

TITLE

REAL-TIME WINDOW ENRICHMENT FOR MACHINE LEARNING

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of and priority under 35 U.S.C. § 119(e) to U.S. Provisional Application Serial No. 63/491,855, filed March 23, 2023, entitled "REAL-TIME WINDOW ENRICHMENT FOR MACHINE LEARNING," which is incorporated by reference in its entirety herein.

TECHNICAL FIELD

[0002] At least some aspects of the present disclosure are directed generally to machine learning models, and more particularly, to real-time window enrichment for machine learning.

SUMMARY

[0003] In one aspect, the present disclosure provides a computer-implemented method. The method can include defining a window based on a time range, adding an operator to the window, and adding a new event from a data stream into a series store. The series store can store events from the data stream. The events can include values and timestamps. The events can be sorted in the series store based on the timestamps. The method can further include sliding the window along the series store based on the timestamps to capture the new event and the events preceding the new event up to a limit of the time range. The method can further include updating a state of the operator based on the values of the events captured by the window, modifying the new event based on the state of the operator to generate a modified event, and outputting the modified event.

[0004] In another aspect, the present disclosure provides a server. The server can include a processor and a memory. The memory can include instructions executable by the processor to implement an enrichment model for modifying a data stream. The enrichment model can include a series store, a window, and an operator. The series store can be configured to store events received by the enrichment model from the data stream. The events can include values and timestamps. The events can be sorted in the series store based on the timestamps. The window can be configured to capture at least a portion of the events stored by the series store. The window can be defined by a predetermined range. The window can be slidable along the series store to an updated position based on receiving a new event. The operator can correspond to the window. The operator can be configured to change state based on the values of the events captured by the window. The enrichment model can be configured to modify the new event based on the state of the operator at the updated position of the window.

[0005] In yet another aspect, the present disclosure provides a computer-implemented method. The method can include receiving a first new event from a data stream. The first new event can include a first timestamp. The method can further include sorting the first new event into a series store based on the first timestamp. The series store can store events from the data stream. The method can further include setting a window along the series store at a first position to capture the first new event and the events preceding the first new event up to a predetermined time range. The method can further include computing a function based on the events captured by the window in the first position to generate a first modified event. The method can further include receiving a second new event from the data stream after receiving the first new event. The second new event can include a second timestamp. The second timestamp can be earlier than the first timestamp. The method can further include sorting the second new event into the series store based on the second timestamp, setting the window along the series store at a second position to capture the second new event and the events preceding the second new event up to the predetermined time range, and invertibly computing the function based on the events captured by the window in the second position to generate a second modified event.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] In the description, for purposes of explanation and not limitation, specific details are set forth, such as particular aspects, procedures, techniques, etc. to provide a thorough understanding of the present technology. However, it will be apparent to one skilled in the art that the present technology may be practiced in other aspects that depart from these specific details.

[0007] The accompanying drawings, where like reference numerals refer to identical or functionally similar elements throughout the separate views, together with the detailed description below, are incorporated in and form part of the specification, and serve to further illustrate aspects of concepts that include the claimed disclosure and explain various principles and advantages of those aspects.

[0008] The apparatuses, systems, and methods disclosed herein have been represented where appropriate by conventional symbols in the drawings, showing only those specific details that are pertinent to understanding the various aspects of the present disclosure so as not to obscure the disclosure with details that will be readily apparent to those of ordinary skill in the art having the benefit of the description herein.

[0009] FIG. 1 is a table illustrating an example of windowed analytic feature enrichment for a machine learning (ML)-based spam-email detector, according to at least one aspect of the present disclosure.

[0010] FIG. 2 illustrates an architecture for scalable windowed analytic feature enrichment in real time (SWAT-RT) and an example life cycle of a windowed analytic feature enrichment task, according to at least one aspect of the present disclosure.

[0011] FIG. 3 illustrates an invertible-incremental analytic function triggered by window slides to process an in-order event, according to at least one aspect of the present disclosure.

[0012] FIG. 4 illustrates an invertible-incremental analytic function triggered by window slides to process an out-of-order event, according to at least one aspect of the present disclosure.

[0013] FIG. 5 illustrates a graph showing enrichment latency with out-of-order events for different window sizes used in an example implementation of SWAT-RT, according to at least one aspect of the present disclosure.

[0014] FIG. 6 is a table illustrating an example data stream for an ML-based spam-email detector, according to at least one aspect of the present disclosure.

[0015] FIG. 7 illustrates windows in sliding window aggregation, according to at least one aspect of the present disclosure.

[0016] FIG. 8 illustrates windows in analytical functions, according to at least one aspect of the present disclosure.

[0017] FIG. 9 illustrates a window enrichment process, according to at least one aspect of the present disclosure.

[0018] FIG. 10 illustrates a model for real time window enrichment (RT-WinE), according to at least one aspect of the present disclosure.

[0019] FIG. 11 illustrates a window orbital buffer (WOB) processing an in-order event, according to at least one aspect of the present disclosure.

[0020] FIG. 12 illustrates a WOB processing an out-of-order event, according to at least one aspect of the present disclosure.

[0021] FIG. 13 illustrates a WOB implementing a row store, where records are stored consecutively in a node, according to at least one aspect of the present disclosure.

[0022] FIG. 14 illustrates record and a directory record, according to at least one aspect of the present disclosure.

[0023] FIG. 15 illustrates a WOB implementing a column store, according to at least one aspect of the present disclosure.

[0024] FIG. 16 illustrates a method of implanting a MIN operator implemented with invertible two stack, according to at least one aspect of the present disclosure.

[0025] FIG. 17 illustrates an example implementation of RT-WinE in RedisTimeSeries as a Redis Module, according to at least one aspect of the present disclosure.

[0026] FIG. 18 illustrates a graph showing an insertion time comparison for different experimental data processing implementations, according to at least one aspect of the present disclosure.

[0027] FIG. 19 illustrates a graph showing the total time to scan through an entire data structure in the forward direction for different experimental data processing implementations, according to at least one aspect of the present disclosure.

[0028] FIG. 20 illustrates a graph showing the total time to scan through an entire data structure in the reverse direction for different experimental data processing implementations, according to at least one aspect of the present disclosure.

[0029] FIG. 21 illustrates a graph showing the memory allocated to insert 2^{16} items into the data structure for different experimental data processing implementations, according to at least one aspect of the present disclosure.

[0030] FIG. 22 is a block diagram of a computer apparatus with data processing subsystems or components, according to at least one aspect of the present disclosure.

[0031] FIG. 23 is a diagrammatic representation of an example system that includes a host machine within which a set of instructions to perform any one or more of the methodologies discussed herein may be executed, according to at least one aspect of the present disclosure.

DESCRIPTION

[0032] The following disclosure may provide exemplary systems, devices, and methods for conducting a financial transaction and related activities. Although reference may be made to such financial transactions in the examples provided below, aspects are not so limited. That is, the systems, methods, and apparatuses may be utilized for any suitable purpose.

Scalable Windowed Analytic Feature Enrichment in Real Time (SWAT-RT): Low Latency Windowed Analytic Feature Enrichment on Time Series Data with Out-of-order Support

[0033] Machine learning data pipelines can be used for applications that process real-time data streams. The data pipelines can enrich data streams in real time before applying the data stream to a machine learning model. For example, analytical features can be

derived from portions of a data stream that are defined over sliding windows to capture the data stream's historical behaviors and signals. The process of deriving analytical features from a window of a data stream is sometimes referred to as feature enrichment. In many instances, the latency of the feature enrichment process may need to be as low as a few milliseconds to meet the requirements of various time-sensitive applications, such as network threat and anomaly detection.

[0034] For example, machine learning models can be used in real-time systems, such as network intrusion detection, to generate low-latency decisions on every transaction. Windowed analytic features on time series (e.g., transaction count in the last minute) can be an important class of features used by machine learning models to model historical patterns and identify whether the current transaction deviates from the history. Thus, enriching the source stream with windowed analytic features can be a beneficial step to technologically enhance machine learning pipelines that analyze time series data.

[0035] To illustrate a need for enriching the time-series data with windowed analytic features, the present disclosure now turns to an example of a machine-learning-based (ML-based) spam email detector that runs on an incoming email stream.

[0036] FIG. 1 is a table 100 illustrating an example of windowed analytic feature enrichment for an ML-based spam-email detector, according to at least one aspect of the present disclosure. The data pipeline for the ML-based spam email detector includes a raw stream of emails 102 that is augmented with a feature that measures an email frequency 110 for the corresponding sender 104 over the past 10 seconds.

[0037] As illustrated by FIG. 1, each of the emails 102 includes corresponding timestamps (e.g., send time 106, receive time 108) that are represented by integers values. Each of the emails 102 is received from a corresponding sender 104 (e.g., sender A, sender B). The sender's email frequency can be an important input analyzed by the detector's ML model. Thus, the number of emails received from this sender in the previous 10 seconds (e.g., frequency 110) can be a useful windowed analytic feature, which can be computed before feeding emails (e.g., the data stream) to the ML model. The table 100 of FIG. 1 presents a snapshot of a stream modified (e.g., enriched) with a 10-s email frequency 110 count feature and a corresponding spam flag 112. Notably, the event e_h (e.g., the email e_h) includes a positive spam flag 112 (e.g., marked as T or true), due to sender A's frequency 110 count over e_h 's 10-second range (determined by the send time 106 t_s) being higher than a predetermined threshold of 3 emails per 10 seconds. Thus, the data pipeline for the ML-based spam email detector can be modified (e.g., enriched) with a 10-s email frequency 110 and/or a spam flag 112 corresponding to each of the emails 102.

[0038] Adding an enrichment step to a real-time ML pipeline can be challenging and present various technical problems. For example, first, generating features for each window in a time series can be computationally expensive, especially for high-frequency data with large windows, which can therefore increase processing time. Time-sensitive applications with demanding latency requirements, such as network intrusion detection often cannot afford the additional delay of feature generation. Second, time series events can arrive out of order, meaning that an event may arrive in a different order than the generation order. In the example illustrated by FIG. 1, email e_k is sent before email e_j (send time 106 of email e_k is 13, earlier than the send time 106 of 15 for email e_j) but received after email e_j due to network delays (receive time 108 of email e_k is 17, later than the receive time 108 of 16 for email e_j). Processing these out-of-order events may require additional time and storage space for feature enrichment. Accordingly, there exists a need for approaches to windowed analytic feature enrichment that account for out-of-order data.

[0039] In various aspects, the present disclosure provides devices, systems, and methods for modifying (e.g., enriching) a data stream with windowed analytic features in real-time streams, sometimes referred to herein as scalable windowed analytic feature enrichment in real time (SWAT-RT). In one aspect, the present disclosure provides an example implementation of SWAT-RT developed as a Redis module. The Redis module can be incorporated into a real-world data pipeline for enriching data streams. The present disclosure demonstrates how real-time streams can be enriched with windowed analytic features for downstream machine learning-based applications. The present disclosure further demonstrates how SWAT-RT can process in-order and out-of-order events with low latency and storage costs. By way of experimental results, the present disclosure shows that on a laptop, an implementation of SWAT-RT can process one thousand events per second on a single core while achieving a sub 200 micro-second latency and allowing for around 80 second lateness. Accordingly, various aspects of SWAT-RT (and RT-WinE) disclosed herein can improve the functioning of a computer. Furthermore, as discussed in detail herein, SWAT-RT (and RT-WinE) can provide a technical solution to technical challenges related to enriching data streams in real time.

[0040] SWAT-RT can provide a framework for low-latency windowed analytical feature enrichment on time series data with out-of-order support. To address challenges described herein associated with processing out-of-order events, SWAT-RT can employ various techniques. First, by performing incremental computation to amortize the time required for enriching a single event, the latency of every event can be reduced, and the enrichment process can be made more effective. Second, by introducing the concept of invertible computation, the state of an operator can be inverted to enrich out-of-order events

effectively. Using a scenario of enriching a real-time stream of network requests for cybersecurity threat detection, the present disclosure demonstrates how feature enrichment can be achieved with low latency, high throughput, and out-of-order data support. Accordingly, various aspects of SWAT-RT disclosed herein can improve the functioning of a computer.

SWAT-RT FRAMEWORK

[0041] The present disclosure now turns to an example SWAT-RT framework and enrichment life cycle that can be implemented by any of the devices, systems, and methods disclosed herein.

[0042] FIG. 2 shows a model 200 for SWAT-RT and a life cycle of a windowed analytic feature enrichment task, according to at least one aspect of the present disclosure.

[0043] To configure a window-based enrichment task, a user can define a window 204 based on a “range” argument, which can define the window’s size, for example, in terms of a time range or a number of events. The user can then add one or more operators 206 (e.g., operator 206a, operator 206b) to the window. The one or more operators 206 can include an analytic function and its associated arguments.

[0044] After configuring, an upstream application can push events 208 from a data stream into the model 200 (e.g., apply a data stream to the model 200), and the enrichment life cycle can begin. Each event 208 can include a timestamp 210 and a value 212. As illustrated by FIG. 2, a new event 208a can be ingested and sorted into a series store 202 based on timestamp 210. Each of the events 208 in the series store 202 can be associated with a different window 204. Each window 204 can slide (b) according to the timestamp 210 of the new event 208a to capture all the events 208 that occurred before the new event 208a, within the range of the window 204. The state of each of the operators 206 can be updated (c) based on their corresponding window 204 sliding to include or exclude an event 208. The new event 208a can be modified (e.g. enriched) (d) with analytic results determined based on the corresponding window 204 and operators 206. The model 200 can output (e) the modified event (e.g., enriched event 210a).

[0045] Accordingly, when a new event 208a is ingested into the series store 202, the model 200 can first slide the windows 204 to trigger updates on the operators 206, and then enrich the results on and output the enriched event 210a. For example, as illustrated by FIG. 2, the new event 208a is enriched to include the maximum value (e.g., 7) of the events 208 in its corresponding window 204 and the sum of the values in its corresponding window 204, thereby resulting in the enriched event 210a. The above enrichment process (a–e) can repeat for every incoming event 208 into the model 200. The series store 202 can expire (f)

and delete a stored event 208b if the event 208b is no longer referenced by any window 204.

[0046] The present disclosure now turns to various features that may be included or otherwise employed by in any of the series stores described herein, such as, for example, the series store 202. The series store can be responsible for storing values from the raw events in a manner that enables efficient sequential scan and maintains a low memory footprint. For each raw event, the series store can store its timestamp with the raw fields that are to be used in the operators for calculation. For example, to enrich the running average of prices of shopping orders, the series store can store the timestamp and the price of each order.

[0047] The series store can include an unrolled linked list as a base. The underlying data structure of series store can be an unrolled linked list with adaptive sized chunks. For example, initially, the size can start at 16 bytes and double each time, up to a maximum of 4KB. Unrolled linked lists can be more memory-efficient than vectors (e.g., dynamic growing arrays) and may not require data movement when the list size reaches its allocated memory limit. Compared to a standard linked list, an unrolled linked list can include pointers only between chunks, not between each value, making it smaller and faster to scan.

[0048] The series store can employ compression with support of bi-directional scans. The series store can compress its values to conserve memory while maintaining the capability to perform sequential forward and backward scans. This can be done by recognizing that some numerical fields may have a large domain, but a majority of their values may be small. For example, in a social media platforms like Twitter, where the number of likes on a post can range up to millions, but the majority values are often below 256 (28). The series store can employ a compression algorithm using variable-length storage based on the value distribution, storing values using the smallest possible data type in size (byte, short, int, or long). Two additional encoding bytes, one at the beginning and one at the end, can be used to decode the values in both directions. An encoding byte can include bits for null values, sign, and length. For example, an integer value -23 can be stored using only three bytes 41-17-41 in hex.

[0049] The series store can employ timestamp optimization. For example, according to the present disclosure, the storage of timestamps in the series store can be further optimized. For the first event in a chunk of the list, the full timestamp may be stored. For the remaining events, the difference between current event's and previous event's timestamp can be stored. During sequential scans, the full timestamps can be reconstructed by summing the stored differences. Although some differences between successive events can be large, they are often small, especially for high-frequency streams where events are only a

few milliseconds apart. These differences can be further compressed using the encoding method described earlier.

[0050] The present disclosure now turns to various features that may be included in or otherwise employed by any of the windows described herein, such as, for example, the window 204. Windows can be represented or otherwise defined by “start” and “end” pointers, which can be iterators that point to an element of a series store. In one aspect, the “start” pointer can always point right behind the event being enriched, as the start of a window. And the “end” pointer can point to the end of the window determined by the size. According to the present disclosure, a window can be considered to have captured the events between the two pointers.

[0051] Windows can employ incremental and invertible slides, as described further herein. For example, the pointers can be incremented or decremented to slide a window on a series store across the stored data. These movements of the pointers can enable incremental computation by triggering the operators each time they move. Moreover, as the window can slide both forward and backward, incremental and/or invertible slides can facilitate invertible computation.

[0052] Windows can employ awareness and management of iterators. According to the present disclosure, the idea of iterator-awareness is introduced, where the series store handles iterators in a way that ensures they remain valid whenever it is updated, for example, when an element is inserted, appended, or removed.

[0053] The present disclosure now turns to various features that may be included in or otherwise employed by any of the operators described herein, such as, for example, the operators 206. An operator can maintain several states in addition to the raw events in series store to compute the most recent analytic feature of its corresponding window. Each operator can update its state incrementally as events enter and leave the window. For example, a sum operator can maintain the most recent summation of the associated window and can update the summation incrementally by adding each new value entering the window and subtracting each value leaving the window. This can ensure that the state of each operator is synchronized with its corresponding window.

INVERTIBLE-INCREMENTAL ANALYTIC FUNCTIONS

[0054] The present disclosure now turns to describe an interface that can be employed for implementing analytic functions on windows as they slide forward and backward on the Series Store, event by event. The interface can be used to achieve functions operate in an incremental and/or an invertible manner, as described further herein.

[0055] FIGS. 3 and 4 illustrate invertible-incremental analytic functions that can be triggered by window slides, enabling the processing of in order and out-of-order events, where FIG. 3 illustrates processing an in-order event and FIG. 4 illustrates processing an out-of-order event, according to at least one aspect of the present disclosure.

[0056] An incremental computation (e.g. analytic function) can distribute the computation to each event and can therefore aid in decreasing latency. Incremental interfaces provided by the present disclosure can include:

[0057] • insert(e: Iter). Insert an element to increment the current state.

[0058] • evict(e: Iter). Remove an element to decrement the current state.

[0059] • getResult(). Return the analytical result of the current window.

[0060] The processing of in-order events (e.g., an event including a timestamp that is later than the events' timestamp currently in the series store) can be enriched with an incremental API. For example, FIG. 3 illustrates a snapshot of a series store 302 while processing events 308. Each event can include a timestamp 310 and a value 312. A window 304 of size 5 (W_5) is positioned with a start pointer before event 308b (H), capturing events in the timestamp range of [4, 9]. A SWAT-RT model can employ three steps to enrich an in-order event 308a (X) with a timestamp 310 of 10: (i) the model can move the start pointer and scan forward during a first scan until it points to an event 308 whose timestamp 310 is equal to or greater than (e.g., later than) 10, triggering insert(startPtr) for all the first scanned events; (ii) the model moves the end pointer and scans forward during a second scan until it points to an event 308 whose timestamp 310 is smaller than 5 (e.g. timestamp of 10 – window size of 5 = 5), triggering evict(endPtr) for all the second scanned events; and (iii) the model can insert the event 308a (X) at the position pointed by the start pointer.

[0061] Invertible interface provided by the present disclosure can include:

[0062] • invInsert(e: Iter). Invert adding of an element.

[0063] • invEvict(e: Iter). Invert removing of an element.

[0064] The processing out-of-order events (e.g., an event including a timestamp that is earlier than at least some of the events' timestamp currently in the series store) can be achieved with invertible APIs. For example, FIG. 4 illustrates a snapshot of a series store 402 while processing events 408. A SWAT-RT mode can employ three steps to enrich an out-of-order event 408a (X) with a timestamp 410 of 6: (i) the model can move the start pointer and scan backward during a first scan until it points to an event 408 whose timestamp 410 is less than or equal to 6, the timestamp 410 of event 408a (X), triggering invInsert(startPtr) for all the first scanned events; (ii) the model can move the end pointer

and scan backward during a second scan until it points to an event 408 whose timestamp 410 is less than or equal to 1 (e.g., timestamp of 6 – window size of 5 = 1), triggering `invEvict(endPtr)` for all the second scanned events; and (iii) the model can insert the event 408a (X) at the position pointed by the start pointer.

[0065] The present disclosure now turns to describe various implementations of different analytical operators that can be supported by SWAT-RT to operate on events within a window.

[0066] In some aspects, analytical operators can include sum, count, and/or average operators. These operators can represent examples of invertible functions. The `invInsert` can be equivalent to `evict`; and the `invEvict` can be equivalent to `insert`.

[0067] In some aspects, analytical operators can include min, max, lead, and/or lag operators. These can be examples of functions that do not have a naive invertible implementation. Using MIN as an example, suppose a min operator stores the current min value of a window of 10 events, which is 2. After window slides, if value 2 is evicted from the operator, in order to get the next min value, the operator would require a full scan of the events for re-calculation, which can be expensive in time. To compute MIN in an incremental manner, the Two-Stacks algorithm can be extended to be invertible.

[0068] In some aspects, analytical operators can include a distinct count operator. To compute distinct count, a hashmap of (event value, count) can be maintained. Here, an insert operation can increment the count and an evict operation can reduce the count. When the count reaches zero, the entry can be deleted. The hashmap's size can be returned as the result of the operator. Each hashmap, which is one partial aggregate takes $O(N)$ space, thus the algorithms which store $O(N)$ aggregates can result in a $O(n^2)$ of space. The operations on the hashmap can be $O(1)$ and naturally invertible.

EXAMPLE IMPLEMENTATION AND RESULTS

[0069] The inventors have developed a fully functional prototype employing an example SWAT-RT framework as a Redis module using C++. Redis can provide flexible extension API, enterprise support, and the fine-grained memory management. This non-limiting example SWAT-RT implementation uses custom data types and commands that are fully compatible with standard Redis clients, enabling easy adoption of SWAT-RT in a production data pipeline for feature enrichment. For instance, commands `SWAT.DEFINE` were added to configure a new enrichment task. The present disclosure provides a collection of analytic operators that supports invertible-incremental API, including SUM, AVG, MIN, MAX, COUNT, DISTINCT COUNT, LEAD, LAG.

[0070] The example SWAT-RT was evaluated using a sample dataset of financial transactions that includes anonymized cardholder and merchant IDs, along with a timestamp and transaction amount for each transaction. The machine used for timings was a MacBook Pro with 2.3 GHz 8-Core Intel Core i9 processor with 16GB memory. Data was ingested at a rate of 1000 records per second, with some transactions arriving late.

[0071] FIG. 5 illustrates a graph 500 showing enrichment latency with out-of-order events (e.g., the processing time for late events) for different window sizes used in this example implementation of SWAT-RT, according to at least one aspect of the present disclosure. Events with lateness greater than the window size were ignored since they were not within the sliding window. This can explain why the time required to process late events increases and then drops to near zero after the lateness exceeds the window size.

DEMONSTRATION SCENARIOS

[0072] The present disclosure now provides various non-limiting example demonstration scenarios that can illustrate how a user can interact with various SWAT-RT implementations processing various data sets and executing analytic-window feature enrichment tasks.

[0073] As one non-limiting example, a CIC-DDoS2019 dataset can be processed. CIC-DDoS2019 is a public dataset that can include network traffic logs that can resemble traffic logs generated true real-world PCAPs. The dataset can include benign and common DDoS attacks (distributed denial-of-service attaches) such as PortMap, NetBIOS, LDAP, MSSQL, UDP, UDP-Lag, SYN, NTP, DNS and/or SNMP.

[0074] Example Scenario #1: Creating enrichment tasks. The user of a system implementing SWAT-RT can define new windows and add analytic operators to create new enrichment tasks and observe the enriched output stream. On a Redis client, the user can execute a SWAT.DEFINE to define analytical functions along with the corresponding window definitions. The stream events can be pushed into SWAT-RT by invoking SWAT.PUSH [*|ts] key value command. The user can observe the output stream with the new enriched field by invoking a SWAT.AGG key command.

[0075] Example Scenario #2: Monitoring SWAT-RT status. Users can monitor and examine the status of the SWAT-RT, such as, for example, total memory, latency and throughput, at the desired time using standard Redis commands. For example, to obtain latency information, users can invoke redis-cli info latency stats for SWAT-RT operators. The user also may add/remove windows as well as analytic operators to compare the status of SWAT-RT under different loads.

[0076] Example Scenario #3: Handling of out-of-order events. In this example scenario,

the present disclosure provides streams with simulated out-of-order events. To do so, some events' ingestion order can be arbitrarily modified so they are emitted later than their assigned event time. The user can observe how an out-of-order condition is handled by a series store, as well as any changes in latency that occur as a consequence. The user can adjust the frequency, the lateness of out-of-order events, and the stream's ingestion rate to observe the effect on the latency of analytical functions.

RT-WinE: Low Latency Analytical Window Feature Generation on Real-time Streams with Out-of-order Support

[0077] Having disclosed various devices, systems, and methods for modifying (e.g., enriching) a data stream with windowed analytic features in real-time streams, the present disclosure now turns to various other devices, systems, and methods for modifying a data stream with windowed analytic features. Any of the features described above (e.g., FIGS. 1-5) can similarly apply to or otherwise be implemented by the other devices, systems, and methods for enriching a data stream with windowed analytic features described below (e.g., FIGS. 6-24), and vice versa.

[0078] Features generated by analytical-window functions can capture historical behaviors and thus provide an important signal for predictive analysis with machine learning. In real-time data analysis pipelines, such analytical-window features can be required to be generated online before being fed to machine learning models. Many time-sensitive real-time applications, such as, for example, fraud detection and anomaly detection, can require the latency to be as low as a few milliseconds. Out-of-order events, for example caused by clock drift and network delay, can present technical challenges in online stream processing. For example, some window algorithms that guarantee an amortized or a worst-case $O(1)$ latency often do not support out-of-order events and can have huge trade-off on space, which can cause these approaches to be not applicable in practice.

[0079] Features can be signals that are given to a machine learning model as input. For example, in addition to individual raw data points applied to a machine learning model, aggregated features can be derived from the raw data and applied to the model, providing contextual information related to the raw data. A common way to generate contextual features is to use time window aggregation. These contextual features can be significant for models that are sensitive to the latest data trend until the prediction moment. For example, the performance of fraud detection models for credit card fraud can be improved based on receiving the count of transactions and average price of transactions during a window of time prior to the transaction being analyzed (e.g., the 30 minutes prior to the transaction). Similar applications such as price predictions, real-time recommendations, spam detection, anomaly detection and ranking systems and/or machine learning models achieve similar

improvements based on receiving aggregated features by time. However, enriching raw datasets with aggregated features in real time presents various technical problems, for example, such as the technical problems described below under the PROBLEM SETTING heading. Accordingly, there is a need for devices, systems, and methods that can enable enrichment of data streams in real time.

[0080] In various aspects, the present disclosure provides devices, systems, and methods that that can generate analytical window features. Devices, systems, and methods provided herein can enable out-of-order analytical function support via real time window enrichment, which is sometimes referred to RT-WinE, and which can be similar in many respects to SWAT-RT as described above. RT-WinE can enable devices, systems, and methods to generate analytical window features with low-latency and high-throughput, while addressing of out-of-order events. As discussed further herein, the inventors have shown that an example implementation of RT-WinE can uses 32% of space compared to baseline methods, which achieving a low P99 latency. Thus, the devices, systems, and methods provided herein can improve the functioning of a computer.

[0081] To support ML applications, there is often a need to enrich a raw event dataset with the aggregated features as a step of the data pipeline. For an offline processing, such an enrichment is commonly conducted with SQL OVER WINDOW clauses. Various aspects of the present disclosure can enable the time-window-based aggregation feature enrichment in a real-time setting.

PROBLEM SETTING

[0082] The present disclosure now turns to discuss various technical challenges associated with of real-time analytical window feature generation.

[0083] Analytical functions can run over a real-time stream of events to modify (e.g., enrich) the events with features. The features can be used by downstream AI/ML models for inference. For example, stock prediction models may benefit from features such as the running average of prices, volume of trade for the past 10 minutes. As another example, anomaly detection models can utilize features such as min or max signal in the recent 20 signals. As AI/ML models that inference on real-time data stream can generally only see the past data, various aspects of the present disclosure are related to solving the technical problem of enriching an event with analytical features based on a set of events that are processed before the event in time.

[0084] FIG. 6 shows a table 600 illustrating an example data stream for an ML-based spam-email detector, according to at least one aspect of the present disclosure. The data stream is modified (e.g., enriched) using the email frequency 610 of the senders 604 of

emails 602. Email e_h is considered to be spam (e.g., marked with T or true under the spam 612 column), as there are more than 3 emails received from sender 604 A within 10 seconds range before receiving e_h . Email e_k is received out-of-order because there is an event e_j with a send time 606 later than e_k but a received time 608 earlier than e_k . As discussed further herein, there are technical problems associated with calculating the email frequency 610 of e_k 's sender 604 in the 10 second window associated with e_k .

[0085] One of the features used by the example detector's ML model is the email frequency 610 of a sender 604. Thus, before feeding each email 602 to the ML model, the detector calculates the email frequency 610 of the sender 604 frequency by counting how many emails 602 have been received from this sender in the past 10 seconds. For the context of this example, it is assumed that if the count (frequency 610) is more than 3, the emails from the sender are considered by the model as spam, as shown with T (true) or F (false) tags in the spam 612 column of table 600. FIG. 6 illustrates a snapshot of the stream along with the generated features of the spam-email detector. For the purpose of this example, integers starting from 1 to represent timestamps (send time 606, receive time 608) are used. It is also assumed that normally there is a 1-second delay between the send time 606 and the receive time 608 for spam detection, yet the delay may be larger for some emails. In this example, e_h is labeled as spam because the email frequency 610 of its sender A is 4, which satisfies the predetermined spam threshold limit of greater than 3. The email frequency is counted over events in the 10-s range $[ea, eg)$, which is measured based on either send time 606 T_s or receive time 608 Tr . Abnormally, there could be emails that are sent early but received later than other emails, due to some network delays (e.g., an out-of-order event). For instance, in this example, e_k is sent before e_j (13 vs 15) but received after e_j .

[0086] An event can be a set of values denoted as ϵ (values). A stream (denoted as S) can be modeled as a sequence of events, $S = \{\epsilon 1, \epsilon 2, \dots\}$. A running example is enriching an input stream Si by adding analytical features onto each event, result in an output stream So . For simplicity, consider input stream Si with events of a single input value ϵt (vin), where the events in the output stream are added with a single output value ϵt ($vin, vout$). In the running example, each email is an event with the input value being its sender 604, and the output value is the 10-second email frequency 610 of the sender.

[0087] In the example, event-time and processing-time are used as two different notions for timestamps as they can have different implications for analytical functions. Specifically, the event-time (denoted as $te \epsilon$) marks when the event is generated from the end devices, and the processing-time (denoted as $tp \epsilon$) marks when the event is being processed. Due to

hardware latency and network, the event-time and the processing-time often naturally have some gap and may not be in the same order. Thus, when considering event-time as the timestamp in an event, the analytical engine may receive events out of order. For instance, in the running example, a particular email may be sent earlier but received later than other emails by the detector. An event ϵ_x can be out-of-order if there exists another event with a later event-time but with an earlier processing time.

[0088] The set of out-of-order events of a stream Si can be defined as follows:

$$\mathbb{C}(Si) = \{\epsilon \mid \epsilon \in Si, \exists \epsilon' \in Si, (t_{\epsilon'}^e > t_{\epsilon}^e] \& t_{\epsilon'}^p > t_{\epsilon}^p)\}$$

[0089] In the example, email e_k is an out-of-order event because it is sent before e_j but received by the detector after e_j .

[0090] A window frame can capture or otherwise define a set of events that are to be aggregated.

[0091] A time-based window frame can define a set of events before a target event in terms of event time. The range can be described with an inclusive start timestamp and an exclusive end timestamp. The time window frame of length s regarding an event ϵ can be defined as followed:

$$W(s) = \{\epsilon' \in Si \mid (t_{\epsilon}^e - s) \leq t_{\epsilon'}^e < t_{\epsilon}^e\}$$

[0092] For example, referring to FIG. 6, the 10-second window for email e_h in the running example can be denoted as $We_h(10)$, which aggregates all emails from e_a to e_g .

[0093] A count-based window frame can be a special case of time-based window frame. A count-based window frame can calculate the range based on the count of elements in the set.

[0094] A series of analytical functions can perform on a window frame. An analytical function can be a function that takes in a window frame and outputs a single value. An analytical function of a window frame can be denoted as $\phi(W)$, where ϕ can be one of SUM, COUNT, AVG, MIN, and MAX, as well as their distinct version of them.

[0095] The result of analytical functions can include one or more than one contextual feature that can be added to a current event as its feature for downstream processing.

[0096] FIG. 9 illustrates a window enrichment process 900, according to at least one aspect of the present disclosure. Events 902 can be ingested as inputs. For each of the events 902, window frames 904 can define a set of the events 902 occurring within a predetermined window frame size of the corresponding event 902. An analytical function can

take action on the window frame 904 and output a feature 906 for the event that corresponds to the window frame 904, thereby generating a modified event (e.g., an enriched event 908). The enriched events 908 can be output and, for example, fed to a machine learning model.

[0097] The present disclosure now turns to various technical problems to be solved by various techniques described herein. Given a stream of events S_i , such as transactions and logs. For each input event ϵ , the system can measure a window frame $W\epsilon$ and perform an analytical function ϕ on the window frame. The analytical function's output value can be added onto the original event ϵ as its new feature: $\epsilon(vout) = \phi(W\epsilon)$.

[0098] A system that supports real-time window analytical feature generation can have the following requirements, which add many technical challenges, as discussed below.

[0099] Out-of-order events. When applying analytical window functions based on event-times, out-of-order events may occur, which can add a hard requirement on systems to support them.

[0100] Semantics of handling out-of-order events. To perform analytical feature generation for an out-of-order event, a system often needs to treat an out-of-order event as if it is received with its event-time order, even if it is received with its processing-time order. Returning to the example of FIG. 6, at this snapshot of table 600, an out of-order event e_k is received, with its event-timestamp (send time 606) to be 13 and the sender field to be A. The corresponding 10-second window frame of e_k would be $W_e(10s) = \{et \mid t \in [3, 13)\}$. The number of emails sent by A in this window frame is 4.

[0101] Out-of-order events add pressure to both latency and memory space. For a FIFO (first-in first-out) data stream, the window frames of two adjacent events could be similar and have some overlaps, which can provide an opportunity for the two window frames to share common storage and calculation. However, an out-of-order event's window frame can be very different from that of the event processed right before it, which can require additional storage and calculation to handle the difference in a window frame. Thus, there are often costs to pay for the analytical engine to support out-of-order events, and more out-of-order events can cause more fluctuation in the overall latency and memory used.

[0102] Low Latency at high throughput. In some aspects, to generate analytical features in a real-time stream, the throughput of the system has to be large enough to keep the stream flow. Meanwhile, in other aspects, the latency for each event needs to be small enough to satisfy the downstream operators, such as machine learning models, in a prompt manner.

[0103] Limited Memory Constrains. To provide a low latency solution, one approach is to

keep as much data as possible in memory to avoid disk I/O. The memory requirement to store and compute a window frame is a function of window size and event frequency $M(Ws, f req) = Ws * f req$, where Ws is the size of window, and $f req$ is the event frequency. The amount of window frames can also be a function of event frequency, since each event has a corresponding window frame, thus the total space requirement of all windows can be $Ws * f req^2$. For example, a window with the size of 1 minute and event frequency of 20 Hz could result in 24,000 events or references of events to be stored. In practice, window sizes are often in weeks or months and the event frequency are far more than 10 Hz, thus the memory usage can be easily scaled up. In particular, analytical window functions typically consist of the following heavy storage components:

- (1) raw events storage;
- (2) window frame reference storage; and
- (3) analytical state storage.

[0104] High density of window frames with limited resources. Analytical window frames can have an extremely fine granularity of event level, and thus could have an extremely high density of window frames when event frequency is high. For example, when employing an analytical window with a size of 10 minutes with a data stream having the event frequency of 10Hz, as shown in FIG. 8, the window frame's granularity is at each individual event, and each event is in a maximum of 1000 window frames that are densely packed. A naive method could configure a window frame and collect events for each event before performing the aggregation. Collecting and calculating the events can be slow when the window frame is large in size and dense in frequency. The two window frames for two consecutive events could have the majority of prior events to be repeated, and only differ on a few events. Many windows frames could have overlaps, for example, a time window with size of 20 minutes contains the entire window frame of a time window with size of 10 minutes. For the overlapped window frames, the storage could be repeated which adds burden to memory constrain, and the computation could also be repeated which adds burden to latency and throughput.

[0105] FIGS. 7 and 8 illustrate window granularity comparison, where FIG. 7 illustrates a chart 700 of windows 702 in sliding window aggregation, according to at least one aspect of the present disclosure, and FIG. 8 illustrates a chart 800 of windows 802 in analytical functions, according to at least one aspect of the present disclosure.

RT-WinE ARCHITECTURE

[0106] The present disclosure now turns to an example RT-WinE framework and enrichment life cycle that can be implemented by any of the devices, systems, and methods

disclosed herein.

[0107] FIG. 10 shows a model 1000 for RT-WinE, according to at least one aspect of the present disclosure. The model 1000 can be similar in many respects to the model 200 (FIG. 2). Any of the features disclosed herein with respect to the model 1000 can apply to or otherwise be implemented by the model 200, and vice versa.

[0108] The model 1000 can include a window orbit buffer (WOB) 1002. Raw events 1008 can be received and added to the WOB 1002. The model 1000 can further include windows 1004 and one or more corresponding operators 1006. The windows 1004 can shift on the WOB 1002, triggering analytical function results to be calculated by operators 1006, thereby generating modified events (e.g., enriched events 1010). Raw events 1008 are stored in the WOB 1002 for future windows 1004 to use and are expired and removed from WOB 1002 when no longer used by any window 1004. The WOB 1002 can be similar to the series store 202 (FIG. 2).

[0109] The present disclosure now turns to an example method that can be carried out by the model 1000. Referencing FIG. 10, an event 1008 (e) from the stream is received (a) and pushed into the system. The model 1000 can align (b) (e.g., shift) windows 1004 according to $te \in$. The model 1000 can modify (e.g., enrich) (c) the events 1008 e by calculating or otherwise executing the analytical functions (e.g., operators 1006) on windows 1004, resulting in an enriched event e' . The model 1000 can insert (d) a copy of the raw event 1008 e into the WOB 1002. The model can yield (e) an output of the enriched event 1010 e' with the aggregates. The model 1000 can expire (f) an event 1008 and evict (e.g., delete) it from the WOB 1002 when no window 1004 frame is using the event 1008.

[0110] The model 1000 can handle events 1008 received out of order and events 1008 received in order differently. For example, the model 1000 can execute the align (b) and insert (d) processes differently for out-of-order events and in-order events. For an out-of-order event 1008, the corresponding window 1004 may be far away in the middle part of WOB 1002, thus it can require more window shifts to update the position of the window 1004. The out-of-order event is also inserted in the middle of WOB 1002 rather than appended at the end of WOB 1002. With the understanding of the life cycle, the present disclosure turns to the various components that can be included in the model 1000 to implement RT-WinE.

[0111] In various aspects, as noted above, the model 1000 can include a window orbit buffer (WOB) 1002. The WOB 1002 can provide a core data structure. For example, the WOB 1002 can store raw events 1008. Many analytical function algorithms can require a copy of raw events to work. In addition, some algorithms also require storage of partial

results. The WOB 1002 can be configured to store the raw events to satisfy these storage requirements, thereby allowing for fast calculation and modifications.

[0112] The WOB 1002 (e.g., portions of the WOB 1002) can be captured by the windows 1004. The windows 1004 can therefore include a cheap and flexible representation to accommodate high data density.

[0113] The WOB 1002 can record the event-time sequence. For example, to support out-of-order events, the WOB 1002 can store the events with respect to their event-time (e.g., timestamp). The sequence of the historical events can help the operators 1006 retrieve correct analytical results for out-of-order events.

[0114] The WOB 1002 can include or otherwise implement any of the following features:

[0115] (1) The events stored in the WOB 1002 can be sorted with respect to their event-timestamp;

[0116] (2) In-order events can be appended at the tail (e.g., of the WOB 1002) when being inserted, and popped from the head (e.g., of the WOB 1002) when they are no longer needed by any windows 1004;

[0117] (3) Out-of-order events can be inserted in the middle (e.g., of the WOB 1002);

[0118] (4) The windows 1004 can shift forward and backward by using the WOB 1002 as an orbit.

[0119] In various aspects, as noted above, the model 1000 can include windows 1004. The windows 1004 can keep a reference to which events are being calculated by the operators 1006. The window 1004 frames can be represented in RTWinE by using a start and an end pointer to the events 1008 in the WOB 1002. The two pointers naturally define a range which is the content of the window frame. Regardless of the frame size, the representation only requires the size of two pointers, which can be small in size and easy to modify.

[0120] Each window 1004 can maintain a list of operators 1006.

[0121] A window 1004 can shift along the WOB 1002, either forward or backward, which updates the window frame it represents. In some aspects, the minimal granularity of the shift is an event 1008. When a window 1004 shifts, its callback API can trigger the associated operators 1006 to update their computation accordingly (e.g., to include or exclude a value associated with an even 1008 added or removed from the window 1004).

[0122] In various aspects, as noted above, the model 1000 can include operators 1006. An operator 1006 can maintain some additional states besides the raw events 1008, to

compute the latest analytical feature of its corresponding window 1004. The operator can listen to callbacks triggered by the window shifts to update its state. For example, when a new event 1008 enters the window 1004, the operator 1006 can include the event's field value into its state by calling `insert(ele)`. When an event 1008 exits the window 1004, the operator 1006 can exclude the event's field value by calling `evict(ele)`. For example, a sum operator can maintain the latest summation of the window it is associated, wherein `insert(ele)` adds the value to the latest summation and `evict(ele)` subtracts the value from the latest summation (e.g., thereby incrementally computing the function). Thus, the states of the operator 1006 states can maintain a synced configuration with their corresponding window 1004.

INVERTIBLE INCREMENTAL ANALYTICAL FUNCTIONS

[0123] In various aspects of the present disclosure, invertible incremental analytical functions can be performed on window frames, as window frames move forward and backward on the WOB, event by event.

[0124] When RT-WinE receives an event whose timestamp is newer than the latest event's timestamp, the event can be considered to be in-order.

[0125] FIG. 11 illustrates a WOB 1102 processing an in-order event, according to at least one aspect of the present disclosure. The WOB 1102 has multiple events 1108 stored already and the latest event 1108b has a timestamp of 10-second. There is a window 1104 frame W of size 5 seconds for tH 's query, the start pointer points to tF and the end pointer points to tH , representing $[F, H)$. At this very moment, a new event 1008a tJ is received. As its timestamp is larger than the latest event 1108b in the WOB 1102, which is tH , it is an in-order event. The analytical window frame of event tJ is supposed to be $[G, J)$, thus RT-WinE shifts the window 1104 forward (e.g., to the tail of the WOB) to perform the query action. To do so, RT-WinE first shifts the end pointer forward until it is pointing to an event whose timestamp is smaller than tJ . For all the events scanned by the end pointer, the `insert(ptr)` callbacks are triggered. Then RT-WinE shifts the start pointer forward until it is pointing to an event whose timestamp is equal to or bigger than tG 's timestamp. For all the events scanned by the start pointer, the `evict()` callbacks are triggered. After finishing querying, RT-WinE inserts the event tJ at the location pointed by the end pointer, ready for the next event.

[0126] When RT-WinE receives an event whose timestamp is older than the latest event's timestamp, the event can be considered to be out-of-order.

[0127] FIG. 12 illustrates a WOB 1202 processing an out-of-order event, according to at least one aspect of the present disclosure. The WOB 1202 has multiple events 1208 stored

already and the latest event is event 1208b. With the similar setup with the in-order processing, a new event 1208a tE is received. As the timestamp for event 1208a tE is larger than that of the latest event 1208b in the WOB 1202, which is t_H , it is an out-of-order event. The analytical window frame of event tE is supposed to be $[A, E)$, thus RT-WinE shifts the window 1204 backward (e.g., to the head of the WOB 1202) to perform the query action. To do so, RT-WinE first shifts the start pointer backward until it is pointing to an event whose timestamp is smaller than or equal to t_A . For all the events scanned by the start pointer, the `inv_evict(ptr)` callbacks are triggered. Then RT-WinE shifts the end pointer backward until it is pointing to an event whose timestamp is smaller than or equal to tE 's timestamp. For all the events 1208 scanned by the end pointer, the `inv_insert()` callbacks are triggered. After finishing querying, RT-WinE inserts the event tE at the location pointed by the end pointer, ready for the next event.

[0128] In various aspects, RT-WinE can employ an invertible Subtract-on-Evict (SoE) technique. The invertible SoE, as disclosed herein, can expand and/or modify on the Subtract-on-Evict (SoE) discussed in the publication by Martin Hirzel et al, titled Sliding-Window Aggregation Algorithms: Tutorial, published in Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (Barcelona, Spain) (DEBS '17), the disclosure of which is incorporated by reference herein in its entirety.

[0129] In various aspects, invertible SoE can employ an operator to define the following functions:

[0130] `insert(e: Iterator)` Insert an element to increment the current state.

[0131] `evict()` Remove an element to decrement the current state.

[0132] `inv_insert()` Invert adding of an element, to decrement the current state.

[0133] `inv_evict(e: Iterator)` Invert removing an element, to increment the current state.

[0134] `get_result()` Returns the aggregation result as for the current window frame.

[0135] The above interfaces can support implementations of invertible operators. It also can fall back to Recalculate from Scratch when the window frame is small enough or the function is not invertible.

[0136] The present disclosure now turns to discuss various properties that may be employed by an of the operators discussed herein.

[0137] In various aspects, operators can be invertible. For example, there can exist an invert transformation for every transformation of the operator. SUM is an example of invertible functions in view of subtraction $C - B = A$ to invert $A + B = C$.

[0138] Table 1 below includes various operators that may be employed and properties that can be associated therewith:

Operation	Invertible	Communicative	Associative
SUM/AVE/COUNT	√	√	√
MIN/MAX	X	√	√
DISTINCT COUNT	X	√	√
FIRST/LAST/LEAD/LAG	√	X	X

Table 1

[0139] In various aspects, the operators can process events in a first-in-first-out (FIFO) order. In some aspects, the operators do not need to be commutative, because events can be processed in the FIFO order on the timestamp. For out-of-order events, the model can invert the operator back to a previous state, process the out-of-order in its correct order, and then reprocess events in FIFO order.

[0140] In some aspects, the operators may not be aware of the event timestamp(s) in their implementations. Each operator may only process the events in the given order with the interface invoked by the window pointers.

WINDOW ORBIT BUFFER (WOB)

[0141] According to various aspects of the present disclosure, the WOB can serve as main event storage. The WOB can be designed to be space efficient while maintaining $O(1)$ complexity of push and pop. The WOB storage can support the window frames to move forward and backward on it efficiently.

[0142] The WOB can have the following properties:

[0143] (1) $O(1)$ push and $O(1)$ pop at both the head and the tail of an element.

[0144] (2) iterator access to allow $O(N)$ read only-scan, and random insertion of an element in the middle.

[0145] (3) elements are stored in a sorted order.

[0146] (4) allow compression and adaptivity for space-efficiency.

[0147] (5) persist-able in a sequential manner, including pointers and iterators.

[0148] To support $O(1)$ push and pop at both ends, and $O(N)$ random insert, generally, a list can be used. This can be implemented, for example, in either doubly linked list or array-based list. Array-based Lists have $O(1)+$ (Amortized) complexity for push, because can need

to re-allocate a bigger array and perform memory copy when space is not enough. When inserting an element, it can also need to shift elements after the insertion position. Doubly Linked Lists can have exact $O(1)$ for push but a much higher constant since it requires fine-grained memory allocation for each element. The overhead of scanning linked lists can also be high due to random memory access by pointers, comparing to sequential access in array-based lists.

[0149] An unrolled linked list (ULL) is a hybrid of doubly linked lists and array-based lists, which can store multiple elements in a list node. An example implementation of the unrolled linked list would be a doubly linked list with an array as its list node payload. Comparing to the doubly linked list, to store the same quantity of elements, the unrolled version can have fewer list nodes and thus acquire fewer node pointers. Additionally, due to fewer pointers and more adjacent memory access, a sequential scan on an unrolled linked list would be significantly faster than on a doubly linked list.

[0150] According to the present disclosure, various techniques can be implemented to further improve space utilization rate on top of the unrolled linked list.

[0151] In one aspect, variable length numerical types can be employed. A standard unrolled linked list can take an array of a certain data type, e.g., long integers, which can waste space. Many numerical fields can be within a certain domain and with a certain distribution. For example, the domain of a person's age is roughly the integral numbers within the range of 0 to 150, and the domain of a car's price is roughly the decimal numbers within the range of 0.00 to 3,000,000,000.00. Systems often use a numerical type that is larger than the cap of the domain to store a field. In this case, all ages can be stored as 2 bytes shorts, and the car prices can be stored as 8-bytes decimals, just to accommodate the largest value in the domain. Additionally, values may not be evenly distributed in the domain, but can fit in a certain probabilistic distribution. Storing all values as a fixed sized value can waste on space in the majority of the applications.

[0152] RT-WinE can store numerical types in a variable length manner. For example, the following techniques can be employed:

[0153] 1) PadLong: Depending on value, the closest data type in size (e.g., byte, short, int, or long) can be used to physically store it and use 1 byte to describe the encoding. The 1-byte encoding can include a 1-bit flag for null values, a 1-bit flag for the sign, and a 4-bit to differentiate the actual length. For example, -23 in decimal can be stored as 01000001-00010111 in binary, or 41-17 in hex. This format can only support forward reading but not backward reading.

[0154] 2) BiPadLong: In some aspects, BiPadLong can be similar to PadLong, but

saving the 1-byte encoding twice, before and after the value. This can support reading the value from both directions, by using a little more space. See TABLE 2 for detailed comparison between different encodings, where TABLE 2 shows variable-length Long encodings, displayed in HEX.

Values	-23		1024	
	Repr	Space Saving	Repr	Space Saving
Original(Long)	FFFFFFFF FFFFFFE9	/	00000000 00000400	/
PadLong	4117	75%	020400	62.5%
BiPadLong	411741	62.5%	02040002	50%

TABLE 2

[0155] In some aspects, variable length string types can be employed. The variable length strings could be stored similarly. For the 1-byte encoding, the 1-bit flag can be used for null values, 1-bit flag for encoding continuity (e.g., if set to true, the next byte is also an encoding byte, to support arbitrary length), and 6-bit to describe length of the string. The actual string value can be followed after all the encoding bytes.

[0156] In various aspects, timestamp compression can be employed. Timestamp can be a top citizen of the ULL since each element in the list may need to have an associated timestamp stored for sorting. WOB can store timestamp in ISO format which is a standard Long type. Since Timestamps can be relatively long numbers, but can also occur frequently real-time applications, two timestamps may have only a small delta in difference. The small delta can be relatively smaller (e.g., in terms of storage), and can be represented with a few bytes or even bits. Based on that, a full timestamp of a list node's first event can be stored, and for each further occurrence of timestamps in this node, only the delta to the full timestamp may be stored.

[0157] In some aspects, this timestamp compression technique can further benefit extremely frequent event streams. For example, when the event frequency is too high, it is possible to have two events having the same timestamp. The WOB can accept duplicate timestamps and the compressed timestamp can simply have the value 0 as the delta, which only takes 1 byte.

[0158] In some aspects, an events collapse technique can be employed. When incoming events have a high frequency, it can be common for two events to have same Unix timestamp, since the granularity of most timestamp is millisecond. There are some

techniques to differentiate them such as, for example, assigning a sequence number for repeated timestamp (e.g., 1662661100813-1 and 1662661100813- 2). This can present in issue because the granularity of the timestamp may not be the same as the granularity of the application. In analytical window functions setting, a typical window size can often be larger than millisecond level, and the granularity can be seconds, minutes, hours or even days. Events can be collapsed if their timestamps fall under the same unit of the granularity. For example, when the required granularity is at second level, the WOB can collapse events within the same second and store only the aggregates instead of the raw events. This can greatly reduce the memory consumption. In some aspects, this technique can cost the opportunity of operator computation sharing.

[0159] In various aspects, many data can share the same timestamp. For example, a 10-field event has 10 values that share the same timestamp. Two events that have the same Unix timestamp can often be collapsed together in a record. The record can be used to store those fields in a space-efficient way in the WOB. There can be different arrangements to store a record in the WOB, wherein each can have different benefits and suitable use cases. As in relational databases, the arrangements can be categorized as row-store based and columns-store based. In some aspects, those arrangements are discussed with the assumption that all events have a fixed schema, e.g., each event has same number of fields and the same sequence of field types.

[0160] FIG. 13 illustrates a WOB 1302 implementing a row store, where records 1312 are stored consecutively in a node, according to at least one aspect of the present disclosure. As shown in FIG. 13, an event (e.g., record) can be stored in a WOB 1302 consecutively.

[0161] FIG. 14 illustrates record 1412a vs. directory record 1412b, according to at least one aspect of the present disclosure. As shown in FIG. 14, at least two kinds of row-store designs are disclosed:

[0162] 1) Record 1412a. A timestamp and a count can describe how many values are following, which all share the same timestamp. All the field values can be stored in PadLong encoding, with one record length in the end to allow reading the entire record backwards (e.g., jump back to the head of the record and read).

[0163] 2) Directory Record 1412b. Directory Records can share the same space usage as the Record 1412a, but data can be rearranged internally. Each field's encoding (1-byte) can be stored consecutively, followed by the values. This can allow for better pipelining when reading record sequentially.

[0164] FIG. 15 illustrates a WOB 1502 implementing a column store, comprising a TS

sub-WOB 1514, and one or more than one fields sub-WOB(s) 1516 (e.g., Field 1 sub-WOB 1516a, Field 2 sub-WOB 1516b), according to at least one aspect of the present disclosure. As shown in FIG. 15, iterators can be synced on all the sub-WOB(s). Records also can be stored in column-store fashion. This can require multiple WOB instances working together as a logical WOB. For example, the TS sub-WOB 1514 can stores only timestamps (denoted as TS). The WOB 1502 can include the field sub-WOB(s) for each field of an event. Iterator movements can be done on the TS sub-WOB 1514 only, and after measuring the distance of the movement, the same on field sub-WOB(s) can be applied accordingly.

[0165] In some aspects, column store can provide various benefits compared to row store. For example, column store can provide better compression opportunity since each of the WOBs saves same type and domain of values. One example is the timestamp WOB can contain compress more timestamps in a node. As another example, in column store, moving on the timestamp's WOB can determine out how many steps the iterator needs to shift. When moving synchronize on the sub-WOB (s), a for-loop can be used to enable better pipelining.

[0166] In some aspects, a hybrid column/row store can be employed. For example, depending on the count of fields, operator numbers on each field, and/or event frequency, row-store and column-store may have different benefits. As event frequency could change on the fly, the decision may not be well-made ahead of time. A hybrid solution can make WOB adaptively changes from using row-store and column store. The interface of WOB can allow some nodes to be row-store while some nodes are in column-store, which can provide a better storage solution.

[0167] The present disclosure now turns to discuss various aspects of iterator awareness and management. The access to the WOB can be managed by iterators. In standard container implementations such as STL data structures, iterators are often not respected when the data structure is altered or updated, e.g., when elements are added or removed from the data structure. Using `std::vector` as one example, when an element is erased, iterators that are pointing to and beyond the element may all be invalidated.

[0168] In some aspects, to support incremental analytical functions, the WOB can be aware of iterators on it and manage them correctly as it changes. Iterators can be used extensively in the WOB. For example:

[0169] (1) Each window frame can include two iterators representing the range, and shifting window frames can substantially include just incrementing and decrementing the corresponding iterators.

[0170] (2) An insertion iterator can be used to push or insert an element on the WOB at

the correct position.

[0171] (3) An expiration iterator can be used to pop element from the head of the WOB.

[0172] Forward & Backward Scan. Window frame's shifting can substantially include just iterators' increment and decrement. Each of those iterators' movements can be independent from each other, thus no special actions need to be taken care of.

[0173] Insertion. To insert a new element, at the tail or any middle position of the WOB, all window frames can be first shifted to prior to the insertion position. This can result it at least two benefits: 1) any window's end iterator can serve as the insertion iterator of the new element, and 2) all iterators can be prior to the insertion position therefore inserting the new element can be done without affecting any iterators because there are no iterators beyond it.

[0174] Eviction. To evict an element from the head of the WOB, all window frames can be first shifted to beyond the eviction position to ensure no iterator is pointing to the element that is being evicted. The WOB can employ lazy eviction by incrementing the head iterator for one unit, thus all iterators can still point to the correct element(s) without being affected.

INVERTIBLE INCREMENTAL ANALYTICAL ALGORITHMS

[0175] The present disclosure now turns to various algorithms that may be employed by the operators disclosed herein.

[0176] In some aspects, if an operator itself is invertible, such as SUM and COUNT, then implementation can be straightforward. In other aspects, operators may not be invertible, such as MIN, MAX. In this section, the present disclosure mainly discusses invertible algorithms for non-invertible operators. Operators are discussed group by group, where the operators in a group share similar properties.

[0177] SUM, COUNT, and AVERAGE are examples of invertible and associative operators. It can be quite simple to implement these operators with RT-WinE framework. The `inv_insert` can be equivalent to `evict`; while the `inv_evict` can be equivalent to `insert`.

[0178] MIN is an example of non-invertible operator. For instance, suppose a window frame of 10 events has a min value of 2. In a naive algorithm, a min operator would store a state of the current min value, which is 2. When a new event with value 1 is added to this window frame, the operator compares 1 with 2 and updates the state to 1. However, if then the event with 1 is removed from the window frame (e.g., `inv_insert(1)`), the operator cannot arrive at 2 based on the updated state of 1. Thus, the operator itself is non-invertible.

[0179] One way to compute non-invertible operators in an incremental manner is to use Two Stack, and Two Stack is extended to be invertible.

[0180] For example, referencing the algorithm below, 1 line 1–20 shows the standard Two Stack algorithm, which takes $O(1) + (\text{Amortized})$ complexity to execute each query. Line 2–31 are the extended invert APIs for Two Stack. The invert APIs can essentially cancel the effect of the original API.

[0181] Algorithm 1 Invertible Two Stack

```

1: function qery
2:   return F.top().agg + B.top().agg
3: end function
4:
5: function insert(ptr)
6:   F.push(WOB [ptr], F.top().agg + WOB [ptr])
7: end function
8:
9: function evict
10:  if F.isEmpty() then
11:    flip(F, B)
12:  end if
13:  F.pop()
14: end function
15:
16: function flip(src, dst)
17:  while not src.isEmpty() do
18:    dst.push(src.top().val, src.top().val + dst.top().agg)
19:  end while
20: end function
21:
22: function inv_evict(ptr)
23:  B.push(WOB [ptr], B.top().agg + WOB [ptr])
24: end function
25:
26: function inv_insert
27:  if B.isEmpty() then
28:    flip(B, F)
29:  end if
30:  B.pop()
31: end function

```

[0182] FIG. 16 illustrates a method 1600 of implementing a MIN operator with Invertible Two Stack, according to at least one aspect of the present disclosure. As an example, steps (A) - (H) of the method 1600 illustrate insertion of 4, 2, 5, 3 and then evicts 4, finally inserts 7. As shown in FIG. 16, the states change for an MIN operator implemented with Invertible Two Stack. Steps (I) - (L) of the method 1600 illustrate inversion of the state of Two Stack back to equivalent to (D), and step (M) inserts the new value 1.

[0183] In some aspects, another way to implement a MIN/MAX operator can be using algorithm DABA.

[0184] The present disclosure now turns to discuss employing a distinct count operator.

An operator for Two Stack and DABA that can be challenging to implement is DISTINCT COUNT, for example, due to its heavy partial aggregates in space. A naive but effective method to implement DISTINCT COUNT can be to maintain a hashmap of (event value, count). Each a hashmap, which can be one partial aggregate, can takes $O(N)$ space. Thus the algorithms which store $O(N)$ aggregates can result in a $O(n^2)$ of space. However, in some aspects, the operations on the hashmap are $O(1)$ and naturally invertible, since maintaining the count is invertible.

IMPLEMENTATION

[0185] FIG. 17 illustrates an example implementation of RT-WinE model 1700 in RedisTimeSeries as a Redis Module for experiments, according to at least one aspect of the present disclosure. The model 1700 can include a partitioner 1720 that partitions to a plurality of Redis modules (e.g. Redis module 1720₁, Redis module 1720₂, . . . Redis module 1720_m)

[0186] In some aspects, employing a RedisTimeSeries can provide the following benefits:

[0187] (1) It can be implemented in C/C++ which can provide fine-grained memory control.

[0188] (2) It can be a data storage layer. In some aspects, bottlenecks related to window enrichment can related to storage rather than computation. Thus, employing a RedisTimeSeries can help to relieve bottlenecks related to window enrichment.

[0189] The present disclosure now turns to discuss various aspects related to user experience. In some aspects, a user can define enrichment metadata to describe the enrichment field, enrichment window properties, and aggregation operations. According to the metadata, RTWinE can construct empty WOB(s), window(s), and/or operator(s) and await inputs. In some aspects, a user can then connect streams (or keyed streams with a partitioner) to the corresponding WOB. User can also to specify the consumers of the output stream. With the input and output configured, RT-WinE can start, and outputs can be observed.

[0190] The present disclosure now turns to discuss various data types that may be processed. In some aspects, as a space compression technique, a WOB may expect only type of long integer and strings to be stored internally. To satisfy user's need for window enrichment, conversions for other data types can done. For example, data types can include:

[0191] (1) Char, Int, Short & Long: Integral types can be stored as longs.

[0192] (2) Decimal: Decimals can be stored as two Long types, one for precision and one for scale, both can be in variable-length.

[0193] (3) Float & Double: Due to the nature of floating-point types, having incremental calculations may result in rounding issues. To prevent from round issues, for all floating-point types, RT-WinE can store it as decimal types to support incremental aggregation operations.

[0194] (4) Date, Time, Timestamp, & Datetime: Date-related types can be stored as longs directly.

[0195] The present disclosure now turns to discuss various aspects related to event expiration. When a new event is being enriched and added to a WOB, the event can be added to an expiration list. A callback or a busy check loop can check if an event is no longer needed by RT-WinE and then initiate the expiration of the event. When expired, the event can be evicted (e.g., deleted) from the corresponding WOB and no longer visible to the RT-WinE.

[0196] The present disclosure now turns to discuss various other optimization techniques that can be implemented with the devices, systems, and methods disclosed herein.

[0197] In some aspects, adaptive tuning can be implemented. For example, many knobs can be tuned in practice for WOB and different configurations can benefit in different scenarios. Being able to adaptively tune those knobs makes the WOB flexible to handle different sorts of workloads.

[0198] In some aspects, adaptive node size can be implemented. For example, for an unrolled linked list, depending on the frequency of input events and their size, node size can be increased and decreased adaptively.

[0199] In some aspects, an adaptive node fill rate can be implemented. For example, in addition to the size of capacity on each node, a node fill rate can be introduced to pre-allocate spaces for potential out-of-order events to be inserted.

[0200] In some aspects, an adaptive collapse can be implemented. For example, the collapse technique discussed herein also can be done adaptively. In one aspect, storing aggregations can save on space but may introduce compression and decompression overhead, which could eventually affect the overall throughput and latency.

[0201] In some aspects, less-frequently-used nodes can be spilled. For example, a long run window enrichment pipeline supporting many window configurations could consume a lot of memory. Consider a RT-WinE with 3-day, 7-day and 30-day windows and 1-day out-of-order allowance. The WOB may need to store 31 days events in total to support enrichment

of all possible events. However, the majority of insertions can happen in-order on the last node, and eviction can happen on the first node only. The middle nodes which are storing events for day 8 to day 29 are therefore not changing very frequently, thus they can be viewed as cold data. Those cold data can be store onto a disk and can be pre-loaded back in memory before they need to be used.

[0202] In some aspects, a hybrid recalculate and invertible incremental enrichment technique can be employed. For example, when event frequency is not that high and aggregation states are not big, it may be beneficial to re-calculate instead of maintaining events in memory for invertible incremental enrichment. Each operator which supports invertible incremental enrichment can fallback to re-calculate.

[0203] In one general aspect, the present disclosure provides a system that can allow low latency and high throughput computation of windowed aggregates, which can then be fed to machine learning models. In one aspect, the present disclosure describes different data structures and methods for window aggregates that can provide space-savings, low-latency, and high throughput to increase machine efficiency, thereby improving the functioning of a computer. For example, in various aspects, the devices, systems, and methods provided herein can implement improved data structures, incremental computation, and invertible algorithms to reduce latency, memory usage and data storage, and improve computational throughput.

[0204] In one aspect, the devices, systems, and methods provided herein can provide window-aggregate-based features to capture historical behaviors and thus can provide a signal for machine learning models such as fraud detection, object detection, and product suggestion/recommendation. These devices, systems, and methods can enable the enrichment of raw events using the windowed aggregate features, in real time using a low-latency and high-throughput approach, before feeding the raw data to a machine learning model.

[0205] Existing window algorithms that employ an amortized or a worst-case $O(1)$ latency often assume unlimited space is available as a trade-off for achieving this latency. Window enrichment in accordance with the present disclosure can employ a more stringent space requirement that can cause existing window algorithms to be inapplicable or inoperable. In one aspect, the present disclosure provides real-time window enrichment (RT-WinE) techniques that can execute window enrichment for machine learning models under a limited memory space while achieving low-latency and high throughput. Example experiments conducted using various devices, systems, and methods according to the present disclosure show that RT-WinE can use 32% of memory space compared to baseline

methods and can have a low P99 latency.

EXAMPLES

[0206] The present disclosure provides various non-limiting experimental implementations of various devices, systems, and methods disclosed herein.

Experiment Setup

[0207] Experiments were done on a machine with an Intel Xeon Gold 6142 CPU running at 2.60GHz with 1.5 TiB memory distributed across 4 NUMA nodes. Each CPU socket has 1 MiB L1 data cache, 32 MiB L2 cache and 44 MiB L3 cache. The time for memory access will be shorter if memory local to the CPU socket is accessed, compared to if it accesses memory local to a different CPU socket. To mitigate this issue during evaluation, memory allocations and computation were restricted to a single NUMA node. Each experiment is the mean of 100 runs to smooth away random noise from running the experiments.

Baseline implementations

[0208] There are no existing buffer data structures in stream-processing engines to support out-of-order window enrichment. Thus a few baselines are provided.

[0209] Baseline 1: Flat Vector based Buffer is implemented with a `std::vector<long>`. For each event, the timestamp is pushed into the vector, followed by the number of fields of this event, then followed by each field.

[0210] Baseline 2: Vector of Vectors Buffer is implemented with a `std::vector<std::vector<long>>`. Each event is represented by an inner `std::vector<long>` to store timestamp, and all the fields. Then the inner vector is pushed onto the outer vector.

[0211] Baseline 3: Link List of Vectors Buffer is implemented with a `std::list<std::vector>`. Each event is represented by an inner `std::vector<long>` to store timestamp, and all the fields. Then the inner vector is appended to the outer list.

[0212] Example method 1: Aulist (Record & binarypad) Buffer is implemented with an unrolled linkedlist of long types, where each node of the list can store values up to 4K bytes. For each event, it is represented as a row-based record discussed in section 5. The record is appended to the unrolled linkedlist.

[0213] Example method 2: Aulist (Directory Record & binarypad) (Low priority) Y-axis -> Throughput/Latency, and Size X-axis -> length of the list

[0214] (Low priority) Y-axis -> Throughput/Latency, and Size X-axis -> Record size

Operation Time for Different Implementation of WOB

[0215] FIG. 18 illustrates a graph 1800 showing an insertion time comparison for the experimental implementations, according to at least one aspect of the present disclosure. As seen in graph 1800, when the number of insertions is small, a similar insertion time of roughly 160ns is observed for baseline2 (vector of vector) and baseline3 (linked list of vectors), the WOB insertion time takes about 50ns, and baseline1 (flat vector) performs the best with roughly 25ns. However, when the number of insertions scales up, baseline methods (vector of vector, linked list of vectors, flat vector) hit the L2 cache limit and starts to incline drastically. The WOB implementation also inclines as the number of insertions scales up, but much more linearly.

Time to do Scan Through All Data in Structure

[0216] FIG. 19 illustrates a graph 1900 showing total time (in ms) to scan through an entire data structure in the forward direction for the different experimental implementations, according to at least one aspect of the present disclosure.

Memory Allocated for Different Implementations

[0217] End-to-end enrichment latency

[0218] FIG. 20 illustrates a graph 2000 showing the total time (in ms) to scan through an entire data structure in the reverse direction for the different experimental implementations, according to at least one aspect of the present disclosure.

[0219] FIG. 21 illustrates a graph 2100 showing the memory allocated to insert 2^{16} items into the data structure for the different experimental implementations, according to at least one aspect of the present disclosure. As shown by the graph 2100, WOB can support inserting compressed data, which can allow for a smaller memory footprint.

[0220] FIG. 22 is a block diagram of a computer apparatus 3000 with data processing subsystems or components, according to at least one aspect of the present disclosure. The subsystems shown in FIG. 22 are interconnected via a system bus 3010. Additional subsystems such as a printer 3018, keyboard 3026, fixed disk 3028 (or other memory comprising computer readable media), monitor 3022, which is coupled to a display adapter 3020, and others are shown. Peripherals and input/output (I/O) devices, which couple to an I/O controller 3012 (which can be a processor or other suitable controller), can be connected to the computer system by any number of means known in the art, such as a serial port 3024. For example, the serial port 3024 or external interface 3030 can be used to connect the computer apparatus to a wide area network such as the Internet, a mouse input device, or a scanner. The interconnection via system bus allows the central processor 3016

to communicate with each subsystem and to control the execution of instructions from system memory 3014 or the fixed disk 3028, as well as the exchange of information between subsystems. The system memory 3014 and/or the fixed disk 3028 may embody a computer readable medium.

[0221] FIG. 23 is a diagrammatic representation of an example system 4000 that includes a host machine 4002 within which a set of instructions to perform any one or more of the methodologies discussed herein may be executed, according to at least one aspect of the present disclosure. In various aspects, the host machine 4002 operates as a standalone device or may be connected (e.g., networked) to other machines. In a networked deployment, the host machine 4002 may operate in the capacity of a server or a client machine in a server-client network environment, or as a peer machine in a peer-to-peer (or distributed) network environment. The host machine 4002 may be a computer or computing device, a personal computer (PC), a tablet PC, a set-top box (STB), a personal digital assistant (PDA), a cellular telephone, a portable music player (e.g., a portable hard drive audio device such as an Moving Picture Experts Group Audio Layer 3 (MP3) player), a web appliance, a network router, switch or bridge, or any machine capable of executing a set of instructions (sequential or otherwise) that specify actions to be taken by that machine. Further, while only a single machine is illustrated, the term "machine" shall also be taken to include any collection of machines that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methodologies discussed herein.

[0222] The example system 4000 includes the host machine 4002, running a host operating system (OS) 4004 on a processor or multiple processor(s)/processor core(s) 4006 (e.g., a central processing unit (CPU), a graphics processing unit (GPU), or both), and various memory nodes 4008. The host OS 4004 may include a hypervisor 4010 which is able to control the functions and/or communicate with a virtual machine ("VM") 4012 running on machine readable media. The VM 4012 also may include a virtual CPU or vCPU 4014. The memory nodes 4008 may be linked or pinned to virtual memory nodes or vNodes 4016. When the memory node 4008 is linked or pinned to a corresponding vNode 4016, then data may be mapped directly from the memory nodes 4008 to their corresponding vNodes 4016.

[0223] All the various components shown in host machine 4002 may be connected with and to each other or communicate to each other via a bus (not shown) or via other coupling or communication channels or mechanisms. The host machine 4002 may further include a video display, audio device or other peripherals 4018 (e.g., a liquid crystal display (LCD), alpha-numeric input device(s) including, e.g., a keyboard, a cursor control device, e.g., a mouse, a voice recognition or biometric verification unit, an external drive, a signal generation device, e.g., a speaker,) a persistent storage device 4020 (also referred to as

disk drive unit), and a network interface device 4022. The host machine 4002 may further include a data encryption module (not shown) to encrypt data. The components provided in the host machine 4002 are those typically found in computer systems that may be suitable for use with aspects of the present disclosure and are intended to represent a broad category of such computer components that are known in the art. Thus, the system 4000 can be a server, minicomputer, mainframe computer, or any other computer system. The computer also may include different bus configurations, networked platforms, multi-processor platforms, and the like. Various operating systems may be used including UNIX, LINUX, WINDOWS, QNX ANDROID, IOS, CHROME, TIZEN, and other suitable operating systems.

[0224] The disk drive unit 4024 also may be a Solid-state Drive (SSD), a hard disk drive (HDD) or other includes a computer or machine-readable medium on which is stored one or more sets of instructions and data structures (e.g., data/instructions 4026) embodying or utilizing any one or more of the methodologies or functions described herein. The data/instructions 4026 also may reside, completely or at least partially, within the main memory node 4008 and/or within the processor(s) 4006 during execution thereof by the host machine 4002. The data/instructions 4026 may further be transmitted or received over a network 4028 via the network interface device 4022 utilizing any one of several well-known transfer protocols (e.g., Hyper Text Transfer Protocol (HTTP)).

[0225] The processor(s) 4006 and memory nodes 4008 also may comprise machine-readable media. The term "computer-readable medium" or "machine-readable medium" should be taken to include a single medium or multiple medium (e.g., a centralized or distributed database and/or associated caches and servers) that store the one or more sets of instructions. The term "computer-readable medium" shall also be taken to include any medium that is capable of storing, encoding, or carrying a set of instructions for execution by the host machine 4002 and that causes the host machine 4002 to perform any one or more of the methodologies of the present application, or that is capable of storing, encoding, or carrying data structures utilized by or associated with such a set of instructions. The term "computer-readable medium" shall accordingly be taken to include, but not be limited to, solid-state memories, optical and magnetic media, and carrier wave signals. Such media also may include, without limitation, hard disks, floppy disks, flash memory cards, digital video disks, random access memory (RAM), read only memory (ROM), and the like. The example aspects described herein may be implemented in an operating environment comprising software installed on a computer, in hardware, or in a combination of software and hardware.

[0226] One skilled in the art will recognize that Internet service may be configured

to provide Internet access to one or more computing devices that are coupled to the Internet service, and that the computing devices may include one or more processors, buses, memory devices, display devices, input/output devices, and the like. Furthermore, those skilled in the art may appreciate that the Internet service may be coupled to one or more databases, repositories, servers, and the like, which may be utilized to implement any of the various aspects of the disclosure as described herein.

[0227] The computer program instructions also may be loaded onto a computer, a server, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer-implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0228] Suitable networks may include or interface with any one or more of, for instance, a local intranet, a PAN (Personal Area Network), a LAN (Local Area Network), a WAN (Wide Area Network), a MAN (Metropolitan Area Network), a virtual private network (VPN), a storage area network (SAN), a frame relay connection, an Advanced Intelligent Network (AIN) connection, a synchronous optical network (SONET) connection, a digital T1, T3, E1 or E3 line, Digital Data Service (DDS) connection, DSL (Digital Subscriber Line) connection, an Ethernet connection, an ISDN (Integrated Services Digital Network) line, a dial-up port such as a V.90, V.34 or V.34bis analog modem connection, a cable modem, an ATM (Asynchronous Transfer Mode) connection, or an FDDI (Fiber Distributed Data Interface) or CDDI (Copper Distributed Data Interface) connection. Furthermore, communications also may include links to any of a variety of wireless networks, including WAP (Wireless Application Protocol), GPRS (General Packet Radio Service), GSM (Global System for Mobile Communication), CDMA (Code Division Multiple Access) or TDMA (Time Division Multiple Access), cellular phone networks, GPS (Global Positioning System), CDPD (cellular digital packet data), RIM (Research in Motion, Limited) duplex paging network, Bluetooth radio, or an IEEE 802.11-based radio frequency network. The server 4030 can further include or interface with any one or more of an RS-232 serial connection, an IEEE-1394 (Firewire) connection, a Fiber Channel connection, an IrDA (infrared) port, a SCSI (Small Computer Systems Interface) connection, a USB (Universal Serial Bus) connection or other wired or wireless, digital or analog interface or connection, mesh or Digi® networking.

[0229] In general, a cloud-based computing environment is a resource that typically combines the computational power of a large grouping of processors (such as within web servers) and/or that combines the storage capacity of a large grouping of computer

memories or storage devices. Systems that provide cloud-based resources may be utilized exclusively by their owners or such systems may be accessible to outside users who deploy applications within the computing infrastructure to obtain the benefit of large computational or storage resources.

[0230] The cloud is formed, for example, by a network of web servers that comprise a plurality of computing devices, such as the host machine 4002, with each server 4030 (or at least a plurality thereof) providing processor and/or storage resources. These servers manage workloads provided by multiple users (e.g., cloud resource customers or other users). Typically, each user places workload demands upon the cloud that vary in real-time, sometimes dramatically. The nature and extent of these variations typically depends on the type of business associated with the user.

[0231] It is noteworthy that any hardware platform suitable for performing the processing described herein is suitable for use with the technology. The terms “computer-readable storage medium” and “computer-readable storage media” as used herein refer to any medium or media that participate in providing instructions to a CPU for execution. Such media can take many forms, including, but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media include, for example, optical or magnetic disks, such as a fixed disk. Volatile media include dynamic memory, such as system RAM. Transmission media include coaxial cables, copper wire and fiber optics, among others, including the wires that comprise one aspect of a bus. Transmission media also can take the form of acoustic or light waves, such as those generated during radio frequency (RF) and infrared (IR) data communications. Common forms of computer-readable media include, for example, a flexible disk, a hard disk, magnetic tape, any other magnetic medium, a CD-ROM disk, digital video disk (DVD), any other optical medium, any other physical medium with patterns of marks or holes, a RAM, a PROM, an EPROM, an EEPROM, a FLASH EPROM, any other memory chip or data exchange adapter, a carrier wave, or any other medium from which a computer can read.

[0232] Various forms of computer-readable media may be involved in carrying one or more sequences of one or more instructions to a CPU for execution. A bus carries the data to system RAM, from which a CPU retrieves and executes the instructions. The instructions received by system RAM can optionally be stored on a fixed disk either before or after execution by a CPU.

[0233] Computer program code for carrying out operations for aspects of the present technology may be written in any combination of one or more programming languages, including an object-oriented programming language such as Java, Smalltalk, C++, or the like

and conventional procedural programming languages, such as the "C" programming language, Go, Python, or other programming languages, including assembly languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

[0234] Examples of the method according to various aspects of the present disclosure are provided below in the following numbered clauses. An aspect of the method may include any one or more than one, and any combination of, the numbered clauses described below.

[0235] Clause 1: A computer-implemented method, comprising: defining a window based on a time range; adding an operator to the window; adding a new event from a data stream into a series store, wherein the series store stores events, wherein the events from the data stream comprise values and timestamps, and wherein the events are sorted in the series store based on the timestamps; sliding the window along the series store based on the timestamps to capture the new event and the events preceding the new event up to a limit of the time range; updating a state of the operator based on the values of the events captured by the window; modifying the new event based on the state of the operator to generate a modified event; and outputting the modified event.

[0236] Clause 2: The computer-implemented method of Clause 1, comprising: adding a subsequent new event into the series store after adding the new event, wherein the new event comprises a first timestamp, and wherein the subsequent new event comprises a second timestamp that is not newer than the first timestamp; sliding the window along the series store based on the timestamps to capture the subsequent new event and the events preceding the subsequent new event up to the limit of the time range, wherein the window does not capture the new event; and updating the state of the operator based on the values of the events captured by the window.

[0237] Clause 3: The computer-implemented method of any of Clauses 1-2, comprising expiring and deleting from the series store an event that is not captured by the window.

[0238] Clause 4: The computer-implemented method of any of Clauses 1-4, comprising compressing the values stored in the series store.

[0239] Clause 5: The computer-implemented method of any of Clauses 1-4, comprising

optimizing storage of the timestamps in the series store, wherein the optimizing comprises: storing a full timestamp for a first added event; storing a difference between a timestamp of a current event and a timestamp of a previous event for events added into the series store subsequent to the first added event; and reconstructing a full timestamp for the events added to the series store subsequent to the first added event by summing the stored differences.

[0240] Clause 6: The computer-implemented method of any of Clauses 1-5, wherein the window comprises a start pointer to point to a first one of the events stored in the series store and an end pointer to point to a second one of the events stored in the series store.

[0241] Clause 7: The computer-implemented method of Clause 6, wherein the sliding the window along the series store comprises incrementing or decrementing at least one of the start pointer or the end pointer.

[0242] Clause 8: The computer-implemented method of Clause 7, wherein the updating the state of the operator based on the values of the events captured by the window comprises incrementally updating the state of the operator based on the incrementing or of the decrementing at least one of the start pointer or the end pointer.

[0243] Clause 9: The computer-implemented method of any of Clauses 1-8, wherein the operator defines a function, and wherein the updating the state of the operator based on the values of the events captured by the window comprises computing the function using the values captured by the window.

[0244] Clause 10: The computer-implemented method of Clause 9, wherein the new event is an in-order event, and wherein the in-order event comprises a timestamp that is newer than the timestamps of the other events stored in the series store, the method further comprising incrementally computing the function based on sliding the window along the series store.

[0245] Clause 11: The computer-implemented method of Clause 9, wherein the new event is an out-of-order event, and wherein the out-of-order event comprises a timestamp that is not newer than at least one of the timestamps of the other events stored in the series store, the method further comprising invertibly computing the function based on sliding the window along the series store.

[0246] Clause 12: The computer-implemented method of any of Clauses 1-11, wherein an underlying data structure of the series store is an unrolled linked list with adaptive sized

memory portions.

[0247] Clause 13: A server comprising a processor and a memory, wherein the memory comprises instructions executable by the processor to implement an enrichment model for modifying a data stream for use with a machine learning model, the enrichment model comprising: a series store configured to store events received by the enrichment model from the data stream, wherein the events comprise values and timestamps, and wherein the events are sorted in the series store based on the timestamps; a window configured to capture at least a portion of the events stored by the series store, wherein the window is defined by a predetermined range, and wherein the window is slidable along the series store to an updated position based on the receiving new event; and an operator corresponding to the window, wherein the operator is configured to change state based on the values of the events captured by the window, and wherein the enrichment model is configured to modify the new event based on the state of the operator at the updated position of the window.

[0248] Clause 14: The server of Clause 13, wherein the series store is configured to delete an event that is not captured by the window in the updated position.

[0249] Clause 15: The server of any of Clause 13-14, wherein the series store is configured to: store a full timestamp for a first received event of the events; store differences between timestamps associated with events received by the enrichment model subsequent to the first received event; and reconstruct full timestamps for the events received by the enrichment model subsequent to the first received event based on the stored differences.

[0250] Clause 16: The server of any of Clauses 13-15, wherein the window comprises a start pointer to point to a first one of the events stored in the series store and an end pointer to point to a second one of the events stored in the series store.

[0251] Clause 17: The server of any of Clauses 13-16, wherein the operator defines a function, and wherein the operator is configured to change state by computing the function using the values of the events captured by the window.

[0252] Clause 18: The server of Clause 17, wherein the operator is configured to incrementally compute the function for the updated position of the window based on the new event being an in-order event, and wherein the in-order event comprises a timestamp that is newer than the timestamps of the other events stored in the series store.

[0253] Clause 19: The server of Clause 17, wherein the operator is configured to invertibly compute the function for the updated position of the window based on the new

event being an out-of-order event, and wherein the out-of-order event comprises a timestamp that is not newer than at least one of the timestamps of the other events stored in the series store.

[0254] Clause 20. A computer-implemented method, comprising: receiving a first new event from a data stream, wherein the first new event comprises a first timestamp; sorting the first new event into a series store based on the first timestamp, wherein the series store stores events from the data stream; setting a window along the series store at a first position to capture the first new event and the events preceding the first new event up to a predetermined time range; computing a function based on the events captured by the window in the first position to generate a first modified event; receiving a second new event from the data stream after receiving the first new event, wherein the second new event comprises a second timestamp, and wherein the second timestamp is earlier than the first timestamp; sorting the second new event into the series store based on the second timestamp; setting the window along the series store at a second position to capture the second new event and the events preceding the second new event up to the predetermined time range; and invertibly computing the function based on the events captured by the window in the second position to generate a second modified event.

[0255] The foregoing detailed description has set forth various forms of the systems and/or processes via the use of block diagrams, flowcharts, and/or examples. Insofar as such block diagrams, flowcharts, and/or examples contain one or more functions and/or operations, it will be understood by those within the art that each function and/or operation within such block diagrams, flowcharts, and/or examples can be implemented, individually and/or collectively, by a wide range of hardware, software, firmware, or virtually any combination thereof. Those skilled in the art will recognize that some aspects of the forms disclosed herein, in whole or in part, can be equivalently implemented in integrated circuits, as one or more computer programs running on one or more computers (e.g., as one or more programs running on one or more computer systems), as one or more programs running on one or more processors (e.g., as one or more programs running on one or more microprocessors), as firmware, or as virtually any combination thereof, and that designing the circuitry and/or writing the code for the software and or firmware would be well within the skill of one of skill in the art in light of this disclosure. In addition, those skilled in the art will appreciate that the mechanisms of the subject matter described herein are capable of being distributed as one or more program products in a variety of forms, and that an illustrative form of the subject matter described herein applies regardless of the particular type of signal bearing medium used to actually carry out the distribution.

[0256] Instructions used to program logic to perform various disclosed aspects can be stored within a memory in the system, such as dynamic random-access memory (DRAM), cache, flash memory, or other storage. Furthermore, the instructions can be distributed via a network or by way of other computer readable media. Thus a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, compact disc, read-only memory (CD-ROMs), and magneto-optical disks, read-only memory (ROMs), random access memory (RAM), erasable programmable read-only memory (EPROM), electrically erasable programmable read-only memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the non-transitory computer-readable medium includes any type of tangible machine-readable medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer).

[0257] Any of the software components or functions described in this application, may be implemented as software code to be executed by a processor using any suitable computer language such as, for example, Python, Java, C++ or Perl using, for example, conventional or object-oriented techniques. The software code may be stored as a series of instructions, or commands on a computer readable medium, such as RAM, ROM, a magnetic medium such as a hard-drive or a floppy disk, or an optical medium such as a CD-ROM. Any such computer readable medium may reside on or within a single computational apparatus and may be present on or within different computational apparatuses within a system or network.

[0258] As used in any aspect herein, the term “logic” may refer to an app, software, firmware and/or circuitry configured to perform any of the aforementioned operations. Software may be embodied as a software package, code, instructions, instruction sets and/or data recorded on non-transitory computer readable storage medium. Firmware may be embodied as code, instructions or instruction sets and/or data that are hard-coded (e.g., nonvolatile) in memory devices.

[0259] As used in any aspect herein, the terms “component,” “system,” “module” and the like can refer to a computer-related entity, either hardware, a combination of hardware and software, software, or software in execution.

[0260] As used in any aspect herein, an “algorithm” refers to a self-consistent sequence of steps leading to a desired result, where a “step” refers to a manipulation of physical

quantities and/or logic states which may, though need not necessarily, take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It is common usage to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. These and similar terms may be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities and/or states.

[0261] A network may include a packet switched network. The communication devices may be capable of communicating with each other using a selected packet switched network communications protocol. One example communications protocol may include an Ethernet communications protocol which may be capable of permitting communication using a Transmission Control Protocol/Internet Protocol (TCP/IP). The Ethernet protocol may comply or be compatible with the Ethernet standard published by the Institute of Electrical and Electronics Engineers (IEEE) titled "IEEE 802.3 Standard", published in December 2008 and/or later versions of this standard. Alternatively or additionally, the communication devices may be capable of communicating with each other using an X.25 communications protocol. The X.25 communications protocol may comply or be compatible with a standard promulgated by the International Telecommunication Union-Telecommunication Standardization Sector (ITU-T). Alternatively, or additionally, the communication devices may be capable of communicating with each other using a frame relay communications protocol. The frame relay communications protocol may comply or be compatible with a standard promulgated by Consultative Committee for International Telegraph and Telephone (CCITT) and/or the American National Standards Institute (ANSI). Alternatively, or additionally, the transceivers may be capable of communicating with each other using an Asynchronous Transfer Mode (ATM) communications protocol. The ATM communications protocol may comply or be compatible with an ATM standard published by the ATM Forum titled "ATM-MPLS Network Interworking 2.0" published August 2001, and/or later versions of this standard. Of course, different and/or after-developed connection-oriented network communication protocols are equally contemplated herein.

[0262] Unless specifically stated otherwise as apparent from the foregoing disclosure, it is appreciated that, throughout the present disclosure, discussions using terms such as "processing," "computing," "calculating," "determining," "displaying," or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[0263] One or more components may be referred to herein as “configured to,” “configurable to,” “operable/operative to,” “adapted/adaptable,” “able to,” “conformable/conformed to,” etc. Those skilled in the art will recognize that “configured to” can generally encompass active-state components and/or inactive-state components and/or standby-state components, unless context requires otherwise.

[0264] Those skilled in the art will recognize that, in general, terms used herein, and especially in the appended claims (e.g., bodies of the appended claims) are generally intended as “open” terms (e.g., the term “including” should be interpreted as “including but not limited to,” the term “having” should be interpreted as “having at least,” the term “includes” should be interpreted as “includes but is not limited to,” etc.). It will be further understood by those within the art that if a specific number of an introduced claim recitation is intended, such an intent will be explicitly recited in the claim, and in the absence of such recitation no such intent is present. For example, as an aid to understanding, the following appended claims may contain usage of the introductory phrases “at least one” and “one or more” to introduce claim recitations. However, the use of such phrases should not be construed to imply that the introduction of a claim recitation by the indefinite articles “a” or “an” limits any particular claim containing such introduced claim recitation to claims containing only one such recitation, even when the same claim includes the introductory phrases “one or more” or “at least one” and indefinite articles such as “a” or “an” (e.g., “a” and/or “an” should typically be interpreted to mean “at least one” or “one or more”); the same holds true for the use of definite articles used to introduce claim recitations.

[0265] In addition, even if a specific number of an introduced claim recitation is explicitly recited, those skilled in the art will recognize that such recitation should typically be interpreted to mean at least the recited number (e.g., the bare recitation of “two recitations,” without other modifiers, typically means at least two recitations, or two or more recitations). Furthermore, in those instances where a convention analogous to “at least one of A, B, and C, etc.” is used, in general such a construction is intended in the sense one having skill in the art would understand the convention (e.g., “a system having at least one of A, B, and C” would include but not be limited to systems that have A alone, B alone, C alone, A and B together, A and C together, B and C together, and/or A, B, and C together, etc.). In those instances where a convention analogous to “at least one of A, B, or C, etc.” is used, in general such a construction is intended in the sense one having skill in the art would understand the convention (e.g., “a system having at least one of A, B, or C” would include but not be limited to systems that have A alone, B alone, C alone, A and B together, A and C together, B and C together, and/or A, B, and C together, etc.). It will be further understood by those within the art that typically a disjunctive word and/or phrase presenting two or more

alternative terms, whether in the description, claims, or drawings, should be understood to contemplate the possibilities of including one of the terms, either of the terms, or both terms unless context dictates otherwise. For example, the phrase “A or B” will be typically understood to include the possibilities of “A” or “B” or “A and B.”

[0266] With respect to the appended claims, those skilled in the art will appreciate that recited operations therein may generally be performed in any order. Also, although various operational flow diagrams are presented in a sequence(s), it should be understood that the various operations may be performed in other orders than those which are illustrated or may be performed concurrently. Examples of such alternate orderings may include overlapping, interleaved, interrupted, reordered, incremental, preparatory, supplemental, simultaneous, reverse, or other variant orderings, unless context dictates otherwise. Furthermore, terms like “responsive to,” “related to,” or other past-tense adjectives are generally not intended to exclude such variants, unless context dictates otherwise.

[0267] It is worthy to note that any reference to “one aspect,” “an aspect,” “an exemplification,” “one exemplification,” and the like means that a particular feature, structure, or characteristic described in connection with the aspect is included in at least one aspect. Thus, appearances of the phrases “in one aspect,” “in an aspect,” “in an exemplification,” and “in one exemplification” in various places throughout the specification are not necessarily all referring to the same aspect. Furthermore, the particular features, structures or characteristics may be combined in any suitable manner in one or more aspects.

[0268] As used herein, the singular form of “a,” “an,” and “the” include the plural references unless the context clearly dictates otherwise.

[0269] Any patent application, patent, non-patent publication, or other disclosure material referred to in this specification and/or listed in any Application Data Sheet is incorporated by reference herein, to the extent that the incorporated materials is not inconsistent herewith. As such, and to the extent necessary, the disclosure as explicitly set forth herein supersedes any conflicting material incorporated herein by reference. Any material, or portion thereof, that is said to be incorporated by reference herein, but which conflicts with existing definitions, statements, or other disclosure material set forth herein will only be incorporated to the extent that no conflict arises between that incorporated material and the existing disclosure material. None is admitted to be prior art.

[0270] In summary, numerous benefits have been described which result from employing the concepts described herein. The foregoing description of the one or more forms has been presented for purposes of illustration and description. It is not intended to be exhaustive or limiting to the precise form disclosed. Modifications or variations are possible

in light of the above teachings. The one or more forms were chosen and described in order to illustrate principles and practical application to thereby enable one of ordinary skill in the art to utilize the various forms and with various modifications as are suited to the particular use contemplated. It is intended that the claims submitted herewith define the overall scope.

CLAIMS

What is claimed is:

1. A computer-implemented, comprising:
 - defining a window based on a time range;
 - adding an operator to the window;
 - adding a new event from a data stream into a series store, wherein the series store stores events from the data stream, wherein the events comprise values and timestamps, and wherein the events are sorted in the series store based on the timestamps;
 - sliding the window along the series store based on the timestamps to capture the new event and the events preceding the new event up to a limit of the time range;
 - updating a state of the operator based on the values of the events captured by the window;
 - modifying the new event based on the state of the operator to generate a modified event; and
 - outputting the modified event.
2. The computer-implemented method of claim 1, comprising:
 - adding a subsequent new event into the series store after adding the new event, wherein the new event comprises a first timestamp, and wherein the subsequent new event comprises a second timestamp that is not newer than the first timestamp;
 - sliding the window along the series store based on the timestamps to capture the subsequent new event and the events preceding the subsequent new event up to the limit of the time range, wherein the window does not capture the new event; and
 - updating the state of the operator based on the values of the events captured by the window.
3. The computer-implemented method of claim 1, comprising expiring and deleting from the series store an event that is not captured by the window.
4. The computer-implemented method of claim 1, comprising compressing the values stored in the series store.

5. The computer-implemented method of claim 1, comprising optimizing storage of the timestamps in the series store, wherein the optimizing comprises:
 - storing a full timestamp for a first added event;
 - storing a difference between a timestamp of a current event and a timestamp of a previous event for events added to the series store subsequent to the first added event; and
 - reconstructing a full timestamp for the events added to the series store subsequent to the first added event by summing the stored differences.
6. The computer-implemented method of claim 1, wherein the window comprises a start pointer to point to a first one of the events stored in the series store and an end pointer to point to a second one of the events stored in the series store.
7. The computer-implemented method of claim 6, wherein the sliding the window along the series store comprises incrementing or decrementing at least one of the start pointer or the end pointer.
8. The computer-implemented method of claim 7, wherein the updating the state of the operator based on the values of the events captured by the window comprises incrementally updating the state of the operator based on the incrementing or decrementing of the at least one of the start pointer or the end pointer.
9. The computer-implemented method of claim 1, wherein the operator defines a function, and wherein the updating the state of the operator based on the values of the events captured by the window comprises computing the function using the values captured by the window.
10. The computer-implemented method of claim 9, wherein the new event is an in-order event, and wherein the in-order event comprises a timestamp that is newer than the timestamps of the other events stored in the series store, the method further comprising incrementally computing the function based on sliding the window along the series store.

11. The computer-implemented method of claim 9, wherein the new event is an out-of-order event, and wherein the out-of-order event comprises a timestamp that is not newer than at least one of the timestamps of the other events stored in the series store, the method further comprising invertibly computing the function based on sliding the window along the series store.

12. The computer-implemented method of claim 1, wherein an underlying data structure of the series store is an unrolled linked list with adaptive sized memory portions.

13. A server comprising a processor and a memory, wherein the memory comprises instructions executable by the processor to implement an enrichment model for modifying a data stream for use with a machine learning model, the enrichment model comprising:

a series store configured to store events received by the enrichment model from the data stream, wherein the events comprise values and timestamps, and wherein the events are sorted in the series store based on the timestamps;

a window configured to capture at least a portion of the events stored by the series store, wherein the window is defined by a predetermined range, and wherein the window is slidable along the series store to an updated position based on receiving a new event; and

an operator corresponding to the window, wherein the operator is configured to change state based on the values of the events captured by the window, and wherein the enrichment model is configured to modify the new event based on the state of the operator at the updated position of the window.

14. The server of claim 13, wherein the series store is configured to delete an event that is not captured by the window in the updated position.

15. The server of claim 13, wherein the series store is configured to:

store a full timestamp for a first received event of the events;

store differences between timestamps associated with events received by the enrichment model subsequent to the first received event; and

reconstruct full timestamps for the events received by the enrichment model subsequent to the first received event based on the stored differences.

16. The server of claim 13, wherein the window comprises a start pointer to point to a first one of the events stored in the series store and an end pointer to point to a second one of the events stored in the series store.
17. The server of claim 13, wherein the operator defines a function, and wherein the operator is configured to change state by computing the function using the values of the events captured by the window.
18. The server of claim 17, wherein the operator is configured to incrementally compute the function for the updated position of the window based on the new event being an in-order event, and wherein the in-order event comprises a timestamp that is newer than the timestamps of the other events stored in the series store.
19. The server of claim 17, wherein the operator is configured to invertibly compute the function for the updated position of the window based on the new event being an out-of-order event, and wherein the out-of-order event comprises a timestamp that is not newer than at least one of the timestamps of the other events stored in the series store.
20. A computer-implemented method, comprising:
- receiving a first new event from a data stream, wherein the first new event comprises a first timestamp;
 - sorting the first new event into a series store based on the first timestamp, wherein the series store stores events from the data stream;
 - setting a window along the series store at a first position to capture the first new event and the events preceding the first new event up to a predetermined time range;
 - computing a function based on the events captured by the window in the first position to generate a first modified event;
 - receiving a second new event from the data stream after receiving the first new event, wherein the second new event comprises a second timestamp, and wherein the second timestamp is earlier than the first timestamp;
 - sorting the second new event into the series store based on the second timestamp;

setting the window along the series store at a second position to capture the second new event and the events preceding the second new event up to the predetermined time range; and

invertibly computing the function based on the events captured by the window in the second position to generate a second modified event.