Research and Analysis of the Stream Materialized Aggregate List

Marcin Gorawski(✉) and Krzysztof Pasterak

Silesian University of Technology,
Institute of Computer Science,
Akademicka 16, 44-100 Gliwice Poland
{Marcin.Gorawski,Krzysztof.Pasterak}@polsl.pl

Abstract. The problem of low-latency processing of large amounts of data acquired in continuously changing environment has led to the genesis of Stream Processing Systems (SPS). However, sometimes it is crucial to process both historical (archived) and current data, in order to obtain full knowledge about various phenomena. This is achieved in a Stream Data Warehouse (StrDW), where analytical operations on both historical and current data streams are performed. In this paper we focus on Stream Materialized Aggregate List (StrMAL) – a stream repository tier of StrDW. As a motivating example, the liquefied petrol storage and distribution system, containing continuous telemetric data acquisition, transmission and storage, will be presented as possible application for Stream Materialized Aggregate List.

Keywords: Materialized aggregate list · Stream data warehouse · Stream processing

1 Introduction

Nowadays, the necessity of processing, storing and analyzing of very large data volumes (considered also as *BigData*) is constantly growing. This implies development of newer and more advanced systems, that are able to satisfy this need. Moreover, from the perspective of various enterprises, organizations and other data producers and consumers, the outcome information is expected to be reliable, most up-to-date and obtained in the shortest time possible. These requirements determine the attractiveness of solutions already present on market, as well as constitute new objectives for developers [22].

In the following paper we focus on Stream Data Processing Systems (SPS). They are designed to process current and continuously generated data with relatively high frequency. When non-stream solutions (i.e. those relying on persistent data) are concerned, processing unit enforces collecting data from sources. Stream oriented systems have to process incoming data almost instantly as they arrive, since data are produced and actively delivered by sources. There are representative examples of Stream Processing Systems [1–6, 26, 27], however they are relatively not as popular as classic, traditional data storage systems.

A. Amine et al. (Eds.): CIIA 2015, IFIP AICT 456, pp. 269–278, 2015.
DOI: 10.1007/978-3-319-19578-0_22

The example of application involving instant and immediate analysis of data delivered continuously is a liquefied petrol storage and distribution system. Such an installation consists of multiple petrol stations, where various measurements are gathered and transmitted to the centralized or distributed analysis platform.

Each petrol station is equipped with fuel tanks where liquefied fuel is stored and dispensing devices which act as sale endpoints. These appliances generate two streams of data supplemented with delivery records entered by station workers or detected automatically. Usually fuel volume and temperature is measured in tanks, whereas the amount of sold fuel is returned from meters installed in dispensers.

The common analysis performed upon the aforementioned values aims to detect various anomalies and other adverse phenomena that can occur at petrol stations. The most dangerous example is fuel leak [15, 24], which introduces very serious consequences to the environment. In order to prevent such a threat, it is crucial to detect any volume of fuel leaked from tank and piping as fast as possible.

This paper is organized as follows. Section 2 contains information concerning data stream storage problems with theoretical base of a Stream Materialized Aggregate List (StrMAL) described. In Section 3 the architecture of StrMAL is presented along with examples of its most important features. Section 4 contains test results performed over a StrMAL engine, whereas Section 5 summarizes the paper.

2 Data Stream Storage

A Data stream [8, 12–14] can be defined as an infinite sequence of tuples with unique timestamps and attributes carrying information describing various phenomena at subsequent moments of time. Stream Processing Systems usually do not provide any storage operation in their work flow, since they are designed to produce answers immediately as new data arrive. Optional data storage is sometimes used to provide static data as an extension to stream data.

Under certain circumstances an instant access to historical data stored in a database, as well as efficient processing performed on current data is required. Analyzing the history is necessary in learning process, where different trends, dependencies, and rules are discovered and remembered [21]. Later, gathered knowledge is used to filter current stream in order to detect any desired events. This process frequently involves browsing data on a certain level of detail – in other words – on different aggregation levels. Moreover, data retrieval and aggregation operations should not interfere with insertion of newly arrived data, which often cannot be completely eliminated.

The problem of processing both stored and current data has lead to the idea of the Stream Data Warehouse [7, 9, 19, 20, 23, 25]. It is an unified processing platform capable to produce immediate answers to complex queries concerning current and archived data. In current mode, data can be processed before they are persisted in any data structure, as in Stream Processing Systems.

2.1 Problems and Issues

As a consequence of data stream nature, it is virtually not possible to store a whole stream in a memory. In addition, at a given moment of time, the stream contains only the most current tuples, since all read before have been already removed and archived, which forces searching the history (database).

Moreover, data in a stream are produced relatively frequently and in a unpredictable manner, which causes database to be updated very often and irregularly. High intensity of modifying transactions, being made in parallel with queries consisting of large range data retrieval, can lead to serious decrease in overall performance.

Relative database systems are designed to execute versatile CRUD (Create, Read, Update and Delete) operations on the whole dataset stored inside their internal memory. However, in stream appliances, updates take place only at the end of a time frame, i.e. tuples arrive and are organized ascending by timestamps. In this cause, it is not possible that once stored piece of data is updated.

Repetitive execution of the same or similar operations (e.g. aggregation) of the same datasets is usually time consuming and thus leading to unnecessary delays. In order to prevent these adverse situations, results of time costing operations can be stored along with query parameters to provide access to once computed values. As mentioned before, there are no updates on historical data causing the materialized data to be immutable.

2.2 Stream Materialized Aggregate List

Many items that are sequentially arranged (as tuples in a data stream) can be stored in a list data structure. In such a form, it is easy to view all subsequent elements in proper order. When browsing tuples from a stream, consecutive retrieving is the only operation considered here, which can be described as forward iterating over list.

An aggregate list [16–18] can be defined as a sequential data structure, containing a subset of an aggregated stream (stream of aggregated tuples). It is stored in memory and acts as a physical representation of stream, beginning from certain moment of time. Because of the limited capacity of list, it is assumed that all aggregates already read can be replaced with more fresh data.

When considering various data collections, extracting data access operations into a separate interface is a common practice. Such an interface is called an *iterator* and is used to traverse any data structure (as aggregate list for example). Thus, an iterator can be used for retrieving tuples from a stream.

The aforementioned issues became a motivation for designing a solution which is capable to provide an uniform access to any data stream, efficiently manage available memory, and avoid redundant operations. It is done by using aggregate list, iterator interface, and aggregate materialization techniques. The solution has been named a Stream Materialized Aggregate List [11] and is designed to act as a data storage tier of a Stream Data Warehouse [10].

It is possible to create several iterators attached to a single list and pointing to different elements. In such a situation, the distance (measured in time units)

between them is unconstrained and can be arbitrary long, causing the whole list to occupy very large amount of memory. In order to prevent this situation, the following solution is used: the aggregate list itself is not located in memory, instead its active fragments (tuples being currently in use) are stored inside each iterator.

Moreover, in order to increase memory management efficiency, the following solution is used [17, 18]: each iterator contains a static array which corresponds to an iterator-specific aggregate list fragment. As far as successive aggregates are retrieved from an iterator and become outdated, they are replaced by newer ones. Each array is logically divided into pages (basic units). Due to that, certain number of ready-to-read aggregates is always available. When the need of new aggregates creation occurs, a whole page at once is produced and replaces the old one.

3 Architecture of StrMAL

The Stream Materialized Aggregate List was implemented using multilayer concept. Each of them is realizing separate functionalities and is responsible for another stage of aggregates production. Figure 1 presents the overall architecture of StrMAL. Four layers have been denoted by the following acronyms: SDL, APL, DML, and CL – they are described later in the text.

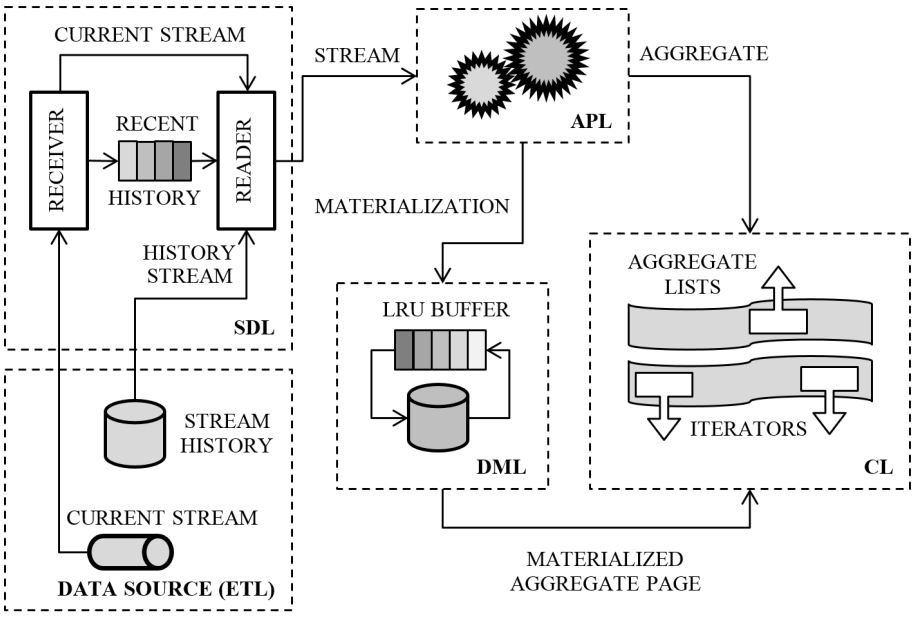

Fig. 1. Architecture of the StrMAL engine

In the Stream Distribution Layer (SDL) the process of aggregate list production begins with collecting data required for aggregation. It is done, depending on start time specified in query, by using current or historical stream. This layer provides a uniform access to data streams, irrespective of their origin (source) and start time.

The Aggregate Production Layer (APL) retrieves desired streams from the SDL and, basing on parameters obtained from client, performs aggregation. Outcome aggregates are delivered to clients and materialized (persisted for future use).

The Data Materialization Layer (DML) involves persisting aggregates in database, along with query parameters. Besides storing, this layer also provides searching and retrieving mechanisms. Cache memory (LRU buffer) is used to achieve better performance of I/O operations.

The Client Layer (CL) is responsible for communication with clients and providing them aggregates produced in the APL with data retrieved from the SDL or materialized aggregates read from the DML. It integrates all mentioned layers and uses them to prepare, produce, and serve results.

3.1 Current and Historical Stream Support

One of the major tasks of the Stream Distribution Layer is to collect tuples from current stream and store them temporally in a buffer called History Table (HT). It is performed in order to provide a flexible bridge between current and historical data. When the SDL is queried for a data stream beginning from a certain timestamp, first it performs a lookup over the HT to determine whether the desired data have been already produced – and when it is true – if they are still in the HT or have been persisted in a database.

The Stream Distribution Layer can operate in four states, depending on the distance between current and searched time. Each state determines the source from which data are retrieved and other working principles, such as next state reached under state-specific circumstances. These states are named as follows:

1. TAB – tuples are read from the History Table,
2. DB – tuples are read from a database,
3. SYNC – synchronization with the current stream,
4. CUR – tuples are read from the current stream.

The TAB is the starting state, when the SDL is queried with a specific timestamp and the History Table is searched to find whether the desired tuple is present in it. When the tuple is not found in the HT, it either has not been generated and acquired by the system or it has been archived and stored in a database. In the former case the SDL remains in the TAB state waiting for tuple to appear in the HT, whereas in latter cause, the SDL switches into the DB state and reads tuples from a database until the end of batch (certain number of tuples) is reached. After that, the SDL switches back into the TAB state.

In the other situation, after successful lookup, SDL remains in TAB state until the end of HT is reached (there are no more tuples to read). Such a circumstance denotes that next tuple ought to be retrieved from the current stream. However this process cannot simply be performed by switching into CUR state (when subsequent tuples are read from the current stream). SDL needs to synchronize with the current stream – it is done by entering SYNC state. In that state the SDL assures that no tuples will be omitted during switching – i.e. tuples being removed from the current stream and not yet written into the HT. Figure 2 presents state diagram of the Stream Distribution Layer.

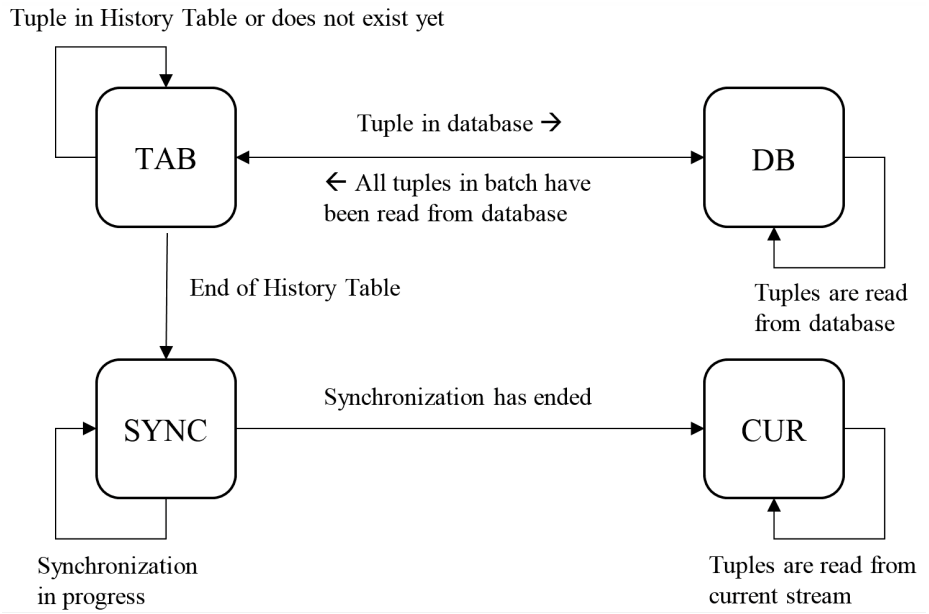

Fig. 2. State diagram of the SDL

4 Test Results

First test was conducted in order to verify the impact of History Table on archived tuple read time. The objective to that study was to simulate the situation involving reading subsequent tuples from current stream with variable time gap (delay) between each read operation. In such a case client reading tuples desynchronizes with stream and is obliged to perform a lookup in database containing archived data. When HT is used, it stores recent history of stream and allows the client of SDL to retrieve desired data from buffer instead of database.

Figure 3 presents tuple read time (in microseconds) with buffering in HT applied depending on HT size (in number of tuples). Three different delays were used: 1 s, 2 s, and 5 s. The starting size of HT was set to 8 and it was doubled respectively when there were any calls to database. The test was finished when all tuples were read from the buffer allowing client not to operate on persistent storage at all. Results show that tuple read time when using HT in 100% is about 6 times shorter than in the 5 s delay example (where almost 100% of read operations were made on a database).

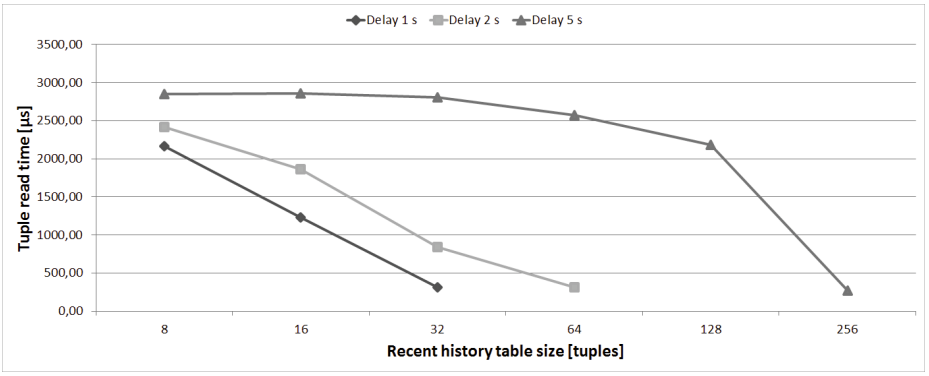

Fig. 3. Tuple read time depending on HT size

Next test was conducted to verify the percentage of HT calls in historical data retrieval depending on aggregate consumption time. Aggregates were read from CL and the following delays were introduced: 300 ms, 400 ms, 500 ms, 550 ms, 600 ms, 750 ms, and 1000 ms. Such values have been selected after preliminary tests which showed that below 300 ms there are no calls to any historical data because every tuple is read directly from the current stream. Between 500 ms and 600 ms an additional measure was performed (at 550 ms) due to high variability in that range. Four different sizes of History Table were used: 8, 16, 64, and 256 (measured in number of tuples).

Figure 4 shows that for two first examples (HT sizes: 8 and 16) the percentage of calls to HT suddenly dropped from 100% to about 10% at delay set to about 600 ms. It means that 90% of calls to historical data sources were made to database causing the overall aggregate production time to be longer. When HT size was set to 256 tuples about 40% of calls were still made to HT, even when aggregate consumption time was equal to 1 s.

Results of performed tests showed that using the History Table as buffering mechanism, while performing seamless switching between historical and current data sources, is legitimate. Tuple read time is noticeably shorter and database system is less loaded causing the whole process of aggregate production more efficient.

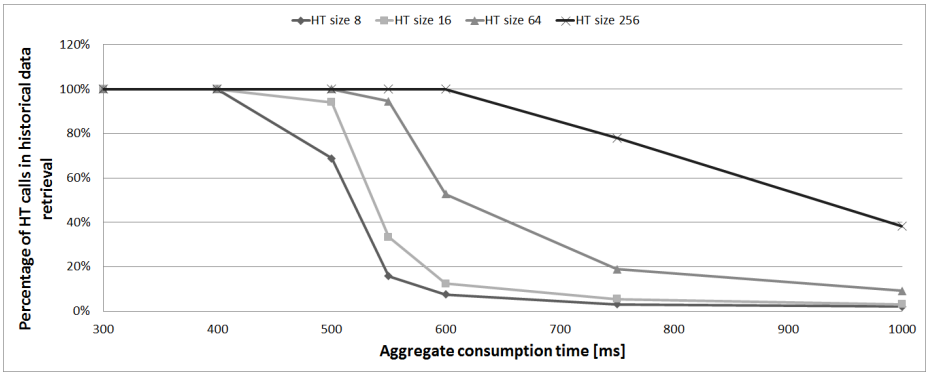

Fig. 4. Percentage of HT calls depending on aggregate consumption time

5 Summary

In this paper we have described the architecture of the Stream Materialized Aggregate List engine, which is a component of Stream Data Warehouse, responsible for storing and serving data streams on various levels of aggregation. The StrDW itself is still at the planning stage, while its components, such as described StrMAL engine, are being intensively developed and tested.

In the nearest future we intend to design all concepts and modules of the StrDW, with spatial indexing, distributed architecture and low-latency query processing issues included. The target system is expected to process data streams in OLAP manner, allowing the analysis on currently changing multidimensional aggregated data to be performed in decision supporting applications with critical time requirements with distributed environment and concurrency issues involved, such as the aforementioned liquefied petrol storage and distribution system.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR, pp. 277–289 (2005)
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A new model and architecture for data stream management. The VLDB Journal 12(2), 120–139 (2003)
3. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab (2003)
4. Arasu, A., Widom, J.: A denotational semantics for continuous queries over streams and relations. SIGMOD Rec. 33(3), 6–11 (2004)

5. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002, pp. 1–16. ACM, New York (2002)
6. Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: A vision for event stream processing. In: CIDR, pp. 363–374, http://www.cidrdb.org
7. Bateni, M., Golab, L., Hajiaghayi, M., Karloff, H.: Scheduling to minimize staleness and stretch in real-time data warehouses. Theory of Computing Systems 49(4), 757–780 (2011)
8. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB 2001, pp. 79–88. Morgan Kaufmann Publishers Inc., San Francisco (2001)
9. Golab, L., Johnson, T., Shkapenyuk, V.: Scheduling updates in a real-time stream warehouse. In: IEEE 25th International Conference on Data Engineering, ICDE 2009, pp. 1207–1210 (2009)
10. Gorawski, M.: Advanced data warehouses. Habilitation. Studia Informatica 30(3B), 386 (2009)
11. Gorawski, M.: Time complexity of page filling algorithms in materialized aggregate list (mal) and mal/trigg materialization cost. Control and Cybernetics 38(1), 153–172 (2009)
12. Gorawski, M., Chrószcz, A.: The design of stream database engine in concurrent environment. In: OTM Conferences (2), pp. 1033–1049 (2009)
13. Gorawski, M., Gorawska, A., Pasterak, K.: Evaluation and development perspectives of stream data processing systems. In: Kwiecień, A., Gaj, P., Stera, P. (eds.) CN 2013. CCIS, vol. 370, pp. 300–311. Springer, Heidelberg (2013)
14. Gorawski, M., Gorawska, A., Pasterak, K.: A survey of data stream processing tools. In: Information Sciences and Systems, pp. 295–303. Springer International Publishing (2014)
15. Gorawski, M., Gorawska, A., Pasterak, K.: Liquefied petroleum storage and distribution problems and research thesis. In: Kozielski, S., Mrozek, D., Kasprowski, P., Malysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015. CCIS, vol. 521, pp. 540–550. Springer, Heidelberg (2015)
16. Gorawski, M., Malczok, R.: Multi-thread processing of long aggregates lists. In: PPAM, pp. 59–66 (2005)
17. Gorawski, M., Malczok, R.: On efficient storing and processing of long aggregate lists. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 190–199. Springer, Heidelberg (2005)
18. Gorawski, M., Malczok, R.: Towards storing and processing of long aggregates lists in spatial data warehouses. In: XXI Autumn Meeting of Polish Information Processing Society Conference Proceedings, pp. 95–103 (2005)
19. Kakish, K., Kraft, T.A.: Etl evolution for real-time data warehousing. In: 2012 Proceedings of the Conference onInformation Systems Applied Research New Orleans Louisiana (2012)
20. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.: Meshing streaming updates with persistent data in an active data warehouse. IEEE Transactions on Knowledge and Data Engineering 20(7), 976–991 (2008)
21. Sigut, M., Alayón, S., Hernández, E.: Applying pattern classification techniques to the early detection of fuel leaks in petrol stations. Journal of Cleaner Production 80, 262–270 (2014)

22. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. SIGMOD Rec. 34(4), 42–47 (2005)
23. Thiele, M., Bader, A., Lehner, W.: Multi-objective scheduling for real-time data warehouses. Computer Science - Research and Development 24(3), 137–151 (2009)
24. United States Environmental Protection Agency. Preventing Leaks and Spills at Service Stations. A Guide for Facilities (2003), http://www.epa.gov/region9/waste/ust/pdf/servicebooklet.pdf
25. Vassiliadis, P., Simitsis, A.: Near real time etl. In: New Trends in Data Warehousing and Data Analysis. Springer US (2009)
26. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 407–418. ACM, New York (2006)
27. Zdonik, S.B., Stonebraker, M., Cherniack, M., Çetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa projects. IEEE Data Eng. Bull. 26(1), 3–10 (2003)