CHALMERS
UNIVERSITY OF TECHNOLOGY

TinTiN: Travelling in time (if necessary) to deal with out-of-order data in streaming aggregation

Downloaded from: https://research.chalmers.se, 2026-04-04 22:26 UTC

Citation for the original published paper (version of record):

Van Rooij, J., Gulisano, V., Papatriantafilou, M. (2020). TinTiN: Travelling in time (if necessary) to deal with out-of-order data in streaming aggregation. DEBS 2020 - Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems: 141-152. http://dx.doi.org/10.1145/3401025.3401769

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Industry Paper: TinTiN: Travelling in Time (if Necessary) to deal with out-of-order data in streaming aggregation

Joris van Rooij[*]
joris.vanrooij@goteborgenergi.se
Göteborg Energi
Gothenburg, Sweden

Vincenzo Gulisano
vinmas@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Marina Papatriantafilou
ptrianta@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Abstract

Cyber-Physical Systems (CPS) rely on data stream processing for high-throughput, low-latency analysis with correctness and accuracy guarantees (building on deterministic execution) for monitoring, safety or security applications. The trade-offs in processing performance and results' accuracy are nonetheless application-dependent. While some applications need strict deterministic execution, others can value fast (but possibly approximated) answers. Despite the existing literature on how to relax and trade strict determinism for efficiency or deadlines, we lack a formal characterization of levels of determinism, needed by industries to assess whether or not such trade-offs are acceptable. To bridge the gap, we introduce the notion of D-bounded eventual determinism, where D is the maximum out-of-order delay of the input data. We design and implement TinTiN, a streaming middleware that can be used in combination with user-defined streaming applications, to provably enforce D-bounded eventual determinism. We evaluate TinTiN with a real-world streaming application for Advanced Metering Infrastructure (AMI) monitoring, showing it provides an order of magnitude improvement in processing performance, while minimizing delays in output generation, compared to a state-of-the-art strictly deterministic solution that waits for time proportional to D, for each input tuple, before generating output that depends on it.

CCS Concepts

• **Hardware → Energy metering**; **Smart grid**; • **Information systems → Data streaming**; • **Software and its engineering → Consistency**.

Keywords

Data Streaming, Advanced Metering Infrastructure, Smart Meter

ACM Reference Format:
Joris van Rooij, Vincenzo Gulisano, and Marina Papatriantafilou. 2020. Industry Paper: TinTiN: Travelling in Time (if Necessary) to deal with out-of-order data in streaming aggregation. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401769

[*]Also with Chalmers University of Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401769

1 Introduction

Data stream processing [19] is widely adopted for analysis of continuous streams of data produced in Cyber-Physical Systems (CPSs), for extraction of information useful for the operation, protection and dependability of the systems (e.g., smart meters data validation [20, 21] or vehicular data analysis [8, 12, 22]). Moreover, it is compliant with the needs for decentralized processing and furthermore, the research community is investing significant efforts in encompassing parallelism for stream processing for a large spectrum of devices, from embedded edge units to high-end servers.

Streams consist of sequences of tuples and are unbounded by definition. Therefore one-pass analysis is commonly performed on *windows* of data, whose boundaries change following the time carried by tuples' *timestamps*. A key challenge in processing data from distributed sources resides in its processing order, since the latter can influence the results. Simply put, the results for a certain window of tuples are accurate and can be produced as *deterministic* outcomes, depending on the condition that there are no still-to-be-processed tuples (because of late arrivals) contributing to such window. In this sense, totally ordered streams with no late arrivals simplify the generation of accurate, deterministic results.

Tools such as Viper [22] make sure that results from processing parallel streams are deterministic, by building on sorting techniques. *Relaxed determinism* guarantees are nonetheless desirable and preferable for some applications for which fast (but possibly not accurate) results are more valuable than accurate but late ones [5, 23]. Notice that, sorting of all input data and delaying processing due to few late arrivals, can unnecessarily penalize parts of the analysis that do not depend on late arrivals. An example application is data validation in an Advanced Metering Infrastructure (AMI) system of an electricity grid, to distinguish out-of-range values or value-patterns by Smart Meters (SM) that malfunction [21].

Why existing approaches fall short? Available approaches for relaxed determinism fall short for at least three reasons. First, there is lack of a formal characterization of the possible results produced by a streaming application with relaxed determinism guarantees. Such a characterization is needed by data analysts in order to understand and estimate whether the effects of relaxed determinism are adequate or not for sensitive applications, when the latter's outcomes influence the dependability of a system.

Second, existing approaches that deal with out-of-order tuples are either integrated within a specific Stream Processing Engine (SPE) or require ad-hoc coding to maintain fine-grained control.

The Apache Flink Streaming API [4] and Apache Beam Streaming pipelines [3] are examples of SPE-specific solutions. Both allow for processing of out-of-order tuples by introducing watermarks and multiple evaluations of windows, as discussed in the Dataflow model [2]. However, both require careful considerations about how duplicate or updated results are handled within the query. Enhanced stateful operators [15] or storing and restoring state for late arrivals [17] are other example approaches that, by requiring additional functionality of the SPE, can result in limited usability. Data analysts might not have the option to choose which SPEs should be used and could also lack the advanced programming skills needed to integrate an approach in a given SPE. Avoiding enhanced operators allows the analysts to use any SPE that supports basic aggregation operators.

Third, existing solutions can have prohibitive memory overheads when keeping all data in memory for a given lateness interval, as detailed later in the paper. Hence, their usage in large CPSs, composed of computationally-constrained devices, can also be limited.

Contributions Motivated by these observations, we formalize the concept of *D-bounded eventual determinism* (*D* being a known bound on the timestamp-based out-of-order delay of late input tuples). We also propose TinTiN; *a streaming middleware*, that can top-up the guarantees of aggregation applications that originally ensure determinism for sorted input sequences, to enforce D-bounded eventual determinism for out-of-order input sequences, without requiring modification of the application or the SPE, if the application conforms to a set of assumptions (essentially providing some information about its semantics and the data fed to it; cf. § 3).

TinTiN does not delay results that can be accurately generated when no data is missing, while it "replays" portions of input data, when there are late arrivals. The "replayed" data is fed to an *application's replica*; to prevent the arbitrary time order (and possible overlap) of the relayed data, TinTiN manipulates their timestamps, (hence, their "time travelling") safely and according to the application's semantics. Since data can be "replayed" by TinTiN, some results are not guaranteed to be delivered exactly once. Such a behavior has been proposed in pioneer SPEs such as Borealis [5] (with the introduction of special UNDO or TENTATIVE tuples) and more recently in the Dataflow model [2]. Existing approaches, nonetheless, have large memory requirements since they maintain large windows and also demand that all operators can handle updated results.

In summary, we show that TinTiN enables, without changes on an SPE's operators, (i) timely processing of data, i.e. as soon as it is available, allowing the user to act on preliminary results immediately, (ii) the generation of final results, identical to the ideal case of no late arrivals, as soon as the relevant data has arrived, and (iii) small memory and time overheads, compared with state-of-the-art solutions (e.g. Apache Flink), which can guarantee determinism by processing data when the *D* bound expires (i.e., when late arrivals can no longer be seen), with output latency proportional to *D* and large memory overheads. These properties can make the difference between an approach being impossible and possible to consider in deployments, as we show in the example massive-data industrial use-case on data validation in our evaluation; in particular this use-case has been the key motivation for working on the problem.

The rest of the paper is organized as follows: the preliminaries are covered in § 2 after which we describe our system model in § 3, followed by the formal definition, goals and evaluation metrics of *D*-bounded eventual determinism in § 4. TinTiN's overview and core functionality, including algorithmic design are in § 5 and § 6, while § 7 evaluates TinTiN with our real-world application. Other related work and concluding remarks are discussed in § 8 and § 9.

2 Preliminaries

2.1 Stream processing

In data streaming *applications*, data is processed by *queries*, i.e., Directed Acyclic Graphs of *streams* and *operators*, deployed and run by SPEs. In the remainder, we use the terms query and application interchangeably. Each stream carries *tuples* sharing a schema $\langle ts, A_1, \ldots, A_n \rangle$, where ts is the tuple's *event* timestamp (which carries the notion of time for the query's operators [2]), and $A_1, \ldots, A_n$ are application-specific attributes.

We focus on applications composed by a sequence of one or more *stateful* aggregate operators, as well as by an arbitrary number of *stateless* operators. We use the term *user-defined application* (UDA) to refer to one such application. *Stateful aggregate operators* (also refered to as *Aggregates*) produce results that depend on multiple input tuples (e.g., to compute an average value). *Stateless operators* on the other hand process tuples without maintaining state that depends on the processed tuples (e.g., by filtering tuples based on their values or mapping their input schema to a different output schema). Stateless operators do not change timestamps, as in e.g. Apache Flink [4].

Commonly each Aggregate maintains a *sliding window*, a portion of the recent input tuples, that are processed to deliver results as output tuples. More specifically, we consider that the stateful aggregation of each Aggregate of a UDA is defined over a time-based sliding window W, characterized by its size WS and advance WA, and a set of functions $\{f_1, f_2, \ldots\}$.For example, an Aggregate could maintain a sliding window with WS 2 hours and WA 1 hour to maintain consecutive readings for a smart meter (SM) in order to calculate the hourly consumption by taking the difference between the readings. Notice that different Aggregates of the same UDA can have different size and advance parameters for their windows.

Each Aggregate defines an optional *key-by paramete*r (a subset of the input tuples' schema). If such a parameter is set, the aggregate maintains dedicated windows for each distinct set of key-by values observed in the stream. For the AMI measurements validation example, the input schema is $\langle ts, smID, cc \rangle$, where $smID$ is the ID for the SM and cc is the cumulative consumption that the meter has registered at timestamp ts. For ease of notation, we assume such a key attribute, denoted by k, is defined for all tuples. This does not affect generality, since all input tuples can share the same k value if they are to be aggregated in the same window.

In the remainder, we use the term *window* to refer to the object that is maintained by an Aggregate for each key-by value and evolves according to the tuples being processed, while we use the term *window instance* to refer to the *window covering a specific time interval.* As an example, for an Aggregate with WS and WA set to 1 hour and 30 minutes, respectively, a window instance could refer to the window covering the interval [08:00,09:00), while the subsequent instance is the one covering the interval [08:30,09:30).

Being W the window an Aggregate maintains for a key-by value, each window instance W^i covers the information of all tuples $t_i | t_i.ts \in [W_L^i, W_R^i)$, where W_L^i is the *left boundary* of W^i, and $W_R^i = W_L^i + WS$ is the *right boundary* of W^i. Initially, W^0 covers the first WS-long interval at event time 0. Then, the evolution of window W depends on three methods: `add`, `fire` and `remove`. These methods are invoked by the SPE maintaining W as specified in the following:

S1 Method `add` is invoked for each input tuple $t | t.ts \in [W_L^i, W_R^i)$ and (optionally) used to update the state of functions $\{f_1, f_2, \dots\}$ (if the latter can be updated incrementally).

S2 Method `fire` is invoked as soon as an input tuple $t | t.ts \geq W_R$ is received. Then, the outcome of functions $\{f_1, f_2, \dots\}$ is retrieved and forwarded as an output tuple. The timestamp of such output tuple is set to W_L. Let that tuple be called the *result* of that window instance, denoted by result(W_L).

S3 Method `remove` is invoked immediately after the `fire` method is invoked. All tuples $t | t.ts \in [W_L^i, W_L^i + WA)$ are removed from W and the state of functions $\{f_1, f_2, \dots\}$ is updated accordingly (if they define one). Then, W is shifted forward by WA (i.e., W_L^i and W_R^i are updated to $W_L^i + WA$ and $W_R^i + WA$, respectively), thus moving to window instance W^{i+1}.

Methods `fire` and `remove` are repeatedly invoked, one after the other, until the input tuple t' triggering the invocation of the `fire` method falls within W's left and right boundaries. Continuing the previous example, if the current window instance covers [08:00,09:00) and the next input tuple has timestamp 10:15, methods `fire` and `remove` would be invoked 3 times each, for W to eventually cover [09:30,10:30), to which the input tuple falls in.

We assume method `fire` is only invoked for window instances containing at least one tuple. Hence, no results are initially produced for window instance $W^0, \dots, W^i$, where W^i is the earliest window instance to which the first tuple of a given key falls in.

We assume that all the windows maintained for the different keys observed in the input stream are aligned. That is, if a window for a certain key shifts to a certain $[W_L^i, W_R^i)$ period, so do all the windows of other keys maintained by the application. This is enforced by invoking methods `fire` and `remove` on all existing windows when an input tuple $t' | t'.ts \geq W_R^i$ is processed. Also, we assume that a window for a new key value is created when an input tuple carrying such value is processed and deleted if, after invoking the method `remove` for it, no tuple is left in the window.

Running Example We introduce an example query, based on a real world use case to illustrate the concepts in the paper. Smart Meters (SM) can break down gradually, resulting in unreliable readings. In the example the start of the breakdown can be detected by a *pattern*: one or more large hourly consumption values followed by a negative hourly consumption within four hours. As soon as the pattern is detected for a specific SM, a technician is deployed to replace the SM. The pattern is identified by a query consisting of two Aggregates. Aggregate 1 calculates the hourly consumption by taking the difference between every two consecutive readings. It therefore has a *WS* 2 hours and *WA* 1 hour. Aggregate 2 has *WS* 4, *WA* 1 hour, and produces an alert if the pattern is matched. The value of the alert is the number of large consumption values in the match. Figure 1 shows a "cross-section" of an execution of the query and the results it produces for an input stream consisting of 10 tuples for a single SM.

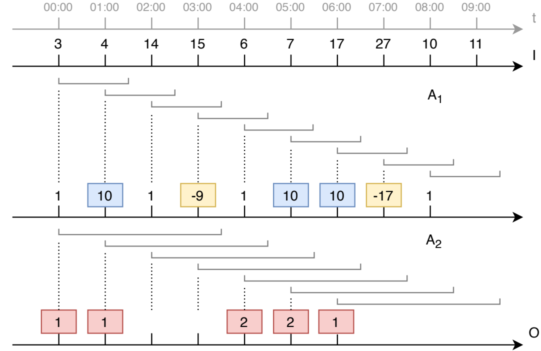


Figure 1: ***Example "cross-section" of a query execution, processing data for one Smart Meter. The input stream I shows the readings, while their timestamps are given on the time axis t. The query has two aggregate operators. A_1 calculates the hourly consumption by taking the difference between two consecutive readings. A_2 outputs alerts if a pattern is found in its window. The pattern is one or more large consumption values (marked in blue) followed by a negative value (marked in yellow) within four hours. The value of the produced alerts (in red) indicates the number of large values in the match. A_1 has WS 2 and WA 1, while A_2 has WS 4 and WA 1. The timestamp of results is equal to the left (inclusive) boundary of the window instance that produced it. The right boundary of a window is exclusive.***

2.2 Strict determinism

The execution of a stateful operator is deterministic if the operator's semantics are correctly enforced, independently of (i) the operator implementation and deployment and (ii) the input data ordering. For instance, when running an aggregate operator counting tuples on a per-key basis over a window with *WS* and *WA* set to one hour, if 5 tuples referring to key k and having timestamps $\in [08{:}00, 09{:}00)$ are delivered in the input stream, the Aggregate's execution is deterministic if the operator produces the correct count for key k and window $[08{:}00, 09{:}00)$ independently of whether (i) the operator is run sequentially by one thread or in parallel by several threads (e.g., assigning each thread a subset of keys) and (ii) the input tuples are delivered in timestamp order or not to the thread(s) running the Aggregate's analysis.

For a *single-threaded aggregate operator* whose sliding window's execution evolves over time upon the invocations of methods `add`, `fire` and `remove` as described in § 2.1, it is known from the literature that deterministic execution is enforced if:

(1) Functions' $\{f_1, f_2, \dots\}$ analysis uses no randomness and depends exclusively on input tuples' attributes, and
(2) Input tuples are fed in timestamp order.

If these conditions hold, the method `fire` is invoked for each window W^i only after the method `add` is invoked for all the tuples contributing to W^i, and each output tuple depends exclusively on the values carried by the tuples contributing to it (including the timestamp, which determines the order in which tuples are added

to the window). For a *multi-threaded aggregate operator*, in which distinct threads carry out the analysis of different keys, the above set of sufficient conditions to imply determinism, are commonly complemented with the following one:

(3) Exactly one of the threads running the analysis in parallel is responsible for the analysis of a given key.

This is a sufficient condition to prevent inconsistencies of state updates in the analysis. It can be possible to prevent the latter with other methods, however this is a common one practice.

Regarding guaranteeing *determinism for a UDA*, a *sufficient condition* is to ensure (i) determinism on the operator level and (ii) timestamp-ordering in the data flows to the operators, including the internal, inter-operator ones [6, 14, 22]. In the following, we say that a streaming aggregation application enforces *strict determinism* if such a condition is met. We use the term strict to differentiate the determinism from the relaxed one we propose here.

3 System Model

Here we specify some more detail about the type of the User-Defined Applications (UDAs, serial aggregation applications) targeted. We assume the UDA is fed with one input stream and it can run in parallel the analysis of different *keys*. We assume that strict determinism is guaranteed for the UDA by guaranteeing determinism for all the operators composing the UDA and the inter-operator flows, as explained in § 2.2. We wish to note that each *result* t_o of the UDA, given that it is a tuple that bears the timestamp of the last aggregate in the UDA, can be uniquely indexed by that timestamp (this is due to the fact that output tuples of operators are timestamped using the left boundary of the window instance they correspond to, as mentioned in S2 in § 2).

We also require the following to hold (we refer the reader to § 6 for further discussions and the justification of these assumptions):

A1 By observing the last two tuples t_A, t_B received for a certain key k s.t. $t_B.ts > t_A.ts$, it is possible to know whether a *hole* exists, i.e. there exists a tuple with timestamp $\in (t_A.ts, t_B.ts)$ that can either arrive late or not at all. This is the case, for instance, when the input data sampling period is known or when an enumerator attribute is defined for the input data.

A2 A known maximum out-of-order delay D allows to distinguish late arrivals that can still be received, from those that will not (i.e., that can be ignored). More concretely, given any arbitrary point in any arbitrary execution, being τ the highest timestamp received by the application, late arrivals with timestamps $\in [\tau - D, \tau]$ can still be received, while those with timestamps smaller than $\tau - D$ cannot (i.e., they can be ignored).

A3 Analysis of data might be related to its seasonality, i.e. results could differ depending on e.g. the hour of the day, or the day of the week. We define the **periodicity** P of a UDA as the maximum period that is relevant for the seasonality of the data; e.g., P is 24 hours if tuples are treated different depending on the time of day of their timestamp, while it is one week if treatment also depends on the day of the week. If the analysis of the UDA is not related to the seasonality of the data, we consider $P = 1$, else, we assume that its P is known.

A4 A sorted sequence of tuples, denoted RC_{flush} that triggers at least one output tuple is made available to TinTiN.

A5 The sequence of stateless and aggregate operators composing the UDA is known, as well as the finite window sizes and advances of all the aggregate operators in the UDA.

Note that the UDA developer who wants to use TinTiN to deal with out-of-order input data will have all the information required.

4 D-bounded eventual determinism

As mentioned in § 1, outputting information in a timely fashion is useful or critical in certain applications. We propose *D-bounded eventual determinism* to formalize guarantees that enable timeliness of output that depends on timely available input, while loosening only part of the requirements, compared to strict determinism.

Definition 1. *Given a stream I that is not timestamp-sorted, we say that I is* **within lateness bound** *D if, for any t in I, for all subsequent tuples t' in I for which $t'.ts < t.ts$, the condition $t.ts - t'.ts \leq D$ holds.*

Definition 2. *Given:*

- *A, a streaming application that supports deterministic execution for timestamp-sorted streams,*
- *I, a timestamp-sorted input stream,*
- *O, the output stream produced by A when all input tuples from I are fed to A, and*
- *E, the set of all possible executions in which A is fed with a permutation of I that is within lateness bound D (Definition 1);*

*we say that A is extended to a D-***bounded eventually deterministic** *streaming application A' if, for A, A' and any $e \in E$, being:*

- *$o_{ts,k} \in O$ the tuple output by A for timestamp ts and key k,*
- *$o'^{1}_{ts,k}, \ldots, o'^{n}_{ts,k}$ the ordered sequence of output tuples produced by A' for timestamp ts and key k (with $n \geq 1$),*

then $o_{ts,k} = o'^{n}_{ts,k}$.

Definition 2 says that an eventually deterministic streaming application A' (derived from A that is strictly deterministic when it processes timestamp-ordered tuples), processing a stream within lateness bound D, produces all tuples that A produces when it processes the same input but sorted. A' might produce more tuples than A, but for every tuple produced by A, the latest tuple with identical timestamp and key produced by A' will be equal. Note that, if A can produce multiple output tuples for a specific timestamp and key, then these should be distinguishable, by e.g. another attribute.

Evaluation metrics When comparing the results observed for a streaming application A' with relaxed deterministic guarantees (when fed with an input stream I that might be unsorted), with those of a UDA A that enforces strict determinism for a sorted version of I, we say an output tuple produced by A' is (i) **exact** if an output tuple carrying the same attribute values (for the same timestamp and key k) is produced by A, (ii) **duplicate** if another, exact tuple (for the same timestamp and key k) has already been produced by A', or (iii) **different** if an output tuple with different attribute values for the same timestamp and key k (i.e. not exact) is produced by A. (iv) It is also possible to have tuples **omitted** by A', i.e. tuples produced by A but not by A'.

Note that if A' enables D-bounded eventual-determinism for an input stream I that is within lateness bound D, no omitted output tuples exist and, for each different output tuple (if any), one

or more exact output tuples are also later produced for a given timestamp and key k. If, on the contrary, A' does not enable D-bounded eventual determinism, both omitted and different output tuples not followed by at least an exact output tuple can be observed.

Example continued: Recall the running example where occurrences of a specific pattern indicating Smart Meter hardware failure are identified. Figure 1 shows the query processing data for a single SM where the pattern occurs twice. Figure 2 shows the same example, but with two missing input tuples. The first occurrence of the pattern is not identified when this input is missing, the alerts from this occurrence are *omitted*. The second occurrence is only partially affected by the missing input and is identified. The output with timestamps 04:00 and 05:00 is *different*. The output for timestamp 06:00 does not depend on the missing data and is *exact*. If exact output is produced once more, for example by processing the relevant data at a later time when the missing tuples have arrived, it would instead be *duplicate*. The figure also illustrates one of TinTiN's advantages: by being able to continue the processing even if data is missing, TinTiN identifies the second occurrence of the pattern without waiting for the missing data to arrive. A technician can be dispatched to the affected SM immediately after identification, minimizing the amount of unreliable data sent by the SM.

Let us also define the **logical latency** of an output tuple t, the difference between $t.ts$ and the highest timestamp observed in the stream when the exact result for t is produced.

Note that for each output tuple t, if strict determinism is enforced by simply postponing the processing of each input tuple by D time units from its timestamp (e.g., as done by Apache Flink's Complex Event Processor) its logical latency is D, while it can be made (significantly) smaller than D, by leveraging finer-grained techniques for managing out of order data, as we show with TinTiN, when enforcing D-bounded eventual determinism.

Similarly to logical latency, **responsiveness** is defined as the difference between the highest timestamp observed in the stream when a late tuple t_l arrives and the highest timestamp in the stream when the final exact result for t_l is produced. In the case where strict determinism is enforced by postponing processing as described above, the responsiveness will be D minus the lateness of t_l, while it can be made smaller by TinTiN just as the logical latency.

5 TinTiN's overview

Here we give an overview of TinTiN, while in the subsequent section we describe its core design and its algorithmic implementation.

TinTiN processes data even though some tuples are late. Our pattern matching example with Smart Meter data shows that this allows some matches to be identified despite missing data. Figure 2 illustrates this, the alerts with timestamps 04:00, 05:00 and 06:00 are produced. The alerts from 04:00 and 05:00 have a different value compared with the alerts produced when no data is missing. The implications of such differences are application specific. In this particular example a technician will be deployed regardless of the value of the alert, so there is no direct implication. In order to eventually deliver the exact output, TinTiN *replays* portions of data when late data arrives. Such portions are processed by a copy of the UDA which produces updated results. This is illustrated in Figure 4 where a portion of data is replayed. The extra alerts due to replaying might cause another technician to be dispatched; i.e. the dispatcher has to take extra care when such results are produced.

Considering the example, let us proceed with the description of the middleware: TinTiN does not delay results that can be accurately generated in the cases of no missing data. When intervals of data with missing tuples have been processed, as also mentioned in the example, it later *replays* sufficient portions of the input, when late data arrives. Moreover, it aims at achieving the aforementioned behaviour efficiently, both from the point of view of computational and memory overheads, as well as from the perspective of limiting the amount of "unnecessary", partial, results. Furthermore, it works as a wrapper of the UDA in any SPE, without requiring to modify the internals of the latter. The following list of steps and Figure 3, outline at a high level the aforementioned procedures.

(1) TinTiN forwards to the UDA the input tuples that arrive in increasing timestamp order.
(2) TinTiN also temporarily maintains a sufficiently large portion of the input stream that initially contained some *hole(s)* (caused by tuple(s) being late), named **relevant context** of the holes(s).
(3) Later, if late tuples arrive within the bound D, TinTiN "replays" the relevant context of the respective hole(s), to get refined results. To avoid interference with the processing of the in-order data, the replayed data is fed to a *replica of the UDA* (*UDA_R*; c.f. Fig 3).
(4) To prevent that the arbitrary time-order and the potential overlap of relevant context of late tuples to affect consistency of results, when replayed at *UDA_R*, TinTiN shifts forward by a given offset all the timestamps of each relevant context (hence, "time travelling"), safely and according to the application's semantics and shifts back the final results' timestamps to the original ones when forwarding those results to the user.

Why can TinTiN guarantee eventual determinism? The tuples that arrive in timestamp order and without holes, generate the same result as the strictly deterministic case when fed to the UDA. If a portion of input forwarded to the UDA contains a hole though, such a portion is also sent to the UDA_R (at least once) when holes are later filled in (with tuples arriving with maximum lateness D). This implies that multiple and possibly different, improved, versions of the same result could be delivered to the user. However each result will be based on a relevant context of hole(s) that is gradually filled in by late tuples. Following this procedure, TinTiN eventually delivers the output that would be generated in the ideal case (in which there would have been no late arrivals) for the respective window instances, i.e. it satisfies D-bounded eventual determinism when fed with a stream that is within bound D.

How is TinTiN's processing safe and consistent? The aforementioned 4 steps need to be carried out in order to make sure that safety in processing is preserved, i.e. that state created by the processing of different portions of data (overlapping intervals, intervals that are forwarded out of order) is not inconsistently mixed up when data is being replayed. This is achieved by replication, in a two-folded fashion: (i) for separating the processing of replayed tuples from the processing of in-order tuples, the replaying of the relevant context of holes are carried out by UDA_R and not by UDA; (ii) for avoiding UDA_R to mix up replaying of overlapping relevant

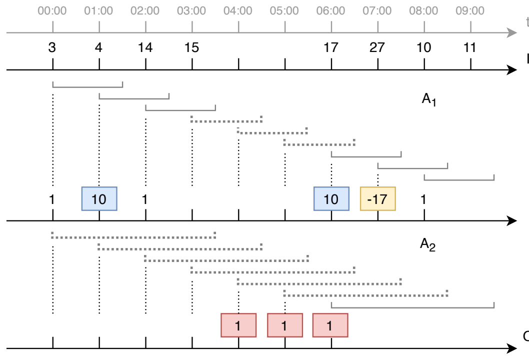


Figure 2: ***The same "cross-section" as seen in Figure 1, but with readings from 04:00 and 05:00 missing at the time of processing. All window instances that are affected by the missing input are dashed.***

contexts of holes, the latter are replayed with modified time, which is modified back to the original, when the result is produced.

For TinTiN to efficiently carry out the above, the following questions need to be answered, as explained in the next section.

Q1 Which results can be improved and forwarded to the end-user? In case of late data, it is straightforward to identify such results for a query composed by a single aggregate operator, but there is more to think about for arbitrary UDAs.
Q2 What relevant context to replay? For each result that can be improved, how to identify which source data is sufficient to maintain in memory, in order to replay if/when late tuples arrive?
Q3 How to replay efficiently? If, due to one or more late tuples, TinTiN needs to re-produce multiple results, can it do it efficiently without replaying many times overlapping relevant contexts?
Q4 How to replay safely? How to ensure that the "time travelling" is correct, i.e. that the processing state of the UDA_R does not get mixed up with that of other replayed data?

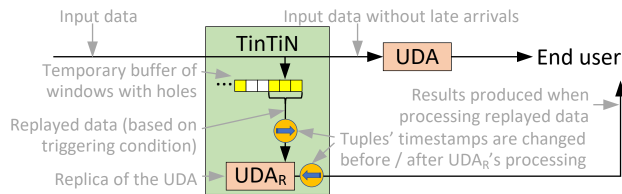


Figure 3: ***Overview of TinTiN's architecture.***

6 TinTiN's core functionality

This section covers TinTiN's design and its "*time travel*" mechanism, answering the questions of the previous section. Then, it presents TinTiN's algorithmic implementation and the main argument for satisfying D-bounded eventual determinism.

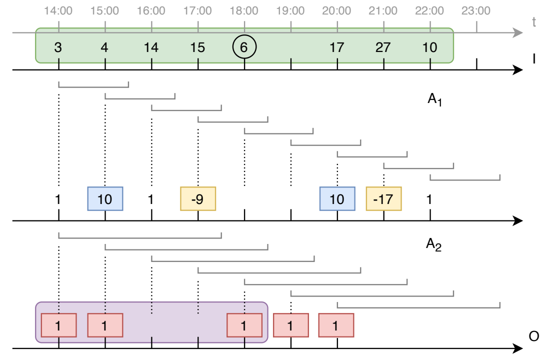


Figure 4: ***Example "cross-section" of an execution at the UDA_R showing the relevant context of a hole (§ 6.2, Figure 2), in green, as it will be replayed, when the hole is filled by the encircled late arrival. The timestamps of the input data have been shifted by TinTiN (§ 6.4), e.g. timestamp 00:00 in Figure 2 is shifted to 14:00. The output affected by the late arrival, i.e. that can be improved by the late arrival (§ 6.1), is shown in purple. For example, the alert with timestamp 14:00 could not be produced without the late arrival, cf. Figure 2.***

6.1 Answering Q1: Which results can be improved and forwarded to the end user

A hole in the input stream of a UDA can affect multiple results, as shown in Figure 2. The quality of such results can be improved by reprocessing the relevant context of the holes once the late tuples have arrived. Let us first determine the results that are *affected* (i.e., that can be improved) by late values for a UDA with a *single Aggregate*:

LEMMA 1. *Given an Aggregate A with window size WS, the timestamps of A's results that are potentially affected by a missing input tuple t with timestamp ts, are in $(ts - WS, ts]$.*

PROOF. The timestamp of any affected result, potentially improvable by a late input t, equals the left boundary of any window instance that contains t (cf. S2 in § 2). The window with the earliest left boundary that contains t starts no earlier than $ts - WS$. The window with the latest left boundary containing t cannot start after ts, since the left boundary of a window is inclusive and windows with a left boundary larger than ts do not include t. □

Consider again the *example* in Figure 2 with two Aggregates. A hole in the input stream affects results from A_1, which in turn affects more results from A_2. An interval that contains the timestamps for all of the affected results is given by the following lemma.

LEMMA 2. *Consider a UDA with n Aggregates in series with window sizes $WS_i; i \in [1, n]$, and a missing input tuple t with timestamp ts. The timestamps of the UDA results that are potentially affected due to the missing input are in $(ts - \sum_i WS_i, ts]$.*

PROOF. The timestamps of the affected results of the first Aggregate A_1 are contained in the interval $(ts - WS_1, ts]$ (Lemma 1). The same argumentation, applied for the second Aggregate A_2 and for $ts - WS_1$ and ts, allows to find the bounding interval for timestamps of A_2's affected results, implying the interval $(ts - (WS_1 + WS_2), ts]$.

The same reasoning applied recursively for any subsequent Aggregate, results in the interval in the lemma statement. □

It should be noted that Lemma 2 implies:

Observation 1. *The results to forward to the end user when data is replayed due to a late input tuple with timestamp* ts*, are the ones with timestamps in* $(ts - \sum_i WS_i, ts]$.

This is illustrated in Figure 4 where the relevant context for the late arrival is *replayed*. The updated results to forward are marked, while results outside the interval in Observation 1 (i.e. not to be forwarded) are removed from the output stream.

6.2 Answering Q2: Sufficient input to replay

To determine the exact content of the sufficient relevant context of a hole, we need to find all input tuples that are relevant to the potentially affected results, so that those tuples are stored and replayed together with late tuples if/when the latter arrive.

Lemma 3. *Consider a* result tuple t *with timestamp* ts*, produced by an Aggregate* A *with window size* WS*. The timestamps of all input tuples to* A *that are relevant for (i.e. potentially affect)* t *are in* $[ts, ts + WS)$.

Proof. The lemma follows directly from the fact that a result with timestamp ts is produced by a window instance whose inclusive left boundary is ts and exclusive right boundary is $ts + WS$. □

Figure 4 shows the *running example*, marking all relevant input tuples for a set of results from the example-UDA. The bounding interval for these tuples follows from the following lemma.

Lemma 4. *Consider a UDA with* n *Aggregates in series with window sizes* $WS_i; i \in [1, n]$*, and a* result tuple t *with timestamp* ts*. The timestamps of all input tuples to the UDA that are relevant for* t *are in* $[ts, ts + \sum_i WS_i)$.

Proof. Lemma 3 implies that the timestamps of the relevant input to the first Aggregate are in $[ts, ts+WS_1)$. The same argumentation can be applied for the second Aggregate for ts and $ts + WS_1$, to find the bounding interval for timestamps of affected results from the second Aggregate, implying the interval $[ts, ts + (WS_1 + WS_2))$. The same reasoning can be applied recursively for any subsequent Aggregate, resulting in the interval in the lemma statement. □

Combining the lemmas from § 6.1 with lemmas 3 and 4, we get:

Lemma 5. *Consider a UDA with* n *aggregate operators in series with window sizes* $WS_i; i \in [1, n]$*, and a* hole *in the input stream with timestamp* ts*. The relevant context of the hole is contained in the interval* $(ts - \sum_i WS_i, ts + \sum_i WS_i)$.

This is illustrated in Figure 4 for the *running example* UDA and the relevant context of the encircled late arrival.

6.3 Answering Q3: Replaying efficiently

The logic with which the relevant context*s* of late tuples, temporarily stored at TinTiN, are replayed, depends on a user-defined *triggering condition TC*. TCs can imply a trade-off between different properties, e.g. between efficient processing and how fast a result for a late arrival is produced, as explained in the following.

An *eager TC* could trigger the replay of a relevant context for a hole as soon as the late arrival filling it is received. Such a condition, reacting immediately to each late arrival, minimizes the time between receiving the late arrival and producing potential results to which it contributes. This could be beneficial for applications that need up-to-date (possibly different) results as soon as possible.

Alternatively, a *lazy TC* could instead trigger the replay of a relevant context of hole(s) for a certain key k when multiple holes have been filled. For use cases where late arrivals arrive in batches, this can be achieved by delaying the firing of the trigger until an on-time tuple is observed for k. Such TC trades increased logical latency for better processing throughput. Note that it is possible to construct a TC that favors efficient processing even more by waiting even longer before triggering. More efficient processing is achieved by combining the relevant context for multiple late arrivals where it overlaps. This is possible due to the associative property of the relevant context, shown here:

Lemma 6. *Consider a UDA with* n *aggregate operators and two holes with timestamps* ts *and* $ts + x$*, for any* $x > 0$ *s.t. there is overlap in their respective relevant contexts. The set of affected results produced by replaying each relevant context separately is equal to the set of affected results produced by replaying the union of the relevant contexts once.*

Proof. The affected results produced by replaying the relevant context for the late arrival with timestamp ts have timestamps in the interval $(ts - \sum_i WS_i, ts]$ according to Lemma 2. Results for the late arrival with timestamp $ts + x$, have timestamps in interval $(ts+x-\sum_i WS_i, ts+x]$. The size of the relevant context for the late arrival with timestamp ts is $(ts - \sum_i WS_i, ts + \sum_i WS_i)$, according to Lemma 4. Since the relevant context*s* of the two late arrivals overlap, $x < \sum_i WS_i$. Therefore $ts + x - \sum_i WS_i < ts$, implying that intervals for the affected results overlap and are contained in the interval $(ts - \sum_i WS_i, ts + x]$. This interval is equal to the interval obtained for the affected results by the union of the relevant context for ts and $ts + x$. □

Special consideration is due for holes that are filled after a *short amount of time* (ie. before their entire relevant context has arrived), while to obtain eventual determinism, it is required to replay *all* tuples that contribute to a result that can be improved by a late arrival. One way to ensure this, is to delay replaying until a tuple with timestamp larger than the largest timestamp in that relevant context arrives.

6.4 Answering Q4: Replaying safely

Replaying the relevant context of a late arrival directly to the UDA would cause the input stream of the UDA to become out of order and interfere with data currently being processed. For this reason, TinTiN replays data to a replica of the UDA, UDA_R. Note that UDA_R, being a replica of UDA, guarantees deterministic processing only for timestamp sorted input. However, replaying the relevant context of two different holes can be problematic for two reasons:

(1) It causes the input stream to UDA_R to become out of order if the second relevant context starts with a timestamp lower than the highest timestamp of the first relevant context.

(2) It can cause erroneous results if the data for the different late arrivals ends up in common windows.

Intuitively, a straightforward solution to prevent this from happening, is to deploy a "fresh" UDA_R (i.e., that has not yet processed any tuple) before replaying any past portion of input tuples. Deploying a fresh UDA_R incurs large overhead though. Relying on a method that resets the UDA_R's state is also not an option, since, for the sake of generality, we do not assume that the SPE or the UDA's programmer provide it.

TinTiN's novel approach is to (i) shift the timestamps of the tuples in the relevant context with an offset, so that UDA_R is fed with an input stream with strictly monotonically increasing timestamps, and tuples from one replay cannot interfere with other replays; and (ii) shift back the timestamp of the result by the same offset.

6.4.1 Shifting timestamps of input to UDA_R When processing data in the UDA, an input tuple will belong to a specific number of window instances for the first Aggregate. The same is true for a tuple produced by the first Aggregate, it will belong to a specific number of window instances for the second Aggregate and so on and so forth. The number of window instances a tuple belongs to depends on its timestamp as well as the window advance and size of the Aggregate. An example of this can be seen in Figure 1, where the output of the first Aggregate belongs to either one or two window instances of the second Aggregate.

In order to obtain correct results when reprocessing data, it is required that every tuple contributes to the same number of window instances as for the on-time processing. The way the tuples timestamps can be changed without affecting the number of window instances they contribute to is based on the following observations:

Observation 2. *Consider a UDA with n aggregate operators in series with window advances $WA_i; i \in [1, n]$. The starting times of the windows for all aggregate operators are aligned at time 0, since all windows start in 0. Such alignment also occurs at timestamps that are a multiple of $LCM(WA_i)$ (the least common multiple of all WA_i).*

Observation 3. *Consider a UDA and an input tuple t with timestamp ts. Let d denote the difference between ts and the nearest alignment of windows in the UDA smaller than ts. Now consider another tuple t' with timestamp ts', with difference equal to d between ts' and its nearest alignment of windows smaller than ts'. The number of window instances that t contributes to is equal to the number of window instances that t' contributes to.*

Consider for example a query with two Aggregates, the first one having *WS* 3, *WA* 2 and the second one *WS* 3, *WA* 1. Since subsequent windows for the first Aggregate overlap one hour, input tuples can contribute to either one or two window instances. Window alignment occurs every multiple of 2 ($LCM(1, 2)$). The tuples with timestamps 5 and 7 both have distance 1 to their nearest window alignment and both contribute to two window instances.

We conclude from the observations that when changing timestamps, any multiple of $LCM(WA_i)$ can be added to the original timestamps to ensure that all tuples contribute to the same number of window instances in UDA_R and the on-time UDA.

If the UDA has an internal periodicity P as described in assumption A3, cf. § 3, then this should be taken into consideration as well when shifting timestamps of data to replay. Tuples should not only contribute to the same number of window instances, but the periodicity should be preserved as well. For example if a UDA has P of one week, a tuple with timestamp 12:00 on a Monday should also have a timestamp 12:00 on a Monday after the timestamp is changed. This is accomplished based on the following observations:

Observation 4. *The periodicity P of a UDA is conceptually the same as a window with WS and WA equal to P. In other words: a new period starts at every multiple of P and has a duration of P.*

Observation 5. *Consider a UDA with n aggregate operators in series with window advances $WA_i; i \in [1, n]$ and periodicity P. The windows for all aggregate operators, as well as the start of period, are aligned at time 0, since all windows start in 0, as does the periodicity. Alignment also occurs at timestamps that are a multiple of $LCM(WA_i, P)$ (the least common multiple of all WA_i and P).*

The final consideration when shifting timestamps is that one sequence of replayed tuples should not interfere with another sequence of replayed tuples. This is achieved when the sets of results are produced by the sequences are disjoint. This can be accomplished by separating the timestamps from both sequences with a *safety distance, SD*. The size of SD is given by the following lemma, which follows directly from Lemma 2.

Lemma 7. *Given a UDA with n aggregate operators with window sizes WS_i and two sequences of tuples S_1 and S_2 where all timestamps in S_2 are larger than any timestamp in S_1, no results are affected by tuples from both S_1 and S_2 if the smallest timestamp in S_2 is separated from the greatest timestamp in S_1 by at least* $\mathrm{SD} = \sum_i WS_i$.

In conclusion, the following lemma justifies how timestamps can be shifted in a way to guarantee safety in the processing.

Lemma 8. *Consider a UDA with periodicity P and n aggregate operators in series with window advances $WA_i; i \in [1, n]$, processing two tuple-sequences S_1 and S_2. Adding $z \cdot LCM(WA_i, P)$ to the timestamps of the tuples in S_2, where $LCM(WA_i, P)$ is the least common multiple of all WA_i and P, and $z \in \mathbb{Z}$ so that $z \cdot LCM(WA_i, P) > SD$, guarantees that (1) all tuples in S_2 still contribute to the same number of window instances as they would have, had their timestamps not been changed, and (2) no results are produced that are affected by tuples from both S_1 and S_2.*

Proof. Property 1 follows directly from Observation 5. Property 2 follows directly from Lemma 7. □

6.4.2 Restoring timestamps of UDA_R results Results produced by the UDA_R cannot be forwarded to the end user without restoring the timestamps, for obvious reasons. Therefore TinTiN should store a *mapping* of the changed and original timestamps in order to restore the timestamps for produced results. There is no guarantee that replaying a relevant context to UDA_R will produce any results. Mappings that were stored by TinTiN can therefore become *stale*. Whether a mapping is stale or not cannot be inferred by setting a timeout for the result of a relevant context, since the UDA processing latency is outside TinTiN's control.

Notice that, since UDA_R supports deterministic execution and is fed a timestamp-sorted stream (based on TinTiN's manipulation of timestamps), it results in a timestamp-sorted output stream (cf.

§ 2.2). Hence, if a replayed sequence S results in output, stale mappings of changed and original timestamps for sequences replayed earlier can be safely discarded by TinTiN, since results for them will not be produced after S's.

To prevent the size of changed and original timestamp mappings to grow beyond a maximum size, TinTiN can replay the sample sequence of tuples that is known to trigger an output (assumption A4 in § 3) to flush stale mappings (in this case, without forwarding the result to the UDA user).

6.5 Synthesis: TinTiN's algorithmic design

Table 1: *Abbreviations and parameters used in Algorithm 1*

Parameter	Definition
UDA, UDA_R	User-Defined Application and TinTiN's replica of it
$WA[]$	All *WA* in the UDA
D	Max delay on UDA's input tuples
TC	Triggering condition
SD	Safety distance between successive replays by UDA_R as defined in § 6.4
P	UDA's periodicity as defined in § 3
$\beta[]$	TinTiN's internal array of sorted tuples (one array per key k, used to store the most recent tuples)
$B[]$	TinTiN's internal array of sorted tuples (one array per key k, used to store tuples contributing to the relevant contexts of holes)
$T[]$	TinTiN's internal array of timestamps and corresponding manipulated timestamps
$\lambda[]$	TinTiN's internal array for late arrivals
max_ts	highest timestamp seen so far
max_ts_{man}	highest manipulated timestamp replayed to UDA_R
RC_t	The relevant context for tuple t
RC_{flush}	Sample data that triggers an output from the UDA

Here we focus on the algorithmic description of TinTiN, also shown in Algorithm 1 (based on the abbreviations and parameters listed in Table 1). For each key k, each input tuple t with a timestamp greater than or equal to that of the previous input tuple observed by TinTiN is forwarded to the UDA. When t is observed, TinTiN could identify t as part of the relevant context of a previously observed hole, if such hole is within time-distance $\sum_i WS_i$ from t (Lemma 5). Even if no such hole has been observed, t could still turn out to be part of the relevant context for a hole later observed within time-distance $\sum_i WS_i$ from t (Lemma 5). To be efficient, TinTiN aims at maintaining t only if t is part of at least one relevant context. To do this, TinTiN initially adds each new incoming tuple that is not a late arrival into a key-dedicated map $\beta[t.k]$ of size $\sum_i WS_i$. If a hole is observed while t is in $\beta[t.k]$, then t is part of a relevant context that could be replayed in the future. Only in this case, t is moved to a larger key-dedicated map $B[t.k]$, in which relevant contexts are kept as long as a late arrival within bound D could still be received (L5). If t is a late tuple, but it is no more than D time units late compared to the highest timestamp observed so far, t is added to both $B[t.k]$ (L7) and $\lambda[t.k]$, a map that stores late arrivals. If t is more than D time units late, it is discarded.

Subsequently, the triggering condition is checked for all late arrivals in $\lambda[t.k]$. For each late arrival for which the triggering

Algorithm 1: TinTiN's algorithm, upon receiving tuple t

1 $max_ts = \max(max_ts, t.ts)$;
2 **if** $t.ts \geq max_ts$ **then**
3 | forward t to UDA and add t to $\beta[t.k]$;
4 | **if** $\beta[t.k]$ *contains holes* **then**
5 | | add $\beta[t.k]$ to $B[t.k]$ (excluding duplicates);
6 **else if** $t.ts > max_ts - D$ **then**
7 | add t to $B[t.k]$;
8 | add t to $\lambda[t.k]$
9 **for** *all* t^i *in* $\lambda[t.k]$ *for which TC holds* **do**
10 | `replay` (RC_{t^i});
11 **if** $\exists t^1$ *in* $\lambda[t.k]$ *so that* $t.ts < t.ts - D$ **then**
12 | `replay` (RC_{t^i});
13 **if** $\exists t^i$ *in* $B[t.k]$ *so that* $t^i.ts < t.ts - (D + size(RC_t))$ **then**
14 | remove t^i from $B[t.k]$;
15 **if** *size*$(T[\,])$ $>$ *threshold* **then**
16 | `replay` (RC_{flush})
17 **replay**(RC_t)
18 | $ts_{min} = min(ts \in RC_t)$;
19 | $ts_{max} = max(ts \in RC_t)$;
20 | $M = LCM(WA[\,], P)$;
21 | find min $(z \in \mathbb{Z}) : ts_{min} + z \cdot M > max_ts_{man} + SD$;
22 | shift RC_t tuples' timestamps with $z \cdot M$;
23 | **for** *results affected by* t **do**
24 | | store $t.ts, t.ts_{man}$ pairs in $T[\,]$;
25 | $max_ts_{man} = ts_{max} + n \cdot M$;
26 send RC_t tuples to UDA_R in timestamp-order; get results $t_0[\,]$;
27 flush old state from $T[\,]$;
28 **if** $RC_t \neq RC_{flush}$ **then**
29 | shift $t_0[\,]$'s timestamps back;
30 | forward $t_0[\,]$ to output ;

condition TC holds (cf. § 6.3), method `replay` (L21-30) is invoked. In this case, RC_t, i.e. the tuples in the relevant context for the late arrivals from $B[t.k]$ are forwarded to UDA_R once their timestamp is changed, while respecting the periodicity P of the UDA as well as the window advances of the UDA, as described in § 6.4.

Manipulated timestamps for which an affected result can be produced by the UDA are paired with the original timestamps and stored in $T[]$, to accommodate shifting back the timestamps. If the size of $T[]$ exceeds a pre-defined threshold, RC_{flush} is replayed to remove stale mappings from the array (L16), as described in § 6.4. If results are produced by UDA_R, from any RC_t that is not RC_{flush}, the timestamps of the results are then moved back and the results are forwarded to the end user. All tuples in $B[]$ contributing to a relevant context for which late arrivals will not be received (based on D) are replayed to UDA_R and then removed from $B[]$ (L14).

Lemma 9. *Given a UDA that supports deterministic execution for timestamp-ordered input streams, Algorithm 1 guarantees D-bounded eventual determinism when the input is within lateness bound D.*

Proof. Based on the questions in § 6, we need to ensure that (i) all relevant contexts for all D-bounded late arrivals are stored; (ii) after all late arrivals have arrived, the relevant contexts are forwarded in timestamp-order at least once to UDA_R (which will run method `fire` for all relevant window instances); (iii) when

forwarding relevant context, all tuples in it neither precede other UDA_R-maintained tuples nor contribute to any of the windows maintained by UDA_R (once the timestamps of the tuples in the relevant context are changed). Algorithm 1 implies this is achieved since (i) all late arrivals are stored together with other tuples in the relevant context they contribute to (L1-5 and L7) as described in Lemma 4, (ii) tuples are removed from $B[t.k]$ only when no more late arrivals will be received for the relevant context they belong to (L14) and (iii) method `replay` is run at least once (after all possible late arrivals have been added to it in timestamp order) once its timestamps have been changed according to Lemma 8 (L14). □

7 Use Case and Evaluation

TinTiN is implemented in Apache Flink [4] and evaluated using a UDA based on a real-world validation application for Smart Meter (SM) readings in an Advanced Metering Infrastructure (AMI). 55 days of hourly data from 50.000 SMs are validated in the use case. We evaluate TinTiN's *output* (cf. Definition 4), *throughput*, *processing latency*, *logical latency*, *responsiveness* and *memory requirements*.

Use-case and experiment set-up The SMs periodically send the cumulative energy consumption to the utility's central servers. The readings, used for billing (among other things), are validated by calculating SMs' hourly consumption (by taking the difference between two consecutive readings) and verifying that the latter is positive and bounded by the installed fuses. Invalid readings are marked and processed to identify patterns indicating hardware failure (when readings exceeding the bounds are followed by a negative one within 24 hours). The data validation application outputs alerts for matched patterns as well as excessive or negative consumption values. Hourly readings can reach the utility up to 40 days late. Hence, parameter D is set to 40 days. There are two Aggregates in the query, one with WS 2 hours for calculating the hourly consumption and one with WS 24 hours, the pattern's maximum length. The size of $\beta[]$ is therefore set to 26 hours (the sum of the window sizes, cf. § 6.5). Both aggregates have WA 1 hour. TinTiN and the UDA are evaluated for (sub)sets of increasing size of the 50K SMs (statistics are given in Table 2), and run on a virtual server with 4 dedicated 2.6 GHz cores and 16 GB RAM. Throughput and processing latency results are averaged over 10 runs.

Parameters for TinTiN and baselines for comparison We evaluate TinTiN with the triggering conditions (TCs) from § 6:

TC-eager (TinTiN-TCE) reprocesses the relevant context for holes as soon as late data fills them. This approach minimizes logical latency but its output (cf. Definition 4) can contain multiple different and duplicate results before the final exact result is produced.

TC-lazy (TinTiN-TCL) reprocesses relevant context as soon as the next in-order reading (i.e. $t.ts \geq \tau$, the largest timestamp seen so far by TinTiN) arrives. This TC reduces the amount of different and duplicate results at the cost of a higher logical latency.

TinTiN and these TCs are compared against the following baselines:

SortedNoWait (SNW): an ideal baseline fed timestamp-sorted input and thus strictly deterministic. Note SNW cannot be used in practice (data is not sorted in the real-world application), it is included to characterize TinTiN's and other baselines' output in terms of different, duplicate, omitted and exact tuples, and logical latency.

UnsortedWait (UW): the baseline where the allowed delay of the input data is based on D (i.e., 40 days). The UDA processes data after storing it and waiting for D time units, incurring D time units logical latency penalty and large memory requirements but enforcing strict determinism. UW is essentially the option for system experts that are not stream processing experts to get accurate results.

UnsortedDiscard (UD): a baseline that discards all late arrivals, thus not validating all data and omitting final results when there are late arrivals. This can be the fastest but least accurate, hence not really usable in systems where accuracy and reliability are required.

Table 2: Data statistics for the used datasets.

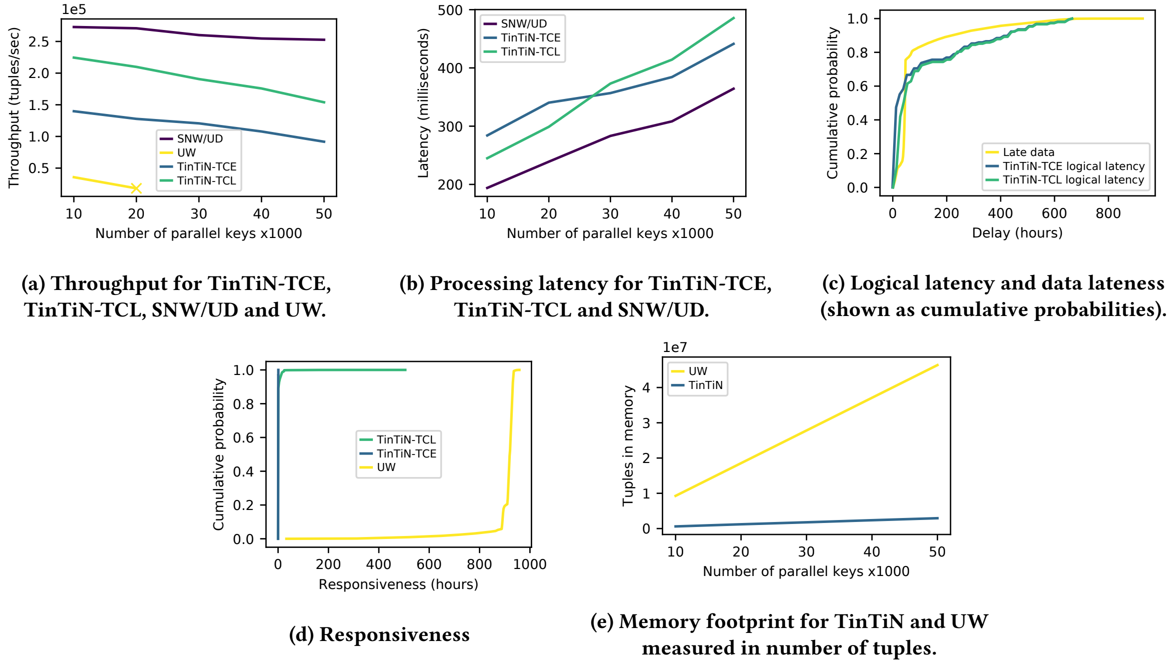
Dataset size (keys)	10k	20k	30k	40k	50k
Number of tuples	11.4M	22.8M	34.2M	45.5M	56.9M
Number of late tuples	90.1k	179k	262k	350k	435k
Number of holes	406k	808k	1.22M	1.66M	2.03M
Number of relevant contexts with holes	899k	1.79M	2.66M	3.58M	4.41M

Evaluation of quality of output Table 3 compares the output of both TCs and UD. As expected, TinTiN does not omit any results; UD omits approximately 10 percent of the exact results. TinTiN-TCL also gives fewer duplicate results than TinTiN-TCE, since the latter prioritizes reprocessing as soon as possible. This causes a result to be produced multiple times if it is affected by more than one hole, which in turn can result in multiple duplicate results.

Table 3: Output of TinTiN's TCs and UD compared with strictly deterministic output (such as from SNW or UW).

Dataset	AlgID	Exact	Omitted	Different	Duplicate
10k	TinTiN-TCE	240	0	0	351
10k	TinTiN-TCL	240	0	0	0
10k	UD	213	27	0	0
20k	TinTiN-TCE	388	0	0	645
20k	TinTiN-TCL	388	0	0	5
20k	UD	350	38	0	0
30k	TinTiN-TCE	518	0	0	2261
30k	TinTiN-TCL	518	0	0	37
30k	UD	460	58	0	0
40k	TinTiN-TCE	729	0	0	2536
40k	TinTiN-TCL	729	0	0	37
40k	UD	659	70	0	0
50k	TinTiN-TCE	850	0	0	2780
50k	TinTiN-TCL	850	0	0	39
50k	UD	765	85	0	0

Processing throughput Figure 5a shows the *processing throughput*, i.e. the number of processed tuples per second, for increasing number of parallel keys for TinTiN's TCs, SNW, UW and UD. Due to its processing overhead, TinTiN's throughput is lower than that of SNW's or UD's (notice though the latter baselines cannot be used in production). Nonetheless, it largely improves UW, which cannot sustain more than 20K keys (since it runs out of memory for larger number of keys). As expected, TinTiN-TCE's throughput is lower than TinTiN-TCL's since the former prioritizes low logical latency over the number of times data is potentially reprocessed.



(a) Throughput for TinTiN-TCE, TinTiN-TCL, SNW/UD and UW.

(b) Processing latency for TinTiN-TCE, TinTiN-TCL and SNW/UD.

(c) Logical latency and data lateness (shown as cumulative probabilities).

(d) Responsiveness

(e) Memory footprint for TinTiN and UW measured in number of tuples.

Figure 5: Evaluation graphs

Processing latency Figure 5b shows the *processing latency*, i.e. the difference in wall clock time between the creation time of an output tuple and the ingestion time of the input tuple that triggers such output. TinTiN adds approximately 100 ms to the latency when compared with UD and SNW. Also for this metric, it nonetheless performs significantly better than UW (not plotted since it is orders of magnitude larger, 98 and 191 seconds for 10K and 20K keys, respectively). As discussed in § 6, deploying a fresh UDA_R before replaying a window is not a viable solution. In our experiments, the time taken to deploy a fresh UDA_R instance is 1 order of magnitude larger than TinTiN's processing latency (between 3 and 4 seconds).

Logical latency Figure 5c shows the logical latency (cf. Definition 4); it naturally depends on the input data's lateness, which is drawn on the plot for convenience. As shown, TinTiN-TCE's logical latency is some hours smaller than TinTiN-TCL's and, for both *TC*s, it is substantially better than UW's (40 days). Since SM late data often arrives in batches, the logical latency penalty for TinTiN-TCL is relatively small compared with the throughput gain over TinTiN-TCE.

Responsiveness Figure 5d shows one of TinTiN's key strengths: its responsiveness compared to UW (i.e. the time between the arrival of a late tuple and the processing of the window this tuple belongs to, cf. Definition 4). Both TinTiN's TCs enable faster processing of late data. While TinTiN-TCE prioritizes swift reprocessing over performance, TinTiN-TCL offers a compromise between fast reprocessing of late data and performance. TinTiN reprocesses 90% of late data within 2 hours and 99.8% within 24 hours. For UW, 95% of the late data is processed more than 37 days after its arrival.

Memory Figure 5e shows the extra memory (in number of tuples maintained temporarily in order to process all data) required by TinTiN and UW. SNW and UD are not shown since they do not temporarily maintain tuples. The amount of memory required by TinTiN is two orders of magnitude smaller than UW's. This is expected, since UW needs to keep 1920 tuples in memory for every key (24 hours·40 days·2 aggregates), while TinTiN keeps 26 tuples per key in addition to the tuples that belong to a relevant context.

Our results show that TinTiN processes on-time data without delays, providing timely results for late data, based on its triggering condition, thus minimizing utilities' response time for actions.

8 Related Work

One of the eight requirements for real time stream processing as defined in [19] is resiliency against missing and out-of-order data. We propose a way for resiliency against missing and out-of-order data, one of the key requirements for streaming processing [19]. Earlier methods to handle such stream imperfections are *slack*[1] and *punctuation*[16], both methods introduce waiting in order to deal with out-of-order data which we aim to minimize. Recent work [23] utilizes a Slack-ScaleGate data structure in order to process out-of-order input strictly deterministic as long as a logical latency constraint can be fulfilled, but without guarantees otherwise. Slack can be combined with speculative processing and buffering for event processing [17], but this method requires the event processor to be able to export its internal state in order to be consistent. Our approach does not require any changes to the application, that is wrapped in order to be able to process out-of-order events.

The term eventual determinism has earlier been used also in a different context, i.e. algorithms with a probabilistic and a deterministic mode, for problems where randomization is needed

to break symmetries; processes eventually enter, and stay in, the deterministic mode [18]. Differently, here, the term is to characterize the output of processing whose input can be influenced by non-deterministic reorderings due to e.g. varying network delays.

An alternative approach to handle out-of-order data is to enhance the stateful operators in the streaming queries; [15] is early work in this direction which allows all stateful operators to store their state when data is late and to process late data with this stored state. Unlike ours, this method requires changes in the SPE or the original streaming application, and does not guarantee determinism.

The dataflow model [2], adopted by SPEs as Apache Flink [4] and Google Cloud Dataflow [10], allows for multiple evaluations of window instances, if the late data arrives no later than specified by an *allowed lateness* parameter. However the dataflow model cannot identify holes in the input stream and therefore cannot determine which window instances can receive late arrivals. For this reason all window instances need to be stored until the allowed lateness has expired, leading to excessive memory demands.

Orthogonal work, studying efficient merge-sorting of interleaving streams for strictly deterministic analysis, is presented in [22].

Data stream processing is a good match for smart grid challenges, as shown in [20, 21] where both applications disregard late data. Yet since occurrence of late data is common for smart energy meters, both are examples of applications that could leverage TinTiN.

9 Conclusion and Future Work

We introduce the concept of *D-bounded eventual determinism* to control streaming applications' trade-offs, in result correctness and quality versus timeliness, in CPS contexts where data fed to such applications comes out of timestamp order. We also present TinTiN, a middleware that enforces D-bounded eventual determinism, and evaluate it for a real-world Smart Grid use case. As shown, TinTiN induces minimal overhead in logical latency and enables processing of larger streams of data compared to other state-of-the-art methods. It enables out-of-order stream processing for 50K keys in parallel, where the strictly deterministic baseline is bound to 20K.

Future work includes the extension and refinement for different granularity of eventual determinism, including *weak* and *strong* variations (specifying whether multiple – possibly different – results can be produced for the same window of tuples) and variations of use-cases [7, 9, 11]. Since the processing order also impacts the cost of parallelization techniques for stream processing [5, 13, 23], it is also worth investigating (i) how TinTiN's semantics can be encapsulated in basic streaming operators, in order to leverage SPEs' distribution and parallelization techniques, and (ii) how TinTiN's methodology can benefit distribution and parallelization of queries, to provide guarantees about their degree of determinism.

Acknowledgments

Work partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, the collaboration framework of Göteborg Energi and Chalmers Energy Area of Advance project STAMINA, the Swedish Research Council proj. "HARE" grant nr. 2016-03800, the Swedish Foundation for Strategic Research proj. FiC, grant nr. GMT14-0032, the Chalmers Energy Area of Advance proj. INDEED, and the EU Horizon 2020 Framework Programme, grant nr. 773717.

References

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

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. 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*, volume 8, pages 1792–1803. 2015.

[3] Apache Beam. https://beam.apache.org/, 2016. last accessed: October 26, 2020.

[4] Apache Flink. https://flink.apache.org/, 2014. last accessed: October 26, 2020.

[5] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1), 2008.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 181–192. ACM, 2012.

[7] V. Botev, M. Almgren, V. Gulisano, O. Landsiedel, M. Papatriantafilou, and J. van Rooij. Detecting non-technical energy losses through structural periodic patterns in ami data. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3121–3130. IEEE, 2016.

[8] S. Costache, V. Gulisano, and M. Papatriantafilou. Understanding the data-processing challenges in intelligent vehicular systems. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 611–618. IEEE, 2016.

[9] Z. Fu, M. Almgren, O. Landsiedel, and M. Papatriantafilou. Online temporal-spatial analysis for detection of critical events in cyber-physical systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 129–134. IEEE, 2014.

[10] Google Cloud Dataflow. https://cloud.google.com/dataflow, 2015. last accessed: October 26, 2020.

[11] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas. Deterministic real-time analytics of geospatial data streams through scalegate objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 316–317, 2015.

[12] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, A. C. Koppisetty, and M. Papatriantafilou. Driven: a framework for efficient data retrieval and clustering in vehicular networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1850–1861. IEEE, 2019.

[13] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer. Quality-driven continuous query execution over out-of-order data streams. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 889–894. ACM, 2015.

[14] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[15] I. E. Kuralenok, N. Marshalkin, A. Trofimov, and B. Novikov. An optimistic approach to handle out-of-order events within analytical stream processing. In *CEUR Workshop Proceedings*, volume 2135, pages 22–29. RWTH Aahen University, 2018.

[16] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.

[17] C. Mutschler and M. Philippsen. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In *DEBS 2013 - Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, pages 147–158, 2013.

[18] J. R. Rao. Eventual determinism: Using probabilistic means to achieve deterministic ends. *J. of Parallel, Emergent and Distributed Systems*, 8(1):3–19, 1996.

[19] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[20] J. van Rooij, V. Gulisano, and M. Papatriantafilou. Locovolt: Distributed detection of broken meters in smart grids through stream processing. In *12th ACM International Conference on Distributed and Event-based Systems*, pages 171–182. ACM, 2018.

[21] J. van Rooij, J. Swetzén, V. Gulisano, M. Almgren, and M. Papatriantafilou. echidna: Continuous data validation in advanced metering infrastructures. In *2018 IEEE International Energy Conference*, pages 1–6. IEEE, 2018.

[22] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas. Viper: A module for communication-layer determinism and scaling in low-latency stream processing. *Future Generation Computer Systems*, 88:297 – 308, 2018.

[23] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas. Maximizing determinism in stream processing under latency constraints. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 112–123. ACM, 2017.