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

Yunhao Zhang, Rong Chen, Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contacts: {rongchen, haibochen}@sjtu.edu.cn

Abstract

Applications like social networking, urban monitoring and market feed processing require stateful stream query: a query consults not only streaming data but also stored data to extract timely information; useful information from streaming data also needs to be continuously and consistently integrated into stored data to serve concurrent and future queries. However, prior streaming systems either focus on stream computation, or are not stateful, or cannot provide low latency and high throughput to handle the fast-evolving linked data and increasing concurrency of queries.

This paper presents Wukong+S[1], a distributed stream querying engine that provides sub-millisecond stateful query at millions of queries per-second over fast-evolving linked data. Wukong+S uses an integrated design that combines the stream processor and the persistent store with efficient state sharing, which avoids the cross-system cost and sub-optimal query plan in conventional composite designs (e.g., Storm+Wukong). Wukong+S uses a hybrid store to differentially manage timeless data and timing data accordingly and provides an efficient stream index with locality-aware partitioning to facilitate fast access to streaming data. Wukong+S further provides a decentralized vector timestamp with bounded snapshot scalarization, which scales Wukong+S with nodes and massive queries at efficient memory usage.

We have designed Wukong+S conforming to the RDF data model and Continuous SPARQL (C-SPARQL) query interface and have implemented Wukong+S by extending a state-of-the-art static RDF store (namely Wukong). Evaluation on an 8-node RDMA-capable cluster using LSBench and CityBench shows that Wukong+S significantly outperforms conventional system designs (e.g., CSPARQL-engine, Storm+Wukong, and Spark Streaming) for both latency and throughput, usually at the scale of orders of magnitude.

[1] The source code and a brief instruction of Wukong+S are anonymously available at https://github.com/DistributedRSP/WSP.

SOSP'17 October 28–31, 2017, Shanghai, China

DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Much information is most valuable when they first appear such that timely processing it is super-critical. The real-time demand has driven the design and implementation of a number of stream processing systems [20, 29, 32, 36, 37, 42, 45]. Yet, the exponentially increasing data rate is constantly pushing the limit of processing capability. For example, the Options Price Reporting Authority (OPRA)[2], which gathers market data feed by aggregating all quotes and trades from the options exchanges, has observed a peek messages rate at 10,244,894 per second in March 2015 and the rate is continuing doubling annually [8]. Similarly, the peak rate of new tweets per second has reached 143,199 in August, 2013.

With the increasing volume of streaming data and stored data, it is vitally important to timely query useful information from them. For public datasets and streams, there may be a massive number of users registering different stream queries, making it necessary to support highly concurrent queries. Moreover, streaming data usually contains tremendous useful information; such data should be consistently and instantly integrated to the stored knowledge base for future continuous and one-shot queries[3].

However, many prior systems target for stream computation (like PageRanking) over changing datasets (like Naiad [34], Spark Streaming [46] and TimeStream [37]). Online computation differs from online query such that the former usually favors serialized computation over a large-portion of streaming data, while the later focuses on concurrent queries over a specific set of both streaming and stored data. Most prior systems also do not consistently integrate streaming data for concurrent queries or do not query the persistent store for the knowledge base, and thus are stateless [10, 11]. While many streaming databases have clear semantics and well-defined SQL interfaces, they have inferior performance over fast-growing linked data when facing massively concurrent queries due to costly join operations [38, 47] and over-fitting semantics (e.g., ACID).

In this paper, we describe Wukong+S, a distributed stream querying engine that achieves millions of queries per-second and sub-millisecond latency. Wukong+S is stateful such that a query can touch both streaming and stored data,

[2] https://www.opradata.com/

[3] Continuous query is registered and periodically executed. One-shot query is a query that runs immediately and only once.

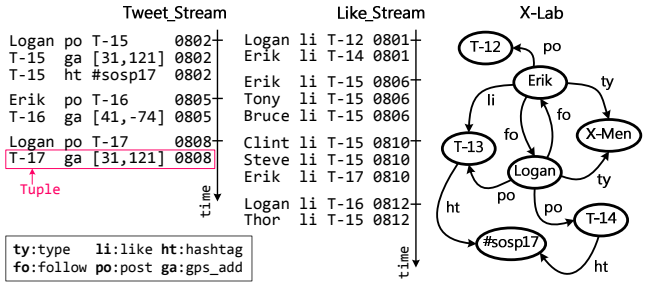

Fig. 1: A sample of streaming and stored data in social networking.

and the timeless data[4] in a stream is simultaneously integrated into the persistent store and consistently visible to concurrent queries. Wukong+S provides several features required by stateful streaming query, including querying over multiple varied-scale streams, supporting online aggregation over these streams and the stored data, while satisfying consistency requirements (e.g., early data item in a stream is visible first) of queries without sacrificing freshness. Wukong+S adopts a standard declarative streaming query interface called Continuous-SPARQL (C-SPARQL) [16] and represents the underlying linked data using the Resource Description Framework (RDF) [9], which is recommended by W3C and used by many knowledge bases [3, 17, 23, 35, 44]. This provides users with a familiar programming interface and data model and allows easy integration of existing RDF datasets for extended queries due to good interoperability [31].

The conventional way is to combine a stream processor with a static graph store. For example, the de-facto CSPARQL-engine [41] combines Esper [5] with Apache Jena [1]. However, performance measurement with both CSPARQL-engine and our better-performed implementation (Apache Storm [42] with a fast persistent store called Wukong [38]) shows that this design style incurs high communication cost and redundant insertions between the stream processor and the persistent store (§ 2.3). Such a *composite design* further leads to suboptimal query plans and limited scalability in term of queries due to inefficient data sharing among queries. Being aware of this, we design Wukong+S as an *integrated design* that intensively shares states and streaming data between the stream processor and the persistent store.

Wukong+S is made efficient with several key design choices. First, to respect data locality and minimize data transfer, Wukong+S uses a hybrid store comprising a time-based transient store and a continuous persistent store to provide differentiated management for timing and timeless data. Yet, the two stores share the same set of data such that each tuple is only inserted once. Second, Wukong+S provides a stream index for fast accesses to streaming data. The streaming data is sharded with locality-aware partitioning with some stream index dynamically replicated among nodes. This saves lookup cost and provides efficient load balance. Third, to provide consistent stream query over multiple varied-scale streams, Wukong+S uses a decentralized vector timestamp to derive a most-recent consistent state of streaming data insertion. Wukong+S uses a bounded scalarization scheme to projecting the vector timestamp into a scalar snapshot number, by coordinating updates from multiple streams to the underlying persistent store. Such a design scales Wukong+S with nodes and massive queries at efficient memory usage.

We have implemented Wukong+S by extending Wukong to support continuous updates to the RDF store and C-SPARQL queries. To our knowledge, Wukong+S is the first distributed C-SPARQL query engine. To demonstrate the effectiveness and efficiency of Wukong+S, we have conducted a set of evaluation using two popular streaming benchmarks: LSBench [27] and CityBench [13]. Evaluation on an 8-node RDMA-capable cluster shows that Wukong+S achieves up to 1.08M queries per second with 0.11ms median latency. Wukong+S significantly outperforms CSPARQL-engine [41], Apache Storm [42] over Wukong [38] and Spark Streaming [46] for both latency and throughput, usually at the scale of orders of magnitude.

2. Motivation

2.1 A Motivating Example: Social Networking

Many real-world applications[5], such as social networking [27] and urban computing [13], require *query processing systems* to interact with massive users and concurrently serve a large number of stream queries over high-volume stored and streaming data. The initially stored data is the base knowledge such as existing user profiles and friend relationships, while the streaming data is continuously generated from social network activities (e.g., tweets and likes).

Fig. 1 illustrates a simplified sample of data in social networking. The initially stored data (X-Lab) is represented as a directed graph, which captures the relationships (i.e., edges) among various entities (i.e., vertices). There are six categories of edges linking entities, namely, type (ty), follow (fo), post (po), like (li), gps_add (ga), and hashtag (ht). The currently *stored* data includes two members of X-Men (Logan and Erik) following each other, and several tweets (T-12, T-13, and T-14) with some hashtag (#sosp17) posted by them. There are two streams, Tweet_Stream and Like_Stream, which continuously update tweets with some interesting information (e.g., like, location and hashtag). The streaming data is represented as a time-based sequence of tuples, continuously produced by a data source (e.g., sensors or social network applications). Each tuple consists of a triple and its timestamp, like ⟨Logan, po, T-15⟩ 0802, which means "Logan" post (po) a tweet (T-15) at 0802.

[4] Timeless data contains the factual data without timing information and thus should be absorbed to update the knowledge base.

[5] https://www.w3.org/community/rsp/wiki/Use_cases

```
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 {                          Query Condition
  GRAPH Tweet_Stream  { ?X po ?Z }
  GRAPH X-Lab         { ?X fo ?Y }
  GRAPH Like_Stream   { ?Y li ?Z }
}
```

(b) Continuous Query

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

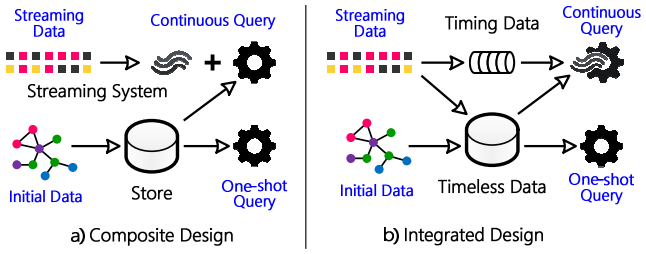

Fig. 3: A comparison between composite and integrated design.

A typical one-shot query and a continuous query (uniformly called stream queries) on the sample dataset is shown in Fig. 2. The one-shot query only consults the stored data to retrieve matching results once. For example, the one-shot query Q_S retrieves all tweets (?X) that were posted (po) by "Logan" with the hashtag (ht) "#sosp17" and liked (li) by "Erik". The current result only includes "T-13". To cope with dynamic knowledge, the continuous query considers both stored and streaming data to provide continuously renewed outputs. For example, the continuous query Q_C retrieves pairs of people (?X and ?Y) and tweets (?Z) from the streams (Tweet_Stream and Like_Stream) and the stored data (X-Lab) every second such that one person (?X) posts (po) a tweet (?Z) in the last 10 seconds that is liked (li) by another person (?Y) in the last 5 seconds whom he follows (fo). Suppose that the query is registered at 0810; the query result at 0810 includes "Logan Erik T-15" since the following tuples match all conditions.

⟨Logan, po, T-15⟩, 0802
⟨Logan, fo, Erik⟩
⟨Erik, li, T-15⟩, 0806

We summarize the workload characteristics as follows.

- **Data Connectivity.** The streaming data from many real-world applications like social networking and urban computing, are highly-connected to mirror the relationship among entities.
- **Query Statefulness.** A continuous query may retrieve both holistic knowledge from stored data and recently generated knowledge from multiple streams.
- **Data Integration.** To provide fresh results for one-shot queries, the stored data should continuously absorb useful timeless information (e.g., tweets and likes) from streams to derive an up-to-date information base.

2.2 Stream Querying System

Based on the workload characteristics, we propose a *stream querying system* that aims at supporting a massive amount of (continuous and one-shot) queries over streaming and stored data. There are several unique requirements that distinguish a stream querying system from other streaming systems.

Stateful. Stream query is *stateful*. More specifically, the continuous query will retrieve results not only from streaming data for the latest information, but also from stored data for holistic knowledge. Meanwhile, the one-shot query also demands continuous evolvement of the stored data to generate up-to-date knowledge.

Sharing among concurrent queries. All streaming and stored data will be shared by concurrent queries from massive users which register continuous queries or submit one-shot queries. Meanwhile, a single query may access multiple streams and stored data.

Partial data. A stream query usually touches only a small fraction of dataset matching the query conditions. Meanwhile, different queries may be interested in different parts of streaming and stored data.

Latency-oriented. To catch up the fast-changing linked data, like the stock market and quantitative trading, the latency should be the first-class citizen.

Alternative#1: Stream computation systems. Much prior work [7, 22, 26, 34, 37, 37, 39, 42] in system community focus on large-scale *stream computation*. First, stream computation is usually *stateless*, which usually only considers the streaming data and leaves the computation on stored data to other systems. Second, stream computation normally consumes the *entire* streaming data to generate new features like twitter user ranking [22] and crowdsourced traffic estimation [45]. Third, stream computation usually dedicates all resources to run a single computation task at a time. Finally, stream computation is usually done in a batch-oriented manner to maximize the resource usage (e.g., network bandwidth), which may compromise the latency [22, 45].

Alternative#2: Stream processing engine. There is also a much broader context including the work from database community [10, 11, 19, 30, 33, 40]. Such prior work also focuses on processing continuous queries over data streams, which, however, widely adopts the relational data model (e.g., table) and query operators (e.g., filter, join). Prior work [4, 38, 47] has summarized some inherent limitations and shown notably inferior performance of such an approach for querying highly-linked data.

2.3 Conventional Approach: Composite Design

Existing approach recommended by the community adopts the combination of a relational stream processor and a conventional query-oriented store. For example, CSPARQL-engine [41] combines Esper [5] with Apache Jena [1]. The one-shot query will be directly sent to the store, while the continuous query will be split into two parts (i.e., streaming and stored part specified by the GRAPH clause) which are

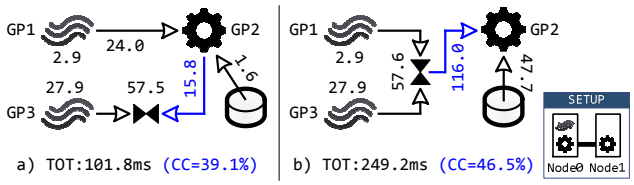

Fig. 4: The breakdown of execution time (ms) of (Q_C) on Storm+Wukong with different query plans. The blue number is cross-system cost (CC).

separately executed on stream processor and query-oriented store. The two parts of results will be joined to project the final results. To facilitate further discussions, we call such an approach *composite design* in this paper (Fig. 3(a)).

Such a design maximizes the reuse of existing systems and optimizations in decades. However, the composite design is still not completely *stateful*, since the one-shot queries always run on a static stored data without observing updates of timeless data from streaming data. Beside, there are several key deficiencies. Since CSPARQL-engine only runs on a single node with very limited performance (§6.2), we build a better-performed system by composing two state-of-the-art systems: Apache Storm [42] and Wukong [38]. The continuous query (Q_C) in Fig. 2 is used to illustrate the issues. As shown in the legend of Fig. 4, we carefully co-locate Storm and Wukong to reduce network traffic and run Storm on a single node as small workloads do not scale well on Storm (details in §6.2 and §6.9). The first and third query conditions (GP1 and GP3) in Q_C will run on Storm and the second (GP2) will run on Wukong.

ISSUE#1*: Cross-system cost (CC)* Since the composite design runs a single continuous query on two separate systems, the *cross-system cost* (CC) is an obvious overhead, which generally consists of the data transformation and transmission cost. Such a cost becomes significant when combining two high-performance components. In Fig. 4(a), the query results of GP1 (831 tuples) in Storm should be transformed into the format of query for Wukong. Similarly, the results of GP2 (9,532 tuples) will be transformed back and join with the results of GP3 (85,927 tuples) to get the final results (1,918 tuples) in Storm [6]. Consequently, around 40% execution time is wasted due to the cross-system cost.

ISSUE#2*: Sub-optimal query plan* To reduce the number of cross-system execution, the composite design can change the query plan, the execution order of query conditions. For Q_C, as shown in Fig. 4(b), Storm can execute GP1 and GP3 and join the results first. After that the intermediate results (83,099 tuples) will be sent to the Wukong to retrieve the final results (1,918 tuples). However, the total execution is even slower due to inefficient query plan, which is caused by insufficient pruning of intermediate results. Two separate systems also restrict the choice of an optimal query plan due to the lack of global information. On the other hand, prior work [4, 38, 47] shows that using rational query processing for highly-connected data intensively relies on costly join operations, which usually generate huge redundant intermediate data, especially when no sophisticated indexing. The lengthy execution time within Storm in Fig. 4 also confirms the claim. Unfortunately, to our knowledge, there is no streaming system supporting fast graph-exploration.

ISSUE#3*: Limited scalability* A continuous query usually touches only a small subset of streaming and stored data, making it not worthwhile to dedicate all resources to run a single query. However, most existing streaming systems are designed for computation and focus on scaling a single task execution. When facing millions of queries, such systems cannot efficiently share the same streaming data among different queries, since it is non-trivial to maintain the execution of millions queries with different window sizes and steps. On the other hand, duplicating streaming data to difference queries will significantly increase the memory pressure and limit the throughput of streaming systems.

3. Approach and Overview

Our approach: integrated design. Wukong+S uses an *integrate design* that focuses on targeting a mixture of continuous and one-shot queries over both streaming and stored data. The key principle of integrated design is to treat the data store as the first-class citizen, namely store-centric design. As shown in Fig. 3(b), the data store will be extended to the *integrated store* that can absorb the streaming data and concurrently serve both continuous and one-shot queries.

Our store-centric design naturally fits the stateful streaming query, where the continuous query can freely access both streaming and stored data, as well as the one-shot query can retrieve evolving stored data. This provides the benefit of low-latency and global semantic to generate an optimal query plan. Further, all queries can share the same streaming and stored data without redundant resource consumption, such as duplicated streaming data. Finally, concurrent queries interested in different parts of streaming and stored data can run concurrently, which further improves the efficiency and scalability of the system.

However, a naive design would lead to quick growth of space and cause excessive garbage collection operations, which ultimately limits the query performance and scalability. We demonstrate this using a system design that extends Wukong to support both continuous and one-shot query (§6.2).

First, the streaming data consists of both *timeless* (e.g., tweets and likes) and *timing* (e.g., GPS address) data. Therefore, the integrated store should consider not only how to absorb the *timeless* data, but also how to flexibly and efficiently sweep the timing data and the timestamp of timeless data after they are expired. Wukong+S addresses this chal-

[6] Noted that we carefully optimize the execution by embedding all tuples into a single query to minimize the transformation cost, co-locating two systems to minimize the transmission cost, as well as executing GP3 in parallel. The uncertain task scheduling cost in Storm is also not included.

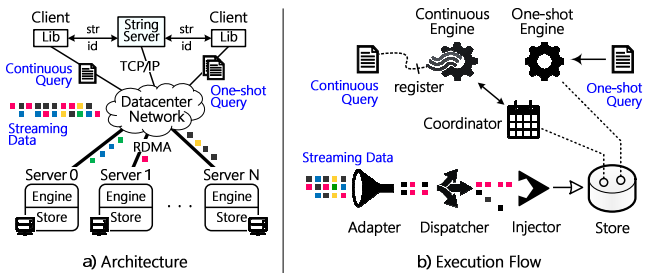

Fig. 5: The architecture and execution flow of Wukong+S.

lenge with a hybrid store to provide differentiated managed of timing and timeless data (§4.1).

Second, in the integrated store, the streaming data may be scattered over the entire stored data, which will significantly hurt the data locality of continuous queries to access streaming data. Wukong+S provides an efficient stream index with locality-aware partitioning to provide fast access to streaming data with good locality (§4.2).

Third, for a distributed stream querying system, it is important to ensure data consistency in a scalable way, including consistency among multiple streams from querying different stream windows and consistency among concurrent queries and insertions. Wukong+S provides a decentralized vector timestamp scheme with bounded scalarization to embrace both scalability and memory efficiency (§ 4.3).

Architecture. As shown in Fig. 5(a), Wukong+S follows a decentralized architecture on the server side, where each node can directly serve clients' requests. Each client[7] contains a client library that can parse continuous and one-shot queries into a set of stored procedures, which will be immediately executed for one-shot queries or registered for continuous queries on the server side respectively. Alternatively, Wukong+S can use a set of dedicated proxies to run the client-side library and balance client requests. To avoid sending long strings to the server and thus save network bandwidth, each string is first converted into a unique ID by the string server, as done in Wukong [38].

Wukong+S assumes a cluster that is connected with a high-speed, low-latency network with RDMA features. It scales by partitioning the initially stored data into a large number of shards across multiple nodes and dispatching streams to different nodes. The query engine layer binds a worker thread on each core with a logical task queue to continuously handle requests. The data store layer manages a partition of the data store, which is shared by all worker threads on the same node. Each node contains a continuous query engine and a one-shot query engine, which handle different queries accordingly.

Execution flow. Fig. 5(b) illustrates the execution flow of Wukong+S. Users can register continuous queries to the *Continuous Engine*, which passes the registered queries to the *Coordinator*. The coordinator prepares the input of each continuous query and invokes queries when data is ready. Each stream will first be handled by the *Adaptor*, which uses a batch-based model that groups tuples by individual timestamps. This model is similar to "mini batches" of small time intervals in Spark Streaming [45]. During the batching process, the Adaptor will also discard unrelated tuples and indicate whether each tuple is timing or timeless. Then *Dispatcher* will partition and dispatch the tuples in a batch to multiple nodes. Meanwhile, the *Injector* on each node will insert the batches received from dispatchers into the *Store*, where query workers will access the store accordingly to extract data in a particular stream window. Users can still send one-shot queries to the *one-shot query engine*, which will concurrently execute the queries on the data store and return the results to users.

4. Detailed Designs

4.1 Hybrid Store

Wukong+S provides a hybrid store to differentially handle streaming and stored data. For timeless data that is accessed by both continuous and one-shot queries, an *(*incremental persistent store) is used to absorb timeless data in streams, along with the initially stored data. For timing data that is only accessed by continuous queries, Wukong+S uses a time-based transient store, which consists of a sequence of transient slices. Once some transient slices have expired (i.e., all registered continuous queries no longer need to access it), Wukong+S will use a garbage collector to clean it up periodically.

Base store. Wukong+S inherits Wukong [38] as the basic store. Since Wukong+S assumes that the initially stored data is represented as a directed graph, our basic store uses a combination of vertex ID (vid), edge ID (eid) and in/out direction (d) as the key (in the form of $[vid|eid|d]$) and the list of neighboring vertex IDs as the value. Except for the normal vertices in the graph, the basic store also creates *index* vertices to assist queries that rely on retrieving a set of normal vertices connected by edges with a certain label. It can be viewed as a reverse mapping from a kind of edge to the normal vertices.

Fig. 6 shows how the initially stored data (i.e., X-Lab) in Fig. 1 is stored in the base store. An additional mapping table (ID-mapping) maintained by String Server [38] is used to convert each string into a unique ID for both data and queries. As an example, the neighbor list for all out-edges of Logan labeled with po is specified by key $[1|4|1]$ and the value is 5 and 6 (i.e., T-13 and T-14). The index key $[0|4|0]$ points to all normal vertices that have an in-edge labeled with po so that the value is 4, 5, and 6 (i.e., T-12, T-13, and T-14).

For each query, Wukong+S uses the constant part of a query condition as the key to do the graph-exploration [38] starting from the normal vertex or the index vertex. For example, all posts match the first query condition (Logan po

[7] The client may not be the end user but the front-end of Web service.

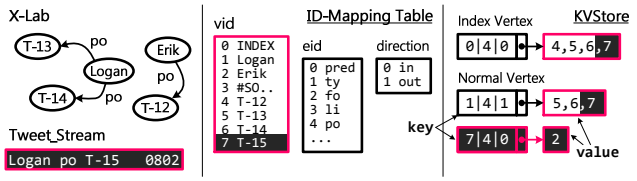

Fig. 6: The injection of a triple on continuous persistent store.

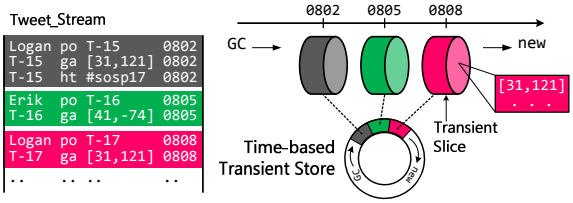

Fig. 7: The design of time-based transient store in Wukong+S.

?X) of the one-shot query Q_S in Fig. 1 can be retrieved by using Logan and po as key to access the value (i.e., T-13 and T-14).

Continuous persistent store. To continuously absorb the timeless data (e.g., tweets, likes and hashtags) of streams, Wukong+S extends the basic store to support incremental key/value update, resulting in a *continuous persistent store* [8]. The incremental timeless data will be accessed by both continuous queries before expired and one-shot queries at all time. For each timeless tuple in the stream, Wukong+S will inject the useful part of timeless tuple into the persistent store, while the timestamp is maintained by stream index (§4.2); the injections may require appending to multiple existing key/value pairs or even adding new ones.

Fig. 6 shows the impact on key/value stores due to adding a new tuple ($\langle$Logan, po, T-15$\rangle$, 0802) in the Tweet_Stream. First, the new string (e.g., T-15) would be converted into a unique ID (e.g., 7) and updated to the ID mapping table in the String Server. Second, the key/value pair for new vertices should be inserted (e.g., $[7|4|0] \rightarrow [2]$). Third, the vid of new vertex (7) should be appended to the value of its neighboring vertices (e.g., 7 of $[5,6,7]$ for $[1|4|1]$). Finally, the involved index vertex (po) should be also updated (i.e., 7 of $[4,5,6,7]$ for $[0|4|0]$). The timestamp (0900) would be simply discarded due to the timeless semantics of persistent store.

The injector on each node will be in charge of inserting the timeless tuples received from dispatchers on different nodes into the local shard of the persistent store. When multiple injector threads are required due to massive streams or high streaming-rate, Wukong+S will statically partition the key space of the store and exclusively assign one partition to one thread, which can avoid using locks during injection.

Time-based transient store. The timing data in the streams will only be accessed by involved continuous queries in a certain window; they should be efficiently swept out upon expiration. Therefore, Wukong+S uses a new *time-based transient store* to hold timing data. Since streams may have different windows sizes and steps, Wukong+S maintains a transient store for each stream. Meanwhile, Wukong+S uses the same sharding approach for both persistent and transient stores, which co-locates the timeless and timing data in the stream.

As shown in Fig. 7, the time-based transient store consists of a sequence of *transient slices* arranged in time order. Each transient slice stores the timing data of a stream batch. The injector thread will continuously create new transient slices and append them to the later side, while the garbage collector (GC) thread will also continuously free old transient slices and remove them from the earlier side. Wukong+S uses a contiguous ring buffer with fixed user-defined memory budget to store time-based transient slices. The GC thread will be periodically invoked in the background, or explicitly invoked when the ring buffer is full.

4.2 Stream Index

The continuous query is different from the one-shot query, as it needs to frequently access the latest steaming data. Compared to the stored data, the size of streaming data in a window is quite small. After the persistent store absorbs the streaming data, such small data will be scattered over the entire persistent store. Accessing streaming data through the *normal path* is very slow, as it will first use query condition to locate the key and then walk the value to retrieve the data matching involved window. For example, as shown in Fig. 8, suppose that the continuous query wants to retrieve all persons that like the tweet T-15 from 0807 to 0811.

It has to use $[7|3|0]$ to locate key in the persistent store first, and then walk the value to find matched results Clint and Steve (i.e., 12, 13). Worse even, all the timestamps of triples must also be stored in the persistent store, which will result in additional memory cost as they are useless once expired and GC would significantly interfere with the execution of one-shot queries.

To remedy this issue, Wukong+S proposes *stream index* to provide a *fast path* for continuous queries to access streaming data. The stream index also eliminates the timestamp from timeless data in the stream to save memory and avoid interference to one-shot queries. Fig. 8 illustrate 3 steaming indexes of the Like_Stream corresponding to timeless data in the left sample stream. Similar to the transient store, Wukong+S arranges the stream indexes in a time-based order and also create and remove the stream index in two sides. Each stream index consists of several entries that have the same structure to the key in the persistent store. The only difference to the key in key/value store is the pointer may locate to the middle of value[9]. For example, the continuous query can use the timestamp and $[7|3|0]$ to locate the

[8] Wukong only provides a preliminary support to update, which only allows single stream and adopts centralized injection mechanism.

[9] The point in the stream index and the persistent store is a 96-bit fat pointer that includes the address (64-bit) and size (32-bit).

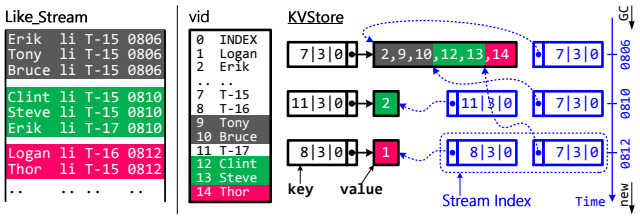

Fig. 8: The design of stream index in Wukong+S.

key in a certain stream index first, and then directly retrieve the matched results. The search space is extremely decreased and independent to the size of stored data.

Locality-aware partitioning. The general approach to partitioning stream index is to co-locate the index and data, which can provide the data locality for the continuous query. However, it may result in the execution of continuous queries spilt into to multiple nodes, namely migrating execution. We observe that the continuous queries are generally small and selective, since the streaming data is relatively few. Wukong [38] has shown that migration execution is not optimal for such queries due to the cost of network traffic and additional scheduling latency. Alternatively, the continuous query can also fetch the streaming data from remote nodes and run on a single node, namely migrating data, which can further leverage one-sided RDMA READs to bypass remote CPU and OS. Unfortunately, the partitioned stream index would incur an additional RDMA READs.

Considering that the stream index is relatively small, Wukong+S replicates the index to avoid additional RDMA READs. However, blindly replicating the stream index to all nodes is not scalable and may cause large overhead when data injection and garbage collection. Therefore, Wukong+S will only replicate the stream index to the node where the registered continuous queries demand to access this stream. Meanwhile, the registration of continuous queries will also consider their involved streams. The continuous query can always access stream index locally and uses RDMA READs to access the remote data if necessary. According to the registration information, Wukong+S can even dynamically create and replicate stream index on demand. Our partitioning mechanism for stream index exploits the *query locality* rather than the *data locality*. Fig. 9 shows a sample partition for the stream indexes of Like_Stream and Tweet_Stream. Note that stream indexes are always shared by all local continuous queries.

4.3 Consistent Query with High Memory Efficiency

To provide good scalability, the streaming and stored data in Wukong+S are shared by all queries, which avoids redundant resource consumption. However, it is non-trivial to ensure consistency for all continuous queries on different nodes with different window sizes and steps (continuous queries), or even one-shot queries.

For a continuous query, the question to answer is when to trigger it. Wukong+S adopts a data-driven execution model

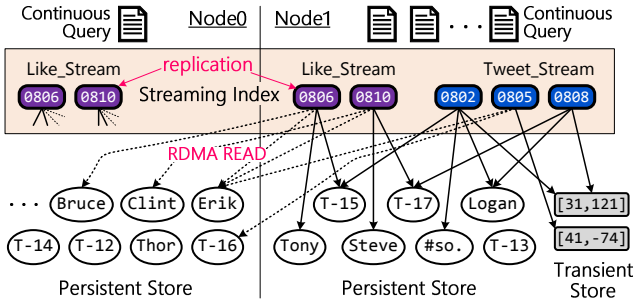

Fig. 9: The locality-aware partitioning for stream index.

for streams, where a query will be invoked when its windows of involved streams are ready. Since a stream batch will be distributed and inserted into multiple nodes, it is necessary to make streaming data visible only when they have been inserted at all nodes. To this end, Wukong+S adopts *vector timestamps* (VTS) to reflect the insertion state of multiple streams on each node. Each node has a local vector timestamp (*Local_VTS*) to record the current state of all streams on this node as shown in Fig. 10 (e.g., batch#4 of S0 and batch#12 of S1 has finished insertion on Node0). The Coordinator on each node will compute a *stable* vector timestamp (*Stable_VTS*) from the *Local_VTS* on all nodes, and invoke the continuous queries registered on the engine when all timestamps in their *Next* windows are smaller than or equal to the corresponding timestamps in the *Stable_VTS*. For example, in Fig. 10, the query (Q_C) needs the batch#5 of S0 so that it can not be executed according to the current *Stable_VTS* ([4,12]).

For one-shot query, since the timeless data of streams (e.g., tweets and likes) will be updated to the persistent store, the stored data is no longer static for one-shot queries. Therefore, Wukong+S treats one-shot queries as read-only transactions and the streaming data insertion as append-only transactions. Snapshot isolation is used to ensure all one-shot queries can read a consistent state of the persistent store. Meanwhile, Wukong+S ensures that stream batches within a stream will be inserted in order and the order of batches from different streams is indifferent. Consequently, the append-only transactions will never conflict under snapshot isolation.

A straightforward solution to implement snapshot isolation is to reuse the timestamp of streams to stamp all streaming data in the persistent store; each one-shot query will first read the stable vector timestamp (i.e., *Stable_VTS*) as the timestamp of the query ((i.e., *Query˙VTS*)), and always read the value whose timestamp is not larger than it. However, this solution requires all streaming data in the value associated with a vector timestamp, which will incur large memory overhead and degrade performance due to the pollution of timestamps on key/value pairs.

On the other hand, moving the timestamps to the key can avoid the pollution to the value, while the memory space overhead may further increase due to additional pointers,

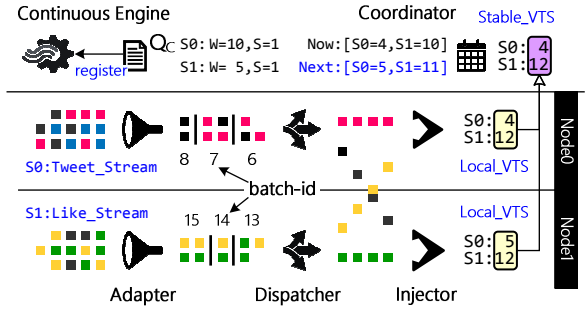

Fig. 10: Distributed stream injection and punctual query trigger.

and it is still non-trivial to maintain infinite timestamps in the key.

Bounded snapshot scalarization. Wukong+S addresses this issue by transferring the vector timestamps into a scalar snapshot number (SN), which is a novel *tradeoff* between the CPU/memory cost with the staleness of one-shot query's results. First, the coordinator will announce a plan of the mapping between a snapshot number (SN) and the range of vector timestamps (VTS) in advance (i.e., *SN-VTS Plan*, Second, the injector on each node ensures that all stream batches with the same snapshot number (SN) are consecutively stored in key/value pairs; in this way, a snapshot of each key is only associated with one memory interval. Finally, each query will obtain a stable snapshot number (*Stable_SN*) instead of *Stable_VTS*, and use it to read a consistent snapshot from the persistent store.

There are two key points of this solution. First, the coordinator can leverage the interval of the mappings to control the *staleness* of query results. If the increase of mappings is limited to 1 (e.g., from SN=2:[S0=3,S1=9] to SN=3:[S0=4,S1=9]), then the one-shot query can also provide the latest results. However, it will restrict the insertion of streaming data since the injector can only insert the stream batch#4 of stream S0 on all nodes. On the other hand, if the increase of mappings is large, the injector has more flexible to insert stream batches, while the query result will be staled to the last snapshot. Generally, the more balance of the speed of injectors on each node is, the smaller of the interval of mappings will be chosen.

Second, the coordinator can leverage the progress of the plan to control the *memory overhead* (the number of SNs in the key). For example, if the coordinator always publishes a single new mapping after the current mapping is finished on all nodes, each key only need to maintain the information (i.e., SN and pointer) for two snapshots, namely the version is using and the version is inserting.

As shown in Fig. 11, the current *Stable_SN* is 2, which means that the streaming data of S0 before batch#3 and S1 before batch#9 has been visible as stored data to both continuous and one-shot queries; currently the snapshot 3 is inserting, while Node1 is stalled to wait for the new plan of the mapping for SN#4.

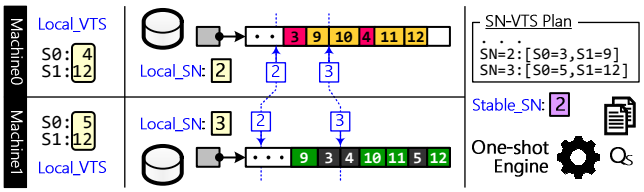

Fig. 11: A sample of bounded snapshot scalarization. The blue boxes with snapshot information point to the start of the corresponding snapshots, which is physically stored in the key.

After batch#5 of S0 is inserted on Node0, the *Local_VTS* on Machhine0 will become [S0=5,S1=12]. The coordinator will find that the injection status has met the requirement of SN#3 according to the SN-VTS Plan and updates *Local_SN* on Node0 and *Stable_SN* to 3. The coordinator will further publish the new plan of SN#4. The injector can continue to absorb the streaming data and overwrite the snapshot information from 2 to 4.

5. Implementation

The Wukong+S prototype comprises around 6,000 lines of C++ code and runs atop an RDMA-capable cluster.

Fault tolerance. Wukong+S assumes upstream backup [25] such that the stream sources buffer recently sent data and replay them. It also can be implemented by external durable message service (e.g., Kafka [2] and Scribe [6]). Meanwhile, Wukong+S provides at least once semantic to the continuous queries. This means that there may exist twice results on the same window of streams in case of machine failure. It would be trivial to address it in the client by checking the time information of results.

Wukong+S uses two different mechanisms to handle failures at the query engine layer and the data store layer. In the query engine layer, Wukong+S only need to log all continuous queries to the persistent storage and simply re-register them after recovery from a machine failure. In the data store layer, Wukong+S adopts incremental checkpointing by periodically logging in background. Each machine will only log the streaming data in local. First, Wukong+S will use the stream index to locate the data since the latest checkpoint and store them (key and value) to the persistent storage. Note that the data of different streams within a checkpoint can be logged in parallel even sharing the same key-value pair. The order of them is not important after recovery. The local and stable vector timestamps (*Local_VTS* and *Stable_VTS*) should also be persistent. Finally, Wukong+S will notify the source of streams to flush buffered data until the new checkpoint.

Since the stream index and data in the transient store will expire after the involved windows slide, Wukong+S can choose to avoid logging such data. The inspection approach is similar to that of the garbage collection. However, it may delay the end of the checkpoint and increase the pressure of the stream source if the window of queries is too large.

To recover from a machine failure, Wukong+S will reload initial RDF data first and then all durable checkpoints in a proper order. The latest stream index and the transient store will be reloaded if needed. Wukong+S will further re-register continuous queries and the latest local and stable vector timestamps (*Local_VTS* and *Stable_VTS*). The *SN_VTS_Plan* and the *Local_SN* will be reset to the *Stable_VTS*.

Currently, Wukong+S does not support dynamic reconfiguration, while some mechanisms in prior work [22] can similarly be applied to our system.

Leveraging RDMA. As Wukong+S is based on Wukong, which excessively leverages RDMA to optimize throughput and latency, Wukong+S also targets at supporting stream processing while maintaining low latency and high throughput.

First, we observe that queries on streams normally access only a modest amount of data. Hence, instead of using a distributed fork-join style execution, Wukong+S leverages RDMA's low latency feature and the fact that the latency is usually insensitive to the payload size to a certain extent [38] to provide in-place execution. That is, Wukong+S mainly uses a single thread on a single machine to handle a query. The parallelism is still achieved since the data of stream is distributed to all machines and a lot of queries can be processed in parallel.

Second, in the split design of Wukong+S, metadata is another type of location cache and provides another layer of indirection to reduce data duplication. As a result, a normal remote access to a key/value pair requires at least two RDMA READs: read key (lookup) and read value (as metadata includes the key and the location of value). To this end, Wukong+S accumulates the metadata for each stream within one machine. Hence, a query only needs to use one RDMA read to retrieve the key/value pair since the metadata may already be locally accessible. As the metadata is usually much smaller than data, it is usually feasible to accumulate all metadata for one stream in one machine.

6. Evaluation

6.1 Experimental Setup

Hardware configuration: All evaluations were conducted on a rack-scale cluster with 8 nodes. Each node has two 12-core Intel Xeon E5-2650 v4 processors and 128GB DRAM. Each node is equipped with a ConnectX-3 MCX354A 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps IB Switch, and an Intel X540 10GbE NIC connected to a Force10 S4810P 10GbE Switch. All nodes run Ubuntu 16.04 with Mellanox OFED v3.4-1.0.0.0 stack.

Benchmarks: We evaluate Wukong+S using two popular RDF streaming benchmarks, LSBench [27] and CityBench [13]. LSBench models a typical social network which comprises *streaming data* continuously generated from user activities and *stored data* representing properties of the social graph (e.g., user profiles and friend relationships). LSBench provides 6 types of continuous queries over both stored data and 5 RDF data streams for post (PO), post-like (PO-L), photo (PH), photo-like (PH-L), and GPS (GPS). We use 118 million tuples as initially stored dataset for evaluations on a single node, and 3.75 billion tuples for those on 8 nodes. The default stream rate of the 5 streams in total is 133K tuples per second. For brevity, the window and slide size over the 5 streams in all queries are 1s and 100ms respectively.

Table 1: The initial size of RDF data, the default stream rate and the usage of RDF streams for each query.

LSBench	Size / Rate	L1	L2	L3	L4	L5	L6
Initial	3.75B Triples		X	X		X	X
PO	10K Tuples/s	X	X	X	X		X
PH	86K Tuples/s					X	
PO-L	10K Tuples/s						X
PT-L	7.5K Tuples/s					X	
GPS	20K Tuples/s						

CityBench	Size / Rate	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Initial	139K Triples	X	X	X	X	X	X	X	X	X		
VT1	19 Tuples/s	X		X								
VT2	19 Tuples/s	X	X	X		X						
WT	12 Tuples/s		X									X
UL	7 Tuples/s				X		X					
PK1	4 Tuples/s						X	X	X	X		
PK2	4 Tuples/s							X	X	X		
PL1-5	4 Tuples/s										X	

CityBench simulates a smart city application, which conducts continuous queries over real-time IoT data streams generated by various sensors in the city of Aarhus, Denmark. CityBench provides 11 types of continuous queries over 11 RDF streams for vehicle traffic (VT1-2), parking (PK1-2), weather (WT), user location (UL), and pollution (PL1-5). We use the default setting of stream rate and initially stored data and set the size and step of all windows to 3s and 1s respectively.

Table 1 shows the default settings of both LSBench and CityBench, and the usage of data streams for each query.

Comparing targets: We compare the performance of Wukong+S with systems representing state of the art. As for the *composite design*, we evaluate the performance of CSPARQL-engine [41] and the integration of Wukong [38] with Apache Storm [42]. We also compare Wukong+S with Spark Streaming [15, 46], which stands for a popular solution for building streaming applications. Both streaming and stored data are represented as in-memory RDDs (i.e., the data abstraction in Spark). More specifically, we use DataFrames, the wrapper of RDDs, to store RDF data and Spark SQL [15] to perform the queries.

We further build an extension of Wukong, namely Wukong+, which can incrementally absorb all data (timing and timeless) in RDF streams and run stream queries over all the data. It can be roughly regarded as Wukong+S without stream index in the latency evaluations. Note that Wukong+ does not implement GC because it is non-trivial to find and remove timestamps and timing data from the tremendous dataset in graph stores.

It should be noted that the 2 composite solutions and Spark Streaming do not support evolving persistent store and thus perform queries over a static stored dataset. This keeps them away from the consistency problems faced by Wukong+S. In contrast, Wukong+S is the only system that supports evolving graph and distinguishes between timing and timeless data in streams.

CSPARQL-engine has limited capacity for processing stored data so that we remove all untouched tuples for a much smaller initial dataset for it. The number of cores for serving each continuous query on each node is restricted to 1, since they are usually light-weight. It means that performance (latency) of Wukong+S could be improved when necessary, and this will be discussed later in § 6.4.

6.2 Latency

We first study the performance of Wukong+S for each single query using the LSBench dataset. All experimental results are the median latency of one hundred runs.

Since CSPARQL-engine is a single-node system, we first run Wukong+S, Storm+Wukong and CSPARQL-engine on a single node and report their in-memory performance for LSBench-118M. As shown in Table 2, Wukong+S significantly outperforms Storm+Wukong by up to 53.8X (from 2.9X) and CSPARQL-engine for three orders of magnitude, mainly due to the benefit of its integrated design. CSPARQL-engine usually needs hundreds of msec to execute a single query, indicating its inefficiency from both its composite design and slow building blocks (e.g., Apache Jena). We decompose the latency of Storm+Wukong to better understand why it is higher. First, the cross-system overhead (i.e., total latency excluding latency in both components) for L2, L3, L5 and L6 is relatively high, ranging from 21.4% to 63.5% of the total latency. The overhead consists of data transformation and transmission between subcomponents acquired by composite design. L1 and L4 only touch streaming data, so that the query is solely within Storm. Second, comparing with Wukong+S, Storm+Wukong incurs a higher latency in Wukong, which adopts the same graph exploration strategy for query processing. This is caused by the inefficient query plan endured by systems adopting the composite design. Worse query plan leads to larger intermediate results and further results in higher latency in the Wukong subcomponent. These further back up our claim in §2.3.

We further compare Wukong+S in the distributed setting using LSBench-3.75B, as shown in Table 3. Wukong+S also significantly outperforms Storm+Wukong by up to 52.1X (from 3.4X) and Spark Streaming for three orders of magnitude. The cross-system overhead of Storm+Wukong is up to 68.7% (from 24.6%). It means that even if we replace Storm with a much faster streaming processor, the total latency will remain high due to cross-system overhead. As for Spark Streaming, the latency is always at the level of hundreds of msec, indicating its inefficiency to handle continuous queries.

Table 2: The query performance (msec) on a single node.

LSBench 118M	Wukong+S	Storm+ Wukong	Storm	Wukong	CSPARQL -engine
L1	0.12	0.35	0.35	-	155
L2	0.10	1.44	0.69	0.13	708
L3	0.13	1.91	1.14	0.16	872
L4	1.19	63.97	63.97	-	291
L5	2.89	67.89	12.49	12.23	1,984
L6	2.14	91.43	63.68	8.18	3,395
Geo. M	0.48	8.53	-	-	757

Table 3: The query performance (msec) on a 8-node cluster.

LSBench 3.75B	Wukong+S	Storm+ Wukong	Storm	Wukong	Spark Streaming
L1	0.10	0.34	0.34	-	219
L2	0.08	1.47	0.66	0.13	527
L3	0.11	1.56	0.78	0.18	712
L4	1.78	61.98	61.98	-	346
L5	3.50	53.94	13.14	3.73	2,215
L6	1.68	87.68	63.88	2.21	1,422
Geo. M	0.46	7.84	-	-	679

Table 4: The performance impact of stream index.

LSBench	L1	L2	L3	L4	L5	L6	Geo.M
Wukong+S	0.10	0.08	0.11	1.78	3.50	1.68	0.46
Wukong+	0.19	0.10	0.17	6.91	7.36	7.33	1.02
Speedup	1.9X	1.2X	1.6X	3.9X	2.1X	4.4X	2.2X

Stream index. To study the impact of stream index, we compare Wukong+ with Wukong+S. The key difference between Wukong+S and Wukong+ is in their strategy for maintaining timestamps and extracting the subgraph for a certain time period from underlying stores. Wukong+S utilizes *stream index* to extract graph data in a stream window. The stream index is built along with the injection of data streams, and stale indexes are periodically removed to save memory. Wukong+ directly inserts both streaming data and their timestamps into the underlying store, leading to two consequences. First, extracting data in a certain time period is inefficient without index, incuring high latency in Wukong+. As shown in Table 4, Wukong+S outperforms Wukong+ by up to 4.4X (from 1.2X), and the speedup is more obvious for large queries (more than 2X). Second, Wukong+ does not support efficient garbage collection, since deletion is costly and non-trivial after data and timestamps are coupled together in the underlying store. The stale and useless timestamps will accumulate in Wukong+, resulting in unnecessary memory and execution cost along with the injection of streams. By contrast, the stream index in Wukong+S is carefully organized in memory and hence are very friendly to garbage collection.

6.3 Scalability

We evaluate the scalability of Wukong+S with respect to the number of nodes. Note that we omit the evaluation on a single server as LSBench-3.75B (amounting to 653GB in raw format) cannot fit into memory.Since continuous queries are usually lightweight, we assign only 1 core on each node to a single query. We categorize the six queries on LSBench dataset into two groups according to the relation between their result size and the data size they access as done in prior work [38, 47]. Group (I): L1, L2, and L3; such queries are selective and produce quite fixed-size results regardless of

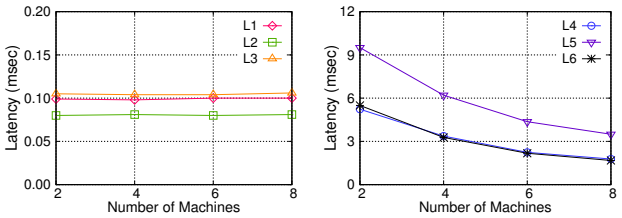

Fig. 12: The latency of queries in group (I) and (II) with the increase of nodes on LSBench-3.75B.

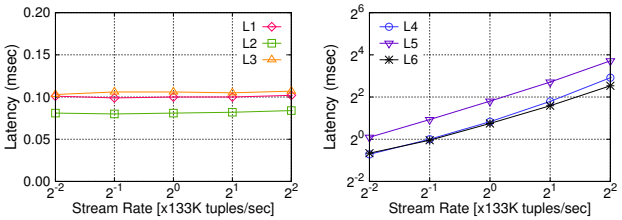

Fig. 13: The latency of queries in group (I) and (II) with the increase of streaming rate on LSBench-3.75B.

the total data size. Group (II): L4, L5, and L6; the results size of such queries increase with the growth of dataset.

Fig. 12(b) shows the speedup of Wukong+S for group (II) ranging from 2.8X to 3.2X, with the increase of servers from 2 to 8. This implies that Wukong+S can efficiently utilize the parallelism of the fork-join execution mode. For group (I), since the intermediate and final results are relatively small and fixed-size, using more machines do not improve the performance as shown in Fig. 12(a), but the performance is still stable by using in-place execution to reduce the network overhead.

6.4 Stream Rate

We also evaluate the scalability of Wukong+S in terms of stream rate. Stream rate in social network applications usually experience changes over time. Higher stream rate means a larger amount of data in a stream window and may further cause the latency of continuous queries to increase. We use the setting of LSBench in the latency evaluation and then increase the stream rate from 33K to 533K tuples per second. The number of tuples in each window increases proportionally.

As shown in Fig. 13, Wukong+S can achieve stable performance for queries in Group (I) regardless of the increasing stream rate, since such queries produce fixed-size results irrelevant to the amount of data in their windows. For queries in Group (II), latency increases with the growth of stream rate, since the result sizes of such queries also increase. Yet, it is still at a low level (smaller than 16ms in this test).

To avoid excessive resource consumption, we previously assigned only 1 core on each node for one continuous query. Adding more cores is an efficient way to relieve the tension. Assigning 4 cores on each node can speed up L4, L5 and L6 by 3.0X, 3.5X and 2.7X respectively (i.e., latency drops from 8.5ms, 15.1ms and 7.3ms to 2.85ms, 4.37ms and 2.67ms accordingly). The result demonstrates that clients are able to trade resources for performance when low latency is critical.

6.5 Injection Cost

We then evaluate the influence of data injection on the latency of Wukong+S. Different to prior stream computation systems, Wukong+S injects the streaming data into the underlying stores before launching stream queries. If the query execution overlaps the injection of corresponding stream batch, the injection may interfere the latency of such queries.

As shown in Table 5, the data injection usually costs up to 2.2ms delay for 100ms stream batch using the default streaming rate. To further break down the injection cost, the delay for building stream index is from 0.21ms to 0.43ms. The impact of data injection to the query latency is visualized in the CDF graph at Fig. 14(b) and 15(b) as the tail.

Table 5: The data injection and indexing cost (ms) per mini-batch (100ms) for all streams of LSBench with default streaming rate.

LSBench	PO	PO-L	PH	PH-L	GPS
Injection	0.52	1.77	0.45	0.16	1.18
Indexing	0.23	0.43	0.22	0.21	0.39
Total	0.75	2.20	0.67	0.37	1.57

6.6 Throughput of Mixed Workloads

Unlike prior systems, Wukong+S is designed to provide high query throughput at millions of continuous queries per second. In contrast, CSPARQL-engine executes queries sequentially, so that its throughput is directly related to latency. Spark Streaming and Apache Storm do not support data sharing between different queries, making them difficult to improve query throughput in this scenario.

We build emulated clients and two mixed workloads to study the performance of Wukong+S serving large number of concurrent queries registered by these clients. Each node runs 4 emulated clients and 16 worker threads for continuous query on dedicated cores. All clients will register as many queries as possible until the accumulated throughput saturated.

We first use a mixed workload consisting of 3 classes of queries (L1-L3). The query in each class has similar behavior except that the start point is randomly selected from the same type of vertices (e.g., Photo Album0, Photo Album1, etc.). The distribution of query classes follows the reciprocal of their average latency. As shown in Fig. 14, Wukong+S achieves a peak throughput of 1.08M queries/second on 8 nodes, 4.2X than that on 2 nodes (254K queries/second). Under the peak throughput, the geometric mean of 50th (median) and 99th percentile latency is 0.11ms and 0.90ms respectively.

We then use another mixed workload consisting of all 6 classes of queries (L1-L6). The mixture strategy is similar to the previous test and we obtain the results in Fig. 15. Wukong+S achieves a peak throughput of 802K queries/s on 8 nodes, 5.0X than that on 2 nodes (161K queries/second). The super scalability of throughput is mainly due to decreasing latency of L4-L6 in a larger cluster. Under the peak throughput, the median, 90th and 99th percentile la-

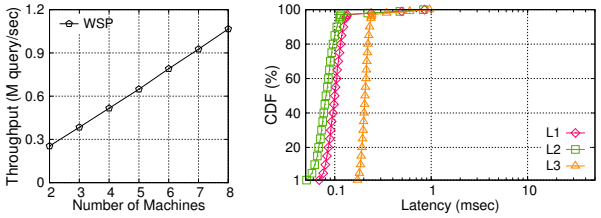

Fig. 14: (a) The throughput of a mixture of 3 classes of queries with the increase of nodes, and (b) the CDF of latency on 8 nodes.

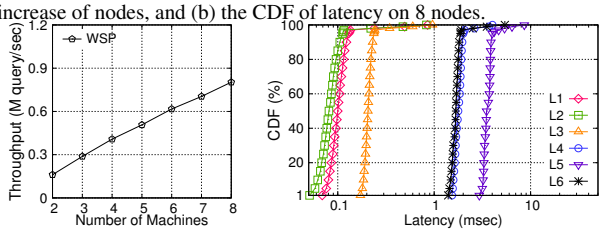

Fig. 15: (a) The throughput of a mixture of 6 classes of queries with the increase of nodes, and (b) the CDF of latency on 8 nodes.

Table 6: The memory usage (MB/min) comparison between streaming data and stream index for LSBench in default setting.

LSBench	Post	Po-like	Photo	Pt-like	GPS	Total
data	6.39	38.22	4.76	7.90	5.45	62.73
index	2.96	0.60	1.89	0.51	–	5.95
ratio	46.3%	1.6%	39.7%	6.5%	–	9.5%

tency for L4 class query is 2.3ms, 2.7ms and 4.1ms respectively.

6.7 Memory Consumption

In our design, we assume that *stream index* is small enough to be replicated in the cluster. We also assume that *bounded snapshot scalarization* can efficiently reduce memory usage. We now evaluate the two design choices using the default setting of LSBench-3.75B on our 8-node cluster.

Since GPS stream is not timeless (i.e., data will be injected into the transient store), stream index is not needed for executing related continuous queries. As shown in Table 6, 62.73MB raw data arrive in every minute with the default setting, resulting in 5.95MB memory consumption for the index. It means that if we wish to reserve all timestamps for 10 minutes, around 60MB should be reserved on each node for the stream index, which is quite affordable comparing to the raw data size (i.e., 627MB).

The benefits from bounded snapshot scalarization. When registering 2 streams and reserving 2 snapshots, the memory footprint for stored RDF data is 37.7GB on each node, and increases to 44.0GB without bounded snapshot scalarization. The memory footprint further increases to 40.9GB and 50.4GB with and without bounded snapshot scalarization when reserving 3 snapshots. Registering all 5 streams will cause no memory increase with bounded snapshot scalarization but increases memory footprint to 53.6GB otherwise.

6.8 Fault-tolerance Overhead

We have implemented a simple logging and checkpointing mechanism in Wukong+S for fault tolerance. We conduct the throughput test again with fault tolerance mechanisms enabled using the mixed workload of L1-L3 mentioned in *S*6.6 (i.e., LSBench-3.75B on 8 nodes). The result shows that logging delay for each batch is around 0.3ms and the throughput drops from 1.07M to 803K queries per second, about 11.2% decrease. The 99th percentile latency increases from 0.15ms to 0.73ms due to checkpointing, while the 90th percentile latency is largely unchanged.

Table 7: The query performance (ms) on a single node.

CityBench 137K	Wukong+S	Storm+ Wukong	Storm	Wukong	Spark Streaming
C1	0.24	4.40	0.66	0.64	872
C2	0.37	4.48	0.80	0.64	1,557
C3	0.26	4.10	0.70	0.31	675
C4	0.98	2.67	0.25	0.99	802
C5	0.94	4.10	0.36	2.10	790
C6	0.26	1.91	0.25	0.24	764
C7	0.24	2.23	0.40	0.34	762
C8	0.27	2.05	0.43	0.30	692
C9	1.15	3.91	1.12	1.15	1,088
C10	0.78	1.18	1.18	-	1,086
C11	0.16	0.17	0.17	-	193
Geo. M	0.41	2.21	-	-	766

6.9 Other Workload: CityBench

We further study the performance of Wukong+S, Spark Streaming and Storm+Wukong over CityBench with setting shown in Table 1. The stream rate and data size are very small because the data was collected in a relatively small city. Therefore, we deploy the systems on a single node to test their latency. For a similar application in a megacity such as New York City, the stored data and stream rate would increase thousands of times.

As shown in Table 7, Wukong+S outperforms Storm+Wukong by up to 18.3X (from 2.7X) for queries touching both stream and stored dataset and significantly outperforms Spark Streaming. For Storm+Wukong, the cross-system overhead dominates the latency (i.e., from 40.1% to 75.4%) and the latency of Wukong subcomponent is up to 2.67X than that of Wukong+S. These results demonstrate again the issues of a composite design (i.e., cross-system cost and inefficient query plan), especially for complex queries.

7. Other Related Work

Stream processing engines. The increasing importance of real-time data processing has stimulated considerable interests in both academia and industry [10, 11, 19, 30, 33, 40]. They usually adopt a relational data model and thus are not suitable for processing highly linked streams due to a phenomenon called "Join Bomb" [4]. Furthermore, they do not consider the combination of stream data and stored data (i.e., perform queries solely on stream data). MaxStream [19] provides a unified interface for users to query on both streaming and stored data, and adopts a composite design to manage streaming systems and databases.

There are several related domains in database community, including temporal database [18], time series database [24] and graph database [14]. Compared to Wukong+S, such

designs target at different scenarios and data models, and thus present different design decisions.

Stream computation systems. Different from stream processing engines, there are also many streaming computation platforms [7, 12, 21, 26, 28, 34, 42] on which programmers can write and submit their own code for computation on streaming data. These systems tend to provide a general abstraction for all kinds of streaming problems, so that their performance is hard to reach the same level of Wukong+S for stream querying workload over fast-evolving linked data, and most of them do not consider the combination of streaming and stored data.

Several systems [22, 39, 43] are developed to support large-scale computation on streaming *graphs*, like PageRank, SSSP and KMeans. Kineograph [22] also adopts distributed consistent snapshots with epoch commits, which however results in high latency up to the order of minutes. Tornado [39] also uses graph snapshots and splits the iterative graph computation into main and branch loops to converge the results fast. KickStarter [43] can produce safe and profitable results for computation on streaming graphs via trimmed approximations. The latency of such work usually reaches several seconds or even minitues. TripleWave [31] is a recently published system for generating RDF streams and converting existing streams into RDF format on the web, which can be used to provide RDF stream sources for Wukong+S.

8. Conclusion

This paper described Wukong+S, the first distributed RDF streaming engine that provides real-time consistency query over streaming datasets. Several key designs like separating timeless and timing data, extracting stream windows from both timeless and timing data by metadata records and using vector timestamps to derive most recent snapshot made Wukong+S fast and consistent. Evaluation on an 8-node RDMA-capable cluster using LSBench and CityBench shows that Wukong+S significantly outperforms CSPARQL-engine, Storm+Wukong and Spark Streaming for both latency and throughput, usually at the scale of orders of magnitude.

Acknowledgments

We sincerely thank our shepherd Roxana Geambasu and the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61402284, 61572314, 61525204), the National Youth Top-notch Talent Support Program of China, and Singapore NRF (CREATE E2S2).

References

[1] Apache Jena. https://jena.apache.org/.

[2] Apache Kafka. http://kafka.apache.org/.

[3] Dbpedia's sparql benchmark. http://aksw.org/Projects/DBPSB.

[4] Demining the "Join Bomb" with Graph Queries. https://neo4j.com/blog/demining-the-join-bomb-with-graph-queries.

[5] Esper. http://www.espertech.com/esper/.

[6] Facebook Scribe. https://github.com/facebook/scribe.

[7] Microsoft StreamInsight Official blog. https://blogs.msdn.microsoft.com/streaminsight/.

[8] Options price reporting authority. https://en.wikipedia.org/wiki/Options_Price_Reporting_Authority.

[9] RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts/.

[10] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[11] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[12] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 2013.

[13] M. I. Ali, F. Gao, and A. Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *Proc. ISWC*, 2015.

[14] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[16] D. F. Barbieri, D. Braga, S. Ceri, E. D. VALLE, and M. Grossniklaus. C-sparql: a continuous query language for rdf data streams. *International Journal of Semantic Computing*, 4(01):3–25, 2010.

[17] Bio2RDF Consortium. Bio2rdf: Linked data for the life science. http://bio2rdf.org/, 2014.

[18] M. H. Böhlen. Temporal database system implementations. *SIGMOD Rec.*, 24(4):53–60, Dec. 1995.

[19] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. Haas, K. Kim, C. Lee, G. Mundada, M.-C. Shan, N. Tatbul, et al. Design and implementation of the maxstream federated stream processing architecture. *ETH Zurich, Computer Science, Tech. Rep*, 632, 2009.

[20] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *PLID*, pages 363–375. ACM, 2010.

[21] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proc. SIGMOD*, 2016.

[22] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[23] Google Inc. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[24] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 106–115. IEEE, 1999.

[25] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[26] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proc. SIGMOD*, 2015.

[27] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *Proc. ISWC*, 2012.

[28] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. of NSDI*, pages 439–454, 2016.

[29] C. Liu, R. Correa, H. Gill, T. Gill, X. Li, S. Muthukumar, T. Saeed, B. T. Loo, and P. Basu. Puma: policy-based unified multiradio architecture for agile mesh networking. *IEEE/ACM Transactions on Networking*, 22(6):1897–1910, 2014.

[30] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *International Conference on Data Engineering*, pages 555–566. IEEE, 2002.

[31] A. Mauri, J.-P. Calbimonte, D. DellAglio, M. Balduini, M. Brambilla, E. Della Valle, and K. Aberer. Triplewave: Spreading rdf streams on the web. In *International Semantic Web Conference*, pages 140–149. Springer, 2016.

[32] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 2015.

[33] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system–. In *CIDR*. Citeseer, 2003.

[34] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, 2013.

[35] National Center for Biotechnology Information. Pubchem-rdf. https://pubchem.ncbi.nlm.nih.gov/rdf/, 2014.

[36] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[37] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys*, pages 1–14, 2013.

[38] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proc. OSDI*, 2016.

[39] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 417–430, New York, NY, USA, 2016. ACM.

[40] C. Sirish, C. Owen, D. Amol, H. Wei, K. Sailesh, M. Samuel, R. Vijayshankar, and R. Frederick. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 46–58, 2003.

[41] Streamreasoning Reserach Group. CSPARQL Engine. https://github.com/streamreasoning/CSPARQL-engine.

[42] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proc. SIGMOD*, 2014.

[43] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. ASPLOS, 2017.

[44] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 481–492. ACM, 2012.

[45] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. SOSP*, 2013.

[46] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[47] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proc. VLDB*, 2013.