# H-STREAM: Composing Microservices for Enacting Stream and Histories Analytics Pipelines

Genoveva Vargas-Solar, Javier Espinosa-Oviedo

# H-STREAM: Composing Microservices for Enacting Stream and Histories Analytics Pipelines

Genoveva Vargas-Solar[1] and Javier A. Espinosa-Oviedo[2]

[1] French Council of Scientific Research (CNRS), LIRIS, Lyon, France
genoveva.vargas-solar@liris.cnrs.fr
[2] University of Lyon, ERIC-LAFMIA, Bron, France
javier.espinosa-oviedo@univ-lyon2.fr

**Abstract.** This paper introduces H-STREAM, a stream/data histories processing pipelines enactment engine. H-STREAM is a framework that proposes microservices to support the analytics of streams produced by systems collecting data stemming from IoT (Internet of Things) environments. Microservices implement operators that can be composed for implementing specific analytics pipelines as queries using a declarative language. Queries (i.e., microservices compositions) can synchronise online streams and histories to provide a continuous and evolving understanding of the environments they come from. H-STREAM microservices can be deployed on top of stream processing systems and data storage backends, tuned according to the number of things producing streams, the pace at which they produce them, and the physical computing resources available for continuously processing and delivering them to consumers. The paper summarises results of an experimental setting for studying H-STREAM scale-up possibilities according to the number of thins and production rate. Then it shows a proof of concept of H-STREAM in a smart cities scenario.

**Keywords:** Stream processing · Cloud · microservices.

## 1 Introduction

The Internet of Things (IoT) is the network of physical devices enabling objects to connect and exchange data. IoT enables the construction of smart environments (grids, homes, and cities) where streams are produced at different paces. The status of these environments can be observed, archived and analysed online, managing and processing streams. To have a thorough understanding, modelling and predicting smart environments behaviour, processing, and analytics tasks must combine streams and persistent historical data. For example, at "9:00, start computing the average number of people entering a shopping mall every morning and identify points of interest in the mall according to peoples flow in the last month". Answering this query is challenging because it is necessary to determine: (i) the streams that must be discarded or persist into histories (do we

store the average/hour or every event representing a person entering the mall? or the person visiting an area in the mall?); (ii) how to properly combine histories with streams within analytics tasks (do we combine the whole history with the average observation/hour? do we compute POIs of the last month and correlate them with new computed POIs observed online?). Existing stream platforms provide efficient solutions for collecting and processing streams with parallel execution backends for example Apache Flink [3], Kafka [4] and message-based infrastructures like Rabbit MQ [5]. Programmers rely on these platforms to define stream processing operations that consume "mini"-batches of streams observed through temporal windows thanks to query engines like Elasticsearch, Amazon Athena, Amazon Redshift and Cassandra. These engines use a list of passive queries to analyze and sequence data for storage or use by other processors. The focus has been oriented to ensure performance since streams can be massive and analytics computationally costly. The processing operations can be rather complex and ad-hoc to target application requirements. Analytics-based applications must build ad-hoc programs that process postmortem data and streams to perform online analytics tasks. Since programs are ad-hoc and queries are passive, the use of specific processing operations (e.g., clustering, windowing, aggregation) are (hard)coded, and they should be modified and calibrated if new requirements come up.

Current advances in data processing and data analytics have shown that it is possible to propose general operations as functions or operators that can be called, similar to queries within databases applications (e.g. Spark programs). Still, the stream/data processing operations remain embedded within programs, and this approach hinders data-program independence that can imply high maintenance costs. Addressing this requirement often implies integrating data processing systems and programming this kind of hybrid solution. We believe that platforms with processing and analytics operators that can be agnostic to processing streams and stored data histories are still to come. The challenge is defining a platform that can wrap operators as self-contained services and compose them into pipelines of analytics tasks as queries that can continuously deliver aggregated streams/historical data to target applications. This paper introduces H-STREAM [6] an analytics pipelines' enactment engine. It provides stream processing microservices for supporting the analysis and exploration of streams in IoT environments. A microservice is a software development technique that structures an application as a collection of loosely coupled services. H-STREAM combines stream processing and data storage techniques tuned depending on the number of things producing streams, the pace at which they produce them, and the physical computing resources available for processing them online and delivering them to consumers. H-STREAM deploys stream operators pipelines, called

[3] https://flink.apache.org
[4] https://kafka.apache.org/intro
[5] https://www.rabbitmq.com
[6] *Here the Github address and a Youtube address of a demonstration of the system.*

stream queries, on message queues (Rabbit MQ) and data processing platforms (Spark) to provide a performant execution environment.

The paper summarises results of an experimental setting for studying H-STREAM scale-up possibilities according to the number of thins and production rate. Then it shows a proof of concept of H-STREAM in a smart cities scenario.

Accordingly, the remainder of the paper is organised as follows. Section 2 introduces related work regarding stream processing. The section discusses some limitations and underlines how our work intends to overcome certain limitations concerning those solutions. Section 3 describes the general architecture of H-STREAM with operators as microservices that are deployed on high-performance underlying infrastructures. It also introduces the microservices composition language for defining stream processing pipelines as queries. Section 4 introduces the core of our contribution, a stream processing microservice. It also describes the experimental scenario that applies series of microservices to evaluate scale up in terms of the number of things and volume of streams. Section 5 introduces a proof of concept use case for analysing connection logs in cities and show how to define queries that have to synchronise histories with streams to analyse them online using H-STREAM query language. Section 6 concludes the paper.

## 2 Related work

Streams can result from a fine-grained continuous reading of phenomena within different environments. Observations are done in different conditions and with different devices. Therefore, streams must be processed for extracting useful information. Stream processing refers to data processing in motion or computing on data directly as it is produced or received. In the early 2000s, academic and commercial approaches proposed stream operators for defining continuous queries (windows, joins, aggregation) that dealt with streams [6, 10]. These operators were integrated as extensions of database management systems. Streams were often stored in a database, a file system, or other forms of mass storage. Applications would query the data or compute over the data as needed. These solutions evolved towards stream processors that receive and send the data streams and execute the application or analytics logic. A stream processor ensures that data flows efficiently and the computation scales and is fault-tolerant. Many stream processors adopt stateful stream processing [4, 3, 2, 11] that maintains contextual state used to store information derived from the previously-seen events.

We analyse stream processing systems that emerged to process (i.e., query) streams from continuous data providers (e.g. sensors, things). These systems are designed to address scalability including (i) streams produced at a high pace and from millions of providers; (ii) computationally costly processing tasks (analytics operations); (iii) online consumption requirements. Apache Storm[7] is a distributed stream processing computation framework that is distributed, fault-tolerant and guarantees data processing. A Storm application is designed as a

[7] https://storm.apache.org

"topology" in the shape of a directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. Edges on the graph represent named streams flows and direct data from one node to another. Together, the topology acts as a data transformation pipeline. Apache Flink is an open-source stateful stream processing framework. Stateful stream processing integrates the database and the event-driven/reactive application or analytics logic into one tightly integrated entity. With Flink, streams from many sources can be ingested, processed, and distributed across various nodes. Flink can handle graph processing, machine learning, and other complex event processing. Apache Kafka is an open-source publish and subscribe messaging solution. Services publishing (writing) events to Kafka topics are asynchronously connected to other services consuming (reading) events from Kafka - all in real-time. Kafka Streams lacks point-to-point queues and falls short in terms of analytics. Spring Cloud Data Flow[8] is a microservice-based streaming and batch processing platform. It provides tools to create data pipelines for target use cases. Spring Cloud Data Flows has an intuitive graphic editor that makes building data pipelines interactive for developers. Amazon Kinesis Streams[9] is a service to collect, process, and analyse streaming data in real-time, designed to get important information needed to make decisions on time. Cloud Dataflow[10] is a serverless processing platform designed to execute data processing pipelines. It uses the Apache Beam SDK for MapReduce operations and accuracy control for batch and streaming data. Apache Pulsar is a cloud-native, distributed messaging and streaming platform. Apache Pulsar[11] is a high-performance cloud-native, distributed messaging and streaming platform that provides server-to-server messaging and geo-replication of messages across clusters. IBM Streams[12] proposes a Streams Processing Language (SPL). It powers a Stream Analytics service that allows to ingest and analyse millions of events per second. Queries can be expressed to retrieve specific data and create filters to refine the data on your dashboard to dive deeper.Source[13]. Event stream query engines like Elasticsearch, Amazon Athena, Amazon Redshift, Cassandra define queries to analyze and sequence data for storage or use by other processors. They rely on "classic" ETL (extraction, transformation and loading) processes and use query engines to execute online search and aggregation, for example, in social media contexts (e.g., Elastichsearch) and SQL like queries on streams (e.g. Amazon Athena, Redshift and Cassandra).

**Discussion.** The real-time stream processing engines rely on distributed processing models, where unbounded data streams are processed. Much data are of no interest, and they can be filtered and compressed by orders of magnitude [9, 12, 13]. Stream querying and analytics are often performed after the complete scanning of representative data sets. This strategy is inconvenient for real-time

[8] https://spring.io/projects/spring-cloud-dataflow
[9] http://aws.amazon.com/kinesis/data-streams/
[10] https://cloud.google.com/dataflow
[11] https://pulsar.apache.org/
[12] https://www.ibm.com/cloud/streaming-analytics
[13] https://deepsource.io

processing. Windowing mechanisms have emerged for processing data stream in a predefined topology with a fixed number of operations such as join, aggregate, filter, etc. The challenge is to define these filters in such a way that they do not discard information and that process streams produced at high pace to fulfill consumers requirements. Furthermore, online analysis techniques must process streams on the fly and combine them with historical data to provide past and current analytics of observed environments. Despite solid stream processing platforms and query engines, solutions do not let programmers design their analytics pipelines without considering the conditions in which streams are collected and eventually stored. H-STREAM has been designed as a stream processing and analytics cartridge for defining stream analytics pipelines and enacting them by composing microservices that hide the underlying platforms dealing with low-level tasks for collecting and storing streams.

## 3 H-STREAM for building querying pipelines for analysing streams

We propose H-STREAM, an analytics' pipelines enactment engine with microservices that can be composed for processing streams (see figure 1). H-STREAM operators implement aggregation, descriptive statistics, filtering, clustering, and visualisation wrapped as microservices. Microservices can be composed to define pipelines as queries that apply a series of analytics operations to streams collected by stream processing systems and stream histories. H-STREAM relies on (i) message queues for collecting streams online from IoT farms; and (ii) a backend execution environment that provides a high-performance computing infrastructure (e.g., a virtual data centre [1], a cloud) with resources allocation strategies necessary for executing costly processes.

*Composing microservices.* Microservices can work alone or be composed to implement simple or complex analytics pipelines (e.g., fetch, sliding window, average, etc.). A query is implemented by composing microservices. For example, consider observing download and upload speed variations within users' connections when working on different networks. Assume that observations are monitored online but that previous observations are also stored before the query is issued. A network analyst willing to determine if she obtains the expected bandwidth according to her subscription to a provider can ask *every two minutes give me the fastest download speed of the last 8 minutes* (see a) in Figure 2). Figure 2 b) shows the composition implementing this query example that starts calling a Fetch, and a Filter operators that retrieve respectively the streams produced online with a history filtering the *download_speed* collected the *last 8 minutes*. Results produced by these services are integrated by the operator MAX that synchronises the streams with the history to look for the maximum speed. The result is stored by a service Sink that contacts Grafana. This query is executed every two minutes by an operator window. The operator Fetch interacts with a RabbitMQ service that collects streams from devices and with a service that con-

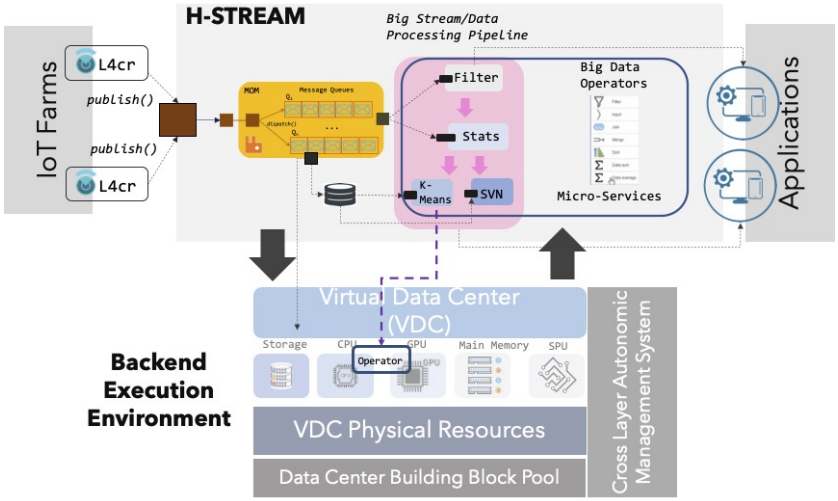


**Fig. 1.** H-STREAM General Architecture.

tacts InfluxDB to store the streams for building a history. Finally, an operator window triggers the execution of the query every two minutes.

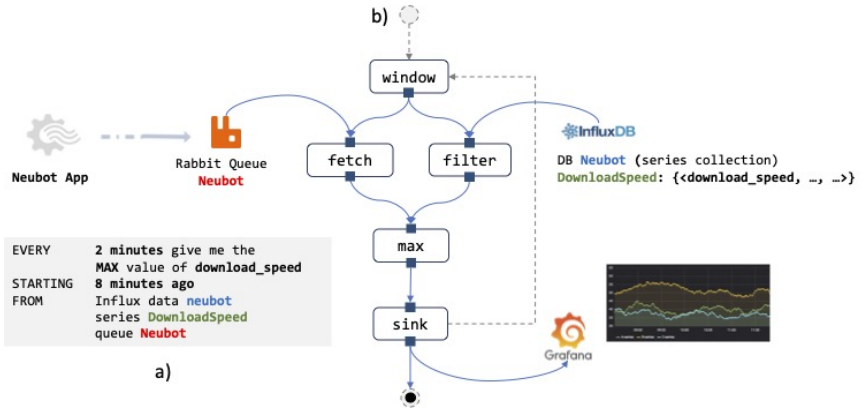


**Fig. 2.** Microservices Composition Example.

The approach for composing microservices is based on a composition operation that connects them by expressing a data flow (IN/OUT data). We currently compose aggregation services (min, max, mean) with temporal windowing services (landmark, sliding) that receive input data from storage support or a continuous data producer. We propose connectors, namely Fetch and Sink mi-

croservices that determine the way microservices exchange data from/to things, storage systems, or other microservices.

*Stream Processing Pipelines Query Language.* We proposed a simple query language with the syntax presented in figure 3, used to express:

- The frequency in which data will be consumed (EVERY(number:Integer, timeUnit:{minutes, seconds, hours})).
- The aggregation function applies to an attribute of the input tuples (min, max, mean).
- The observation window on top of which aggregation functions will perform. The window can involve only streams produced online (the last 5 seconds) or include historical data (the last 120 days).

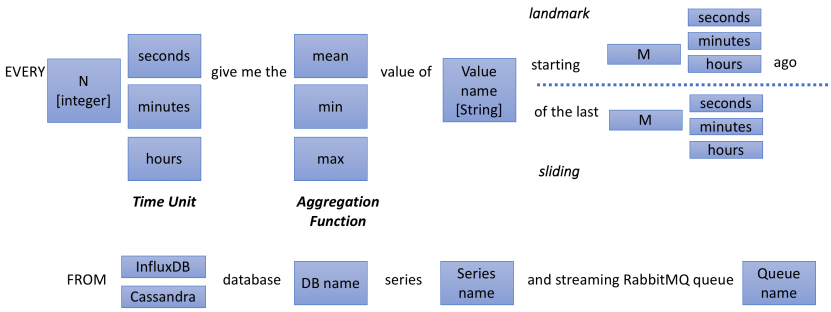


**Fig. 3.** Taxonomy of queries that can be processed by composing microservices.

The observation can be done starting from a given instance in the past (starting(number:Integer, timeUnit:{minutes, seconds, hours})) until something happens. For example, from the instant in which the execution starts until the consumer is disconnected. This corresponds to a landmark window [7]. It can also be done continuously starting from a moving "current instant" to several {minutes, seconds, hours} before. This case corresponds to a sliding window. The expression includes the logic names of the data producers that can be a store (Influx, Cassandra) and/or a streaming queue provided by a message-oriented middleware (e.g., RabbitMQ). A query expression is processed to generate a query-workflow that implements it (see figure 2). Activities represent calls to microservices; they are connected according to a control flow that defines the order they should be executed (i.e., in sequence or parallel). The control flow respects a data flow that defines data Input/Output dependencies. H-STREAM enacts the query-workflow coordinating the execution of microservices, retrieving partial output that serves as input or a result (see figure 2).

## 4 Stream processing microservice

Figure 4 shows the general architecture of a stream microservice. A microservice consists of three main components, Buffer Manager, Fetch and Sink, and OperatorLogic. The microservice logic is based on a scheduler that ensures the recurrence rate in which the analytics operation implemented by the microservice is executed. Stream processing is based on "unlimited" consumption of data ensured by the component Fetch that works if a producer notifies streams. This specification is contained in the logic of the components OperatorLogic and Fetch.

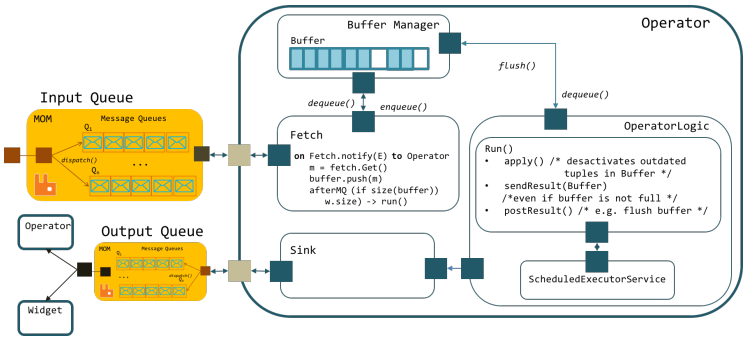


**Fig. 4.** Architecture of a stream processing microservice for processing data streams.

As shown in the figure, a microservice communicates asynchronously with other microservices using a message-oriented middleware. As data is produced, the microservice fetches and copies the data to an internal buffer. Then, depending on its logic, it applies a processing algorithm and sends it to the microservices connected to it. The microservices adopt the tuple oriented data model as a stream exchange model among the IoT environment producing streams and the microservices. A stream is a series of attribute-value couples where values are atomic (integer, string, char, float) from a microservice point of view. The general architecture of a microservice is specialised in concrete microservices processing streams using well-known window-based stream processing strategies: tumbling, sliding and landmark [8, 7]. Microservices can also combine stream histories with continuous flows of streams of the same type (the average number of connections to the Internet by Bob of the last month until the next hour).

Since RAM assigned to a microservice might be limited, and in consequence, its buffer, every microservice implements a data management strategy by collaborating with the communication middleware to exploit buffer space, avoiding losing data and generating results on time. A microservice communicates asynchronously with other microservices using a message-oriented middleware. As data is produced, the microservice fetches and copies the data to an internal buffer. Then, depending on its logic, it applies a processing algorithm and sends it to the microservices connected to it. There are two possibilities: (i) on-line

processing using tree window-based strategies [8, 7] (tumbling, sliding and landmark) well known in the stream processing systems domain; (ii) combine stream histories with continuous flows of streams of the same type (the average number of connections to the Internet by Bob of the last month until the next hour).

### 4.1 Interval oriented storage support for consuming streams

A microservice that aggregates historical data and streams includes a component named HistoricFetch. This component is responsible for performing a one-shot query for retrieving stored data according to an input query (for example, by a user or application). As described above, we have implemented a general/abstract microservice that contains a Fetch and Sink microservices. The historical fetch component has been specialized to interact with two stores: InfluxDB[14] and Cassandra[15]. The microservice HistoricFetch exports the following interface:

```
def queryToHistoric(function: String, value: String,
                    startTimeInMillis: Long, endTimeInMillis: Long,
                    groupByTimeNumber: Int, groupByTimeTimeUnit: String) :
                                                          List[List[Object]]
```

The method queryToHistoric(), shown in the code above, implements the connection to a data store or DBMS, sends queries and retrieves data. It returns the results packaged in the Scala structure List[ List[Row] ] objects, where Row is a tuple of three elements:

- Timestamp: Long, a timestamp in the format epoch;
- Count: Double, the number of tuples (rows) that were grouped;
- Result: Double, the result of the aggregation function.

As shown in the following code, a component HistoricFetch is created by the microservice specifying the name of the store (historicProvider), the name of the database managing the stream history (dbName.series) and the execution context. The current version of our microservice runs on Spark, so the execution context represents a Spark Context (sc).

```
val hf : HistoricFetch = new
HistoricFetch(historicProvider,dbName,series,sc)
```

The component HistoricFetch of the microservice creates an object HistoricProvider (step 1) that is used as a proxy for interacting with a specific store (i.e., InfluxDB or Cassandra). The store synchronously creates an object Connection (step 2). The Connection object will remain open once the query has been executed and

[14] InfluxDB is a time series system accepting temporal queries, useful for computing time tagged tuples (https://www.influxdata.comis)

[15] Cassandra is a key-value store that provides non-temporal read/write operations that might be interesting for storing huge quantities of data (http://cassandra.apache.org)

results received by HistoricFetch. Then, the component HistoricalFetch will use it for sending a temporal query using its method queryToHistoric() (step 3). The result is then received in the variable Result that is then processed (i.e., transformed to the internal structure of the operator see below) and shared with the other components through the Buffer of the microservice (step 4).

Consider the query introduced previously *every two minutes give me the fastest download speed of the last 8 minutes*. It combines the history of observations of the *last 8 minutes* with those produced continuously and this every two minutes. A particular situation to consider is how to synchronise the observations stored in the history with those fetched online. Figure 5 shows the general principle of the functional logic of a microservice (MAX) dealing with the streams harvested before the execution of the current query. The challenge is double, first retrieve batches of historical data according to different temporal filters. For example, the temporal filter for data produced the last 8 minutes observed from time $t_1$ is $[t_1\text{-}8,\ t_1]$ whereas for time $t_1 + i$ is $[t_1 + i\text{-}8,\ t_1 + i]$. Second, successively combine these batches with incoming flows arriving at time $t_1$, .. $t_1 + i$ every 2 minutes (as stated in the query). In technical terms, the query implies looking for the maximum down_load speed by defining windows of 8 minutes for observing the downlaod_speed in the connections. To get the fastest speed every 2 minutes (as stated in the query), we divide the 8 minutes into buckets of 2 minutes (see Figure 5 (1)) and look within the window for the max value, that is, the fastest download_speed (i.e., the fastest speed within the 2 minutes buckets), and keep it as the "local" maximum speed. We combine every bucket with the historical data filtered according to the corresponding time interval (see Figure 5 (2)). This strategy is valid only if the production timeliness of the stream producers and the operator microservice are synchronised. Finally, the global max will be the maximum of all this set of local maximum speeds that will be the fastest download speed in the last 8 minutes.

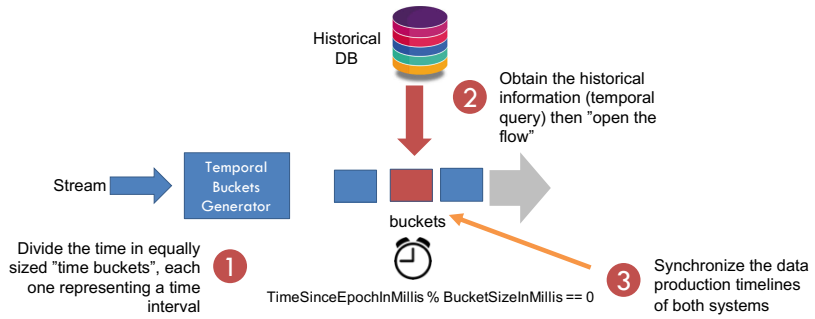


**Fig. 5.** Synchronizing stream windows with historic data for computing aggregations.

### 4.2 Microservices execution

Microservices are executed on top of a Spark infrastructure deployed on a virtual machine provided by the cloud provider Microsoft Azure (see Figure 6). As shown in the Figure, a microservice exports two interfaces: the operator interface as a SpepsIoT Component with methods to manage it (e.g., start/stop, bind/unbind) and to produce results in a push/pull mode; the DB interface (queryToHistorc in the upper part of the Figure) to connect and send temporal queries to a temporal database management system (e.g., Cassandra, InfluxDB). The microservice wraps the logic of a data processing operator that consumes time-stamped stream collections represented as series of tuples. We assume that it is possible to navigate through the tuple structure for accessing attribute values where one of the tuple attributes corresponds to its time-stamp. The time-stamp represents the arrival time of the stream to the communication infrastructure (i.e., rabbitMQ queue). The operator logic is implemented as a Spark program. Spark performs its parallel execution. Produced results can be collected by interacting with the operator through its interface; it can be connected to another microservice (e.g., the operator sink) as shown in the left part of the figure.

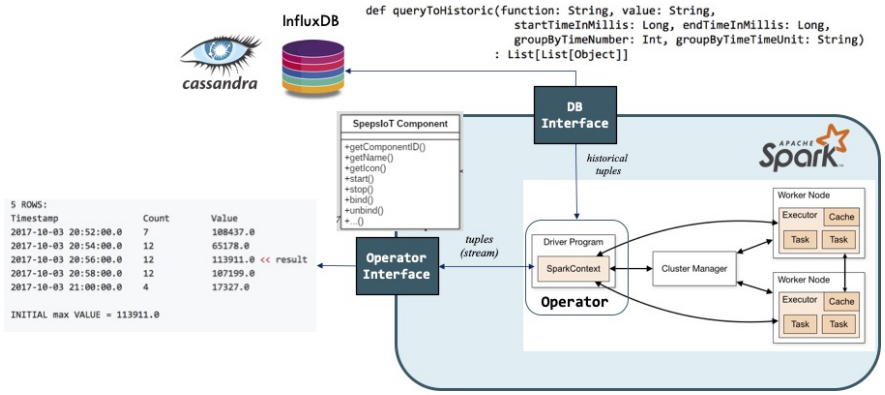


**Fig. 6.** Microservice execution.

A microservice running on top of the Spark platform processes streams considering the following hypothesis:

- There is a global time model for synchronizing different timelines (batch and stream).
- We use the Spark timeline as a global reference in the implementation, and we execute aggregations recurrently according to "time buckets". The size of the time bucket is determined by a query that defines an interval of observation (e.g. the average number of connections of the last 5 hours) and a moving temporal reference (e.g., every 5 minutes, until "now" if "now" is a moving temporal reference).

### 4.3 Experimental validation

We conducted experiments for validating the use of our microservices. For deploying our experiment, we built an IoT farm using our Azure Grant[16] and implemented a distributed version of the IoT environment to test a clustered version of Rabbit MQ. Therefore, we address the scaling-up problem regarding the number of data producers (things) for our microservices. Using Azure Virtual Machines (VM), we implemented a realistic scenario for testing scalability in terms of: (i) Initial MOM (RabbitMQ) installed in the $VM_2$ in figure 7. (ii) Producers (Things) installed in the $VM_1$ in figure 7. (iii) microservices installed in the $VM_3$ in figure 7.

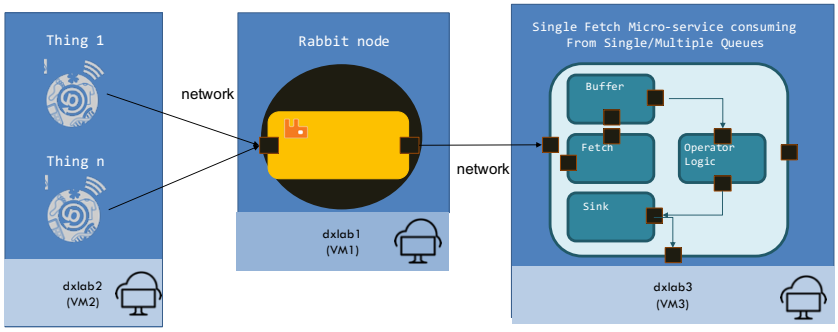


**Fig. 7.** General experimental setting deployed on Windows Azure.

As shown in figure 7, in this experiment, microservices and testbeds were running on separate VMs. This experiment leads to several cases scaling up to several machines hosting until 800 things with a clustered version of Rabbit using several nodes and queues that could consume millions of messages produced at rates in the order of milliseconds see Figure 8).

Observations in figure 9 showed the behaviour of the IoT environment regarding the message-based communication middleware when the number of things increased, when the production rate varies and when it uses one or several queues for each consuming microservice. We also observed the behaviour of the IoT environment when several microservices were consuming and processing the data. Of course, the most agile behaviour is when nodes and virtual machines increase independently of the number of things. Indeed, note that the performance of 800 things against 3 things does not change a lot by increasing nodes, machines and queues. Note also that devoting one queue per thing does not lead to essential changes in performance.

[16] The MS Azure Grant was associated with a project to perform data analytics on crowds flows in cities. It consisted of credits for using cloud resources for performing high-performance data processing.

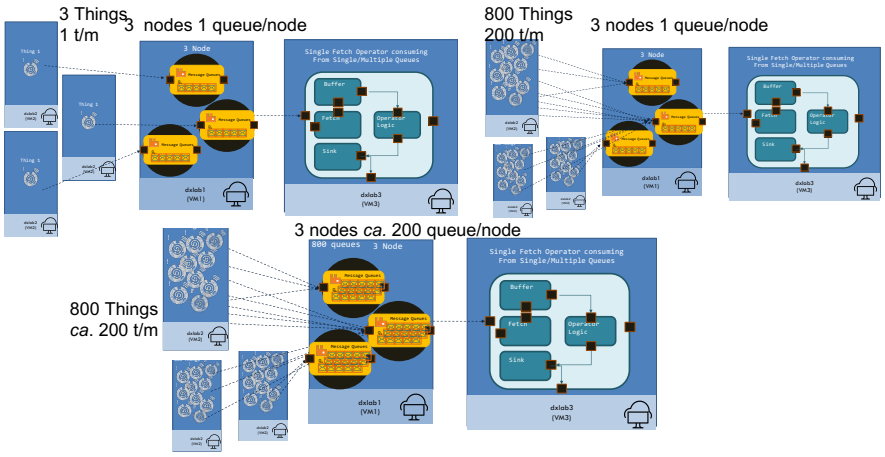


**Fig. 8.** Scale up scenarios

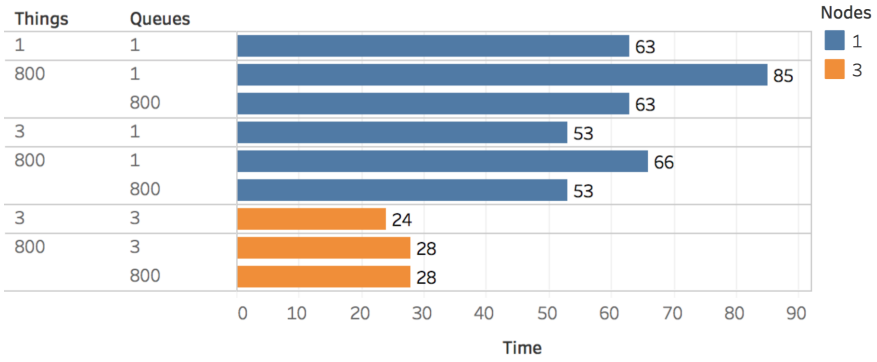


**Fig. 9.** Scale up results

For our experiments, we varied the settings of the IoT environment according to the properties characterising different scenarios. We used fewer things and queues, and more nodes to achieve data processing in an agile way. In this scenario, we assumed that there were few connected things with a high production rate. This scenario concerns an experiment conducted in the Neuroscience Laboratory at CINVESTAV Mexico (details can be found in [5]). Regarding connectivity in cities (see Section 5), with many people willing to connect devices in different networks available in different urban spaces, we configured more things and queues and nodes for the second one.

## 5 Use Case: Analysing the behaviour of network services

The use case scenario gives insight into the way microservices can be composed to answer continuous queries. For deploying our experiment, we used the IoT farm on Azure (see Section 4) that we implemented to address the scaling-up problem in terms of several data producers (things) to be consumed by our microservices.

The experimental scenario aims at analyzing the connectivity of the connected society. The data set used for the use case has been produced in the context of the Neubot project[17]. It consists of network tests (e.g., download/upload speed over HTTP) realized by different users in different locations using an application that measures the network service quality delivered by different Internet connection types[18] The idea is that people install the Neubot application on

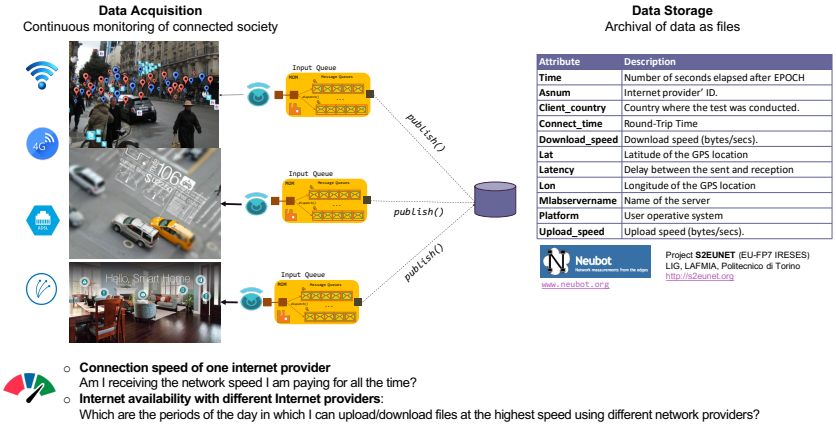


**Fig. 10.** Analyzing the connectivity of connected society.

their computers and devices. Every time they connect to the Internet using different networks (4G, Ethernet, etc.), the application computes network quality metrics. The data is used then to answer queries such as:
- *Am I receiving the network speed I am paying for all the time?*;
- *Which are the periods of the day in which I can upload/download files at the highest speed using different network providers?* A first simple example of the queries we tested is the following.

```
EVERY 2 minutes compute the max value of download_speed
      of the last 8 minutes
FROM influxdb database neubot series speedtest and streaming
```

[17] Neubot is a project devoted to measuring Internet from the edges by the Nexa Center for Internet and Society at Politecnico di Torino (https://www.neubot.org/).

[18] Here is the reference to the project that produced the dataset. We omit this information to respect double-blinded evaluation requirements.

```
RabbitMQ queue neubotspeed
```

The result of an H-STREAM query as a microservices composition can be seen in the right lower part of Figure 2. It corresponds to a visualisation of the data sunk on Grafana. One of the functions not directly integrated into existing stream processing systems is the possibility of synchronising long-term historical data (stored streams) with streams. The use case provides queries that combine long histories with online streams for the complete validation of H-STREAM. Through queries implemented by H-STREAM queries, we prove that it is possible to provide a hybrid post-mortem and online analytics.

```
EVERY 60 seconds compute the max value of download_speed
      of the last 3 minutes
FROM  cassandra database neubot series speedtests and streaming
      rabbitmq queue neubotspeed

EVERY   30 seconds compute the mean value of upload_speed
        starting 10 days ago
FROM  cassandra database neubot series speedtests and streaming
        rabbitmq queue neubotspeed

EVERY  5 minutes compute the mean of the download_speed
        of the last 120 days
FROM  cassandra database neubot series speedtests and streaming
        rabbitmq queue neubotspeed
```

The microservices query data histories stored as post-mortem collections, and they also connect to online producers observing their connections according to the observation window size. Figure 11 compares the execution time of

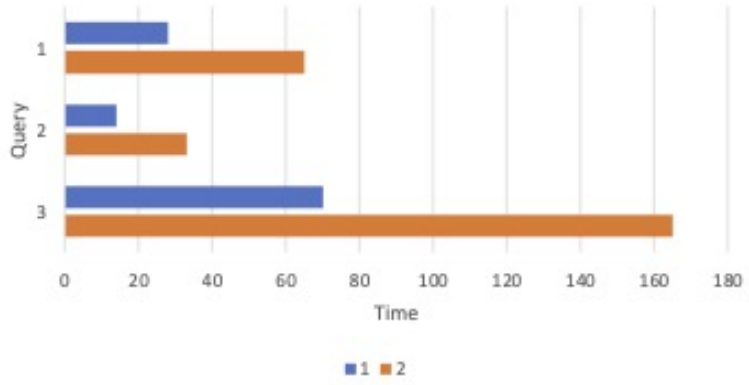


**Fig. 11.** Queries execution time vs. deployment setting and history length.

these queries according to two settings: (1 in the figure) 800 things producing streams through 3 queues: (2 in the figure 800 things and 1 queue deployed

on one node). The good news is that H-STREAM microservices can deal with farms with many stream producers connected to few queues. The combination of streams produced at recurrent paces combined with long histories is done in a "reasonable" execution time in the order of seconds. The query execution cost depends on its recurrence and the history size. The overhead implied by the streams' production pace is delegated to the message passing middleware.

## 6 Conclusions and Future Work

This paper proposes H-STREAM that composes microservices deployed on high-performance computing backends (e.g., cloud, HPC) to process data produced by farms of things producing streams at different paces. Microservices compositions can correlate online produced streams with post-mortem time series. This synchronisation of past and present enables the discovery and model of phenomena or behaviour patterns within intelligent environments. The advantage of this approach is that there is no need for full-fledged data management services; microservices compositions can tailor data processing functions personalised to the requirements of the applications and IoT environments.

Future work consists of developing a microservices composition language that can be used for expressing the data processing workflows that can be weaved within target application logics [1]. We are working on two urban computing projects regarding the modelling and management of crowds and smart energy management in urban clusters.

## References

1. Akoglu, A., Vargas-Solar, G.: Putting data science pipelines on the edge. arXiv preprint arXiv:2103.07978 (2021)
2. Alaasam, A.B., Radchenko, G., Tchernykh, A.: Stateful stream processing for digital twins: Microservice-based kafka stream dsl. In: 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). pp. 0804–0809. IEEE (2019)
3. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink®: consistent stateful distributed stream processing. Proceedings of the VLDB Endowment **10**(12), 1718–1729 (2017)
4. Cardellini, V., Nardelli, M., Luzi, D.: Elastic stateful stream processing in storm. In: 2016 International Conference on High Performance Computing & Simulation (HPCS). pp. 583–590. IEEE (2016)
5. Enrique Arriaga-Varela, Javier A. Espinosa-Oviedo, G.V.S., Pérez, R.D.: Supporting real-time visual analytics in neuroscience. In: Advanced vector architectures for future applications - Book of abstracts. Barcelona Supercomputing Center (2017)
6. Fragkoulis, M., Carbone, P., Kalavri, V., Katsifodimos, A.: A survey on the evolution of stream processing systems. arXiv preprint arXiv:2008.00842 (2020)
7. Golab, L., Özsu, M.T.: Data stream management. Synthesis Lectures on Data Management **2**(1), 1–73 (2010)

8. Krämer, J., Seeger, B.: Semantics and implementation of continuous sliding window queries over data streams. ACM Transactions on Database Systems (TODS) **34**(1), 4 (2009)
9. Lyman, P., Varian, H.: How much information 2003? (2004)
10. Rao, T.R., Mitra, P., Bhatt, R., Goswami, A.: The big data system, components, tools, and technologies: a survey. Knowledge and Information Systems **60**(3), 1165–1245 (2019)
11. To, Q.C., Soto, J., Markl, V.: A survey of state management in big data processing systems. The VLDB Journal **27**(6), 847–872 (2018)
12. Woo, M.Y.: What's the big deal about big data. Engineering and Science **76**(3), 16–23 (2013)
13. Zikopoulos, P., Eaton, C., et al.: Understanding big data: Analytics for enterprise class hadoop and streaming data. McGraw-Hill Osborne Media (2011)