UNIVERSITAT POLITÈCNICA DE CATALUNYA · BARCELONATECH

DOCTORAL THESIS

Scalable Processing of Aggregate Functions for Data Streams in Resource-Constrained Environments

Author:
Álvaro VILLALBA

Supervisor:
Dr. David CARRERA

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Departament d'Arquitectura de Computadors

April 29, 2019

Declaration of Authorship

I, Álvaro VILLALBA, declare that this thesis titled, “Scalable Processing of Aggregate Functions for Data Streams in Resource-Constrained Environments” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

[illegible]

Date: *April 29, 2019*

UNIVERSITAT POLITÈCNICA DE CATALUNYA · BARCELONATECH

Abstract

Facultat d'Informàtica de Barcelona

Departament d'Arquitectura de Computadors

Doctor of Philosophy

Scalable Processing of Aggregate Functions for Data Streams in Resource-Constrained Environments

by Álvaro VILLALBA

The fast evolution of data analytics platforms has resulted in an increasing demand for real-time data stream processing. From Internet of Things applications to the monitoring of telemetry generated in large data centers, a common demand for currently emerging scenarios is the need to process vast amounts of data with low latencies, generally performing the analysis process as close to the data source as possible. Devices and sensors generate streams of data across a diversity of locations and protocols. That data usually reaches a central platform that is used to store and process the streams. Processing can be done in real time, with transformations and enrichment happening on-the-fly, but it can also happen after data is stored and organized in repositories. In the former case, stream processing technologies are required to operate on the data; in the latter batch analytics and queries are of common use.

Stream processing platforms are required to be malleable and absorb spikes generated by fluctuations of data generation rates. Data is usually produced as time series that have to be aggregated using multiple operators, being sliding windows one of the most common abstractions used to process data in real-time. To satisfy the above-mentioned demands, efficient stream processing techniques that aggregate data with minimal computational cost need to be developed. However, data analytics might require to aggregate extensive windows of data. Approximate computing has been a central paradigm for decades in data analytics in order to improve the performance and reduce the needed resources, such as memory, computation time, bandwidth or energy. In exchange for these improvements, the aggregated results suffer from a level of inaccuracy that in some cases can be predicted and constrained.

This doctoral thesis aims to demonstrate that it is possible to have constant-time and memory efficient aggregation functions with approximate computing mechanisms for constrained environments. In order to achieve this goal, the work has been structured in three research challenges.

First we introduce a runtime to dynamically construct data stream processing topologies based on user-supplied code. These dynamic topologies are built on-the-fly using a data subscription model defined by the applications that consume data. The subscription-based programing model enables multiple users to deploy their own data-processing services.

On top of this runtime, we present the Amortized Monoid Tree Aggregator general sliding window aggregation framework, which seamlessly combines the following features: amortized $O(1)$ time complexity and a worst-case of $O(\log n)$ between insertions; it provides both a window aggregation mechanism and a window slide policy that are user programmable; the enforcement of the window sliding policy exhibits amortized $O(1)$ computational cost for single evictions and supports bulk evictions with cost $O(\log n)$; and it requires a local memory space of $O(\log n)$. The framework can compute aggregations over multiple data dimensions, and has been designed to support decoupling computation and data storage through the use of distributed *Key-Value Stores* to keep window elements and partial aggregations.

Specially motivated by edge computing scenarios, we contribute Approximate and Amortized Monoid Tree Aggregator (A^{2}MTA). It is, to our knowledge, the first general purpose sliding window programable framework that combines constant-time aggregations with error bounded approximate computing techniques. A^{2}MTA uses statistical analysis of the stream data in order to perform inaccurate aggregations, providing a critical reduction of needed resources for massive stream data aggregation, and an improvement of performance.

Acknowledgements

Ara entenc el que vol dir trobar-se sobre les espatlles de gegants.

El treball presentat aquí i la meva conseqüent carrera professional són el fruit d'una cadena de casualitats que m'han apropat a persones extraordinàries, a les que estic molt agraït per moltes raons. Aquesta cadena de casualitats té varies fites especialment importants.

En acabar les assignatures d'enginyeria en informàtica i motivat pels excel·lents projectes finals d'amics que ja eren enginyers, vaig decidir el que volia fer pel meu projecte final de carrera, seguint les meves inquietuds i amb intenció de divertir-me: un manegador d'esdeveniments provocats per sensors, telemetria i programes de tercers. No hagués dit mai que la temàtica que estava escollint llavors seria la llavor d'un doctorat. El professor que em va acceptar el projecte sense posar cap inconvenient va ser el Juanjo Costa. Aquest projecte és un èxit en el meu expedient acadèmic, i això va ser gràcies als seus consells i indicacions setmanals.

Entrant ja al món laboral com a enginyer, el Juanjo em va recomanar que em mirés una oferta de feina publicada al taulell d'anuncis del Barcelona Supercomputing Center, i que portava un professor amb el que em va prometre que aprendria moltíssim: David Carrera.

Amb el David (director d'aquesta tesi) vaig entrar en un projecte per fer computació Big Data en streams de dades, que casualment era la evolució natural del que vaig començar a fer en el projecte final de carrera. A les dos setmanes de contractar-me, el David em presentava a IBM com l'expert en processat d'streams pel projecte conjunt que començàvem. Ja no havia marxa enrere, havia de tornar-me un expert o aquesta gent tant important s'ensumaria alguna cosa. Temps més tard, després de passar pel Síndrome de l'Impostor i d'aprendre molt, l'escena es repetia amb Cisco i després amb Microsoft. En molt poc temps vaig passar a treballar amb enginyers de primer nivell en projectes importants, tant de grans empreses com del propi BSC.

Treballant en aquests projectes, vaig tenir una idea que podia no dur enlloc: un concepte nou d'agregadors eficients per streams de dades. Ho vaig presentar al David, i ràpidament em va donar llum verda i els recursos per a que pogués desenvolupar-ho. Ara aquella idea s'ha convertit en una patent que és la contribució central d'aquesta tesi, la meva contribució inicial a una empresa de la que en sóc cofundador, i també un mal de cap que s'ha tornat crònic. Malgrat tot això, la raó per la que estic més agraït al David és per obrir-me els ulls i fer-me canviar d'ecosistema de càmeres fotogràfiques. Aquí en queda constància.

Per suposat, paral·lelament a tot això, he rebut molta ajuda i suport tant dels meus companys de la carrera com dels meus companys de feina. Amics meus, són els millors enginyers que he conegut i m'han obligat a millorar per poder apropar-me al seu nivell: Cesare, que ens vam conèixer el dia que vam començar els dos a treballar al BSC, i que en poc temps es va tornar un molt bon amic; Marcelo i Nicola, amb els que he compartit unes quantes cerveses mentre ens convencíem mútuament de que això del doctorat paga la pena; Josep Lluís i Alberto, que m'han ajudat amb la part d'estadística i a posar sobre paper les idees que tenia al cap; Òscar, Tom i Sergi, ex-companys al BSC i actuals companys en la recent aventura de fer una empresa amb el David, gràcies a ells ha sigut possible tancar aquest projecte al mateix temps que en començava un altre encara més ambiciós; I en general tots els companys que han passat pel C6-E201 i el K2M-S204b.

Vull també fer una menció especial a la persona amb la que he compartit tota aquesta experiència dia a dia, el meu principal suport i qui més estimo: Eli. Ha estat al meu costat quan em semblava que la lògica m'havia abandonat, ha sigut comprensiva quan sortíem a prendre algo però jo encara tenia el cap treballant, i ha cobert les meves absències quan se m'apropaven deadlines. Sense ella hagués perdut totalment la cordura fa temps.

Finalment vull agrair als meus pares pels valors que m'han inculcat durant tota la meva vida, i perque sempre han potenciat la meva curiositat i el meu interès per la ciència i la tecnologia.

Gràcies a tots.

Contents

List of Figures

List of Tables

List of Abbreviations

A^{2}MTA	**A**pproximate & **A**mortized **M**onoid **T**ree **A**ggregator
ADWIN	**Ad**aptive **Win**dow
AMTA	**A**mortized **M**onoid **T**ree **A**ggregator
API	**A**pplication **P**rogramming **I**nterface
ASA	**A**zure **S**tream **A**nalytics
CAP	**C**onsistency, **A**vailability & **P**artition (tolerance)
CRUD	**C**reate, **R**eplace, **U**pdate & **D**elete
DABA	**D**e-**A**mortized **B**anker's **A**ggregator
DAG	**D**irected **A**cyclic **G**raph
DPP	**D**ata **P**rocessing **P**ipelines
FIFO	**F**irst **I**n - **F**irst **O**ut
IoT	**I**nternet **o**f **T**hings
JSON	**J**ava**S**cript **O**bject **N**otation
KDA	**K**inesis **D**ata **A**nalytics
KVS	**K**ey-**V**alue **S**tore
LMTA	**L**ogarithmic **M**onoid **T**ree **A**ggregator
MQTT	**M**essage **Q**ueuing **T**elemetry **T**ransport
MTA	**M**onoid **T**ree **A**ggregator
OSS	**O**pen **S**ource **S**oftware
RA	**R**eactive **A**ggregator
REST	**Re**presentational **S**tate **T**ransfer
SO	**S**ervice **O**bject
SPL	**S**tream **P**rocessing **L**anguage
SQL	**S**tructured **Q**uery **L**anguage
STOMP	**S**imple (or **S**treaming) **T**ext **O**riented **M**essage **P**rotocol
SU	**S**ensor **U**pdate

SWAG	**S**liding-**W**indow **Ag**gregation
WO	**W**eb **O**bject
WSP	**W**indow **S**lide **P**olicy

Chapter 1

Introduction

1.1 Motivation

Over the last years, Internet of Things (IoT) and Big Data platforms are clearly converging in terms of technologies, problems and approaches. IoT ecosystems generate a vast amount of data that needs to be stored and processed, becoming a Big Data problem. IoT devices and sensors generate streams of data across a diversity of locations and protocols that in the end reach a central platform that is used to store and process it. Processing can be done in real time, with transformations and enrichment happening on-the-fly, but it can also happen after data is stored and organized in repositories. In the former case, real-time processing technologies like Storm [19] [96] are required to operate on the data; in the latter batch processing like Hadoop [14] is of common use. Stream processing prioritizes low latency above throughput and is continuously calculating the results, in contrast to batch processing that gives preference to throughput and runs in larger time spans after accumulating larger amounts of new data. IoT use cases usually involve immediate reaction to event detection, or continuous telemetry monitoring. In such scenarios, low latency is a the priority and stream processing is a clear solution data analytics.

When an entity wants to access a feature's data stream, i.e. last hour average temperature in a location, that entity has two main options to retrieve that data. The first option is to deploy the infrastructure needed to retrieve the data from scratch: sensors and related connectivity. The second option is to get access to an existing data stream from another entity that contains information related to the target feature. From the previous example, a city council might have temperature sensors deployed all over the city generating temperature data streams. If these streams are shared to the interested party and accessible from a stream processing platform, the only thing left to do would be to perform the average aggregation on the stream. It is cheaper to share data streams in a multi-tenant data stream processing platform, rather than deploy the same set of sensors per tenant. Moreover, an entity might be interested in the composition of several streams. Consider an entity interested on the wind chill factor. The wind chill factor is calculated using the wind velocity and the air temperature. This entity could use third party's wind velocity and air temperature streams in order to make a continuous wind chill factor calculation.

The operations performed on the data might not depend on a single tenant, like the owner of the sensors or the owner of the data processing infrastructure. Furthermore, the results from the stream analytics on sensor updates end up being new streams and other third parties might be interested on them. So a pipeline of data stream operations might be performed by a combination of tenants, and would need to grow dynamically while it is running. This is a very demanding environment in which the execution topologies are potentially vast directed acyclic graphs (DAG), with each vertex being an operation that in some cases might be challenging to run in terms of time and space. With such an execution topology, the vertices need to be loaded to memory dynamically whenever they receive an stream update. Otherwise the resources will easily become scarce. We refer to dynamic pipelining as the combination of operations to subscribing to an existing stream on-the-fly while the operations are only loaded when need to compute a stream update.

Aggregate functions operate on extensive amounts of data to produce a single result. Big Data traditionally solves the problem of data aggregation with batch processing. Batch processing uses programming models such as MapReduce with efficient algorithms. Such programming models enable efficient and linear scalable data analytics of massive amount of data with high throughput and fault tolerance. This linear scalability consists on distributing the computation and storage of the data, which by replicating said data we can also obtain fault tolerance. If more computation resources are needed, it can be solved by just adding new computation nodes to the batch processing system.

On the other hand, stream processing is generally used to constantly aggregate relatively small amounts of data because it lacks efficient aggregators for massive data that fits the following requirements for real-time computation:

- Efficient programming model.
- Low-latency incremental computation.
- Fault-tolerance.
- Constantly distributed and replicated data.

Since a data stream is virtually infinite, the data to be aggregated needs to be narrowed down in stream sliding windows, i.e. data from the last year. This also the case of batch processing, where a batch might also contain data from last year. Nevertheless, batch processing provides a single results from a closed set of data, and a new batch is required to produce a new result. Although the throughput is high, the latency to produce this single result is also relatively high. A stream processing sliding window generates a stream of real-time results as the time interval shifts forward, with very low-latency for the computation of each incremental result. As a consequence of the lack of Big Data aggregators for stream processing, the window aggregators are not linearly scalable. Therefore, batch processing is traditionally used for massive data aggregations while stream processing is used for continuous aggregations of constantly changing small data.

In this work, we will demonstrate that it is possible to use batch processing paradigms for stream processing aggregations, getting as a result continuous aggregations of vast amounts of data.

Aside from the scalable and distributed programming models, there are other paradigms that are relevant for the aggregation of Big Data. One of the paradigms widely used in Big Data batch processing is Approximate Computing. Approximate Computing improves the performance of data analytics algorithms and decreases the resources needed. However, the results of Approximate Computing algorithms may have some degree of inaccuracy. The computation strategies in Big Data Approximate Computing are usually software-level approximation rather than hardware, such as memoization, skipping loops in iterations or skipping data elements in an aggregation. The inaccuracy can be predicted, with a margin of error, and controlled or even limited. In a linearly scalable environment, Approximate Computing not only can increase throughput and reduce latencies, but also would reduce the number of computation nodes needed for an aggregation making it cheaper.

We want to avoid to the possible extent that an operation as relevant for data analytics as an aggregate function becomes a bottleneck in a data processing pipeline that will chain the latencies of multiple operations. The aggregate functions require to have an update computation time close to the update input frequency.

The tenant-shared execution environment that we described requires stream operations and their data to be only loaded when needed, and to load a minimal amount of data. Having the data distributed and replicated in a scalable data store frees local resources and makes the aggregation fault-tolerant. If also only a little portion of that data is used on each aggregation, it enables the operation to be loaded fast when needed and frees even more local resources.
The contents exposed in here motivated the following Doctoral Thesis statement:

It is possible to leverage dynamically pipelined topologies to combine scalable stream processing with the approximate computing paradigm to build efficient sliding window aggregators for resource-constrained environments.

In order to achieve this goal, we divided the work into three main contributions. The first one provides a stream processing platform with dynamically pipelined topologies. The second one is a constant-time and footprint efficient framework for general purpose sliding window aggregations. The third contribution applies approximate computing mechanisms to the sliding window framework.

1.2 Contributions and Publications

The contributions of this Doctoral Thesis aim to deliver a scalable and efficient aggregate functions framework for massive data in stream processing. This goal will

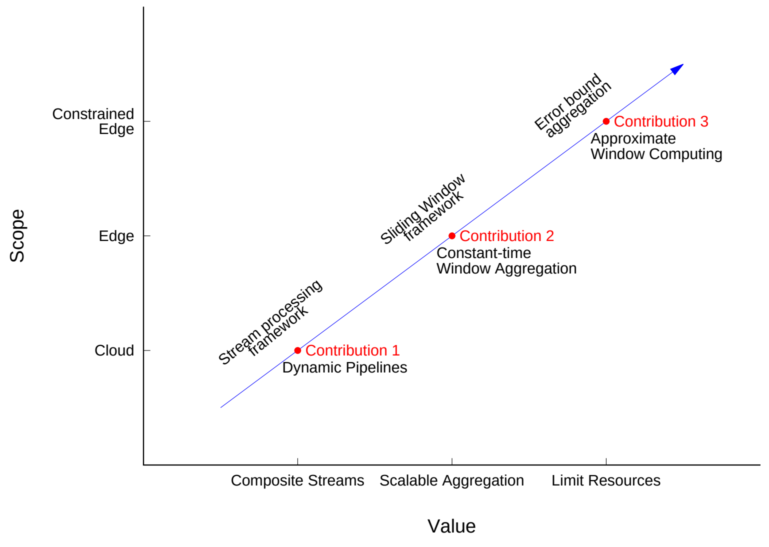


FIGURE 1.1: Contribution's milestones

be achieved by incrementally pushing each contribution towards resource-scarce environments execution, such as Fog or Edge computing deployments. Furthermore, there will be a constant focus on having a fairly simple programming model to define aggregate functions.

The chart in Figure 1.1 shows in its axes the progression of the main two goals of this Thesis. The y axis represents the different increasingly resource-constrained scopes this work covers, while the x axis represents the development in the value it achieves in terms of computational efficiency. Each contribution can be found by crossing the milestones from each goal.

The work in this Doctoral Thesis is divided in the following three main contributions supported by multiple peer-reviewed publications.

1.2.1 Dynamically Pipelined Processing for Composite Data Streams

Devices and sensors generate streams of data across a diversity of locations and protocols. That data usually reaches a central platform that is used to store and process the streams. Processing can be done in real time, with transformations and enrichment happening on-the-fly, but it can also happen after data is stored and organized in repositories. In the former case, stream processing technologies are required to operate on the data; in the latter batch analytics and queries are of common use.

This contribution introduces a runtime to dynamically construct data stream processing topologies based on user-supplied code. These dynamic topologies are built on-the-fly using a data subscription model defined by the applications that consume data. Each user-defined processing unit is called a Service Object. Every Service Object consumes input data streams and may produce output streams that others can consume. The subscription-based programming model enables multiple users to deploy their own data-processing services. The runtime does the dynamic forwarding of data and execution of Service Objects from different users. Data streams can originate in real-world devices or they can be the outputs of Service Objects. Furthermore, a Service Object can subscribe to multiple streams to produce a single composite stream.

The runtime leverages Apache STORM for parallel data processing, that combined with dynamic user-code injection enables multi-tenant stream processing topologies. In this work we describe the runtime, its features and implementation details, as well as a performance evaluation of some of its core components.

This contribution is supported by the following publications:

- Villalba, Á., Pérez, J. L., Carrera, D., Pedrinaci, C., & Panziera, L. (2015). servIoTicy and iServe: a Scalable Platform for Mining the IoT. *Procedia Computer Science*, 52, 1022-1027.

- Villalba, Á., & Carrera, D. (2018, August). Multi-tenant Pub/Sub Processing for Real-Time Data Streams. In *European Conference on Parallel Processing* (pp. 251-262). Springer, Cham.

- Pérez, J. L., Villalba, Á., Carrera, D., Larizgoitia, I., & Trifa, V. (2014, April). The COMPOSE API for the internet of things. In *Proceedings of the 23rd International Conference on World Wide Web* (pp. 971-976). ACM.

1.2.2 Constant-Time Sliding Window Framework with Reduced Memory Footprint and Efficient Bulk Evictions

The fast evolution of data analytics platforms has resulted in an increasing demand for real-time data stream processing. From Internet of Things applications to the monitoring of telemetry generated in large data centers, a common demand for currently emerging scenarios is the need to process vast amounts of data with low latencies, generally performing the analysis process as close to the data source as possible. Stream processing platforms are required to be malleable and absorb spikes generated by fluctuations of data generation rates. Data is usually produced as time series that have to be aggregated using multiple operators, being sliding windows one of the most common abstractions used to process data in real-time. To satisfy the above-mentioned demands, efficient stream processing techniques that aggregate data with minimal computational cost need to be developed.

In this contribution we present the Monoid Tree Aggregator general sliding window aggregation framework, which seamlessly combines the following features: amortized $O(1)$ time complexity and a worst-case of $O(\log n)$ between insertions; it provides both a window aggregation mechanism and a window slide policy that are user programmable; the enforcement of the window sliding policy exhibits amortized $O(1)$ computational cost for single evictions and supports bulk evictions with cost $O(\log n)$; and it requires a local memory space of $O(\log n)$. The framework can compute aggregations over multiple data dimensions, and has been designed to support decoupling computation and data storage through the use of distributed *Key-Value Stores* to keep window elements and partial aggregations.

This contribution is supported by the following publications:

- Villalba, Á., Berral, J. L., & Carrera, D. (2018). Constant-Time Sliding Window Framework with Reduced Memory Footprint and Efficient Bulk Evictions. *IEEE Transactions on Parallel and Distributed Systems.*

- Villalba, Á., & Carrera D. (2017) Distributed data structures for sliding window aggregation or similar applications. *European Patent EP17382202.4*, filed May 30, 2017.

1.2.3 Approximate Sliding Window Framework with Error Control

The principal kind of aggregator for data streams is the sliding window, which defines boundaries on the aggregated stream values. However, data analytics might require to aggregate extensive windows of data. Approximate computing has been a central paradigm for decades in data analytics in order to improve the performance and reduce the needed resources, such as memory, computation time, bandwidth or energy. In exchange for these improvements, the aggregated results suffer from a level of inaccuracy that in some cases can be predicted and constrained.

In this contribution we present the Approximate and Amortized Monoid Tree Aggregator (A^{2}MTA). It is, to our knowledge, the first general purpose sliding window programable framework that combines constant-time aggregations with error bounded approximate computing techniques. It is very suitable for adverse stream processing environments, such as resource scarce multi-tenant edge computing. The framework can compute aggregations over multiple data dimensions, error bounding any of them, and has been designed to support decoupling computation and data storage through the use of distributed Key-Value Stores to keep window elements and partial aggregations.

This contribution is supported by the following publication:

- Villalba, Á., & Carrera, D. (2019). Constant-Time Approximate Sliding Window Framework with Error Control. *22nd IEEE International Symposium On Real-time Computing*.

Chapter 2

Background

This chapter sets a conceptual baseline on data stream processing for the rest of the Doctoral Thesis. The goal is to familiarize the reader to existing concepts that configure this research field.

2.1 General Concepts

In this section we introduce general concepts that are central to stream processing, with the goal to help with the comprehension of the rest of the work.

The concepts listed next are the main objects of discussion, in which everything else is based:

- *Update*: A time-dependent change on the state of a data feature. A data feature refers to information of a specific object or scenario, i.e. current temperature in Barcelona. Updates from the same data feature share the data structure and differ on its values. All updates' data structures usually have timestamp and offset fields, which sets temporal context in the data feature and combined provide a unique ID to the update. Updates are the atomic unit in a data stream.

- *Data stream*: Unbound sequences of ordered atomic updates on the same data feature. E.g., a stream associated to the temperature of a physical device D contains a sequence of updates of such temperature information coming from device D, each update replacing the previous one. A stream emits updates indefinitely, they do not have finite size and lack boundaries.

- *Stream processing*: Transformation of one or more updates streams into one or more derivative update streams. Output stream updates are triggered by input stream updates. Stream processing can be simply transforming updates one to one from input to output, i.e. transforming temperature update values from Fahrenheit to Celsius degrees. However, the output streams can be the result of complex data analytics updates, i.e. anomaly detection in telemetry streams using Kalman filters. One output streams might aggregate several updates from several input streams.

- *Stream operator*: From a high level perspective, stream processing is performed using atomic stream operators. Stream operators are ideally low-latency operators for stream updates. The number of streams or updates from those streams computed as operands depends on the operator. Examples would be transform, filter, or window aggregation.
- *Partition or Shard*: Stream processing can parallelize the computation at three levels; at stream level, partition level and operation level. Different streams run in parallel as they are independent from each other. Operations can have their inner mechanisms to also run in multiple threads to speed up its execution. Partitions are stream divisions by some criteria that run in parallel. However, the same operations are applied to all the partitions from the same stream. Strong ordering between partitions can only be guaranteed by buffering them and performing a sort algorithm right before merging. Partition division are usually represented by the *Group By* SQL operator.
- *Stream processing node*: In a stream processing platform, pipelined operators can be grouped in different nodes in order to improve the job throughput. Dividing the computation into several nodes improves throughput, but it can add latency because of the transport of updates between nodes.

All the modern scalable and distributed stream processing platforms for Big Data share the same elements and basic structure in their architectures, which are the following:

- *Producers*: External to the platform itself, producers send data streams to be processed to the platform, i.e. readings from a sensor. Therefore, they need connectivity to the platform. Multiple producers can emit updates of the same stream, and one producer can emit updates from multiple streams.

- *Consumers*: Like producers, consumers are external to the platform. They collect the updates generated by the stream processing platform to use them somehow. For example, consumers can trigger actuators from the analysis of sensor streams. Multiple consumers can read the same stream, and multiple streams can be read from one consumer.
- *Queue messaging system*: It is in charge of update communication between consumers/producers and the platform. The communication channels are divided by partitions or topics. Each partition is usually treated as an independent queue, although the in/out policies do not need to be FIFO. Partitions are strongly ordered, and so consumers with a FIFO policy will receive updates in the same order as they were received by the partition. Other policies affect consumers apart from in/out policies, i.e. maximum number of retained messages or retention time interval. Queues messaging systems can be distributed and have partitions and/or partition replicas in different machines.
- *Topology*: The actual stream computation happens in the topology, which is a computation pipelines' directed acyclic graph (DAG). Each node contains a section of the computation that will performed on an update. Nodes run in parallel and can be replicated to improve throughput. The edges between the nodes define how the updates flow between the nodes. A node with multiple output edges will emit updates to a subset of these edges per input update. Pipelines have source nodes that retrieve messages from the input queue messaging system, and sink nodes that publish the updates to the output queue messaging system. Sources can retrieve updates from multiple partitions and sinks can emit updates to multiple partitions.

Figure 2.1 is a general diagram of a stream processing platform architecture and the update flow from producer to consumer.

There is also a set of characteristics that differentiates the different stream processing platforms, that makes them more convenient in specific situations. Some

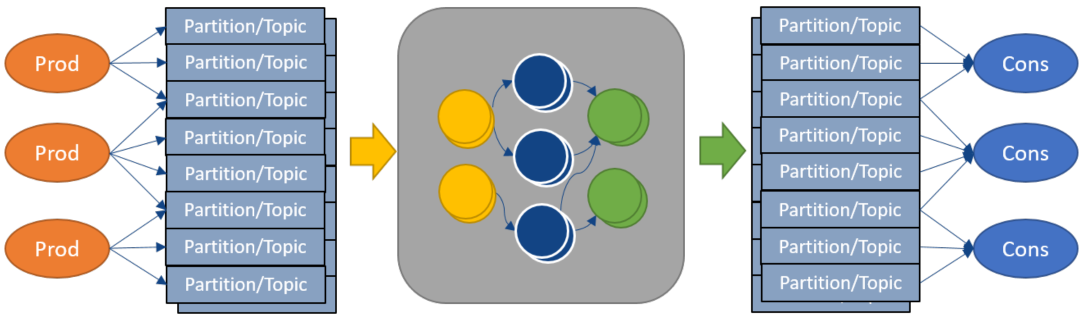


FIGURE 2.1: Big Data stream processing platforms basic architecture

of these characteristics are simply performance metrics like latency and throughput, which generally one is increased by decreasing the other. Furthermore, these metrics are also affected by the following characteristics:

- Strong ordering guarantee: Processed updates can either be emitted in the same order they were generated or not. Update order can be altered by parallel computation of the updates. Most stream processing platforms can guarantee strong ordering if necessary.
- Update processing guarantees: Failures can happen on the pipeline while computing a number of updates. Update processing guarantee defines the implications for the updates being processed in case of a failure, in terms of how many times an update can be processed and emitted to the output queue. There are four options.
  - At least once: It guarantees that each update inserted in the input queue will be processed, but it does not specify how many times. The same output update can be emitted multiple times.
  - At most once: Updates are not processed and emitted more than once, but some updates might be lost due to a failure.
  - Exactly once: All updates are always processed once and only once, regardless of failures.
  - None: There is no guarantee on how many times an update will be processed, the behavior is a best effort reducing the loss and repetition of updates.

- Fault-tolerance method: In order to enforce the strong ordering and update processing guarantees, some update processing control procedures need to be enforced. The following are some examples:
  - Update acknowledgement: Each update that has been processed from a topology node sends back to the previous node an acknowledgement that it has been processed. The source of the topology keeps a backup of all the tuples it generates. Once a source update has received acknowledgements from all generated updates until the sinks, it can safely be discarded from the upstream backup. At failure, if not all acknowledgements have been received, then the source update is replayed. This guarantees no data loss, but does result in out of order updates and duplicate updates passing through the system (at least once processing). Update acknowledgment also works as part of a backpressure handling mechanism, having control on all the updates on-flight in the topology.
  - Micro batches: In order to overcome the complexity and overhead of update-level synchronization that comes with the model of continuous operators that process and buffer updates, a continuous computation is broken down in a series of small, atomic batch jobs (called micro-batches) with a transactional id assigned. Each micro-batch may either succeed or fail. At a failure, the latest micro-batch can be simply recomputed. This method enforces exactly once processing and strong ordering, degrading the computation latency. However, with batch related operators, it can improve throughput.
  - Transactional updates: Atomically log update deliveries together with updates to the state. Upon failure, state and record deliveries are repeated from the log. This guarantees exactly once processing and strong ordering.
  - Checkpointing: This scenario can be considered a composition of micro-batch and transactional update mechanisms. During intervals of updates,

nodes update their state in a distributed snapshot so they can be recovered on failure. For the sinks, they buffer the updates up to the next checkpoint, and then emits them all together.

- Pull/push communication: Update communication between topology nodes can be either pull-based or push-based. Pull communication will require a node to request for more updates from the previous nodes when it is free. This method is not the most latency efficient, but provides a very straightforward backpressure handle mechanism. Furthermore, as messages are passed between nodes as batches, it has good throughput performance. Push-based communication consists on nodes actively sending updates to the following nodes, which will store the updates in input buffers until they are processed. This method is more latency efficient.

- Backpressure handling: Backpressure is the situation in which the input update rate is higher than the processed update rate. Backpressure handling mechanisms rely in buffers and a durable queue-based messaging system. Update drop policies can be applied in such mechanisms.

2.2 Stream Processing Platforms

In the last decade and during the course of this work, there have been great efforts from different fronts on the research and development of stream processing platforms and programming models for scalable big data stream analytics. Two of the most relevant fronts on these efforts are the open-source community and the commercial cloud providers.

The open-source community generated multiple widely-adopted platforms for the computation of data stream analytics. **Apache Storm** [19], first Backtype Storm, has been a popular stream processing platform since its release in 2011. Its runtime works on JVM and it is written in Clojure, a JVM language based on Erlang with its main focus on parallel computation. Storm is a multi-language runtime, thanks to working with an Apache Thrift [20] definition in its core that disengages the topology code's language from the runtime execution. Storm initially based its runtime on ZeroMQ [56] sockets with a *pub-sub* pattern between stream processing nodes, which later became optional and were replaced by default by the more JVM-specific Netty [80] sockets. Storm, like most modern stream processing platforms, is usually paired with Kafka as an input queue. Storms work in a fairly low-level on which each computation node is programmed and the connections between them configured by the user. It guarantees an at-least-once update processing and uses the update acknowledgment fault tolerance method. However, it has a high-level abstraction called Trident which organizes the topology from a query-like instruction from the user. When Trident is used, Storm can guarantee exactly-once update processing and works with micro-batches.

Apache Flink [13] has become a well-known stream processing platform, since its first release in 2015. Running in JVM and written in Scala and Java, it is built upon the Akka toolkit. Topologies can be developed in Scala, Java, Python and SQL through its APIs: DataStream, DataSet and Table. While DataStream provide an API for both bounded and unbounded data streams, DataSet works only for bounded streams. Table API is more high-level and it is programmed in a SQL-like language. Flink also works with the Apache Beam programming model, an open-source unified programming model to generate topologies for both batch and stream processing. Flink provides exactly-once update processing and its fault tolerance is based on checkpointing. Furthermore, **Akka** [5] also provides a stream processing runtime by itself with a rich set of operators. It can be programmed in Scala and Java, and among its characteristics it guarantees an at-most-once update processing.

A more throughput-centered platform can be found in **Apache Spark Streaming** [18]. Spark Streaming shares API with Spark, so streaming jobs are programmed the same way as batch jobs. Spark Streaming jobs can be written in Java, Scala and Python. Instead of performing batch jobs, as Apache Spark is designed to, it reduces the size of the batches to micro-batches and so it can use the same logic. It guarantees exactly-once update processing.

Apache Kafka Streams [15] leverages a Java library for stream processing for Kafka client applications. Instead of using ZeroMQ, Netty or Akka for update communication between stream computation nodes, Kafka is the main messaging bus and not only the input queue system. It provides means to distribute and parallelize a stream topology and operators to perform efficient analytics. It guarantees exactly-once update processing.

The main cloud providers also offer their own stream processing platforms integrated with the rest of their commercial solutions. **Microsoft Azure Stream Analytics** (ASA) is the stream processing service in Microsoft Azure. ASA is programmed in a SQL-like language that considers stream updates rows in a database table. The query runs continuously with each result row being a new stream update. That SQL-like query is compiled to generate a topology that uses Trill [38] as a query processor. ASA can also be executed on the edge in order to improve latency by using the Azure IoT Edge ecosystem. It guarantees at-least-once update processing.

Amazon Kinesis Data Analytics (KDA) is Amazon Web Services' stream data stream processing service. Like ASA, KDA is programmed in SQL which in turn generates the stream processing topology. It offers a selection of pre-built stream processing templates and advanced high level operators. KDA uses an at-least-once processing and delivery model in the event of an application interruption for various reasons.

Google Cloud Dataflow is a cloud service that computes both batches and streams, with Apache Beam as its programming model. Dataflow provides exactly-once update processing guarantee.

IBM Streaming Analytics is a platform that can either run on premise or in IBM Cloud as a service. Its programming language is the Stream Processing Language (SPL), a topology composition specific language with operators that can be programmed in Java or C++. SPL has a rich set of toolkits and efficient operators for data streams along. Like most of the other platforms, IBM Streaming Analytics guarantees at-least once update processing.

2.3 Big Data Architectures

Stream processing can be found in multiple Big Data architectures depending on what problem it is solving. However, there are two main architecture trends called

Lambda Architecture and *Kappa Architecture*. The first one sets aside stream processing, which is used only to provide immediate partial results while using batch processing to perform the goal analytics. *Kappa Architecture* is completely centered on stream processing and considers all data produced as immutable, continuously performing reliable analytics on the input data. We propose an scalable system consistent with *Kappa Architecture* for large computation topologies, with efficient aggregators.

In this section we make a summary of the two architectures.

2.3.1 Lambda Architecture

Batch processing and real-time has been widely considered to complement each other. In summary, real-time technologies are usually seen as fast but complex and unreliable in terms of fault-tolerance and consistency. On the other hand, batch processing is considered the robust option because of its simplicity, but it has a big delay from update to update. The architecture combining both kinds of technologies to process data is known as Lambda Architecture [75] [74]. This architecture aims to avoid the CAP theorem [52] problems when sacrificing consistency.

To avoid (or minimize) the CAP problem, the Lambda Architecture considers three main conceptual characteristics of the data and queries. The first characteristic is that the data is inherently time based. When a new atomic piece of data on a dataset is received, the information it gathers is always true when you consider its temporal context. For instance, an update on the temperature in Barcelona might be 30 degrees Celsius today at 17:00. Later a new update can be 29 degrees Celsius at 18:00. This new update does not invalidate the previous one, it is still true that at 17:00 Barcelona was at 30 degrees Celsius.

That leads to the second characteristic. Being the data time based, the data is immutable. The functions on storage must be CR instead of the typical CRUD. This is not a new approach to deal with data by several processes in parallel. For instance, it is very usual for functional programming languages like Erlang [23] to work with immutable variables, for this same reason.

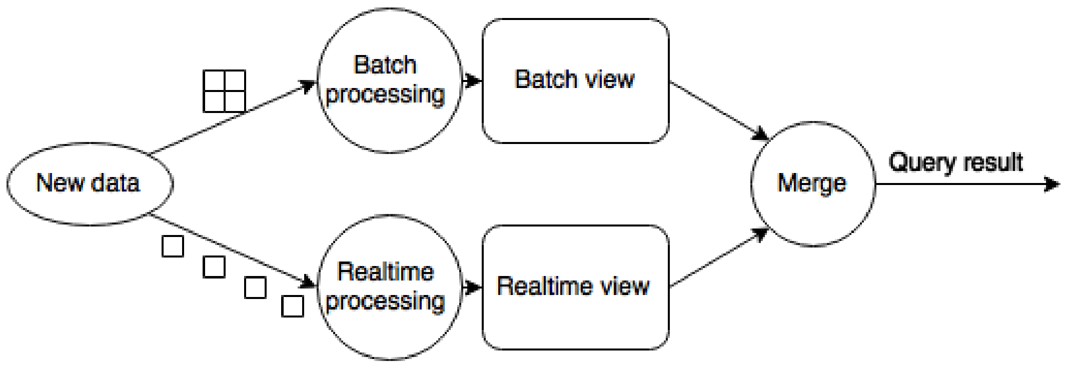


FIGURE 2.2: Computing a query on the Lambda Architecture

The third main characteristic is more related to the queries on the data. Considering a query any function using a data set as an input, the queries must be pre-computed before they are addressed using incremental algorithms. Reliable batch processing technologies like Hadoop take the main responsibility of this part. The description of this architecture defends that the responsibility falls on batch processing because of its robustness given by the simplicity of the databases it relies on. The databases for batch processing like Voldemort [89] only have support for batch writes, avoiding the inherent complexity of random writes.

Batch processing operates on the whole data set with sets of hours of data, and so an atomic update might take hours to be processed. The real-time layer works in parallel with the batch layer to cover the last few hours of data that have not been taken care of by the batch layer. The reasoning behind this is that real-time technologies perform better on little sets of data for solving consistency issues. Usually Storm is used for this purpose with databases like Cassandra [93], Couchbase [42].

This situation leaves the system with two discrete views of the data, batch view and real-time view. The merge of the two views is left for the query invocation time, as it can be seen on Figure 2.2.

The set-up of this architecture, including the replication of code for both the batch processing layer and real-time layer, is up to the developers. However, Twitter released Summingbird [34] as an open-source project. Summingbird works as a Scala abstraction to the Lambda Architecture. Data analytics are written once in Scala and are deployed to Hadoop and Storm. Summingbird leverages Algebird [7] aggregators. Algebird enforces a programming model in which the *Reduce* phase of

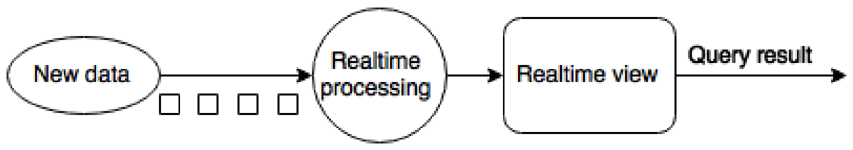


FIGURE 2.3: Computing a query on the Kappa Architecture

MapReduce is an associative function with identical types for the inputs and the output (called semigroup or monoid [73]). Associativity enables the platform for high parallelism and very fast real-time aggregations.

2.3.2 Kappa Architecture

The Lambda Architecture has important downsides. It needs two different platforms two run the same analytics on the data. These analytics are written for two different programming models and so it is duplicated. Summingbird isolates this from the development, but still the single code is translated to the batch and real-time layers. On the one hand this makes it difficult to debug, and on the other hand the two platforms are still there.

The claimed motivations for such an architecture are that real-time processing is generally said to be less powerful and less reliable than batch processing, and that with this mixture of different data systems the CAP theorem is avoided.

The Kappa Architecture [67] is defined as an improvement over the Lambda Architecture, considering the two previous motivations as false. Although it is true that batch processing technologies are much more mature than real-time technologies, that does not mean that real-time technologies need to be more unstable. Furthermore, even if batch-write only databases are simpler, the Lambda Architecture does not achieve to beat the CAP theorem [28].

This criticism to the Lambda Architecture leads into the conclusion that the batch processing layer is not necessary, as it is shown in Figure 2.3. Some implementations of the Kappa Architecture are Samza [16] or Flink [13].

2.4 Operations on IoT data

In this section we identify the kinds of operations that we identified as usual to perform analytics on sensor generated data streams and their derivative streams, and that also have been adopted by high-level stream processing platforms during the course of this work. All these operations need to work efficiently in order to provide an environment suitable for the assimilation of *Kappa Architecture*, and in some cases they have to follow a set of rules in order to keep time consistency on the results.

IoT generated data streams are characterized to be generally produced originally by sensors producing synchronous telemetry from multiple dimensions of specific features. Although the updates from a data stream might not be in the same order as they were sensed, the streams have a strict time order which can be identified by a generation timestamp in each update. There exist mechanisms in order to act upon unsorted data stream updates, such as micro-batches and stream buffers [4]. However, the reordering of data streams is out of the scope of this work.

Most of the operations described in this section are meant to produce new derivative data streams. Each update is operated either alone or with other updates, and therefore produce a new result update that will follow another data stream. New derivative streams can also have operators applied to them. These sequences of operations will be called Data Processing Pipelines (DPP) from now on.

2.4.1 Index and query

A very basic operation on a data set is performing queries to obtain a subset of data from it. In order to do that, the data needs to be properly indexed as it is stored. This kind of operation can be found on any relational database and in most Key-Value stores (KVS). In stream processing, indexing and querying historical data is not generally a desirable scenario. Streams are unbounded, and therefore they will contain virtually infinite data. When used in the context of data analytics, queries usually perform aggregations. In the specific scenario of stream processing, this translates to a pipeline of operations that efficiently produce incremental aggregations from a set of stream updates. Any query with the final purpose to aggregate the data results should be computed while the data is being produced. However, there are frequent and viable kinds of queries related to IoT data stream updates. It usually requires a small window of updates in a stream, like the newer update. For example, it is very usual to query the last update of a stream or all the stream updates produced near an specific location. To be able of doing such queries with a big volume of streams in little time, the updates need to be indexed when received. Elasticsearch [47] or Solr [17] are search engines that provide such functionality.

2.4.2 Filter

In a DPP, it is very common to discard updates that do not follow some parameters. A filter is a set of conditions applied to the input update that, when not fulfilled, the update will not continue on that branch of the processing pipeline. It acts upon a single stream. Sometimes we are only interested on updating values inside a threshold or avoiding clearly erroneous values. For example, detecting sound peaks in decibels or working only with extreme temperatures in order to trigger an action.

The kind of conditions found in a filter are expected to be resolved in $O(1)$ time or $O(k)$, being k the number of data dimensions found in each update. Filters do not require any kind of update memory storage or persistence.

2.4.3 Transform

Similarly to filters, transformations are operations with a single input stream. In the MapReduce programming model, a transformation would be the *map* phase. It applies the same transformation to every input update, generating a new stream. For instance, a transformation can perform unit conversions or reconfigure the data dimensions in the updates in order to perform an aggregation in further stages. The code of the transformation would be provided by the user of the operator.

A transformation is expected to be resolved in $O(1)$ time or $O(k)$, being k the number of data dimensions found in each update. Furthermore, no update memory storage or persistence is required to perform a transformation.

2.4.4 Aggregate

Data aggregation is the most complex operation, because it potentially involves vast amounts of data stream updates. Aggregations performed with the *Lambda Architecture* rely on most of the computational cost of a final aggregation being done with batch processing. However in the *Kappa Architecture* we do not rely on batch processing, and a new total aggregation result must be provided for each new stream update. In this work we will focus on aggregator frameworks, in which the user can define the updates to be aggregated and the specific aggregation to be performed, following an specific programming model. In the MapReduce programming model, all of them would be the *reduce* phase, in contrast to the transformations.

There are different kinds of aggregators, depending on the origin data being aggregated. Some examples of aggregator frameworks would be:

- Accumulator: Aggregates all-time updates. For example, the all-time average temperature from a sensor stream. Its cost can be $O(1)$ using binary associative operations. It only requires one update stored on memory or persisted, for the current aggregation result.
- Sliding Window: It performs an aggregation of a subsequence of updates from the stream, always including the newest update. For instance, it would aggregate last hour temperature from a sensor stream. As it will be demonstrated in this work, its cost can be $O(1)$ using binary associative operations. Furthermore, it has a $O(n)$ memory cost, being n the number of updates aggregated in the window. However, that memory cost can be reduced in exchange of losing aggregation accuracy.

2.4.5 Union

Multiple streams might produce complementary updates on the same feature. For example, multiple temperature sensors might produce updates at different times. The updates will be produced in different streams, but they can be merged in a single richer stream. This require that the streams are equivalent in form, with same data dimensions, types and units. Transformations can be used to adapt different streams to the same format.

This operation only requires two or more streams that will be re-emitted under the same stream, therefore with a negligible cost by itself. Furthermore, it does not require to update memory storage or persistence.

2.4.6 Group

Partitions are divisions from a stream, each with the same inner structure in its updates. Updates from the same stream will be placed on different partitions depending on some criteria. The group operation defines that criteria. One way to define it is the same as in SQL *Group By* operation; placing in the same partition those updates that share the same value in an specific channel.

This operation requires an input stream, which will be partitioned by the user defined criteria. This criteria might imply some computation to combine different channels in the update, but its cost is expected to be $O(1)$. Updates are simply emitted from one partition to another, no memory storage or persistence is required.

2.4.7 Compose

Sometimes called *Zip* operator, it is a function with a closed set of parameters, each parameter a different stream, that combines their last updates to produce a new stream. The input streams do not need to be equivalent, and in some cases the number of these inputs is closed to two. If more input streams need to be operated, then more compose operations can be pipelined. It can be seen as the SQL *Join* operator, where rows from different tables are aggregated into a single new value/row. An example of using the compose operation would be to use a wind speed sensor stream and a temperature sensor stream to produce a wind chill stream.

This operation requires to keep the last update from each input stream in order to use them when a computation is triggered. Being n the number of input streams, it is $O(n)$ memory-wise and it is expected to be $O(n)$ time-wise with a small n in order to not become a bottle neck.

Chapter 3

Dynamically Pipelined Processing for Composite Data Streams

3.1 Introduction

In the last years, Big Data and Internet of Things (IoT) platforms are clearly converging in terms of technologies, problems and approaches. IoT ecosystems generate a vast amount of data that needs to be stored and processed, becoming a Big Data problem. Devices and sensors generate streams of data across a diversity of locations and protocols that in the end reach a central platform that is used to store and process it. Processing can be done in real time, with transformations and enrichment happening on-the-fly, but it can also happen after data is stored and organized in repositories.

This situation implies an increasing demand for advanced data streams management and processing platforms. Such platforms require multiple protocols support for extended connectivity with the objects. But also need to exhibit uniform internal data organization and advanced data processing capabilities to fulfill the demands of the application and services that consume these streams of data.

To provide answer to this growing demand, ServIoTicy[1] is a state-of-the-art platform for hosting real-time data stream workloads in the Cloud. It provides multi-tenant data stream processing capabilities, a REST API, data analytics, advanced queries and multi-protocol support in a combination of advanced data-centric services. The main focus of ServIoTicy is to provide a rich set of features to store and

[1]servioticy.com

process data through its REST API, allowing objects, services and humans to access the information produced by the devices connected to the platform. ServIoTicy allows for a real time processing of device-generated data, and enables for simple creation of data transformation pipelines using user generated logic. Unlike traditional service composition approaches, usually focused on addressing the problems of functional composition of existing services, one of the goals of the ServIoTicy is to focus on data processing scalability. Other components that can be connected to ServIoTicy provide added capabilities to automatically create compositions of high-level services using existing tools [84].

The core of the ServIoTicy runtime relies on a novel programming model that allows users to dynamically construct data stream processing topologies based on user-supplied code. These topologies are built on-the-fly according to a data subscription model defined by the applications that consume data. Once a stream subscriber finishes its work, it is freed from the platform until it is needed again. Each user-defined processing unit is called a Service Object (SO). Every Service Object consumes input data streams and may produce output streams that others can consume. Data streams can originate in real-world devices or they can be outputs of Service Objects deployed in the platform.

Advanced streaming and analytics platforms such as ServIoTicy are complex pieces of software that integrate a large set of components under the hood. They hide their complexity behind simple REST APIs and multi-protocol channels, but the reality is that their deployment and configuration is complex. ServIoTicy leverages Apache STORM runtime for parallel data processing, that combined with dynamic user-code injection provides dynamic stream processing pipelining.

We provide insights on the performance properties of ServIoTicy as an starting point for the construction of advanced cloud provisioning strategies and algorithms. The work presented here focuses on the processing topologies built in ServIoTicy, although some details about other platform components are also provided.

Security is one of the main concerns on IoT platforms because they deal with big amounts of sensitive data. Although the applied security policies are not in the scope of this work, there has been efforts in that matter. Each update contains provenance data including the data owners and the operations that has been applied. The provenance data is used with a security policy manager to decide if an application can make use of the update.

The source code of ServIoTicy is freely available as an open source project[2] in GitHub. The platform is also available for single node testing as a vagrant box, downloadable from a github repository[3].

The main contributions are:

- A technique for user-code injection on a data stream processing runtime that allows for dynamic creation and execution of stream processing topologies. This runtime is the core of the ServIoTicy platform.
- Detail on the operator that composes multiple streams into a single composite stream.
- An insight on the performance of the code-injection technique, including response time end-to-end in a processing pipeline and across stages.

The next sections of the chapter are organized as follows: Section 3.2 introduces the general architecture and components of the platform; Section 3.3 introduces a set of abstractions defined in ServIoTicy for managing data associated to objects; Section 3.4 describes in detail the stream processing runtime of ServIoTicy; Section 3.5 presents the evaluation methodology and the experiments; Finally, Section 3.6 goes through the related work and Section 3.7 provides some conclusions and future lines of work.

3.2 Architecture of ServIoTicy

The Front-End of platform is a Web Tier that implements the REST API that sits at the core of ServIoTicy. The API contains parts of the logic of the Service Objects and Data Processing Pipelines, related to authentication, data storage and data retrieval actions. The Stream Processing Topology is responsible for the execution of the code associated to Data Processing pipes as well as the forwarding of data across Service Objects and to external entities (e.g. external subscribers that want data forwarded

[2]https://github.com/serviotic y

[3]https://github.com/servioticy/servioticy-vagrant

on real-time using a push model on top of MQTT or STOMP). Finally, the data Back-End includes the Data Store that provides scalable, distributed and fault-tolerant properties to ServIoTicy, and the Indexing Engine that provides search capabilities across sensors data using different criteria, like timestamps, string patterns or geo-location. In this section we describe in more detail the main properties of each component of the ServIoTicy architecture.

3.2.1 Web Tier

The Web Tier for the REST API is composed of a Servlets Container and a REST Engine. As a HTTP Web Server and Java Servlet container we use Jetty [62]. Jetty is often used for machine-to-machine communications, usually within larger software frameworks. As a JSON processor we use Jackson [61], which is a high-performance suite of data-processing tools for Java, including the flagship JSON parsing and generation library, as well as additional modules. The Jackson Project also has handlers to add data format support for JAX-RS implementations like Jersey.

3.2.2 Stream Processing Topology

The Stream Processing Topology is implemented on top of Apache STORM [96], which is a state-of-the-art stream processing runtime. Out-of-the-box, STORM provides the availability to build topologies composed of spouts (sources of data) and bolts (processing units). Topologies are static after their deployment, and data keeps flowing through their bolts until the topology is stopped. STORM provides auto-scaling capabilities that make it particularly suitable for cloud deployments. Note that in case that a different topology is needed, the user needs to stop the running topology and deploy the new one. This situation will not affect the final platform, as it will be explained in more detail in following sections. The Stream Processing Topology also requires the support of a queuing system that will act as the spout for the STORM topology. In ServIoTicy, this is implemented using Kafka [98].

3.2.3 Data Store

A distributed data store is used to keep track of all the object produced data. For that purpose, CouchBase [42] has been chosen as the data store because it provides the benefits of NoSQL data stores (highly distributed, high-availability properties, scalable), and it is document oriented (which fits well for many different data sources and formats). Couchbase has native support for JSON documents. The definition of all Service Objects in ServIoTicy and their associated streams are stored as JSON documents in Couchbase.

3.2.4 Data Indexing

The search infrastructure to resolve queries is provided by an underlying component that performs high-performance indexing and search operations. In particular Elasticsearch [47] is leveraged as it is one of the most powerful and extended search engines that can be integrated with scalable data back-ends (in particular Couchbase). The integration between Couchbase and Elasticsearch enables full-text search, indexing and querying and real-time analytics for variety of use cases such as a content store or aggregation of data from different data sources.

3.2.5 Multi-Protocol Brokerage

In an attempt to make ServIoTicy platform more accessible to udevices, particularly those with less computing capacity or with more power constraints, the REST API is also reachable using other protocols and transports. In particular, STOMP over TCP and WebSockets, and MQTT over TCP are also available. All these features are implemented in ServIoTicy using a combination of newly developed bridges between components and Apache Apollo [12] as the core message brokering engine.

3.3 Abstractions used in ServIoTicy

Several abstractions are used in ServIoTicy to embrace the different entities involved in the existence of IoT ecosystems.

- Web Object: The platform gathers information from objects, either connected to the Internet or not. The group of objects not directly connected to the Internet (e.g. a bottle of wine with a RFID or NFC tag) will need a proxy to represent them in the ServIoTicy. There is also a group of objects which may have network capabilities, but limited programmability and support for advanced network protocols. These devices, such as simple sensors, still will need the use of proxies to be able to communicate with ServIoTicy. Finally, there is a group of advanced devices (so-called Smart Objects, such as a Smart Phone, tablet, or an Arduino device) that already hold the capabilities to talk to the COMPOSE platform directly. Each one of the mentioned objects (enabled with a communications proxy when needed) is known as a Web Object (WO) in ServIoTicy. Web Objects are physical objects sitting on the edge of the ServIoTicy and capable of keeping for example HTTP-based bi-directional communications, such that the object will be able to both send data to the platform and receive activation requests and notifications. Not all such objects will support the same set of operations, but a minimum subset will have to be guaranteed to make them usable to ServIoTicy.

- Service Object: Service Objects are standard internal ServIoTicy representations of Web Objects. ServIoTicy specifies an API by which it expects to communicate with the Web Objects, in order to obtain data from them, or set data within them. That API can be embedded directly in the Objects or can be provided by a mediating proxy that will connect the Objects to their corresponding ServIoTicy Service Objects. This entity serves mainly for data management purposes and has a well-defined and closed API. That API is needed in order to streamline and standardize internal access to Service Objects, which can in turn represent a variety of very different Web Objects providing very different capabilities. ServIoTicy, in an effort to embrace as many IoT transports as possible, allows Web Objects to interact with their representatives in the Platform (the Service Objects) using a set of well-known protocols: HTTP, STOMP [87] over TCP, STOMP over WebSockets [94], and MQTT [77] over TCP.

- Data Processing Pipeline: A Data Processing Pipeline is a data service and aggregation mechanism, which relies on the data processing and management back-end component to provide complex computations resulting from subscriptions to different Service Objects as data sources. This construct can support pseudo-real time data stream transformations, combined with queries concerning historical data. Data analytics code defined by the user may be provided as well. The end result of a Data Processing Pipeline is inserted into the ServIoTicy registry along with its description and may be used by higher level constructs as yet another kind of Service Object building block. Just like a Service Object, this entity serves mainly for data management purposes and has a well-defined and closed API.

- Subscription: Data subscriptions are a mechanism in ServIoTicy that allow Service Objects, Data Processing Pipelines and external data consumers to get data updates automatically and asynchronously forwarded for further processing.

- Sensor Update: Sensor Updates are the unit of data sent by a Web Object to its Service Object. It contains the different synchronously sensed values and a timestamp that is maintained all over the pipelines. A subscription or a query to a Service Object will get the data in this format.

3.4 Data Processing Pipelines

Service Objects store their associated data in abstractions called *streams*. The unit of data that can be observed for one stream is called a *Sensor Update* (SU). Applications can subscribe to or query data associated to any stream. Streams can be of two different types:

- Simple data streams store data generated in the physical world by a sensing device, assuming that a device with N sensors will generate N streams of data that will be grouped in a Service Object abstraction that represents the device.

- Composite data streams represent operations (aggregate, merge, filter or join, among other possibilities) performed on other data sources (either by devices located in the physical world or by Service Objects existing in the ServIoTicy platform). They can be thought about as a virtual (non-physical) sensor of the SO.

From an API perspective there is no difference between a simple stream and a composite stream, as they both support queries and subscriptions. Therefore, the inputs of composite stream can be streams or other composite streams. These chained transformations of SUs are called *Data Processing Pipelines*.

Listing 3.1 is a snippet from a SO descriptor that illustrates the case of a composite stream that takes temperature reads in Fahrenheit degrees as input SUs and produces temperatures in Celsius degrees as outputs if and only if the temperature is below 0 °C. Note how the *current-value* of the stream is calculated first by transforming the °F into °C. The following sections will describe in more detail the purpose of the elements of this example and their semantics.

LISTING 3.1: Example of data transformation: convert from °F to °C

```
"streams":{
 "frozencelsius": {
  "channels": {
   "temp": {
    "type": "number",
    "current-value": "({$fahrenheit.channels.temp.current-value}-32)/1.8"
} } } }
```

3.4.1 Data Structures

The structure of a Sensor Update that corresponds to a given stream is basically composed of a series of *Channels* associated to the dimensions of the data represented by the stream (e.g. a geo-location stream may contain two channels representing the latitude and the longitude correspondingly), and a timestamp reported by the data source as the time at which the Sensor Update was generated.

The composite stream structure is similar to the structure of a SU. It contains channels, and each channel contains a so-called 'current-value' field that represents the output value that the composite stream will emit after ingesting a new SU, assuming that the output is not filtered. In a SO document, the content of a 'current-value' field is an operator definition. The result of the assignment to 'current-value' will always be numeric, a Boolean, a string or an array of the previous types. It will be stored and emitted to its subscribers.

3.4.2 Stages of the Processing Pipeline

Once a SU reaches a composite stream as one of its inputs, it goes through a number of stages in order to transform it into a new output SU. This process of ingesting a SU and processing it until a new SU is produced can be summarized as the following set of stages:

1. Subscriber dispatching: A sensor update gets into the processing pipeline, along with its origin information. This stage looks for the subscribers of its origin and if they are composite streams, they are requested and sent to the next stage with the SU.

2. Data Fetching: The composite stream may need access to the data stored by other streams that are inputs involved in the data transformation. In each stage, the sources needed by the stream are queried and their data made available for the rest of the stages, altogether with the original SU.

3. Operation: Operators are executed by taking all the SUs extracted from all the data sources, interpreting the operator in the SO and executing it using the extracted SUs and, optionally, operator-related data in memory. For instance, a simple transformation uses plain JavaScript code to finally obtain a single value for the new SU to emit.

4. Store, trigger actions and emit: Finally, the generated SU gets stored and emitted to the stream subscribers. Additionally, in this final stage, actions to be sent back to SOs are triggered. Such actions will end up being sensor actuations that will be driven through the WOs that embed the actual physical objects.

In ServIoTicy, basic physical object actuation is driven through SOs. When a SO gets an action invoked through the SO actions API, the action is initiated on the corresponding WO, that will act as a proxy for the physical actuator. If a user needs to be able to manually request the execution of a composite action (involving multiple SOs), it is necessary to create a SO that includes the desired action and references to the individual SOs representing each of the physical objects to be actuated, so that the composite action can be properly triggered.

3.4.3 Design Principles

The data processing pipelines introduced in this work are intended to be scalable in accordance with other works found in the literature [88]. In particular, the key design principles considered for the data processing pipelines were:

- Event-driven: A new SU calculation is triggered in a stream when it receives a SU.
- Lock-free: A stream that needs of several different SUs to generate a new one will not lock until all of them are received. It makes use of the received SU, and queries the last SUs from the other needed streams.
- Real-time data processing oriented: Each new SU is processed individually without waiting for a batch.

The approach followed by ServIoTicy is an asynchronous model for which only one of the sources needs to issue a sensor update to trigger the processing of the composite stream. It enforces a high rate of updates and avoids locking the generation of new updates because one sensor is idle. This situation would lock an entire pipeline.

Figure 3.1 illustrates the actual approach implemented in ServIoTicy using a lock-free scalable model. An update owned by stream *B* is sent to ServIoTicy through the API and is stored. A composite stream is subscribed to the streams *A*, *B* and *C*, and so it receives their outputs SUs as inputs. It generates a new SU, stores it and becomes sent to further composite streams if any. In this particular case, the generation of *SU 4* also requires of *SU 1* and *SU 3*, so the composite stream queries them to streams *A* and *C*. A single event (receiving a SU) generates a single output in the composite stream.

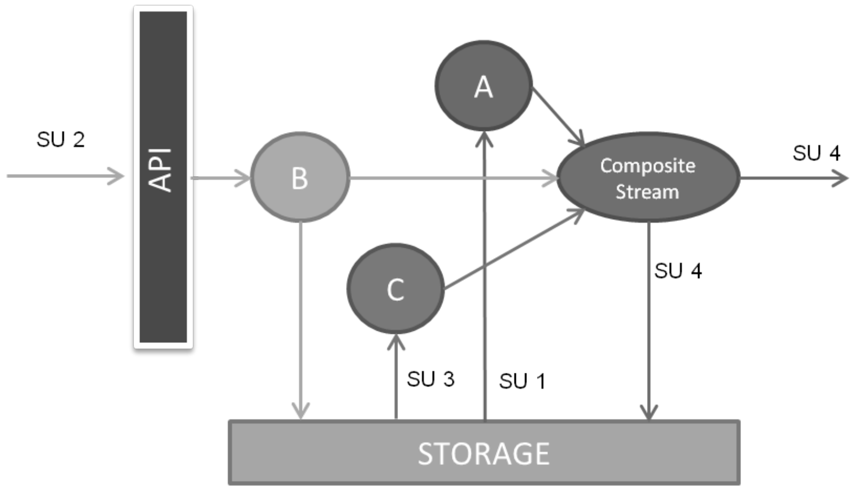


FIGURE 3.1: Lock-free asynchronous model used in ServIoTicy

3.4.4 Time, Data Consistency and Efficiency

A composite stream can take as inputs the most recent SU from any stream declared in the platform, either from its own Service Object or from any other Service Object. In the context of a particular data stream, that receives SUs as inputs and stores data associated to its outputs in the platform, some restrictions need to be in place to keep chronological consistency of the data being produced by a given composite stream.

More formally, let S be a composite stream that takes as inputs the SUs generated by N streams. Let $su_i^{t_i}$ be the the most recent SU associated to the i^{th} stream that is a data source for S, where $0 \leq i < N$, and let be t_i the associated timestamp to $su_i^{t_i}$. Also, let $su_s^{t_s}$ be the most recent SU associated to the stream S. Notice that it is possible that $\exists_i$ such that $i = s$ if S consumes its own previously generated data to produce new outputs.

Then we can define $SU_{s,in}^t = \{su_0^{t_0}, su_1^{t_1}, \ldots, su_{n-1}^{t_{n-1}}\}$ as the set of N inputs that S will use to produce one new output $SU_{s,out}^t$ with timestamp t. This output will be defined as a function $SU_{s,out}^t = f(SU_{s,in}^t)$ that is user-defined.

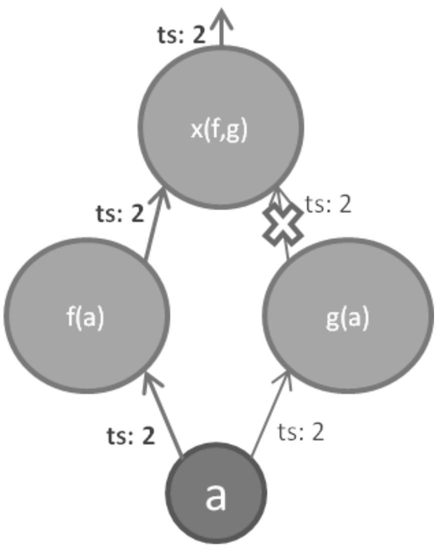
Given these definitions, ServIoTicy needs to guarantee that the function f is calculated (and an output $SU_{s,out}^t$ emitted) only once for the same set of input values, and that at least one of the SUs in $SU_{s,in}^t$ needs to be updated (with a more recent timestamp) to trigger the computation again. Furthermore, it is necessary that the set $SU_{s,in}^t$ satisfies that $\exists su_i^{t_i} \in SU_{s,in}^t$ such that $t_i > t$ to initiate the computation of f to emit $SU_{s,out}^t$.

This restriction can be enforced by checking all the elements of $SU^{t}_{s,in}$ everytime that an element of the set is updated. But this approach can result in performing large amounts of costful operations just to decide that the conditions were not satisfied and that no new output needs to be emitted.

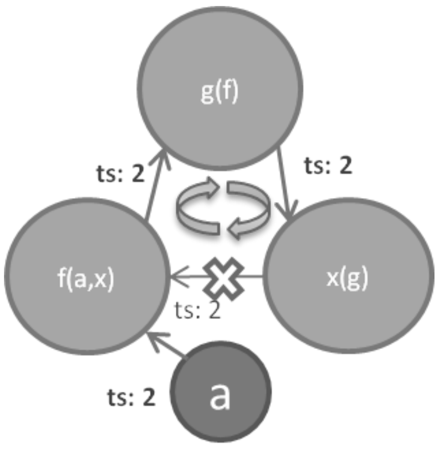
LISTING 3.2: Algorithm used to generate new updates

```
def generateNewUpdate(receivedUpdate, currentStream, streamSubscriptions):
        previousSelfUpdateFuture = getLastUpdateAsync(currentStream)
        originStream = receivedUpdate.getStream(receivedUpdate)
        streamSubscriptions.remove(originStream)
        queriedUpdatesFuture = getLastUpdatesAsync(streamSubscriptions)

        // Block to receive the stream last update
        previousSelfUpdate = previousSelfUpdateFuture.get()
        if receivedUpdate.getTimestamp() <= previousSelfUpdate.getTimestamp()):
                return null

        // Block to receive the remaining updates
        queriedUpdates = queriedUpdatesFuture
        lastUpdates = [receivedUpdate, previousSelfUpdate]
        lastUpdates.appendAll(queriedUpdates)

        // Obtain highest timestamp from the updates
        timestamp = receivedUpdate.getTimestamp()
        for update in lastUpdates:
        if update.getLastUpdate() > timestamp :
                timestamp = update.getLastUpdate()

        streamCode = currentStream.getCode()
        newUpdate = executeCode(streamCode, lastUpdates, timestamp)

        return newUpdates
```

To mitigate this problem, ServIoTicy relaxes the previously stated restriction to the form $t_j > t$ where $0 \leq j < N$ and $su^{t_j}_{j}$ is the actual element in $SU^{t}_{s,in}$ that triggered the computation. This relaxation is possible because if an element exist in the set other than the one triggering the computation that has a more recent timestamp than t, then this it is very unlikely that this element has been computed before in time, because then t would have to be as recent as its timestamp. Otherwise, if the element with more recent timestamp has not yet triggered the computation, then it



(A)



(B)

FIGURE 3.2: Old data discard

means that the SU has been stored for the source stream and it must be awaiting in a queue its time to be processed, and therefore it will trigger the computation soon.

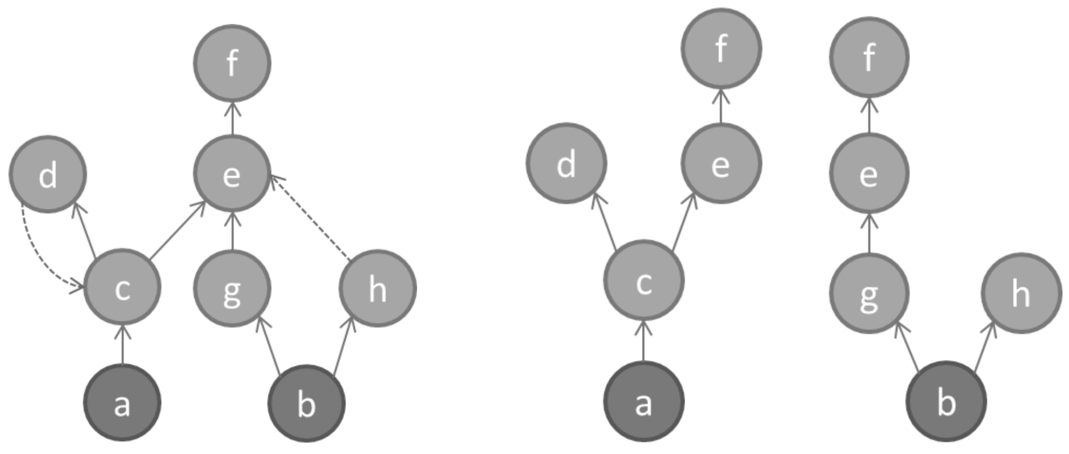
Listing 3.2 summarizes this time-consistency keeping algorithm.

3.4.5 Execution trees of the Data Processing Pipelines

The structure of a pipeline created using the ServIoTicy subscription model is by definition a directed graph. In practice, though, it behaves more like a set of trees. The reasoning behind this statement is discussed in this section.

When an update reaches a stream, if it is newer than the last generated update, the computation will be triggered. But if the received update is as new as the last generated update, the computation will be discarded. Consider a stream that has several inputs and they originally come from the exact same entry stream to the pipeline (source). When one of the inputs receives an update, at some point all the other inputs will receive an update with the same timestamp and the subsequent computations will always be discarded. Only the first update to reach the stream will trigger the computation. An example of this situation can be seen in the Figure 3.2a.

Suppose that all the streams on this pipeline have a SU with timestamp 1 in their historic data. *a* is the only source of the pipeline, which has two streams subscribed, *f(a)* and *g(a)*. Both of them send their results to *x(f,g)*, but the SU from *f(a)* is the



(A) Pipeline digraph (B) Execution trees

FIGURE 3.3: Relation between a pipeline and its execution trees

first one to reach *x(f,g)*. The one coming from *g(a)* is discarded because by the time it reaches *x(f,g)*, there already is in the stored data a SU with timestamp 2 that was generated using the SUs from *f(a)* with timestamp 2 and from *g(a)* with timestamp 1.

This situation is equally valid for cycles, shown in Figure 3.2b, as an input closing a cycle shares exactly the same sources as all the other inputs in the stream.

From this reasoning it can be deduced that the set of paths of the triggered computations from a single source will always end up looking like a tree. For example Figure 3.3a represents the graph of a valid pipeline. The computations that would be generated from the subscriptions $d \rightarrow c$ and $h \rightarrow e$ are discarded for the explained reasons. Therefore the execution graphs look like in Figure 3.3b, and updates from *d* to *c* and from *h* to *e* will only be queried.

Another interesting property of a pipeline is the novelty of its generated data, and it is useful for evaluating the quality of a stream. A stream generates novel data when it has an input with a source that no other input of the same stream has. The further a stream is in a path from the last new source addition, the less novel its generated SUs are. For example in Figure 3.3a, *c*, *g*, *h* and *e* are 1 level more novel than *f* and *d*. See that *e* gets data sourced on *b* from two inputs, but theres also another input sourced on *a*. On the other hand *f* and *d* are one vertex away from the most novel source. At the levels of data novelty of this example, getting data from *f* or *d* is not a problem. The problem comes when the distance from the most novel stream is too far away will always take too much time to process an SU that will not

TABLE 3.1: Pseudo-random topologies

Type	Small		Medium		Big	
Id	1	2	3	4	5	6
Max in-degree	9	8	14	16	29	24
Mean in-degree	1.42	1.94	3.54	3.51	5.28	6.18
In-degree std. dev.	2.22	2.63	4.36	5.05	7.43	7.38
Max out-degree	4	7	15	15	25	28
Mean out-degree	1.42	1.94	3.54	3.51	5.28	6.18
Out-degree std. dev.	1.07	2.14	4.59	4.44	7.71	9.48
Edges	30	37	149	151	423	458
Nodes	21	19	42	43	80	74
Sources	11	9	17	18	30	24
Sinks	4	7	15	15	25	28
Density	0.14	0.21	0.17	0.16	0.13	0.16
Connectivity	1	1	1	1	1	1
Edge-connectivity	1	1	1	1	1	1

add much value to what it is already evaluated, and will generate several discarded computations which will end up being time consumed without a result. Novel data means faster dispatch, less noise in the pipeline and more added value on the data.

3.4.6 Runtime implementation and user-code injection

The software that dispatches the incoming SUs and executes the pipelines runs on STORM. STORM topologies are static, but the pipelines can easily change over time, add connections between them, and have arbitrary sizes. For this reason the STORM topology in ServIoTicy runs the stages described in Section 3.4.2, common to all the pipelines to be processed. On the subscribers dispatch stage, the target streams are requested, with the code to be executed in them (previously deployed by the owner of the Service Object using the REST API). In the different execution stages (operators), the JavasScript code related to it is executed on a JavaScript engine. The JavaScript engine used is Rhino.

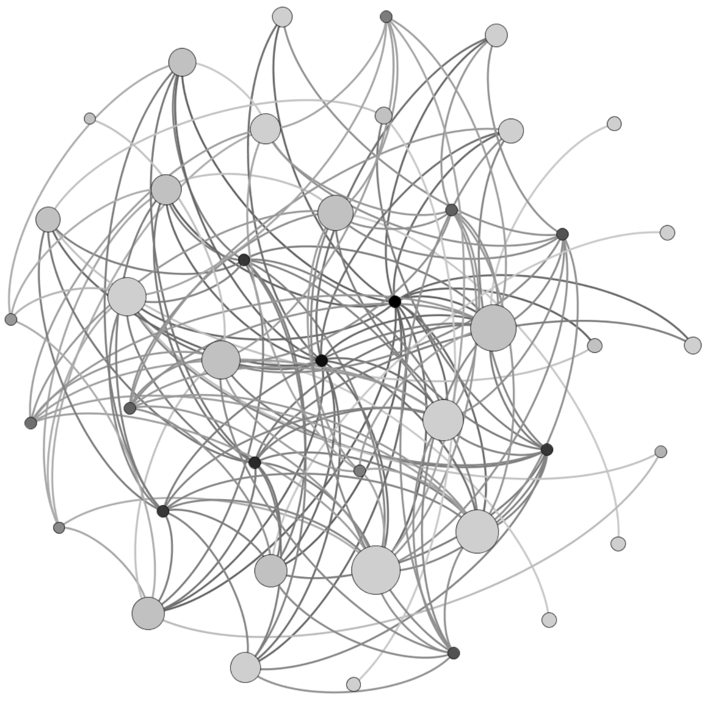


FIGURE 3.4: Representation of topology #3

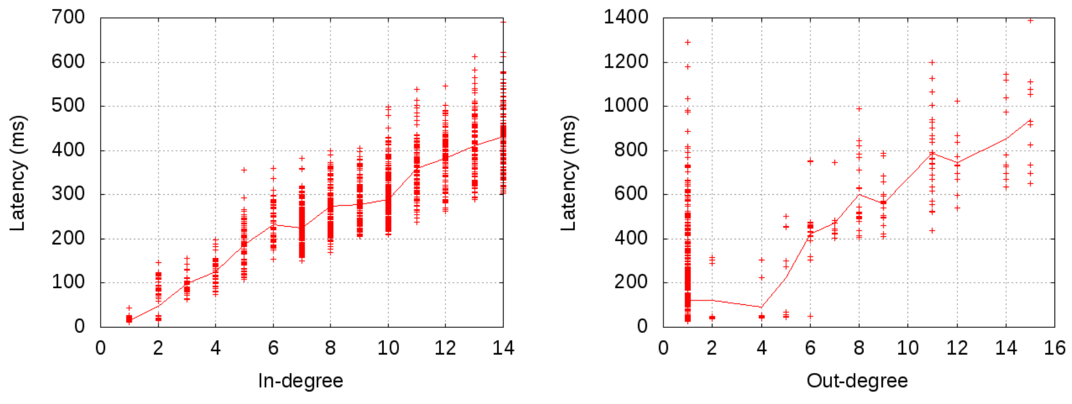
3.5 Evaluation

This section presents a performance evaluation of the implementation of the ServIoTicy Data Pipelines.

3.5.1 Evaluation Methodology and Infrastructure

The evaluation is organized in two different experiments. In Experiment 1, we explored the performance of several randomly-generated topologies. We present here the average results for all of them and the specific results of one illustrative case. In Experiment 2 individual properties of the graphs, like depth of the in and out degree for a DPP, were isolated and studied in more detail. For each experiment, a number of SUs were submitted to the topologies, and we measured the time it took for each SU to be propagated to all the streams that were subscribed directly or indirectly to the SU.

To drive the evaluation we developed a tool to automate the generation and deployment of randomly generated Data Processing Pipelines. The tool provides several control knobs to customize the properties of the topologies being generated. The



(A) Input stage latencies (B) Output stage latencies

FIGURE 3.5: Node latency by degree

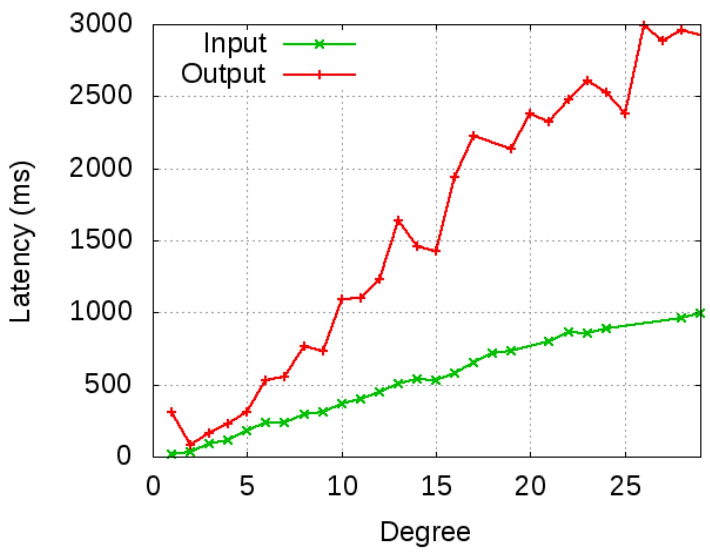


FIGURE 3.6: Stage latency by degree

most relevant controls are the number of streams, the number of composite streams, the number of operands per stream and how the operands are distributed between the streams.

The tests were run on two sets of nodes: one set for running the client emulators and one set for running the servers of the system under test. The 'server' set was composed of 16 two-way 4-core Xeon L5630 @2.13GHz Linux boxes, for a total of 8 cores per node and 16 hardware threads because hyperthreading was enabled. Each 'server' machine was enabled with 24GB of RAM. The 'client' set was composed of 2 two-way 6-core Xeon E5-2620 0 @2.00GHz Linux boxes, for a total of 12 cores per node and 24 hardware threads because hyperthreading was enabled. Each 'server' machine was enabled with 64GB of RAM. All nodes were connected using GbE links to a non blocking 48port Cisco 3750-X switch. The ServIoTicy data processing runtime was deployed on 2 server machines, and 1 client machine was used to generate the SUs. The REST API used the other nodes to host its components. For the data processing pipelines we used Apache STORM v0.9.2-incubating, Kafka v0.9and ZooKeeper v3.4.5.

3.5.2 Experiment 1

For this experiment, we generated six different testing topologies for ingesting data produced by a Service Object. The characteristics of these topologies are summarized in Table 3.1. They can be grouped based on their size (small, medium or large), and we randomly produced 2 samples of each complexity level. Based on our experience, topologies 1 and 2 emulate two realistically sized situations. Topologies 3 and 4 are large cases. Finally, topologies 5 and 6 are extreme cases. A graphical representation of topology number 3 is shown in Figure 3.4. In this figure, dark nodes indicate a high out-degree and big nodes represent high in-degree. The in and out degree related properties are also very relevant for this experiment, as they have a big impact on the metrics taken.

For each data source, 10 Sensor Updates were sent to the platform in sequence: a new update was generated only after the previous pipeline computation was finished. During the topology execution, two metrics were measured for each stream. The first metric is the execution time to perform all the data queries required to complete the processing, named the *input stage*. This metric measures the effect of using several inputs to generate a new update. The second metric is the time difference between the instant at which a new update is emitted and the time at which all subscribers have received it: this metric measures how the topology processing time is affected by the number of subscribers at each stage of the processing pipeline. This is named in this section as the *output stage*.

Other stages were also measured, such as the injected code processing time or the time an update remained in the Kafka queue. The function to generate a new update was always a summation of the inputs, and so had complexity $O(n)$, being

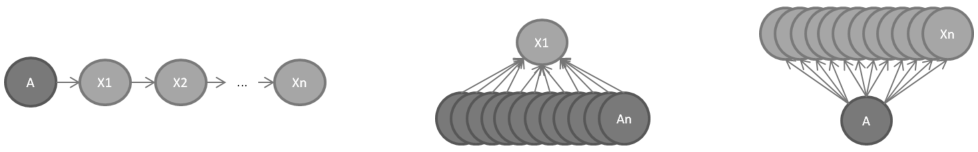
n the in-degree. However, these measures resulted on negligible times and have not been included in the discussion.

Figures 3.5a and 3.5b show all the latencies measured for topology number 3. Each dot in the plot represents one execution of a topology node with a given in-degree or out-degree that corresponds to the value in the X-axis. The average latency for each degree is also drawn in both charts as a solid line. As it can be observed, latency grows linearly with the degree level as some sequential operations are required for each operation. Although the communication is made asynchronous, the stages need to be closed before jumping to the next step for the topology, and therefore it is necessary to wait for all on-the-fly operations to complete at some point, what results in a waiting time that is proportional to the number of initiated operations and therefore the degree of the stage.

Finally, Figure 3.6 shows the average latency on the input and output stages for every related degree, across all six topologies. As it can be observed, the latency of both the input and output stages grow linearly, but in a higher pace in the output stage. While the in-degree latencies look almost the same to Figures 3.5a, the out degree grows faster. The reason for this worse performance is that this Figure reports average values that are affected by the higher latencies of the bigger topologies. Therefore, the time of the output stage not only depends on the out-degree, but also on the total size of the topology. And in particular, as it will be shown in the next experiment, the topology length is the most important factor that affects the performance of the topologies. The larger the topology is, the more operations are run in parallel in the topology and therefore the largest the response times of the components, resulting in a slightly higher latency to complete the processing of an update.

3.5.3 Experiment 2

Following the results of Experiment 1, a second experiment was performed to separately stress the importance of the in degree, the out degree and the length of a topology path. The latter measure is also stressed because it can not be parallelized, and so affects greatly to the overall topology execution.



(A) Length

(B) In-degree

(C) Out-degree

FIGURE 3.7: Types of tested pipeline, each one maximizing a property

For the second experiment, three groups of 100 pipelines were deployed, each one emphasizing one of the following main properties shown in Figure 3.7:

- *Length*: The length of a pipeline is the maximum number of composite streams from one of the sources to any sink. It affects the performance of the pipeline because each one of these streams depends on the result of the previous one, so there is no possible parallelism.
- *Out-degree*: A pipeline's out-degree is the average number of subscribers (operators) its streams have. This is directly affected by the parallelism, the less available threads on the machines the more it will influence negatively the performance.
- *In-degree*: The in-degree is the average number of subscriptions (operands) its streams have. It alters the performance of the execution of a single stream, mainly. The reason is that having a big amount of operands in a composition function means that there are more SU queries to perform. The impact on the performance of the in-degree will depend on the number of available threads, because the set of queries are asynchronous.

Each pipeline in a group exhibits a different number of streams, ranging from 2 to 101. In the case of the 'in-degree' type, the pipelines ranged from 1 source and 1 sink to 100 sources and 1 sink. 'Out-degree' type ranged from 1 source and 1 sink to 1 source and 100 sinks. Finally, the 'length' type goes from 1 source and 1 sink only to 1 source and 1 sink with 99 intermediate chained composite streams. This makes 300 pipelines tested.

For each pipeline, 10 Sensor Updates were sent to the platform, at a rate of one SU per second. During the time all SUs were propagated, several metrics were collected

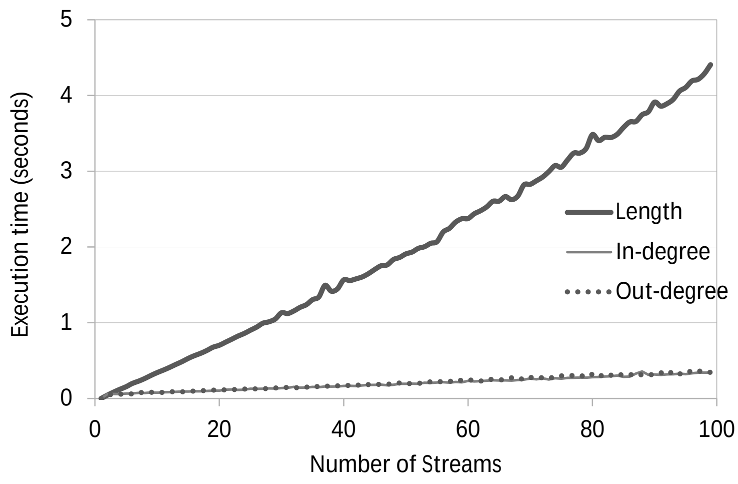


FIGURE 3.8: Time to dispatch a SU through an entire topology.

on the ServIoTicy runtime to determine the delays introduced at each stage and the end-to-end time to process every SU generated.

Figure 3.8 shows the average total execution time of all the pipelines, for each one of the 3 types of pipelines considered. As it can be observed, for the three cases the execution time grows linearly with the number of streams.

As it was expected, the time to propagate a SU through the entire pipeline grows significantly more for the 'length' pipelines because they can not take advantage of any parallelism: all streams are calculated sequentially because they contain sequential data dependencies that can not be skipped.

For the in-degree and out-degree pipelines, it can be observed that the there is almost no difference on how the execution time is affected. In comparison with Figure 3.5, the latencies are significantly lower. Specially in the case of the out-degree, that in this last experiment had a mean latency of 350ms to complete a pipeline with out-degree 100. Notice that in Figure 3.5, the mean latency value for the output stage with out-degree 15 is 950ms. Yet the bigger topology on this experiment is not far from the one of Figure 3.5 in terms of subscriptions, and is bigger in terms of streams. The output stage of a stream is then highly affected by the longitude of the pipeline, and not by the overall size as concluded on the first experiment. Nodes with distance 1 from the source will end up competing for resources with nodes with distance higher than 1 if the initial out-degree is high enough. There is room for improvement by prioritizing nodes near to the sources, otherwise some paths on the pipeline will be faster than others.

3.6 Related Work

In the last years several stream processing platforms have emerged, being Storm [96] the most popular and it is used in this contribution as a platform runtime. Storm is a distributed, reliable, and fault-tolerant stream processing system, which was open sourced by Twitter after acquiring BackType and now distributed by the Apache Software foundation. ZeroMQ or Netty are the messaging interfaces between the computation units. In the last versions multi-tenancy was added in terms of several tenants deploying isolated topologies. This topologies are always in memory whether are being used or not, and there is not data subscription between tenants. Also open-source and distributed by the Apache Software Foundation are Apache Samza [65] and Apache Flink [13] and Apache S4 [81]. Apache Samza uses Kafka for the whole messaging between the computation units and YARN for resource management. Apache Flink is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. It has two APIs, one for data streams and another for data sets or batch processing. Flink also bundles libraries for domain-specific use cases like complex event processing and machine learning. Apache S4 is an already deprecated project started by Yahoo with a very similar topology based philosophy to Storm and an architecture resembling the Actors model. Microsoft Research developed a proprietary solution for complex event processing called StreamInsight [8]. It also leverages a programming model for temporal data streams, operator algebra and continuous queries. Other relevant foundations on stream processing in real-time from Microsoft come the CEDR [25] project. It is centered in the problem of keeping time consistency on event streaming. Other well known research related projects on data streams are Aurora [1] and its forks Medusa [24] and Borealis [2]. None of this projects are maintained anymore. From the perspective of data stream sharing, StreamGlobe [71] offers a Grid Computing solution using a P2P approach. It consist then in stream sharing between machines but not multi-tenancy.

Data Centric view of the IoT is not something new for ServIoTicy as it was widely covered in the survey presented in [85]. What ServIoTicy uniquely provides is an open source solution that challenges the features of commercial solutions such as Xively [99] and Evrythng [48], while extending their capabilities with the ability to inject user-defined code into its stream processing runtime.

There are other open source platforms for IoT in the market, but they are focused on other aspects of the Internet of Things. The DeviceHive [44] project offers a machine-to-machine (M2M) communication framework for connecting devices to the IoT. It includes easy-to-use Web-based management software for creating networks, applying security rules and monitoring devices. Devicehub.net [45] is a cloud-based service that stores IoT-related data, provides visualizations of that data and allows users to control IoT devices from a Web page. The IoT Toolkit [60] project provides a variety of tools for integrating multiple IoT-related sensor networks and protocols. The primary project is a Smart Object API, but it also aims to develop an HTTP-to-CoAP Semantic mapping. Mango [45] is a popular open source Machine-to-Machine (M2M) software, which is web-based and supports multiple platforms. Key features include support for multiple protocols and databases, and user-defined events among others. Nimbits [82] can store and process a specific type of data previously time- or geo-stamped. A public platform as a service is available, but it can also be downloaded and deployed on Google App Engine, any J2EE server on Amazon EC2 or on a Raspberry Pi. Netquest [26] is a programming model to ease the development of ubiquitous applicactions on sensor networks. On paper [27], Netquest is used to work on a small network of iMote devices. OpenRemote [83] offers four different integration tools for home-based hobbyists, integrators, distributors, and manufacturers. It supports dozens of different existing protocols, allowing users to create nearly any kind of smart device they can imagine and control it using any device that supports Java. The SiteWhere [86] project provides a complete platform for managing IoT devices, gathering data and integrating that data with external systems. SiteWhere releases can be downloaded or used on Amazon's cloud. It also integrates with multiple big data tools, including MongoDB and Apache HBase. Finally, ThingSpeak [95] can process HTTP requests and store and process data. Key features of the open data platform include an open API, real-time data collection, geolocation data, data processing and visualizations, device status messages and plugins.

Deployment of IoT platforms on the Cloud is also covered in the literature. In [33], authors propose strategies for deciding the best approach at the time of making cloud-based deployments of IoT applications using nowadays regular cloud technologies. Another recent work [11] studies the implementation of IoT platforms on top of cloud-based pub/sub communication infrastructures. Finally, authors go one step beyond in [78] by leveraging completely Software Defined Environments for managing the Cloud infrastructures in which IoT applications are deployed.

3.7 Conclusions

In this chapter we have introduced a stream processing platform with dynamic pipeline processing and a programming model based on the stream subscription. On the one hand the topologies can be constructed on the fly while they are being executing, enabling environments with multiple tenants performing analytics of other tenants' streams. On the other hand, each computation node is loaded when it is required, avoiding unused nodes taking resources in big topologies. STORM provides auto-scaling capabilities that make it particularly suitable for cloud deployments. Furthermore, the compose operation is presented as a method to produce composite streams.

The ServIoTicy runtime allows for users to deploy custom service code inside Service Objects in the form of composite streams, and subscribe those streams to multiple sources of data (either outside the platform on real-world devices or in other streams defined in the ServIoTicy platform by other users). The user-code will be automatically injected in the STORM topology and executed when a unit of data is generated from a source to which the composite stream is subscribed.

The runtime is designed to be highly scalable, following a lock-free model that combines operations triggered by new data being generated inside or outside the platform, with queries performed over historic data logged for existing Service Objects. The design imposes some restrictions mainly related to the timestamps of the updates being processed, and some optimizations are applied to improve the scalability of the platform.

A basic evaluation of the runtime is included in this work, showing how acceptable response times of less that 100ms can be delivered by basic composite streams, and that for most realistic pipelines can be processed in the range of less than a second. The work presented in this chapter is, to our knowledge, the first IoT data processing platform with dynamic pipelining for the Cloud.

The next steps to follow after this contribution will be to extend the programming model to enable some new features. One of them is having sliding window aggregators defined by static size, time interval and random events. Being this the scenario of data streams in real-time, the programming model needs to enforce efficient incremental algorithms for the aggregators so the computation time with millions of updates is ideally lower than the interval between the arrival of each update.

Moreover, another interesting feature is dynamic data stream subscriptions. To subscribe to one or several streams it is needed to provide their unique ids. A more flexible way to do that would be subscribing dynamically to the streams that match some specific features. Every time a stream is added to the platform and it matches a dynamic subscription criteria, it will be bound automatically to its subscribers.

Chapter 4

Constant-Time Sliding Window Framework with Reduced Memory Footprint and Efficient Bulk Evictions

4.1 Introduction

Stream Processing, or processing data on-the-fly, is a critical demand in many environments requiring low latency and reduced data movement. Scenarios like telemetry data analysis in large data centers, or advanced analytics for the Internet of Things (IoT), often require fast processing and aggregation of vast amounts of data. Moreover, processing data close to the source becomes an important factor when data movement is expensive due to high volume of data or poor connectivity. Due to these reasons, over the past five years a number of Stream Processing platforms have emerged, including Apache Storm [96], Apache Flink [37], Apache Samza [16] and Twitter Heron [70] as the most noteworthy open-source solutions. Furthermore, commercial solutions from the most important players in the IT industry are also offered, such as Amazon Kinesis [9], Google MillWheel [3], IBM Streams [58] and Microsoft Azure Stream Analytics [64].

Data streams are unbounded sequences of ordered atomic updates on the same information feature. E.g., a stream associated to the temperature of a physical device *D* contains a sequence of updates of temperature information coming from device

D, each update substituting the previous one. Given that a stream emits updates indefinitely, such sequences of updates can not be traversed upstream as they do not have finite size and lack boundaries. Instead, selecting a limited window on the updates within a data stream is commonly considered the most affordable method for analyzing the data and information coming from a data source. It is for this kind of processing that projecting data from streams into sliding windows becomes a convenient mechanism towards data analysis and aggregation.

More formally, a Sliding Window is an abstraction representing a projection over a data sequence. Sliding windows are usually implemented as FIFO structures containing timestamped data updates, all of the same type. Updates enter the window when they are received from the data source, and are evicted according to a Window Slide Policy (WSP) that defines the criteria that older updates need to meet to leave in the window. Therefore, sliding windows define a contiguous sequence of strictly ordered data updates, whose length is defined by the WSP, and always containing the most recent updates generated to the moment.

Applications that process data streams usually define a set of aggregation operations that when computed produce a result associated to the stream. Due to the unbound nature of streams, sliding windows are a convenient approach to processing such aggregations, by defining the subset of updates to be considered for processing. Therefore, for their computational purpose, sliding windows are associated with at least one aggregation function, that will be computed for the contained values whenever the window content is updated.

There are two key aspects of a Sliding Window aggregation framework that define its applicability and efficiency across different scenarios:

- Firstly, the computational cost associated to the process of adding and evicting values into the structure through the WSP, and recomputing the values of the aggregations represented by the Window. Therefore, the algorithms used to operate the sliding window and the aggregations must be as efficient as possible, avoiding the computing time to grow with the window size. Naive implementations that recompute all the aggregations for every new update, thus having linear cost $O(n)$, are not able to keep up with large window sizes and high arrival rates. The Framework introduced in this chapter exhibits amortized constant $O(1)$ time-complexity between updates, and $O(\log n)$ for bulk eviction, positioning itself ahead of the existing state of the art.

- The second aspect is the memory footprint of the Window data structures. Existing time efficient implementations tend to pre-allocate all the needed memory, with space cost $O(N)$ from the pre-defined maximum window size N. While this approach is convenient in terms of computational complexity, it imposes serious limitations in terms of the applicability of the technique across domains. For instance, cloud-based deployments may require extra-large VMs to host them with their associated additional cost, and Fog-based deployments will struggle to accommodate these implementations in memory-constrained edge devices. Furthermore, resizing the maximum window capacity results in a $O(n)$ time complexity operation. The Framework introduced in this chapter leverages an efficient decoupling of the computation and data store through the use of a *Key-Value Store*, which results in a local space allocation of only $O(\log N)$ from the pre-defined maximum window size N, which also improves on the existing state of the art.

We introduce the Monoid Tree Aggregator (MTA) General Window Aggregation Framework, which advances the state of the art in the following aspects:

- Seamlessly combines amortized constant $O(1)$ time-complexity between updates and logarithmic $O(\log n)$ cost in the worst-case scenario
- Its data structures only need to statically preallocate space for $O(\log n)$ elements, being n its maximum capacity.
- The window aggregation mechanism and the Window Slide Policy (WSP) are user-programmable. Aggregations are described as associative operations, based on monoids, and they do not need to be invertible. The WSP, instead, defines the criteria that data to be evicted must meet.
- The WSP enforcement mechanism exhibits amortized $O(1)$ computational cost to perform single evictions on the window and $O(\log n)$ for bulk eviction operations. The mechanism is similar to performing searches on *Binary Search*

Trees [66]. This aspect is of paramount importance to implement flexible WSPs, like for instance time ranges (e.g. *data accumulated in the last 5 minutes*) for a source that produces data at variable rates. This situation leads to windows containing a changing number of elements over time and mass evictions at certain moments in time.

The general purpose and efficient Sliding Window aggregation framework leveraged here could be used as an operator for Stream Processing platforms such as Apache Storm or Apache Flink. They would additionally benefit from the fault-tolerance provided by the distributed KVS based data structure.

The rest of the chapter is structured as follows: Section 4.2 introduces the main concepts related to Sliding Window frameworks; Section 4.3 discusses the state of the art in the field of efficient Sliding Windows for Stream Processing; Section 4.4 discusses the main characteristics of the MTA Window Framework; Section 4.5 discusses the implementation details of the framework proposed in this work and provides the results of an experimental evaluation of the MTA Window Framework; Finally Section 4.6 discusses the conclusions of the work.

4.2 Background: Real-Time Sliding Windows

4.2.1 Sliding Windows: Concept

Sliding Windows are an abstraction representing projections on data sequences, organized as FIFO structures containing elements of the same type (the data updates), and a timestamp associated to each element. Data updates enter the sliding window when they are received from the data source, and are evicted according to a Window Slide Policy defining the conditions to be satisfied by data updates for leaving the projected window. Sliding windows define a contiguous sequence of strictly ordered values, with a length depending on the slide policy, and always containing the most recently generated updates.

Therefore, the three main building blocks used by sliding windows as mechanisms to aggregate streams of data and their features are:

FIFO data structure: An update in the window is not removed until all the older updates are removed too. Contents are a complete and ordered portion of the stream being aggregated. Updates are inserted at the end and removed from the beginning of the structure.

Aggregation algorithm: Aggregations are applied to the data covered by the window in a specific moment. For instance, the aggregation could be a total sum and it could be executed every time a new update is inserted to the window. Some sliding window aggregation frameworks only accept invertible operations as aggregators, with their inverse functions. This way, the eviction of data is done in constant-time easily.

Slide policy: Updates leave the FIFO structure according to a window *slide policy* (WSP). WSP defines the conditions to be satisfied by the older updates in order to be evicted from the window. The result after applying the policy must be a subsequence including the most recent updates in the window. Traditionally WSP in sliding window aggregation frameworks are delimited by maximum window size and time-based windows. Regarding the number of updates to be removed by the policy, it is usually determined by three choices: a single update, a fixed amount of updates and all the window updates. However, a WSP can be expressed in terms of the window aggregation itself and become more rich, customizable and efficient.

4.2.2 Sliding Windows: Running Example

For the sake of clarity, we include here a running example of a Sliding Window used to compute the *maximum* of the values of the updates that fall within the windows according to a WSP. The WSP is complex enough to need to be expressed in terms of the window aggregation. This will provide an understanding on the need of general purpose and user-programmable WSP based on the aggregation. The WSP dictates that the window will contain the updates that add up to a value which is less or equal than 10, the rest will be removed. From the resulting window we want to extract the maximum value. For this purpose two aggregations will be used over the window: the first one will be *max*, and the other one will *sum*. The former will be used to compute the result of the operation that needs to be calculated for the window. The latter will be used to estimate the updates that have to be removed from the window after an update insertion, according to the WSP. This is also an example of multi-dimensionality of the window in terms of aggregation operations; the window could be used also considering multiple data dimensions across the window elements.

More formally, let S be a stream of ordered data and $(d_i)_{i=1}^{n}$ be the current data updates in S where i is its timestamp and n is the oldest timestamp in S. Then the WSP on the window W is:

$$\forall d_i \in S : d_i \in W \iff \sum_{j=i}^{n} d_j \leq 10$$

In this context, consider a window with the values
$[2, 2, 3, 3]$, ordered in ascending order of their timestamp - that is the leftmost 2 is the oldest update in the window while the rightmost 3 is the most recent update (corresponding to d_n following the notation used above). Therefore, when a new update with value 4 is inserted to the window, the policy removes the oldest 2 updates and the result window becomes $[3, 3, 4]$. This slide policy is enforced using the sum aggregation that is calculated over the values in the stream: $3 + 3 + 4 \leq 10$. The max aggregation value would have been 3 before the last insertion, and 4 immediately after.

A naive design of these features can be achieved via the use of a simple queue that aggregates all its contents every time it is queried. The WSP enforcement pops updates until the window contents comply the policy. It clearly entails $O(n)$ to aggregate the window, n being the number of updates that are inside; hence it would quickly struggle to scale with high frequency streams and densely populated windows.

4.2.3 Sliding Windows: Monoids for Aggregators

A monoid is an algebraic structure with an associative binary operation and a neutral element. They are extensively used in the literature for the implementation of data aggregations, and it is the common choice for state of the art Sliding Window implementations, as it will be discussed in Section 4.3.

More formally, where S is a set and $\cdot$ is a binary operation, it composes a monoid if it obeys the following principles:

Associativity: $\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$

For all a, b and c in S, the expression $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ is true.

Neutral element: $\exists e \in S : \forall a \in S : e \cdot a = a \cdot e = a$

Exists a value e in S that for all a the expression $e \cdot a = a \cdot e = a$ is true.

Closure: $\forall a, b \in S : a \cdot b \in S$

For all a and b in S, the result of $a \cdot b$ is in S too.

Aggregators are then programmed in a *map* $\rightarrow$ *reduce* $\rightarrow$ *map* structure, where the first *map* transforms the input value to a member of S, the *reduce* stage is a monoid, and the last *map* converts the monoid result to the desired value out of S. For example, an average aggregator could have a monoid with S defined as integer pairs (s, c) where s is the sum of the values and c is the number of values. The monoid operation would be $(s_1, c_1) \cdot (s_2, c_2) = (s_1 + s_2, c_1 + c_2)$. The first map transforms an input value v to the pair $(v, 1)$ in S, where v is the initial sum and 1 the initial count. $(v, 1)$ is then operated by the monoid with another mapped value or a previous monoid result. The last map transforms the monoid reduced result to $\frac{s}{c}$ which is the final average.

4.3 Related Work

Since the Sliding Window Framework presented in this chapter is compared to the state of the art in order to demonstrate its novelty, this section will make a survey and description of the current Sliding Window Framework approaches in the literature. This will provide context prior to the in-depth description and evaluation of our contribution.

The state of the art in the literature proposes to use a FIFO structure and incremental operations to reduce the complexity of the aggregation algorithms to $O(\log n)$ and amortized $O(1)$ for variable-sized windows.

Tangwongsan et al. propose in their prior work two sliding window aggregation frameworks called *Reactive Aggregator* (RA) [92] and *Sliding-Window Aggregation* (SWAG) [90]. Having important differences between them, both approaches follow Boykin et al. [34] method of using associative operations as programmatic aggregators interface. Both RA and SWAG benefit from using associative aggregation, by enabling the computation of partial results and using the neutral element property to evict elements from their FIFO structures.

The main claim for RA is that it is $O(\log n)$ in all its operations with constant-sized sliding windows. RA's sliding window FIFO structure is a flat fixed-sized binary and complete tree called FlatFAT. Similarly to Log MTA, all the leaves are the raw updates to be aggregated, the root node is the result and the intermediate nodes are partial computations. Every update insertion and deletion propagates the aggregation changes from the leaf to the root. Other work in the literature [76, 100, 22, 29] use tree-like structures in order to keep partial computations in the same way, making use of binary associative operators. They all have a worst-case $O(\log n)$ for all its atomic operations and a complexity $O(n)$ for windows with bulk evictions.

On the other hand, SWAG is a sliding window aggregation framework that runs in worst-case $O(1)$ time for each one of its atomic operations. SWAG's *insert*, *remove* and *query* operations perform in constant time with constant-sized windows. The simplified version of its main algorithm is based on a data structure with two stacks instead of a tree-like structure. One stack receives the new updates, each paired with a partial result generated by aggregating the update with the previous top partial result in the stack. The second stack is generated by reversing the order of the updates from the insertion stack and recomputing the partial results. The *reverse* operation is $O(n)$ but ends up amortized to $O(1)$ during the execution of the window. However, the *reverse* operation can be incrementally performed on *insert* and *remove* operations, turning into a worst-case $O(1)$ process if updates are removed one by one.

The time complexity is better in SWAG than in RA and similar solutions for non-invertible window aggregators, while MTA Window Framework extends major improvements from it. In first place, the window operations are logarithmic for RA-like algorithms and constant for SWAG. However, this is not considering bulk eviction as an atomic operation, and therefore it works in constant time for constant-sized windows. Constant-sized windows remove one update for each one received, keeping always the same number of aggregated updates. As only one element needs to be removed, *remove* operation complies with the $O(\log n)$ time-complexity in RA and $O(1)$ in SWAG. Yet it is a common situation to work with time-based window over a stream with an irregular input frequency. This poses a problem: Variable-sized windows like time-based windows require bulk evictions, and this operation is worst-case $O(n)$ lineal for the state of the art.

Aside from the efficiency issue that the previous point raises, such a situation makes it unfeasible to keep partial results and updates in remote data stores, as n elements might need to be retrieved for a single bulk *remove* operation. Consequently decoupling the majority of data from the local computation is not considered.

Moreover, a general mechanism for framework users to define custom sliding policies is not defined in any of the state of the art solutions.

Table 4.1 summarizes the comparison of RA and SWAG, with the mechanisms introduced in this work: the *Log MTA* and the more advanced *Amortized MTA*. The parameters compared include the amortized and bulk-eviction worst-case case computational complexity of the frameworks, the size of the data structure, the minimum size to be stored locally for processing the stream in the worst-case scenario, the ability to enforce user-defined Window Slide Policies, and the existence of an efficient design that supports decoupling of data and computation (e.g through the use of external key-value stores to keep part of the data).

	RA	SWAG	*Log MTA*	*Amortized MTA*
Amortized time	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Bulk eviction time	$O(n)$	$O(n)$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$
Size	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Min. local size	$O(n)$	$O(n)$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n)}$
Custom WSP	×	×	✓	✓
Data Decoupling	×	×	✓	✓

TABLE 4.1: Sliding window frameworks comparison

Alternative approaches to improve efficiency in sliding windows found in the literature [31, 30, 72, 43, 69] consist on keeping in memory partial aggregations from window updates instead of keeping the original updates. The result is an speed up of the aggregation and removal and also the memory needed is reduced. However, either there is a percentage of error in the number of updates evicted each time, or the algorithm knows the exact number of updates that will be removed in each iteration in order to avoid the error. The most relevant case is the Exponential Histogram from Datar et al. [43], a data structure that maintains an approximation of the number of 1's in a sliding window with logarithmic memory and time complexity. The counting is fragmented over a list, where the number of window updates counted in each list element grows exponentially from tail to head. A general purpose approximate computation similar to Exponential Histogram applied to MTA is a potential subject for future investigation, which would also improve performance and memory consumption.

Bifet & Gavaldà contributed *ADWIN* [31] and *K-ADWIN* [30] mechanisms, which implement a variation of exponential histograms. *ADWIN* is a programmable sliding window framework that automatically adapts its size by detecting changes on the data. When two subwindows have very different average values, the oldest one is evicted. The data kept in the window is considered the currently relevant data from the stream, and *guest algorithms* can perform aggregations from it. *K-ADWIN* combines *ADWIN* with Kalman filter [63], providing better results than both methods separately. *ADWIN* base algorithm can be seen as an adapted MTA WSP that compares the average value between subwindows, and the monoid aggregator as the *guest algorithm*.

Additionally, resource sharing is another methodology discussed in the literature [22, 68] to enhance performance among incremental aggregations. Although our solution is not focused on a resource sharing approach, a basic mechanism to share some resources between aggregations is also present. We introduce window aggregation multi-dimensionality, which consists in performing several aggregations on different data in the same stream, sharing resources such as the window data structure and the WSP. Experiment 4 from Section 4.5 shows the benefits from this approach. Tangwongsan et al. [92] already compared RA with the resource sharing focused solution from Arasu & Widom [22] positioning RA as a more advanced solution, and later SWAG [90] [91] as more advanced than RA.

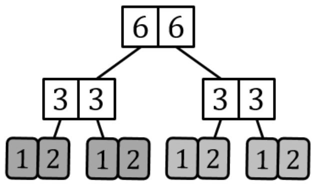
4.4 Framework Design

This section describes the Monoid Tree Aggregator (MTA) Window Framework, which is this chapter's main contribution. The MTA Window Framework is an sliding window framework that aggregates values in amortized constant time between insertions, on par with the most advanced existing solutions in the literature. Additionally, it exhibits logarithmic time complexity in the worst case scenario, which includes bulk element eviction. Efficient bulk eviction is an improvement with respect to the state of the art, and it is of paramount importance for resource-constrained environments and real-time situations, like the ones considered for Edge Computing in emerging IoT scenarios. This time complexity is achieved regardless of whether the aggregation function is invertible or not. Furthermore, it provides programmable aggregation mechanism and Window Slide Policies. All this combined enables the framework to decouple most of the data aggregated from the local memory in which it is being computed, delegating this task to another system such as a distributed data store, a local hard drive or an NVMe. For these reasons, the MTA Window Framework positions itself as a significant advance with respect to the existing state of the art solutions.

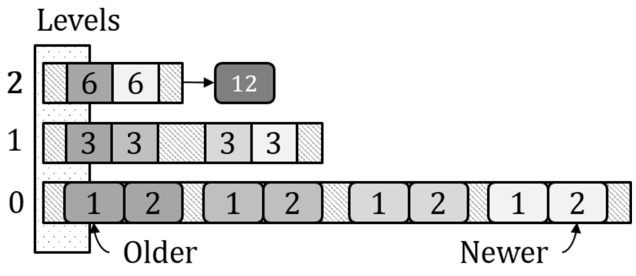
For the sake of clarity, we present the core algorithms of the MTA Window Framework in two steps: first, we describe a set of algorithms (*Log MTA* in Subsection 4.4.1), which create a logarithmic-time window aggregation mechanism, less efficient than the concluding MTA solution, but much simpler to explain; later, in Subsection 4.4.2, we extend the *Log MTA* mechanism to reduce the computation complexity to an amortized constant cost $O(1)$, in the *Amortized MTA* mechanism.

4.4.1 Log MTA

The Log MTA mechanism is a logarithmic-time aggregation window, used as base for the constant-time solution. It sets the foundations on the main MTA Window Framework features which are discussed in detail in this section, being: general user-programmable aggregation, efficient computation, general user-programmable WSP mechanism, data decoupling and efficient bulk element eviction.



(A) Log MTA Tree



(B) Log MTA level division

FIGURE 4.1: Log MTA Structure and Element Location Examples

Structure

The FIFO data structure in Log MTA is a binary Tree designed as a list of queues. Each queue is a Tree level, sorted in the main list from leaves (bottom level) to root (top level). The levels contain the Tree nodes grouped by pairs. The elements in the same pair are Tree siblings. A neutral element ($\oslash$) in a pair means an empty branch. The lowest level contains all the window updates in order. The levels above contain the monoid aggregation results from lower level pairs. E.g., Figure 4.1a shows an abstraction of the full binary Tree of a Log MTA performing a *sum* aggregation, with $[1, 2, 1, 2, 1, 2]$ as data updates. Figure 4.1b shows its representation as a list of queues. New Tree nodes are pushed to the level queues and popped when removed, hence the FIFO behaviour. All the leaves of the Tree will be found in the first level, as stated by Invariants 1 and Invariant 2.

Invariant 1. *Let T be the Log MTA binary Tree, L_i the $i-$essime level from leaves L_1 to root L_h, and h being the height of T. Let $(v_{ij})_{j=1}^{n} \in L_i$ be the nodes of L_i, n be the number of nodes in L_i, $\oslash$ being the monoid neutral element and $*$ any non-neutral element. Then:*

$$v_{i,1} = \langle \oslash, * \rangle \lor v_{i,1} = \langle *, * \rangle$$

$$v_{i,n} = \langle *, \oslash \rangle \lor v_{i,n} = \langle *, * \rangle$$

$$\forall_{j=2}^{n-1} v_{i,j} = \langle *, * \rangle$$

When $n = 1$ then only one of the three statements needs to be satisfied.

Invariant 2. *Having h as the height of T, $n = |L_i|$, and v_{ij} containing a pair of elements:*

$$\forall_{i=1}^{h} \forall_{j=1}^{n} v_{ij} \neq \langle \oslash, \oslash \rangle$$

Theorem 1. *Let $(e_{ijk})_{k=0}^{1}$ be each element in the pair node v_{ij}. Let $children(e_{ijk})$ be a function that returns the children pair of e_{ijk}, $\forall_{i=2}^{h} \forall_{j=1}^{n} v_{ij}$:*

$$v_{i,1} = \langle \oslash, * \rangle \rightarrow children(e_{ijk}) = v_{(i-1),(2j+k-1)}$$
$$v_{i,1} \neq \langle \oslash, * \rangle \rightarrow children(e_{ijk}) = v_{(i-1),(2j+k)}$$

The first pair of a level can be $\langle \oslash, * \rangle$ when the first element has been removed, so the node does not have left child. Also, the last pair of a level can be $\langle *, \oslash \rangle$ when its right child has not been created yet. $\oslash$ can not be found in any other position in the Tree.

Theorem 1 shows how Tree branches can be traversed, derived from Invariants 1 and 2. The greater part of the Tree traversing is performed through the first or last element of every level only, the Tree side branches. However, for bulk eviction we will need random branch traversing from root to leaf in order to find the branches to be removed.

Data Insertion & Aggregation

New updates are inserted to the data structure and aggregated in logarithmic time. Updates in the window are aggregated by grouping them in ordered pairs and applying a user defined monoid on each pair. The results are paired and aggregated again, in a process that is repeated iteratively until a single result is produced. This process is performed incrementally for each insertion using the binary Tree structure, as it can be seen in Algorithm 1.

When an update is inserted to the window, it is pushed at the end of the first level as a new leaf of the Tree. If the last pair in the level is $\langle *, \oslash \rangle$, the new value u is placed as $\langle *, u \rangle$, otherwise a new pair is created and added as $\langle u, \oslash \rangle$. When a new pair is added, it will not have a parent yet.

After adding the new leaf, it is aggregated with its sibling executing the user provided monoid, which Log MTA is oblivious to. The result element will be the

Algorithm 1 Log MTA insertion & aggregation. Inserts update u to Tree T

```
$L \leftarrow levels(T), agg \leftarrow u$
for $l = 1, ..., |L|$ do
    $P \leftarrow L_{l,|L_l|}$
    if $agg \neq \oslash$ then
        if $P_1 = \oslash$ then
            $P_1 \leftarrow agg$
            $agg \leftarrow \oslash$
        else
            $L_l(enqueue(\langle agg, \oslash \rangle))$
        end if
    else
        $Q \leftarrow L_{l-1,|L_{l-1}|}$
        if $P_1 \neq \oslash$ then $P_1 \leftarrow monoid(Q_0, Q_1)$
        else $P_0 \leftarrow monoid(Q_0, Q_1)$ end if
    end if
end for
if $agg \neq \oslash$ then
    $L(\{\langle agg, \oslash \rangle\})$
end if
```

parent of both siblings and need to be inserted in the level above. If the pair already had a parent in the level above, the parent's value needs to be updated with the new one. Otherwise, a new pair needs to be added with the aggregation result. The parent pair now needs to be aggregated, propagating the process towards the root level. If the root level now has a pair without $\oslash$, then a new level will be added that will become the new root.

This operation has time complexity $O(\log n)$, as it executes a fixed set of constant-time operations for each level on the Tree, with logarithmic height with respect to the number of updates.

Window Slide Policy Definition

Like the aggregation operation, the WSP is a user-programmable condition that needs to follow some rules. It has access to the total aggregation of the window after the last insertion and to the aggregation of a random subsequence from the head of the window. This aggregated subsequence is the one being checked for removal. If the condition defined by the WSP using these values is met, then *at least* this subsequence needs to be removed from the window aggregation.

For instance, consider a window with updates that include an ordered timestamp in milliseconds and that it aggregates them with a *max* operation. Therefore, the aggregated result timestamp from a sequence of updates is the latest timestamp. If the WSP example in Listing 4.1 is applied to this window, its result aggregation will always use updates in the last hour. The condition compares the latest timestamp from the subsequence with the lower boundary of the WSP time frame (one hour before the last update). If the subsequence's latest timestamp is not inside this boundary, then the condition is met and it needs to be subtracted from the window aggregation.

This mechanism enables the user to define from the most basic WSP to complex and dynamic scenarios using sophisticated aggregations.

LISTING 4.1: Window Slide Policy Example

```
function wsp(total, old){
  return
    (total.timestamp - old.timestamp) >= 3_600_000;
}
```

Efficient Bulk Eviction

After inserting a new update, the WSP needs to be enforced to find the longest subsequence of updates that need to be removed from the head of the FIFO structure, so a single result is produced. This process takes advantage from the binary Tree based data structure and it is performed in $O(\log n)$. Furthermore, the time complexity is the same for both removing a single update or performing a bulk update eviction from the window.

The importance of performing efficient bulk update evictions from a window resides in the concept of variable-sized windows in contrast with constant-size windows. Constant-size windows remove one update for each one received, keeping always the same number of aggregated updates. However, it is a common situation to work with time-based window over a stream with an irregular input frequency. This poses a problem: after inserting $k \geq 1$ data updates to the window, k updates might need to be evicted at once, triggered by the time based slide policy. In the state of the art, all the updates need to be traversed and possibly removed, multiplying by n the time complexity of removing a single update. Variable-sized windows like time-based windows make the most of performance improvements on bulk evictions, especially on situations in which real-time aggregation is required.

The WSP enforcement performs a $O(\log n)$ root to leaf search in the binary Tree for the oldest valid update in the window, while pruning invalid branches guided by the user-defined WSP. Algorithm 2 is a detailed specification of this operation.

Algorithm 2 Log MTA WSP enforcement on Tree T with efficient bulk eviction

```
 1: $L \leftarrow levels(T), sub \leftarrow \oslash, rm \leftarrow 0$
 2: for $l = |L|, ..., 1$ do
 3:     $P \leftarrow L_{l,1}$
 4:     if $rm > 0$ then
 5:         $L_l(remove\_pairs(rm))$
 6:         $rm \leftarrow 2 \cdot rm$
 7:         if $P_0 \neq \oslash$ then $rm \leftarrow rm - 1$ end if
 8:         $P \leftarrow L_{l,1}$
 9:     end if
10:     if $P_0 \neq \oslash$ then
11:         $subseq \leftarrow monoid(rm\_subseq, P_0)$
12:         if $wsp(subseq, result(T))$ then
13:             if $l = |L| \wedge l \neq 1$ then $L(remove(l))$
14:             else if $P_1 = \oslash$ then $L_l(remove\_pairs(1))$
15:             else $P_0 \leftarrow \oslash$ end if
16:             $rm\_subseq \leftarrow subseq$
17:             $rm \leftarrow rm + 1$
18:         end if
19:     end if
20: end for
21: $P \leftarrow L_{1,1}$
22: $agg \leftarrow monoid(P_0, P_1)$
23: for $l = 2, ..., |L|$ do
24:     $P \leftarrow L_{l,1}$
25:     if $P_0 \neq \oslash$ then $P_0 \leftarrow agg$ else $P_1 \leftarrow agg$ end if
26:     $agg \leftarrow monoid(P_0, P_1)$
27: end for
```

The Tree levels are traversed from root to leaves executing the WSP with the first level value as the aggregated value of its leaves subsequence. If removed, the node's branches will be evicted from the next levels before running the WSP on the new first element. As the *remove_pairs* function in Algorithm 2 only updates the pointer to the first pair of the level queue, it has constant time. The elements can be removed in the background by a garbage collector, minimally affecting the process of getting a result aggregation. When this process is finished, the nodes in the leftmost branch might not be consistent and have aggregated values that have been removed. Therefore, the leftmost branch is recomputed bottom-up, propagating the value changes to the root pair and resulting in a valid aggregation result.

A running example of this process can be found in Figure 4.2, where timestamps are inserted to the window and the WSP only allows a window of 4 time units. When timestamp 8 is inserted it triggers the WSP enforcement to evict all the other updates in the window in three steps. It checks the first valid element of each level from root to leaves with the WSP. All of them are found out from the window, leaving only the newest update.

Time complexity is $O(\log n)$, as this operation performs a fixed number of constant-time operations for each level of the Tree, by visiting them twice.

Once the WSP has been enforced, the aggregation result can be queried to the

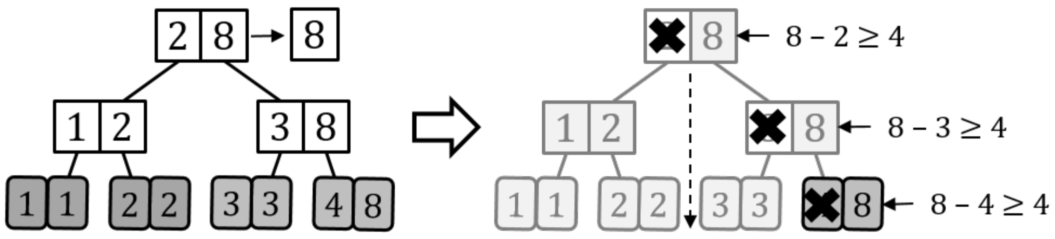


FIGURE 4.2: Log MTA Bulk Eviction. Monoid: $max(x,y)$; WPS: $total - old \geq 4$.

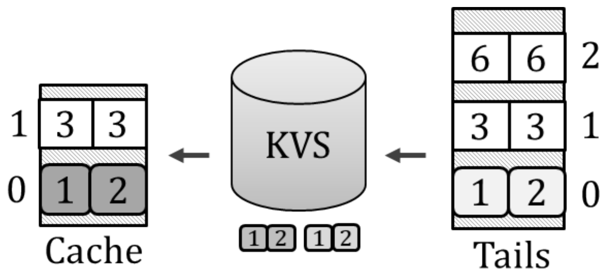


FIGURE 4.3: Log MTA KVS data structure

window. This operation returns the aggregation of the window contents in constant time, by returning the monoid result in the root pair. Furthermore, if we require a reactive behaviour from the window, then the following pipeline needs to be executed when a new update arrives: *insert update* $\rightarrow$ *enforce WSP* $\rightarrow$ *query result*. Every time a new update is introduced, it produces the result in logarithmic time.

Reducing Local Memory Footprint

From inserting a new update to generating a new result, Log MTA needs to traverse at most $O(\log n)$ elements: for each level queue, the tail element and probably an element near the head. Therefore, the rest of the data in the window does not need to be waiting in local memory and the resources could be used to run other aggregations. In the worst case, a bulk eviction will need to traverse a Tree branch that is not currently in local memory, which will require an immediate memory retrieval of only $O(\log n)$ elements.

In the proposed data structure, each level queue has three sections between two different memory layers, as it can be seen in Figure 4.3. The pairs in the tail of the level queues are in the *Tails* list in local memory, the central pairs wait in a shared *Key-Value Store* (KVS), and the pairs in the queues' head can be found in a *Cache* in local memory again.

The rightmost branch of the Tree is found in the Tails list. It receives updates as they are being inserted to the structure, and pushes the replaced pairs to the KVS. The pairs pushed to be sent to the KVS are first kept into a buffer in order to reduce the number of interactions with the data store. When the maximum capacity of the buffer is reached, all its contents are moved to the KVS. The key in each KVS document maps its contents to its Tree level and its position it has inside the queue, so a $O(1)$ single pair retrieval can be achieved. The store can be anything from a local HDD file to a remote and dedicated cluster. Finally, the Cache contains at least the head pair from each level queue, with the exception of the root level. The Cache is refreshed from the KVS and the buffer when its size is under an specific threshold or in a Cache miss situation. Its capacity can be adapted to reduce the interactions with the KVS.

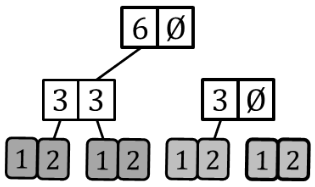
Having the data stored by an external entity, apart from the scalability enhancement it provides, makes it easier to recover aggregation data from a failure.

4.4.2 Amortized MTA

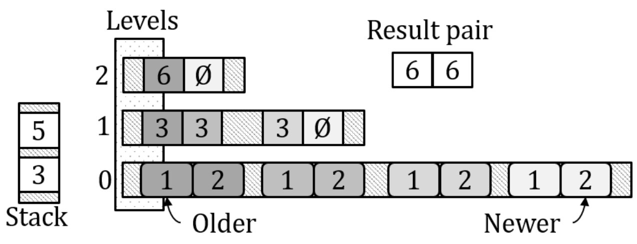
In this section we present the Amortized MTA (AMTA) sliding window mechanism. It is an approach aimed to reduce Log MTA's time complexity without having an impact deteriorating its other benefits in comparison with the state of the art: space complexity, its user-programmable WSP mechanism, efficient bulk evictions and the reduced local memory footprint.

Structure

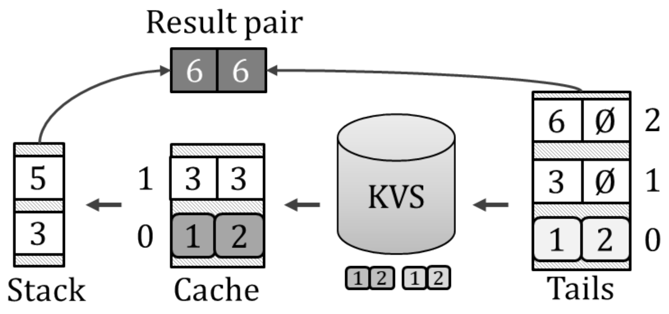
Amortized MTA is an sliding window mechanism that inserts, aggregates and removes elements in amortized constant time, with logarithmic time in the worst case. It satisfies Log MTA Invariants 1 and 2, and shares the data structure level division and the memory layers, although the data structure operates differently. In the new data structure, the Tree is replaced by a Forest of binary trees where the rightmost pair of each level is the root of its own tree, as defined in Invariant 3. Figure 4.4a is an example of an Amortized MTA window performing a *sum* aggregation with the updates $[1, 2, 1, 2, 1, 2, 1, 2]$. The lower level contains the values in the window to



(A) AMTA Forest



(B) AMTA level division



(C) AMTA KVS data structure

FIGURE 4.4: Amortized MTA Structure and Element Location Examples

be aggregated, while the levels above contain partial aggregations of these values. Considering Invariant 3 now, Theorem 1 is also valid as a tree traversing guide.

The data structure also introduces the Stack and the Result Pair, as it can be seen in Figure 4.4b example. As an addition to Tails and Cache, they are the parts of the structure required to be local memory at all times. E.g., Figure 4.4c shows the memory distribution of the data structure.

Result Pair ($R = \langle R_0, R_1 \rangle$) maintains the aggregated result from the leftmost tree in R_0 and the aggregated result from the rest of the Forest in R_1. The Stack contains the aggregated results of the leftmost tree without the first element from each level. The top value from the Stack is always R_0 minus the update in the head of the window. In Figure 4.4b we can see that the Stack top element is 5, which is R_0 minus the head element in the first level: $6 - 1 = 5$. Likewise, the next element in the Stack is 3, which corresponds to R_0 minus the head element in the second level: $6 - 3 = 3$.

Essentially, AMTA insertions aggregate the new values in R_1, while single update evictions pop values from the Stack onto R_1. For instance, inserting 1 to the data structure in Figure 4.4 would result on $R_1 = 6 + 1 = 7$, while evicting the first element would pop 5 from the Stack and put it on R_0. Aggregating R always produces the final result value for the window. The Forest is used to keep the Stack updated, to compute R_1 from scratch when necessary and to perform bulk evictions. Therefore, new updates also need to be inserted and removed from the Forest structure.

The main goal of AMTA is to improve its time complexity without giving up its other benefits in comparison with the state of the art: space complexity, its user-programmable WSP mechanism, efficient bulk evictions and the reduced local memory footprint. More details on how this is achieved using this structure can be found in the following sections.

Invariant 3. *Having h as the height of T and L_1 the leaf level for all the binary Forest, $n = |L_i|$, and v_{ij} containing a pair of elements, $\forall_{i=1}^{h} v_{i,n}$ is a tree root.*

Amortized insertion

Log MTA update insertion is $O(\log n)$ because for every new update, the Tree nodes need to be updated from the leaf to the root. This can be avoided by only adding a node to the Tree when its value is definitive. In other words, a pair will only have a parent if both members in the pair have been inserted. In AMTA, a pair will only have a parent if it is not in the tail of its level queue. Figure 4.4a example shows that the last $\langle 1, 2 \rangle$ pair in level 0 is not aggregated in level 1 and there is a $\oslash$ at its tail instead. The consequence in the shape of the data structure is Invariant 3, the tail pair on each level is the root of its own binary tree. This process is amortized $O(1)$ and $O(\log n)$ in the worst case.

However, the pair in the upper level does not contain the full aggregation result, only a part of it. Therefore, every inserted update is aggregated in R_1 ($O(1)$), which contains the aggregation of all the trees except the leftmost one. R_0 contains the aggregated result of the leftmost tree, so the aggregation of R is the full window aggregation result.

As the window grows, R_1 aggregated trees merge with the leftmost tree. In this situation, part of the aggregation moves from R_1 to R_0. Therefore, R and the Stack

Algorithm 3 AMTA update insertion. Inserts u to data structure C

```
1: $L \leftarrow levels(C), R \leftarrow result\_pair(C)$
2: $agg \leftarrow u, l \leftarrow 1, h \leftarrow max(|L|, 1)$
3: $R_1 \leftarrow monoid(R_1, u)$
4: while $l \leq h \wedge agg \neq \oslash$ do
5:     $P \leftarrow L_{l,|L_l|}$
6:     $next\_agg \leftarrow \oslash$
7:     if $P_1 \neq \oslash$ then
8:         $next\_agg \leftarrow monoid(P_0, P_1)$
9:         $L(\langle agg, \oslash \rangle)$
10:    else
11:        $P_1 \leftarrow agg$
12:    end if
13:    $agg \leftarrow next\_agg$
14:    $l \leftarrow l + 1$
15: end while
16: if $agg \neq \oslash$ then
17:    $L(\{\langle agg, \oslash \rangle\})$
18: else if $l > h$ then
19:    $C(compute\_left\_result())$
20:    $C(compute\_right\_result())$
21: end if
```

need to be recomputed from scratch, which has an amortized $O(1)$ time complexity with $O(\log n)$ in the worst case.

Algorithm 3 describes the operation more formally. The new update u is firstly aggregated to R_1, overwriting its value to keep the Result Pair up to date. Then, u is inserted in the Forest's first level queue and the aggregation is propagated up to its tree root. Finally, when an element is inserted to the already existing highest level, R must be recomputed from scratch using *compute_left_result* for R_0 and *compute_right_result* for R_1, both $O(\log n)$.

Algorithm 4 Compute AMTA R_0 and Stack S in the data structure C

```
1: $S \leftarrow stack(C), L \leftarrow levels(C)$
2: $S(clear()), R \leftarrow result\_pair(C)$
3: for $l = |L|, ..., 1$ do
4:     $P \leftarrow L_{l,1}$
5:     if $P_0 \neq \oslash \wedge P_1 \neq \oslash$ then
6:         $S(push(monoid(P_1, S(peek()))))$
7:     end if
8: end for
9: if $P_0 \neq \oslash$ then $R_0 \leftarrow monoid(P_0, S(peek()))$
10: else $R_0 \leftarrow monoid(P_1, S(peek()))$ end if
```

Algorithm 5 Compute AMTA R_1 in the data structure C

```
1: $L \leftarrow levels(C), R \leftarrow result\_pair(C)$
2: for $l = 1, ..., |L|$ do
3:     $P \leftarrow L_{l,|L_l|}$
4:     $R_1 \leftarrow monoid(monoid(P_0, P_1), R_1)$
5: end for
```

compute_left_result places into R_0 the aggregation of the leftmost tree while repopulating the Stack, as described in Algorithm 4. The head pairs from each level queue are traversed, from root to leaf. When a pair P is $\langle *, * \rangle$, its element P_1 is aggregated with the top of the Stack (or with $\oslash$ if the stack is empty), and then stacked. Once all levels have been visited, the older update in the window is aggregated with the top of the Stack, and placed in R_0. The contents in the Stack will be used to perform eviction of single updates in amortized constant time.

compute_right_result operation aggregates into R_1 all the rightmost pairs (except for leftmost tree) in the Forest, as described in Algorithm 5.

The continuous execution of an update insertion in the Forest makes each element in the data structure to be visited once for the bottom-up propagation. As the space used for the data structure is $O(n)$, the cost of inserting n updates becomes $O(n)$. Then, functions *compute_left_result* and *compute_right_result* affect only $O(\log n)$ in a whole round of n elements insertions, complexity remaining $O(n)$ for inserting n updates. So, the amortized cost for aggregating 1 update to the window becomes $O(1)$.

Single update evictions

In Log MTA, performing a bulk eviction is a $O(\log n)$ operation, and it is the only option to remove any number of updates from the window aggregation. Removing a single update from its data structure and propagating the changes on the head of each level would have the same logarithmic cost. To amortize this cost, the solution we followed for AMTA is to find a way to perform amortized constant single update evictions and to save bulk evictions for when the number of elements to be removed is equal or greater than a factor of $\log n$.

The Stack from AMTA's data structure is the key element to achieve an amortized constant time single update eviction. It contains the future R_0 values after removing the head element from each level, being the oldest update removal always in the Stack's top. The rest of the elements will be used at some point, both for single update evictions and to maintain the Stack in amortized constant time.

The single update eviction operation is formally described in Algorithm 6. The Results Pair R is updated by popping an element from S into R_0. At this point, R aggregation already provides the correct aggregation result, but the Forest and the Stack need some maintenance before removing the next update.

The first update is removed from the head of the leaves level in the Forest, and the parent-child relations in the branch are updated. If a pair is removed from the Forest, then its parent element must be replaced by $\oslash$. However, the tree aggregations will not be updated, leaving inconsistent values in the data structure. The main implication of only updating the branch parent-child relations instead of also updating all the values is that, while it still keeps the tree consistent with Theorem 1, the amortized cost is constant and not logarithmic. During this process, all the new head pairs from each traversed level are pushed in the *new_heads* stack.

The current Stack top element might not be the next R_0. The Stack needs to be updated with new elements, using *update_stack*, which can be found in Algorithm 7. Similarly to *compute_left_result*, it updates the stack using values from *new_heads*. For every pair P popped from *new_heads*, its element P_1 is aggregated with the top of the Stack (or with $\oslash$ if the stack is empty), and then stacked.

If the number of levels has decreased after this process, the first tree has been completely removed and the second one took its place. Therefore R_1 needs to be recomputed.

Figure 4.5 shows an example of this situation. It is a window performing a *sum* aggregation on the sequence $[1, 3, 2, 1, 2, 1, 1, 0, 3, 1]$ with result 15. When the first update is removed, the top of the Stack (6) is moved to R_0 and the update is replaced by $\oslash$ in the Forest. The result is now $6 + 8 = 14$, which corresponds to $15 - 1 = 14$. No further actions are required after this update removal. The same steps are followed for the second update removal, but in this case the head pair from the first level is removed and the head element from the second level is replaced by $\oslash$. Also, $\langle 2, 1\rangle$ is used to update the Stack ($1 + \oslash = 1$).

The continuous usage of this operation results in each element being removed, without updating any value. Furthermore, each pair is traversed once to update the Stack and *computation_right_result* affects only $O(\log n)$. Therefore, the amortized cost for a single removal from the window is $O(1)$.

This process does not make use of the inverse functions of the aggregation operation to subtract the evicted updates, which would run in worst-case $O(1)$ time. For example, if we sum $[1, 2, 3]$ the result would be 6. When evicting 1, we could use the inverse function with result $6 - 1 = 5$ in one step. The problem is that the inverse

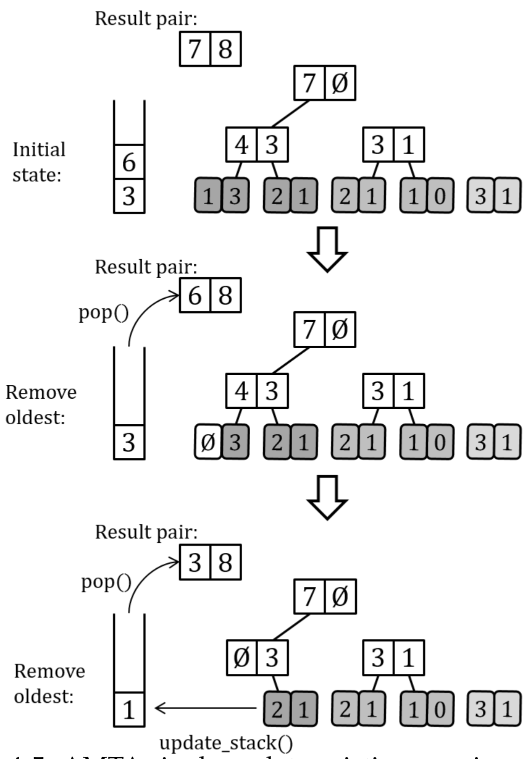


FIGURE 4.5: AMTA single update eviction running example

function does not always exist or is easy to find. AMTA single eviction mechanism provides an equivalent computational cost with a less restrictive aggregation programming interface.

Amortizing Bulk Evictions

Enforcing the WSP, like in Log MTA, removes updates from oldest to newest while the WSP is satisfied. In this case, the WSP enforcement starts by checking the head update of the window. If the WSP condition is met, the update is removed using the single update eviction operation. For constant-size sliding windows, this solution already runs in amortized constant time with logarithmic time in worst case scenario.

However, the worst time would become linear with variable-size windows. Our solution is to use the bulk eviction after the WSP enforcement removed a factor of $\log n$ elements from the Forest using the single update eviction. Theorem 1 is valid for the independent trees in the Forest. Therefore, Algorithm 2 can be applied to a single tree in the Forest while keeping the same cost. This solves the problem of evicting a partial amount of updates from a tree, which only affects one tree. If other trees from the forest have updates evicted, they will be the older ones and will

Algorithm 6 AMTA's single update eviction from the data structure C

```
1: $S \leftarrow stack(C), L \leftarrow levels(C), l \leftarrow 1$
2: $removed\_pair \leftarrow true, new\_heads \leftarrow \{\}$
3: $R_0 = S(pop())$
4: while $removed\_pair \wedge l \leq |L|$ do
5:     $P \leftarrow L_{l,1}$
6:     if $removed\_pair \leftarrow (P_0 = \oslash \vee P_1 = \oslash)$ then
7:         $L_l(remove\_pairs(1))$
8:         $P \leftarrow L_{l,1}$
9:         if $P_0 \neq \oslash \wedge P_1 \neq \oslash$ then
10:            $new\_heads(push(P))$
11:        end if
12:    else
13:        $P_0 \leftarrow \oslash$
14:    end if
15:    $l \leftarrow l + 1$
16: end while
17: $C(update\_stack(new\_heads))$
18: if $removed\_pair$ then
19:     $C(compute\_right\_result())$
20: end if
```

Algorithm 7 AMTA Stack update. Updates S from *new_heads* in the data structure C

```
1: $S \leftarrow stack(C),$
2: while $|new\_heads| \neq 0$ do
3:     $P \leftarrow new\_heads(pop())$
4:     $S(push(monoid(P_1, S(peek()))))$
5: end while
```

be evicted completely. In terms of forest levels traversing, the eviction of elements can be done by traversing the levels only once, similarly to LMTA; either a level is completely removed (level composed exclusively with evicted trees), some elements from the level are removed (level with a partially evicted tree), or there is no element removal at all. The only precondition is to recompute the values from the leftmost branch from the forest, in order to make them consistent, which is a $O(\log n)$ process.

4.5 Evaluation

The evaluation is divided into four experiments concerning different aspects from the MTA Window Framework and state of the art general sliding window solutions.

The analysed algorithms correspond to implementations of Amortized MTA, Log MTA, *DABA* and *Naive* window aggregation. *DABA* is the featured algorithm from the state of the art SWAG framework [90], discussed in Section 4.3. All *DABA* operations are $O(1)$, but it does not feature a bulk eviction mechanism. Therefore, performing an eviction of n elements is $O(n)$, which is amortized with a higher worst-case than AMTA. On the other hand, the *Naive* approach aggregates all the elements in the window every time a new result has to be produced.

All algorithms use monoids as the aggregation mechanism, so we are evaluating sliding window algorithms that do not need to have invertible aggregations. Additionally, we will use MTA's WSP mechanism in all the algorithms, with an adapted WSP enforcement. Both *DABA* and *Naive* will use the head element in the window individually as the subsequence to compare in the WSP, because they don't have efficient bulk eviction mechanisms.

4.5.1 Implementation

All algorithms are implemented in Java 1.8 and executed as operators in an *Apache Storm* based stream processing runtime called *rapids*. *rapids* processes all data units as objects with a shared class and several data dimensions, meaning that updates and partial results will be objects with multiple values rather than single scalar values. The purpose of running the algorithms in *rapids* rather than isolated is to show how they perform in a production environment.

MTA Window Framework will be evaluated in two different implementations: one where the algorithm's data structure resides in pre-allocated local memory, and another with the KVS-based data structure described in the previous sections. The local memory implementations of MTA replaces each level's Cache-based KVS interaction mechanism by a simple CircularFifoQueue. They compare on equal terms with *DABA* and *Naive* aggregation, as neither of them have a data structure adapted to work with remote data stores. For these algorithms, the data structure is preallocated and never reallocated, to avoid evaluating the latency added by performing incremental memory allocation strategies or static resizing. Furthermore, *DABA* implementation contains the optimizations described by its authors regarding caching results (*Cached DABA*). They have been evaluated on its most favourable implementation for the *rapids* runtime. The MTA local memory implementations will be referred as *Mem. LMTA* and *Mem. AMTA*, while the memory decoupled versions will be *KVS LMTA* and *KVS AMTA*.

All tested algorithm implementations include a WSP enforcement mechanism. For Amortized MTA and Log MTA, the WSP enforcement algorithms are the ones described in Section 4.4, including the $O(\log n)$ bulk eviction. *DABA* and *Naive* aggregation WSP enforcement check the first elements in the window, one by one, as the algorithms themselves do not have the capability to perform efficient bulk evictions.

KVS LMTA and *AMTA* buffer up to 512 new elements from the data structure before storing them to a distributed data store. Each level have a cache containing up to 512 elements retrieved from the data store. When a level cache size is less than 256, it synchronizes with the data store to fill it up if possible, depending on the size of the level. Moreover, the data store used in the experiments is Couchbase [42]. Couchbase is a KVS based on memcached [50], with a distributed LRU cache in RAM. It prioritizes access in memory over disk for low-latency.

Naive window aggregation consists of a fixed size circular queue. When an update is inserted or removed to the window, it is simply inserted or removed from the queue. Querying the result aggregates all the updates contained in the queue, if it does not have the result already cached.

4.5.2 Optimizations

On top of the main algorithms that were previously explained, some optimizations were used for the evaluation. Those were not included in the description of the main algorithms for the sake of simplicity.

Aggregation results are cached for all the algorithms evaluated. While a cached result value is valid, no computation needs to be performed to produce a result. After a new insertion or eviction from the window, the cached result is flagged as invalid and the aggregation final result will need to be computed.

In Amortized MTA, both an update insertion and the WSP enforcement might trigger a full Result Pair recomputation. It can happen that the arrival of a new update triggers a full result pair recomputation twice, if both the insertion and WSP enforcement require so. In order to avoid such a situation, result pair recomputations are requested by each stage, but they are executed only once after the operations finished.

KVS LMTA and *AMTA* communication with the data store is done in the background when it is possible. Storing the buffered elements is always a background operation. However, although updating a level cache is also performed by background threads, a cache miss will always require a synchronous update.

DABA contains all the optimizations defined in its corresponding papers (*Cached DABA*).

4.5.3 Environment

The experiments were run in a cluster with 2-way Xeon E5-2630 (broadwell) v4 clocked at 2.20GHz nodes. Each one features 128GB of DDR4-2400 R ECC RAM. All nodes were interconnected using a non-blocking 10GbE switching fabric. Although an external NFS folder was mounted on the systems, it was not used as a backend for the experiments. Instead, all data was stored locally using four 7.2K rpm 2TB SATA HDDs per nodes, mounted as four independent volumes. Experiments comprising *Naive* aggregation, *DABA*, *Mem. AMTA* and *LMTA* only used a single node. *KVS AMTA* and *LMTA* logic was executed in a single node, but Couchbase ran as a cluster in three extra nodes. Therefore, the contents of both algorithms data structures were distributed between 4 nodes.

LISTING 4.2: Experiments' monoid

```
function monoid(left, right){
  Element result = new Element();
  result.count = left.count + right.count;
  result.maxSize = right.maxSize;
  return result;
}
```

LISTING 4.3: Experiments' WSP

```
function wsp(total, old){
  return total.count - old.count >= total.maxSize;
}
```

4.5.4 Experiment 1: Constant-sized window latency

In this experiment we analyze the average latency of inserting a new update and generating a result with a constant-sized window. Its aim is to demonstrate the effective time complexity of each algorithm, and how they compare to each other. Each measurement was performed for different window sizes by inserting one update to the window, removing the oldest one, and retrieving the total aggregation. The user

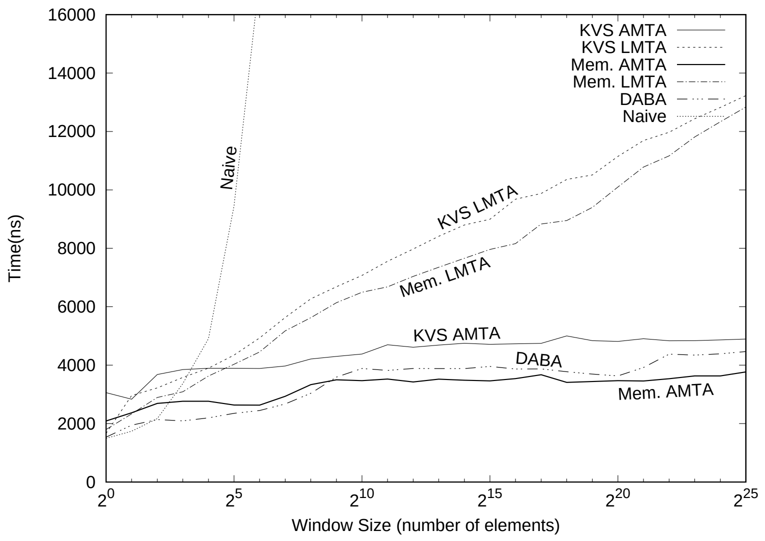
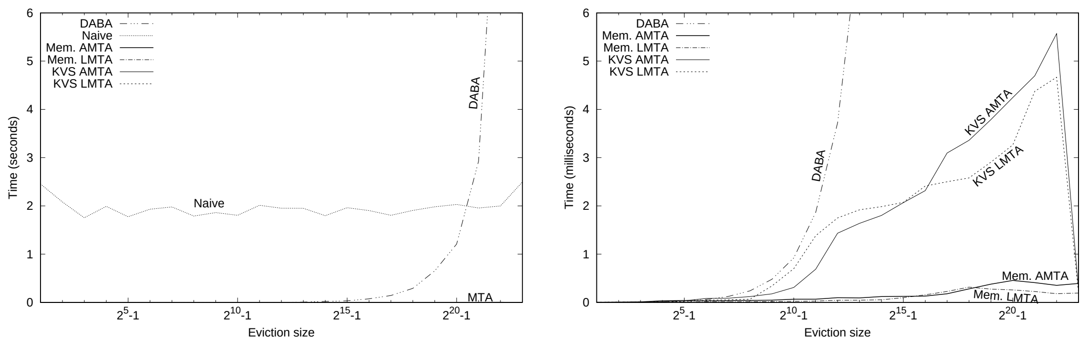


FIGURE 4.6: Average latency for constant-sized windows

defined operations for this experiment are the monoid in Listing 4.2 and the WSP in Listing 4.3. Updates and partial results contain two dimensions: *count* and *maxSize*. *count* is always 1 on an update inserted to the window, as it counts itself. *maxSize* establishes the size of the window, and so it is used by the WSP to remove updates from the window when this size is exceeded. The evaluated window sizes go from 1 to 2^{25}. Each iteration of the experiment starts by filling the window up to *maxSize*. Once the window size is *maxSize*, update insertions are performed until all the initial updates from the filling up stage are removed by the WSP, hence traversing all the window possible states. The latencies shown in Figure 4.6 for each window size correspond to the average latency of the process triggered by an update insertion, including aggregation, WSP check and update removal. The chart is drawn in a logarithmic scale for the x-axis for clarity.

As it can be observed, *Naive* aggregation initially has the lowest latency, but it grows linearly with the window size and rapidly becoming the obvious worst-performant algorithm in terms of time complexity.

As it was expected, *AMTA* and *DABA* show a constant time complexity behaviour. Being *Mem. AMTA* the algorithm with the lowest latency with a window size 2^9 or greater, its distance with *KVS AMTA* is relatively low and affordable given the memory usage benefits. The impact on storing the majority of the data in a distributed data store is around 1 microsecond with the greater window sizes and less than 500 nanoseconds compared to *DABA*. This is the result of keeping data



(A) General View

(B) Detail of *MTA*: Y scale in milliseconds

FIGURE 4.7: Window bulk eviction average latency, using different y-axis scales to show different details

store communications asynchronous when possible. The same difference can also be appreciated in the *Log MTA* implementations, which has a the expected $O(\log n)$ behaviour.

This experiment proves that the theoretical complexity for constant-sized window is also shown in practice. Furthermore, the average AMTA latency for constant-sized windows goes in line with the state of the art, and the data-computation decoupling performed in *KVS AMTA* and *LMTA* have marginal a effect for constant-sized windows.

4.5.5 Experiment 2: Bulk eviction latency

This experiment evaluates the variable-sized windows scenario. In these cases, several updates need to be evicted from the window triggered by a single new update insertion. Using the monoid in Listing 4.2 and the WSP in Listing 4.3, we measured the average latency of the *enforceWSP* operation for each algorithm. The windows are initialized with the same initial size: 2^{23} updates. A series of iterations evict from 1 to $2^{23} - 1$ updates per insertion, averaging its latencies for each removal size. The results can be seen in Figure 4.7, divided in two different y-axis crops to visualize distinct groups of results, one in seconds and the other in milliseconds. The x-axis have a logarithmic scale.

Figure 4.7a is the global view and emphasizes *DABA* and *Naive* windows. *Naive* aggregation bulk eviction latency is around 2 seconds constantly. All updates in the *Naive* window are aggregated when generating a result. On the one hand, it needs to aggregate all the updates checked by WSP after an insertion. On the other hand, it also needs to aggregate the remaining elements to produce a result for the operation. Therefore, the number of aggregated elements remains constant. Furthermore, *DABA* has a clear linear latency growth behaving worst than *Naive* when removing sub-windows with size $2^{20} - 1$ or greater, and becoming an unfitted operation for real-time stream processing.

Figure 4.7b reduces the y-axis scale by three orders of magnitude, and *Naive* window is now out of the scope of the chart. It focuses on comparing the four *MTA* solutions and *DABA*. Bulk eviction latencies are very similar between the *KVS MTA* implementations, growing logarithmically. *KVS LMTA* is a almost a millisecond faster for most periods as its WSP enforcement process has the same complexity but fewer stages, i.e. trying multiple single update evictions. In this scenario, they suffer from the greatest impact of having the majority of the data in a distributed data store. The consistent latency growth from *KVS MTA* compared to the *Mem. MTA* counterparts is due to the data store query time, triggered by cache misses. However, the latencies decrease significantly in the last iteration, because the data can be found locally in the structure buffer. Maximum latency for both algorithms is around 5 milliseconds, 400 times less than *Naive* window running completely in local memory. The effect of having most of the data structure in a *KVS* is noticeable by comparing them with *Mem. AMTA* and *LMTA*. The *Mem. MTA* algorithms have the best time performance: *Mem. AMTA* has 455 microseconds worst latency and *Mem. LMTA* 258 microseconds. Note that *Mem. LMTA* also performs better for the greater part of the iterations than *Mem. AMTA*, like in the *KVS* scenario.

AMTA Framework shows a significant improvement compared to the state of the art for bulk window evictions. For big window bulk evictions, even *KVS MTA* behaves faster than the memory allocated *DABA*, while the *Mem. MTA* solutions is faster throughout the execution. The latency/memory tradeoff offered by *KVS MTA* is demonstrated later on.

4.5.6 Experiment 3: Stream analytics latency

The previous experiments show how the different algorithms behave in terms of latency. In this experiment we evaluate how different real window aggregations

	Naive	KVS LMTA	Local LMTA	DABA	KVS AMTA	Local AMTA
Sum	$3.69 \cdot 10^8$	14 320	10 341	4 288	6 253	4 163
Mean	$3.26 \cdot 10^8$	14 027	10 378	4 389	6 111	4 033
G. Mean	$2.83 \cdot 10^8$	15 267	11 166	4 795	6 198	4 183
Std. Dev.	$3.5 \cdot 10^8$	15 439	12 934	4 880	6 131	3 864
Max	2 554	8 501	6 188	3 500	2 886	1 763
LIS	19 294	22 306	19 794	10 027	7 350	6 476

TABLE 4.2: Window latencies in nanoseconds with different monoids and WSPs

behave with each algorithm. The analysed stream consists on 62 208 000 updates monitoring computer memory usage, one reading per second for two years.

This stream has been subjected to different operations performed by the window monoid: *sum*, *mean*, *geometric mean*, *standard deviation*, *maximum* and *longest increasing subsequence* (LIS). The particular case of *LIS* is the most complex one, since it measures multiple dimensions: initial timestamp, final timestamp, interval covered by the subsequence, and the number of updates in the subsequence.

In terms of WSP, there is a general rule for all the operations: the window contains at most 2^{20} elements. This policy alone makes the window static-sized. However, *max* and *LIS* extend the size limit policy: *max* operation evicts the older subwindow not containing the maximum value in the window, and *LIS* operation evicts the older subwindow not containing any portion of the LIS. Updates older than a max value or a LIS are never going to contain a future new result, it will only be found within newer updates. Therefore, these updates are not necessary to perform the aggregation and the memory they are using can be cleared. By doing that, an efficient bulk eviction mechanism can reduce the total time of evictions performed during the whole data stream analysis.

Table 4.2 shows the mean latency in nanoseconds for each operation and sliding window algorithm. All operations run faster in Local AMTA than in the other algorithms. In DABA they behave slower but similar to Local AMTA, except for *max* and *LIS*, where the difference is more noteworthy. Both operations clearly benefit from reducing the number of single evictions in both KVS and Local AMTA, getting better performance than executed in DABA. These operations also perform well in the Naive algorithm, being Naive the second best algorithm to run *max*. The cost of an insertion in the Naive algorithm without evicting any update is as cheap as performing a single monoid execution, while the evictions cost is very expensive but constant for evicting any number of updates (Figure 4.7a).

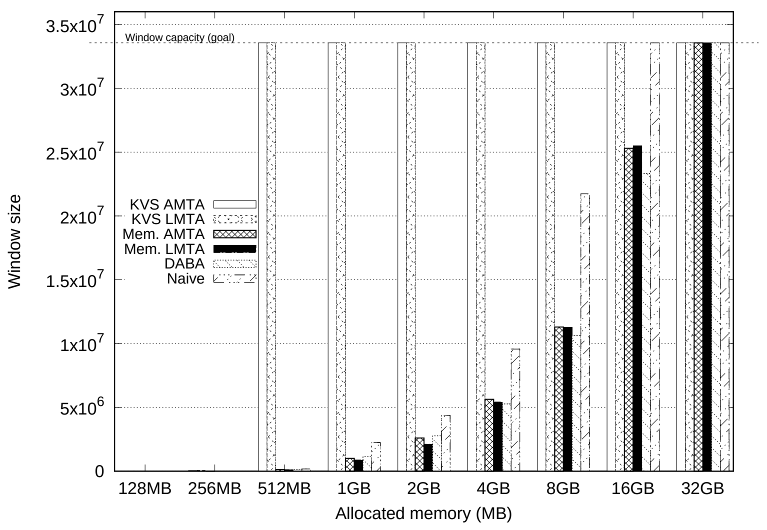


FIGURE 4.8: Average window size reached per allocated memory amount, for a 2^{25} updates capacity.

In this experiment we proved that the performance and time-complexity exhibited in the previous experiments has a relevant impact in different stream analytics on real data. Furthermore, the experiment tests multiple distinctive monoids and WSPs, analysing their impact rather than testing only the algorithms with a minimal aggregation. It shows consistency with the theoretical complexity of each algorithm and their tested performances.

4.5.7 Experiment 4: Memory requirements

This experiment evaluates the local memory requirements in order to run each sliding window algorithm in *rapids*. As previously introduced, *rapids* is a stream processing platform written in Java. For this experiment, we assigned different memory heap sizes for the Java Virtual Machine (JVM), up to 32GB; and for each size, the sliding window algorithms were executed individually, with capacity for 2^{25} updates.

There are three possible outcomes for each execution: In the first one, the window is filled up and older updates start to be removed, showing a normal behavior. In this case the window size reached is its capacity and the goal is met. In the second one, *rapids* runs out of memory as the window requires more memory than the heap provides, and the last window size measured is the reached window size. In the third one, the computation becomes very slow because of the lack of memory and the impact of the JVM Garbage Collector (GC). Given a timeout for update computations set to 5 seconds, when exceeded, the last window size measured is the reached

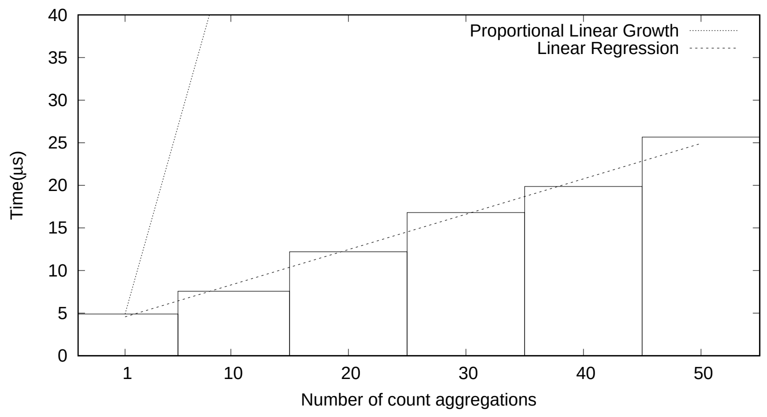


FIGURE 4.9: Average latency for multiple aggregators

window size. This experiment was done using the same GC (Java 8 G1GC) as in the rest of experiments.

Figure 4.8 shows the average window sizes for each tested heap size, for capacities of 2^{25} updates. Not appreciated in the chart but relevant, is that *KVS AMTA* was able to insert 1 281 and 59 276 updates with heap sizes of 128MB and 256MB respectively, while *KVS LMTA* was able to insert 1 459 and 66 171 updates. *Mem. AMTA*, *Mem. LMTA* and *DABA* were able to insert updates from 256MB heap size and greater, starting with 1 015, 877 and 1 120 updates each. *Naive* inserted elements from 512MB heap size and greater, starting with 171 923 updates.

The reasons why the *KVS* algorithms *AMTA* and *LMTA* start inserting messages with less memory is their reduced need of allocated memory for the empty data structure, being $O(\log n)$ compared to $O(n)$ in the other algorithms. Also notice that *KVS AMTA* and *LMTA* reached the window capacity with 512MB of heap memory behaving normally. This size is smaller by far compared to the heap sizes of the other algorithms. Except for *Naive* that reached the window capacity with 16GB of memory heap, the rest did not reached such capacity until memory heaps of 32GB. This proves the memory-wise benefits of using the AMTA Framework by decoupling most of the data from the local memory aggregation. It also shows that *KVS LMTA* has a slightly better performance in terms of memory usage than *KVS AMTA*, in addition to the capacity to perform fast bulk evictions.

4.5.8 Experiment 5: Multi-dimensional aggregation

Finally, we evaluate the impact of operating over multi-dimensional data, by analyzing how adding dimensions to data, and making the aggregations on each dimension share resources like the sliding window data structure and the WSP), affects the average computation latency.

Streams can contain synchronous dimensions of data, and the window can aggregate each one individually in the user defined monoid. E.g. dimensions like wind speed, humidity, and temperature, coming from the same stream, might need to be independently averaged with the same WSP. Here we ran a constant-size *KVS AMTA* window with the WSP from Listing 4.3 and with $maxSize = 2^{15}$, then measured the latency of update insertions for a different number of stream dimensions. The dimensions in the stream are *maxSize* and a k number of *count* dimensions (from $count_1$ to $count_k$). We chose dimensions with simple aggregations in order to quantify the overhead around them. Figure 4.9 shows: 1) the average latency for k from 1 to 50 as a barplot, 2) the linear regression on the collected results, highlighting the latency growth, and 3) how the latency would sum if each dimension was sequentially aggregated in different windows, repeating operations like data structure management or WSP with their corresponding latencies. E.g., the latency from $k = 1$ being 4 895 nanoseconds, with $k = 2$ it would be $4\,895 \times 2 = 9\,790$.

We can see that the linear regression grows slower than the proportional latency, as the monoid computation is a small fraction of the average latency for $k = 1$. The latency of a single *count* aggregation is quantified in 411 nanoseconds and $4,158$ nanoseconds are spent differently and shared between data dimensions. The latency grows linearly with the number of dimensions, although the monoid's impact would be higher depending on the operators used.

4.6 Conclusions

In this chapter we have introduced the Monoid Tree Aggregator Window Framework, a new framework for general sliding window aggregation that advances the state of the art in several aspects: 1) it exhibits an amortized constant $O(1)$ time-complexity between updates, and for the worst-case scenario it exhibits logarithmic cost $O(\log n)$ ahead of the linear cost $O(n)$ of the current existing solutions; 2) it includes a general aggregation mechanism that uses binary associative operations, and a general mechanism to enforce the Window Slide Policy (WSP) with amortized cost $O(1)$, both programmable by framework users; 3) it provides a mechanism to automatically enforce the Window Slide Policy, which enforces efficient bulk data evictions with cost $O(\log n)$ which, to our knowledge, is not supported by any other existing framework; 4) it provides support for multi-dimensional data aggregation, that can be also leveraged to implement the Window Slide Policies; and 5) it was designed to support a scalable implementation backed by a distributed key/value store instead of leveraging local memory only.

The framework has been presented through a detailed description of the main algorithms involved in the manipulation of the critical data structures of the sliding window. The framework has been implemented in two flavours: a local version in which all data is stored in memory and a remote-store version that leverages a distributed Key-Value Store to keep most of the data. In both cases, the algorithms have been implemented on top of Apache STORM, which has been used as the streaming platform, providing a multi-tenant environment to build several sliding window aggregations in parallel. A comprehensive evaluation has been conducted to proof the efficiency of the implementation, and results show that the framework can manage large windows (up to tens of millions of elements) efficiently, with a cost in the order of a few microseconds to insert elements and slide the window. The experiments on bulk data eviction show that the cost of removing large amounts of elements from the window is extremely low, which is a critical requirement for implementing efficient and reactive Window Slide Policies that drive the criteria to include or exclude elements in the sliding window.

Chapter 5

Approximate Sliding Window Framework with Error Control

5.1 Introduction

Data stream aggregations are a critical requirement for many data mining and monitoring scenarios. Such scenarios, like telemetry data analysis in large data centers, or advance analytics for the Internet of Things, often require continuous low-latency aggregation of vast amounts of data and immediacy of the aggregation results in order to produce fast on-site actuations. Processing data close to the source also becomes an important factor when data flow is expensive due to high volume of data and poor connectivity. The environments in which the data analytics need to be computed are not always favorable. Low power consuming hardware, limited resources and unreliable internet access are usual conditions for Smart Cities and Fog Computing [32].

As we saw in Chapter 4, due to the unbound nature of streams, sliding windows are a convenient approach to process aggregations on data streams. However, the size of the contents in a window can still be considerably big and this can have a big impact in terms of performance. Therefore, sliding windows ideally also need to: a) have low latency and low time complexity, b) work with low memory resources and unreliable connectivity. Chapter 4 introduced Amortized Monoid Tree Aggregator (AMTA) as an amortized constant-time sliding window framework with its contents distributed in a *Key-Value Store* (KVS) instead of residing in local memory. AMTA takes advance of incremental aggregation algorithms optimized for distributed fault-tolerance data replication, in order to free the local memory from the window data-structure. The aggregation functions can be provided by the user with a *MapReduce*-like programing model, in which the *reduce* function is an associative operation (*monoid*). However, when the connectivity to the KVS is unreliable, the window aggregation will either fail or its data-structure will indefinitely grow in local memory. On the other hand, when the connectivity is reliable, the data-structure might still be consuming a substantial amount of shared resources from the KVS cluster that could be used for multiple additional aggregations.

In this chapter we introduce the Approximate and Amortized Monoid Tree Aggregator (A^{2}MTA) general window aggregation framework. A^{2}MTA is an approximate aggregation framework that benefits from the work in AMTA as to: 1. Amortized constant $O(1)$ time-complexity between updates, while logarithmic $O(\log(n))$ in the worst-case scenario. 2. Distributed and replicated data-structure in a KVS, freeing local resources and facilitating a fault-tolerance system. 3. Only $O(\log(n))$ of the data in the data-structure sits in local memory. 4. User-programmable window aggregation mechanism and window slide policy. 5. Bulk update evictions triggered by the window slide policy are considered atomic operations and have a worst-case $O(\log(n))$ cost.

On top of AMTA, A^{2}MTA provides a set of mechanisms that reduce considerably the size of the data-structure and the computation time, in exchange of a degree of error in the aggregation results. In other words, provides an approximate computing framework for scalable sliding window aggregations. In this scenario, the granularity in a sliding window contents is divided into multiple aggregated updates, or update buckets, instead of individual stream updates. For instance, in a summation window, we keep only update buckets containing the summation of k updates instead of k separated updates. When evicting stream updates from the window, the minimum unit to be evicted are whole buckets, even if only a portion of the bucket needs to be evicted. More specifically and within a defined confidence level, A^{2}MTA defines buckets by:

- Aggregation error control. The aggregation in a bucket is used to estimate its impact in the window aggregation result and contain it. In cases in which enforcing a level of error is necessary to make buckets grow, that error is bounded by the user.

- Size of predicted bulk evictions. Frequent and highly probable evictions of at least k updates will entail buckets with k aggregated updates.

- Maximum number of buckets, as an ultimate memory resources restriction (constant $O(1)$ size). Updates are spread out among buckets to distribute the weight of the window in order to comply with the restriction.

- Network availability. The number of buckets in local memory will be limited. Therefore, when it is not possible to send them to a KVS, the buckets need to aggregate more updates. This can done by dynamically reducing the maximum number of buckets.

The rest of the chapter is structured as follows: Section 5.3 defines the Approximate AMTA Framework; Section 5.4 provides the results of an experimental evaluation of the Approximate AMTA Framework; Section 5.5 discusses the state of the art in the fields of both approximate computing and efficient sliding windows; Finally, Section 5.6 summarizes the conclusions extracted from this work.

5.2 MTA Enhancements

The work in this chapter is strongly bound to AMTA, which can be found in Chapter 4. After finalizing AMTA related contributions and its evaluation, we spend some efforts to add some improvements to its base algorithms and mechanisms. These enhancements are not part of the main contribution presented in this chapter. However they are introduced here, because they were used in the evaluation performed in Section 5.4, but not implemented for Chapter 4 and its evaluation. In this section we summarize the aspects of AMTA that are affected, and introduce the additional features in the Window Slide Policy definition and its enforcement in bulk evictions.

5.2.1 Window Slide Policy Definition

Like the monoid aggregation operation, the Window Slide Policy is a user-programmable condition with an specific structure. WSP will be divided into two parts, and at least one needs to be defined by the user:

Window Invariant. It is a function that has a window result candidate as an input, and it evaluates if the result candidate is valid. A valid window result candidate does not require any more evictions. If it is not valid, an undisclosed amount of updates need to be evicted to make it valid. For example, if a window result contains the number of aggregated stream updates in the field *count*, the number of stream updates in the window could be limited to 1 000 with this invariant:

$$window.count \leq 1\,000$$

Eviction Invariant. WSP can additionally be defined as a comparison between an aggregated sequence of values considered to be evicted, the window result before the eviction and the window result after the eviction, focusing this time in the eviction and its effect on the window. If the invariant is valid, at least the eviction candidate needs to be evicted from the window. For instance, a sliding window calculating the maximum value within 1 000 updates does not always need to keep the 1 000 updates inside the window. The two requirements are to aggregate at most 1 000 updates and that the oldest update in the window has the maximum value. In order to do that, the user would use the previous example's window invariant and the following eviction invariant:

$$eviction\_candidate.max \leq post\_eviction\_window.max$$

This eviction invariant triggers the eviction of all the updates older than the current max value. This updates will not affect the aggregation result in the future and their eviction both frees resources and boosts performance of future aggregations.

A window will not need any eviction if its result satisfies the window invariant, and if the oldest update does not satisfy the eviction invariant as eviction candidate. This WSP definition mechanism, based on the aggregation contents, enables the user to define from the most basic WSP to complex and dynamic scenarios using sophisticated aggregations.

5.2.2 Bulk Eviction

The enforcement of a WSP finds the greatest window that satisfies the window invariant with the greatest eviction candidate that satisfies the eviction invariant, using the current window contents. WSP's window invariant and eviction invariant have a set of rules for increasing or decreasing the size of evictions:

- Window invariant.
  - When satisfied, any smaller window result candidate is assumed to satisfy the invariant, but bigger window candidates might also satisfy it.
  - When not satisfied, the window candidate needs to evict more updates.
- Eviction invariant.
  - When satisfied, any smaller eviction candidate is assumed to satisfy the invariant, but bigger eviction candidates might also satisfy it.
  - When not satisfied, the eviction candidate needs to be smaller.

The method used to enforce this set of rules can be found in Algorithm 8. It is divided in three phases: find the top level to prune, remove levels of exclusive to evicted trees, and remove elements from levels with a partially evicted tree. It does not only differ from the algorithm used in AMTA by using a more complete WSP, but also by initially searching the top level to prune instead of either doing single evictions or root-to-leaf evictions.

Algorithm 8 is a formalization of this method. wc, w, ec and e stand for *window candidate*, *window*, *eviction candidate* and *eviction*.

The loop from line 3 to line 13 uses the eviction stack to calculate the window candidate and finds the highest level (l) that needs to be pruned. From line 14 to line 30, it removes top levels containing exclusively evicted trees. $result\_pairs\_stack()$ builds a stack containing the data structure result pairs after evicting each tree: the top result pair in the stack is the current result pair, and the following ones aggregate one tree less each. These result pairs are used to This operation is $O(\log n)$ since it consists on incrementally aggregating the root pairs of each tree, see Algorithm 9.

From line 31 to the end of the algorithm, it removes the evicted trees from each level and prunes the partially evicted tree.

Algorithm 8 A^{2}MTA bulk eviction in data structure C

```
1: $L \leftarrow levels(C), S \leftarrow stack(C), R\ getsresult\_pair(C)$
2: $wc, w, ec, e \leftarrow \oslash, r \leftarrow monoid(R_0, R_1), hop \leftarrow 0$
3: for $l = 1, ..., |L|$ do
4:     $L_l(recompute\_first())$
5:     $wc \leftarrow monoid(S(peek)), R_1)$
6:     $ec \leftarrow L_{l,0}$
7:     if $window\_invariant(wc) \wedge \neg eviction\_invariant(ec, wc, r)$ then
8:         $l \leftarrow l - 1$
9:         $w \leftarrow wc$
10:        break
11:    end if
12:    $S(pop())$
13: end for
14: if $l = |L|$ then
15:     $results\_stack \leftarrow result\_pairs\_stack(C)$
16:     for $m = l, ..., 1$ do
17:         $R \leftarrow results\_stack(peek())$
18:         $wc \leftarrow R_1$
19:         $ec \leftarrow monoid(e, R_0)$
20:         if $\neg window\_invariant(wc) \vee eviction\_invariant(ec, wc, r)$ then
21:             $results\_stack(pop())$
22:             $e \leftarrow ec$
23:             $hop \leftarrow |L_l|)$
24:             $L_l(remove())$
25:         else
26:             $w \leftarrow wc$
27:             break
28:         end if
29:     end for
30: end if
31: if $hop = 0$ then
32:     $P \leftarrow L_{l,0}$
33:     $P_0 \leftarrow \oslash$
34:     $hop \leftarrow 1$
35: end if
36: for $n = m, ..., 1$ do
37:     $P \leftarrow L_{l,0}$
38:     $pairs\_hop \leftarrow 2 \cdot hop$
39:     if $P_0 = \oslash$ then $hop \leftarrow hop - 1$ end if
40:     $L_n(remove\_pairs(pairs\_hop))$
41:     if $P_0 \neq \oslash$ then
42:         $wc \leftarrow monoid(P_1, w)$
43:         $ec \leftarrow monoid(e, P_0)$
44:         if $\neg window\_invariant(wc) \vee eviction\_invariant(ec, wc, r)$ then
45:             $e \leftarrow ec$
46:             $hop \leftarrow hop + 1$
47:             $P_0 \leftarrow \oslash$
48:         else
49:             $w \leftarrow wc$
50:         end if
51:     end if
52: end for
```

Algorithm 9 *result_pairs_stack* method in data structure C

```
1: $results\_stack \leftarrow \{\}$
2: $accum \leftarrow \oslash$
3: for $l = 1, ..., |L|$ do
4:     $P \leftarrow L_{l,|L_l|}$
5:     $p \leftarrow monoid(P_0, P_1)$
6:     $results\_stack(push(\{p, accum\}))$
7:     $accum \leftarrow monoid(p, accum)$
8: end for
```

5.3 Approximate AMTA

Approximate computing is a widely used paradigm in data analytics algorithms that can drastically reduce the needed resources in order to obtain a result. It relies on the degree of tolerance a system may have to some loss of quality or optimality in the computation result.

In this section we introduce Approximate AMTA (A^2MTA), a sliding window framework that assumes the AMTA, data-structure, based on binary trees. The leaves level of a tree contains the values inserted to the window, while the rest of the tree levels contain partial incremental aggregations. Depth-wise, the closer a node is to the root, the more updates it aggregates. Breadth-wise, the closer a tree node is to the leftmost branch, the older the aggregated updates are. The aggregation functions are *monoids*: binary associative functions with a neutral element and function inputs and output from the same set; i.e. $+$ monoid is a binary and associative function, its neutral element is 0 and integer inputs result in integer outputs. This data-structure has been demonstrated to keep its amortized constant-time, efficient bulk evictions and enable horizontal scalability with a distributed data store.

In A^2MTA we propose to only keep partial aggregations from consecutive updates, called buckets, building the window as a histogram of updates. For example, a count window with the updates $[1, 1, 1, 1]$ could be $[2, 2]$ in A^2MTA. The required memory can be drastically reduced with this method, and we will prove that the performance is also improved. However, the aggregation result might not be accurate due to having effectively too many or too few updates in the window, due to the coarse granularity given by the buckets.

Consider the scenario pictured in Figure 5.1, in which we have a static size sliding window that performs an update count using buckets. Its WSP limits the number of counted updates up to 10. Since it is a count operation, all the input updates will have value 1 and the result should always be 10. The window requires to evict one update for every insertion. However, once the window result reaches 11, a bucket with value 3 is evicted. The result is finally 8, instead of 10, generating a result error of 2. In other words, any bucket eviction policy may turn out to result in: a *false positive* bucket, by keeping a bucket aggregating updates that need to be evicted; a

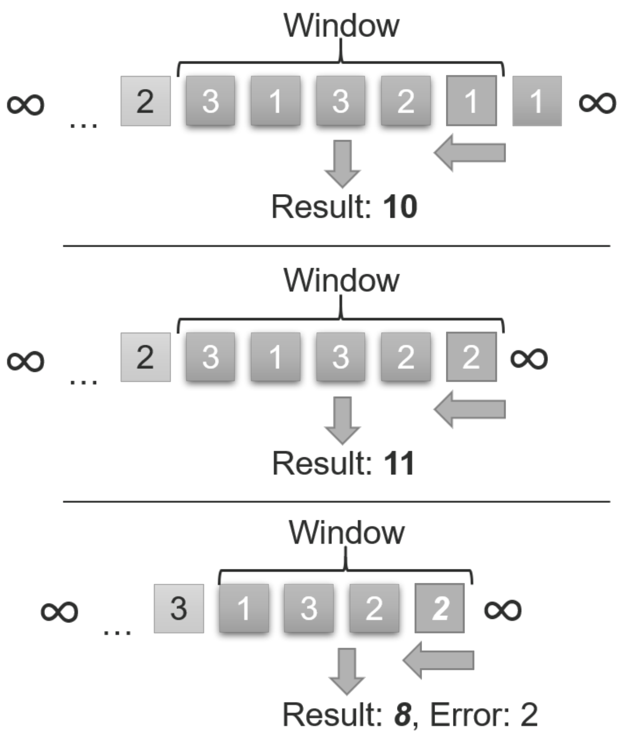


FIGURE 5.1: Error generated by stream update buckets.
Monoid: *count*; WSP: $count > 10$

false negative bucket, by removing a bucket aggregating updates that should be kept in the window. Note that removing a false positive bucket would result in a false negative bucket, and vice versa.

In order to mitigate the effects of false positive/negative bucket error, the proposed methods in this section decide whether a new update must start its own bucket in the window, or it must be aggregated to the newer existing bucket. This is done by either: controlling the result error, keeping a reduced number of inaccurate results, or prioritizing a maximum number of elements in the data-structure.

Different kinds of aggregations need for specific approaches to adjust the error. Hirzel et al. [57] classify the types of window aggregations into five groups: *Sum-like*, *max-like*, *collect-like*, *median-like* and *sketch-like*. *Sum-like* aggregations compute values with invertible functions and include aggregations such as *sum*, *count* and *average*. This kind of aggregation have a single neutral element (i.e. 0 for a *sum*), and therefore the results tend to vary. *Max-like* aggregations generally make a selection of a non-ranked input update, leaving the rest of the updates without any effect on the result. They are not invertible and include algorithms like *max*, *min*, *argMax*, *argMin* and *maxCount*. Neutral elements in a *max* aggregation, for example, would be all values below the current result, therefore there is a high probability that a new update insertion does not affect the result.

Collect-like and *median-like* aggregation algorithms have collections of values as the *monoid* set instead of a single one, and therefore the result error can not be quantified with a single numerical value. *Sketch-like* algorithms are approximate computing algorithms by themselves, such as HyperLogLog or Bloom filter, and therefore will not be considered in this section either.

On the one hand, we propose two approximate computing methods with result error control, one specific for *sum-like* aggregation algorithms and another for *max-like* ones: *Sum-like histogram* and *Max-like histogram*. On the other hand, we propose two more methods that can be combined with the previous ones: *Hop histogram*, which focus aggregating frequent bulk evictions into buckets, and *Maximum size enforcement*, which forces the window data-structure to have a deterministic maximum number of buckets while keeping a uniform bucket size.

5.3.1 Sum-like histogram

Since *sum-like* aggregations only have a single neutral element, its result change whenever a non-neutral value (all except for one) is inserted or evicted. That makes this kind of aggregation improbable to keep without any error while keeping the values in aggregated buckets. The goal of value error control for *sum-like* aggregations is to make buckets grow while keeping the aggregation error under the error tolerance defined by the user.

The bucket error can be calculated as the maximum between its *false positive* and *false negative* bucket errors. The *false positive* bucket error is the maximum absolute aggregation of the oldest updates aggregated in a bucket, not including the single newest one. On the contrary, the *false negative* bucket error is the maximum absolute aggregation of the newer updates aggregated in a bucket, not including the single oldest one. For example, if a bucket contains an aggregation of the sequence of updates $[1, 1, -1, -1]$, the false positive error is $1 + 1 = 2$, while the false negative error is $-1 - 1 = -2$. In this example, both errors have the maximum absolute error value: 2. In case that we want to control the error of multiple dimensions of the aggregation (i.e. *sum* and *count* in an average aggregation), this process can be applied to each dimension.

The error can be constrained by the user either relatively to the result or as an absolute error. When the aggregation requires to calculate the error relative to the result, we need a window result prediction interval. The extremes of the prediction interval will be used to estimate the maximum error the bucket can generate when the bucket becomes a potential false positive or negative bucket. We estimate the aggregation result with a prediction interval using a sample of the previous results and assuming the *central limit theorem* as follows:

$$\left(\bar{x} - t^* s\sqrt{1+\frac{1}{n}}, \bar{x} + t^* s\sqrt{1+\frac{1}{n}}\right)$$

Where $\bar{x}$ is the sample mean, s is the sample's standard deviation, and t^* is the two tailed percentage point of Student's t distribution given a specific confidence level with $n-1$ degrees of freedom. We defined the sample as, at least, all the results generated by each update currently aggregated in the window, with a minimum of 30 elements.

The use of this method can be generalized to any kind of aggregation in terms of controlling the error on the number of elements in the window, instead of controlling the actual result of the window. This way, even non-numerical aggregations can benefit from A^{2}MTA, getting an approximate result.

5.3.2 Max-like histogram

In max-like aggregations (or extreme value aggregations) only a subset of the computed values have any influence on the result. The rest of the elements are irrelevant and the aggregation would provide the same result if they were ignored. The goal of this method is not to discard the irrelevant updates, but to aggregate all the consecutive ones in the same bucket.

For instance, in the window [3, 1] with monoid *max*, '1' might be the result of the window when '3' gets evicted. However, in the window [3, 1, 2], the update '1' will never affect the result in this window. When the update '3' gets evicted from the window, the result will be at least '2'. If we knew that the window results would be between '3' and '2', '1' could have been aggregated in the same bucket as '3', and there would not have been any difference in the result. In case that there were not other dimensions aggregated in the window, the update could even be discarded.

A^{2}MTA estimates a range of value candidates to become the result in the window. If an inserted update is found within this range, it will generate a new bucket. The update will be aggregated to the last bucket otherwise. There is no error constraint specified by the user to be considered, but the aim is to mainly produce accurate results. The range result candidates is estimated using extreme value theory [40].

The Fisher-Tippett [49] theorem states that the cumulative distribution function from a sample of size n with independent and identically distributed random variables converge to the Generalized Extreme Value (GEV) distribution, as $n \to \infty$. There are three parameters for the GEV distribution: μ for the location, σ for the scale, and ξ for the shape. By this theorem, it is possible to estimate a fitting GEV distribution given a sample of extreme values. A^{2}MTA uses Block Maxima (BM) [53] and the GEV probability-weighted moments (PWM) estimation method [59, 46]. The main reason to use PWM rather than Maximum Likelihood Estimation (MLE) method is because it performs better with small samples, and our goal is to keep the minimum amount of data possible. From the estimated GEV we will be able to extract the estimated boundaries for the extreme values.

Following the BM method, the monoid is applied to the inserted updates in blocks of an specified size. The result of each aggregated block will be added to the sample of extreme values. The sample size has been set to 30, and the block size is defined by the user. With this sample a fitting GEV distribution is computed using PWM. From there, the upper and lower bounds can be extracted. If $\xi \geq 0$, then the upper bound that we will consider will be a GEV quantile (e.g. 0.99). Otherwise, if $\xi \leq 0$ then the lower bound considered will be the remaining quantile (e.g. 0.01).

Deciding an optimal block size is out of the scope of this work, and therefore is left as a user decision, although in many situations the block periods can appear naturally [79, 35, 97]. Small block sizes would compute wide GEV boundaries and, therefore, the rate of bucket aggregations would be very low. Also, having small blocks causes a higher ratio of more costly GEV fitting computation. On the other hand, big block sizes would cause a biased GEV fitting computation that would translate to multiple inaccurate results. In Section 5.4 we compare how different block sizes behave.

This method is compatible with a window eviction policy performing bulk evictions of all the updates that precede the update that brings the window result. As it was evaluated in AMTA's article, this kind of eviction policy significantly improves time performance as it reduces average window size. Furthermore, the use of buckets improves these figures and reduce the number of inserted elements in the data-structure and in the KVS.

5.3.3 Hop histogram

Consider sliding windows programmed to evict multiple updates each time. Usually referred as *hopping window*, they remove constant amounts of updates from the window. For example a hopping window could remove 500 updates when the size of the window is 1500. In such a situation, if the updates waiting to be evicted were aggregated in a single bucket, then the size of the window would be reduced and there would not be an effective error in the result. In addition, if we managed to aggregate all the future evictions in the window into buckets, the window size reduction would be vast and produce no error at all. From the previous example, if we aggregated every 500 in a single bucket, we would have only three buckets in memory instead of 1500 updates.

However, AMTA eviction mechanism gives freedom to program dynamic sized windows with evictions sizes changing over time. That can create scenarios in which there is some variability between bulk evictions. We propose a method that aggregates the inserted updates into buckets within a predicted eviction size. We estimate the size of the evictions with a prediction interval using a sample of the previous eviction sizes and assuming the *central limit theorem*, same as in Section 5.3.1. We defined the sample as, at least, all the evictions that fit in the current window, with a minimum of 30 elements.

The prediction interval is used to predict sequences in the window with high probability to include future eviction boundaries. Figure 5.2 shows a window where every square represents an update in a bucket, with eviction size prediction interval of 6 ± 1 so far. As it can be seen, the window updates are divided into sequences

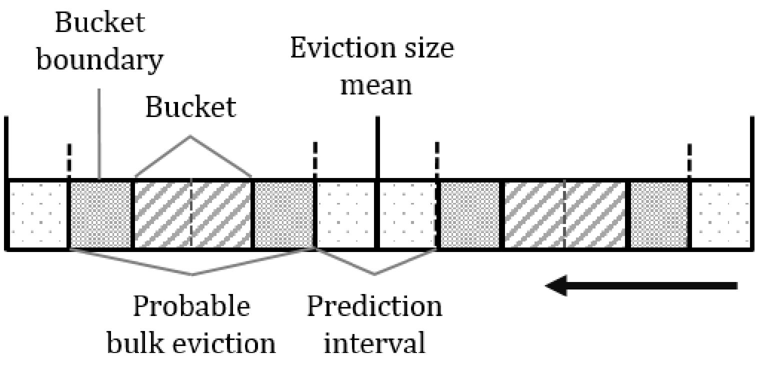


FIGURE 5.2: Bulk eviction buckets. Predicted eviction: 6 ± 1

containing as many elements as the mean eviction size. The number of elements in the probable eviction boundaries (prediction interval) cover the updates around the mean eviction size. These updates will not be aggregated in a bulk eviction bucket, avoiding introducing error to the window according to the prediction. However they will be aggregated using criteria from other methods. That leaves a block of four updates that are predicted to be evicted together in the same bulk eviction. As we saw previously, the size error in a window is generated by the false positive and false negative bucket errors. Considering this, the complete aggregation of the probable bulk eviction generates an error if it becomes the last evicted bucket or oldest bucket in the window, even if the estimation was correct. In this situation we can not know if the eviction was performed as predicted or the prediction was actually incorrect. In order to avoid the error after a successfully predicted eviction, the bucket needs to contain all the updates from the probable bulk eviction block but the ones in its boundaries, one from each side.

In case that the prediction actually failed, it should be reflected in the statistics from the eviction size sample. Otherwise the prediction interval would be biased. Therefore, the worst-case eviction size is added to the statistics sample.

5.3.4 Constrained footprint enforcement

A^{2}MTA can be executed in environments with resource constraints that need to be taken into account, either because it is running on a low-resources environment or because it is a shared multi-tenant environment. Particularly, A^{2}MTA needs a deterministic limit of its data-structure size and the network traffic a distributed data

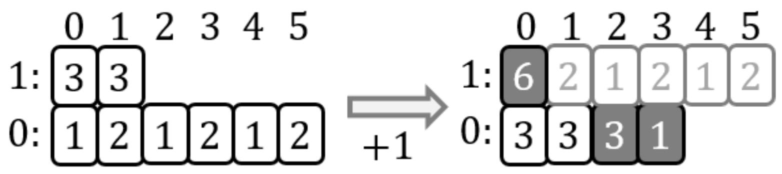


FIGURE 5.3: A^{2}MTA data-structure constrained with 6 leaves

store. The A^{2}MTA maximum size enforcement method uses two mechanisms in order to limit the data-structure size. The first mechanism is level eviction. A^{2}MTA data-structure is based on trees, in which the leaves level contain the buckets, and the rest of the tree levels have aggregations from the nodes on the previous level. Each level is a queue with level nodes and the levels are found in a circular queue. When the number of elements in the leaves level is going to be exceeded, the level is marked as empty and moved from the front to the back of the circular queue. Therefore, the new leaves level contains the previous leaves aggregated into bigger buckets.

Figure 5.3 shows the level eviction mechanism applied to an example A^{2}MTA data-structure. From the constant-time AMTA data-structure, the nodes in level 1 are the parents that aggregate the initial four elements from level 0. The size constraint in this window is of 6 leaves (level 0). After inserting the update '1', the window would have 7 leaves, exceeding the limit set. Before inserting the update, the level 1 is shifted to the position 0 and the previous leaves level is now in the position 1 and marked as empty. Then, the last pair in the previous leaves level is aggregated ('3') and inserted to level 0, and its aggregation propagated upwards generating a new root '6'. Now the window structure is consistent and contains the same aggregation as before starting the insertion, but with an evenly distributed growth of the buckets sizes. The new update is now inserted to the data-structure without exceeding the size limit.

The second mechanism for constraining the window size consist of aggregating updates into buckets with a calculated size continuously after the first level eviction. This mechanism has three main goals: keep uniformity in terms of bucket size and therefore, have a uniform update count error; reduce the number of element insertion operation in the data-structure, which can be more costly than inserting the update to the bucket; reduce the level evictions, and use them as a last resort. The max size of the currently building buckets is calculated as: $\lceil \frac{count}{constraint} \rceil$, where *count* is the number of aggregated *updates* in the window and *constraint* is the max number of *buckets* in the sliding windows. While the number of updates in the window changes, the bucket size increases or decreases with it. This mechanisms reduces the number of elements inserted to the data-structure, by aggregating new updates in existing elements. By reducing insertions, the number of nodes transferred to the AMTA distributed data store is also reduced. Therefore, when the bandwidth to the distributed data store and local memory are too low, the maximum window size can be reduced in order to reduce the data traffic in exchange for aggregation granularity.

Another advantage of having a deterministic data-structure size regardless of the number of updates aggregated is that the time and size complexities are effectively constant in the worst case.

5.4 Evaluation

The evaluation of Approximate AMTA analyzes how it behaves in terms of result accuracy, time performance and footprint. The the data-structure footprint affects both the memory usage and the network traffic. Reducing the data-structure footprint implies a slower growing window and less elements sent to the distributed data store or KVS.

This section is divided into two main parts. The first one analyzes the effect of different values in the user-configurable parameters of each A^{2}MTA method applied to its respective use case. Furthermore, the maximum size enforcement will be tested on the three scenarios: sum-like aggregation, max-like aggregation and hopping window. The second part is focused on the time performance impact of the three scenario-specific methods, compared to the state-of-the-art sliding window framework with best performance to our knowledge.

The data set used as a stream in the following experiments contains two years of a server's RAM memory usage monitoring in KB, where the available memory is 64GB. There is one memory usage update per second, which adds up to 62 208 000 updates. The operation performed in the experiments and its eviction policy will differ between the experiments due to the nature of the different scenarios considered, but they all share a maximum of 2 592 000 aggregated updates, which corresponds to a month worth of updates.

5.4.1 Implementation

All methods are implemented in Java 1.8 and executed in the A^{2}MTA operator in an *Apache Storm* based stream processing runtime called *rapids*. *rapids* processes all data units as objects with a shared class and several data dimensions, meaning that updates and partial results will be objects with multiple values rather than single scalar values. The purpose of running the algorithms in *rapids* rather than isolated is to show how they perform in a production environment.

A^{2}MTA will buffer up to 512 buckets from the data-structure before storing them to a distributed data store. Moreover, the data store used in the experiments is Couchbase [42]. Couchbase is a KVS based on memcached [50], with a distributed LRU cache in RAM. It prioritizes access in memory over disk for low-latency.

5.4.2 Environment

The experiments were run in a cluster with 2-way Xeon E5-2630 (Broadwell) v4 clocked at 2.20GHz nodes. Each one features 128GB of DDR4-2400 R ECC RAM. All nodes were interconnected using a non-blocking 10GbE switching fabric. Although an external NFS folder was mounted on the systems, it was not used as a backend for the experiments. Instead, all data was stored locally using four 7.2K rpm 2TB SATA HDDs per nodes, mounted as four independent volumes. The logic was executed in a single node, but Couchbase ran as a cluster in three extra nodes. Therefore, the contents of the data-structure were distributed between 4 nodes.

5.4.3 Experiment 1: Parameters

On this first experiment we tested different user-configurable parameter values in each bucket aggregation method, for the different considered aggregation scenarios.

	Footprint
Max error	Sum-like histogram
10^{-4}%	44.02%
10^{-3}%	6.591%
10^{-2}%	$8.335 \cdot 10^{-1}$%
10^{-1}%	$9.9 \cdot 10^{-2}$%
1%	$1.022 \cdot 10^{-2}$%
10%	$9.854 \cdot 10^{-4}$%

	Footprint
Block size	Max-like histogram
10	91.33%
10^2	91.1%
10^3	95.49%
10^4	60.97%
10^5	4.394%
10^6	19.88%

	Footprint
Parameter	Hop histogram
×	$5.229 \cdot 10^{-1}$%

TABLE 5.1: Scenario-specific bucket aggregation method's footprint relative to AMTA's

	Footprint		
Max size	Sum-like aggregation	Max-like aggregation	Hopping window
10^6	33.33%	42.98%	38.56%
10^5	3.846%	11.75%	4.583%
10^4	$3.845 \cdot 10^{-1}$%	2.189%	$4.679 \cdot 10^{-1}$%
10^3	$3.858 \cdot 10^{-2}$%	$3.368 \cdot 10^{-1}$%	$4.69 \cdot 10^{-2}$%
100	$3.864 \cdot 10^{-3}$%	$3.676 \cdot 10^{-2}$%	$4.7 \cdot 10^{-3}$%
10	$4.018 \cdot 10^{-4}$%	$3.298 \cdot 10^{-3}$%	$4.9 \cdot 10^{-4}$%

TABLE 5.2: Constrained A^{2}MTA footprint relative to AMTA's

Three scenarios were considered: sum-like aggregation, max-like aggregation and hopping window.

Sum-like aggregation computes the average of the monitored memory values in a static-size window of 2 592 000 updates. This scenario is designed to evaluate the sum-like histogram method's parameters. The sum-like histogram method is configured with a 95% confidence, and it controls two dimensions from the update: the sum value and the count of elements.

Max-like aggregation extracts the maximum value from a window of 2 592 000 update. A relevant consideration about this scenario is that the eviction policy keeps the current maximum update as the oldest one in the window. This is applied in all cases, including the accurate aggregation, as it improves the computation time and the memory footprint. Also, only when the max update changes, a new result is produced. This scenario is used to evaluate the max-like histogram method's parameters. The max-like histogram method is configured with a 95% confidence.

Hopping window will reach 2 592 000 and then perform a bulk eviction with a random number updates with mean 864 000 and standard deviation 100. The aggregation performed is also an average of the monitored memory values. The hop histogram method is also configured with a 95% confidence.

For each scenario we also evaluated the behavior of a constrained footprint window. The evaluated parameter values were incremented exponentially in order to

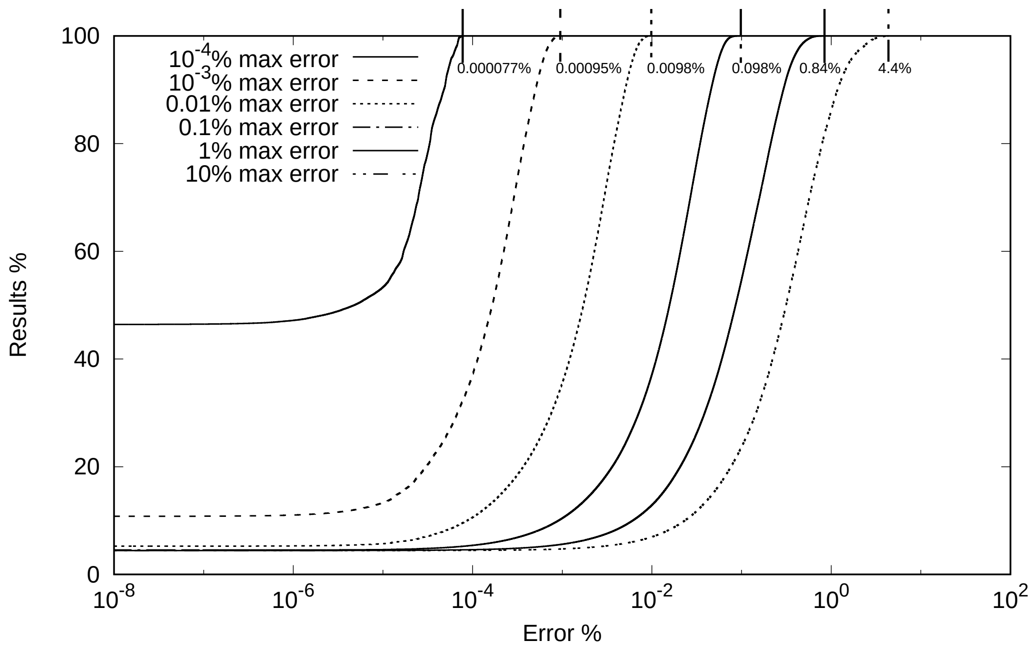


FIGURE 5.4: Effective error in a sum-like histogram

get a clear sense of its impact.

The impact of the parameters is evaluated in terms of the effective error produced on the aggregation when compared to the accurate aggregation, and the generated footprint using the accurate aggregation footprint as baseline. The error will be shown as cumulative distribution. The footprint is calculated as the number of new elements generated in the MTA data-structure. This number affects the memory usage, but also network traffic to the KVS; 1% smaller footprint means using 1% less of memory, but also 1% less of messages exchanged with the KVS.

Sum-like aggregation

In Figure 5.4 we can see the error cumulative distribution of the sum-like histogram, with a *max error* parameter from 10^{-4}% to 10%. The x-axis has a log-scale for readability. The most noticeable outcome from this figure is that, indeed, the max error defined by the user has not been exceed. Particularly, 10^{-4}% has a 45% of accurate results. However, in Table 5.1, where it shows the footprint of each parameter, we can see that the same *max error* has also big footprint (half of A^2MTA's) compared to the rest values. As it can be seen, footprint of the window grows linearly as the specified max error decreases.

For the constrained footprint sum-like window, in Figure 5.5 and Table 5.2, we

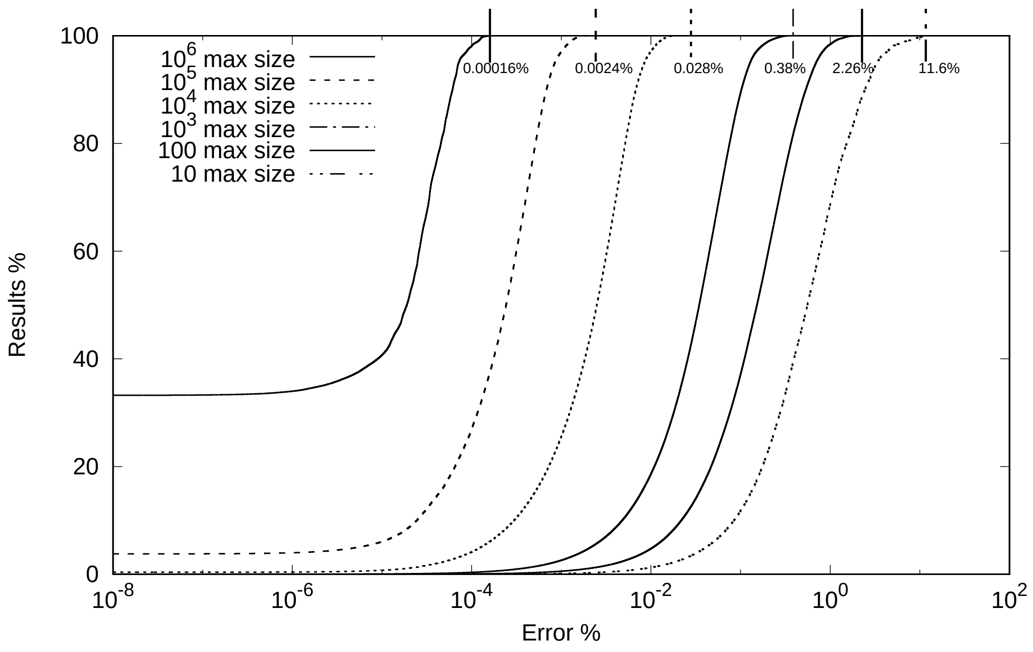


FIGURE 5.5: Effective error in a constrained sum-like window

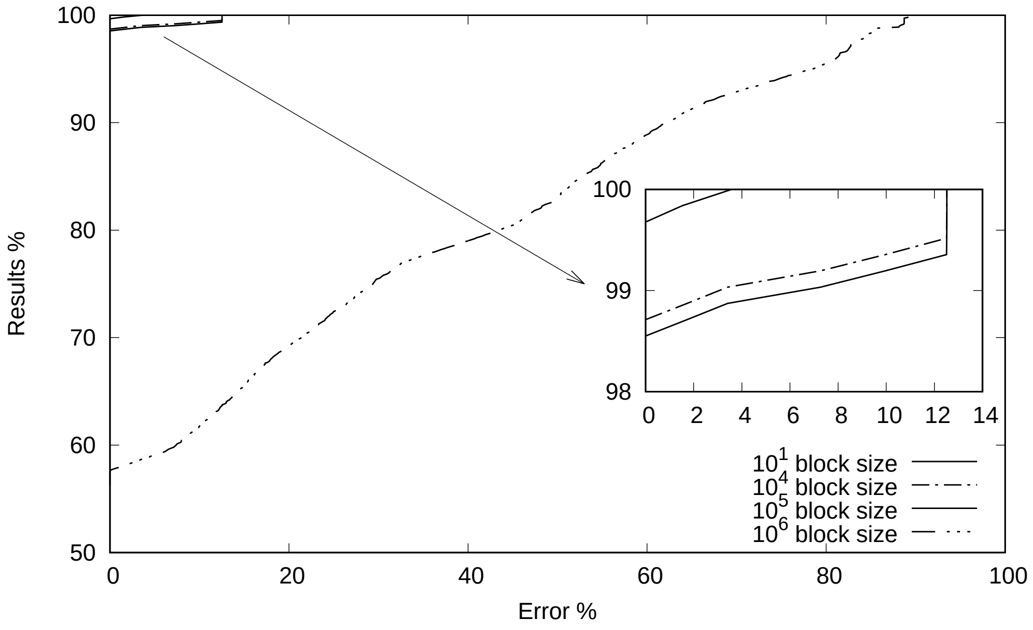


FIGURE 5.6: Effective error in a max-like histogram

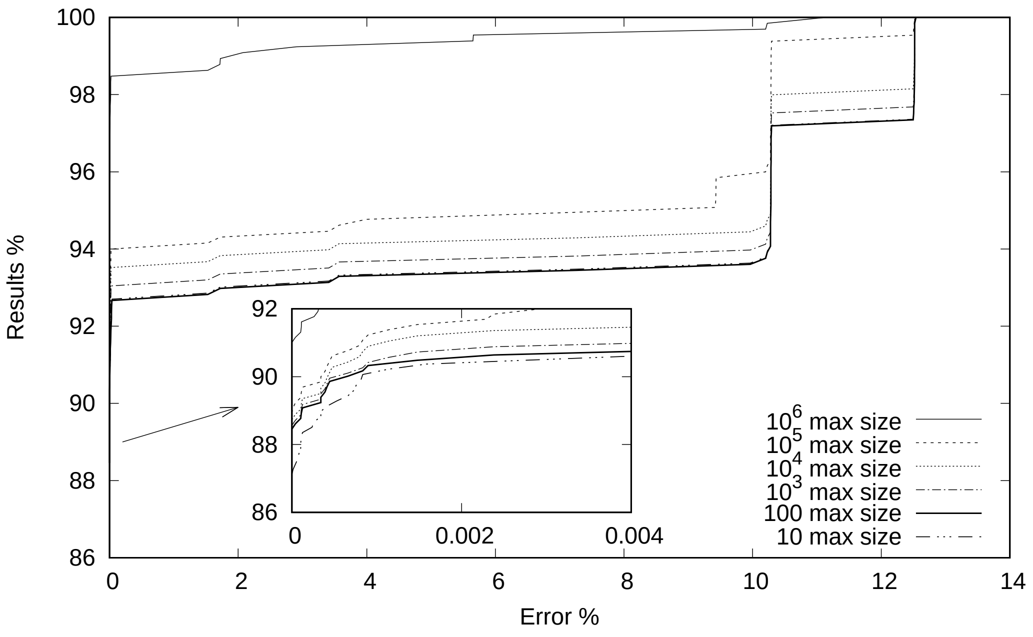


FIGURE 5.7: Effective error in a constrained max-like window

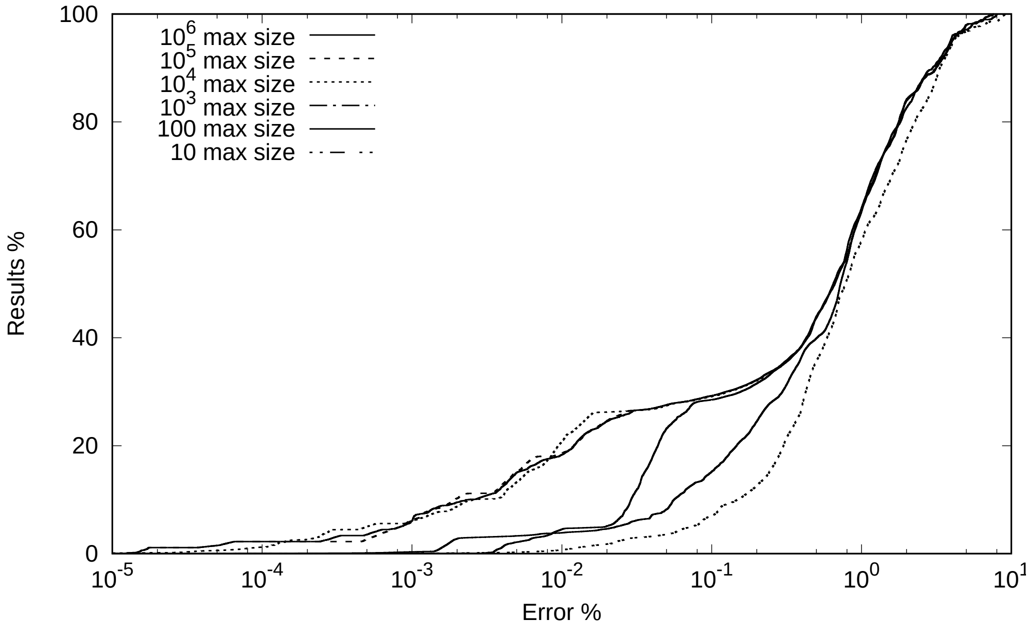


FIGURE 5.8: Effective error in a constrained hopping window

chose *max size* values that generate the same *count* aggregation error as in the sum-like histogram. Since the window have a static size in both cases, the accurate *count* value is always 2 592 000. In order to generate a maximum error of 10^{-4}%, we need to limit its size to 10^6 buckets. We can see that the constrained footprint shows a similar trend to the sum-like histogram, but with smaller footprints. However, as there is no error control (needed for the *sum* aggregation), in Figure 5.5 can be seen that the error is greater in the constrained footprint window in all cases. Also fewer results have accurate results, with parameters from 10^3 to 10 notably having none.

Max-like aggregation

Figure 5.6 shows the max-like histogram's error with different block sizes. As we expected, the a block size too big (10^6) makes a biased estimation of the GEV distribution and ends up generating results with elevated error, and low number of accurate results. The rest of the block sizes have more than 98% of accurate results, and maximum errors from none to 12%. In the figure, block sizes 10^2 and 10^3 can not be found because all their results are accurate. However, in Table 5.1 we can see that small block sizes generate very little footprint reduction. In contrast to sum-like histogram which choosing the parameter is a matter of priorities, in this case we have a clear most convenient parameter: 10^5, which covers a sample of 3 000 000 elements. It is the best managing the trend changes on the data values. It has a 4.583% of footprint, 98.55% of accurate results and a maximum error of 12.51%.

On the other hand, in Figure 5.7 we can see the behavior of a constrained max-like window, with different max number of buckets. We can see that even though the max errors are similar to the max-like histogram and the number of accurate results are acceptable (between 87.11% and 91%), with similar footprints as the optimal parameter in the max-like histogram experiment we get a lower number of accurate results. With a footprint of 11.75% (Table 5.2) we get 88.92% of accurate results, while with 2.189% footprint the number of accurate results is 88.73%. Therefore, the estimation of the GEV distribution has to predict extreme values has a clear effect on the error control and the footprint. It is worth noticing that the footprint is one order of magnitude higher than *Sum-like aggregation*, and that is due to the bulk evictions done when the maximum value is between the newer updates in the window.

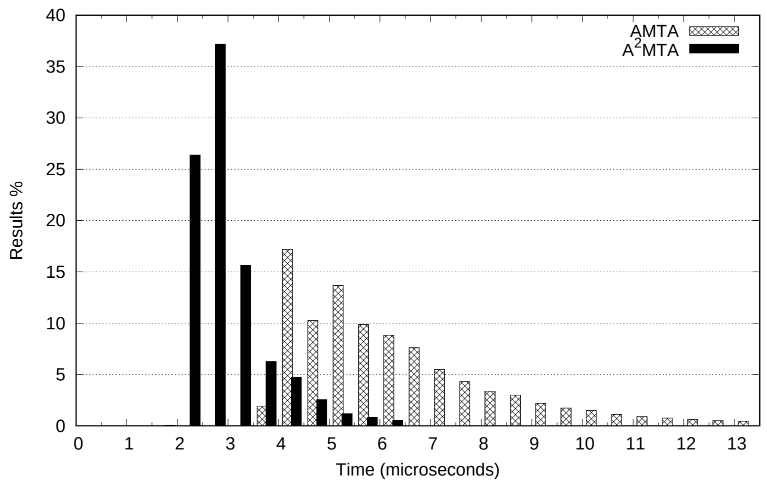


FIGURE 5.9: Sum-like histogram: 0.1% error

Hopping window aggregation

Hopping window histograms do not have any configurable parameters, therefore in Table 5.1 we can only see a single value from the performed experiment. The footprint is as small as the 0.5229% of the accurate window. Furthermore, in the experiment all results were 100% accurate. This clearly demonstrate that non-deterministic hopping windows can be greatly improved by using hopping window histograms. This method requires the window to have a clear hopping pattern in its bulk evictions in order to reduce the footprint. However, when this scenario happens, the footprint reduction is generally at no cost.

However, the constrained windows show a poor behavior in terms of error in Figure 5.8, with no accurate results in any of the tried parameter values. Table 5.2 shows that the footprint is higher than *Sum-like aggregation* due to the sudden size changes, but not as high as *Max-like aggregation*.

5.4.4 Experiment 2: Time performance

In this experiment we will focus on a single parameter value from the scenario-specific methods from Section 5.4.3 in order to compare their time performance with the same aggregation executed in AMTA. The goal is to determine if the additional computation required from the different bucket aggregation methods make the approximate computation more costly in terms of time performance, or if it generally saves computation time from the AMTA baseline.

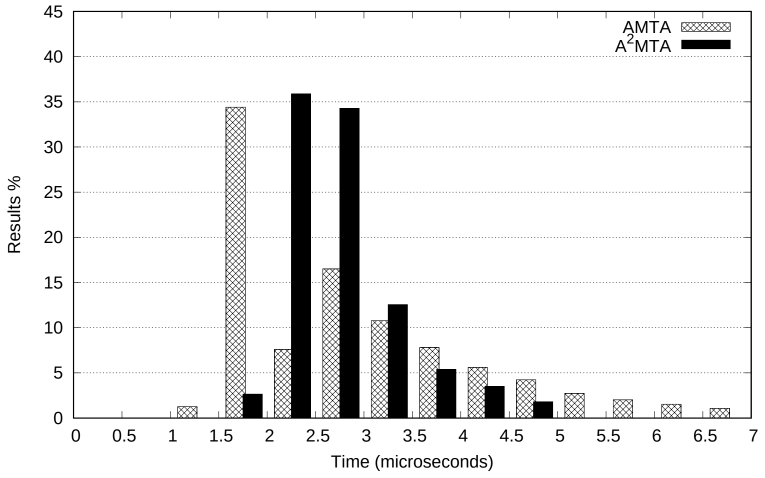


FIGURE 5.10: Max-like histogram: 10^5 block size

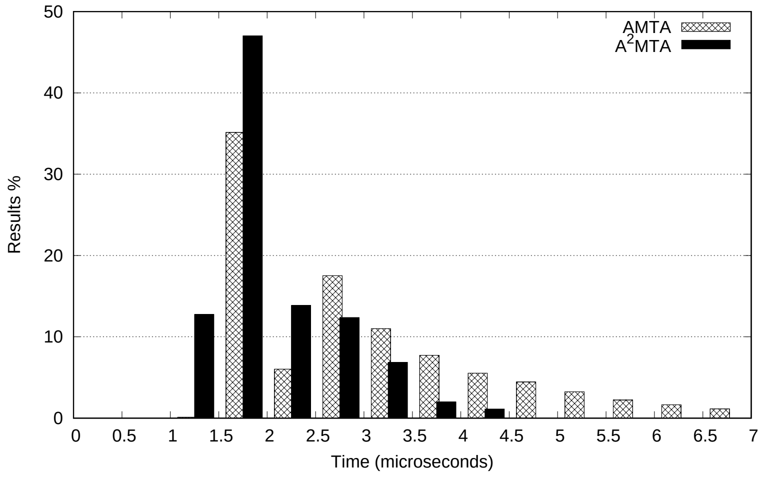


FIGURE 5.11: Hop histogram

From the three scenarios considered, the one that is more time consuming in AMTA is the sum-like aggregation. The reason is that for every insertion it also needs to perform a $O(1)$ eviction, while the two other scenarios perform $O(logn)$ bulk evictions after fewer insertions. For that reason, it can be seen in Figure 5.9 that there is a clear improvement with A^{2}MTA's sum-like histograms. The maximum error in this execution was 0.1$. The time interval that the aggregations take is narrower and lower than in AMTA. A^{2}MTA concentrates all the result times between 1.5μs and 6.5μs with almost 80% of the results between 1.5μs and 3.5μs, while AMTA is spreaded from 3.5μs to 13μs. There are two main reasons for these results. On the one hand, buckets reduce the number of evictions. If an static size window with 1 000 updates is divided by 10 buckets, then there will be 1 eviction for every 100 insertions. On the other hand, when an update is aggregated into a bucket, the insertion cost is always constant.

In Figure 5.10, for the max-like histogram with 10^5 block size, the A^{2}MTA still has a narrower interval of execution times in relation to AMTA, with 70% of the results having times between 2μs and 3μs. Whereas, the same amount of results can be found between 1.5μs and 3.5μs in AMTA. However, the improvement is not as noticeable as in the sum-like aggregation: the peak time in AMTA is already low, because it is getting benefit from bulk evictions that reduce the number of overall evictions.

Finally, Figure 5.11 shows the hopping window scenario. In this case, as the eviction size is estimated and used as the bucket size, then the bulk eviction cost is close to a single eviction (lower). Therefore, the time for A^{2}MTA is globally better both in terms of peak minimum time and interval size, compared to AMTA and to the other A^{2}MTA methods.

5.5 Related Work

Extensive work has been done in the last years on efficient sliding window aggregations and frameworks. Aside from Amortized MTA, which is the initial sliding window framework to which we applied the approximate computing paradigm, the literature propose FIFO data-structures and incremental operations that keep a logarithmic or constant complexity in stream processing aggregation algorithms.

Tangwongsan et al. proposed two sliding window aggregation frameworks called *Reactive Aggregator* (RA) [92] and *Sliding-Window Aggregation* (SWAG) [90]. Being SWAG an important improvement from RA, both approaches follow Boykin et al. [34] method of using associative operations as programmatic aggregators interface. RA has $O(\log n)$ time complexity in all its operations with constant-sized sliding windows. RA's sliding window FIFO structure is a flat fixed-sized binary and complete tree called FlatFAT. All the leaves are the raw updates to be aggregated, the root node is the result and the intermediate nodes are partial computations. Every update insertion and deletion propagates the aggregation changes from the leaf to the root. Other work in the literature [76, 100, 22, 29] use tree-like structures in order to keep partial computations in the same way, making use of binary associative operators. They all have a worst-case $O(\log n)$ for all its atomic operations and a complexity $O(n)$ for windows with bulk evictions. On the other hand, SWAG runs in a constant $O(1)$ time for each one of its atomic operations, bulk eviction not included among them. The algorithm is based on a data-structure with two stacks, one in charge of managing the insertions and the other the single evictions. The lack of an efficient bulk eviction operation for these frameworks' FIFO data-structures, make them unsuitable to distribute the window contents using, for example, a KVS.

Approximate computing for data analytics has been a wide area of study for decades, mainly for aggregations in relational databases, with techniques such as sampling [54, 69, 6, 51], histograms [43, 31, 30, 72], stream sketches [39, 41, 10] or online aggregation [55]. Goiri et al. [54] proposed an approximate computing set of mechanisms for batch processing for Hadoop, called ApproxHadoop. Like A^{2}MTA, ApproxHadoop distinguishes between sum-like and extreme value aggregations. However, the approach is to perform multi-stage sampling, instead of histograms. Regarding sliding window approximate aggregation, Datar et al. [43] propose an exponential histogram count aggregation, with a $O(\frac{1}{\epsilon}\log N)$ overhead and a $1+\epsilon$ loss in accuracy concerning number of elements. This method can be easily applied to other aggregations, but the error is always measured in terms of number of aggregated elements. Bifet et al. contributed *ADWIN*[31] and *K-ADWIN* [30] frameworks, based on Datar's exponential histograms. The two sliding window frameworks aggregate data that has a similar tendency. When two subwindows have very different average values, the oldest one is removed. Having a defined eviction policy based on the difference between buckets containing similar values, the error is kept very low while the time complexity is constant and the window overhead is given by exponential histograms. However, while the aggregation is user-programmable, by design it does not support any kind of programmable eviction policy. Arasu & Manku [21] describe a variety of algorithms to calculate approximate count and quantile sliding windows. Krishnan et al. present IncApprox [69], a general purpose incremental approximate computing framework with error boundaries. Having a logarithmic time complexity, instead of building a histogram, it benefits form an online stratified sampling algorithm guided by an error prediction.

5.6 Conclusions

In this chapter we have introduced the Approximate AMTA (A^{2}MTA) framework, a novel general sliding window aggregation framework that combines a constant-time FIFO data-structure with the resource reduction benefits from the approximate computing paradigm. While a completely user-programable sliding window is bound to have a non-deterministic resource consumption, the leverage of approximate computing techniques delimits it and contributes with better performance. Furthermore, the accuracy of the results can be configured with some confidence levels.

We described A^{2}MTA as a framework with a set of the different approximate computing methods for the different kinds of sliding windows. On the one hand we defined *Sum-like histograms* and *Max-like histograms*, which are applied to different types of aggregations and therefore can not be combined. The first will keep the aggregation error bellow the boundaries set by the user, while the second aims to produce accurate results with a confidence level. On the other hand, *Hop histograms* focuses on the detection of a usual type of eviction policy that corresponds to hopping windows and also aims for accurate results, as long as the *Constrained footprint* *enforcement* prioritizes the window memory footprint over the error. These last two methods can be combined with all the rest.

A thorough evaluation has been performed to give evidence of the impact of the approximate aggregation techniques. In addition, we evaluated the controlled degradation of the aggregation results, confirming that it behaves accordingly to the parameters given by the user. The result show that even having as a baseline the most efficient sliding window aggregation framework to our knowledge, the computation time has been improved in all the tested cases. Furthermore, the impact on the window footprint, which affects both memory and bandwidth resources, makes A^{2}MTA very engaging for adverse environments.

Chapter 6

Conclusions & Future Work

In the course of this Doctoral Thesis, stream processing has been a research and development field that gained a lot of momentum. Many stream processing platforms, either open-source or commercial, appeared in the last few years and already became archetypes in Big Data architectures. Also, there has been a lot of discussion on the role stream processing have to play in Big Data and IoT analytics alongside batch computation. Furthermore, there have been a lot of parallel efforts to provide programming models with specialized operators that match the performance expectations that real-time computation require. Despite all the aligned research happening in this field, this Doctoral Thesis produced a series of contributions to the topic that, to our knowledge, can still be considered state of the art in terms of stream processing programming model and efficient aggregation algorithms. Nonetheless, we found multiple potential contributions close to the work presented here that are worth exploring in future work.

The following are summarized conclusions from the contributions of this Doctoral Thesis:

- **Dynamic Pipelining Programming Model**: A singularity from stream processing analytics is that its results are new streams. Sharing results between tenants with batch processing analytics is not as challenging as in stream processing, since it will generally entail a closed number of results between big intervals of time. We extract summaries from a Big Data scale of information. However, stream processing analytics produce an infinite and unbound amount of data with high frequency.

While cloud stream processing services were still in very early stages during the work on the initial contribution of this Doctoral Thesis, we proposed a stream processing programming model that enables sharing of stream data analytics through a subscription model. We also showed its viability; it has good scalability in terms of the computation topology graph degree, but it is susceptible to bottlenecks in the longitude of its pipelines. Therefore, it is imperative to control the computation latency of its operators.

- **Composite Streams**: One of the usual operations for stream processing of IoT data is the composition of streams. With a programming model based on stream subscriptions, composition is used to enrich one stream information with other related streams. We proposed a set of rules to keep the computation lock-free and time-consistent, while producing the maximum number of results.

- **Constant-time Sliding Window Framework**: Aggregation functions are the most relevant operations in data analytics, since they extract single results from multiple values. They are also the most complex operations and can easily become bottlenecks in the computation. Since streams are infinite and unbounded sequences of data, sliding windows are a recurrent type of aggregation function, because it sets boundaries on the amount of data that is going to be aggregated.

  We proposed a completely programable framework that allows a user to deploy efficient sliding windows. We demonstrated that its time-complexity is amortized $O(1)$ with low latencies for aggregations like average or sum. Aside from giving freedom to program the aggregation by only requiring it to be a *monoid*, it is also viable to program the policy that defines the size of the window. It allows the user to define the size of the window not only in terms of static size, but using other dimensions like time or maximum aggregated value.

- **Efficient Window Bulk Evictions**: Dynamically sized sliding windows have the particularity that they have multiple elements evicted at once from time to time. That has been called *bulk evictions* in this work. In order to avoid that this situation turns into a pipeline computation bottleneck, we demonstrated that this operation can be performed in $O(\log n)$ which is amortized to $O(1)$ in all the insertions in the window.

- **Distributed Scalable Window**: The aggregation of vast window of updates, requires these updates to be kept in order to aggregate and evict them. Since the algorithm proposed is lightweight to not have any gain from distributing its execution. However, keeping all the elements from the computation in local memory can easily become a resources problem. The proposed sliding window algorithms has mechanisms to distribute and replicated the elements like it would be done for a Big Data batch computation. We demonstrated that the effect of not having $O(n)$ elements from the data structure not locally, but distributed, has as low effect on the overall latency and in many cases is compensated by the possibility of having multiple efficient sliding windows in the same node.

- **Approximate Computing Window Aggregation**: Stream processing for the IoT is being tightly bound to Edge and Fog computing, for good reasons. Stream processing in the IoT is meant to be used when low latencies and fast reaction to events are required. In order to reduce that latency between an event happening and its reaction, all the related computations need to be performed in-situ. This usually translates to execution environment with low or unreliable resources, such as energy, memory, CPU or network connectivity.

  Running the proposed window from the previous contribution in a resource-scarce scenario can be problematic, as it either requires memory to store elements locally or reliable network bandwidth to send them to a distributed store. We demonstrated that it is possible to apply the approximate computing paradigm to the our previous efforts on sliding windows in order to reduce the network connectivity — distributed aggregation store — while preserving a low local memory usage, and improve general computation latencies. The properties from the original algorithm are preserved, and only the accuracy of the results is affected. However, we also demonstrated that the error can be predicted and contained, and therefore it is user configurable.

While working on each contribution, multiple research paths opened with potential to become relevant contributions. Due to limited time and strategic convenience, we choose and developed the contributions presented in this document. However, the following summary works as a record of the other research paths that were considered for future work and would still be considered novel contributions:

- **Dynamic Stream Subscriptions**: By having rich stream descriptions in an indexed repository, streams can be easily searchable and queried. In a system as the one presented in this work, we search for streams with a compatible structure with the computations that will follow. Therefore, usual search parameters are the type, unit and metrics of each channel. Other descriptive information from the stream is also relevant, such as the location where the stream is being generated. That location can be in terms of latitude and longitude, or being generated from an entity like a city or a street. The results of such queries are streams to which we want to subscribe our analytics.

  We propose to expand the stream processing programming model to be able to make the *union* operation subscribe to the results of a stream query, instead of an static list of streams. Therefore, the user would describe the kind of stream it is needed instead of choosing some specific streams that might fail in the future. When a query changes its results because a stream was created, updated or replaced, the analytics subscribed to that query would also change its inputs. For example, if we want to compute analytics on the temperature sensors in district, a dynamic stream subscription would characterize the kind of sensor needed: with one number-type channel, using Celsius expressing outdoor temperature and originated in the specified district. Whenever one of these sensors gets replaced, they will be automatically bound to the DPP.

- **Composer Aggregator**: Consider a stream that is the union of a vast amount of sensors and that stream is partitioned by the origin sensor. We might want to perform an aggregation using the last update from each sensor, generating a composite stream from multiple compatible streams. The Composer Aggregator would be an aggregator of the last update of each partition, producing a new stream update for every new update received. Although this aggregator shares with the *compose* operation that they both combine multiple inputs to produce a single one, the aggregator does not combine a closed number of incompatible streams. The number of partitions can change dynamically, i.e. because this operator is combined with a Dynamic Stream Subscription and streams were added or removed. For example, aggregate the mean temperature from all the sensors in a specific area, each sensor producing a partition of the same streams.

However, partitions might become inactive for reasons such as a sensor running out of battery or low-frequency streams, and aggregations might be using expired values. This aggregator needs to consider user-defined policies to remove partitions from the aggregation, such as the WSPs for the Sliding Windows presented in this work.

Following a strategy similar to the one followed in this work for the sliding windows, we estimate that the computation time would be $O(\log n)$ while the memory usage would be $O(n)$, n being the number of aggregated streams.

- **Provenance and Security for Multi-tenancy**: In order to have a controlled multi-tenant environment in which different tenants can control how to share their streams, we require a notion of update provenance. P. Buneman et al. [36] describe data provenance as:

  > Data provenance — sometimes called “lineage” or “pedigree” — is the description of the origins of a piece of data and the process by which it arrived in a database. The field of molecular biology, for example, supports some 500 public databases, but only a handful of these are “source” data in the sense that they receive experimental data. All the other databases are in some sense views either of the source data or of other views. In fact, some of them are views of each other, which sounds nonsensical until one understands that the individual databases are not simply computed by queries, but also have added value in the form of corrections and annotations by experts (they are “curated”). A serious problem confronting the user

> of one of these databases is knowing the provenance of a given piece of data. This information is essential to anyone interested in the accuracy and timeliness of the data.

In a system where the data generated can have multiple origins, with multiple intermediate operations and multiple tenants providing these operations, data provenance have many applications.

Collecting data provenance with different levels of detail can help to monitor the quality of a stream in terms of origins, frequency or latency. Furthermore, it would allow tenants in the system to apply multiple policies to their streams such as white/black lists of input stream tenants or white/black lists of subscriber tenants. Such policies could facilitate the creation of a stream analytics marketplace between tenants.

However, attaching provenance data to every update is challenging. An update can have a long lineage that needs to be stored and processed in order to apply security permissions or extract monitoring information. All this needs to be performed in a way that it does not affect the performance required to run in a real-time environment.

Bibliography

[1] Daniel J Abadi et al. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 12.2 (2003), pp. 120–139.

[2] Daniel J Abadi et al. "The Design of the Borealis Stream Processing Engine." In: *CIDR*. Vol. 5. 2005, pp. 277–289.

[3] Tyler Akidau et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1033–1044. ISSN: 2150-8097. DOI: 10.14778/2536222.2536229. URL: http://dx.doi.org/10.14778/2536222.2536229.

[4] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803.

[5] *Akka*. Accessed in: 10-October-2018. 2009. URL: http://akka.io/.

[6] Mohammed Al-Kateb and Byung Suk Lee. "Stratified reservoir sampling over heterogeneous data streams". In: *International Conference on Scientific and Statistical Database Management*. Springer. 2010, pp. 621–639.

[7] *Algebird*. Accessed in: 4-July-2015. 2012. URL: https://github.com/twitter/algebird.

[8] Mohamed Ali et al. "The extensibility framework in Microsoft StreamInsight". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 1242–1253.

[9] *Amazon Kinesis*. https://aws.amazon.com/kinesis/. Accessed: May 2017. 2017.

[10] Henrique CM Andrade, Buğra Gedik, and Deepak S Turaga. *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[11] Aleksandar Antonic et al. "A Mobile Crowdsensing Ecosystem Enabled by a Cloud-based Publish/Subscribe Middleware". In: *The 2nd International Conference on Future Internet of Things and Cloud (FiCloud-2014)*. 2014.

[12] *Apache Apollo official website*. URL: http://activemq.apache.org/apollo.

[13] *Apache Flink official website*. URL: http://flink.apache.org.

[14] *Apache Hadoop*. Accessed in: 4-July-2015. 2011. URL: http://hadoop.apache.org/.

[15] *Apache Kafka*. Accessed in: 4-July-2018. URL: http://kafka.apache.org/.

[16] *Apache Samza*. http://samza.apache.org. Accessed: May 2017. 2017.

[17] *Apache Solr*. Accessed in: 4-July-2015. URL: http://lucene.apache.org/solr/.

[18] *Apache Spark Streaming*. URL: https://spark.apache.org/streaming/.

[19] *Apache Storm*. Accessed in: 10-October-2018. 2013. URL: http://storm.apache.org/.

[20] *Apache Thrift*. Accessed in: 10-October-2018. URL: http://thrift.apache.com.

[21] Arvind Arasu and Gurmeet Singh Manku. "Approximate counts and quantiles over sliding windows". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2004, pp. 286–296.

[22] Arvind Arasu and Jennifer Widom. "Resource sharing in continuous sliding-window aggregates". In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment. 2004, pp. 336–347.

[23] Joe Armstrong et al. "Concurrent programming in ERLANG". In: (1993).

[24] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. “Load management and high availability in the Medusa distributed stream processing system”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 929–930.

[25] Roger S Barga et al. “Consistent streaming through time: A vision for event stream processing”. In: *arXiv preprint cs/0612115* (2006).

[26] Michel Bauderon et al. “Netquest: An Abstract Model for Pervasive Applications”. In: *Proceedings of the 7th Int’l Conf. on Pervasive Computing (Pervasive 2009)*. 2009, pp. 467–481.

[27] Michel Bauderon et al. “Programming iMote Networks Made Easy”. In: *The Fourth International Conference on Sensor Technologies and Applications*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 539–544. ISBN: 978-1-4244-7538-4.

[28] *Beating the CAP Theorem Checklist*. Accessed in: 4-July-2015. URL: http://ferd.ca/beating-the-cap-theorem-checklist.html.

[29] Pramod Bhatotia et al. “Slider: Incremental sliding window analytics”. In: *Proceedings of the 15th International Middleware Conference*. ACM. 2014, pp. 61–72.

[30] Albert Bifet and Ricard Gavalda. “Kalman filters and adaptive windows for learning in data streams”. In: *International Conference on Discovery Science*. Springer. 2006, pp. 29–40.

[31] Albert Bifet and Ricard Gavalda. “Learning from time-changing data with adaptive windowing”. In: *Proceedings of the 2007 SIAM international conference on data mining*. SIAM. 2007, pp. 443–448.

[32] Flavio Bonomi et al. “Fog computing and its role in the internet of things”. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

[33] Alessio Botta et al. “On the Integration of Cloud Computing and Internet of Things”. In: *The 2nd International Conference on Future Internet of Things and Cloud (FiCloud-2014)*. 2014.

[34] Oscar Boykin et al. "Summingbird: A framework for integrating batch and online mapreduce computations". In: *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1441–1451.

[35] HW Van den Brink et al. "Estimating return periods of extreme events from ECMWF seasonal forecast ensembles". In: *International Journal of Climatology: A Journal of the Royal Meteorological Society* 25.10 (2005), pp. 1345–1354.

[36] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. "Why and where: A characterization of data provenance". In: *International conference on database theory*. Springer. 2001, pp. 316–330.

[37] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[38] Badrish Chandramouli et al. "Trill: A high-performance incremental query processor for diverse analytics". In: *Proceedings of the VLDB Endowment* 8.4 (2014), pp. 401–412.

[39] Kenneth L Clarkson and David P Woodruff. "Numerical linear algebra in the streaming model". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM. 2009, pp. 205–214.

[40] Stuart Coles et al. *An introduction to statistical modeling of extreme values*. Vol. 208. Springer, 2001.

[41] Graham Cormode et al. "Synopses for massive data: Samples, histograms, wavelets, sketches". In: *Foundations and Trends® in Databases* 4.1–3 (2011), pp. 1–294.

[42] *Couchbase official website*. URL: http://couchbase.com.

[43] Mayur Datar et al. "Maintaining stream statistics over sliding windows". In: *SIAM journal on computing* 31.6 (2002), pp. 1794–1813.

[44] *DeviceHive official website*. URL: http://www.devicehive.com/.

[45] *Devicehub official website*. URL: http://devicehub.net.

[46] Jean Diebolt et al. "Improving probability-weighted moment methods for the generalized extreme value distribution". In: *REVSTAT-Statistical Journal* 6.1 (2008), pp. 33–50.

[47] *ElasticSearch official website*. URL: http://elasticsearch.org.

[48] *evrythng official website*. URL: evrythng.com.

[49] Ronald Aylmer Fisher and Leonard Henry Caleb Tippett. "Limiting forms of the frequency distribution of the largest or smallest member of a sample". In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 24. 2. Cambridge University Press. 1928, pp. 180–190.

[50] Brad Fitzpatrick. "Distributed caching with memcached". In: *Linux journal* 2004.124 (2004), p. 5.

[51] Minos N Garofalakis and Phillip B Gibbons. "Approximate Query Processing: Taming the TeraBytes." In: *VLDB*. 2001, pp. 343–352.

[52] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[53] Manfred Gilli et al. "An application of extreme value theory for measuring financial risk". In: *Computational Economics* 27.2-3 (2006), pp. 207–228.

[54] Inigo Goiri et al. "Approxhadoop: Bringing approximations to mapreduce frameworks". In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 1. ACM. 2015, pp. 383–397.

[55] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. "Online aggregation". In: *Acm Sigmod Record*. Vol. 26. 2. ACM. 1997, pp. 171–182.

[56] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[57] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. "Sliding-Window Aggregation Algorithms: Tutorial". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, pp. 11–14.

[58] Martin Hirzel et al. “IBM streams processing language: Analyzing big data in motion”. In: *IBM Journal of Research and Development* 57.3/4 (2013), pp. 7–1.

[59] Jonathan RM Hosking, James R Wallis, and Eric F Wood. “Estimation of the generalized extreme-value distribution by the method of probability-weighted moments”. In: *Technometrics* 27.3 (1985), pp. 251–261.

[60] *IoT Toolkit official website*. URL: `iot-toolkit.com`.

[61] *Jackson official website*. URL: `http://jackson.codehaus.org/`.

[62] *Jetty official website*. URL: `http://www.eclipse.org/jetty/`.

[63] Rudolph Emil Kalman. “A new approach to linear filtering and prediction problems”. In: *Journal of basic Engineering* 82.1 (1960), pp. 35–45.

[64] Scott Klein. “Azure Stream Analytics”. In: *IoT Solutions in Microsoft's Azure IoT Suite*. Springer, 2017, pp. 71–84.

[65] Martin Kleppmann and Jay Kreps. “Kafka, Samza and the Unix Philosophy of Distributed Data”. In: ().

[66] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[67] Jay Kreps. “Questioning the Lambda Architecture”. In: *Online article, July* (2014).

[68] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. “On-the-fly sharing for streamed aggregation”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 623–634.

[69] Dhanya R Krishnan et al. “Incapprox: A data analytics system for incremental approximate computing”. In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pp. 1133–1144.

[70] Sanjeev Kulkarni et al. “Twitter Heron: Stream Processing at Scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 239–250.

ISBN: 978-1-4503-2758-9. DOI: `10.1145/2723372.2742788`. URL: `http://doi.acm.org/10.1145/2723372.2742788`.

[71] Richard Kuntschke et al. "Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures". In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 1259–1262.

[72] Jin Li et al. "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams". In: *ACM SIGMOD Record* 34.1 (2005), pp. 39–44.

[73] Jimmy Lin. "Monoidify! monoids as a design principle for efficient mapreduce algorithms". In: *arXiv preprint arXiv:1304.7544* (2013).

[74] Nathan Marz. "How to beat the CAP theorem". In: *Thoughts from the Red Planet* (2011).

[75] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.

[76] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. "Scalable algorithms for large temporal aggregation". In: *Data Engineering, 2000. Proceedings. 16th International Conference on*. IEEE. 2000, pp. 145–154.

[77] *MQTT official website*. URL: `http://mqtt.org`.

[78] Stefan Nastic et al. "Provisioning Software-defined IoT Cloud Systems". In: *The 2nd International Conference on Future Internet of Things and Cloud (FiCloud-2014)*. 2014.

[79] Philippe Naveau et al. "Modelling pairwise dependence of maxima in space". In: *Biometrika* 96.1 (2009), pp. 1–17.

[80] *Netty*. Accessed in: 10-October-2018. URL: `http://netty.io/`.

[81] Leonardo Neumeyer et al. "S4: Distributed stream computing platform". In: *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE. 2010, pp. 170–177.

[82] *Nimbits official website*. URL: `http://www.nimbits.com`.

[83] *OpenRemote official website*. URL: `http://www.openremote.com`.

[84] Carlos Pedrinaci et al. "iServe: a linked services publishing platform". In: *CEUR workshop proceedings*. Vol. 596. 2010.

[85] Yongrui Qin et al. "When Things Matter: A Data-Centric View of the Internet of Things". In: *CoRR* abs/1407.2704 (2014). URL: http://arxiv.org/abs/1407.2704.

[86] *SiteWhere official website*. URL: http://www.sitewhere.org.

[87] *Stomp repository*. URL: http://stomp.github.io.

[88] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing". In: *ACM SIGMOD Record* 34.4 (2005), pp. 42–47.

[89] Roshan Sumbaly et al. "Serving large-scale batch computed data with project voldemort". In: *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association. 2012, pp. 18–18.

[90] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. "Constant-Time Sliding Window Aggregation". In: *IBM, IBM Research Report RC25574 (WAT1511-030)* (2015).

[91] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. "Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, pp. 66–77.

[92] Kanat Tangwongsan et al. "General incremental sliding-window aggregation". In: *Proceedings of the VLDB Endowment* 8.7 (2015), pp. 702–713.

[93] *The Apache Cassandra Project*. Accessed in: 4-July-2015. 2008. URL: http://cassandra.apache.org/.

[94] *The WebSocket API*. URL: http://dev.w3.org/html5/websockets.

[95] *ThingSpeak official website*. URL: https://thingspeak.com/.

[96] Ankit Toshniwal et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[97] AG Van der Valk et al. "The biomass of an Indian monsoonal wetland before and after being overgrown with Paspalum distichum L." In: *Vegetatio* 109.1 (1993), pp. 81–90.

[98] Guozhang Wang et al. "Building a replicated logging system with Apache Kafka". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1654–1655.

[99] *Xively official website*. URL: xively.com.

[100] Jun Yang and Jennifer Widom. "Incremental computation and maintenance of temporal aggregates". In: *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE. 2001, pp. 51–60.