# Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data

6.886

Joana M. F. da Trindade

Apr 24th 2018

# Motivation

Several data sources generate streaming data continuously

- Sensors, financial markets, social networks, urban sensing, server logs

Some data analytics applications on top of it have near real-time use cases

- Fraud detection, abuse detection, other monitoring use cases, etc

How can we provide support for efficient streaming graph analytics use cases?

- Supporting efficient streaming queries -- instead of just computation?
- Supporting both fast updates and fresh data views w/o consistency issues ?

# How does this paper differ from previous work?

Prior graph streaming systems focused on computations instead of queries:

- PageRanking, Naiad, Spark Streaming, TimeStream

Stream graph computation vs stream graph querying, according to authors

- First favors serialized computation over large portion of streaming data
- Latter focuses on concurrent queries over specific set of both streaming and stored data

Prior systems were also stateless: no integration between streaming data for concurrent queries or no querying of persistent store to base knowledge

Stream processing engines: not applicable, as they use relational model; authors also claim that their performance is significantly lower than their system’s

# What do authors mean by “stateful” querying?

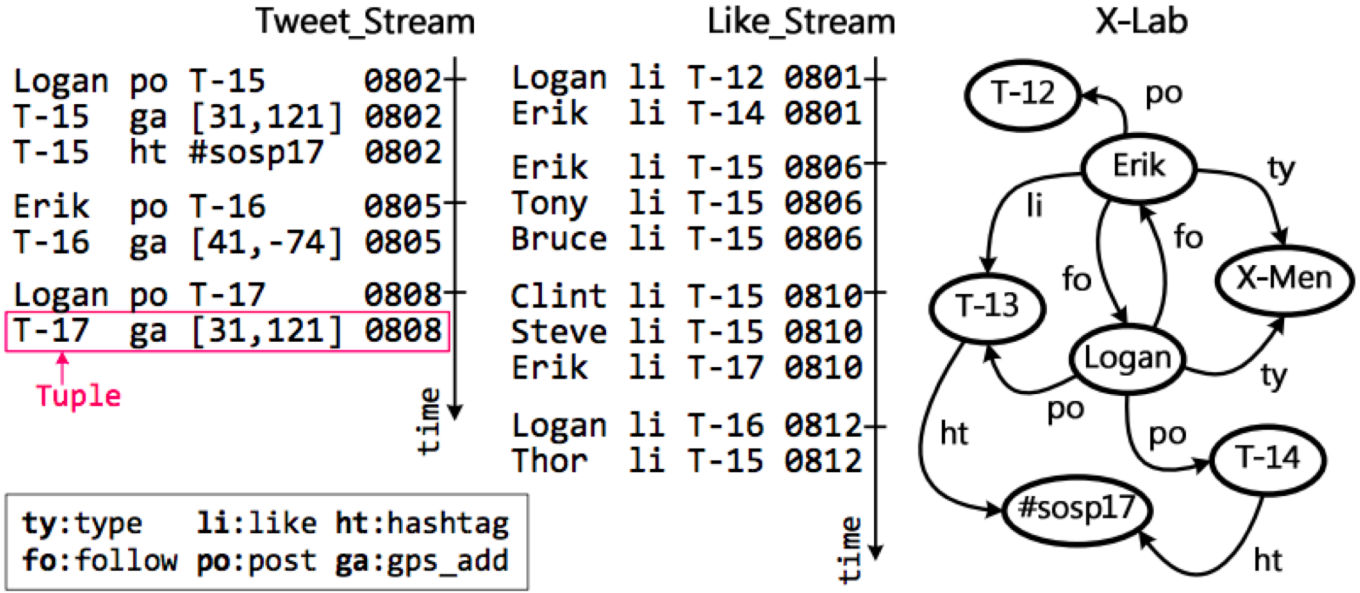



**Fig. 1: A sample of streaming and stored data in social networking.**

# What do authors mean by “stateful” querying?

```
SELECT ?X


FROM X-Lab
WHERE {
  Logan po ?X
  ?X    ht #sosp17
  Erik  li ?X
}
```

(a) One-shot Query

```
REGISTER QUERY Qc
SELECT ?X ?Y ?Z
FROM Tweet_Stream [RANGE 10s STEP 1s]
FROM Like_Stream  [RANGE  5s STEP 1s]
FROM X-Lab
WHERE {
  GRAPH Tweet_Stream  { ?X po ?Z }
  GRAPH X-Lab         { ?X fo ?Y }
  GRAPH Like_Stream   { ?Y li ?Z }
}
```

Query Condition

(b) Continuous Query

**Fig. 2: A sample of one-shot ($\mathbf{Q}_S$) and continuous ($\mathbf{Q}_C$) queries.**

# What do authors mean by “stateful” querying?

```
SELECT ?X


FROM X-Lab
WHERE {
  Logan po ?X
  ?X    ht #sosp17
  Erik  li ?X
}
```

(a) One-shot Query

```
REGISTER QUERY Qc
SELECT ?X ?Y ?Z
FROM Tweet_Stream [RANGE 10s STEP 1s]
FROM Like_Stream  [RANGE  5s STEP 1s]
FROM X-Lab
WHERE {
  GRAPH Tweet_Stream  { ?X po ?Z }
  GRAPH X-Lab         { ?X fo ?Y }
  GRAPH Like_Stream   { ?Y li ?Z }
}
```

Query Condition

(b) Continuous Query

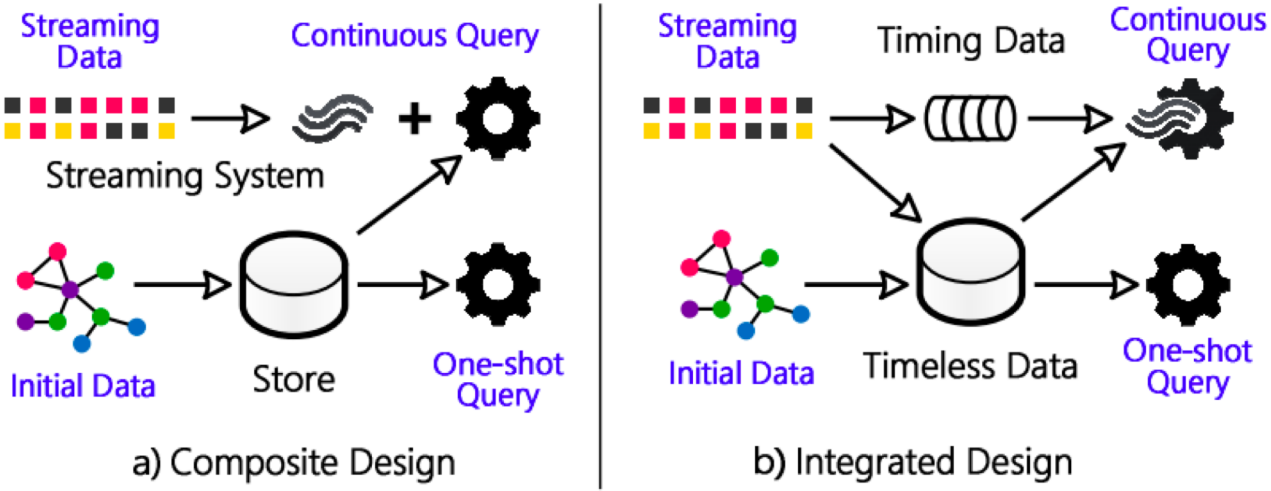
Join between 3 graph streams

**Fig. 2: A sample of one-shot ($Q_S$) and continuous ($Q_C$) queries.**

# Conventional approach vs authors approach

Goal: support near real-time stateful streaming queries over linked data, where each query may access partial data from different streams

"conventional" approach, AKA "composite" design




Authors's approach AKA "integrated" design

**Fig. 3: A comparison between composite and integrated design.**

# Drawbacks of conventional approach

Cross-system Cost -> ~40% execution time wasted due to data transformation and transmission

Inefficient Query Plan -> Semantic gap between the two systems impair query optimization

Limited Scalability -> Stream processing systems dedicate all resources to the improve performance of a single job

In summary -> high latency, low throughput

# Advantages of authors's approach over conventional approach ("composite" design)

Eliminates cross-system cost -> no data transformation needed across stores

More efficient query optimization -> no semantic gap across different systems, single global optimizer

Better Scalability -> shares data across multiple queries, can leverage that for better scalability (though you'd have more chances here for inconsistencies, but authors deal with that)

In summary -> lower latency, higher throughput than existing systems

# Challenges when implementing “integrated” design

Hybrid Store

- efficiently handle streaming data and fast-evolving stored data

Indexing

- fast path to access streaming data in a certain time interval

Consistency

- system provides consistent data views through decentralized vector

timestamps and bounded snapshot scalarization

# Authors's approach: "integrated" design (Wukong+S)

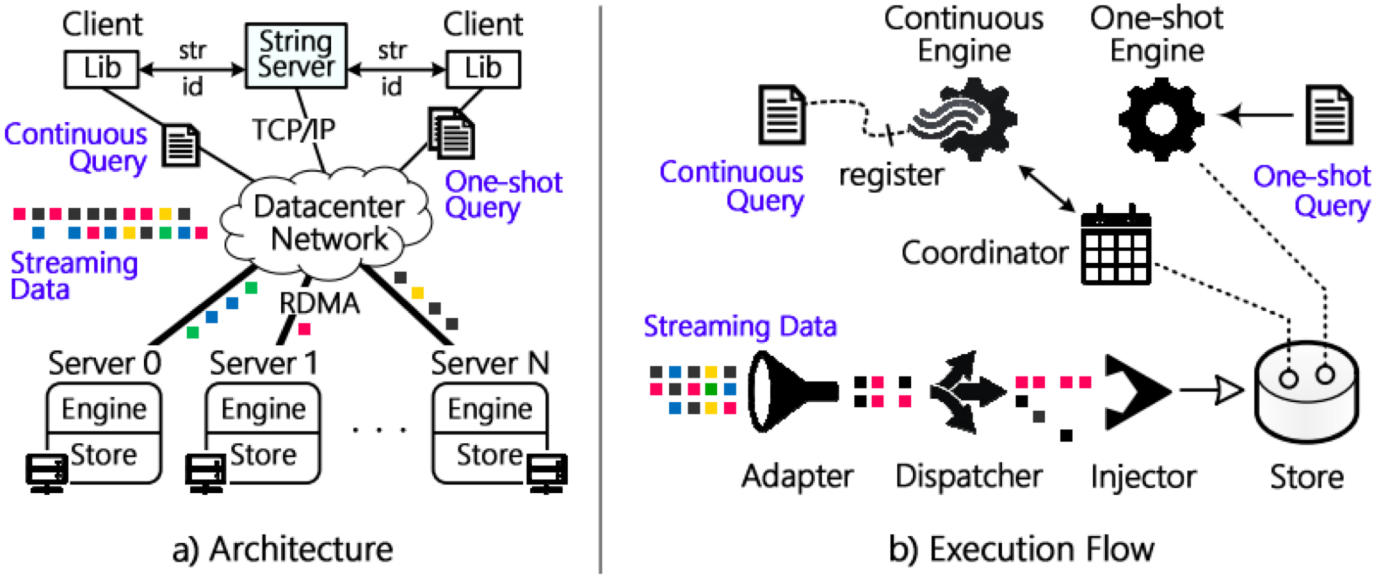



**Fig. 5: The architecture and execution flow of Wukong+S.**

# Hybrid store: persistent vs transient

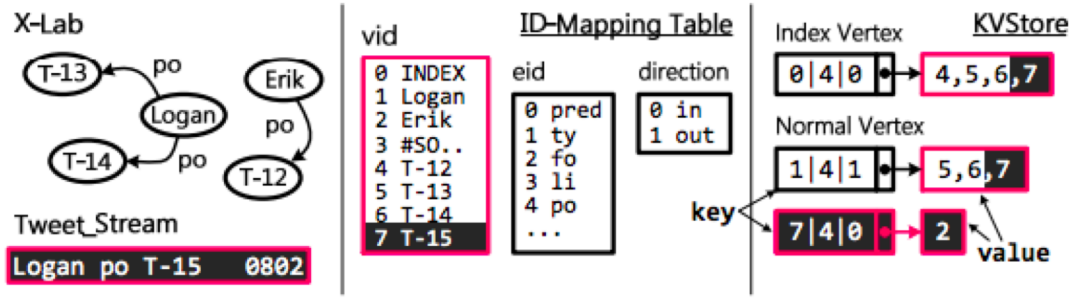



**Fig. 6: The injection of a triple on continuous persistent store.**

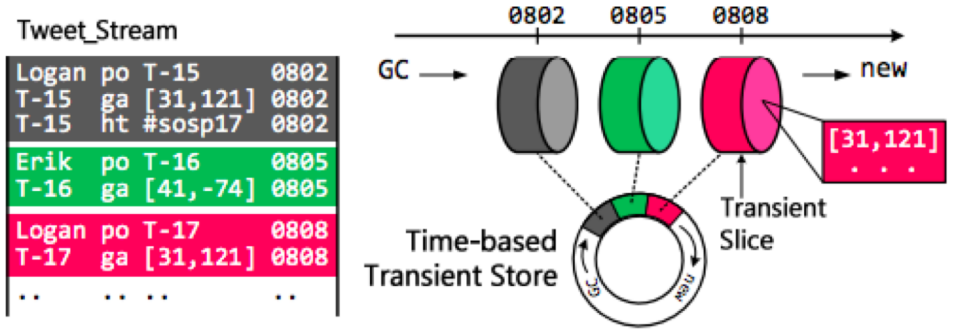



**Fig. 7: The design of time-based transient store in Wukong+S.**

Different data doesn't interfere with each other

Each optimized for different acess patterns

Timeless data: continuous persistent store

Temporal data: time-based transient store

# How does it provide consistency across views?

Consistent views over dynamic data AND with memory efficiency:

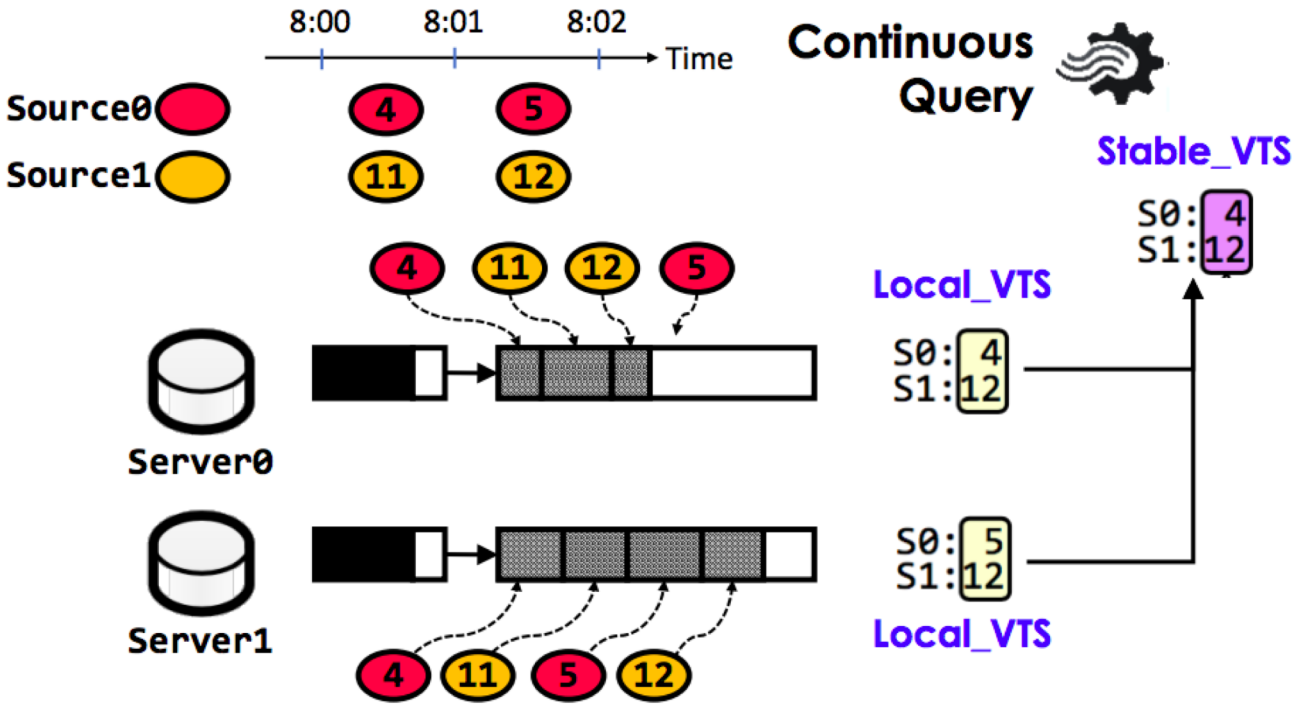
Streaming data contains order information

Early output from a stream source should always be visible before later output

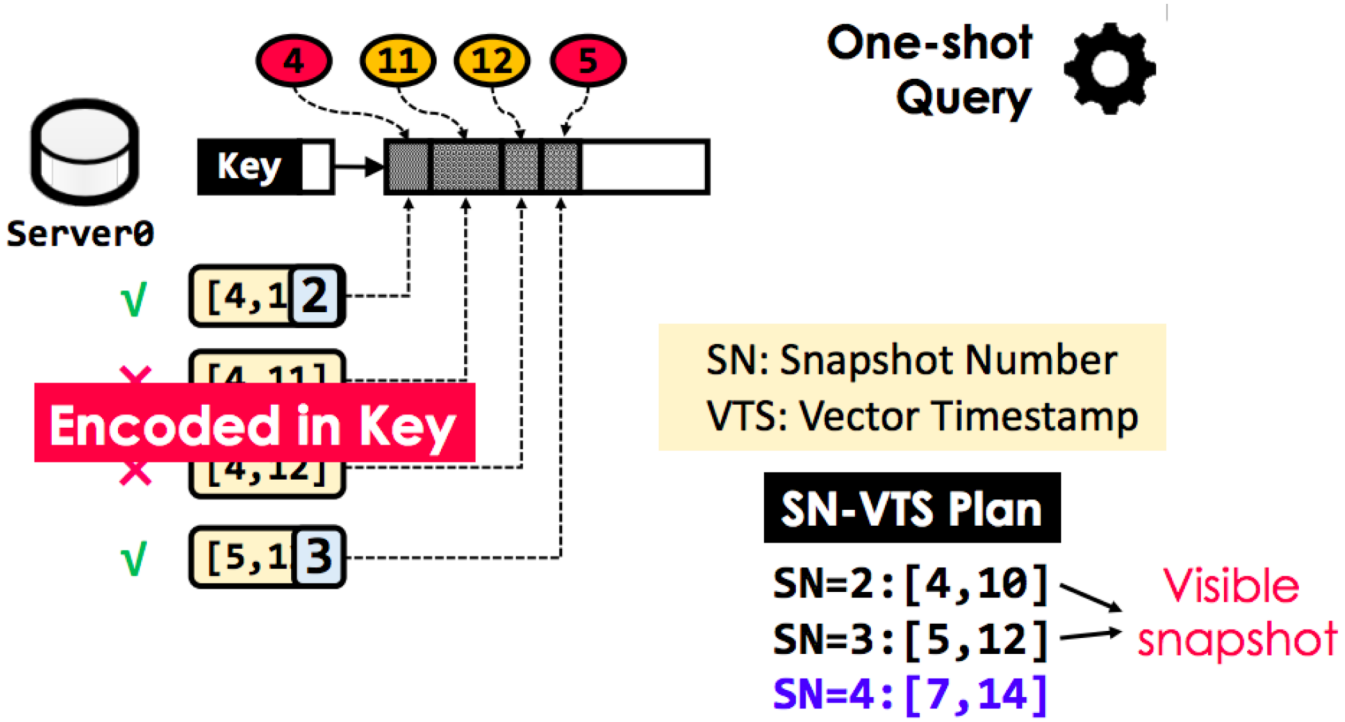
No order relation across data sources

Key intuition: use vector clocks for decentralized vector timestamps across data partitions

# Decentralized vector timestamps in Wukong+S



# Snapshot Scalarization in Wukong+S



# Consistency Model

Provides “prefix integrity” for both continuous and one-shot queries

- Same consistency model as Structured Streaming and Apache Spark Streaming
- *“at any time, the output of the application is equivalent to executing a batch job on a prefix of the data” (databricks blog post on structured streaming, July 2016)*
- Output tables are **always consistent** with all the records in a prefix of the data

Continuous queries

- Uses distributed vector timestamps to ensure ordering of streaming data arrival equals order of visibility to queries

One-shot queries

- Mapping from VTS to SN (snapshot scalarization) preserves order of VTS
- Assumes timestamps in each stream arrive in monotonically non-decreasing order, and hence doesn’t need to handle out-of-order issues in input streams

# Fault Tolerance

Provides at-least-once semantics for continuous queries, e.g., 2 executions on the same window of streams are possible in case of failure (can be addressed by client)

Query engine layer -> logs all messages to persistent storage

Data store layer -> incremental checkpointing by periodic logging on the background

Recovery -> uses stream index to locate data since last checkpoint; local and stable VTS are also stored, and this is used to notify stream sources to flush data accordingly

# Leveraging RDMA (Wukong)

Stream index -> treated as location cache, providing another layer of indirection to fast access streaming data

Normal remote access to KV pair requires at least two RDMA reads: read key (lookup) and read value

Wukong+S accumulates stream index for each stream within one machine

- Query only needs to use one RDMA read to retrieve KV pair since stream index is already locally accessible
- Assumption: stream index is usually much smaller than data -> feasible to accumulate all stream indexes for one stream in one machine

# Evaluation

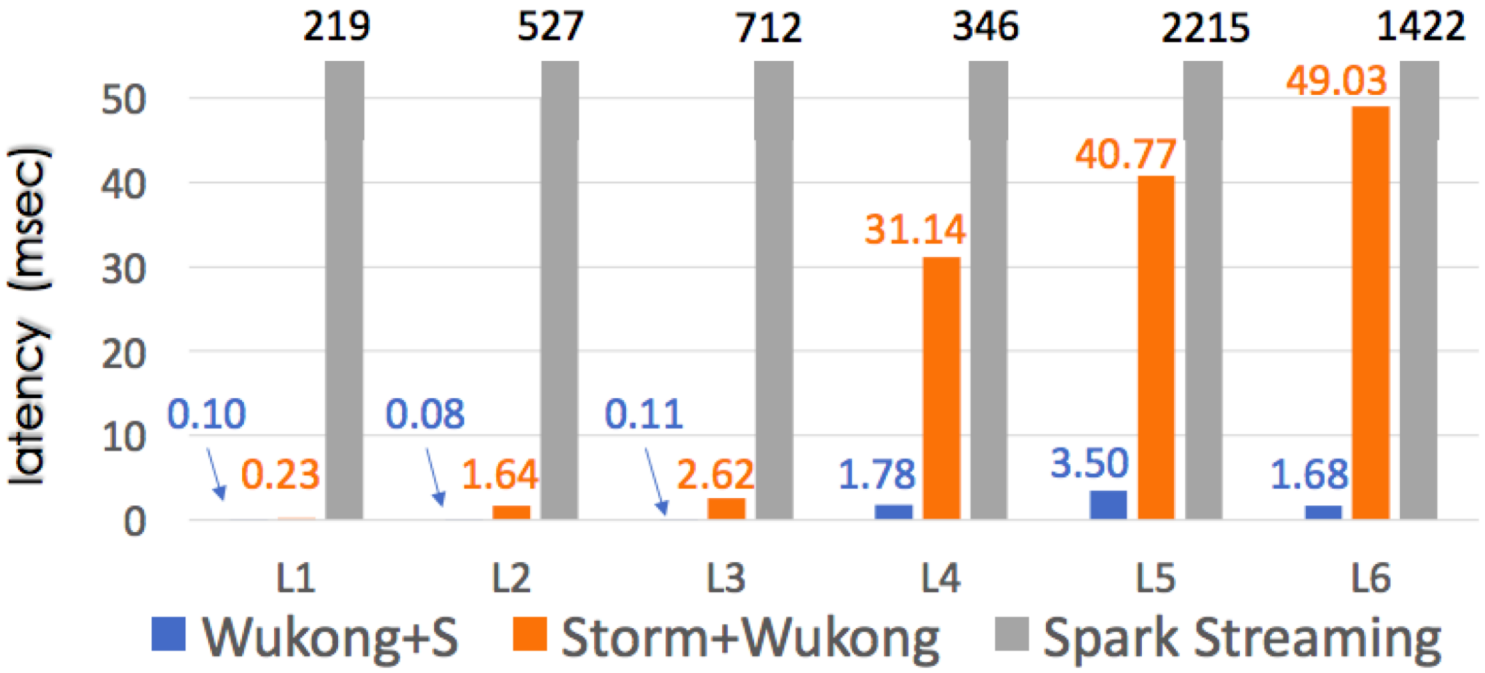
Baseline: 6 state-of-the-art systems -> CSPARQL-engine, Heron+Wukong, Storm+Wukong, Spark Streaming, Spark Structured Streaming, Wukong/ext

Platforms: a rack-scale 8-machine cluster, each with 2 12-core Intel Xeon, 128GB DRAM, w/ RDMA Mellanox 56Gbps InfiniBand NIC, 40Gbps IB Switch

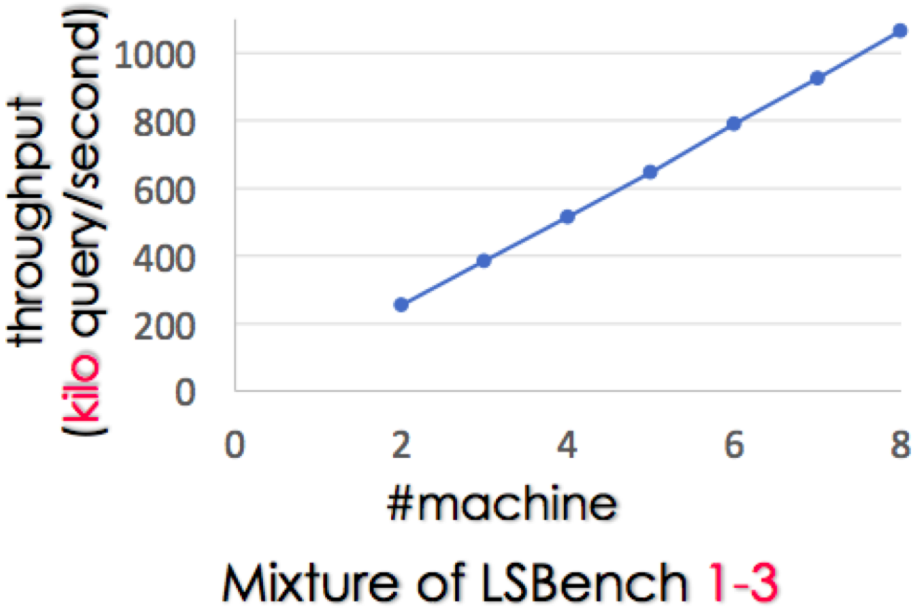
Benchmarks

- LSBench: Social Networking Benchmark w/ 3.75B initial stored data & 5 streams totally 134K tuple/second stream
- CityBench: Smart City Benchmark w/ 11 real-world data streams

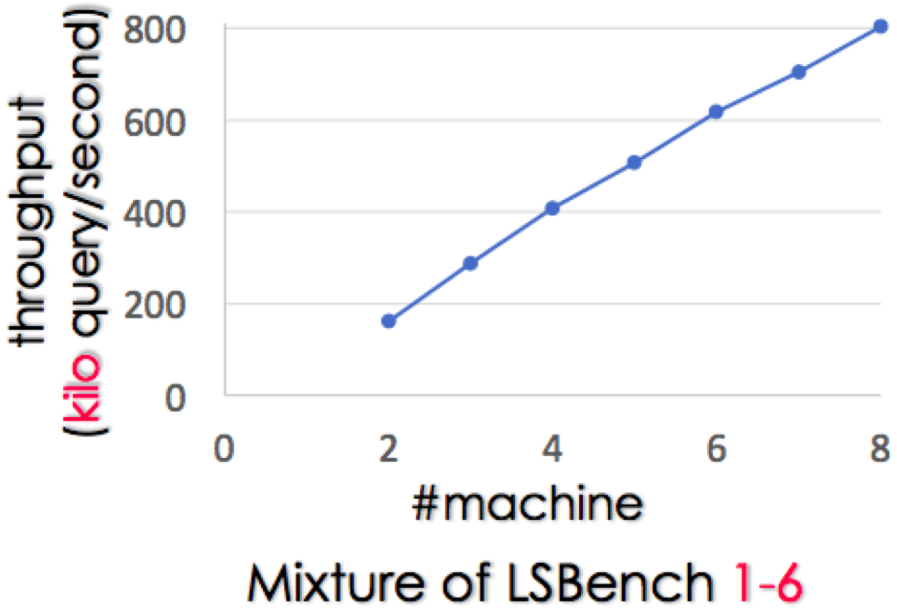
# Evaluation: single query latency



# Evaluation: throughput




Mixture of LSBench 1-3




Mixture of LSBench 1-6

# Evaluation: other experiments in the paper

Influence of different stream rate

Data insertion latency

Performance of one-shot queries

Memory consumption

Fault-tolerance overhead

# Conclusion

Authors propose and implement a new design for a distributed stream query engine that supports stateful queries over graph streams

Design primarily relies on a hybrid graph store engine -> different stores for for continuous persistent graphs and time-based transient graphs

Addresses consistency across data views using vector timestamp and snapshot scalarization

Lower latency and higher throughput than state-of-the-art systems for streaming computation