Maximizing Determinism in Stream Processing Under Latency Constraints

Nikos Zacheilas, Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business
Athens, Greece
zacheilas,vana@aueb.gr

Yiannis Nikolakopoulos, Vincenzo Gulisano,
Marina Papatriantafilou, Philippas Tsigas
Chalmers University of Technology
Gothenburg, Sweden
ioaniko,vinmas,ptrianta,philippas.tsigas@chalmers.se

ABSTRACT

The problem of coping with the demands of determinism and meeting latency constraints is challenging in distributed data stream processing systems that have to process high volume data streams that arrive from different unsynchronized input sources. In order to deterministically process the streaming data, they need mechanisms that synchronize the order in which tuples are processed by the operators. On the other hand, achieving real-time response in such a system requires careful tradeoff between determinism and low latency performance. We build on a recently proposed approach to handle data exchange and synchronization in stream processing, namely ScaleGate, which comes with guarantees for determinism and an efficient lock-free implementation, enabling high scalability. Considering the challenge and trade-offs implied by real-time constraints, we propose a system which comprises (a) a novel data structure called Slack-ScaleGate (SSG), along with its algorithmic implementation; SSG enables us to guarantee the deterministic processing of tuples as long as they are able to meet their latency constraints, and (b) a method to dynamically tune the maximum amount of time that a tuple can wait in the SSG data-structure, relaxing the determinism guarantees when needed, in order to satisfy the latency constraints. Our detailed experimental evaluation using a traffic monitoring application deployed in the city of Dublin, illustrates the working and benefits of our approach.

CCS CONCEPTS

• **Information systems → Stream management**;

KEYWORDS

Complex Event Processing, Stream Processing, Deterministic Processing

1 INTRODUCTION

Distributed stream processing systems (DSPS) such as Apache's Storm [3] and Spark Streaming [2] provide state-of-the-art systems for processing potentially unbounded sequences of tuples with low latency and high velocity. Continuous processing of large volumes of data streams presents an important challenge in a wide range of big data application domains ranging from traffic monitoring [24] to financial data processing [6]. These systems invoke *continuous queries* in which incoming tuples are processed in *sliding windows* and executed on multiple computing nodes, in order to detect events of interest in real-time.

The problem of providing deterministic operation for stream processing applications has been studied recently. A stream operator's implementation is considered *deterministic*, if, given the same sequences of input tuples, the same sequence of output tuples will be produced, independently of the tuples' inter-arrival time. In [8] the *ScaleGate* data structure guarantees that data arriving from different input streaming sources are processed in the correct order by the join operator, while it has also been used for scalable streaming aggregates [5] and analytics [9]. *ScaleGate stalls* the processing of each tuple until it is certain that no other tuple with smaller timestamp will arrive. A similar buffer-based technique has been proposed in [16] where they exploit the use of the *K-slack* data structure for keeping incoming tuples, and process them only when K time units have elapsed since the tuple has been inserted in the data structure.

Existing approaches have two important limitations: First, both *ScaleGate* and *K-slack* focus on the performance of a single operator and do not examine the problem in the context of complex graphs where the application consists of multiple operators. A second important aspect that has not been considered by these techniques is that there are often *real-time* response requirements to be satisfied which conflict with the fact that deterministic operation requires that tuples are stalled often for large amounts of time until they are ready for processing. Many applications, such as traffic and environment monitoring applications have low response time requirements and can tolerate approximate results [12]. For example, in a traffic monitoring application the users want to know the current traffic conditions as fast as possible and can tolerate some inaccuracy in the results (*i.e.,* missing some traffic updates). Nevertheless, it is still desirable to minimize the amount of missed events.

A few recent works [12, 13] have been proposed that aim at providing determinism guarantees and meeting application's end-to-end response time requirements. However, they are operator specific (*i.e.,* in [12] they focus on join operators while in [13] they examine the problem for aggregate operators) and do not provide a generic approach which would work with any operator type and complex application graphs. Furthermore, in [11] the authors examine the problem of placing *K-slack* buffers in an application graph comprising count and join operators in order to share tuples between them and minimize the memory usage. However, they do

DEBS '17, Barcelona, Spain

DOI: 10.1145/3093742.3093921

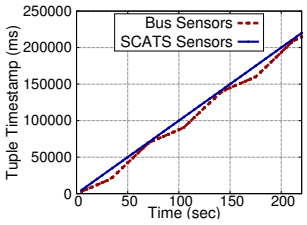


Figure 1: Reported timestamps from the two sources.

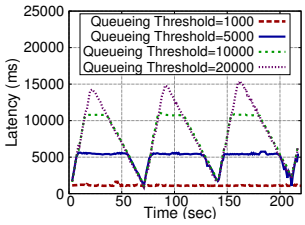


Figure 2: Queueing Threshold's impact on the join operator's latency.

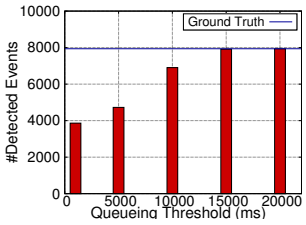


Figure 3: Queueing Threshold's impact on the number of detected events.

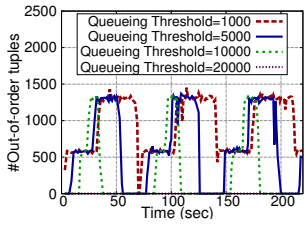


Figure 4: Queueing Threshold's impact on out-of-order tuples.

not consider the end-to-end response time requirements and are limited to two operator types (*i.e.,* join and aggregate).

In this work we study the problem of *maximizing the deterministic operation* for distributed stream processing applications under latency constraints. We consider applications with real-time response requirements (*i.e.,* traffic monitoring systems, financial applications). We define as latency the end-to-end execution time (including queueing delays) of tuples that are processed by a stream processing application graph comprising multiple stream processing operators. The problem is challenging as we have to minimize the impact of out-of-order tuples on the accuracy of the operators' results, and also satisfy the application's response time requirements. To satisfy these requirements we have to bound the amount of time (*i.e.,* provide an upper queueing threshold) that a tuple can be stalled for guaranteeing deterministic processing. When a tuple has resided on the operator's input buffer more than this threshold then it is read and processed by the operator. Using a small queueing threshold improves the latency as tuples will be stalled for a short time period. However, this can lead to processing many tuples that violate the determinism constraints.

In Figures 1, 2, 3 and 4 we illustrate with an example how communication delays can affect the latency of a join operator and their effect on determinism in terms of our ability to accurately detect events. This example is from a traffic monitoring system deployed in the Dublin Smart City [24] where the City Operator performs a join operation (in a 5 second sliding window) on streaming data from two separate input traffic data sources, Bus trajectory data from buses moving around the city and SCATS measurements of traffic flow in junctions, to detect areas of high traffic congestion across the city. Figure 1 shows the timestamps of the tuples of the two input sources over time where we observe that the Bus input source exhibits periods of higher communication delay (*e.g.,* in the first 25 seconds the bus data tuples arrive with a delay of 7 seconds). This increased delay can lead to out-of-order tuples unless we stall incoming tuples on the operator's input buffer using the queueing threshold parameter. In Figure 2 we depict how the amount of time that tuples will wait in the operator's buffer before they are processed (*i.e.,* in order to guarantee determinism), can affect the operator's response time (*i.e.,* latency). During periods of high delay, a large queueing threshold can lead to an increase on the operator's latency. In contrast, when we utilize a small queuing threshold (*i.e.,* 1,000 ms) the latency is kept steady.

However, this can affect the total amount of detected events (*i.e.,* Figure 3) and the number of out-of-order tuples (*i.e.,* Figure 4). More specifically, when we use small thresholds the operator will not read tuples in their correct timestamp-order and therefore it may incorrectly shift its 5 seconds sliding window. Due to the shifting of the sliding widow some comparisons between the tuples of the two input sources will not be performed and thus fewer events will be detected. In Figure 4, we illustrate the number of out-of-order tuples of the corresponding time periods presented in Figure 2 when we use different queueing thresholds. It can be observed that in periods of high delay the number of out-of-order tuples increases when small queueing thresholds are used.

In this work we propose a novel system that enables deterministic stream processing under latency constraints by examining the trade-off between the application's deterministic operation and its real-time response requirements. We propose a novel data structure, called Slack-ScaleGate (*SSG*), which exploits the notion of *slack-ready* tuples and enables us to bound the maximum amount of time that a tuple will be stalled to meet its latency constraint. We *pro-actively* adjust the *SSG*'s slack threshold parameter to satisfy latency constraints and at the same time to minimize the amount of *slack-ready* tuples that are read from *SSG*. To address the effects of the relaxation of the determinism, we handle late-arrivals that may occur in order to maximize the operator's deterministic operation. The main contributions of this work are the following:

- We define the notion of *slack-ready* tuples in order to capture the maximum acceptable latency to satisfy the user's response time requirements. We exploit this definition in our data structure, called *SSG*, which allows us to (a) maximize deterministic processing while (b) meeting real-time latency criteria. We analyze the determinism properties of *SSG* with respect to the input streams and their effect on the object's execution, and show that *SSG* can transition back to determinism even after the latter has been relaxed.
- We formulate the problem of maximizing the deterministic stream processing under latency constraints as a single-objective optimization problem that targets at minimizing the amount of *slack-ready* tuples that are read due to latency-related performance goals.
- We provide a novel system which exploits the *SSG* data structure to guarantee the deterministic stream processing of incoming tuples and also automatically tunes the slack threshold parameters to solve the aforementioned optimization problem. Our approach: (i) applies *Gaussian Processes* to capture the impact of the slack threshold parameter on the application's latency and the amount of *slack-ready* tuples that will be read, and (ii) adjusts the time-window parameters using a greedy algorithm that

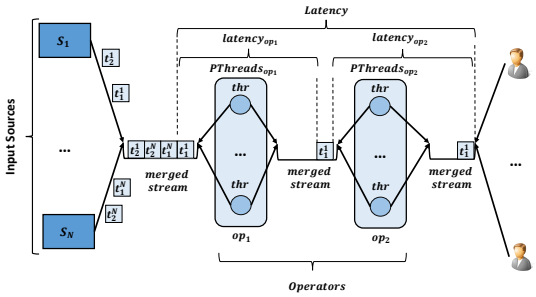


Figure 5: Application graph in our system.

enables us to avoid deadline violations and to minimize the amount of *slack-ready* tuples that will be processed.

- Finally, we provide an extended experimental evaluation which demonstrates the benefits of our approach using a real-world traffic monitoring application. Our experimental results illustrate that our approach is practical, exhibits good performance, and effectively manages to relax the determinism guarantees only when it is truly necessary to meet the application's real-time performance criteria.

2 PRELIMINARIES

In this section, we describe our stream model and provide the necessary determinism definitions.

2.1 Stream Basics

A data stream S_j is an unbounded sequence of tuples t_i^j where i represents the arrival order of the tuple within the S_j stream. Each tuple has a timestamp, denoted by $t_i^j.ts$ which indicates the time that the tuple was generated at the data source. A tuple t_k^j is characterized as a *late-arrival* or an *out-of-order tuple*[1] if there exists another tuple, t_i^j, in S_j such that $i < k$ and $t_i^j.ts > t_k^j.ts$.

An application running in a stream processing system is typically represented as a graph where nodes correspond to processing operators (such as joins or top-k operators) and edges denote the communication among the operators. Let $Operators$ denote the set of operators that comprise the application graph. Each $op \in Operators$ is implemented using multiple threads, $PThreads_{op}$, that concurrently process the streaming data as can be observed in Figure 5. An operator can be either *stateless* or *stateful* [7]. The main difference of the two operator types is that *stateful* operators process and keep the incoming data in memory buffers for a fixed amount of time, while *stateless* operators only process the tuples. We define as SW_{op} the time the information carried by a certain tuple resides on the operator's buffer. If the operator is stateless this metric will be equal to zero as no tuple needs to be kept.

An operator can receive tuples from multiple input streams. As we illustrate in Figure 5, tuples arriving from different input streams are merged into a new stream before they can be read and processed by the operator. We assume that the input streams deliver tuples with increased timestamps (*i.e.,* each individual data stream does not produce *out-of-order* tuples). We argue that this assumption holds true in many use cases and more specifically in traffic monitoring applications like the ones we study in our experimental evaluation (see Section 6) where data are reported by sensors at fixed time intervals. In such applications it is not possible to have out-of-order tuples in the input sources as the sensors produce ordered reports (*e.g.,* a bus sensor cannot produce a report for $10:30$ *pm* and then generate a report for $10:15$ *pm*). However, despite the fact that sources deliver tuples with the correct sequence, there is no guarantee that tuples will be inserted in the correct timestamp-order on the operator's merged stream due to *communication* or *processing delays* which occur in the arrival of the tuples from the input streams. Therefore the operator will receive and process *late-arrivals* from the merged-stream unless a mechanism that orchestrates the tuples processing order is provided.

2.2 Determinism and ready tuples

A stream operator is considered *deterministic*, if, given the same sequences of input tuples, the same sequence of output tuples will be produced, independently of the streams' inter-arrival times and processing order. It is very important to guarantee the deterministic processing of input tuples for applications like click stream analysis and traffic monitoring due to the fact that non-determinism can cause money loss or missed events. Deterministic stream processing is achieved by merging the timestamp-sorted tuples coming from different streams and feeding the operator with a timestamp-sorted stream of *ready* [8] tuples.

Definition 2.1. Ready Tuple. Let t_i^j be the i-th tuple from timestamp-sorted stream S_j. t_i^j is **ready** to be processed if $t_i^j.ts \leq merge_{ts}$, where $merge_{ts} = min_k\{t_l^k.ts\}$ is the minimum timestamp among the timestamps in the set of tuples comprising the latest received tuples t_l^k from each timestamp-sorted stream S_k.

To guarantee that only *ready* tuples will be read by the operators, one approach is to use *dedicated* operators to merge incoming tuples from multiple sources such as *Input Mergers* [7] and *SUnions* [4]. *ScaleGate* [8] is a recently proposed data structure that adopts a different approach, without requiring the use of extra operators. *ScaleGate*[2] encapsulates the necessary communication between the input sources and operator's processing threads in order to decide whether a tuple is *ready* or not. *ScaleGate*'s algorithmic implementation is based on a lock-free implementation of the skip list [21] data structure, to maintain a timestamp-sorted multi-level linked list of the input tuples and enable concurrent insertions with probabilistically logarithmic overhead, together with a novel flag mechanism to efficiently detect *ready* tuples.

3 MODELING OF THE PROBLEM

In this section we model the problem of maximizing the determinism under latency constraints.

In order to guarantee the deterministic processing of tuples, operators must read only *ready* tuples. However, this requires stalling arriving tuples until their timestamp is less than the merge timestamp. The amount of time that a tuple should wait before it is considered *ready* depends on the rate with which tuples arrive in the system. So if the input source experiences delays (*e.g.,* due to

[1] in our model the terms *late-arrival* and *out-of-order tuple* denote the same notion so we use them interchangeably

[2] https://github.com/dcs-chalmers/ScaleGate_Java

network communication) this will lead to an increase of the time required to read a *ready* tuple as the merge timestamp is updated only when we have received input from all the input sources (including the one experiencing the delays). This stall of tuples for satisfying the *ready* condition can be seen as an extra queuing delay that affects the operator's execution time (*i.e.,* latency) and thus the execution time of the entire stream processing application.

In this work we provide a model that considers the impact of these queuing delays on the application's end-to-end execution time. Assume a distributed stream processing application represented as a processing graph whose nodes correspond to processing *Operators* (such as join or aggregate operators) and edges denote the communication among the operators (*i.e.,* see Section 2.1). Let $latency_{op}, \forall op \in Operators$ denote the time required by an operator to read and process an incoming tuple, computed as follows:

$$latency_{op} = max_{thr \in PThreads_{op}}\{queue\_delay_{thr,op} + proc\_time_{thr,op}\}, \forall op \in Operators \quad (1)$$

where the operator's latency is a function of the time required to process the tuple by one of the operator's processing threads (expressed via the $proc\_time_{op,thr}$ metric) and the corresponding queueing delay $queue\_delay_{thr,op}$ at the thread's thr queue. Threads may process tuples with different speeds so the latency of the operator depends on the execution time of the slowest running thread. Deterministic processing indicates higher $queue\_delay_{thr,op}$ and thus higher $latency_{op}$ as tuples would need to be stalled until they become *ready* to be processed. Let $comm_{op->op'}$ depict the communication time between operator op and its downstream operator op'. We can then compute the end-to-end *Latency* of the application graph via the following Equation:

$$Latency = max_{path} \sum_{op \in Operators} (latency_{op} + comm_{op->op'}) \quad (2)$$

where max_{path} is used in the case that the application is represented as a graph with more than one paths, so that the end-to-end execution time of the application is the maximum path latency. Application users can use the *Deadline* constraint to impose a constraint on the time it should take for data tuples to be processed end-to-end. For example, in the traffic monitoring system we described in Section 1, the City Operators expect that traffic events (*i.e.,* congestion, delays) should be detected within *one* minute (from the time the data is generated at different input sources, a join operation is performed and a final result is returned).

When the *Latency* value exceeds the *Deadline* constraint then a deadline violation has occurred. Our goal in this work is to avoid such violations so we have to satisfy the following constraint:

$$Latency < Deadline \quad (3)$$

To satisfy the application's deadline requirements, it is necessary to minimize the amount of time that tuples will be stalled until they are considered *ready*, thus we relax Definition 2.1. More specifically, the idea is to bound the maximum amount of time that a tuple will wait before it is read by the operator's processing threads. Therefore, we propose the notion of *slack-ready* tuples:

Definition 3.1. Slack-Ready Tuple. Let t_i^j be the i-th tuple from timestamp-sorted stream S_j. t_i^j is **slack-ready** to be processed if $max_{ts} - t_i^j.ts > SLT$, where $max_{ts} = max_k\{t_l^k.ts\}$ is the maximum timestamp among the timestamps in the set of tuples comprising the latest received tuples t_l^k from each timestamp-sorted stream S_k and $SLT \in [0, max_{ts} - merge_{ts}]$ is the user-defined slack threshold parameter that controls the time duration that a tuple can be stalled.

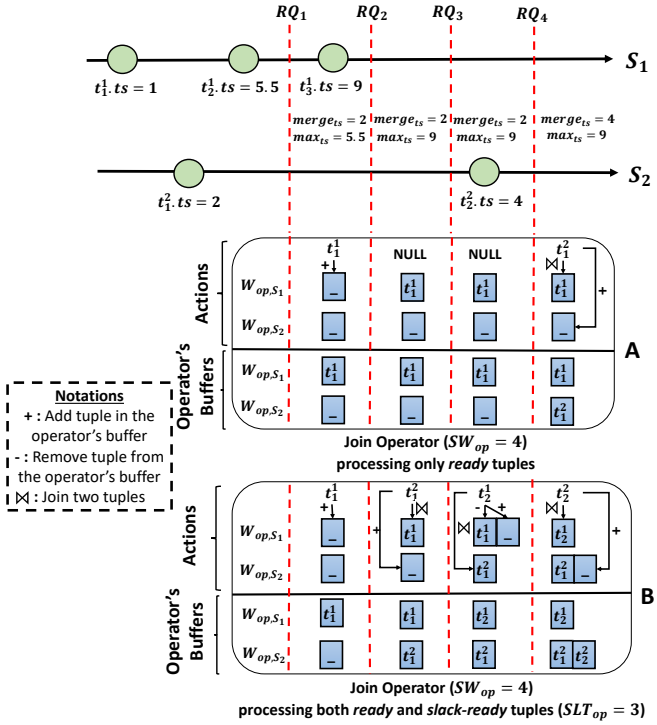


Figure 6: Example of slack-ready tuples' impact on a join operator that processes tuples from two input sources.

Our goal is to exploit the *slack-ready* tuples definition in our system by adding the SLT_{op} parameter on each operator $op \in Operators$. Essentially, feeding an operator with *slack-ready* tuples guarantees that there is a constraint on how much a tuple will be stalled in order to minimize the probability that the application deadline is violated. This approach relaxes the operator's determinism so if an operator reads and processes *slack-ready* tuples then a tuple with lower timestamp arriving from a source that experiences delays will be read and processed by the operator as an *out-of-order* tuple. These out-of-order tuples will contribute on the operator's results as long as the sliding window they belong has not been shifted due to the reading and processing of *slack-ready* tuples.

The *SLT* value is meaningful when it is bounded to $max_{ts} - merge_{ts}$ as this is the maximum difference that can occur between any two tuples that are added by the input sources. When this upper threshold (or a higher one) is used the *slack-ready* tuples end up being equivalent to the *ready* tuples (*i.e.,* $max_{ts} - t_i^j.ts \geq max_{ts} - merge_{ts} \Rightarrow merge_{ts} \geq t_i^j.ts$). We should clarify that Definitions 2.1 and 3.1 are not mutually exclusive. A tuple that is *ready* can also be characterized as *slack-ready* if it is processed when the difference of its timestamp with the maximum timestamp is greater than *SLT* and at the same time its timestamp is smaller than the merge timestamp. However, the *ready* definition provides stronger determinism guarantees as all tuples will be read in the correct

timestamp-order, but at the expense of extra queueing delay as we cannot control the amount of time that tuples will be stalled. We elaborate with the following example on (a) how the *SLT* parameter can bound the amount of time that a tuple will be stalled and (b) how *SLT* may affect the accuracy of the operator's results.

Example Description. In Figure 6 we illustrate a *join* operation on two input sources (*i.e.,* S_1 and S_2) and its read requests overtime (*i.e., RQs* in Figure 6). Incoming tuples are kept in two sliding windows (W_{op,S_1} and W_{op,S_2} respectively) and the size of each sliding window is 4 time units. In Figure 6 we display the operator's actions and the state of its sliding windows when the join operator reads: (i) only *ready* tuples (*i.e.,* diagram *A* in Figure 6) and (ii) both *ready* and *slack-ready* tuples (*i.e.,* diagram *B* in Figure 6).

Exploiting the notion of *slack-ready* tuples, relaxes the determinism guarantees in order to meet the operator's real-time criteria. In Figure 6, diagram *B*, the operator reads the t_2^1 tuple which is a *slack-ready* tuple (*i.e.,* $max_{ts} - t_2^1 = 9 - 5.5 = 3.5 > SLT_{op}$) and it removes t_1^1 from the W_{op,S_1} sliding window as it exceeds the amount of time that it should be kept. In the fourth read request the operator reads t_2^2 which is a *late-arrival* and compares it only against t_2^1 and not t_1^1. However, t_1^1 and t_2^2, should be also compared as they are within the 4 time units sliding window (*i.e.,* $t_2^2.ts - t_1^1.ts = 3$). These tuples will never be compared due to the shift in the sliding window after reading the t_2^1 *slack-ready* tuple. Because we do not perform this comparison we can miss an event that triggers the join predicate. This illustrates that reading and processing *slack-ready* tuples penalizes the operator's accuracy. In contrast, if the operator reads only *ready* tuples, then t_1^1 and t_2^2 are compared but we impose extra queuing delay to the tuples as we have to wait until t_2^2 arrives before the operator is able to read a *ready* tuple from the merged-stream which can violate the operator's real-time requirements.

The number of *slack-ready* tuples, $slrTuples_{op}$, that have been read by the operator's processing threads provide an insight of how much the operator's determinism guarantees have been relaxed. More formally, assuming that we have an indication (*e.g.,* a Boolean variable) whether a tuple has been read by a thread as *slack-ready*, we can compute $slrTuples_{op}$ via the following Formula:

$$slrTuples_{op} = \sum_{thr \in PThreads_{op}} slr_{thr,op}, \forall op \in Operators \quad (4)$$

where $slr_{thr,op}$ is the number of *slack-ready* tuples that have been read by processing thread $thr \in PThreads_{op}$. The number of *slack-ready* tuples read by a thread depend on the SLT_{op} parameter as it controls when a tuple should be read as *slack-ready* (*i.e.,* see Definition 3.1). Based on the amount of $slrTuples_{op}$ read per operator we can compute the number of *slack-ready* tuples read by the whole application graph via the following Equation:

$$SLRTuples = \sum_{op \in Operators} slrTuples_{op} \quad (5)$$

This value should be kept as low as possible to avoid missing events of interest and to guarantee the determinism of the processing. A large SLT_{op} can minimize *SLRTuples* but this can increase the application's *Latency* and cause deadline violations. So there is a trade-off between *SLRTuples* and *Latency* that needs to be considered when we tune $SLT_{op}, \forall op \in Operators$. Our goal in this work is to study the trade-off between these two metrics by minimizing Equation 5 and at the same time satisfying the end-to-end execution time requirements of the user expressed via Equation 3. More formally, the problem can be defined as follows:

Problem Definition. Given a stream processing application graph comprising a set of *Operators*, determine $SLT_{op}, \forall op \in Operators$ such that:

minimize $SLRTuples = \sum_{op \in Operators} slrTuples_{op}$

subject to: $Latency < Deadline$

4 SOLUTION OUTLINE

In this section we provide an outline of our proposed solution for maximizing the determinism under latency constraints. Our objective is to provide a system that meets the following requirements:

- The *Latency* demands of the distributed stream processing application expressed in terms of a user-specified deadline, are met, given that the scheduling of the operator's processing threads is not adversarial.
- The operators' processing threads will read and process only *ready* tuples as long as the stalling of tuples for guaranteeing the *ready* condition (cf. Definition 2.1) does not lead to violations of the deadline constraint.
- When the extra queuing delay for guaranteeing the tuples' deterministic processing leads to the violation of the deadline constraint, then the system will sacrifice its determinism guarantees in order to satisfy this constraint. In such cases we process the merged-stream's *out-of-order* tuples as they may contribute to the output results.

To meet the above requirements our approach makes the following contributions:

- We propose a novel data structure, called *SSG*, which exploits the notion of *slack-ready* tuples and enables us to relax the determinism guarantees in order to satisfy response time requirements while handling *late-arrivals* and ensuring no duplicate tuples.
- We propose a methodology and have developed system components to dynamically and proactively tune the *SLT* parameters of the *SSG* data structures in order to guarantee that the *Latency* constraint is always satisfied and the number of tuples that do not satisfy the strong determinism guarantees (cf. Definition 2.1) is minimized.

In the following sections we first describe the SSG API and then provide a short description of our system components.

4.1 SSG API

A basic building block in our approach is a new API that merges the operator's input streams and enables the operator's processing threads to read and process *slack-ready* tuples. We propose a novel data structure, *SSG* that has the following properties: (1) returns a tuple as *slack-ready* when it has been stalled for more than the *SLT* threshold (cf. Definition 3.1), (2) handles *late-arrivals* that may occur due to the determinism relaxation imposed by the *slack-ready* notion, (3) no duplicate tuples are returned to each operator's processing thread and (4) enables the dynamic tuning of the *SLT* parameter so that we can adjust it at real-time based on the current

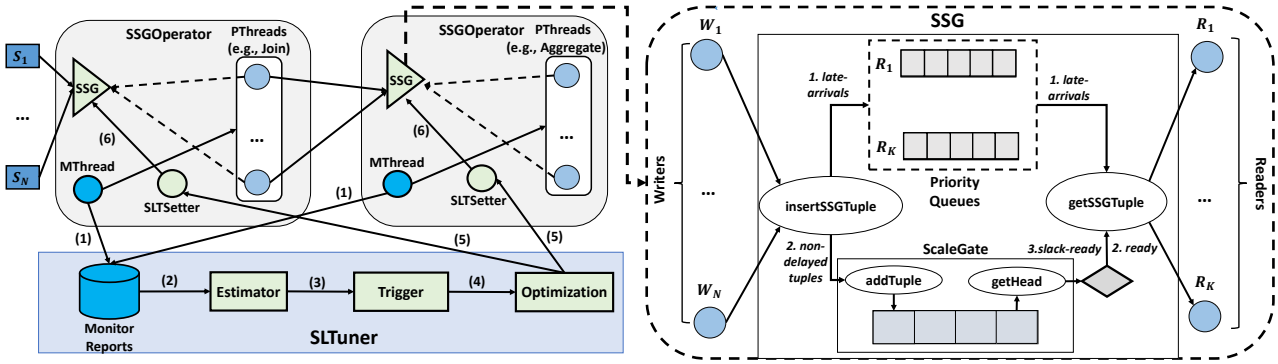


Figure 7: (a) System components and (b) *SSG* implementation details.

system conditions. To satisfy these properties, *SSG* supports the following API:

- *getSSGTuple*(*readerId*): returns to the calling reader entity *readerId* (*e.g.,* an operator's processing thread) the next earliest *ready* or *slack-ready* tuple[3] that has not been yet consumed by the former. Each tuple is guaranteed to be returned at most once to each *readerId*.
- *insertSSGTuple*(*tuple*, *writerId*): inserts into *SSG* a tuple from the writer entity *writerId*.
- *setSLT*(*sltParameter*): sets the *SLT* parameter of *SSG* that controls the amount of time that a tuple can be stalled in the data structure.

There are two entities that can utilize *SSG*'s API, readers and writers. The latter are responsible to insert tuples in *SSG* while readers retrieve tuples from the data structure. Writers in our system correspond to the input sources (*e.g.,* the bus and SCATS input streams in the application example in Section 1) or upstream operators, while readers are the operator's processing threads.

The *setSLT* method can be used for tuning the *SLT* parameter, as *SLT* enables us to control the amount of time that a tuple can be stalled in the data structure. We should emphasize here, that, for a fixed *SLT* value, the determinism level depends on the actual execution and the difference $max_{ts} - merge_{ts}$ that occurs while the tuples arrive into the system, i.e. the difference between the most recent tuples of the fastest and slowest streams respectively.

The effects of the relaxation of the determinism guarantees due to the use of the *SLT* parameter on the *getSSGTuple* method are twofold: i) *out-of-order tuples on the merged stream:* ideally, *getSSGTuple* returns to each *readerId ready* tuples in timestamp order. Under certain conditions, i.e. executions where $max_{ts} - merge_{ts} > SLT$ occurs, tuples that have exceeded the *SLT* threshold are returned, possibly out-of-timestamp order. *E.g.,* assuming that we have two input streams and one of the two streams is delyed in generating tuples for longer than *SLT*, *slack-ready* tuples from the non-delayed stream are returned, and tuples from the delayed stream may be returned later, ii) *missing tuples:* There is no guarantee that all the returned tuples will be used by the *readerId* because e.g., as we mentioned in Section 3 when we process *slack-ready* tuples we may move incorrectly the operator's sliding window and thus some *late-arrivals* will not contribute to the operator's results.

[3]We use the terms *read* a tuple and *return* a tuple interchangeably.

The *insertSSGTuple* method aims at guaranteeing that tuples will be available for reading/processing by all the reader entities. Even when a reader has started reading *slack-ready* tuples, our goal is to provide an insert method that will be able to provide tuples from a delayed stream to the reader, as we argue that these tuples may contribute on the operator's results.

4.2 System Components Overview

Users in our system (*i.e.,* shown in Figure 7) define *SSGOperators* components (*i.e.,* stream processing operators that exploit the use of the *SSG* data structure) only for those operators that have determinism and real-time requirements. In addition, our system has an external component called *SLTuner* for tuning the *SLT* parameters used by the operators' *SSG*. This is an inherently distributed system where operators can be assigned to different computing nodes.

Each *SSGOperator* uses: (a) a single *SSG* data structure for keeping its incoming data, (b) *PThreads* that read and process data from the *SSG* and are a multi-threaded implementation of the stream processing operator (*e.g.,* joins or aggregations), (c) a special monitor thread (*i.e., MThread* in Figure 7) for gathering performance statistics such as the slack threshold used by the *SSG* and the operator's latency and (d) a *SLTSetter* thread which is used to adjust the *SSG*'s *SLT* parameter. The processing threads are responsible to forward generated tuples to the downstream operators' *SSG*s. In order to tune the *SLT* parameters of the *SSGOperators*, the *SLTuner* exploits the three components we illustrate in Figure 7 as follows:

- The *Estimator* component estimates the $latency_{op}$ and $slr\text{-}Tuples_{op}, \forall op \in Operators$ in upcoming time periods. This component uses the previous reports for building two prediction models for each *SSGOperator*.
- The *Trigger* component utilizes the outcomes of the Estimator to determine if the deadline is violated or if the expected latency is significantly smaller than the deadline. The latter condition is used for increasing the *SSG*'s threshold value when we expect low per tuple latency in the upcoming time periods. If one of the two conditions is true then the Optimization component is invoked.
- The *Optimization* component uses a greedy search algorithm for solving the optimization problem we defined in Section 3 and informs the *SLTSetter* threads about the *SLT*s that the *SSG* data structures should utilize.

```
1: PriorityQueue[] priorityQueues[#readerIds]
2: AtomicLong SLT
3: AtomicLong[] maxWriterTs[#writerIds]
4: AtomicLong[][] lastReadTs[#readerIds][#writerIds]
5: // A ScaleGate implementation as in [8] with access to the getHead(readerId)
   //method that returns the last tuple read by each readerId
6: ScaleGate SG
```

Algorithm 1: SSG data structure: main variables

```
1: nonOvertakenReaders ← {}
2: for readerId ∈ lastReadTs do
3:     lt ← max(lastReadTs[readerId])
4:     //Check if the reader has overtaken the writer.
5:     if (lt.ts > tuple.ts) then
6:         priorityQueues[readerId].add(tuple)
7:     else
8:         nonOvertakenReaders.add(readerId)
9: //Add tuple to ScaleGate only if some reader has not overtaken this writer.
10: if (!nonOvertakenReaders.isEmpty()) then
11:     SG.addTuple(tuple, writerId)
12: for all (readerId ∈ nonOvertakenReaders) do
13:     //Get reader's maximum read timestamp.
14:     maxTs ← max(lastReadTs[readerId])
15:     //Get reader's last read tuple from writerId.
16:     ltw ← lastReadTs[readerId][writerId]
17:     //Add tuple to priority queue only if the reader has overtaken the writer and it
        //has not already read this tuple.
18:     if (maxTs > tuple.ts && tuple.ts > ltw.ts) then
19:         priorityQueues[readerId].add(tuple)
20:         nonOvertakenReaders.remove(readerId)
21: maxWriterTs[writerId] ← tuple.ts
```

Algorithm 2: *insertSSGTuple*(*tuple*, *writerId*)

```
1: //Check if reader's priority queue has elements.
2: if !priorityQueue[readerId].isEmpty() then
3:     t ← priorityQueue[readerId].poll()
4:     lastReadTs[readerId][t.writerId] ← t.ts
5:     //Characterize tuple as ready or slack-ready
6:     mergeTs ← min(maxWriterTs)
7:     if t.ts ≤ mergeTs then
8:         t.isSlackReady ← false
9:     else
10:        t.isSlackReady ← true
11:    return t
12: //Get the next tuple that should be read by readerId and check (slack)-ready
    //conditions
13: t ← SG.getHead(readerId)
14: if (t == NULL) then
15:     return NULL
16: mergeTs ← min(maxWriterTs)
17: //Ready tuple condition.
18: if t.ts ≤ mergeTs then
19:     t.isSlackReady ← false
20:     //Update lastReadTs appropriately
21:     lastReadTs[readerId][t.writerId] ← t.ts
22:     return t
23: maxTs ← max(maxWriterTs)
24: //Slack-ready tuple condition.
25: if maxTs − t.ts > SLT then
26:     t.isSlackReady ← true
27:     lastReadTs[readerId][t.writerId] ← t.ts
28:     return t
29: return NULL
```

Algorithm 3: *getSSGTuple*(*readerId*)

5 SYSTEM IMPLEMENTATION

In the following sections we describe in more details the implementation of the *SSG* data structure and the three components that comprise the *SLTuner*.

5.1 SSG's Algorithmic Implementation

In Figure 7 we illustrate the basic components of our proposed *SSG* data structure and how it is utilized by the writers and readers. More specifically, *SSG* uses (a) a *ScaleGate* object for keeping tuples in timestamp-sorted order, (b) an array of priority queues (one queue per *readerId*) for keeping *late-arrivals* that occur after *SSG* returns *slack-ready* tuples, and (c) some additional synchronization variables (see Alg. 1) for handshaking between writers and readers with the help of timestamps of already written and read tuples.

Combining ScaleGate with Priority Queues. *SSG* should be able to return both *ready* and *slack-ready* tuples. ScaleGate by default enables the access only to the next *ready* tuple for each *readerId*, which would not allow us to characterize a tuple as *slack-ready*. Therefore, we modified the ScaleGate data structure and allow it to also include a method to return the next tuple with timestamp larger than the timestamp of *readerId's* last read tuple, regardless if this tuple is *ready* or not (*i.e., getHead* method in Figure 7). Furthermore, when *SSG* returns *slack-ready* tuples, the determinism guarantees are violated and *late-arrivals* can occur in the merged-stream; such late-arrivals are stored in the priority queue of a reader that had returned slack-ready tuples; this is needed since ScaleGate would not be able to provide them to the readers, as the tuple returned from the *getHead* method would have a timestamp larger than the timestamp of the *late-arrival*.

Inserting tuples. Our implementation of the *insertSSGTuple* method (*i.e.,* Alg. 2) handles *late-arrivals* and guarantees that each tuple is returned at most once to each reader, i.e. without duplicates. More specifically, first we check if some reader has read a tuple with timestamp larger than the tuple we are about to insert. If so we add the new tuple in the reader's priority queue. Otherwise we add the tuple in the main *ScaleGate* component. Nevertheless, due to the asynchrony of the system, there is still a possibility for the reader to overtake the writer before the insertion of the tuple is finished and thus be missed. To limit the impact of such cases the insertion mechanism checks again for the respective readers status (cf. lines 12 − 20 of Alg. 2). Alternatively, such tuples could be optimistically added to the priority queues and burden the readers with checking for duplicate tuples between the *ScaleGate* component and the priority queues, thus guaranteeing that no tuples would be missed. Our design choice is to keep the readers more lightweight, and allow this behaviour while we tackle the non-determinism problem in a higher level by regulating the *SLT* parameter.

The method introduces overhead of keeping extra information for each reader, as we will have one priority queue for each reader. Using one priority queue for all the readers would not work correctly as the last read tuple may differ across the readers. Therefore, some readers may actually read correctly the tuple while others may have already read a tuple with larger timestamp. Furthermore, because we keep in the queues only a reference to the actual tuple, the memory overhead is rather small. In modern systems usually memory usage is not the bottleneck thus it is a valid choice to keep *late-arrivals* and mitigate the information loss.

Getting tuples. For the *getSSGTuple* method (*i.e.,* see Alg. 3) we want to guarantee that the reader will be able to read tuples residing in its priority queue (*late-arrivals*) and will exploit both the *ready* and *slack-ready* tuples' definitions. When a reader entity wants to

retrieve a tuple from *SSG* the following steps are performed: (1) the method checks if we have a tuple residing in the reader's priority queue and returns the first such tuple if so (lines 1 – 11 of Alg. 3), (2) If the priority queue is empty then the method checks if the next tuple to be read satisfies the *ready* tuple condition and returns it if so (lines 12 – 22 of Alg. 3), (3) If the tuple is not *ready*, *SSG* examines whether the *slack-ready* definition holds true and if so it returns the tuple (cf. lines 23 – 28 of Alg. 3), and (4) if the tuple is not *slack-ready*, *SSG* returns *NULL* (line 29 of Alg. 3) and the reader entity will have to wait before it re-tries to read a tuple.

5.1.1 Analysis. The interface of *SSG* is, by design, aware of tuples that have been delayed and thus execution-dependent, with respect to how streams are delivered to the system. The properties below describe the possible behavior of *SSG* during different parts of an execution.

First, we define the properties of an ideal deterministic object O that has an interface for receiving tuples from multiple streams and gives an output stream as a result, which essentially models a streaming operator, or a component of it. We assume one writer thread for each input stream S_i and one reader thread that produces the output stream R, using the same identifiers for threads and streams. In the settings of the common shared memory model [10], we consider a *history* H of an object as a sequence of *invocations* and *responses* of its methods. We call *sub-history* a continuous (*i.e.,* not skipping any method invocation or response) sub-sequence of H. We call *thread sub-history* $H|S_i$ the projection of H that includes only methods executed from S_i. In the following, given the streaming setting of the problem, we assume infinite histories. Each tuple t received from an input stream S_i, for O corresponds to an invocation and response of the respective insert method (*e.g.,* in the *SSG* object this would be the $insertSSGTuple(t, S_i)$ method call). Respectively, a tuple t of the output stream R, is the returning result from a response of the appropriate method call (*e.g.,* the $getSSGTuple(R)$ method for *SSG*).

Definition 5.1. Let two histories H and H'. O will have the same input streams (as in the same sequences of tuples per stream) in H and H', if $\forall i$ the thread sub-history $H|S_i$ is equivalent to $H'|S_i$.

Note, that if H and H' have the same input streams, the way that any two projections $H|S_i$ and $H|S_j$ interleave within H, might be arbitrarily different from the way the respective projections interleave within H'. Thus, we can reformulate the notion of a deterministic stream operator for O (cf. Sec. 2.2) as follows:

Definition 5.2. O is deterministic if for any two histories H and H' with the same input streams, then the non-null responses of $H|R$ are equal and in the same order to the non-null responses of $H'|R$ (i.e. the same output stream).

We further say that an object is in a *steady* state, if for all possible future sub-histories that have the same input streams, it is deterministic. Essentially, a deterministic object O is always in a steady state. For example, if two histories with the same input streams were "frozen" to a point where the same input tuples have been received, one could exchange the object O instances and maintain the determinism property for the rest of the histories.

We will now argue that *SSG* will be deterministic for sub-histories where the *SLT* threshold is not violated. We call a sub-history *SLT-compatible* if during the sub-history the condition $max_{ts} - merge_{ts} \leq SLT$ holds, and respectively *SLT-incompatible* if the condition is violated within the sub-history.

Property 1. SSG *will not return* slack-ready *tuples during an SLT-compatible history.*

Proof Sketch. Tuples will be returned as *slack-ready* only if they have been stalled (while $t.ts > merge.ts$ holds and they have not been returned as *ready*) for more than *SLT* time. But this can only happen if $merge.ts$ differs from max_{ts} more than *SLT*, i.e. within an *SL*-incompatible history. □

We can observe that if an entire history H is *SLT*-compatible, then *SSG* behaves deterministically, as it returns only *ready* tuples as *ScaleGate* [8]. However, describing what happens in a history that consists of both *SLT*-compatible and *SLT*-incompatible sub-histories is not straightforward.

Consider that *SSG* may return tuples out of timestamp order when they are returned from a $priorityQueue$ (cf. lines 2 – 11 in Alg. 3). From Algorithm 2 we see that a tuple is added in the priority queue of some reader only if that reader has already read a tuple with higher timestamp. A reader will respectively read a tuple t with $t.ts > merge_{ts}$ only if it is *slack-ready* at that point in the history execution. Thus, we can see that the following holds:

Property 2. *If a tuple t is added to the $priorityQueue$ of a reader, then there exists another tuple t' that has been previously returned as* slack-ready *to the same reader.*

Assuming that within a history H all *SLT*-incompatible sub-histories are of finite length, we can show:

Theorem 5.3. *For a long enough SLT-compatible sub-history, i)* SSG *will reach a* steady *state in a finite amount of time, and ii)* SSG *will remain in* steady *state for the remaining sub-history.*

Proof. Let h be an *SLT*-compatible sub-history of a history H. By Property 1, no *slack-ready* tuple will be returned within h. Given that previous *SLT*-incompatible sub-histories of H are of finite length and by Property 2, in a finite amount of time proportional to the size of the $priorityQueues$, the latter will become empty. This is a *steady* state for the object, since no out-of-order and no *slack-ready* tuples will be returned. Finally, because h is *SLT*-compatible, the object will remain in *steady* state during h. □

Theorem 5.3 shows that not only *SSG* behaves well in *SLT*-compatible histories, but, even after incompatible sub-histories, deterministic behavior can be achieved again given enough time.

The following property characterizes the tuples returned from a $priorityQueue$.

Property 3. *Tuples returned from $priorityQueues$ are at least* slack-ready.

Proof Sketch. By Property 2, a tuple t is inserted into a $priorityQueue$ if another tuple t' with $t'.ts > t.ts$ has been returned as *slack-ready*, i.e. $maxTs - t'.ts > SLT$. Thus, $maxTs - t.ts > SLT$. □

On top of Property 3, Alg. 3 checks if a tuple has become ready before outputting it, further increasing the monitoring accuracy.

5.2 SLTuner's Components

In this section we provide the implementation details of the three components (*i.e., Estimator*, *Trigger* and *Optimization*) that are utilized by the *SLTuner* in order to tune $SLT_{op}, \forall op \in Operators$.

5.2.1 Estimator Component. The Estimator component captures the impact of *SLT* parameter in the operator's latency (*i.e.,* see Equation 1) and the amount of *slack-ready* tuples (*i.e.,* see Equation 4) that the operator will read. The Estimator creates two different prediction models for each operator, the first for estimating its latency (*i.e.,* $latency_{op}$) and the second for estimating the number of *slack-ready* tuples (*i.e.,* $slrTuples_{op}$). Then it applies Equations 2 and 5 to estimate the total end-to-end execution time (*i.e., Latency*) and the total number of *slack-ready* tuples in the system (*i.e., SLRTuples*). For our prediction models we use the following features vector:

$$\vec{x}_{op} = (SLT_{op}, hour_{op}, min_{op}, sec_{op}, |Pthreads_{op}|, SW_{op}), \quad \forall op \in Operators \tag{6}$$

where $hour_{op}$, min_{op} and sec_{op} correspond respectively to the hour, minutes and seconds of the last tuple that has been processed by operator *op*. We added these feature to detect periodic patterns in the input stream similarly to [24]. Furthermore, we use the SW_{op} as a feature because the size of operator's sliding window will affect the operator's processing latency.

We decided to use a well-known technique, *Gaussian Processes* [18], that has been efficiently applied in similar context [24]. *Gaussian process* is a non-linear non parametric model and is an extension of the multivariate *Gaussian distribution* for infinite collection of real-valued variables. In order to estimate the two metrics we used the following *Gaussian distributions*:

$$latency_{op}(\vec{x}_{op}) \sim \mathcal{N}(\bar{f}(\vec{x}_{op}), var_1(\vec{x}_{op})), \forall op \in Operators \tag{7}$$

$$slrTuples_{op}(\vec{x}_{op}) \sim \mathcal{N}(\bar{g}(\vec{x}_{op}), var_2(\vec{x}_{op})), \forall op \in Operators \tag{8}$$

where $\bar{f}(\vec{x}_{op})$ and $\bar{g}(\vec{x}_{op})$ will be the mean of the *Gaussian distributions* while $var_1(\vec{x}_{op})$ and $var_2(\vec{x}_{op})$ will be the corresponding variance or uncertainty of the estimations. These functions are computed using historical data and then the system uses Equations 7 and 8 for estimating the two metrics of interest. In most cases *Gaussian Processes* require more time for the training phase than simpler techniques like *Linear Regression*. However, when the model is trained once as in our case, the training overhead can be tolerated as it will help the system to capture the *SLT* parameter's impact more accurately. It should be clear that the user can easily plug-in different prediction techniques in our system.

5.2.2 Trigger Component. For determining when the optimization algorithm that tunes the slack threshold should be triggered we decided to examine the applicability of a commonly used scheduling metric, laxity [23], that depicts how close the estimated execution time is to the user-defined deadline. More formally, $Laxity = Deadline - Latency$ where *Latency* is the estimated end-to-end execution time. The smaller the *Laxity* value is, the higher will be the probability of a deadline miss. Therefore, the optimization algorithm is triggered when the following condition is true:

$$Laxity \leq 0 \tag{9}$$

We also need to accommodate cases when *Laxity* has a large value and the operator's latency is very small, and thus it may be possible to increase the SLT_{op} to minimize the amount of *slack-ready* tuples that will be processed. For this reason we also run the optimization algorithm when the following condition is true:

$$Laxity \geq \alpha \times Deadline \tag{10}$$

where $\alpha \in (0, 1)$ is a user-defined parameter that determines when the *Laxity* metric has a large value. The greater the α parameter is the lower should be the application's *Latency* in order to trigger the algorithm, as it will be harder to satisfy Equation 10.

5.2.3 Optimization Component. In order to solve our optimization problem we applied a greedy Hill climbing algorithm that enables us to tune the slack threshold parameters fast and efficiently. First, we set the *SLT* parameters to some initial values and then gradually increase their values, estimating at each step the corresponding latency and the total number of *slack-ready* tuples that will be processed. In each step we increase the SLT_{op} that leads to the minimization of the *slack-ready* tuples metric and at the same time satisfies the deadline constraint. We stop the search if we fail to find a valid increase and return the currently found parameters. The performance of the algorithm depends on the *SLTs* and the step sizes we use for increasing them. When we apply the algorithm due to Equation 9, the search should start with small initial *SLT* values in order to quickly satisfy the deadline constraint. Then, by gradually increasing the slack thresholds we will be able to minimize the amount of *slack-ready* tuples that we will read. In case that the algorithm is triggered due to Equation 10 we start the search with the current *SLTs* as these parameters satisfy the deadline constraint. Finally, we decided to invoke this greedy algorithm and avoid more elaborate techniques like [22] in order to detect fast the appropriate slack thresholds as we examine the problem in streaming context.

6 EVALUATION

Setup. We evaluated our approach using a streaming smart city application running in Dublin city. The app receives data from sensors mounted on top of public buses[4] which report how delayed is a bus from reaching the next bus stop. It also receives data from SCATS sensors[5] that measure the traffic flow in road intersections. We have 911 buses and each bus sends a new report every 20 seconds, while we have 3,504 SCATS sensors and each SCATS sends a new report every minute. We increased the rate with which reports are emitted by the sensors, to stretch our system's performance. The goal of the app is to detect areas that are experiencing high traffic congestion. First we split Dublin city into sub-areas and then for each area we monitor reports from both input sources in a sliding window of 5 seconds (*i.e.,* $SW_p = 5,000$) and trigger a congestion event when a bus reports a congestion event and in the same area a SCATS sensor has reported a traffic flow larger than a specified value. Our experiments run on a server equipped with Intel(R) Core(TM) i7-3770 3.40GHz CPU and 16 GB RAM.

Comparison against ScaleGate and K-slack. In the first set of experiments we compare *SSG* against the *ScaleGate* and *K-slack* data structures in terms of the overhead that is added due to the use of extra synchronization parameters as we described in Section 5.1.

[4] https://data.dublinked.ie/dataset/dublin-bus-gps-sample-data-from-dublin-city-council-insight-project

[5] https://data.dublinked.ie/dataset/scats-dcc-jan2013

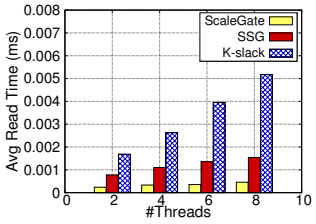


Figure 8: Tuples reading overhead.

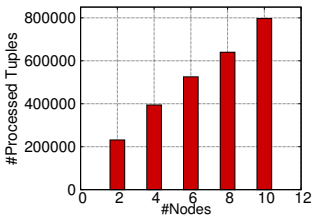


Figure 9: Processed tuples using varying number of nodes.

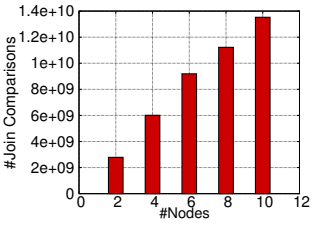


Figure 10: Join comparisons using varying number of nodes.

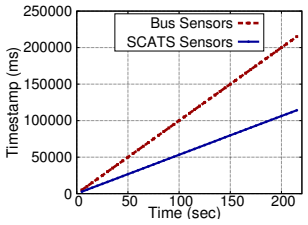


Figure 11: Reported timestamps from the two sources.

Algorithm	$NRMSE_1(\%)$	$NRMSE_2(\%)$	Time (sec)
Gaussian Processes	**10**	**25**	**2**
Linear Regression	20	26	0.06
Isotonic Regression	19	26	0.2
SVM	15	25	4

Table 1: Evaluation of different prediction algorithms.

We decided to compare our data structure against *K-slack* as it was utilized by all previous works that studied the determinism problem under latency constraints [16, 17].

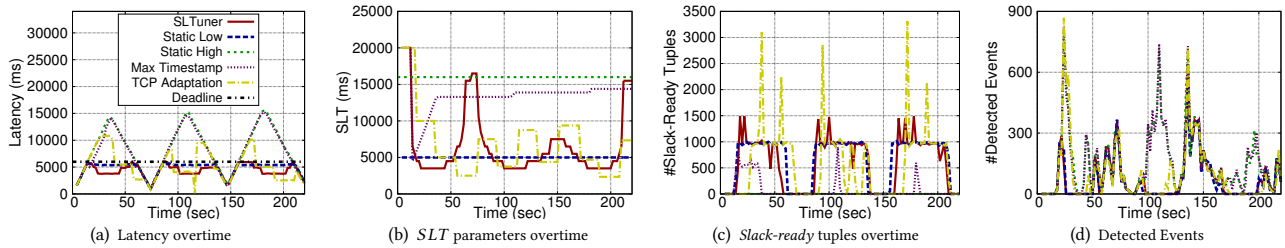
In order to be able to capture the overhead of the synchronization on the reading time, we considered a benchmark that readers only retrieve tuples from the data structure and do not perform any processing. We used two writers and each one generated 20,000 tuples. As it can be observed in Figure 8, ScaleGate has lower average reading time than *SSG* due to the fact that the latter adds extra synchronization overhead due to the extra *AtomicLong* variables that are used for keeping the maximum tuples' timestamps that have been returned to the readers. However, even when 8 threads are executing concurrently the overhead is minimal as the average reading time is less than 0.002 ms. Furthermore, *SSG* is able to support significantly lower reading time than *K-slack* due to the fact that *SSG* is a lock-free approach in contrast to *K-slack* that uses locks for synchronizing the readers and writers.

Scalability. In the second set of experiments we evaluated the scalability of *SSG* when it is utilized on top of Apache Storm which is one of the most well-known distributed stream processing systems. Furthermore, we wanted to illustrate that the synchronization overhead for guaranteeing the deterministic processing is negligible and thus *SSG* can achieve nearly linear scalability when more processing nodes are used. We run our traffic join application on our local 8 nodes cluster where each node is equipped with 8 CPU and 16 GB RAM. We run the experiments for 400 seconds and each writer emitted 1,000 tuples per second. As it can be observed in Figures 9 and 10, we varied the number of nodes and measured the performance of the application in terms of processed tuples and comparisons that are performed by the operators. There is an almost linear increase of the processed tuples and performed comparisons when we increase the number of join operators that execute concurrently in the Storm application. Therefore, this points out that *SSG* is able to exploit the parallelism offered by Storm without penalizing the application's performance due to the synchronization required for guaranteeing the determinism of the processing.

Prediction Model Evaluation. We evaluated the applicability of *Gaussian Processes* used by the Estimator component for predicting the latency and the amount of *slack-ready* tuples read by the join operator when we use varying slack threshold parameter. We run the application for different slack threshold parameters and kept the latency and the amount of *slack-ready* tuples as historical data. We used these historical data for building the prediction models. We compared *Gaussian Processes* against three commonly used approaches, *Linear Regression*, *Isotonic regression* and *Support Vector Machines (SVM)*. In Table 1 we illustrate the performance of the four different techniques. We evaluated the algorithms using 50% of the historical data for training the models (*i.e.,* approximately 923 samples) and used the other 50% as test data were we computed the normalized mean squared error (*NRMSE*). $NMRSE_1$ corresponds to estimation error of the latency metric while $NMSRE_2$ refers to the error when estimating the number of *slack-ready* tuples. As it is shown in Table 1, *Gaussian Processes* is a very powerful technique that is capable of capturing the features of the training examples and minimizes the *NRMSE*. However, *Gaussian processes* lead to high execution time (*i.e.,* 2 seconds). We argue that the overhead is negligible as the training is performed only once.

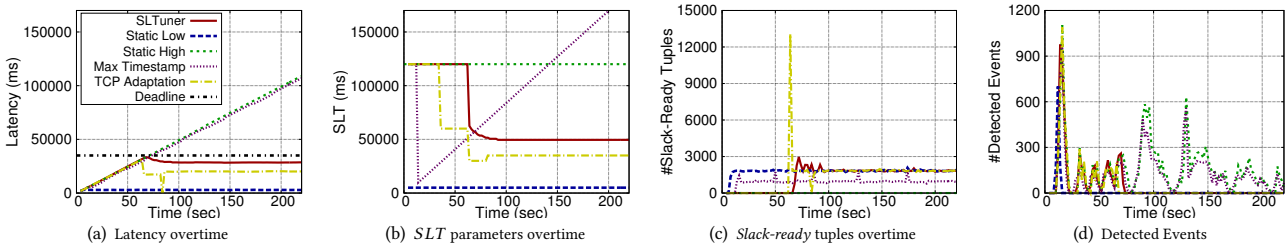
SLTuner Evaluation. We compared the performance of our proposed slack threshold adaptation technique, *SLTuner*, against other commonly applied techniques in similar settings. Specifically, we examined the following approaches: (1) **Static Low** which uses a fixed low slack threshold throughout the experiments, (2) **Static High** which uses a fixed high slack threshold, (3) **Max Timestamp** [16] which adjusts dynamically the K parameter in the *K-Slack* data structure and was applied in our system by setting the K parameter equal to the maximum observed delay in the incoming tuples and (4) **TCP Adaptation** [17] which uses a TCP-like adaptation algorithm that decreases by half the K parameter of *K-slack* when we observe an increase in the latency, while it increases K by a fixed amount (*i.e.,* 1000 *ms*) when latency has decreased.

In these experiments both the bus and the SCATS input sources emitted 500 reports per second. In order to compare our approach against these alternative techniques, we delayed the bus input source to increase the queuing delay and thus pinpoint the need of relaxing the determinism guarantees. We considered two different delaying policies, one that leads to periodic delays of the bus input source, and another that has a constant increase in the delay of the tuples fed by the bus input source and thus depicts an extreme scenario. In Figures 1 and 11 we illustrate the tuples' timestamps that have been inserted by the two input sources in *SSG* as a function of time. As seen in Figure 1 (see Section 1), we have periods with increased delay difference and this can violate the deadline constraint, but also we have periods where the two sources report similar timestamps so in such periods we could use a larger *SLT*



(a) Latency overtime (b) *SLT* parameters overtime (c) *Slack-ready* tuples overtime (d) Detected Events

Figure 12: Comparison of different *SLT* tuning techniques when bus input source is delayed using a Zipf distribution (For results' clarity we added label only on the first figure).



(a) Latency overtime (b) *SLT* parameters overtime (c) *Slack-ready* tuples overtime (d) Detected Events

Figure 13: Comparison of different *SLT* tuning techniques when bus input source is constantly delayed (For results' clarity we added label only on the first figure).

parameter to minimize the number of *slack-ready* tuples that will be processed. In contrast, in Figure 11 it can be observed that the difference between the reported timestamps constantly increases therefore we have to sacrifice determinism guarantees in order to satisfy the deadline constraint.

In the first delay policy we used a Zipf distribution to delay tuples arriving from the bus input source. More specifically, for 20 seconds we delayed the tuples using 1.0 as the exponent with the maximum delay being 10 ms while for the next 50 seconds the Zipf exponent was 0.35 and the maximum delay 2 ms and repeated this delaying pattern throughout the application's execution. The application's deadline was set to 6, 000 ms. In Figures 12(a), 12(b) we illustrate the operator's latency and the *SLT* parameters that were used throughout the experiments. Furthermore, in Figures 12(c), 12(d) we depict the number of *slack-ready* tuples and the detected events. *SLTuner* is able to efficiently satisfy the deadline constraint and at the same time minimizes the amount of *slack-ready* tuples. In contrast, the Static Low approach while it minimizes the latency, it also leads to an increased amount of *slack-ready* tuples that can impact the detected events due to the shift in the operator's sliding window. On the other hand the Static High technique does not have *slack-ready* tuples and detects all the events but at the cost of significant increase in the latency that causes deadline violations during the periods of high delay. The *TCP Adaptation* technique is a re-active technique and thus it adjusts the *SLT* parameter after the deadline has been violated. Nevertheless, it is able to process less *slack-ready* tuples and thus minimize the number of undetected events. Finally, the *Max Timestamp* approach does not process *slack-ready* tuples as it uses the largest possible time difference as the slack threshold parameter; however, this violates the deadline constraint.

In the second delay policy each tuple coming from the bus input source was delayed for 1 ms before it was inserted to the *SSG* data structure. The deadline in this application was set to 35, 000 ms. As it can be observed in this case is not easy to satisfy the deadline (*i.e.,* Figure 13(a)) without increasing significantly the number of *slack-ready* tuples that will be processed (*i.e.,* Figure 13(c)). Furthermore, because the delay between the input sources constantly increases becoming significantly larger than the operator's 5 seconds sliding window, *late-arrivals* will not contribute on the operator's results as the sliding window will shift due to the reading of *slack-ready* tuples and thus no events will be detected (*i.e.,* Figure 13(d)).

SLTuner is able to detect the exact moment that the delay in the input sources will penalize the application's deadline (*i.e.,* at approximately 75 seconds) and determines to decrease the *SLT* parameter to 50, 000 (see Figure 13(b)). In contrast, the *Static High* technique is able to process only ready tuples but it violates the deadline requirement. *Static Low* always satisfies the deadline constraint but even in the first 100 seconds where the constraint would not be violated if we use a large *SLT* parameter; *Static Low* approach processes a large number of *slack-ready* tuples and thus misses more events than the *SLTuner* technique as we illustrate in Figure 13(d). Furthermore, our approach outperforms the other two adaptation techniques as it is able to perform fine-grained tuning of the *SLT* parameter and also it exploits a pro-active mechanism thus it can detect when the deadline will be violated before the violation actually occurs.

7 RELATED WORK

One of the first approaches proposed for the determinism problem in stream processing systems is punctuation based techniques [15]. The basic idea is to use special tuples in the event streams called *punctuations* to indicate that no future tuples will arrive in the stream with timestamp less than the punctuation's timestamp. Most of the existing works assume that punctuations are provided by some external source or can be generated a priori using historical data. However, this is not realistic in many real-world applications. For this reason an adaptive technique [20] has been proposed that

aims to capture the skew between the streams and automatically generate the appropriate punctuations. In order to add this punctuation we have to wait for all the late arrivals, so the problem ends up having this trade-off between latency and results' accuracy.

The second commonly applied technique is using a buffer-based data structure to keep incoming tuples for a time period before processing them. The idea is to keep the data as long as possible to avoid determinism violations. Aurora [1], one of the first stream processing systems, enables the user to set a fixed buffer size. Techniques like [16] propose the dynamic adaptation of the buffer size in order to avoid out-of-order tuples but without examining the implications in the latency. Works like [12, 13] take into account the latency and adjust the buffer size dynamically in order to minimize it but are operator specific (*i.e.,* in [12] they focus on joins while in [13] they examine the problem for aggregations) so they do not provide a generic approach like ours which would work with any operator type. Furthermore, both techniques are reactive while we provide a proactive adaptation enabling us to avoid possible deadline violations before they actually occur.

The third approach for solving similar problems is using speculative techniques [19]. The main idea of such works is to process tuples without any delay and recompute the results in case of out-of-order tuples. However, in many real world applications out-of-order tuples is a common phenomenon so the recomputation cost may affect significantly the system's performance. In [17], the authors combined speculation and buffer based techniques by using a *K-slack* buffer with varying size (*i.e.,* exploiting the TCP adaptation technique described in Section 6) and applying speculation when out-of-order tuples occurred. While this proposal minimizes the operator's latency it may affect its results' accuracy. Finally, the fourth approach for solving this problem is to apply approximation based techniques like [14]. These techniques exploit special data structures (*e.g.,* histograms) to summarize the incoming data and provide results as soon as possible [19].

8 CONCLUSIONS

In this work we have studied the problem of deterministic stream processing under latency constraints. We have advanced state-of-the-art in several ways: (a) We provide a novel data structure, *SSG*, which enables us to trade-off latency for accuracy, (b) We proposed a novel proactive technique for automatically adjusting the slack threshold parameter of *SSG* in upcoming time windows enabling us to avoid deadline violations and at the same time minimizing the amount of *slack-ready* tuples. Our approach exploits the use of *Gaussian Processes* for estimating the impact of the slack threshold parameter on the latency and the amount of *slack-ready* tuples. We use these estimations to determine when deadline violations will occur and solve a single-objective optimization problem in order to minimize the amount of *slack-ready* tuples and satisfy deadline constraints. Our experimental results indicate a clear improvement in the system's performance when our approach is used, outperforming current state-of-the-art techniques.

ACKNOWLEDGMENT

This research has been financed by the European Union through the FP7 ERC IDEAS 308019 NGHCS project and the Horizon2020 688380 VaVeL project, the Swedish Foundation for Strategic Research under the project "Future factories in the cloud (FiC)" (grant GMT14-0032) and the Swedish Research Council (Vetenskapsrådet) project "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures" (grant 2016-03800).

REFERENCES

[1] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB* 12, 2 (2003), 120–139.
[2] Apache Spark. 2017. https://spark.apache.org. (2017).
[3] Apache Storm. 2017. http://storm.apache.org/. (2017).
[4] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. 2008. Fault-tolerance in the Borealis distributed stream processing system. *ACM TODS* 33, 1 (2008), 3.
[5] Daniel Cederman, Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2014. Brief announcement: concurrent data structures for efficient streaming aggregation. In *SPAA, Prague, Czech Republic*. 76–78.
[6] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards Expressive Publish/Subscribe Systems. In *EDBT, Munich, Germany*. Springer, 627–644.
[7] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on* 23, 12 (2012), 2351–2365.
[8] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data* (2016).
[9] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas. 2015. Deterministic Real-time Analytics of Geospatial Data Streams Through ScaleGate Objects. In *DEBS*. ACM, New York, NY, USA, 316–317.
[10] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier.
[11] Yuanzhen Ji, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-Driven Disorder Handling for Concurrent Windowed Stream Queries with Shared Operators. In *DEBS, Irvine, CA*. ACM, 25–36.
[12] Yuanzhen Ji, Jun Sun, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-driven disorder handling for m-way sliding window stream joins. In *ICDE, Helsinki, Finland*. IEEE, 493–504.
[13] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-driven processing of sliding window aggregates over out-of-order data streams. In *DEBS, Oslo, Norway*. ACM, 68–79.
[14] Chuan-Wen Li, Yu Gu, Ge Yu, and Bonghee Hong. 2011. Aggressive complex event processing with confidence over out-of-order streams. *Journal of Computer Science and Technology* 26, 4 (2011), 685–696.
[15] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *VLDB* (2008), 274–288.
[16] Christopher Mutschler and Michael Philippsen. 2013. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *IPDPS, Boston, Massachusetts, USA*. IEEE, 1133–1144.
[17] Christopher Mutschler and Michael Philippsen. 2013. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In *DEBS, Arlington, Texas, USA*. ACM, 147–158.
[18] Carl Edward Rasmussen and Christopher K. I. Williams. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.
[19] Esther Ryvkina, Anurag S Maskey, Mitch Cherniack, and Stan Zdonik. 2006. Revision processing in a stream processing engine: A high-level design. In *ICDE, Atlanta, Georgia, USA*. IEEE, 141–143.
[20] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems. In *SIGMOD, Paris, France*. ACM, 263–274.
[21] Håkan Sundell and Philippas Tsigas. 2003. Fast and lock-free concurrent priority queues for multi-thread systems. In *IPDPS, Nice, France*. IEEE, 609–627.
[22] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS* 31, 1 (2003), 196–205.
[23] Nikos Zacheilas and Vana Kalogeraki. 2014. Real-time scheduling of skewed mapreduce jobs in heterogeneous environments. In *ICAC, Philadelphia, PA, USA*. Usenix, 189–200.
[24] Nikos Zacheilas, Vana Kalogeraki, Nikolas Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. 2015. Elastic Complex Event Processing exploiting Prediction. In *BigData, Santa Clara, CA, USA*. IEEE, 213–222.