Approximate queries and graph streams on Flink

Theodore Vasiloudis, Swedish Institute of Technology

tvas@kth.se
@thvasilo

Hello, I’m Theo!

Motivation

- We want analytics that provide us with immediate answers
  - *Hadoop: Yesterday's insights, tomorrow!*
- Data never stops!
  - Infinite memory?
- Solution: windows and approximations
  - Windows give us a snapshot of the world
  - Approximations allow us to continuously measure the world
  - First part of presentation is about approximate streaming algorithms, second focuses on using windows for graph analytics

Approximate Queries on Flink

Work by Tobias Lindener, KTH
https://github.com/tlindener/ApproximateQueries/

End Goal: Approximate SQL queries on Flink

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2  SECONDS
```

Queries with Time Bounds

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

Queries with Error Bounds

Source: BlinkDB

First step: Sketches for standing queries

Web Site Logs

Time	User ID	Site	Time Spent Sec	Items Viewed
9:00 AM	U1	Apps	59	5
9:30 AM	U2	Apps	179	15
10:00 AM	U3	Music	29	3
1:00 PM	U1	Music	89	10
Billions of rows …				

Financial Transactions System Log

Time	User ID	Site	Purchased	Revenue
9:00 AM	U1	Apps	FaceTune	$3.99
9:30 AM	U2	Apps	Minecraft	$6.99
10:00 AM	U3	Music	Purple Rain	$1.29
Billions of rows …				

- Num. unique users who visited both Apps and Music over the last hour
- Median and 95%ile Time Spent over the last day
- Most frequently purchased songs

Source: Yahoo Datasketches

Sketch Algorithms for Massive Data

- Research area since the 70s (Knuth, Indyk, Flajolet)
- Goal: Efficient (compute+memory) algorithms for “simple” tasks
  - Frequent items
  - Set cardinality
  - Moments (mean, median, variance etc.)
  - Quantiles and histograms
  - Graph algorithms (triangle count, connected components)
  - Nearest neighbors
- We use them as building blocks for more complex algorithms
  - Databases (joins)
  - Machine learning (decision trees)

Yahoo Datasketches

- Highly optimized sketch library
- Apache Licensed
- Available for Pig, Hive

Big Win #1: Size of the Query Process

BIG WIN #1: Small Query Space!

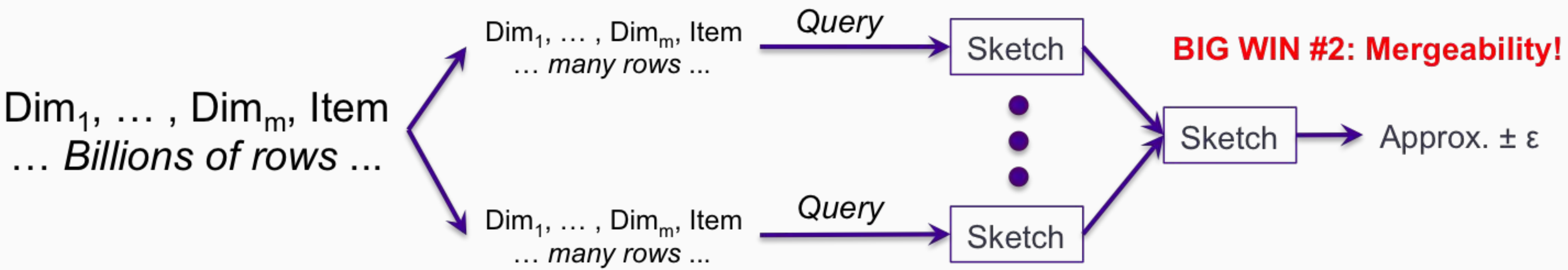
$O(k)$ size: ~ KBytes

$\text{Dim}_1, \ldots, \text{Dim}_m$, Item
… *Billions of rows* ...

Query → Sketch → Approx. ± ε

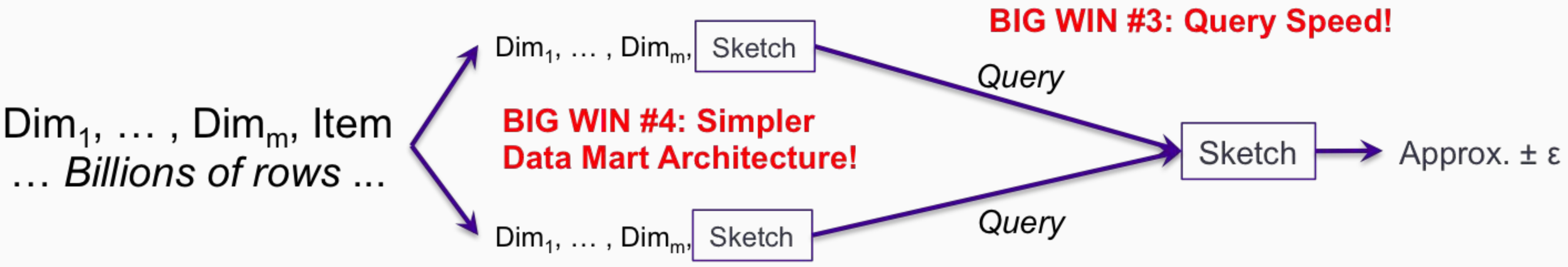
Source: Yahoo Datasketches

Big Win #2: Sketch Mergeability Enables Parallel Processing



Source: Yahoo Datasketches

Big Wins #3 & 4: Query Speed, Architecture Simplicity

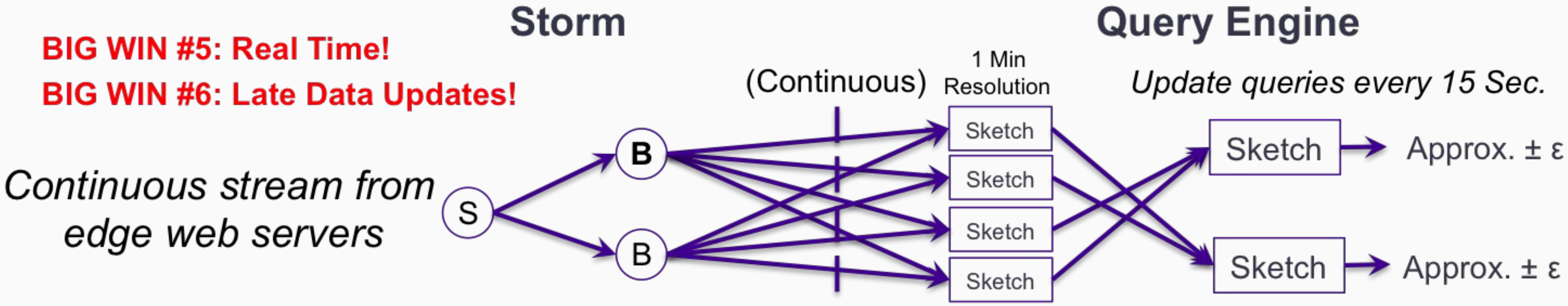


Source: Yahoo Datasketches

Big Wins #5 & 6: Real Time, Late Data Updates

BIG WIN #5: Real Time!

BIG WIN #6: Late Data Updates!



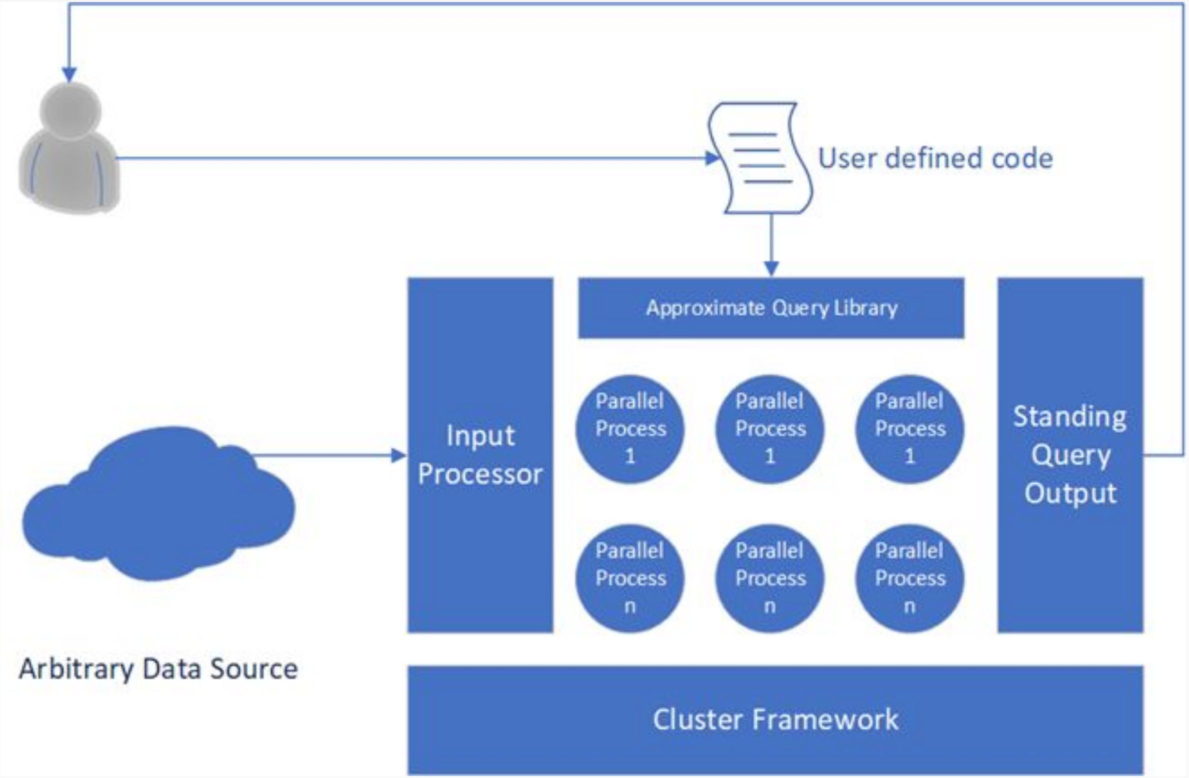
Source: Yahoo Datasketches

Why Flink?

- From PR sketches-core#81: *Sketch now implements Serializable* (closed)
- Lee Rhodes (package author) writes:

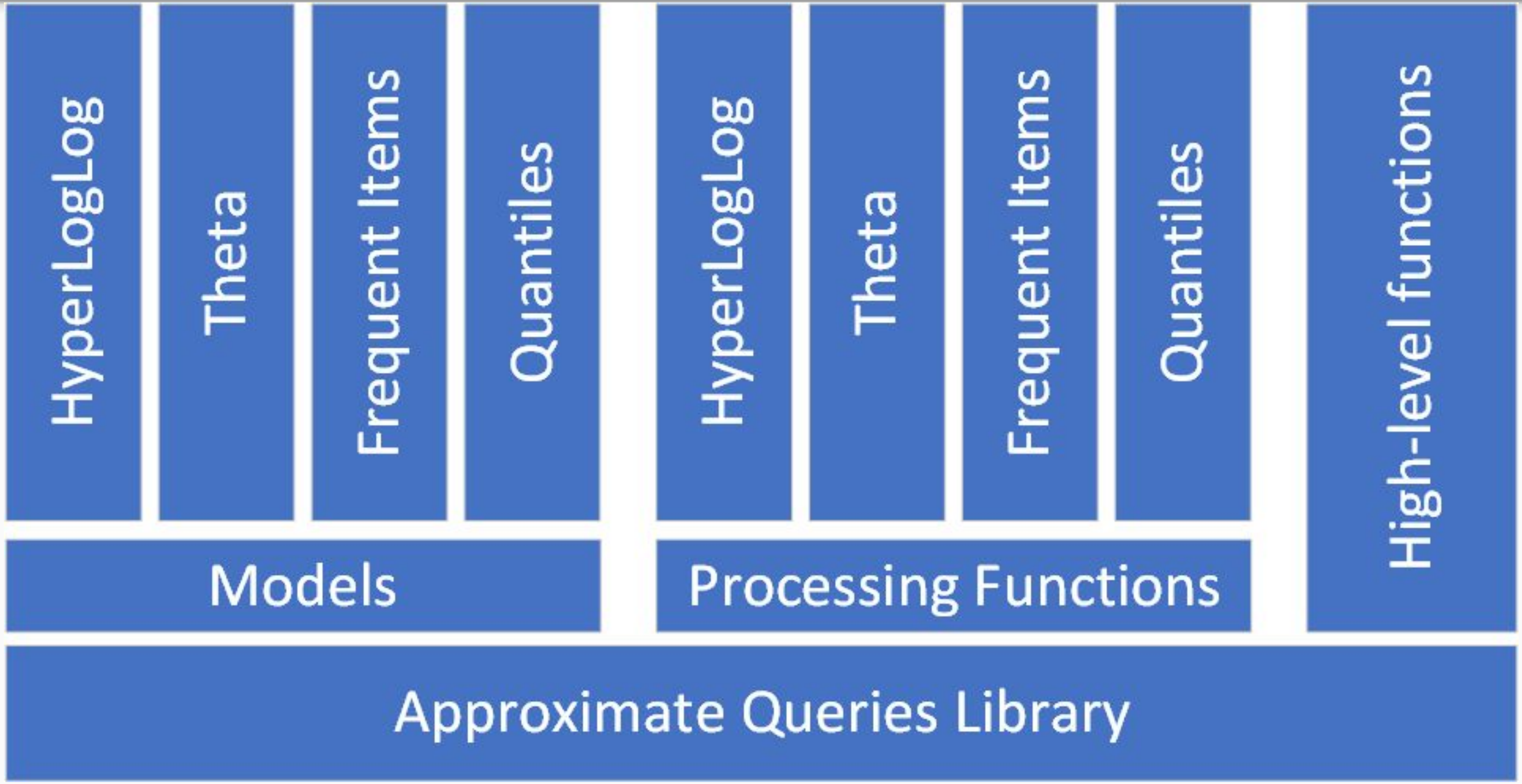
> *Sketches are streaming algorithms and are* ***stateful by design***. *Attempting to force them into a stateless paradigm will result in orders-of-magnitude poorer performance. It is like pounding a square peg into a round hole.*

Library Design



Source: Tobias Lindender

Library Design



Source: Tobias Lindender

Query API

- Cardinality Estimation Queries

```java
public static <T> DataStream<HllSketchAggregation> runContinuousHll(DataStream<T> input, KeySelector keySelector, KeySelector valueSelector, int emitMin)

public static <T> DataStream<ThetaSketchAggregation> runContinuousTheta(DataStream<T> input, KeySelector keySelector, KeySelector valueSelector, int emitMin)
```

Query API

- Frequent Items & Quantiles

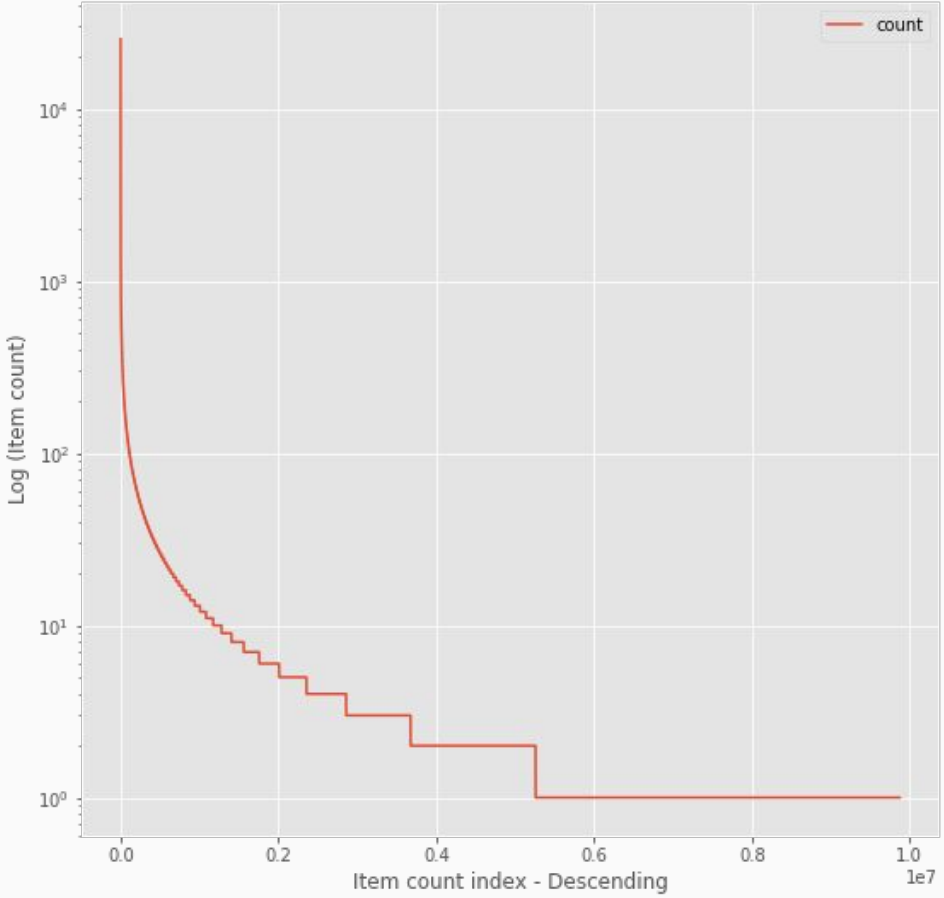
```java
public static <T> DataStream<TopNQueryResult> continuousFrequentItems(DataStream<T> inputStream, KeySelector valueSelector, int maxItems, int emitMin)

public static <T> DataStream<QuantileQueryResult> continuousQuantiles(DataStream<T> inputStream, KeySelector valueSelector)
```

Experiments

Datasets

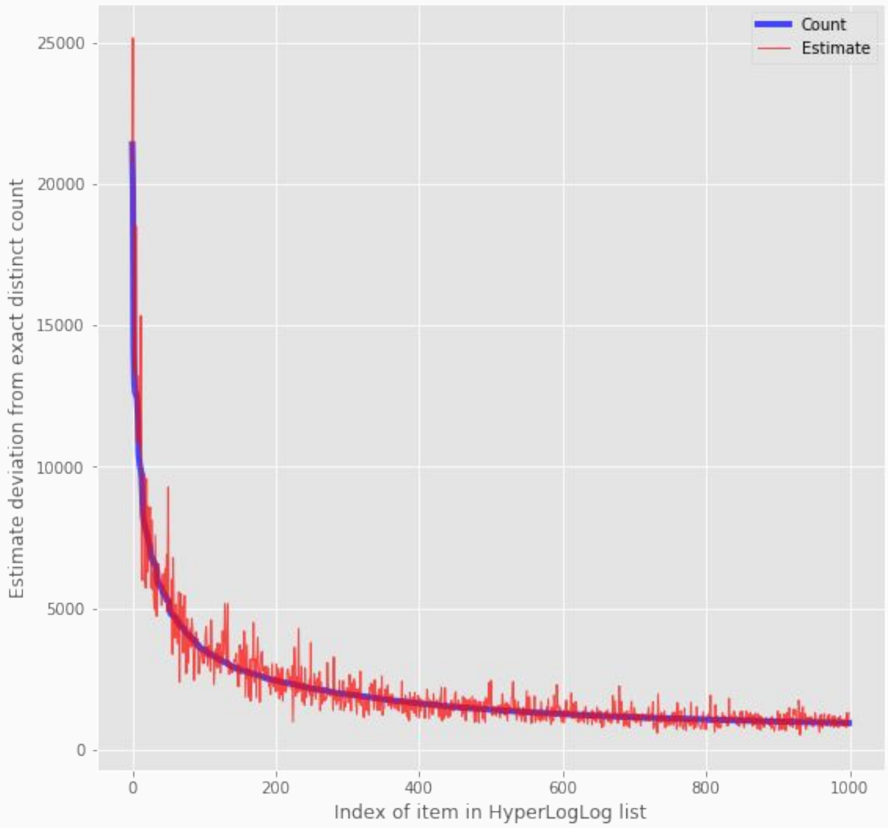
- Amazon Reviews
  - ~84M reviews of Amazon products
  - User, Item, Rating, Timestamp
- WikiTrace
  - Wikipedia access logs
  - ~80M requests, ~7M URLs



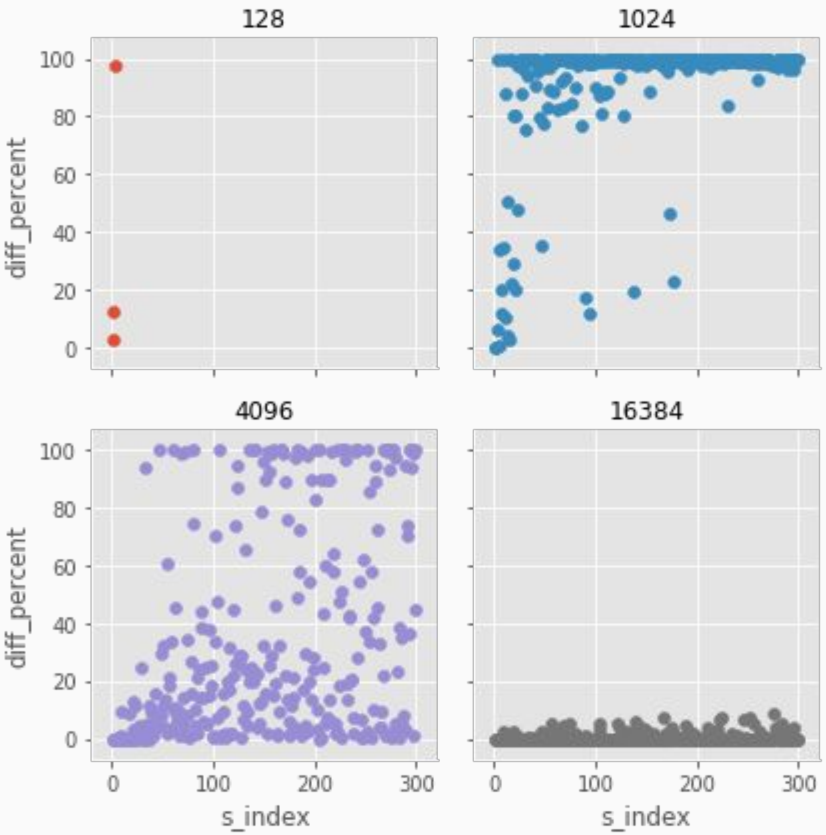
Amazon Reviews per product

Runtime and memory consumption

	Runtime	**Memory**
Exact	~700s	~11GB
Sketch	~90s	~2GB



Wikitraces: Accuracy of HyperLogLog sketch compared to exact



Amazon: Accuracy of frequent item sketch for different map sizes (lower is better)

Summary

- Built library to allow to easily run approximate queries on top of Flink
- End goal is to enable approximate streaming queries with SQL syntax

Gelly Stream:
Streaming Graph Processing

Paris Carbone, KTH, Flink Committer
Vasiliki Kalavri, ETH, Flink PMC

Motivation

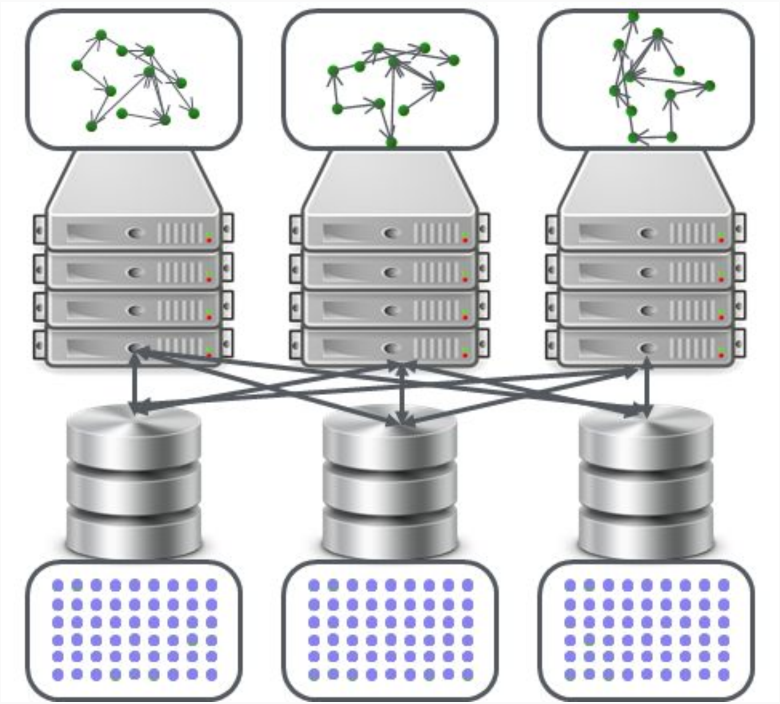
- Graphs are powerful representations of many interactions
  - Social networks
  - Purchases
  - Media views
- Again, data are massive, constantly arriving, and unbounded
- So we need distributed streaming graph processing

Slides by Paris Carbone

Previous work

- Graph snapshots
  - Pregel, Giraph, GraphX
- Graph streams
  - Summaries, approximate algorithms
  - Semi-streaming (disk)

Load-Compute-Store

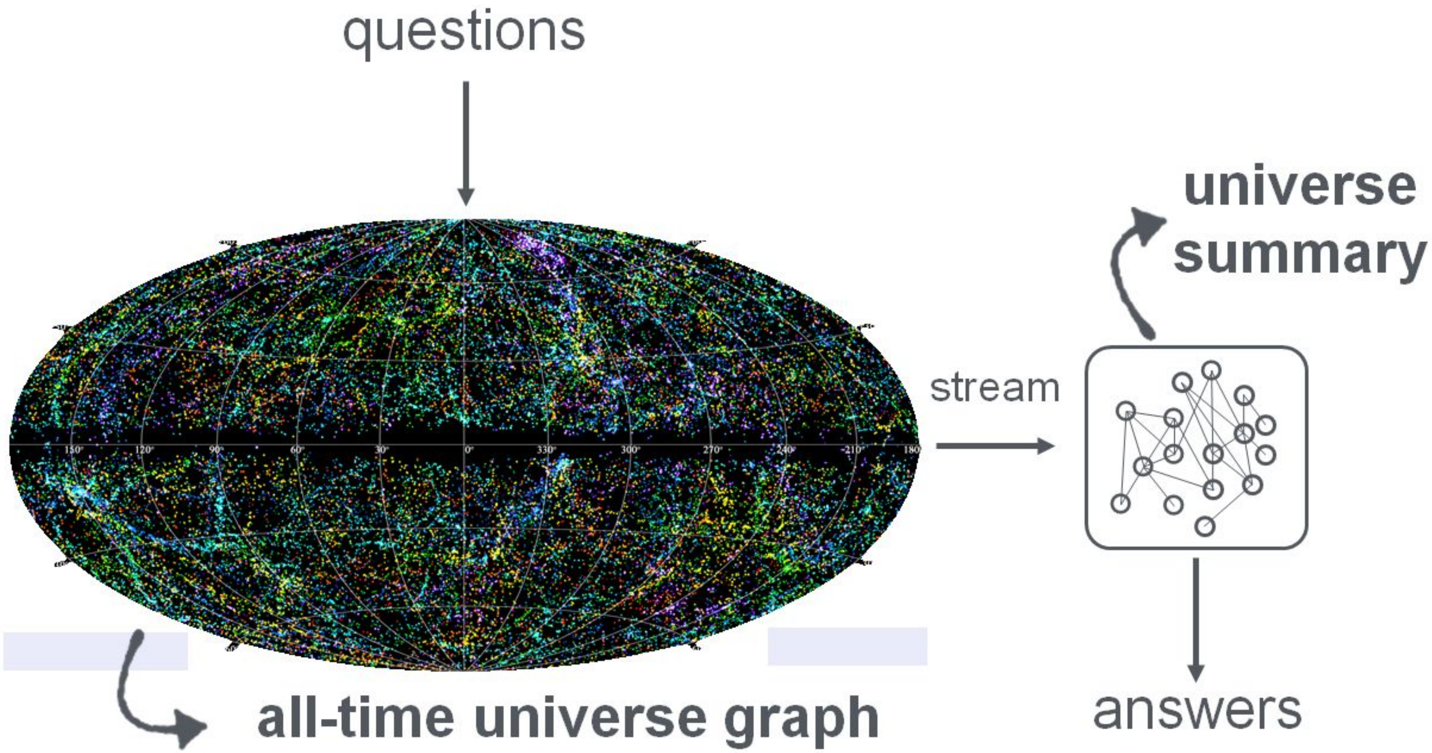


1. **Load** snapshot to memory
2. **Compute** state/superstep
3. **Store** updated graph state
4. Goto 1

Load-Compute-Store

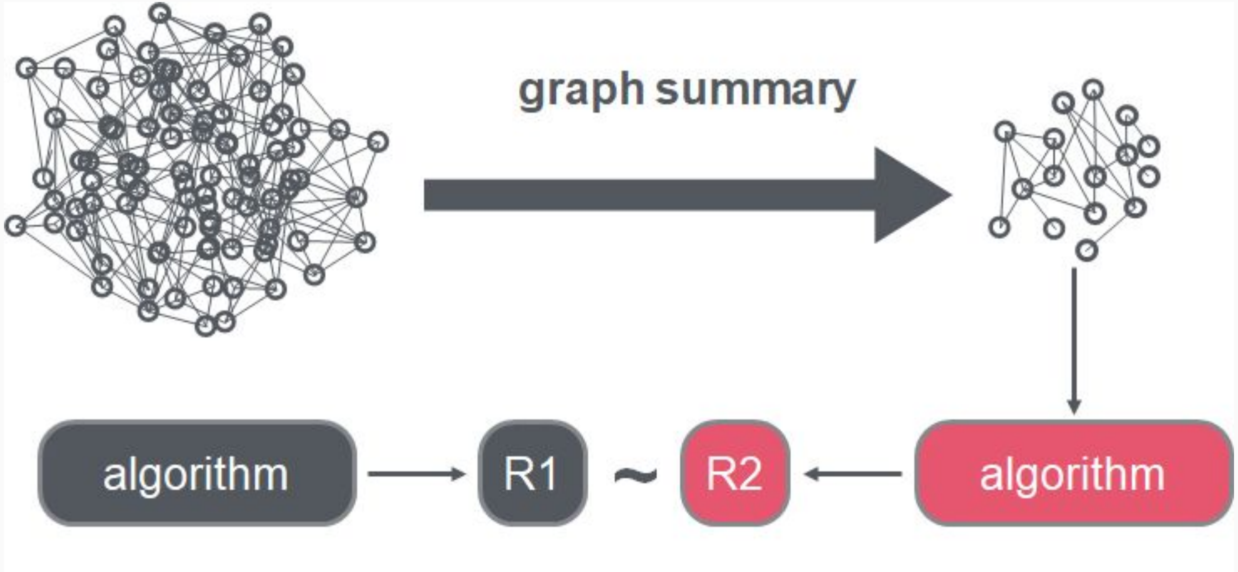
- Wide adoption: Pregel, Graphlab, GraphX
- Interfaces well with existing batch systems
- But has model issues:
  - Unnecessary **latency** for all graph measures.
  - Inefficient for incorporating **updates**
  - Sensitive to the **partitioning** method
  - **Re-computation** across snapshots

Graph Summaries: Intuition



Graph Summary Flavours

- **Spanners** : distance estimation
- **Sparsifiers** : cut estimation
- **Sketches** : homomorphic properties



Engineering benefits of stream processing

- **Low latency** and **high-throughput**
- **Long-running** processes can now pipeline computation
- Production Ready: end-to-end **fault tolerance**

Realizations brought by stream processing

1. Duality of **input** data **+** computational **state**
2. **Out-of-order** processing

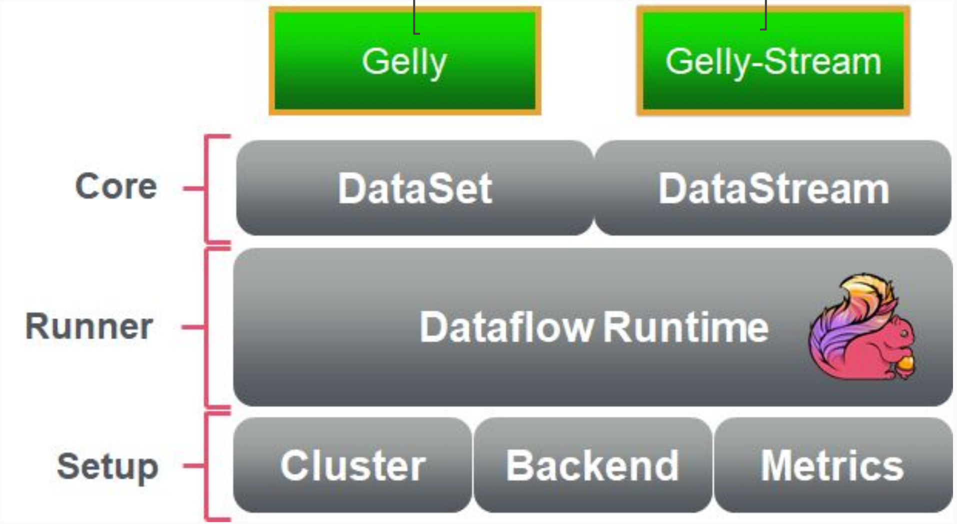
Exploiting stream processing for graphs

1. Duality of **input** data **+** computational **state**
   a. Define evolving graph properties
   b. Graph updates (input) ⇔ properties (state)
2. **Out-of-order** processing
   a. Pre-compute blocking graph operations
   b. Multiplex processing per snapshot or window

Gelly-stream overview

- ➤Static Graphs
- ➤Multi-Pass Algorithms
- ➤Single Answer

- ➤Dynamic Graphs
- ➤Single-Pass Properties/Summaries
- ➤Multi-Pass on Snapshots



Gelly-Stream Data Types

- <u>**EdgeStream**</u> **-> Non-Blocking / Single-Pass Computation**
  - A distributed *data stream* consisting of **graph edge additions**.
  - Edges can contain **state** (e.g. weights).
  - Supports **property** streams, **transformations** and **aggregations**.
- <u>**SnapshotStream**</u> **-> Blocking / Multi-Pass Computation**
  - Each Snapshot is bounded~ i.e., static graph window.
  - It enables **neighborhood aggregations, iterations (e.g., BSP)**

EdgeStream Operations

Property Streams

EdgeStream → DataStream

```
.getEdges()
.getVertices()
.numberOfVertices()
.numberOfEdges()
.getDegrees()
.inDegrees()
.outDegrees()
```

Transformation Streams

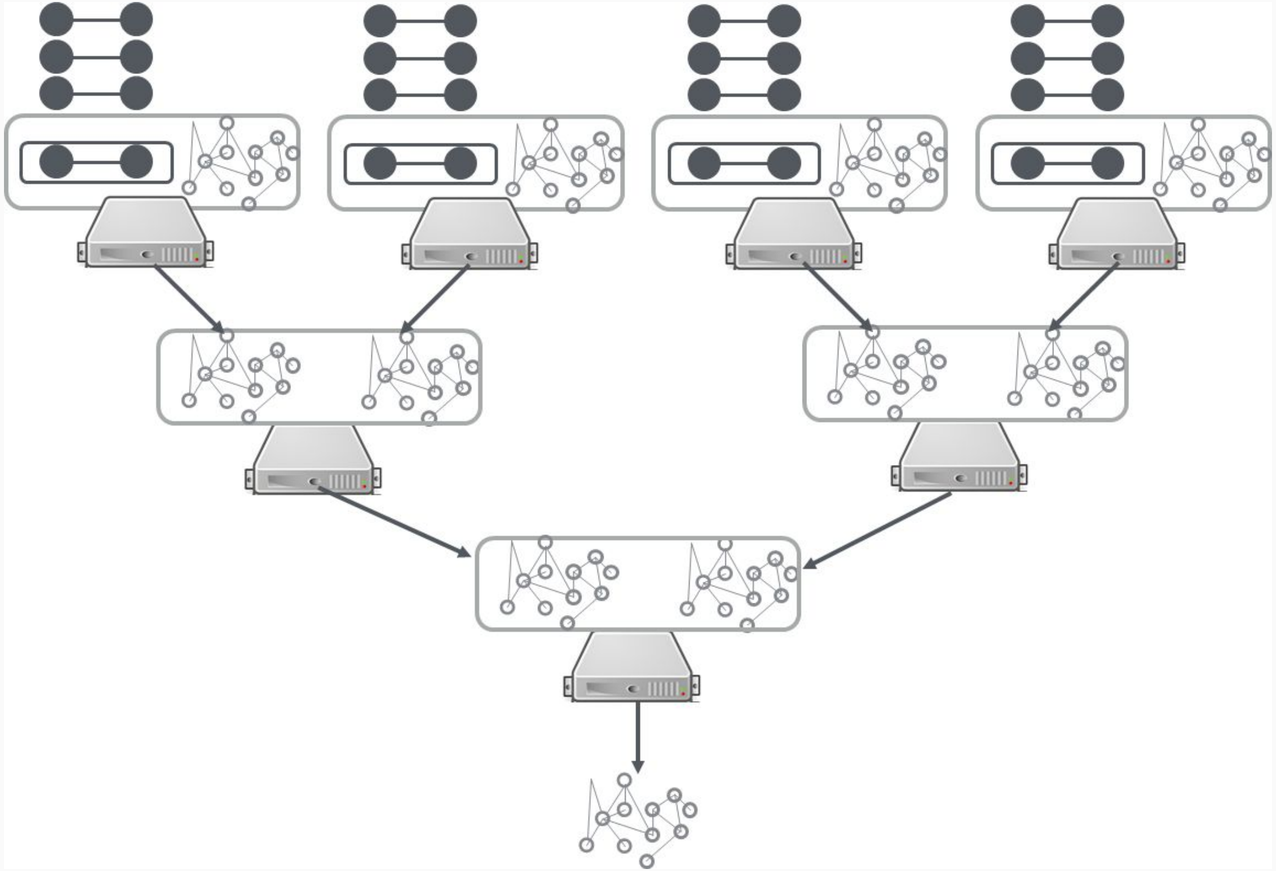
EdgeStream → EdgeStream

```
.mapEdges();
.distinct();
.filterVertices();
.filterEdges();
.reverse();
.undirected();
.union();
```

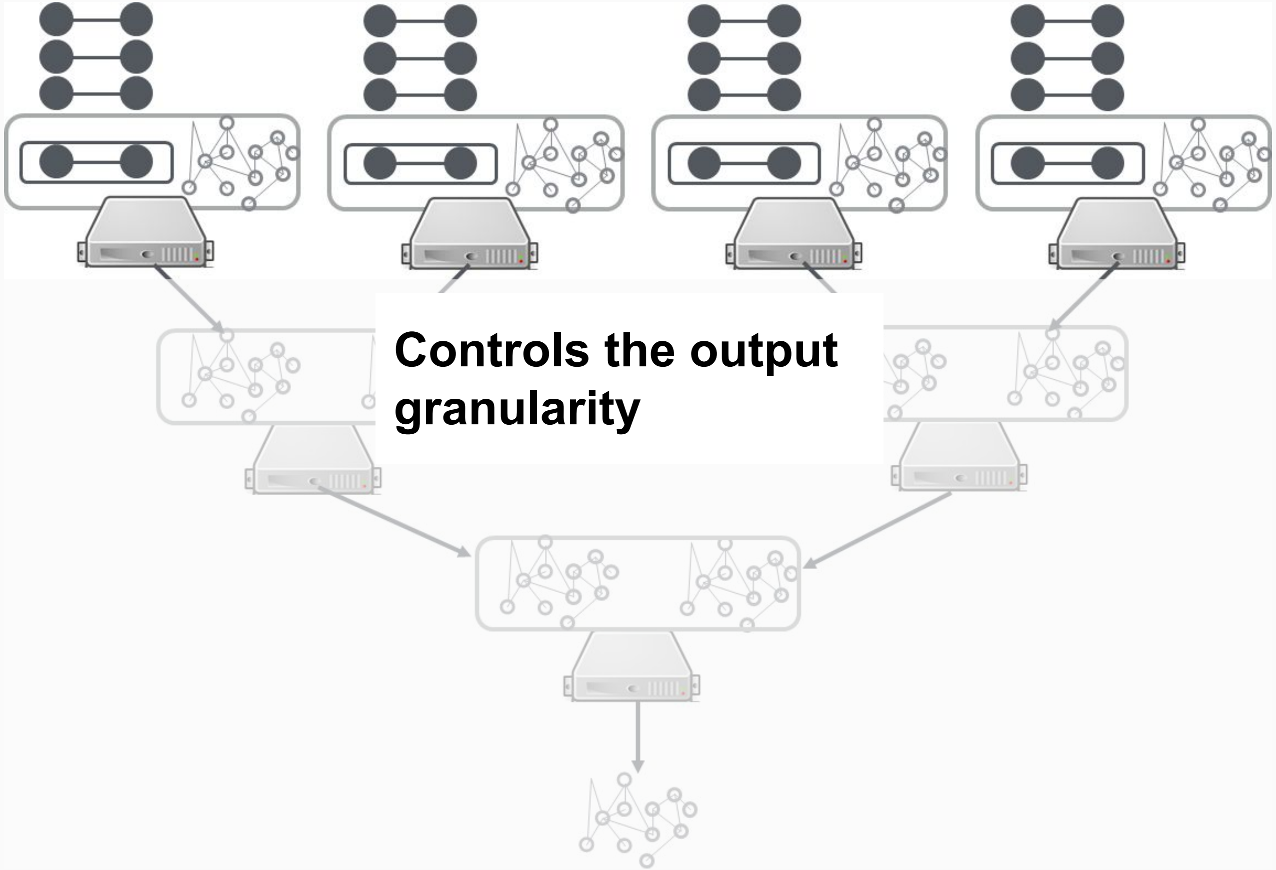
EdgeStream Summaries

```
edgeStream.aggregate
(new Summary(window, fold, combine, lower))
```

```
edgeStream.aggregate
(new Summary(window, fold, combine, lower))
```

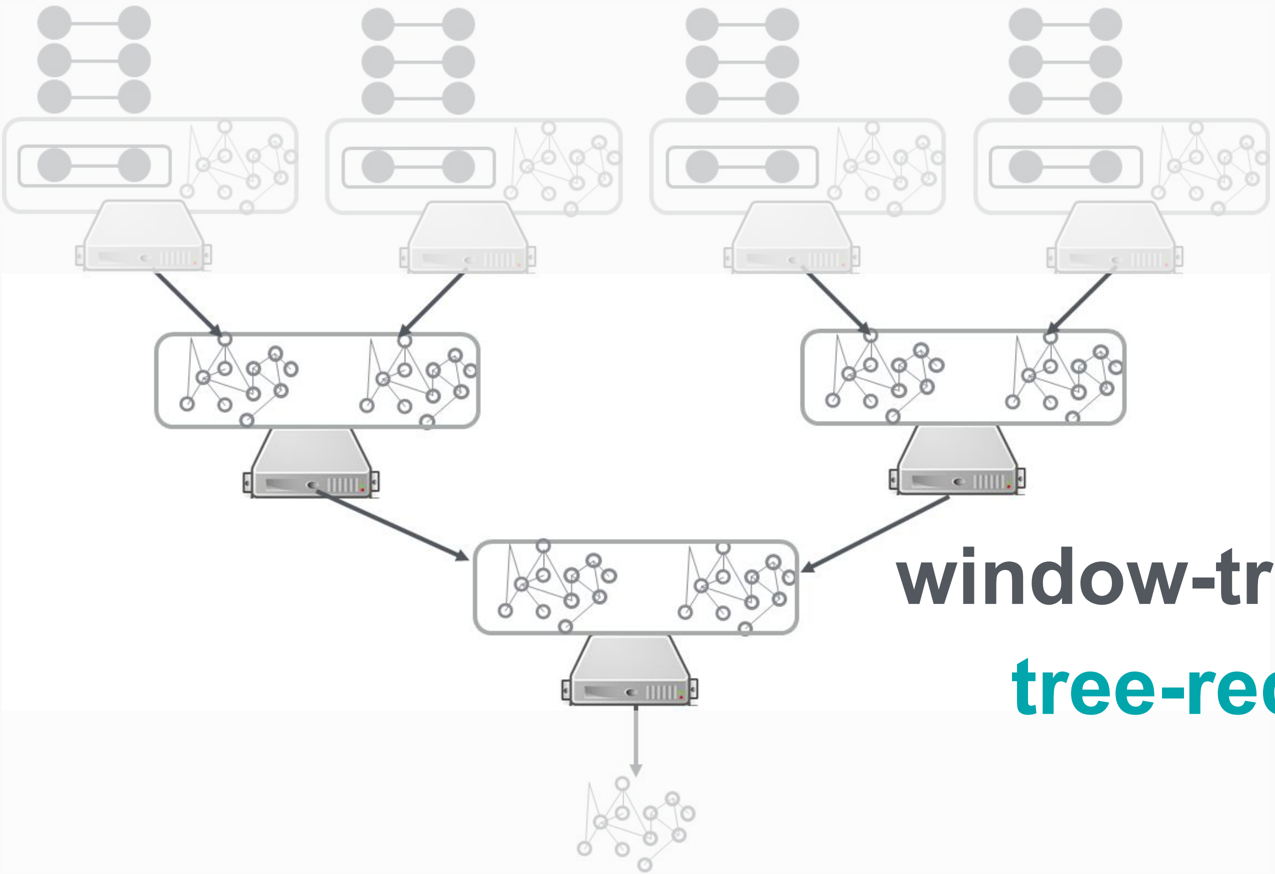


```
edgeStream.aggregate
(new Summary(window, fold, combine, lower))
```



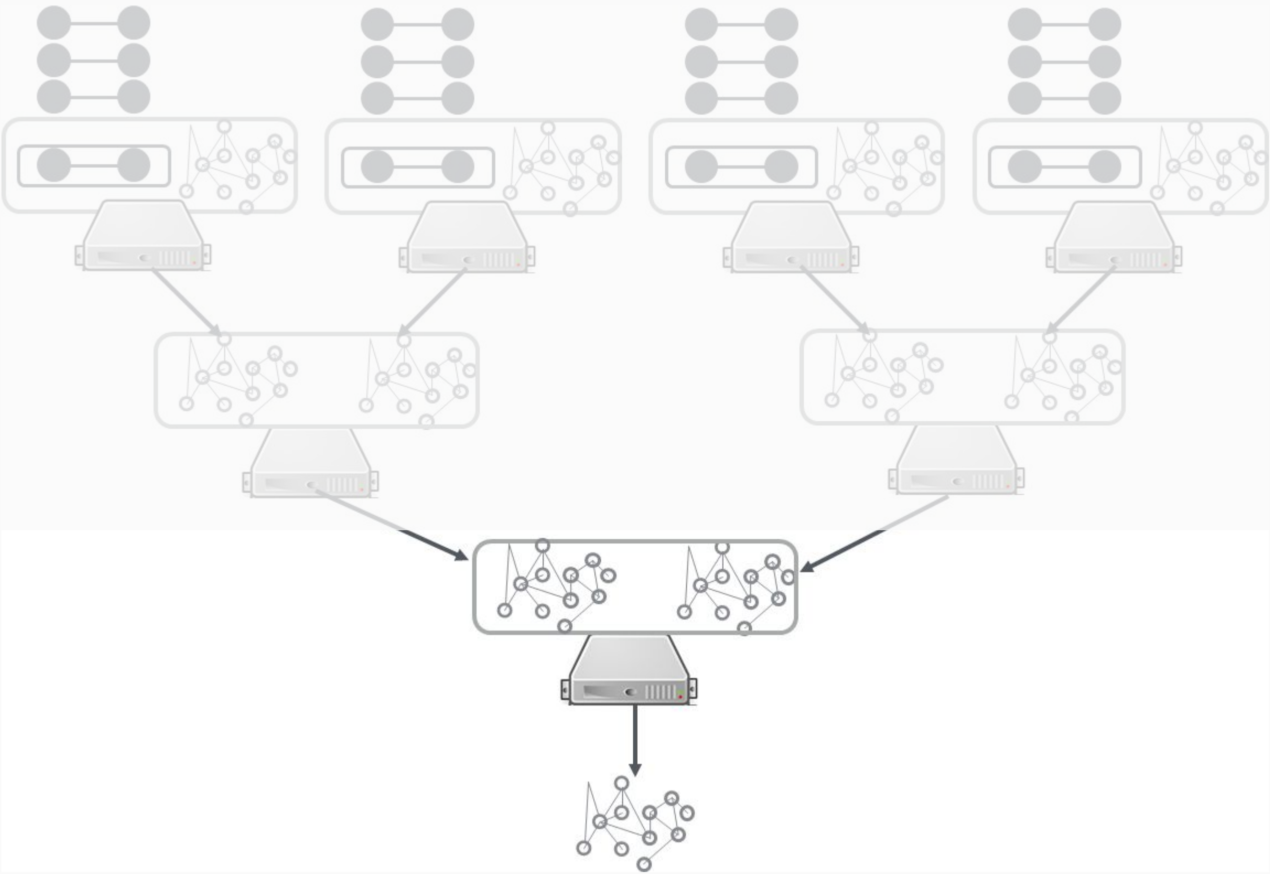
Controls the output granularity

```
edgeStream.aggregate
(new Summary(window, fold, combine, lower))
```



window-triggered
tree-reduce

```
edgeStream.aggregate
(new Summary(window, fold, combine, lower))
```



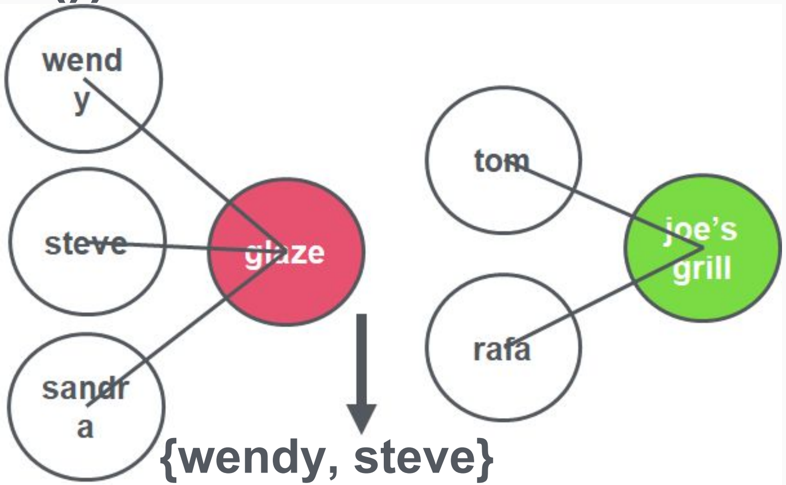
Provided Aggregates/Summaries

- Connected Components
- Bipartiteness Check (Binary)
- Window Triangle Count
- Rolling Triangle Count (Approximate)
- Continuous Degree Aggregate

Neighborhood Aggregation Example

```
edgeStream.filterVertices(DataScientists())
.slice(Time.of(10, MINUTE), EdgeDirection.IN)
.applyOnNeighbors(FindPairs())
```

wendy checked_in glaze
steve checked_in glaze
tom checked_in joe's_grill
sandra checked_in glaze
rafa checked_in joe's_grill



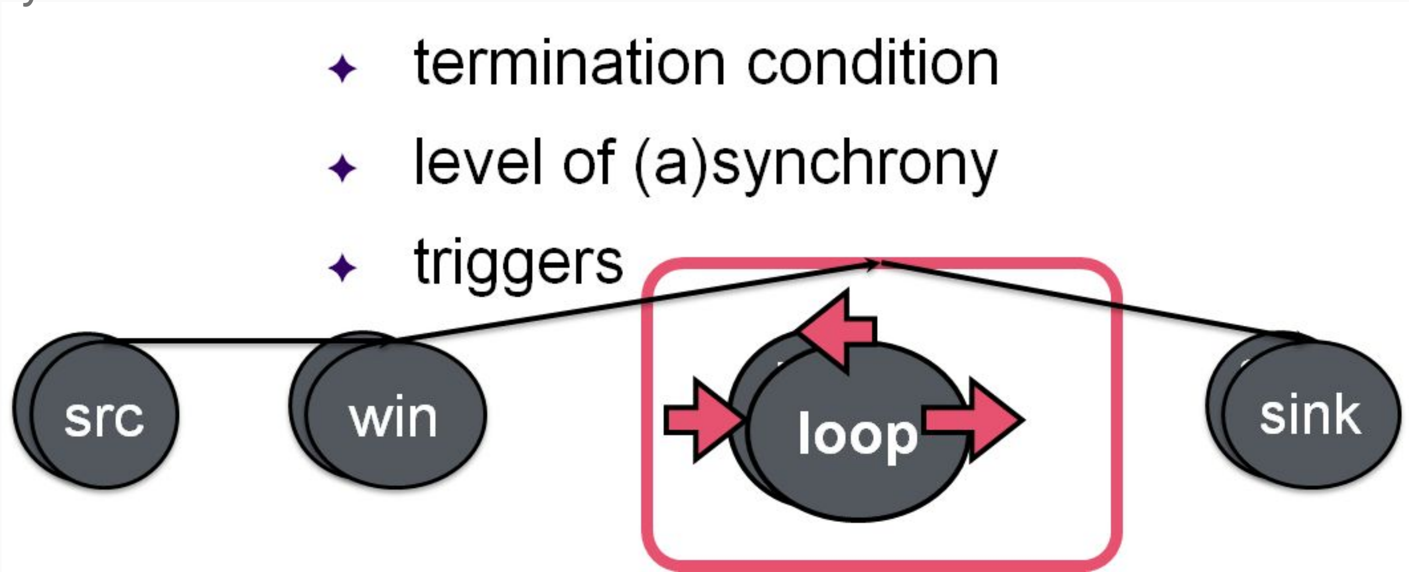
{wendy, steve}
{steve, sandra}
{wendy, sandra}
{tom, rafa}

Snapshot Iterations

- Most “deep” graph properties require **multiple passes**
- Sensitivity to **synchrony** during iterative processing depends on the algorithm and should be flexible (e.g., as in GraphLab).
- Avoiding scheduling delays (e.g. scheduling DataSet Iterations) is crucial for continuous processing.

Flink Stream Iterations

- A logical+physical loop redesign on Flink
- Introduces scoping and custom progress tracking
- Extends out-of-order dataflow processing
- Fully decentralised iterative execution



Take home message

- Streaming means unbounded
- Input <==> State
- Sketches and summaries let you deal with the unbounded nature of data using limited resources

Thank you!

tvas@kth.se
@thvasilo

References

- ApproximateQueries on Flink:
  https://github.com/tlindener/ApproximateQueries/
- Gelly Streaming
  https://github.com/vasia/gelly-streaming
- Yahoo Datasketches
  https://datasketches.github.io/
- Collection of links on streaming algorithms and sketches
  https://gist.github.com/debasishg/8172796