Scalability and State:

A Critical Assessment of **Throughput** Obtainable on Big Data Streaming Frameworks for Applications **with** and **without State Information**

Shinhyung Yang, Yonguk Jeong, ChangWan Hong, Hyunje Jun and Bernd Burgstaller

Department of Computer Science

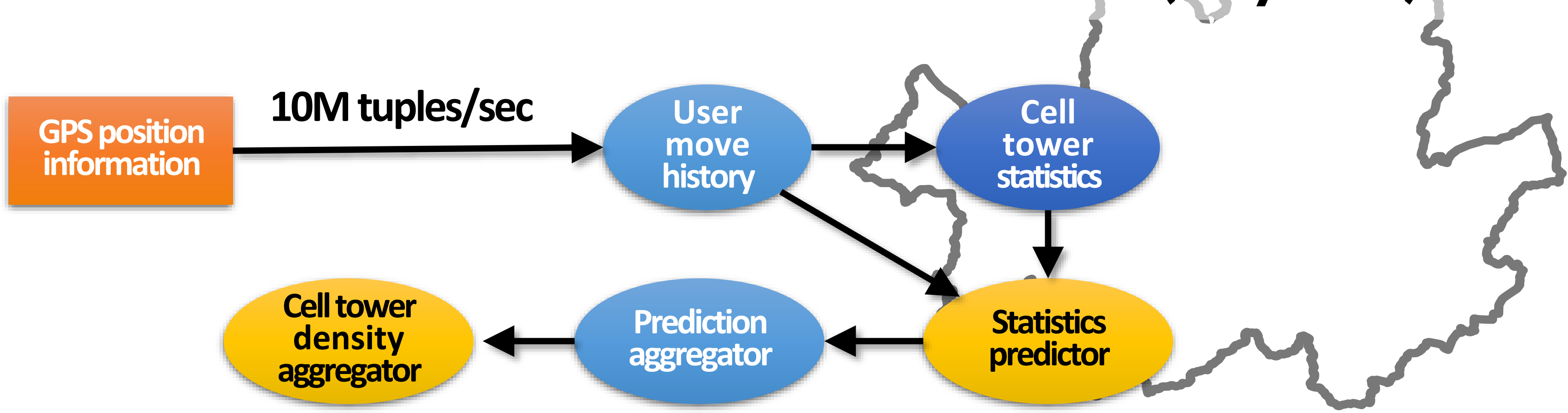
Yonsei University

International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2017), Santiago de Compostela, August 29, 2017.

Motivation

- Characteristics of real-time stream processing:
  - sub-second latency incoming events
  - arriving at high velocity and high density
  - real-time data analysis on incoming streams
  - information perishes over time (e.g., GPS data)
- Example: Urban traffic management

Population of Seoul: 10M (daytime)



- Batch Processing (MapReduce) is unable to meet the sub-second latency requirements of stream analytics applications

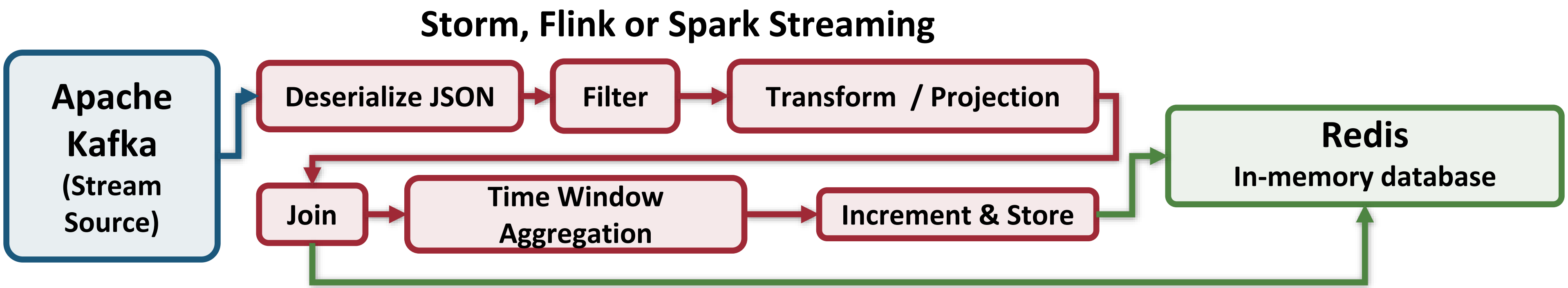
Contributions

1. Determining maximum throughput obtainable from current streaming engines
   - Apache Storm, Apache Flink, Spark Streaming
2. Created and adapted streaming analysis benchmarks
   - **Adapted:** Yahoo streaming benchmark
     - simulation of an advertisement analytics pipeline
   - **Created:** trend detection benchmark
     - real-world streaming analysis identifying and predicting importance of real-world events
3. Dynamic Cloud profiling through Kieker framework
4. Made production-level framework configurations available on GitHub (for reproducibility of results)
5. Compared Cloud trend detector to a hand-tuned single-node lock-less shared memory trend detection re-implementation.
   - To check for possible glass ceiling with streaming framework performance.

Contributions

1. Determining maximum throughput obtainable from current streaming engines
   - Apache Storm, Apache Flink, Spark Streaming
2. Created and adapted streaming analysis benchmarks
   - **Adapted:** Yahoo streaming benchmark
     - simulation of an advertisement analytics pipeline
   - **Created:** trend detection benchmark
     - real-world streaming analysis identifying and predicting importance of real-world events
3. Dynamic Cloud profiling through Kieker framework
4. Made production-level framework configurations available on GitHub (for reproducibility of results)
5. Compared Cloud trend detector to a hand-tuned single-node lock-less shared memory trend detection re-implementation.
   - To check for possible glass ceiling with streaming framework performance.

Benchmark 1: Yahoo streaming benchmark



- Tests the performance of existing Big Data streaming engines:
  - Apache Storm, Apache Flink, and Apache Spark Streaming
- An advertising analytics pipeline of streaming operations:
  - events arrive through Kafka
  - JSON format is deserialized
  - events are filtered, projected, and joined
  - windowed counts of events per campaign are stored in the Redis in-memory database

Experimental Setup

- Cloud setup
  - from Yahoo's publication [YH2016]
  - 30 Cloud nodes are configured on Google Compute Engine
- One Cloud node is equipped with:
  - 16 virtual CPUs (vCPUs)
    - aka 16 Intel hyperthreads
    - Intel Xeon @ 2.50 GHz
  - 24 GB RAM

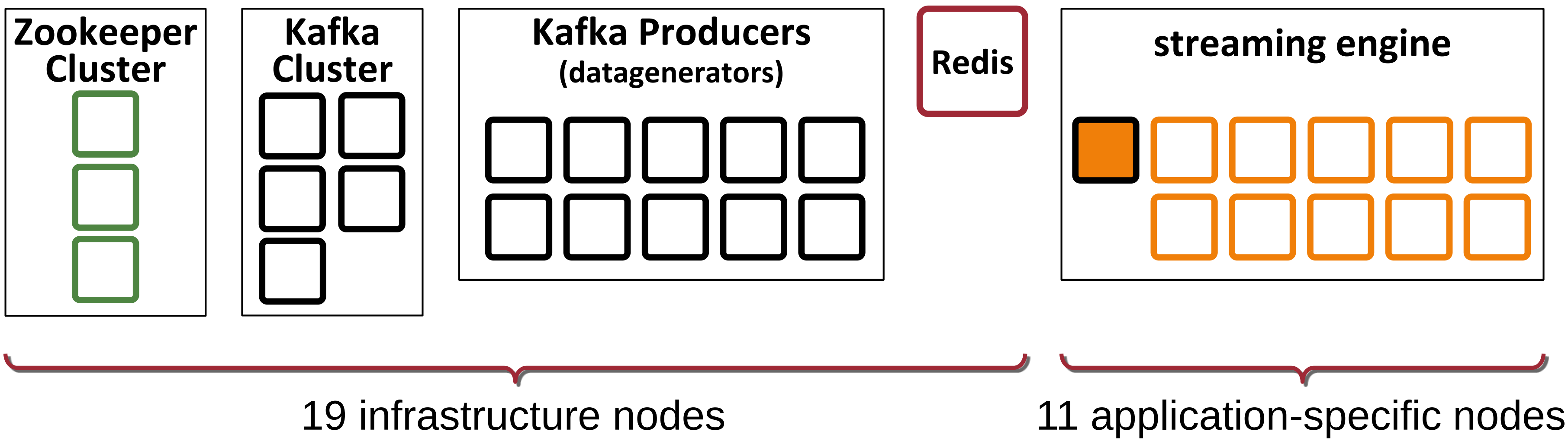
Experimental Setup

- Cloud setup
  - from Yahoo's publication [YH2016]
  - 30 Cloud nodes are configured on Google Compute Engine
- One Cloud node is equipped with:
  - 16 virtual CPUs (vCPUs)
    - aka 16 Intel hyperthreads
    - Intel Xeon @ 2.50 GHz
  - 24 GB RAM

The total provided Cloud resources include:

- 480 vCPUs
- 720 GB RAM

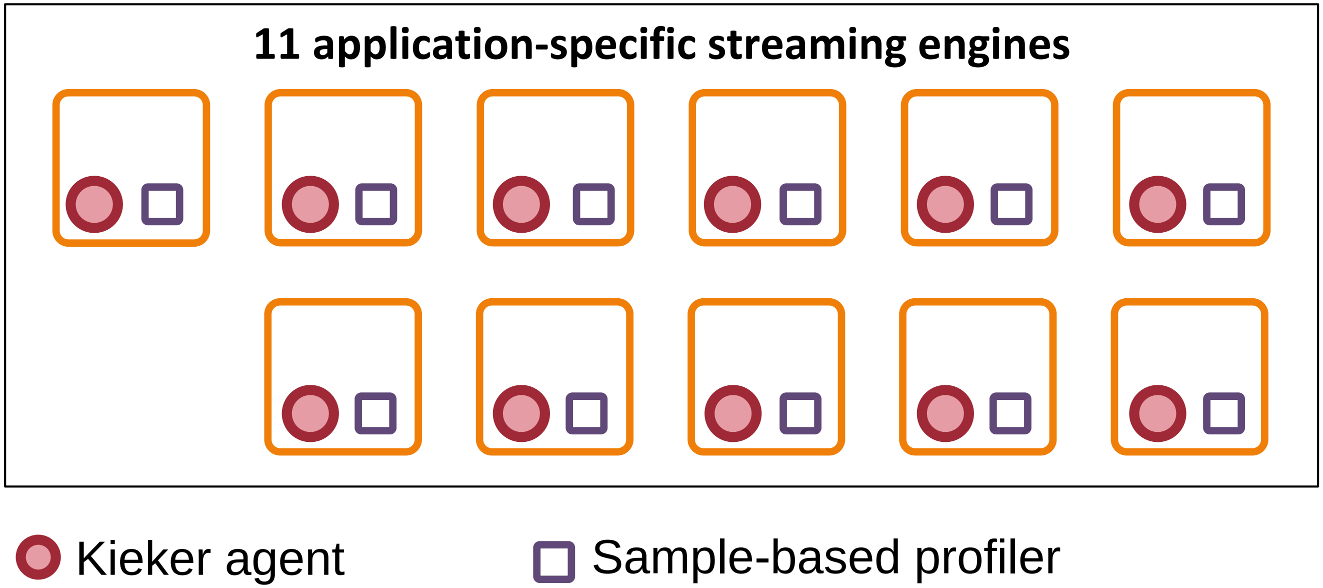
Cloud Infrastructure vs. Application Nodes



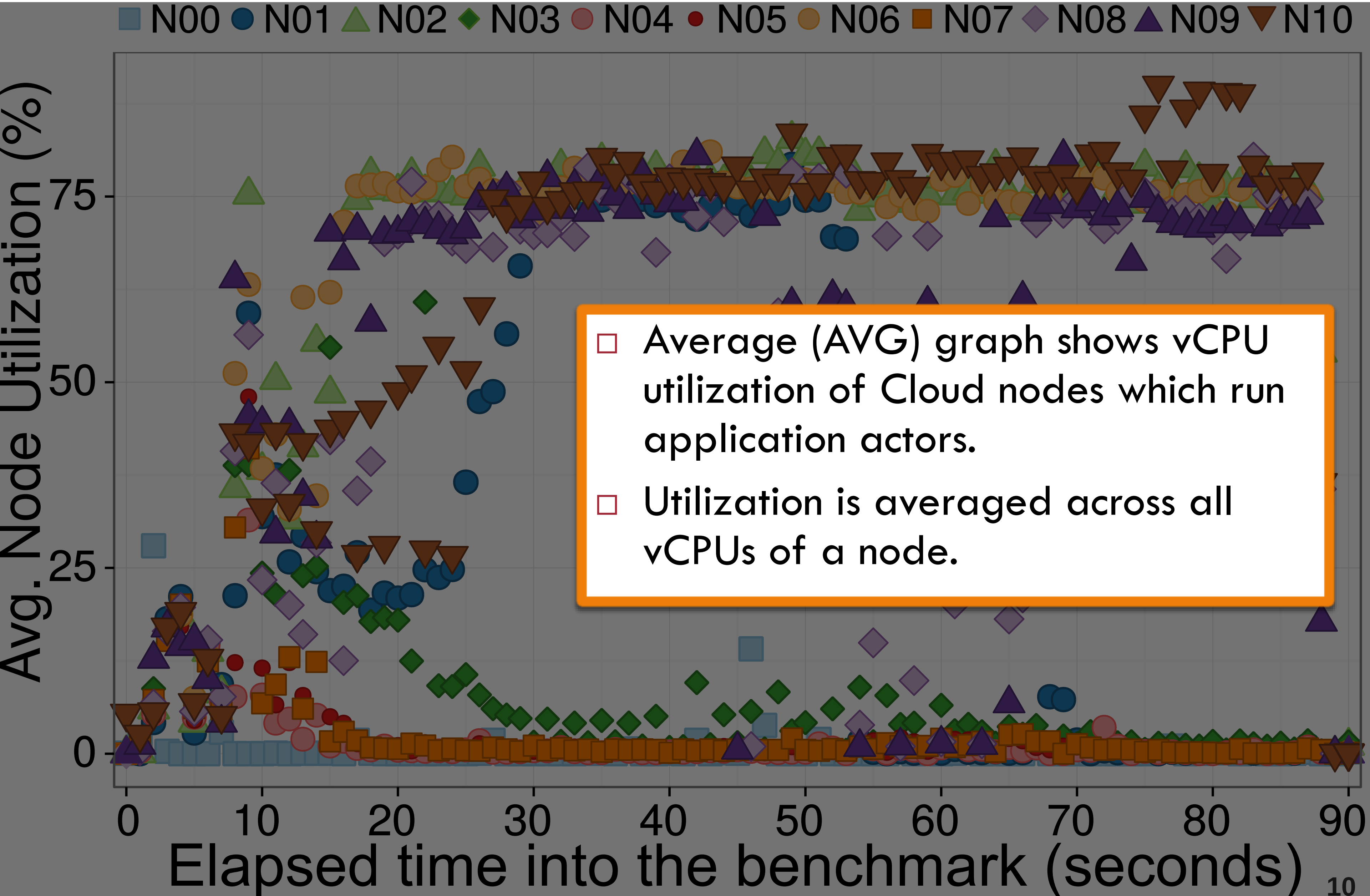
- Cloud infrastructure setup
  - 3 Zookeeper nodes
  - 1 Redis in-memory database node
  - 1 Kafka cluster (5 Kafka broker nodes)
  - 10 Kafka producer nodes

Benchmarking Cloud applications

- Measuring CPU utilization in the Cloud
  - Kieker dynamic profiling framework
    - specialized at measuring performance of Cloud systems
  - Kieker agent and our sample-based profiler deployed with all application-specific nodes
    - per-core, per-second CPU utilization of nodes is sampled every 500 ms (to fulfill per-second sampling rate)
  - Our sample-based profiler accumulates sampling data on all nodes

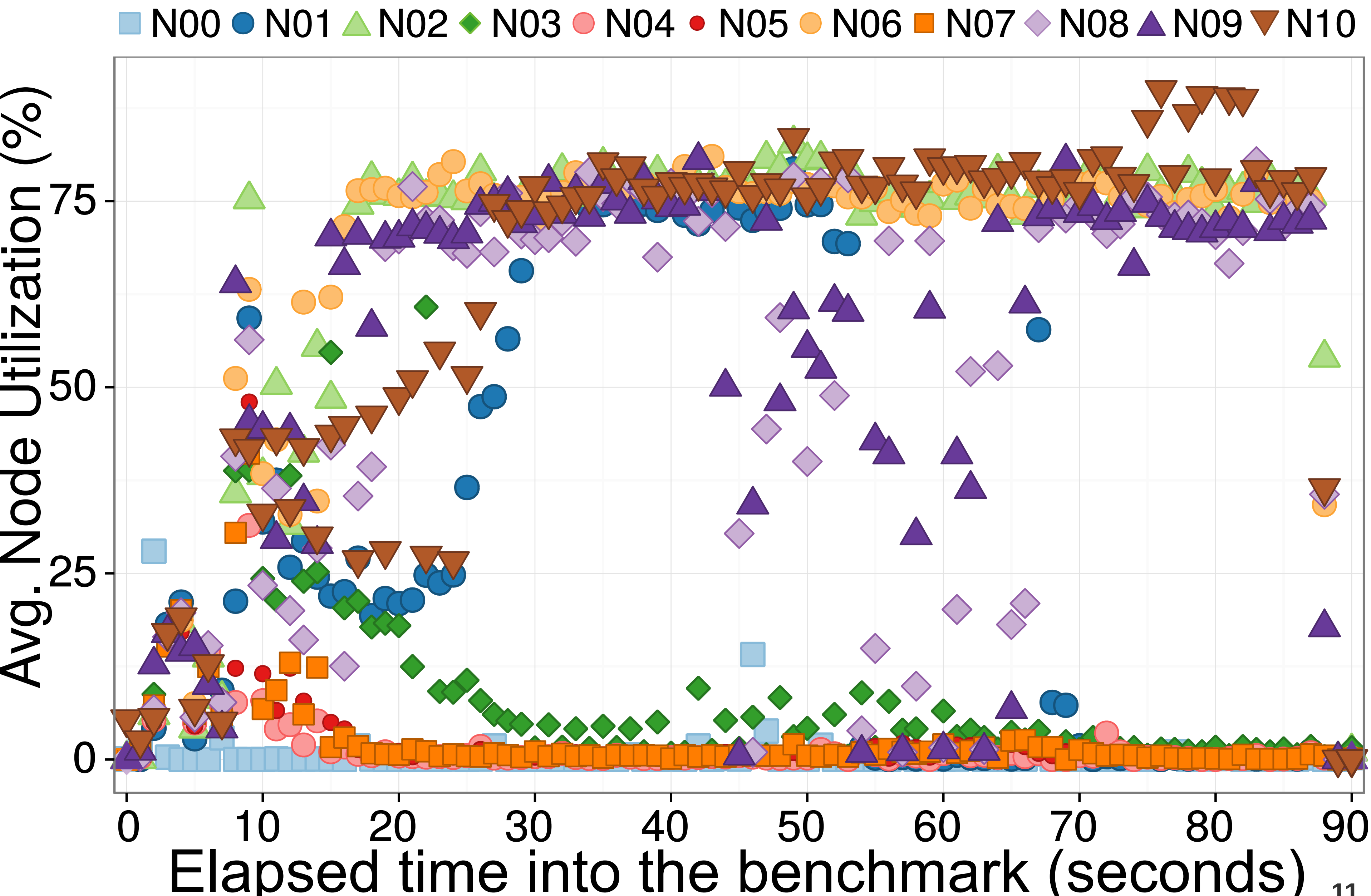


Storm: Average Node vCPU Utilization

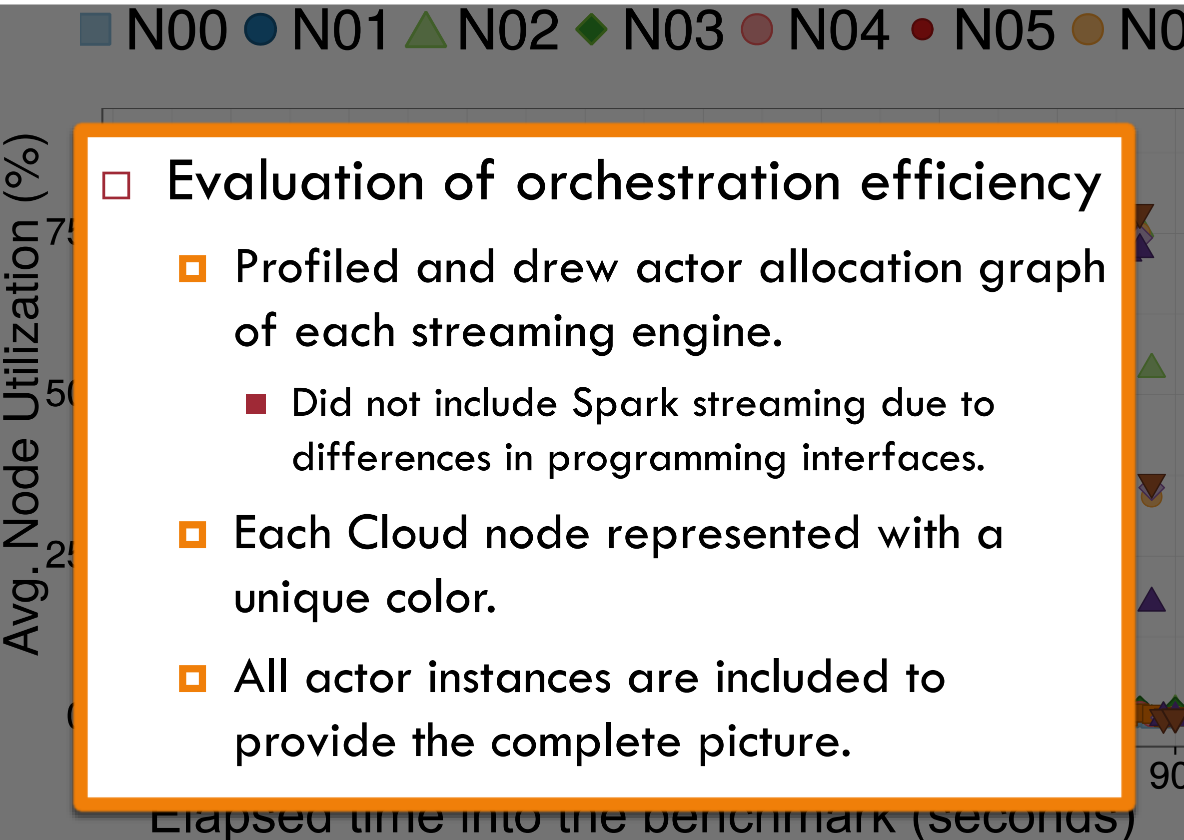


- Average (AVG) graph shows vCPU utilization of Cloud nodes which run application actors.
- Utilization is averaged across all vCPUs of a node.

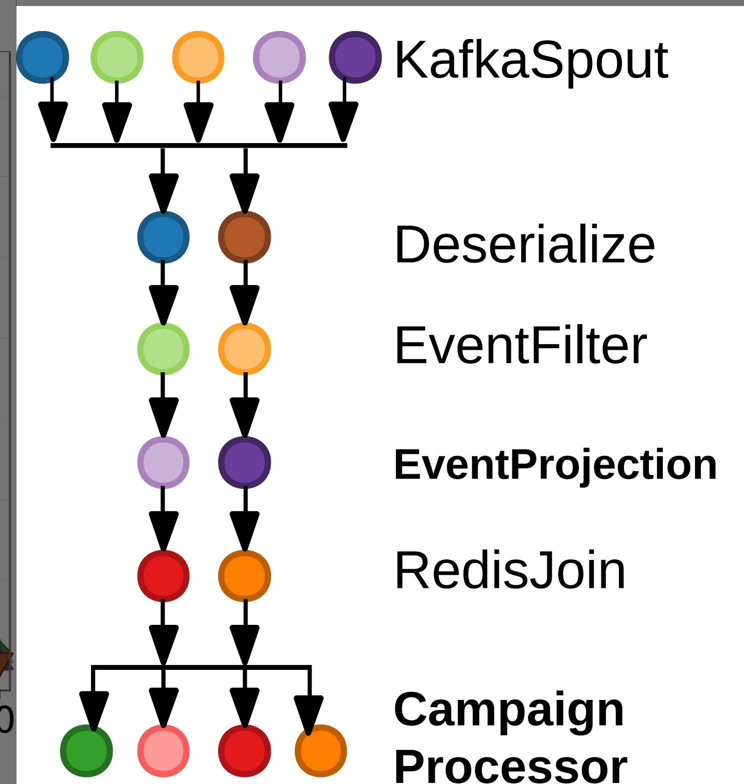
Storm: Average Node vCPU Utilization



Storm: Actor Instance Allocation

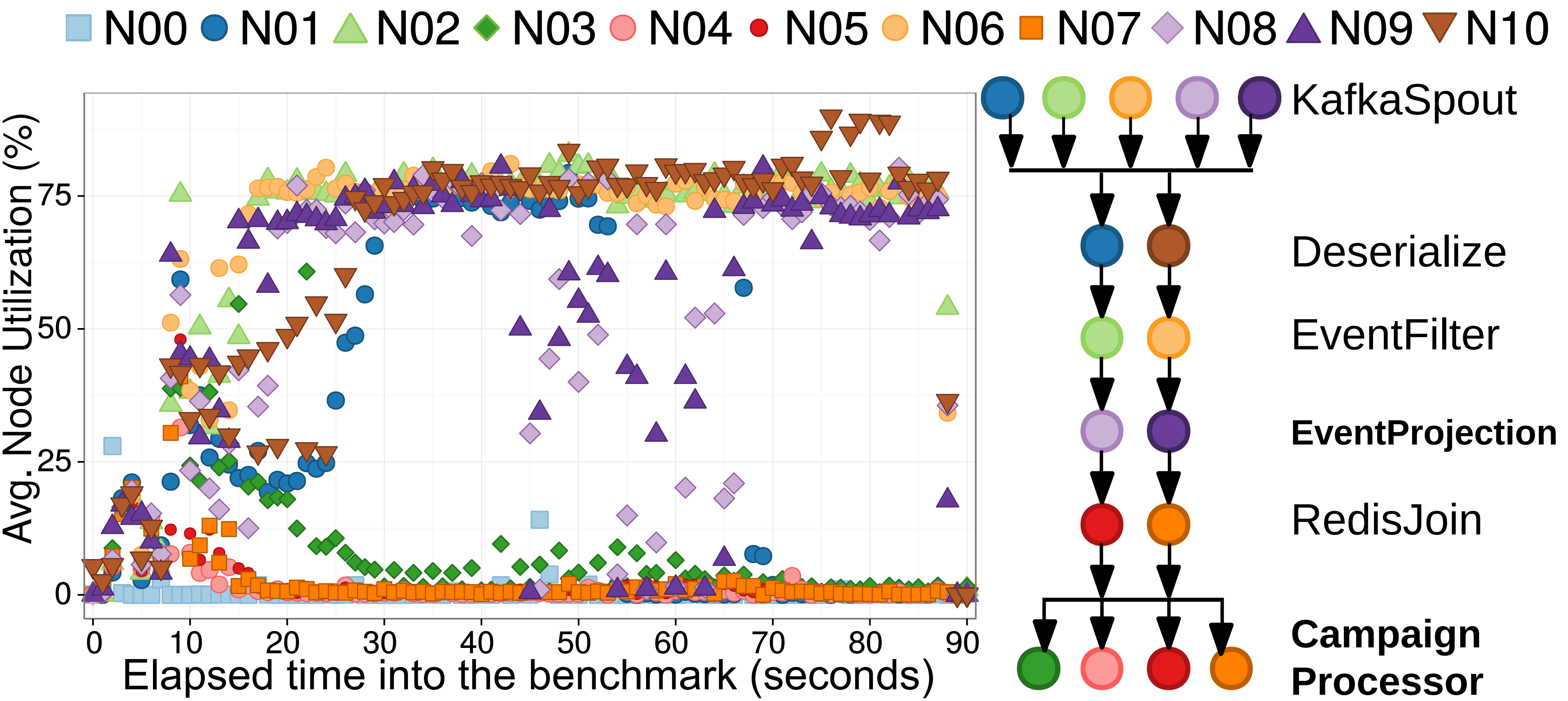


- Evaluation of orchestration efficiency
  - Profiled and drew actor allocation graph of each streaming engine.
    - Did not include Spark streaming due to differences in programming interfaces.
  - Each Cloud node represented with a unique color.
  - All actor instances are included to provide the complete picture.



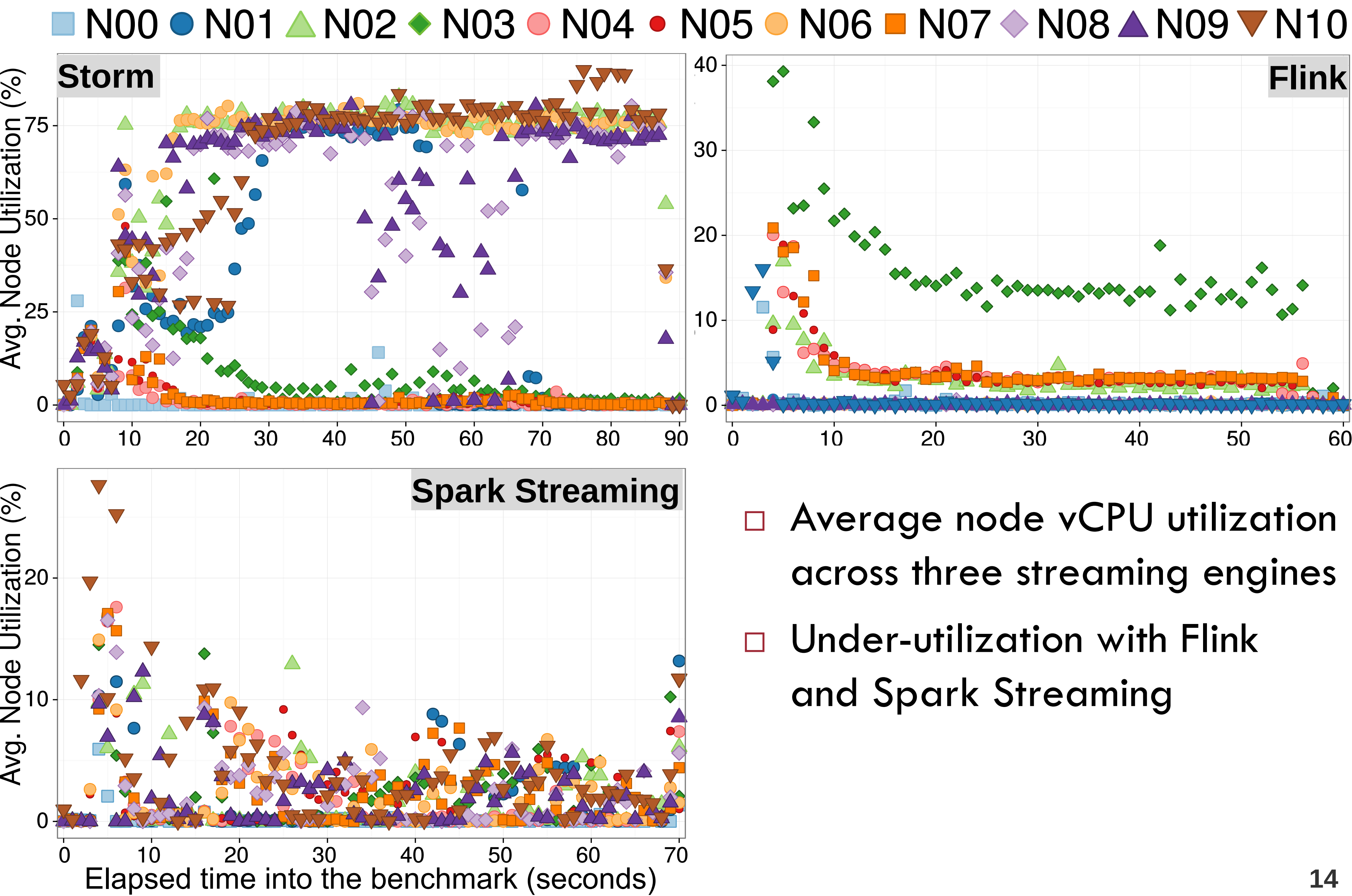
- Same color is used for the same Cloud node in the graph and orchestration diagram

Storm: Actor Instance Allocation



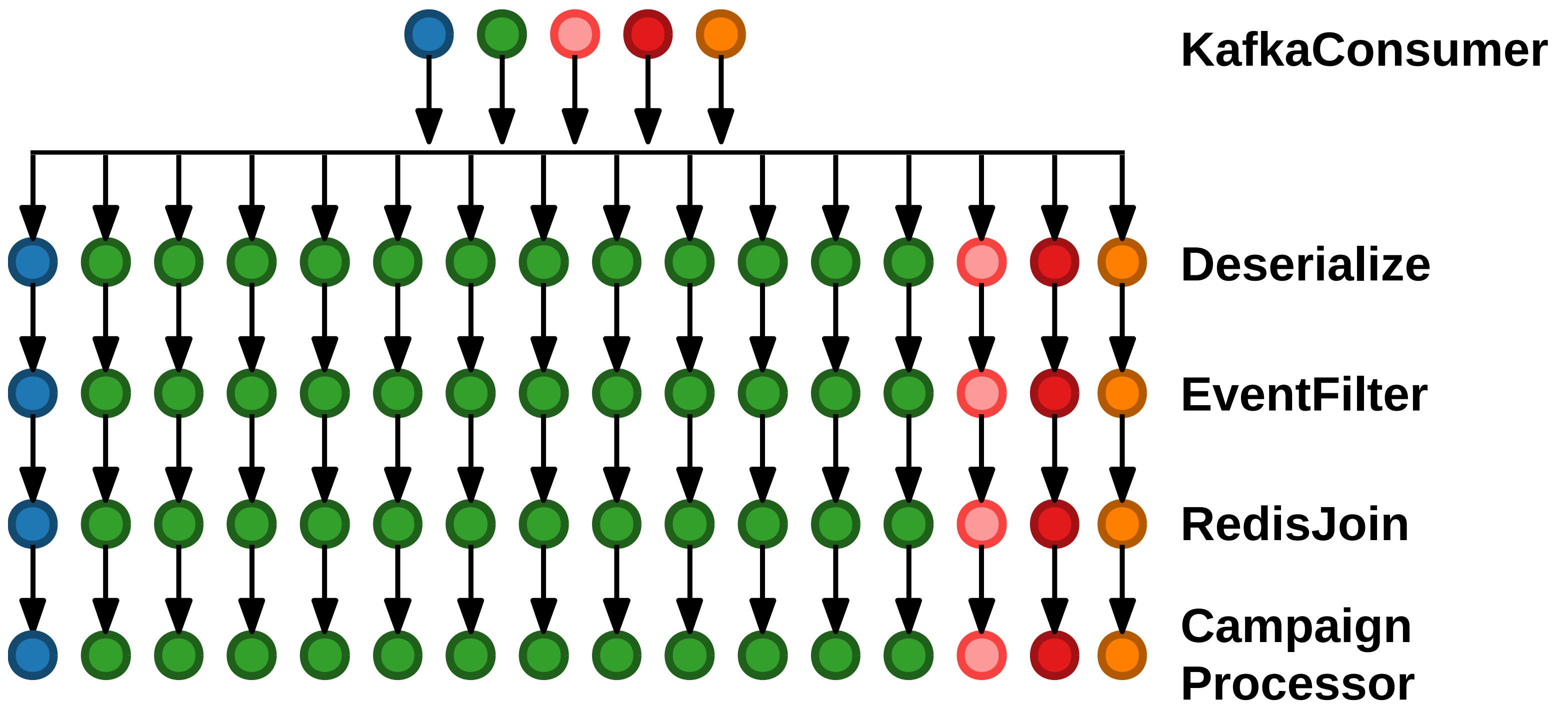
- Same color is used for the same Cloud node in the graph and orchestration diagram

Comparing Streaming Engines



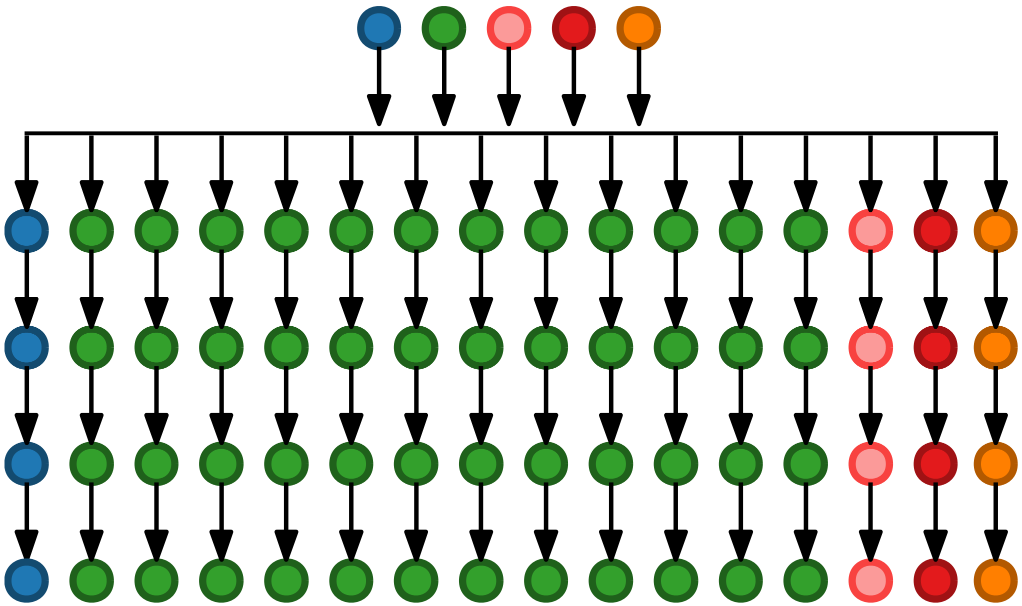
- Average node vCPU utilization across three streaming engines
- Under-utilization with Flink and Spark Streaming

Flink Actor Instance Allocation

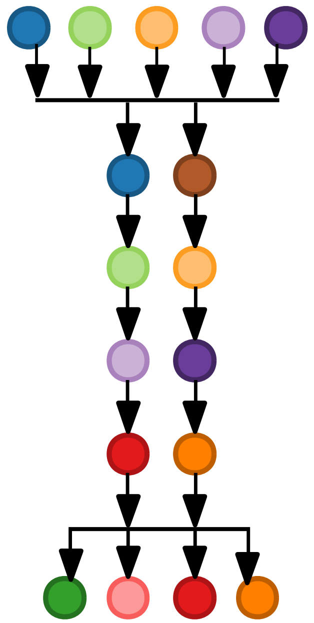


- Flink's orchestration graph
  - Flink actors are confined to 5 nodes; 6 nodes **left idle.**
  - One node (green) overly allocated with actor instances.
  - Flink favors vertical over horizontal scaling, although not load-balanced.

Differences in Orchestration Strategies



Flink's Orchestration

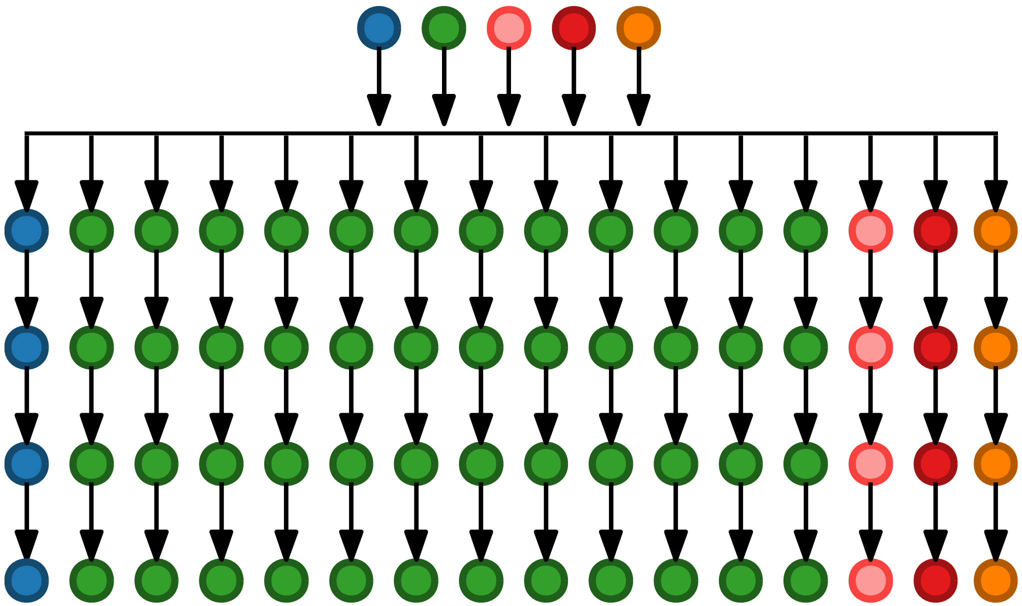


Storm's Orchestration

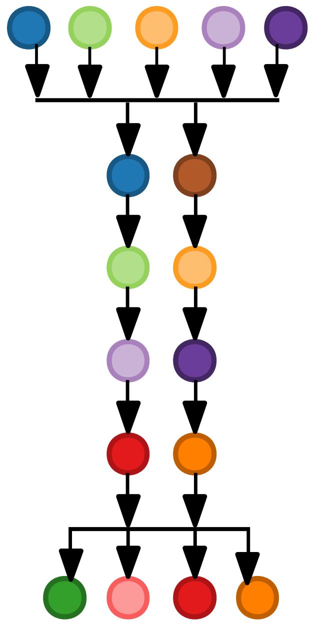
Orchestration Details

Streaming engine	Flink	Storm
Number of participating nodes	5	10
Throughput (tuples/sec)	282,141	24,703

Orchestration Strategies Differences



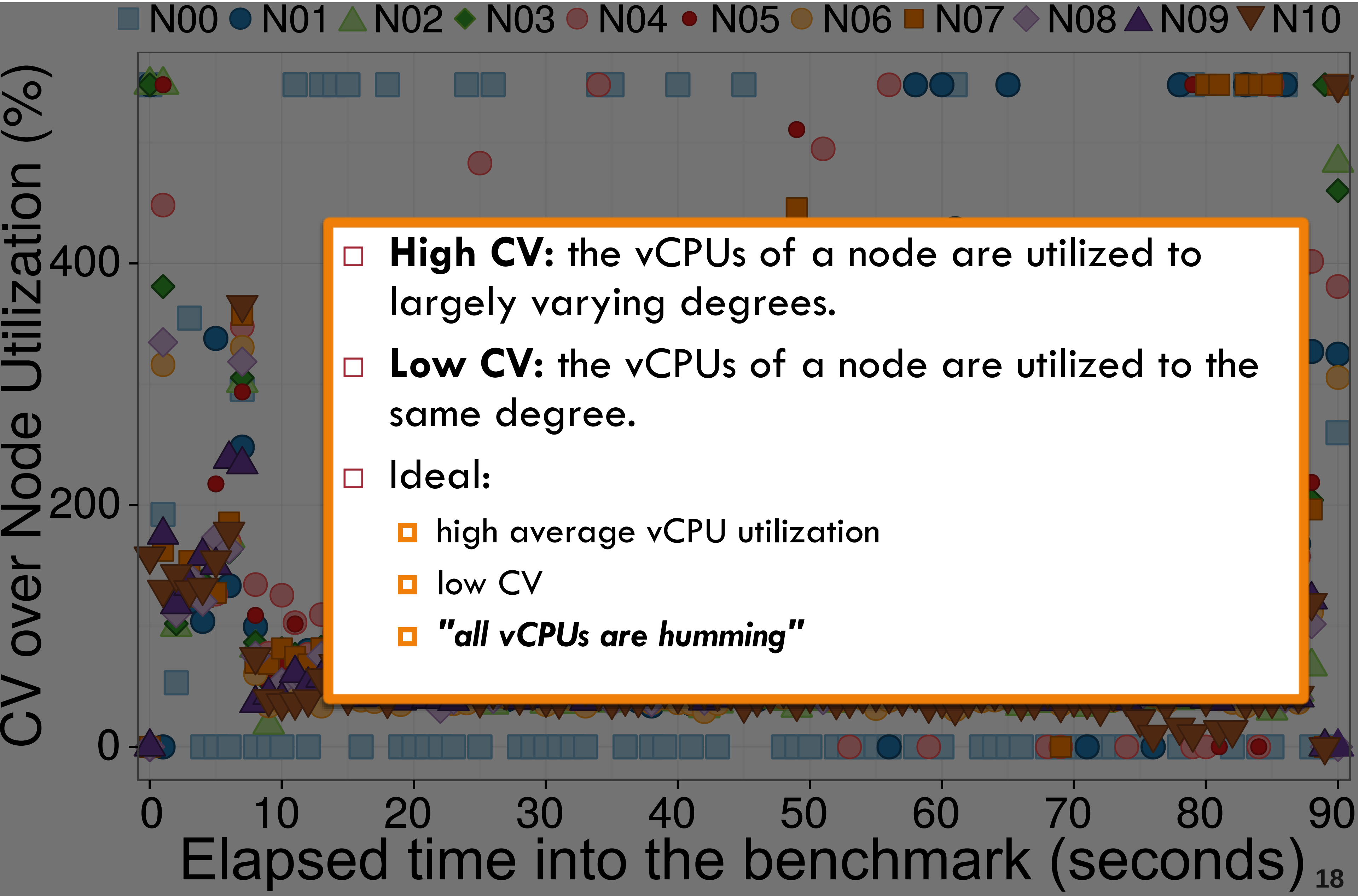
Flink's Orchestration



Storm's Orchestration

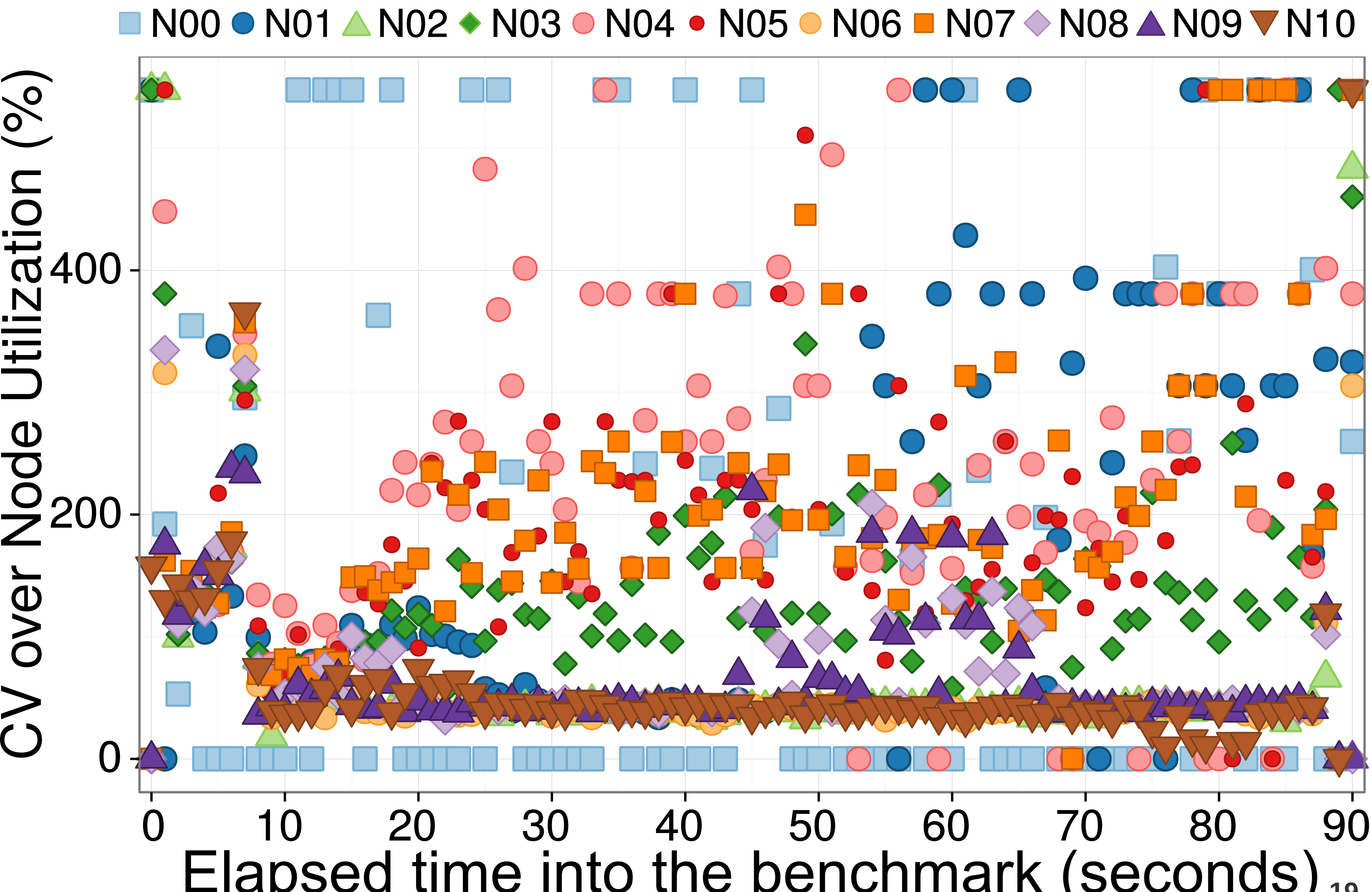
- Remarks
  - Streaming engines employ different orchestration strategies
  - Users are only given with high-level configuration options
    - Users cannot select number of actor instances nor assign actor instances to nodes

Storm: CV of vCPU Utilization per Node

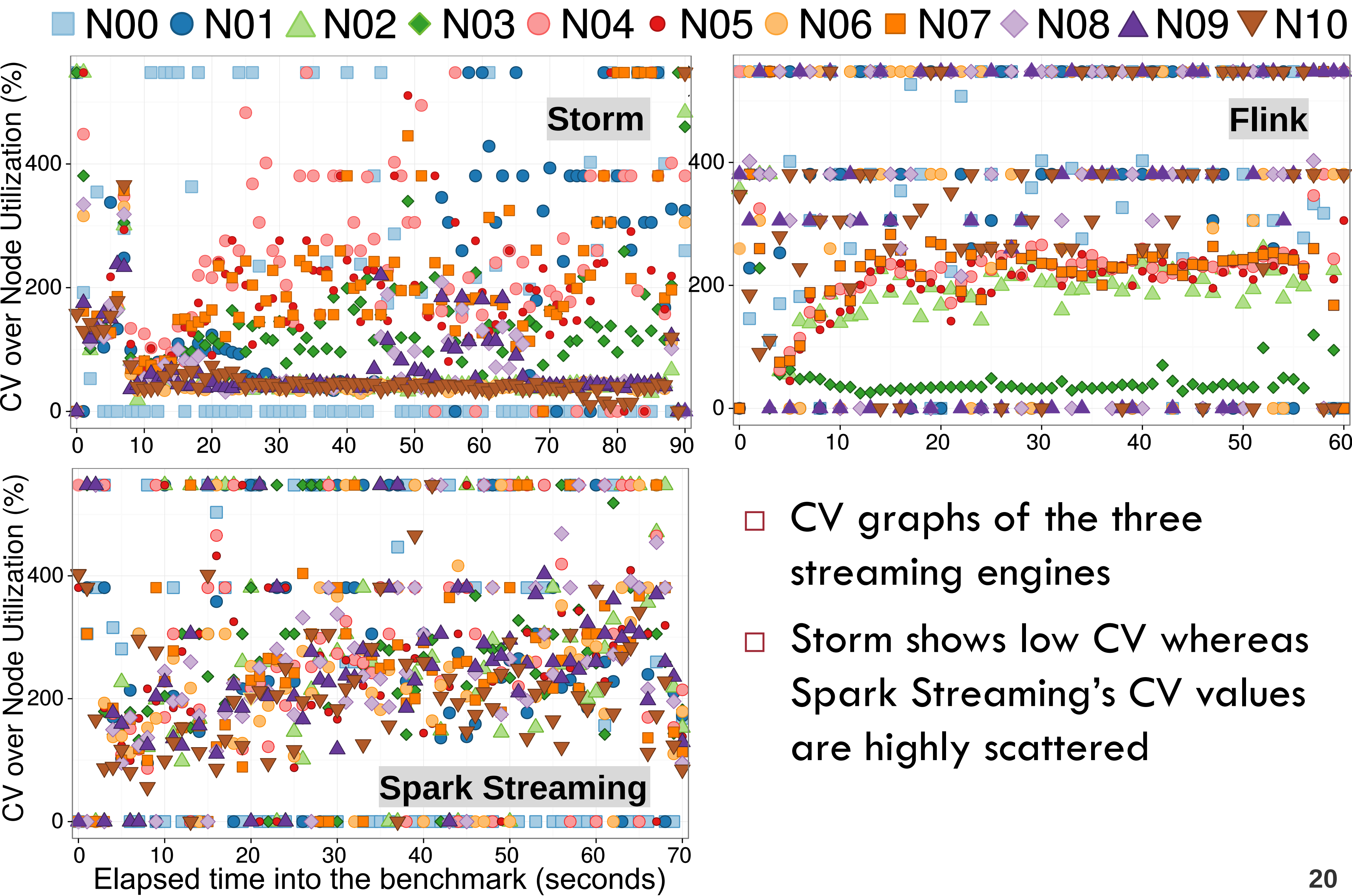


- **High CV:** the vCPUs of a node are utilized to largely varying degrees.
- **Low CV:** the vCPUs of a node are utilized to the same degree.
- Ideal:
  - high average vCPU utilization
  - low CV
  - ***"all vCPUs are humming"***

Storm: CV of Node vCPU Utilization



Comparing Streaming Engines



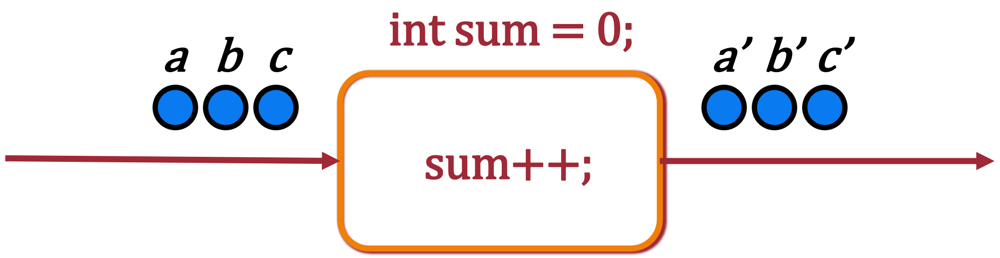
- CV graphs of the three streaming engines
- Storm shows low CV whereas Spark Streaming's CV values are highly scattered

Benchmark 2: Trend Detection

- A popular streaming analysis used in social network services and search engines
  - discovering, measuring, and comparing changes in time series data from online user interactions
- Point-by-point Poisson model
  - example: keyword trending for a soccer match
  - the probability of observing a particular count of some quantity, when many sources have individually low probabilities of contributing to the count
  - most effective for finding trending keywords from small sets of time series data
- Example data set
  - Wikipedia's actual page traffic data collected for three months (150GB, 67M tuples)

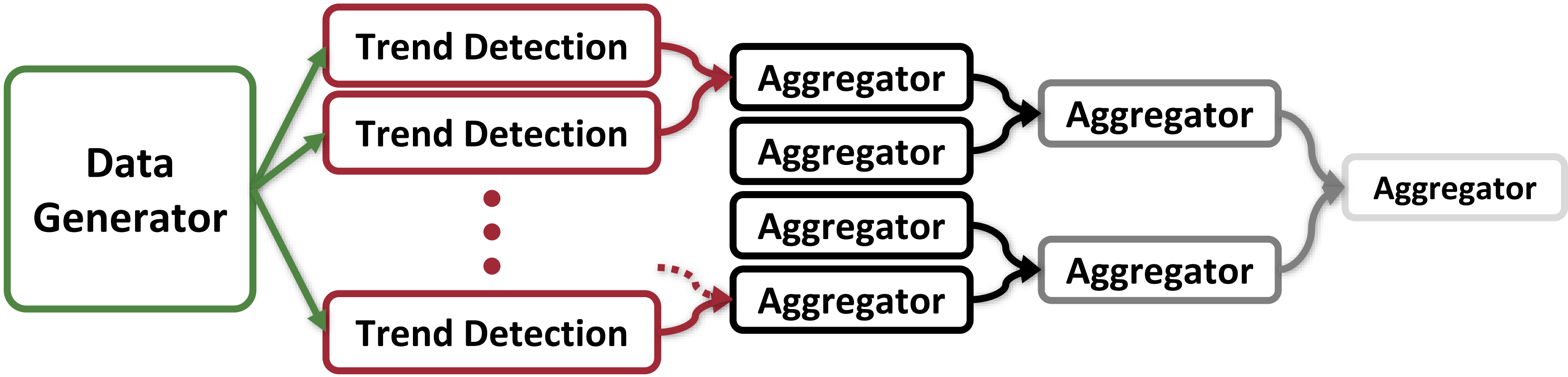
Cloud Trend Detector

- Implemented with Storm API and Java
- Stateful versus stateless actor



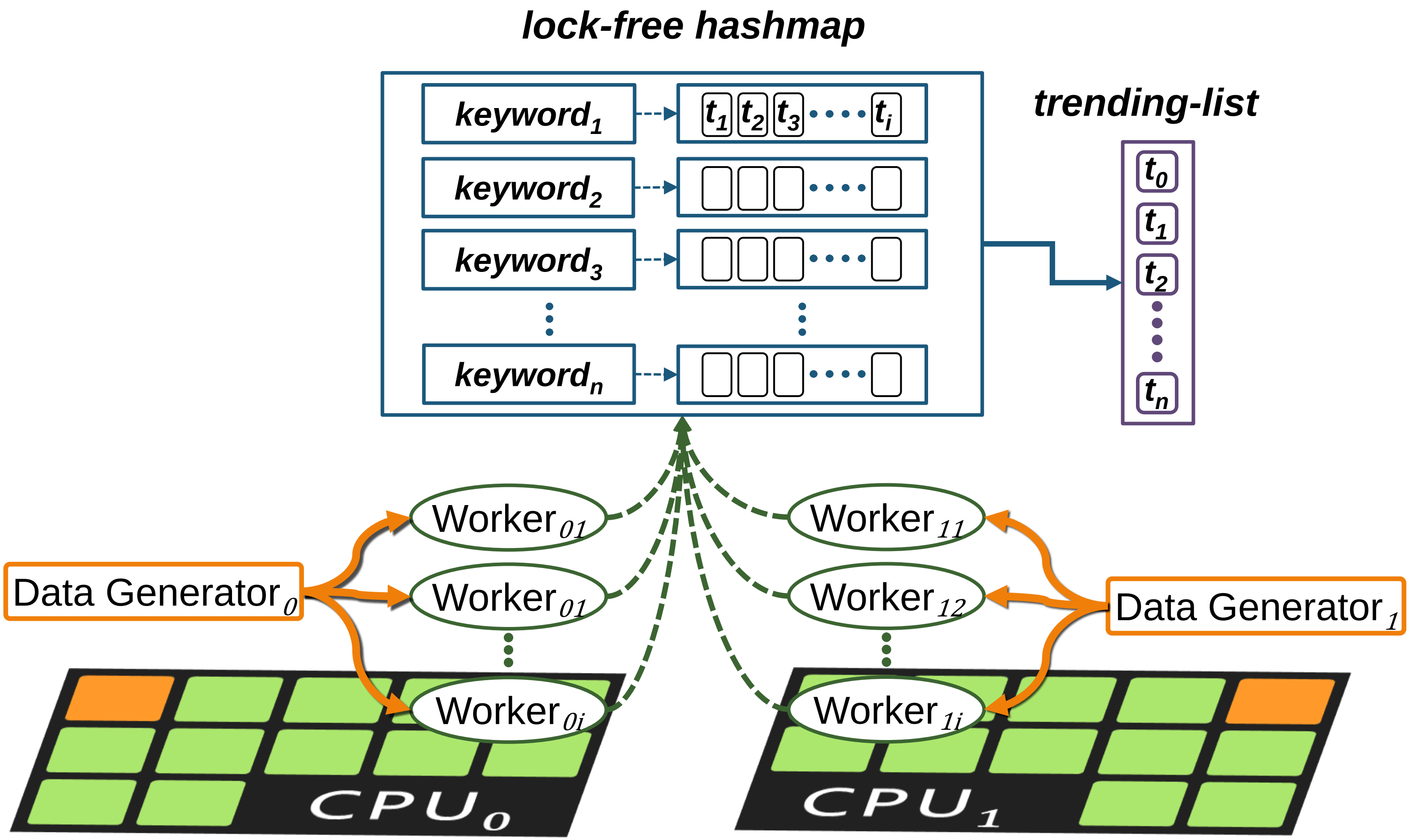
  - stateful: a global data structure is required to maintain all states
  - stateless: remove global data-structure from a Cloud application to avoid **expensive communication overhead**
- Re-designing the trend detector to become **stateless:**
  - introducing speculative trend detection
  - parallel reduction algorithm is a natural fit for this purpose

Parallel Reduction Algorithm



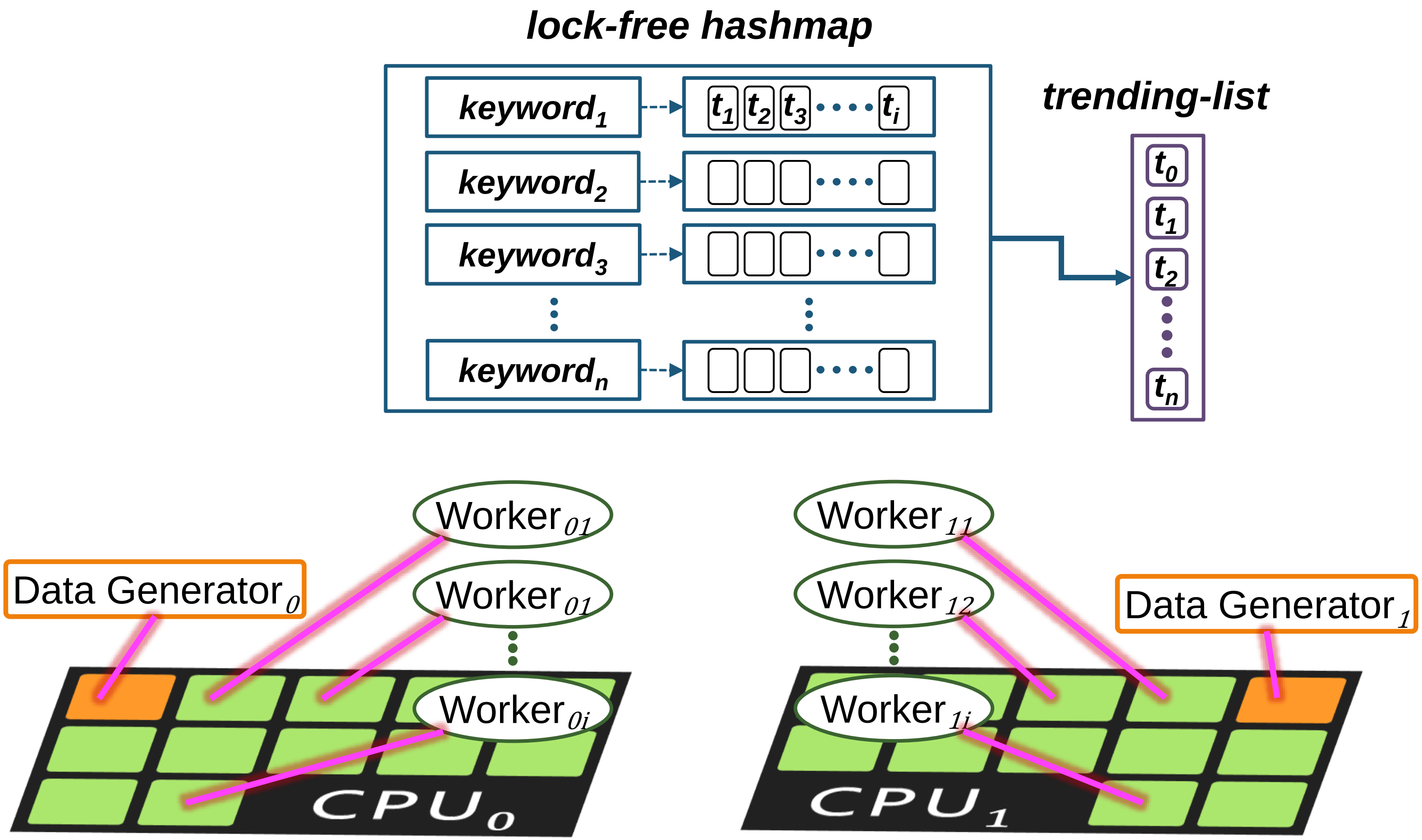
- *N*-layer Cloud trend detector with parallel reduction
  - Cloud trend detector is created dynamically at the beginning of the run-time with given number of layers
  - Each trend detection node receives partial stream and evaluate each keyword's trendiness.
  - Each aggregator node performs evaluation of trendiness from the results of the two precedent nodes.

Single-node Trend Detector



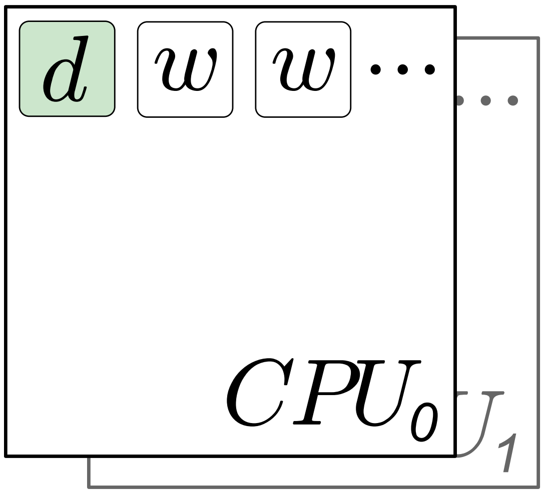
- **Stateful** Trend Detection
  - Implemented in C++ for a shared-memory multicore computer

Thread-to-core Allocation



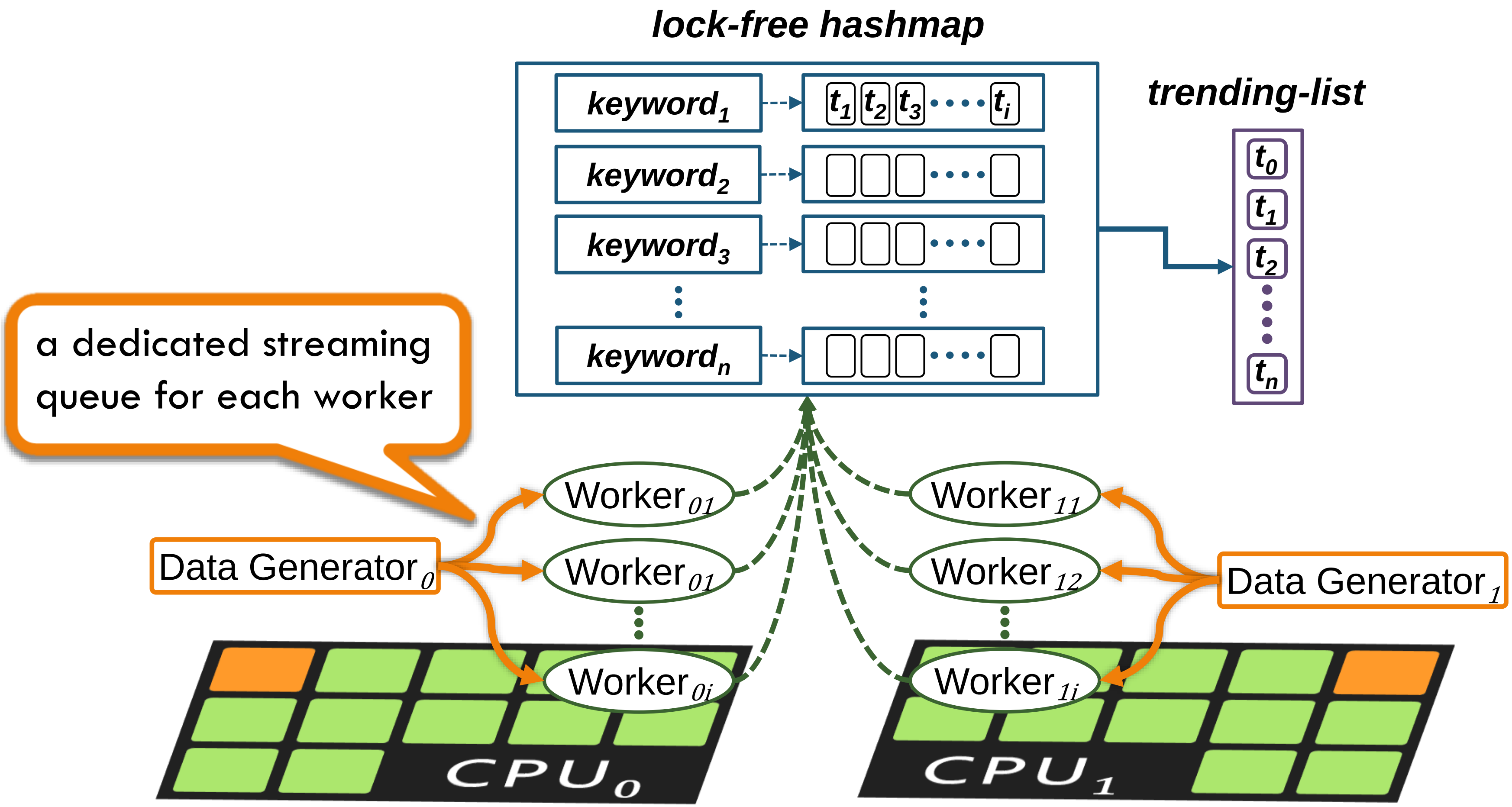
- Each thread is allocated to a single, dedicated core on a CPU

Thread-to-core Allocation (cont.)



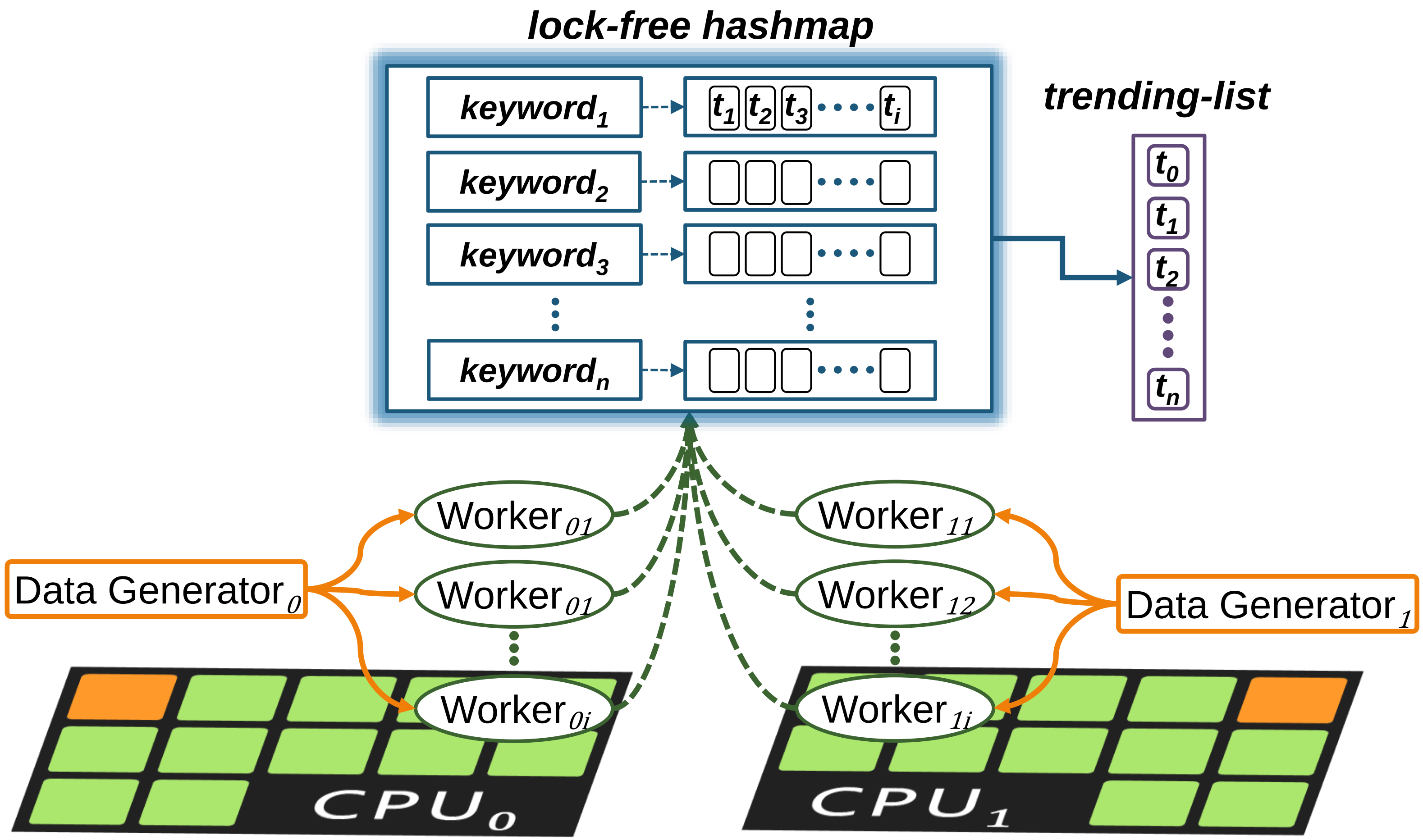
- Thread-to-core allocation
  - one datagenerator d is employed per CPU
  - remaining cores are filled with worker threads w
  - each worker thread has a dedicated streaming queue to receive tuples from a datagenerator
  - the worker threads receive tuples from the datagenerator thread pinned on the same CPU

Lock-free SPSC Queue



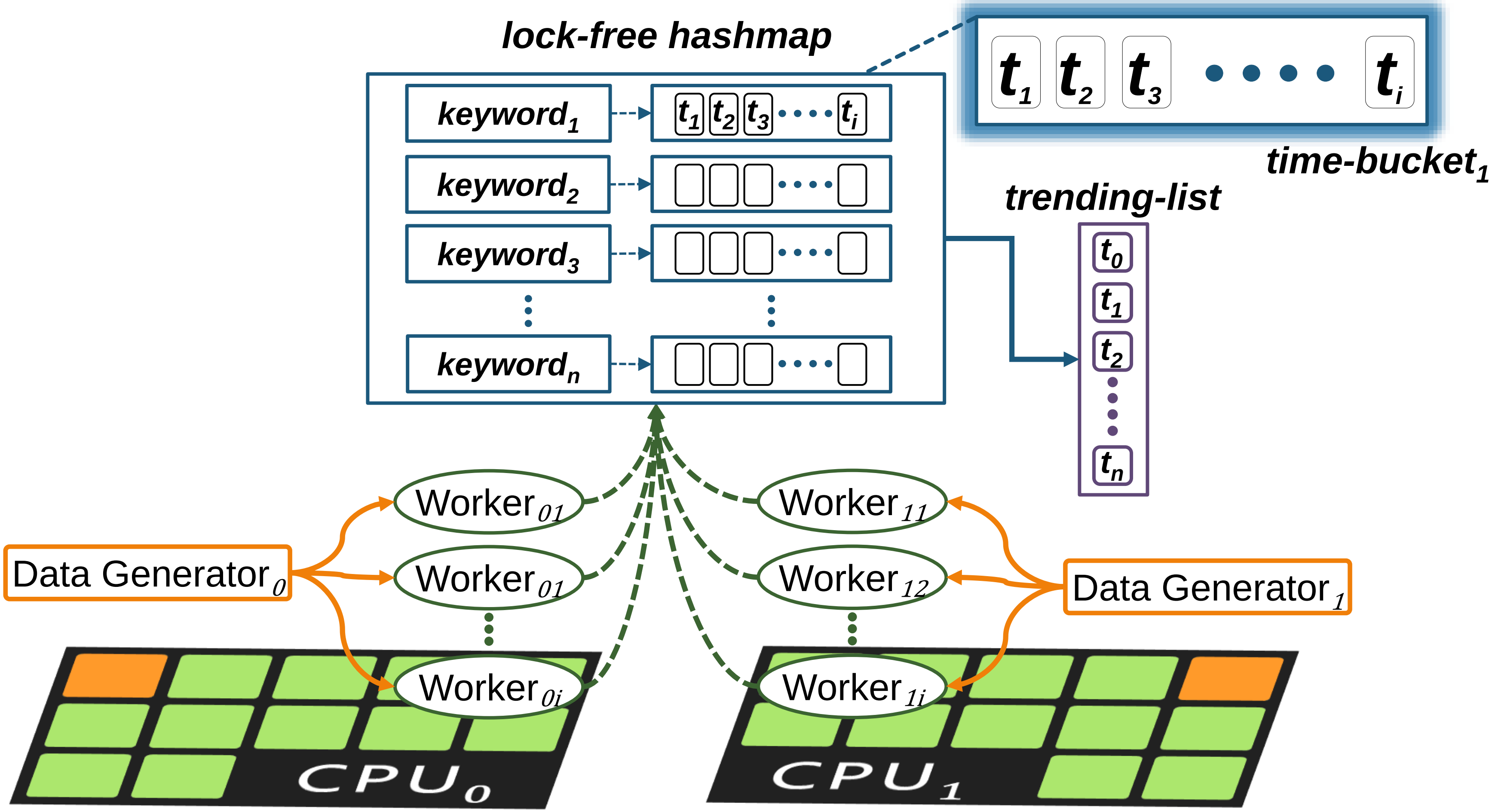
- Lock-free single-producer-single-consumer queues are employed for each and every worker thread

Lock-free Hashmap



- Lock contention is removed by employing a lock-free hashmap
- Correctness is guaranteed by storing all timestamps

Timebucket Evaluation

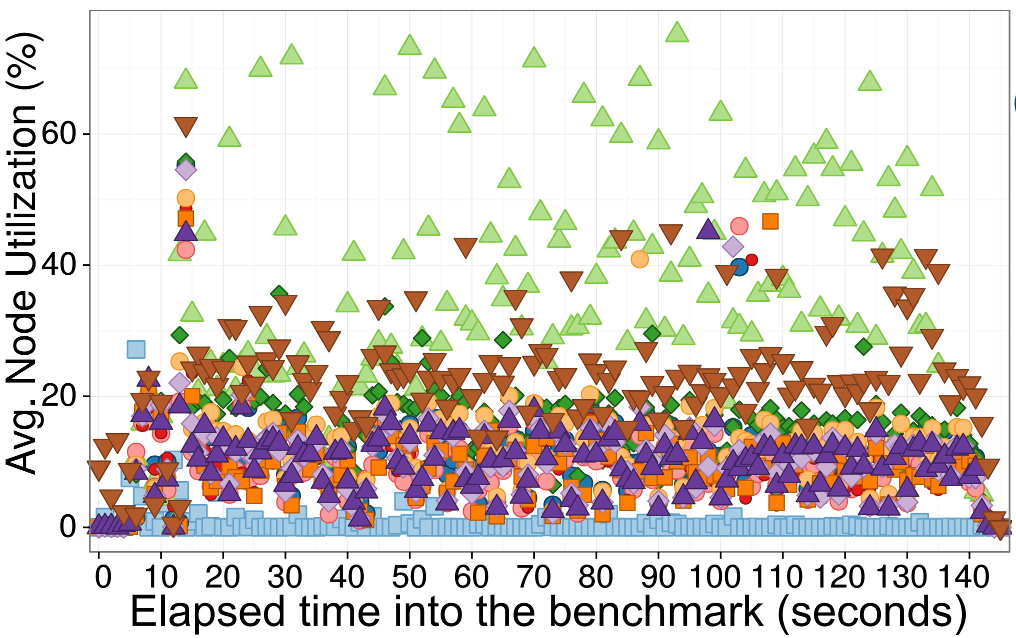


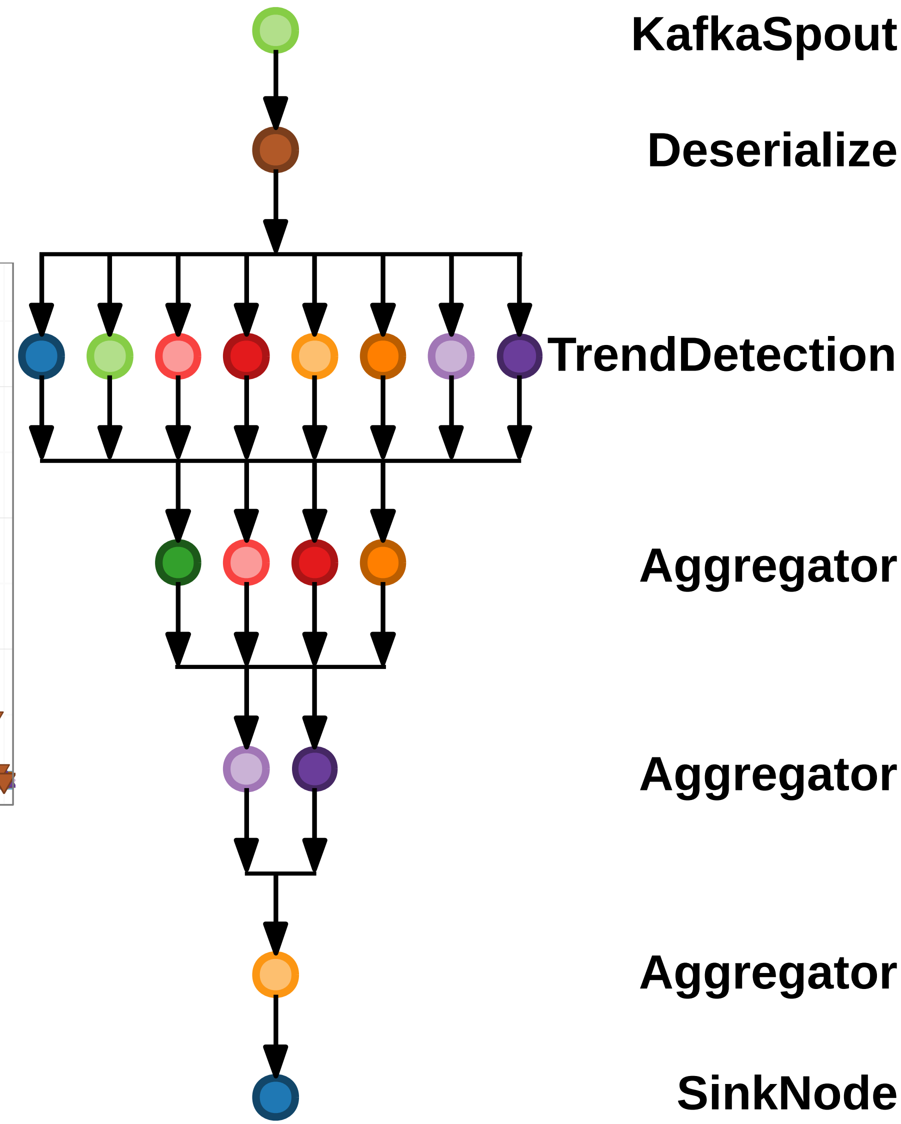
- All timestamps of received keywords are stored.
- Trendiness of a keyword is evaluated periodically.

Experimental Results: Single-node Trend Detector

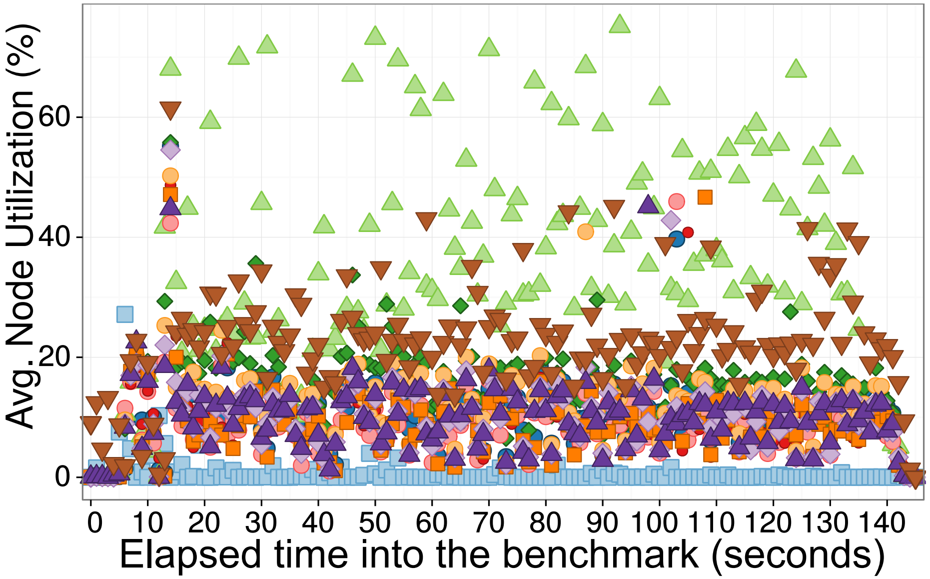
- Single-node trend detector
  - 2 Intel Xeon E5-2699 v4 CPUs (22 physical cores per CPU)
  - 512 GB RAM
- Achieved throughput: 3,217,432 tuples/s

Cloud Trend Detector Orchestration





Utilization & Throughput

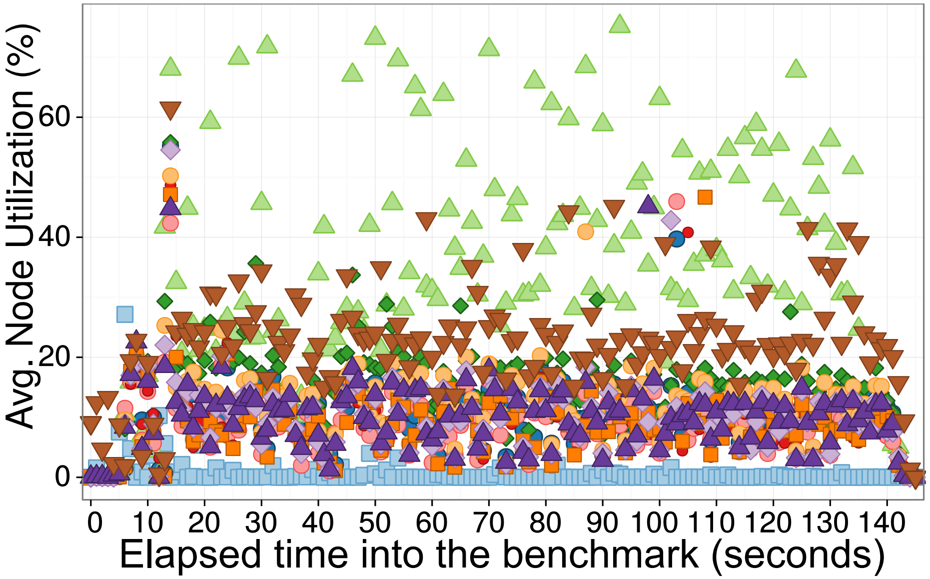


- Cloud trend detector shows under-utilized Cloud nodes
- Comparison of Cloud & single-node trend detectors

Trend Detection

Type	Cloud	Single-node
Participating node counts:	30	1
Throughput (tuples/s):	72,499	3,217,432
Implementation time:	2	3

Utilization & Throughput



- Cloud trend detector shows under-utilized Cloud nodes
- Comparison of Cloud & single-node trend detectors

Trend Detection

Type	Cloud	Single-node
Participating node counts:	30	1
Throughput (tuples/s):	72,499	3,217,432
Implementation time:	2	3

Conclusion

- Big Data streaming platforms exhibit:
  - Low throughput
  - Disadvantageous orchestration decisions:
    - over-subscribed nodes (Flink), under-utilized nodes (all)
    - inconsistent vertical scaling (Flink), inefficient horizontal scaling (Storm)
- Our stateful lock-less single-node trend detector features:
  - vertical scaling on a shared-memory multicore computer
  - it outperformed its Cloud-based counterpart by two orders of magnitude higher throughput
- Envisioned future work:
  - Determine and resolve main bottlenecks of streaming platforms
    - Orchestration? scaling? communication latencies? JVM-induced overhead?
  - Attempt efficient vertical scaling for Cloud applications (inspired by Flink's orchestration).
  - Orchestration of streaming applications for the Cloud

Acknowledgements

Research supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning under grant NRF2015M3C4A7065522.

Thank you…