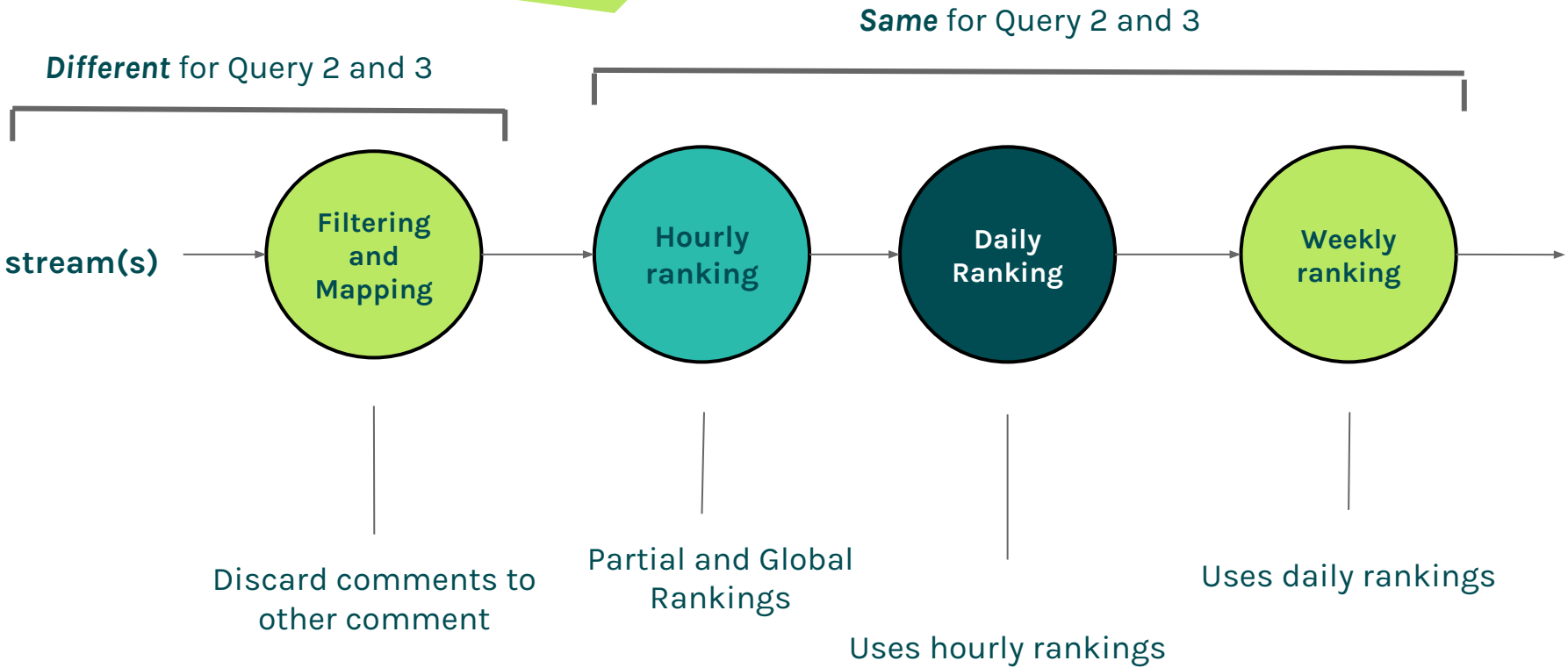
Global rank

event

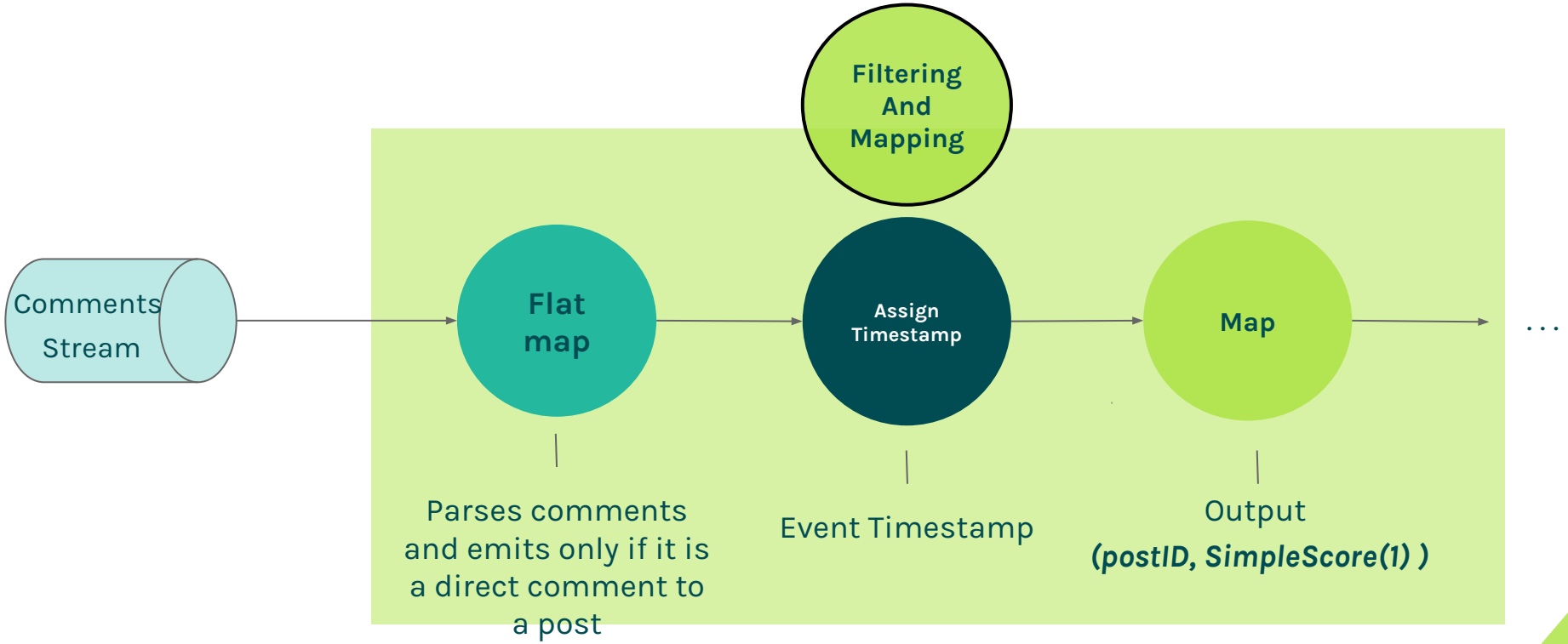
timestamp

r2=(t2,rank2,k2)

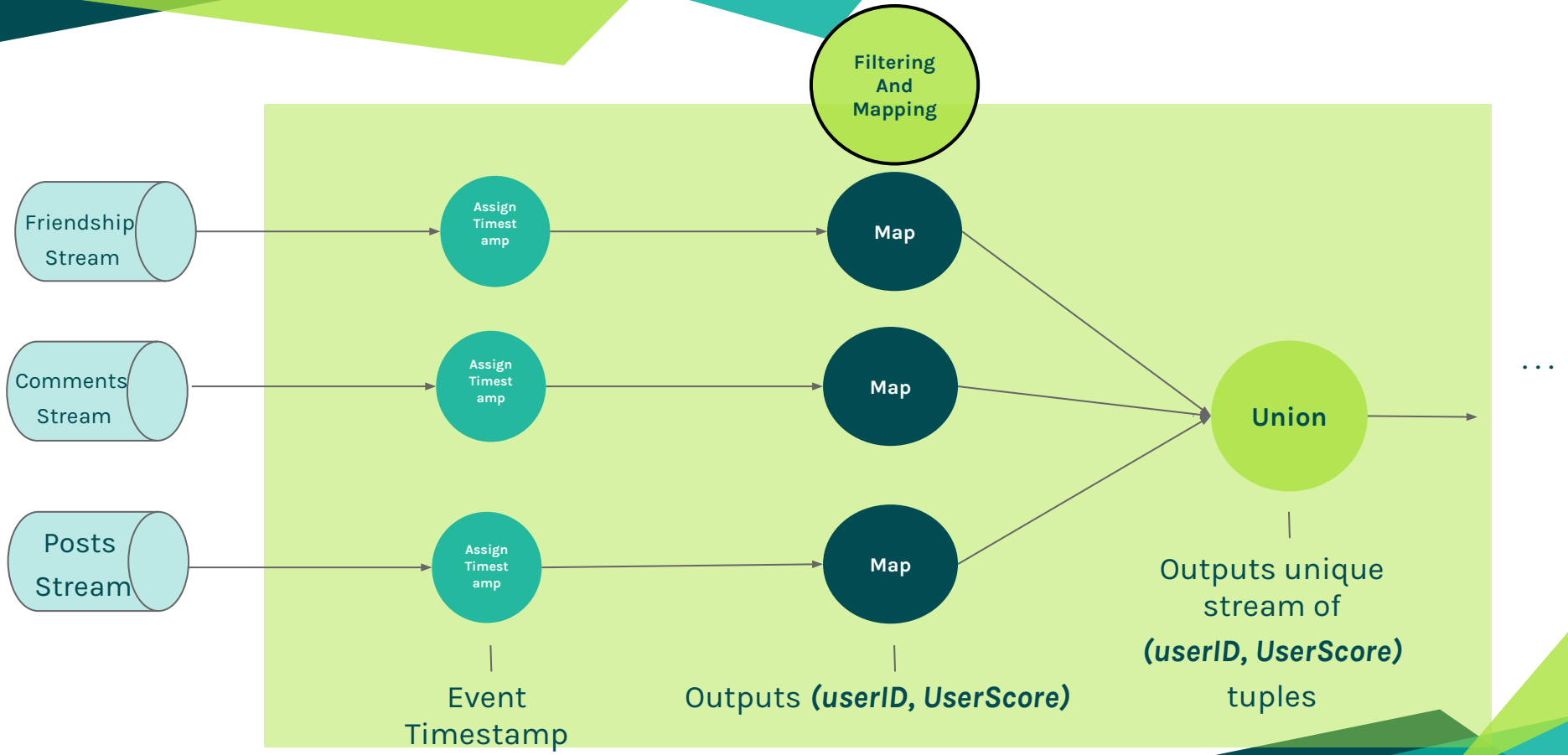
Flink Ranking Queries Overview



Query 2 - Flink



Query 3 - Flink

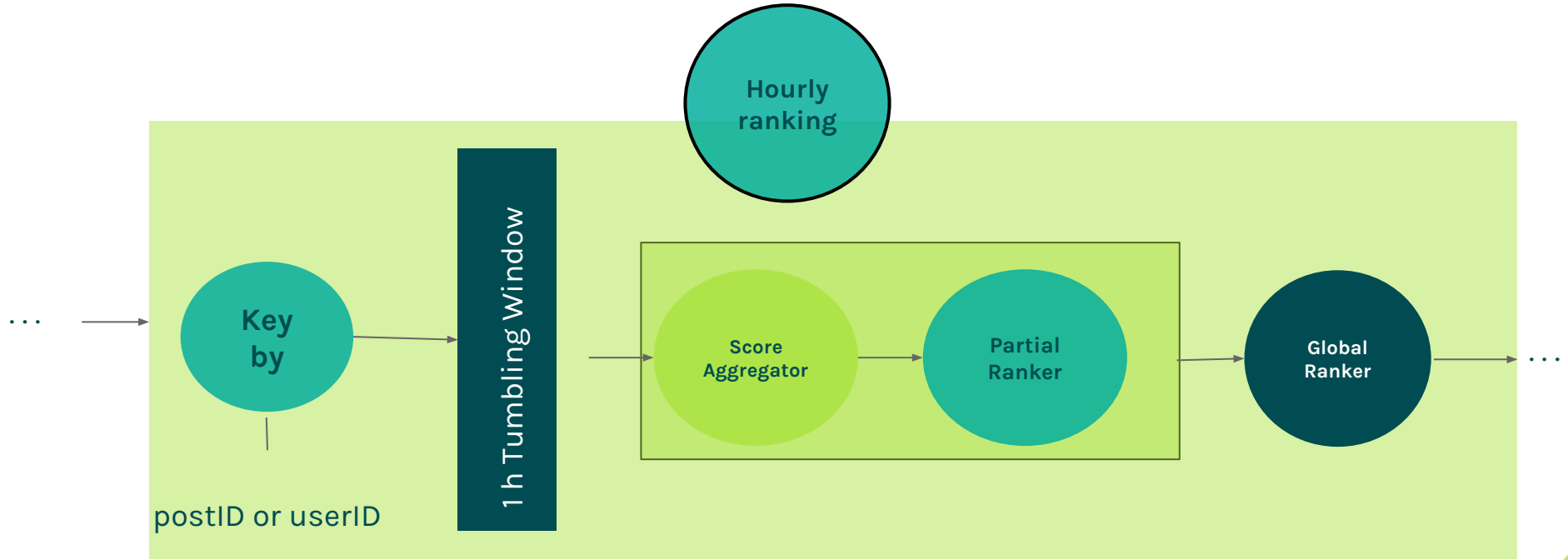


Ranking Operator in Flink

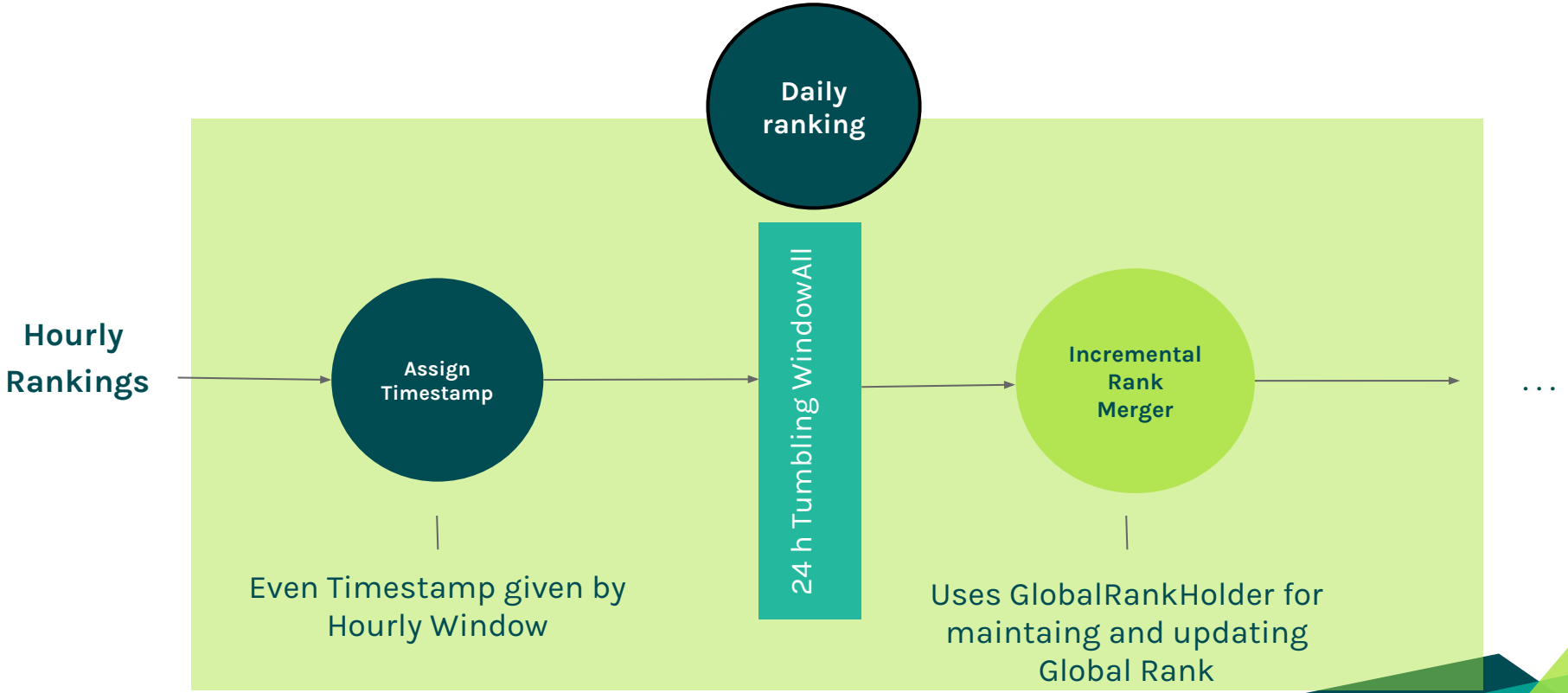
Same general approach for computing ranking for different window types and sizes:

```
<keyed-stream>
  .window(<Tumbling or Sliding Event Time Window>)
  .aggregate(<Score Aggregator>, PartialRanker)
  .setParallelism(< gte 1 >)
  .process(GlobalRanker)
  .setParallelism(1)
```

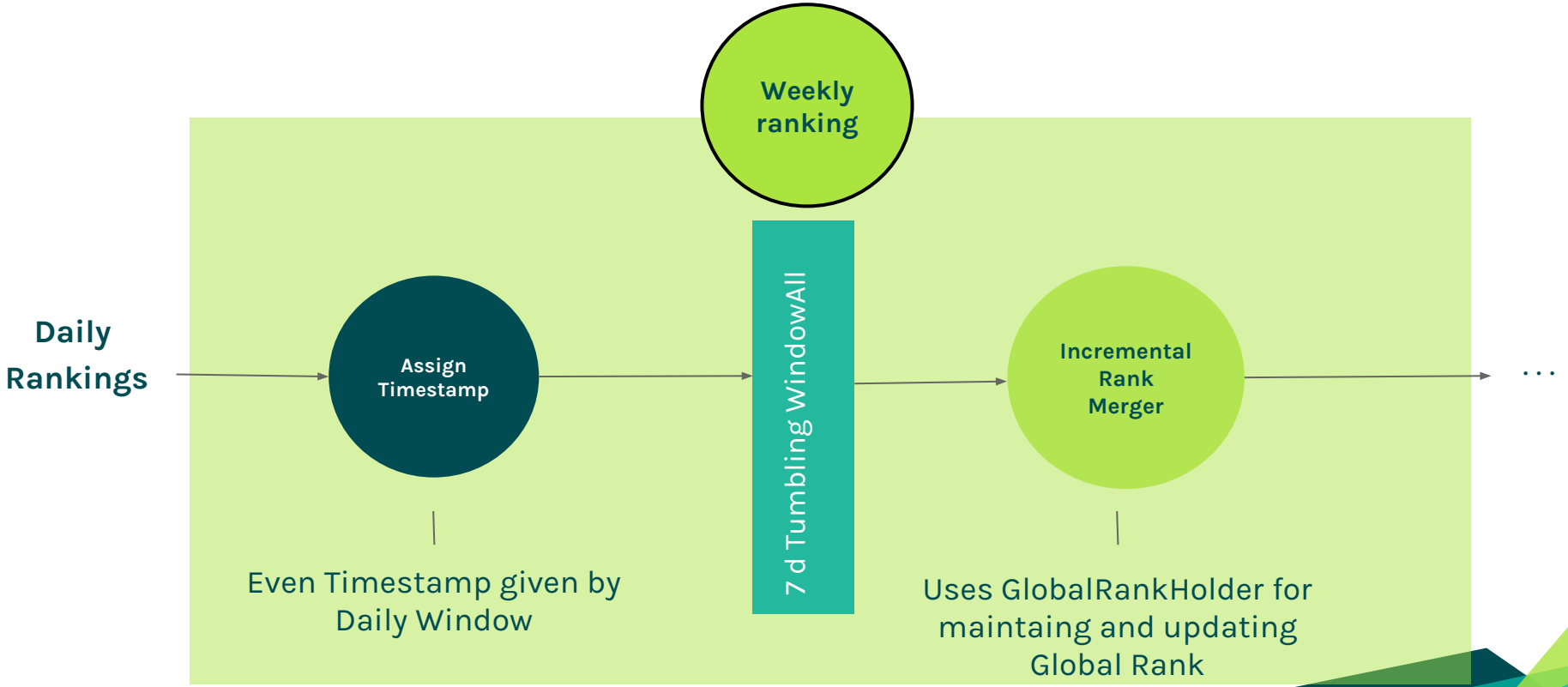
Query 2 & 3 - Flink



Query 2 & 3 - Flink



Query 2 & 3 - Flink



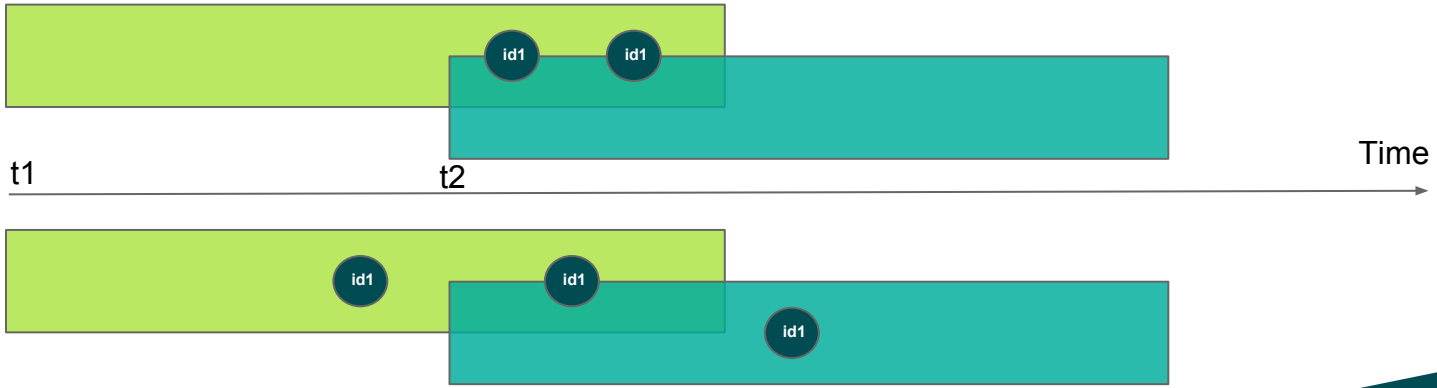
Sliding Windows Ranking Problems in Flink

Given rankins:

- R1 = {t1, { id1, score = 2} }
- R2 = {t2, {id1, score = 2}}

With t1 and t2 timestamp of two overlapping windows.

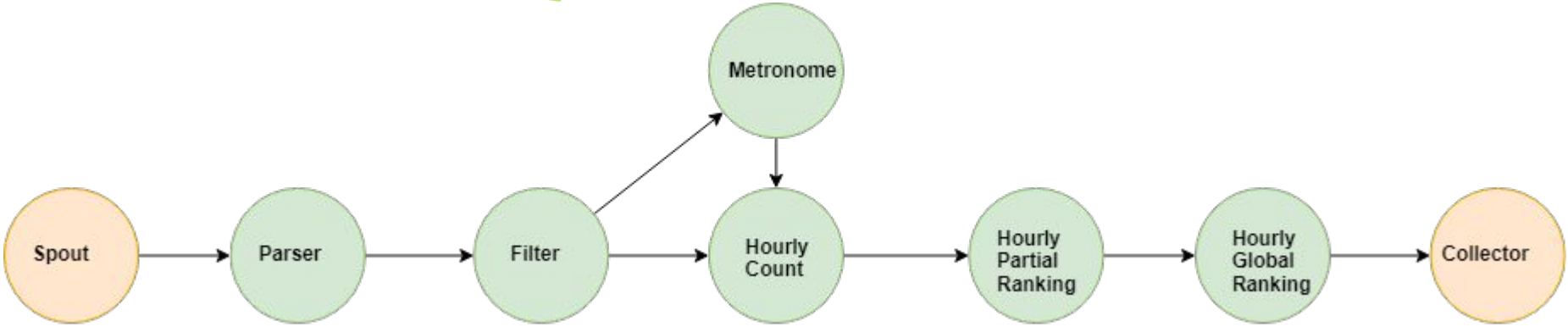
Multiple possible outcomes and can't know when to increment score and by how much.



Ranking Query With Sliding Windows in Flink

- Sliding windows for computing daily and weekly rankings given hourly rankings give inaccurate results
- Can’t know what happened exactly in two overlapping windows
- If we take non-overlapping windows, can lose what happened in the transition from one to another
- **Solution:** replicate input stream and apply general approach with different sliding window sizes
- *Cons*:
  - Same data replicated for many sliding windows
  - Throughput decreases and latency increases

Query 2 Storm



Parse Avro record

Discard comments to comment

1h Tumbling window

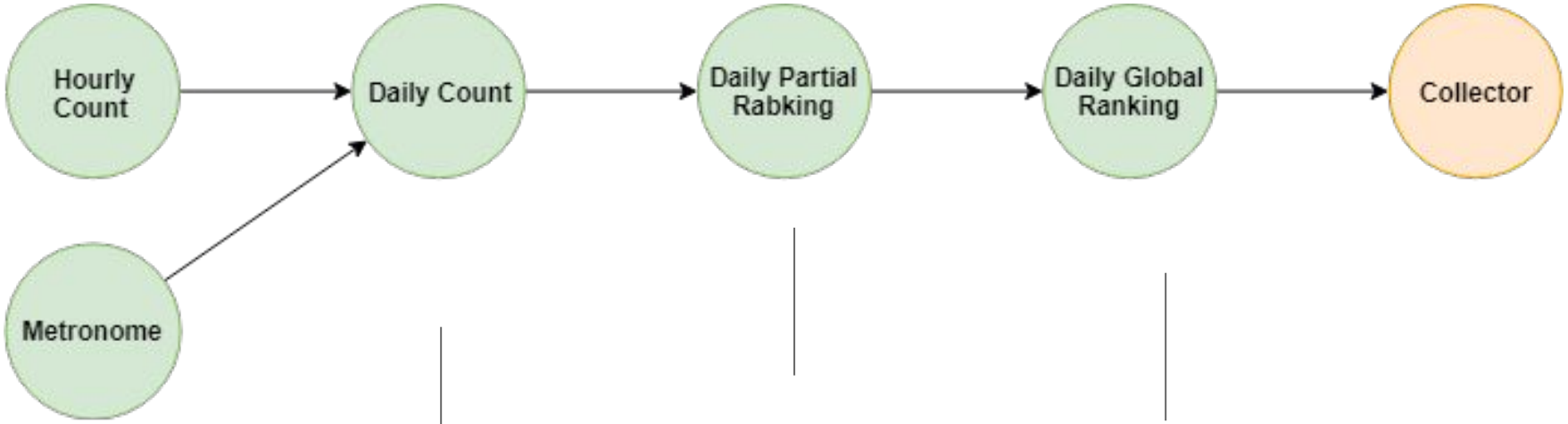
Compute partial Ranking

Merge partial ranking

Query 2 Storm

- **Spout**: Read record from Kafka topic
- **Filter**: Discard comments to comment
- **Metronome** emits a special tuple to beat time based on the record timestamp (event time). It affects the emission rate of the windows (the slide interval)
- **WindowCount**: Keep one windowed count for each post
- **PartialRanking**: Compute ranking on a group of post
- **GlobalRanking**: Merge partial ranking to get final top-K
- **Collector**: Write results in a Kafka topic

Query 2 Storm



Aggregate daily count in a 24h tumbling window

Compute partial ranking for group of post

Merge partial ranking

Query 2 Storm

- Tuples by the Filter are distributed by the *post_id* field to the WindowCount
  - **Fields grouping**
- The **WindowCount** and **PartialRanking** bolts executes as many tasks.
- The **Metronome** tuples must be forwarded to all the windowed bolts
  - **All grouping**: The stream is replicated across all the bolt's task

4.

Deployment

Local and Cloud deployment

Local Deploy

- Every system component is built as a Docker Image
- Made from existing images
- Allows running pseudo-distributed cluster (Flink, Storm and Zookeeper)
- All components run on same network

Cloud Deploy

Google Cloud Platform

Google Cloud Platform was used as the deploy environment.

In particular services offered by the Kubernetes Engine.

Kubernetes

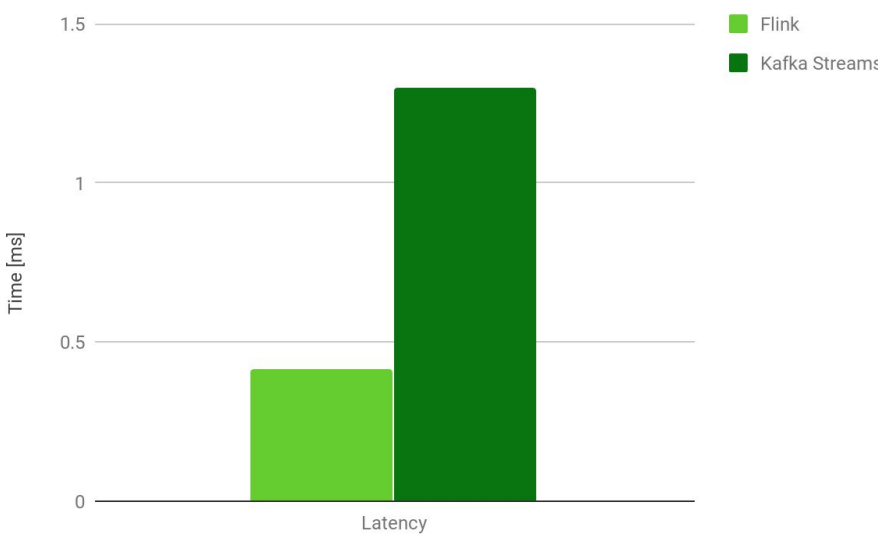
- Used to deploy System components
  using Docker container on a
  3-node cluster with each 2 vCPU and 7.5 GB of memory
- Flink, Kafka and Zookeeper as StatefulSets managed with
  parallel policy and exposed by Headless (intra-cluster)
  and “Normal” (inter-cluster) Services
- Storm workers managed by a ReplicationController while other components (Nimbus, UI) deployed as simple Pods
- Kubernetes Deployment with OpenJDK for simulating Kafka producers

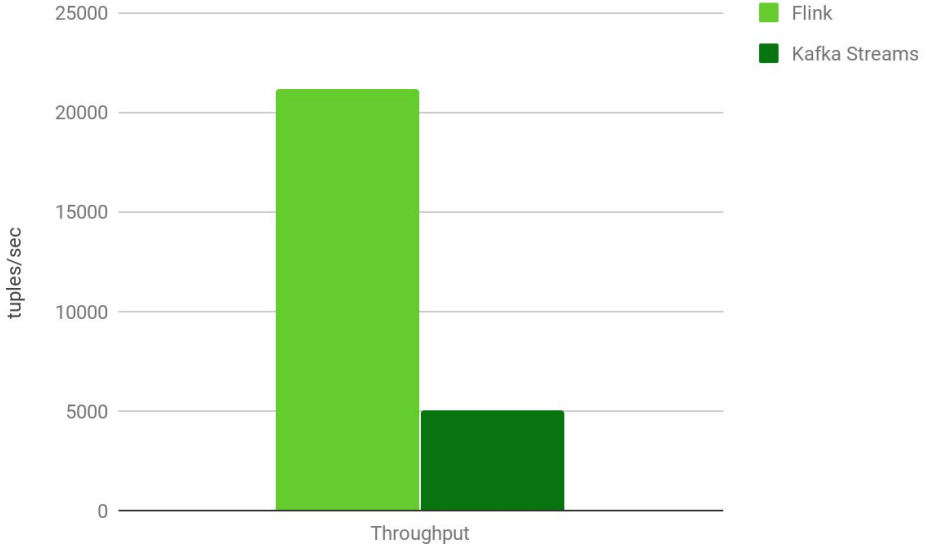
5.

Performance Comparison

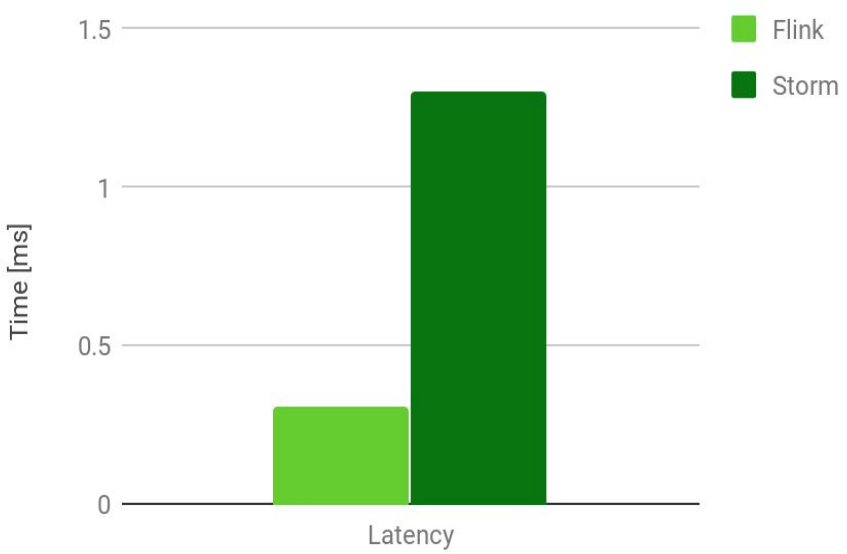
Different Framework Comparison

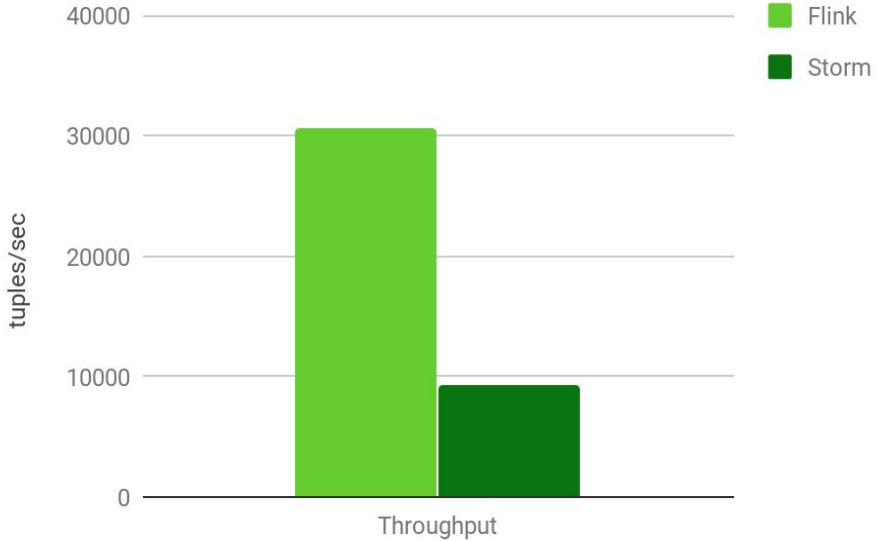
Query 1 performances



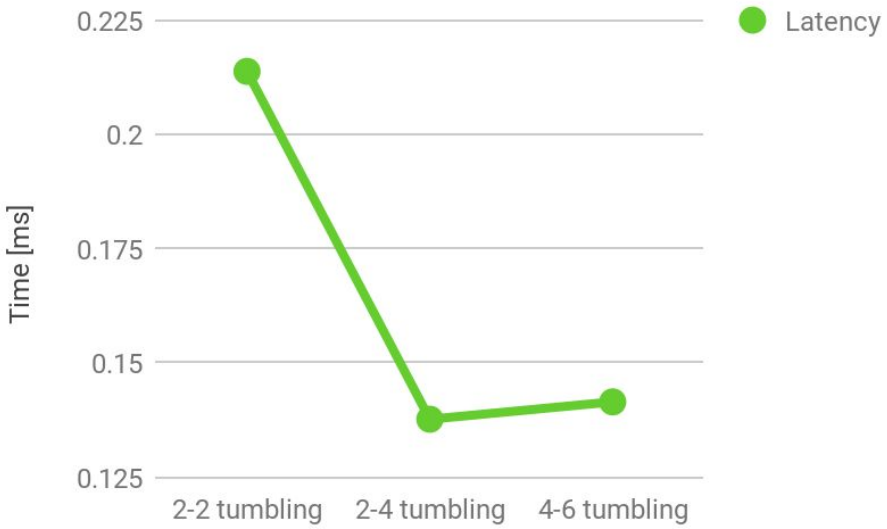


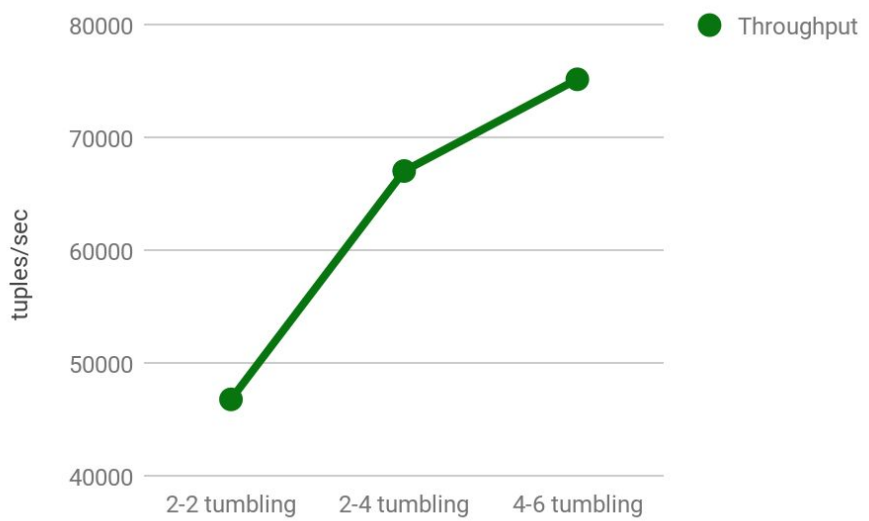
Query 2 performances





Query 3 Flink performance





Thank you for listening!