DOI: http://dx.doi.org/10.26483/ijarcs.v8i7.4483

ISSN No. 0976-5697

Volume 8, No. 7, July – August 2017

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

REAL TIME DATA STREAM AGGREGATION AND WINDOWING

Aditya Kumar Shukla
Tech Lead(Perspica, Noida) /MTech-CSE
Dr. K. N. Modi University
Newai,Tonk, Rajasthan, India

Abstract : Nowadays Real time data analysis and Real time data processing is in demand and its demand is getting increase day by day, as we know Real time data processing is the continuous data stream processing and the data stream could be flow in rate of millions data points per seconds. This real time data stream information which will be further used for various purposes, it could be for commercial use or for scientific use .while processing the real time data stream proper methodologies with good approximations have to be used for stream data preparation and aggregation .In my paper, I have also provided the study about data stream aggregation techniques and their characteristics to improve the performance. Here we would discuss the way of stream aggregation and time based grouping (Windowing) using suitable techniques, the important thing in data stream aggregation is to aggregate right set of data and for using windowing of stream for aggregation, Here we would discuss different windowing techniques.

Keywords: stream aggregation, time windowing, data preparation, data processing, data aggregation, data pipeline

I. INTRODUCTION

The Real-time data stream processing, allows data to be processed as soon as it is made available, without the need of any disk and memory storage system. Data stream is a flow of data which can be consist only time, value or other stream attribute, the stream attributes and characteristic In a Survey[1] of a European company in 2013, the 70% of total participators were in interested and shown need for real time data processing. And from that time, multiple works has been done in stream processing technology and major roll of development has been paid in rapid development of modern stream processing technologies by the open source software community. Many applications need to process streams, for example, application performance monitoring, weather prediction, financial data analysis, network traffic monitoring, and telecommunication monitoring. Several database research groups are building Data Stream Management Systems (DSMS) so that applications can issue queries to get timely information from streams.

The common use of stream processing in now days in application performance monitoring, and a very common task is that of monitoring a large area in regards to some physical sensed value such as cpu, memory, IO and Disk usage. Usually the companies and system admin wants to examine and analyse this information, so they are collect the data through various systems like Vmware Vcenter, collectd, Grafite, and various open source libraries which exposes performance monitoring, the collected continuous data stream collects in a central point we call the sink node. Due to the large amount of data transmitted as the scale of the network increases, it is important to combine several data metric points in intermediate nodes along the way towards the processing system conserve data. This process is known as data aggregation. However due to the nature of the data stream data points can be receive out of order so to analyse this out of order data is error-prone and the aggregation techniques used must be aware and resilient to errors in message transmission.

.

II. REQUIREMENTS REAL TIME DATA STREAM PROCESSING

The basic requirements [2] for real time data stream process to accurate and high performance.

- A real-time stream processing system is to process messages “in-stream”, without any requirement to store them to perform any operation or sequence of operations. Ideally the system should also use an active (i.e., non-polling) processing model [2].
- Support a high-level query language with built-in extensible stream oriented primitives and operators.
- Built-in mechanisms to provide resiliency against stream “imperfections”, including missing and out-of-order data, which are commonly present in real-world data streams.
- Stream processing engine must guarantee predictable and repeatable outcomes.
- Capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data.
- Ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures.
- Capability to distribute processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.
- Stream processing system must have a highly optimised, minimal-overhead execution engine to deliver real-time response for high-volume applications.

In these requirements the first thing is to handle out of order data and to accurate aggregation of data stream. The computations on data streams can be challenging due to multiple reasons, including the size of a dataset.

.

III. STREAM AGGREGATION AND SCALABILITY

Stream data aggregation is a process in which information is gathered and expressed in a summary form, for purposes such as statistical analysis. A common data aggregation[3] purpose is to get more information about particular data points groups based on specific variable such as time, attribute, or metric type.

For statistical analysis of certain metrics of data stream the common case is to calculate sum, count, average, mean, percentile and rate of data set over specific time period, and for calculation of these parameters some time need to iterate over the entire dataset in a sorted order using standard formula/ practices and they may not be the most suited approach, for example, mean = sum of value/ count. For a streaming dataset, this is not fully scalable.

Instead, suppose we store the sum and count and each new data-point is added to the sum. For every new point, we increment the count, and whenever we need the average, we divide the sum by the count. Then we get the mean at that instance.

Some time we need aggregated percentile of data stream[4], while percentile requires finding the location of an data point in a large dataset; for example, 90th percentile would mean the value that is over 90 percent of the values in a sorted dataset. To illustrate, in [9, 1, 8, 7, 6, 5, 2, 4, 3, 0], the 80th percentile would be 8. This means we need to sort the dataset and then find an item by its location. This clearly is not scalable. Scaling this operation involves using an algorithm called tdigest[5]. This is a way of approximating percentile at scale. tdigest[5] creates digests that create centroids at positions that are approximated at the appropriate quantiles. These digests can be added to get a complete digest that can be used to estimate the quantiles of the whole dataset.

Let's assume that a data centre which consist of different type of devices like load balancer, virtual machine, host-systems etc and these devices are managed through inbuilt vcenter plug-in. assume that a device is generating continues stream time series

$S = (a_1,a_2,a_3.........a_n)$ of real data points with time, value and name of metric for example cpu_usage , indexed by time t (time series), At any time t, we need to be able to succinctly and efficiently answer statistical aggregate queries regarding the underlying stream up to time t, i.e. $S_t = (a_1, a_2, . . . , a_t)$.

As we've mentioned before, we ideally want to perform that in (poly)logarithmic space and time, respecting the computational and storage capabilities of our tiny devices. We achieve that by maintaining auxiliary data structures called summaries1 which are clever and succinct synopses of the underlying data set observed so far. The most important operation of a summary is that of merging new elements into it, in order to produce efficiently a new synopsis, describing the new, larger stream. More formally, we need a function F such that if $S_k(S_t)$ is a summary instance describing an underlying stream S_t , then

$S_k(S_t+1) = F(S_k(S_t), a_t+1)$.

For example, for the MAX and AVG aggregates it is trivial to see that can be efficiently computed by maintaining (recursively) summaries with

$S_kMAX(S_1) = a_1$, summary is that of merging new elements into it, in order to produce efficiently a new synopsis, describing the new, larger stream. More formally, we need a function F such that if $S_k(S_t)$ is a summary instance describing an underlying stream St , then

$S_k(S_t+1) = F(S_k(S_t), a_t+1)$

. For example, for the MAX and AVG aggregates it is trivial to see that can be efficiently computed by maintaining (recursively) summaries with

$S_kMAX(S_1) = a_1$, $F_{MAX}(x, y) = \max\{x, y\}$ and

$S_kAVG(S1) = (a_1, 1)$, $F_{AVG}((x_1, x_2), y) = (x_1 + y, x_2 + 1)$, respectively, where the actual mean value can be extracted by the AVG summary by simply computing

AVG(S) = x y , where $S_kAVG(S) = (x, y)$.

Let's take example of application performance metrics like cpu usages, if we have multiple points like

Table I. In memory Stream Aggregation

Metric name	*time(hh:mm:ss)*	*value*	*count*	*sum*	*Avg*	*max*
CPU Usage	11:30:01	23.4	1	23.4	23.4	23.4
CPU Usage	11:30:32	21.0	2	44.4	22.2	23.4

.

A. *Time based aggregation*

As we discussed in data stream aggregation section , the most important part in data aggregation is to aggregate on the basis of time.In case of application performance metrics example embedded time in data messages is like epoch timestamp and it is in milliseconds.

For aggregation of metric stream, each data element in the stream needs to be associated with a timestamp. When we say "events in the last 5 minutes" - which 5 minutes do we mean? This can be done in three ways:[6]

- event-time - a logical, data-dependent timestamp, embedded in the event (data element) itself
- ingestion-time - a timestamp assigned to the event when it enters the system
- processing-time - the wall-clock time when the event is processed

Event-time, while most useful, is also the most troubling: it gives the least guarantees; events may arrive out of order or late, so we can never be sure if we saw all events in a given time window. Processing-time is easiest, as it is monotonic: you know precisely when a 5-minute window ended (by looking at the clock), but also much less useful.

The selection of time depends on the use case and requirement of further processing and analysis.

IV. WINDOWING CONCEPT IN DATA STREAM

The data analysis space is witnessing an evolution from batch to stream processing for many use cases. Although batch can be handled as a special case of stream processing, analyzing never-ending streaming data often requires a shift in the mindset and comes with its own terminology (for example, "windowing" and "at-least-once"/"exactly-once" processing).

Instead of using synopses to compress the characteristics of the whole data streams, window techniques[7] only look on a portion of the data. This approach is motivated by the idea that only the most recent data are relevant. Therefore, a window continuously cuts out a part of the data stream, e.g. the last ten data stream elements, and only considers these elements during the processing. There are different kinds of such windows like sliding windows that are similar to FIFO lists or tumbling windows that cut out disjoint parts. Furthermore, the windows can also be differentiated into element-based windows, e.g., to consider the last ten elements, or time-based windows, e.g., to consider the last ten seconds of data. There are also different approaches to implementing windows. There are, for example, approaches that use timestamps or time intervals for system-wide windows or buffer-based windows for each single processing step. Sliding-window query processing is also suitable to being implemented in parallel processors by exploiting parallelism between different windows and/or within each window extent..

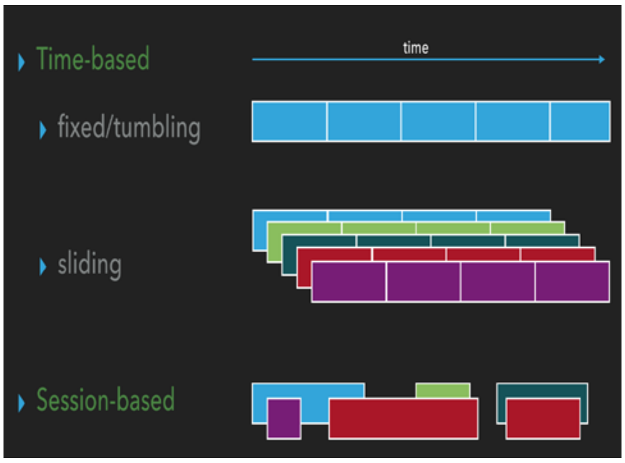

Figure 1. Wndowing Examples

A. *Type of Windows:[8]*

- fixed/tumbling: time is partitioned into same-length, non-overlapping chunks. Each event belongs to exactly one window

- sliding: windows have fixed length, but are separated by a time interval (step) which can be smaller than the window length. Typically the window interval is a multiplicity of the step. Each event belongs to a number of windows.

- session: windows have various sizes and are defined basing on data, which should carry some session identifiers.

B. *Out of order data handling:[9].*

As we discuss event-time, events can arrive out of order. For example in IoT, when you are receiving a stream of sensor readings, devices might be offline, and send catch-up data after some time. That's why we definitely have to allow for some lateness in event arrival, but how much? We can't keep all windows around forever, as this would eat all available memory. At some point, a window has to be considered "done" and garbage collected.

This is handled by a mechanism called watermarks. A watermark specifies that we assume that all events before X have been observed. This is of course a heuristic, as we usually can't know that for sure. The heuristic has to be picked so that it strikes a good balance between including as much late data as possible and not delaying final window processing too much.

Any events older than the current watermark are dropped. An example of a heuristic is a watermark that is always 5 minutes behind the newest event time seen in an event; that is, we allow data to be up to 5 minutes late.

Once we accumulate data (events) in a window, to get value, it needs to be somehow manipulated. There are a number of options:

- basic operations such as map, filter, flatMap, ...
- Aggregate: count, max, min, sum, etc.
- fold/reduce using an arbitrary function

This allows us to get a window result value for each window.

V. Related Work

Let's now compare a couple of popular systems and see how they classify when it comes to windowing data taking into account the above mentioned aspects.

We'll take a look at Spark, Flink, and Kafka Streams. It's by no means a comprehensive list - there are many more streaming systems out there, but these seem to be quite popular.

.

A. *Spark Streaming[10]*

Spark Streaming is one of the most popular options out there, present on the market for quite a long time, allowing processing a stream of data on a Spark cluster. It builds on the usual Spark execution engine, where the main abstraction is the RDD: Resilient Distributed Dataset (you can think about it as a replicated, parallelised collection). In Spark Streaming, the main abstraction is a DStream: a Discretized stream. A DStream is defined by an interval (e.g. 1 second), which is used to pre-group the incoming stream elements into discrete chunks. Each chunk forms an RDD and is processed by the "normal" Spark execution engine. Hence this is not "true" streaming, but micro-batching. However, you can implement quite a lot of streaming operations on top of such architecture. The DStream.window() API has the following capabilities:

- tumbling/sliding windows
- only processing-time; no event-time support
- no watermarks support (which wouldn't make sense with processing-time anyway)
- triggers at the end of the window only

B. *Flink[11]*

Apache Flink reifies a lot of the concepts described in the introduction as user-implementable classes/interfaces. Like Spark, Flink processes the stream on its own cluster. Note that most of these operations are available only on keyed streams (streams grouped by a key), which allows them to be run in parallel. The interfaces involved are:

- Time Characteristic: enumeration of Event, Ingestion, Processing.
- Timestamp Assigner: assigns timestamps to events (when using event-time), but also generates watermarks. There are some built-in options, like generating a watermark in specified event-time intervals, but custom implementations can be provided
- Window Assigner: for each data element, assign windows corresponding to it. Built-in options: tumbling/sliding/global windows. A custom implementation can be used to implement session windows.
- Trigger: event/processing time (when watermark is passed), continuous event/processing time (based on an interval), element count

.

C. *Kafka Stream[12]*

Apache Kafka, being a distributed streaming platform with a messaging system at its core, contains a client-side component for manipulating data streams. The data sources and sinks are Kafka topics. Like in previous cases, Kafka Streams also allows to run stream processing computations in parallel on a cluster, however that cluster has to be managed externally. Like with any other Kafka stream consumer, multiple instances of a stream processing pipeline can be started and they divide the work.

As for windowing, Kafka has the following options:[13]

- TimestampExtractor allows to use event, ingestion or processing time for any event
- windows can be tumbling or sliding
- There are no built-in watermarks, but window data will be retained for 1 day (by default)
- trigger: after every element. The results are stored in an ever-updating KTable. A KTable is table represented as a stream of row updates; in other ways, a changelog stream. The each-element triggering can be a problem if the window function is expensive to compute and doesn't have a straightforward "accumulative" nature.

The options here are much more modest comparing to Flink, but the processing and clustering models are simple to understand, which is definitely a plus when designing a system.

VI. CONCLUSION

As we have seen, the present system varies widely in how data can be windowed. Some offer only the basics, like Spark Streaming, some have a very wide range of windowing features, such as Flink .However, keep in mind that windowing in only one of the aspects of a stream processing engine.

Another important aspect to aggregate data in window in right-way, to use right key of aggregation, that can be grouped over right set of keys so out of order data can be handle in window itself.

The best way of windowing is to use tumbling window if out of order is an issue for further processing, there should be use of embedded event time to identified out of order data.

We would suggest to convert embedded time stamp into at-least one minute granularity so adequate data points can be use for aggregation if we talk performance metrics monitoring data which comes in milliseconds granularity.

VII. ACKNOWLEDGMENT

I take this opportunity to express a deep sense of gratitude for my guide **Mr. Vivek Sharma (H.O.D), Department of CSE**, DKNMU and My organization's seniors for providing excellent guidance and support. It is because of her constant and general interest and assistance that this paper has been successful.

I would also like to thank my teachers, family and friends who have been a source of encouragement and inspiration throughout the duration of this project.

VIII. REFERENCES

[1] Eurofound, European Company Surveys (ECS)2013

[2] Michael Stonebraker , Uğur Çetintemel, Stan Zdonik. The 8 Requirements of Real-Time Stream Processing

[3] Searchsqlserver.techtarget.com,Data aggregation, September 2005

[4] Anant Asthana, Real-Time Aggregation on Streaming Data Using Spark Streaming and Kafka, April 11th, 2016

[5] Cameron Davidson-Pilon, Percentile and Quantile Estimation of Big Data: The t-Digest, Mar 18, 2015

[6] Adam Warski, Windowing data in Big Data Streams - Spark, Flink, Kafka, Akka, 2016

[7] Abadi; et al. Aurora: A Data Stream Management System. SIGMOD 2003.

[8] Adam Warski, Types of window- Spark, Flink, Kafka, Akka, 2016

[9] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner and Murali Mani, Event Stream Processing with Out-of-Order Data Arrival

[10] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica, Discretized Streams: Fault-Tolerant Streaming Computation at Scale, 2013

[11] K.M.J. Jacobs, Apache Flink: Distributed Stream Data Processing, 2016

[12] Martin Kleppmann, Jay Kreps, Kafka, Samza and the Unix Philosophy of Distributed Data,

[13] NIXON PATEL, Chief Data Scientist, Brillio LLC, REAL TIME ANALYTICS WITH SPARK AND KAFKA, 2015