Skitter: A Distributed Stream Processing Framework with Pluggable Distribution Strategies

Mathijs Saey, Joeri De Koster, Wolfgang De Meuter

mathijs.saey@vub.be

Reactive Big Data Applications

- Respond to real-time data streams
- Volume of incoming data requires execution on a cluster

Running Example: Calculating Ad Conversion Rates

Distributed Stream Processing Frameworks (DSPFs)

APACHE STORM™

Flink

samza

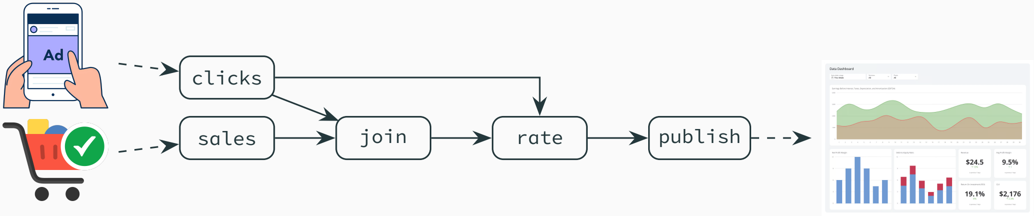
Google Dataflow

APACHE kafka®
A distributed streaming platform

APACHE Spark™

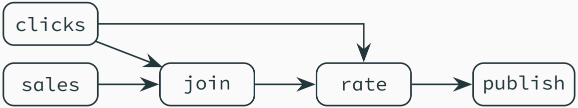
IBM

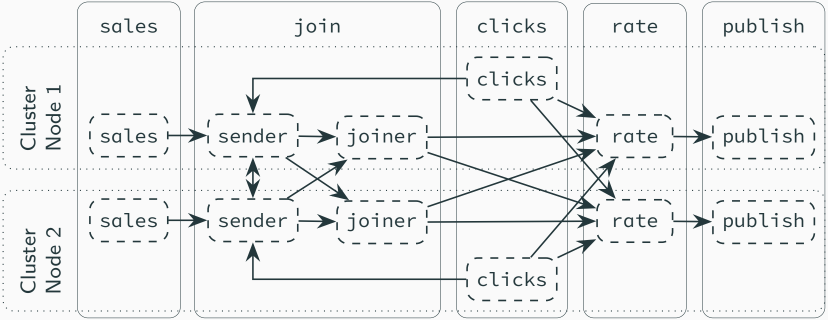
DSPFs: Programming Model



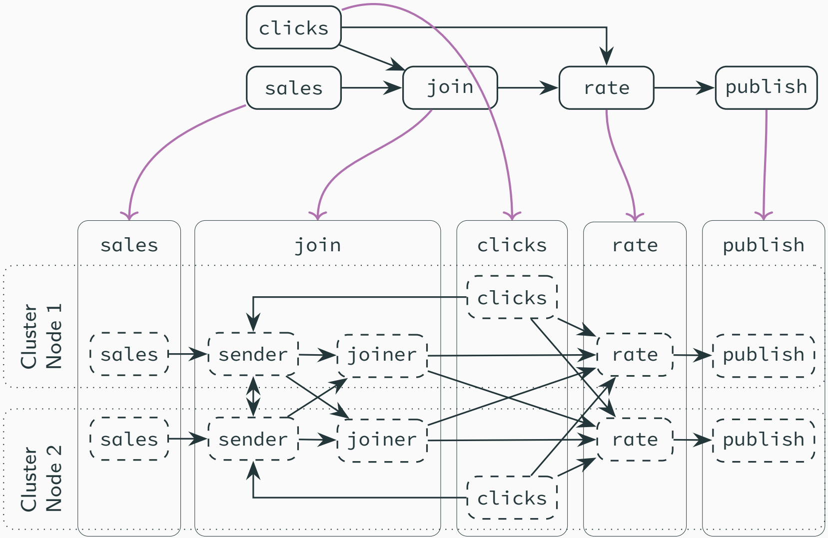
Build application by combining operations into a DAG.

DSPFs: Distribution Over a Cluster



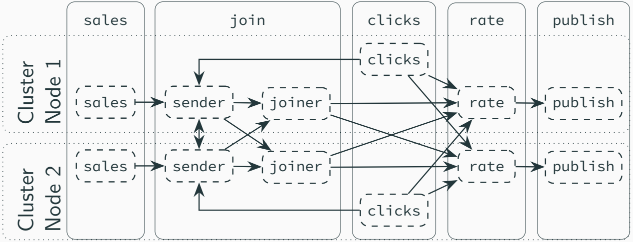


DSPFs: Distribution Over a Cluster



Distribution Strategies

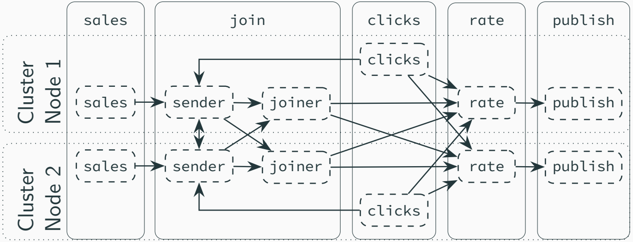
The distribution strategy of an operation determines how it is distributed over the cluster.



- Spawning workers.
- Communication between workers.
- Role performed by each worker.
- Partitioning of state between workers.

Distribution Strategies

The distribution strategy of an operation determines how it is distributed over the cluster.



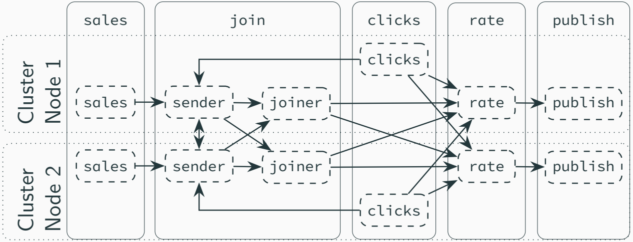
- Spawning workers.
- Communication between workers.
- Role performed by each worker.
- Partitioning of state between workers.

Importance

Distribution strategies are key to the performance of a distributed stream processing application.

Distribution Strategies

The distribution strategy of an operation determines how it is distributed over the cluster.



- Spawning workers.
- Communication between workers.
- Role performed by each worker.
- Partitioning of state between workers.

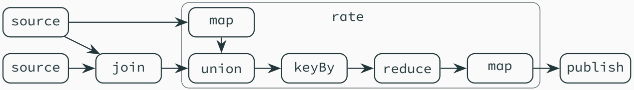
Importance

Distribution strategies are key to the performance of a distributed stream processing application.

Goal

We need a DSPF which makes it easy to select the appropriate distribution strategy.

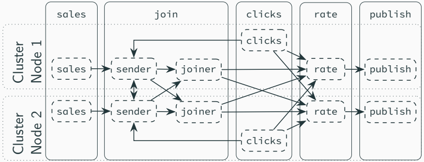
```
clicks = source()
sales = source()

sales.join(clicks)
     .where(… -> …)
     .equalTo(… -> …)
     .apply(… -> …)
     .union(clicks.map(… -> …))
     .keyBy(… -> …)
     .reduce(…, … -> …)
     .map(… -> …)
     .publish()
```



- Limited set of operators.
- Fixed strategy for each operator.

2 – Low-level DSPFs: Wiring DAGs in Storm

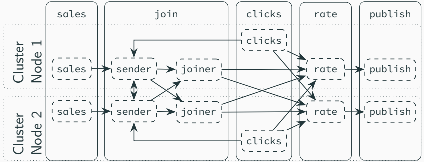
```
b = TopologyBuilder();

b.setSpout("sales", SalesSpout(), 2)
b.setSpout("clicks", ClicksSpout(), 2)

b.setBolt("join-sender", JoinSendBolt(), 2)
 .localGrouping("clicks")
 .localGrouping("sales")
b.setBolt("join-joiner", JoinBolt(), 8)
 .customGrouping("join-sender", JoinBGrouping())

b.setBolt("rate", RateBolt(), 2)
 .fieldsGrouping("clicks", "ad-id")
 .fieldsGrouping("join-joiner", "ad-id")
b.setBolt("publish", PublishBolt(), 2)
 .localGrouping("rate")
```



- Flexible, low-level model.
- Difficult to express strategies.
    - Scattered distribution logic.
    - Tangled distribution and application logic.
    - No support for different worker types.

2 – Low-level DSPFs: Wiring DAGs in Storm

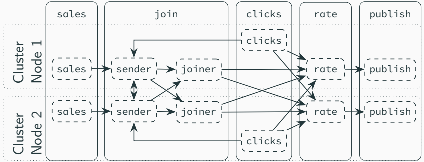
```
b = TopologyBuilder();

b.setSpout("sales", SalesSpout(), 2)
b.setSpout("clicks", ClicksSpout(), 2)

b.setBolt("join-sender", JoinSendBolt(), 2)
 .localGrouping("clicks")
 .localGrouping("sales")
b.setBolt("join-joiner", JoinBolt(), 8)
 .customGrouping("join-sender", JoinBGrouping())

b.setBolt("rate", RateBolt(), 2)
 .fieldsGrouping("clicks", "ad-id")
 .fieldsGrouping("join-joiner", "ad-id")
b.setBolt("publish", PublishBolt(), 2)
 .localGrouping("rate")
```



- Flexible, low-level model.
- Difficult to express strategies.
    - Scattered distribution logic.
    - Tangled distribution and application logic.
    - No support for different worker types.

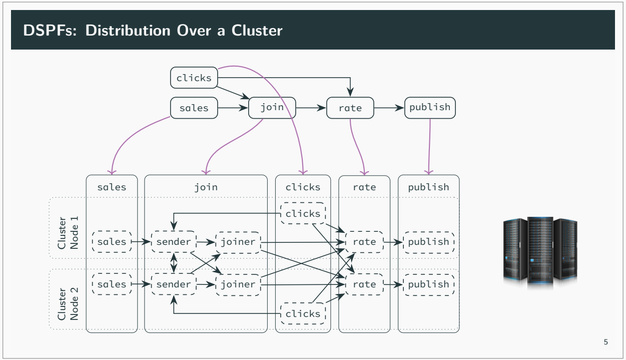
2 – Low-level DSPFs: Wiring DAGs in Storm

```
b = TopologyBuilder();

b.setSpout("sales", SalesSpout(), 2)
b.setSpout("clicks", ClicksSpout(), 2)

b.setBolt("join-sender", JoinSendBolt(), 2)
 .localGrouping("clicks")
 .localGrouping("sales")
b.setBolt("join-joiner", JoinBolt(), 8)
 .customGrouping("join-sender", JoinBGrouping())

b.setBolt("rate", RateBolt(), 2)
 .fieldsGrouping("clicks", "ad-id")
 .fieldsGrouping("join-joiner", "ad-id")
b.setBolt("publish", PublishBolt(), 2)
 .localGrouping("rate")
```



- Flexible, low-level model.
- Difficult to express strategies.
    - Scattered distribution logic.
    - Tangled distribution and application logic.
    - No support for different worker types.

Problem Statement



- High-level model to express applications.
- Flexible model to express distribution strategies.
- In a modular fashion.

skitter.

Skitter

Novel DSPF with Pluggable Distribution Strategies

- Programming model

  Dual Separate abstractions for data processing and distribution logic.

  Open Strategies and operations can be implemented as needed.

- Implementation in Elixir

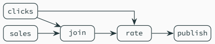
Programming Model(s)

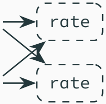
skitter.

```
workflow do
  …
end
```

```
defoperation Rate, … do
  defcb key(data) do
    …
  end

  defcb react(data) do
    …
  end
end
```

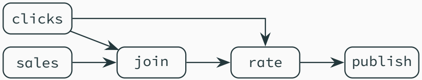
```
defstrategy KeyedState do
  defhook deploy(args) do
    …
  end

  defhook deliver(data) do
    …
  end

  defhook process(data, state, role) do
    …
  end
end
```





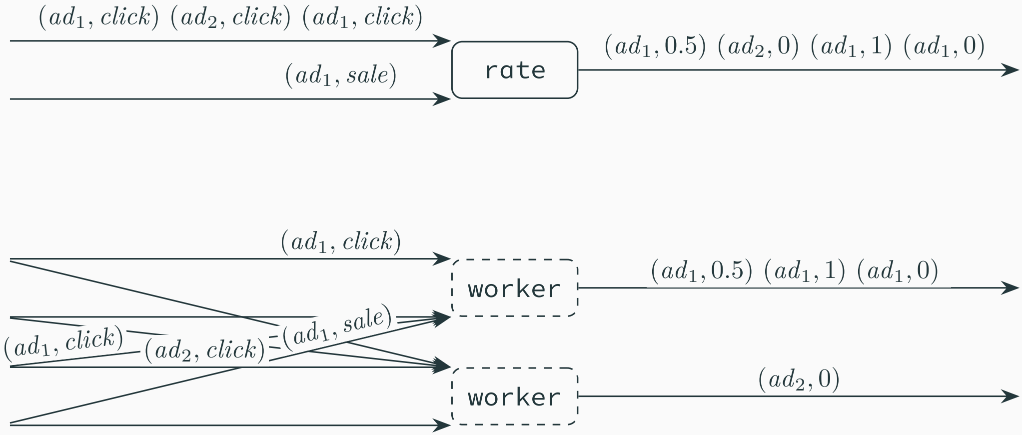
skitter.

```
workflow do
  node(ClicksSource, as: clicks)
  clicks.out ~> join.right
  clicks.out ~> rate.clicks

  node(SalesSource, as: sales)
  ~> node(Join, with: FastJoin, as: join)
  ~> node(Rate, with: KeyedState, as: rate)
  ~> node(Publish)
end
```



Example: Distributing `Rate` with the `KeyedState` Strategy

Example: Distributing `Rate` with the `KeyedState` Strategy



Example: Distributing `Rate` with the `KeyedState` Strategy

skitter.

- skitter. calls strategy hooks (meta level) in response to events.
- Strategy calls operation callbacks (base level) to handle data processing logic.

Event	`KeyedState`	`Rate`
Application start	`deploy(args)`	
Upstream emits *data*	`deliver(data)`	
		`key(data)`
Worker receives *msg*	`process(msg, state, role)`	
		`react(data)`
		`key(data)`

Example: Distributing `Rate` with the `KeyedState` Strategy

skitter.

- skitter. calls strategy hooks (meta level) in response to events.
- Strategy calls operation callbacks (base level) to handle data processing logic.

Event	`KeyedState`	`Rate`
Application start	`deploy(args)`	
Upstream emits *data*	`deliver(data)`	`key(data)`
Worker receives *msg*	`process(msg, state, role)`	`react(data)` `key(data)`

worker

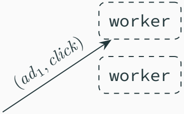
worker

Example: Distributing `Rate` with the `KeyedState` Strategy

skitter.

- skitter. calls strategy hooks (meta level) in response to events.
- Strategy calls operation callbacks (base level) to handle data processing logic.

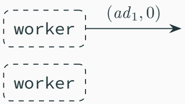
Event	`KeyedState`	`Rate`
Application start	`deploy(args)`	
Upstream emits *data*	`deliver(data)`	`key(data)`
Worker receives *msg*	`process(msg, state, role)`	`react(data)` `key(data)`



Example: Distributing `Rate` with the `KeyedState` Strategy

- skitter calls strategy hooks (meta level) in response to events.
- Strategy calls operation callbacks (base level) to handle data processing logic.

Event	`KeyedState`	`Rate`
Application start	`deploy(args)`	
Upstream emits *data*	`deliver(data)`	`key(data)`
Worker receives *msg*	`process(msg, state, role)`	`react(data)` `key(data)`



Strategies and Operations

- skitter. calls strategy hooks (meta level) in response to events.
  - Hooks are fixed and defined by Skitter.
- Strategy calls operation callbacks (base level) to handle data processing logic.
  - Callbacks to be implemented are defined by the strategy.

```
defstrategy KeyedState do
  defhook deploy(args)

  defhook deliver(data) do
    …
    call(:key, args: [data])
    …
  end

  defhook process(data, state, role) do
    …
    call(:key, args: [data])
    call(:react, state: state, args: [data])
    …
  end
end
```

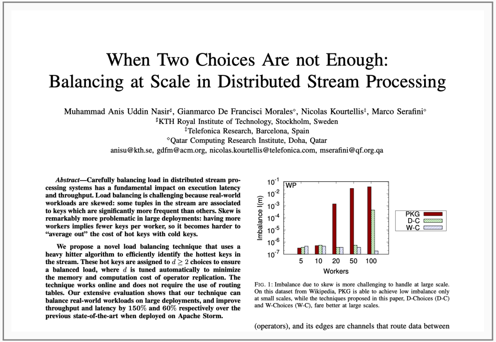
```
defoperation Rate, … do
  defcb key(data) do
    …
  end

  defcb react(data) do
    …
  end
end
```

Evaluation

Research Questions

Qualitative	*Modularity*	Does Skitter enable the expression of distribution strategies in a modular fashion?
Quantitative	*Performance*	Does Skitter influence the performance characteristics of distribution strategies?
	Overhead	Do the Skitter language abstractions introduce a significant amount of overhead?
	Impact	Can application performance be improved by selecting an alternative strategy?

Experimental Setup



When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing

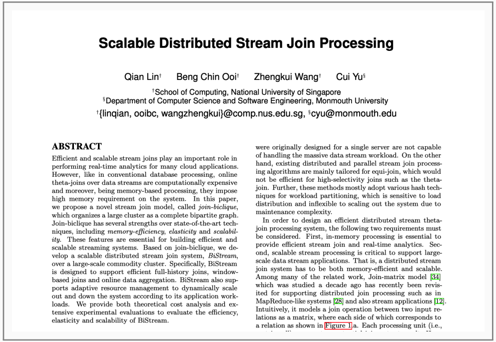
Muhammad Anis Uddin Nasir[♯], Gianmarco De Francisci Morales[◦], Nicolas Kourtellis[‡], Marco Serafini[◦]
[♯]KTH Royal Institute of Technology, Stockholm, Sweden
[‡]Telefonica Research, Barcelona, Spain
[◦]Qatar Computing Research Institute, Doha, Qatar
anisu@kth.se, gdfm@acm.org, nicolas.kourtellis@telefonica.com, mserafini@qf.org.qa

***Abstract*—Carefully balancing load in distributed stream processing systems has a fundamental impact on execution latency and throughput. Load balancing is challenging because real-world workloads are skewed: some tuples in the stream are associated to keys which are significantly more frequent than others. Skew is remarkably more problematic in large deployments: having more workers implies fewer keys per worker, so it becomes harder to "average out" the cost of hot keys with cold keys.**

We propose a novel load balancing technique that uses a heavy hitter algorithm to efficiently identify the hottest keys in the stream. These hot keys are assigned to $d \geq 2$ choices to ensure a balanced load, where d is tuned automatically to minimize the memory and computation cost of operator replication. The technique works online and does not require the use of routing tables. Our extensive evaluation shows that our technique can balance real-world workloads on large deployments, and improve throughput and latency by 150% and 60% respectively over the previous state-of-the-art when deployed on Apache Storm.

FIG. 1: Imbalance due to skew is more challenging to handle at large scale. On this dataset from Wikipedia, PKG is able to achieve low imbalance only at small scales, while the techniques proposed in this paper, D-Choices (D-C) and W-Choices (W-C), fare better at large scales.

(operators), and its edges are channels that route data between



Scalable Distributed Stream Join Processing

Qian Lin[†] Beng Chin Ooi[†] Zhengkui Wang[†] Cui Yu[§]
[†]School of Computing, National University of Singapore
[§]Department of Computer Science and Software Engineering, Monmouth University
[†]{linqian, ooibc, wangzhengkui}@comp.nus.edu.sg, [§]cyu@monmouth.edu

ABSTRACT

Efficient and scalable stream joins play an important role in performing real-time analytics for many cloud applications. However, like in conventional database processing, online theta-joins over data streams are computationally expensive and moreover, being memory-based processing, they impose high memory requirement on the system. In this paper, we propose a novel stream join model, called *join-biclique*, which organizes a large cluster as a complete bipartite graph. Join-biclique has several strengths over state-of-the-art techniques, including *memory-efficiency*, *elasticity* and *scalability*. These features are essential for building efficient and scalable streaming systems. Based on join-biclique, we develop a scalable distributed stream join system, *BiStream*, over a large-scale commodity cluster. Specifically, BiStream is designed to support efficient full-history joins, window-based joins and online data aggregation. BiStream also supports adaptive resource management to dynamically scale out and down the system according to its application workloads. We provide both theoretical cost analysis and extensive experimental evaluations to evaluate the efficiency, elasticity and scalability of BiStream.

were originally designed for a single server are not capable of handling the massive data stream workload. On the other hand, existing distributed and parallel stream join processing algorithms are mainly tailored for equi-join, which would not be efficient for high-selectivity joins such as the theta-join. Further, these methods mostly adopt various hash techniques for workload partitioning, which is sensitive to load distribution and inflexible to scaling out the system due to maintenance complexity.

In order to design an efficient distributed stream theta-join processing system, the following two requirements must be considered. First, in-memory processing is essential to provide efficient stream join and real-time analytics. Second, scalable stream processing is critical to support large-scale data stream applications. That is, a distributed stream join system has to be both memory-efficient and scalable. Among many of the related work, Join-matrix model [34] which was studied a decade ago has recently been revisited for supporting distributed join processing such as in MapReduce-like systems [28] and also stream applications [12]. Intuitively, it models a join operation between two input relations as a matrix, where each side of which corresponds to a relation as shown in Figure 1a. Each processing unit (i.e.,

ICDE'16
115 citations

SIGMOD/PODS'15
113 citations

Comparison of multiple distribution strategies
Performance evaluation in Storm

Experimental Setup

Benchmark	Strategy	Label
WordCount	D-Choices	D-C
	W-Choices	W-C
	Partial Key Grouping	PKG
	Key Grouping	KG
	Shuffle Grouping	SG
Join	Join-Matrix	JM
	Join-Biclique	JB
	Join-Biclique ContRand	JB-CR

- 3 implementations: Storm, Skitter, ad-hoc (Elixir)
- Used to compare modularity and performance (average throughput)

Q1: Modularity

Question

How modular are distribution strategies in Skitter compared to the state of the art (Storm)?

Benchmark	Strategy	Label
WordCount	D-Choices	D-C
	W-Choices	W-C
	Partial Key Grouping	PKG
	Key Grouping	KG
	Shuffle Grouping	SG
Join	Join-Matrix	JM
	Join-Biclique	JB
	Join-Biclique ContRand	JB-CR

- Measure LOC added or modified to change distribution strategy.
- Categorize LOC based on abstractions offered by framework.

Q1: Modularity (Join)

Question

How modular are distribution strategies in Skitter compared to the state of the art (Storm)?

	Strategy	Storm			Skitter		
		Topology	Component	Grouping	Workflow	Operation	Strategy
Q5	JB	29	162	46	3	0	119
	JB-CR	29	162	61	3	0	134
Q7	JB	22	162	46	2	0	119
	JB-CR	22	162	61	2	0	134

Q1: Modularity (Join)

Question

How modular are distribution strategies in Skitter compared to the state of the art (Storm)?

	Strategy	Storm			Skitter		
		Topology	Component	Grouping	Workflow	Operation	Strategy
Q5	JB	29	162	46	3	0	119
	JB-CR	29	162	61	3	0	134
Q7	JB	22	162	46	2	0	119
	JB-CR	22	162	61	2	0	134

Q1: Modularity (Join)

Question

How modular are distribution strategies in Skitter compared to the state of the art (Storm)?

	Strategy	Storm			Skitter		
		Topology	Component	Grouping	Workflow	Operation	Strategy
Q5	JB	29	162	46	3	0	119
	JB-CR	29	162	61	3	0	134
Q7	JB	22	162	46	2	0	119
	JB-CR	22	162	61	2	0	134

Q2: Performance

Question

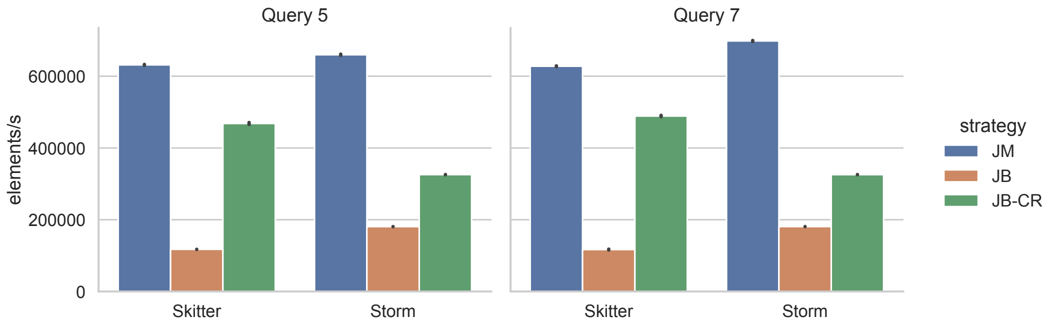
Do strategies implemented in Skitter maintain their performance characteristics?

- Compare the *relative* performance of Storm and Skitter implementations of the same experiments.

Q2: Performance (Join)

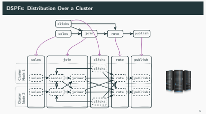
Question

Do strategies implemented in Skitter maintain their performance characteristics?



Conclusion

Problem Statement



- High-level model to express applications.
- Flexible model to express distribution strategies.
- In a modular fashion.

9

Programming Model(s)

skitter.

```
workflow do
  …
end
```

```
defoperation Rate, … do
  defcb key(data) do
    …
  end

  defcb react(data) do
    …
  end
end
```

```
defstrategy KeyedState do
  defhook deploy(args) do
    …
  end

  defhook deliver(data) do
    …
  end

  defhook process(data, state, role) do
    …
  end
end
```

11

Q1: Modularity (Join)

Question

How modular are distribution strategies in Skitter compared to the state of the art (Storm)?

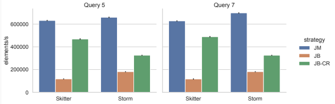
	Strategy	Storm			Skitter		
		Topology	Component	Grouping	Workflow	Operation	Strategy
Q5	JB	29	162	46	3	0	119
	JB-CR	29	162	61	3	0	134
Q7	JB	22	162	46	2	0	119
	JB-CR	22	162	61	2	0	134

22

Q2: Performance (Join)

Question

Do strategies implemented in Skitter maintain their performance characteristics?



24

https://soft.vub.ac.be/~mathsaey/skitter/

Skitter: A Distributed Stream Processing Framework with Pluggable Distribution Strategies

Mathijs Saey, Joeri De Koster, Wolfgang De Meuter

mathijs.saey@vub.be

High-level Style Workflow Definitions

skitter.

```
workflow do
  source()
  ~> flatmap(&String.split/1, with: RepartitionedOutput)
  ~> keyed_reduce(fn word -> word end, fn count -> count + 1 end, 0)
  ~> print()
end

workflow do
  node(SomeSource)
  ~> node(FlatMap, args: [&String.split/1], with: RepartitionedOutput)
  ~> node(KeyedReduce, args: [fn word -> word end, fn count -> count + 1 end, 0])
  ~> node(Print)
end
```

Full Rate Definition

skitter.

```
defoperation Rate, in: [sales, clicks], out: conversion_rate, strategy: KeyedState do
  initial_state {0, 0}

  defcb key(data), do: data.ad_id

  defcb react(data) do
    {clicks, sales} = state()
    {new_clicks, new_sales} = case port_of(data) do
      :sales -> {clicks, sales + 1}
      :clicks -> {clicks + 1, sales}
    end
    state <~ {new_clicks, new_sales}
    {data.ad_id, new_sales / new_clicks} ~> conversion_rate
  end
end
```

Full KeyedState Definition

skitter.

```
defstrategy KeyedState do
  defhook deploy(args) do
    Remote.on_all_workers(fn -> local_worker(Map.new(), :aggregator) end)
    |> Enum.map(fn {remote, worker} -> worker end)
  end

  defhook deliver(data) do
    key = call(:key, args: [data]).result
    aggregators = deployment()
    idx = rem(Murmur.hash_x86_32(key), length(aggregators))
    worker = Enum.at(aggregators, idx)
    send(worker, data)
  end

  defhook process(data, state_map, :aggregator) do
    key = call(:key, args: [data]).result
    state = Map.get(state_map, key, initial_state())
    res = call(:react, state: state, args: [data])
    emit(res.emit)
    Map.put(state_map, key, res.state)
  end
end
```

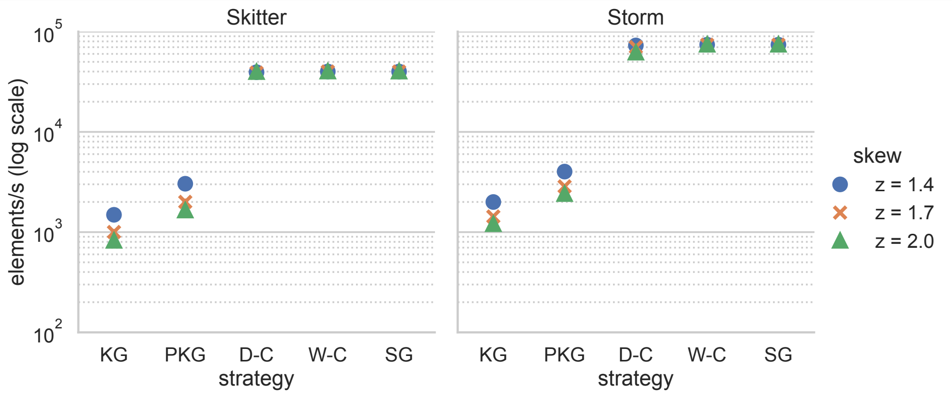
Q1: Modularity (WordCount)

Strategy	Storm			Skitter		
	Topology	Component	Grouping	Workflow	Operation	Strategy
SG	1	0	0	1	0	8
PKG	1	0	0	1	0	46
W-C	1	0	29	1	0	71
D-C	1	0	59	1	0	107
PKG†	4	38	0	1	4	65
W-C†	4	38	29	1	4	90
D-C†	4	38	59	1	4	126

Q2: Performance (WordCount)

Question

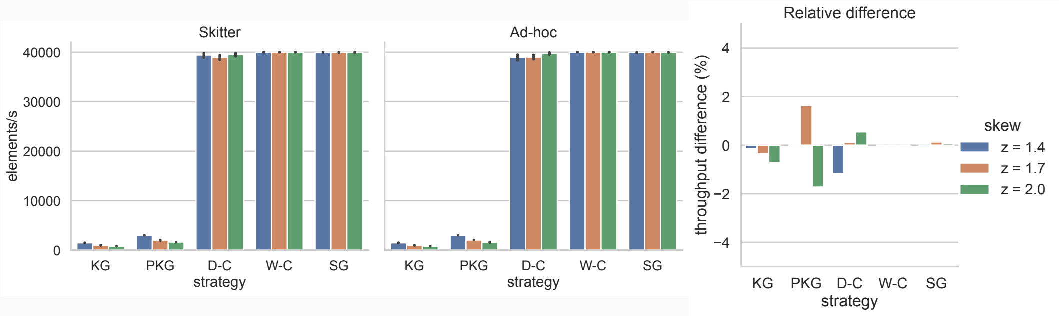
Do strategies implemented in Skitter maintain their performance characteristics?



Q3: Overhead (WordCount)

Question

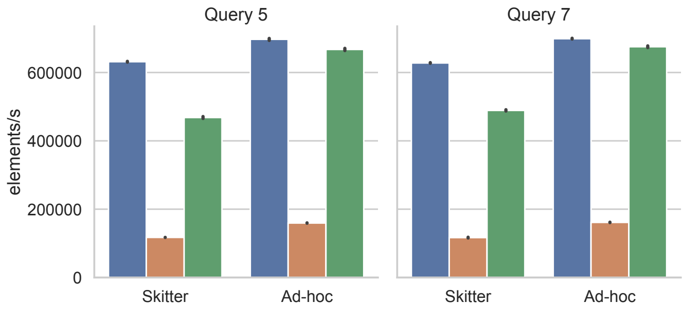
Do the abstractions introduced by Skitter introduce additional overhead?

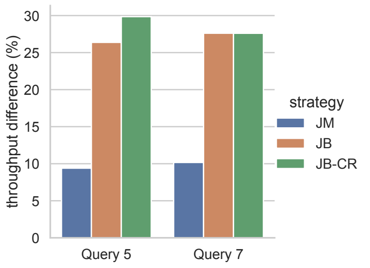


Q3: Overhead (Join)

Question

Do the abstractions introduced by Skitter introduce additional overhead?

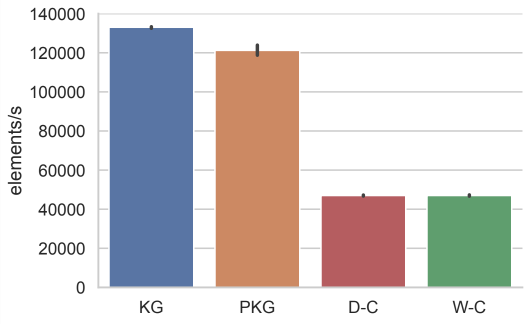




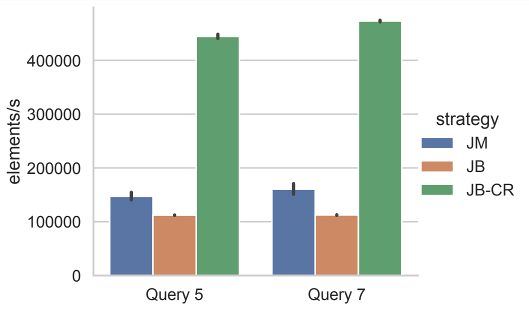
Q4: Impact

Question

Can we improve performance by changing distribution strategy?



WordCount benchmark with key merging and no skew ($z = 0$).



Join benchmark handling 80GB of data.