An Analysis of Stream Processing Languages

Miran Dylan
Department of Computing
Macquarie University, Sydney, Australia
miran.dylan@students.mq.edu.au

Abstract

Stream processing languages and stream processing engines have become more popular as they emerged from several modern data stream intensive applications such as sensor monitoring, stock markets and network monitoring. This study discusses the characteristics and features of the stream processing technology to provide an in-depth high-level guidance and comparison for stream processing systems and its underlying languages and technology with respect to the characteristics and features used by certain applications. The overall aim of this paper is to analyze and to identify the desired features of stream processing languages and to evaluate a few representative stream processing systems and languages on the basis of those desired features. The analysis could help in the identification of a suitable stream processing technology for particular applications as well as aiding the design and development of such languages for new, emerging applications.

1 Introduction

Over the past few years several new platforms and languages have emerged from new requirements of data-intensive applications. A great effort and progress has been made in the area of data stream management systems (DSMS). Those systems were designed using different technologies and platforms to specifically support specific application domains. As the field of stream processing has become more prevalent, there is a pressing need to address the common desired characteristics for these technologies.

Many papers have been published detailing the issues and requirements for stream processing technology. Several prototype and commercial implementations have been produced by different groups (Golab & Özsu, 2003) that have addressed issues in data stream management systems; Stonebraker, et al., (2005) identified the common rules for stream processing engines, and Zdonik, et. al., (2008) compared time-base execution model and tuple base execution model.

This paper describes the common characteristics and desired features of DSMS and the underlying language and technology issues as well as highlighting the recent progress on some of the commercial and prototype systems. The rest of this paper organized as follows. Section 2 presents the background of stream processing technology. Section 3 presents an overview of the common characteristics and critical issues. Section 4 discusses some of the commercial and prototype languages and systems. Section 5 provides a succinct conclusion of the presented work.

2 Background

Traditional database approach has started to have a dramatic impact on the quality of services for certain emerging applications, as it could not handle issues such as scalability, optimization and processing continuous real-time data. Stream processing approaches have been implemented in several stream oriented applications, including telecom call-records, network security, financial applications, sensor networks, real-time computing, and manufacturing processes.

A typical simulation prototype for DSMS known as the Linear Road Benchmark (Arasu, et al., 2004) developed by Aurora team and STREAM team, was designed as a variable tolling system that charges vehicles different toll rates based on different factors such as time of the day, congestion level on the road and accident proximity. Each vehicle uses the toll road equipped with sensors that provide the exact coordinate of the vehicle broadcast in real-time every 30 seconds; the collected vehicle data is analyzed by the system in real-time in order to provide traffic conditions on every section of the toll road. Traffic condition is calculated based on the collected data such as speed, current accident, number of vehicle on given section of the road. The toll charges are determined for a given section of the road based on the calculated data. The Linear road implementation requires features such as real-time processing of data received from sensor networks, providing predictable results (approximation), high availability of sensor data, and fast processing for large volumes of continuous data; (Jain, et al., 2006) discussed the benchmark requirements. These features stretch the capabilities of the traditional technology enormously. Many researchers and vendors have observed that these requirements of such applications as the Linear Road Benchmark could be met by DSMS more effectively as the stream processing technology is designed to process continuous real-time data as well as dealing with historical data.

3 Issues / Features of Stream Languages

Different applications require different requirements. For example some applications may require fast response-time while others require processing high-volume load. Many researchers have identified different features and requirements for DSMS. This section illustrates the common characteristics and critical issues of DSMS based on reviewing the past literature and the current research.

3.1 Low Latency/High Volume Processing

Stream processing applications should accommodate event and data driven processing capabilities; an application must be able to process the data instantly and avoid costly storage operations. The operators in the language used for stream processing should provide a specific execution strategy in order to achieve quick response time, high volume and low latency to meet the requirements of stream processing applications. The traditional approach of storing data in a backend database, increases latency and response time as it will require writing and reading data and constantly accessing the physical storage; instead data should be processed instantly on the fly as it arrives on a real-time basis. In addition, using database operations before processing the data reduces processing power as it will suffer from polling which will result in additional overhead on the system. This will significantly increase the delay on processing the data due to potentially inadequate and/or limited resources of the system.

3.2 Enabling Data Independency

A stream processing language should support data independency by separating the data from its underlying application by hiding the details of how the data are represented inside the application. It also allows the application to be easily modified without affecting the data. The use of high level declarative languages such as SQL enhances the process of computing real-time data. Furthermore, the use of a high level language provides data independency on physical and logical level unlike low level languages where data is represented and stored in the language variables making it difficult to share among various applications. SQL supports complex data manipulation as it is based on a set of powerful data processing operations such as filtering, correlation, merging and aggregation. Many SQL like languages (next generation SQL) have been developed to address the unique requirements of stream processing applications; those languages are variants of the SQL specifically designed to process continuous streams of data (Hwang, et al., 2003).

3.3 Dealing with Incomplete Data Streams

Since processing streams of data involves dealing with real-time data on continuous basis, there will be always cases where streams of data are incomplete (missing, delayed and out of order). Stream data will not be stored before processing unlike dealing with traditional databases where the sets of data are stored locally and presented before processing. Firstly, having delayed data will affect the performance of the system (data input for an operation may or may not arrive in a timely fashion. As a result, there will be an overhead on the processing resources of the system). A stream processing language should address this problem by dedicating flexible windowing by specifying a timely interval for each operation such as time-base and count-based windows in order to allow given operations to terminate or time out so that the system release allocated resources for other pending operations. Secondly, in dealing with out of order data (data that arrive out of sequence or arrive late), the language operators should provide a mechanism to prevent the system from blocking the late arrival data by allowing disordered data to be processed by extending time duration for an opened window for a given operation. A typical example of this implementation is addressed in Aurora, by using `slack` parameters (DJ Abadi, et al., 2005).

3.4 Providing Predictable Output

A stream processing system should provide a built-in mechanism and operators, to process a minimal set of the received data (incomplete data stream for given operation), which may require stream processing languages to support pattern matching and change detection mechanisms, which facilitate the prediction of missing data based on historical data and/or special calculation. There have been studies on predicting the missing data in sensor networks, by using estimation techniques through the application of "data stream association rule mining" to discover relationships between sensors and using them to compensate for missing data (Jiang & Gruenwald, 2008). Another example is MAIDS (Cai, et al., 2004) which uses techniques to find alarming incidents from data streams; it relies on algorithms to discover changes, trends and progress characteristics in data streams and explores frequent patterns and similarities amongst data streams. The ability of providing predictable results is important for fault tolerance and recovery, as it deals with incomplete data.

3.5 Integrating Stored and Stream Data

Most of the stream processing applications should have the capability of seamless integration between both real-time data and stored historical data by enabling access and modification to both sources of data in the same manner. For example, dealing with online bank applications such as credit card checking or fraud transaction detection requires a special process of identifying unusual activity by accessing the usual past activity patterns and then comparing them with the present real-time activity. The system should use a standard data management approach to manage present data, "stream data", and past data, "historical data", as well as the ability to easily convert between the two types using a unified language. Harmonica (Kitagawa & Watanabe, 2007) is a DSMS solution that implements an architecture which combines processing both stream data and traditional relational DBMSs data.

3.6 Guaranteeing High Availability

Like in any mission-critical system, high availability is a critical factor in avoiding interruption in processing real-time data by insuring the integrity of data is maintained at all times and having very high uptime. A data stream management system should use fault tolerance and a high availability solution by allowing the language to use special optimization approaches.

A DSMS can experience failures of its different components (hardware and network infrastructure) especially in a dynamic, distributed environment, when operations are divided across multiple nodes on the network. A failure of processing nodes can cause processing of continuous queries to stop; these failures can affect critical client applications that rely on timely query results. There are a number of techniques addressing these issues (Balazinska, et al.). Addressing these issues will enable the DSMS to handle failures and fault tolerance to enable high availability.

3.7 Scalability and Resource Utilization

Stream processing systems should support multi-threaded and distributed operations over multiple processors and machines, to avoid event blocking. The system should be scalable over a number of machines, by providing automated load balance among those machines so that the application does not get suspended by an overloaded machine. Some of the current SPE like (Borealis) have implemented optimization methods to utilize system resources by balancing the resources over server heavy and sensor heavy optimization problems; see (DJ Abadi, et al., 2005) for a detailed Borealis optimization design.

Stream processing languages should enable DSMS to be easily scaled-up and utilize its resources across its cluster nodes with no trouble and the needs of rewriting low level codes.

3.8 Supporting Complex Event Processing

Complex event processing CEP is a technique that allows applications to monitor multiple streams of events, analyze the data to find meaningful events within the event cloud so as to act upon it accordingly in real time (Wu, et al., 2006). Event processing refers to techniques such as filtering, correlating, aggregating, detecting complex patterns of many events and relations between events to be computed in real-time. With CEP, events are processed as they occur without the need for retaining the processing state. The data should be processed and responded instantly to avoid overheads by allowing high optimization techniques. Stream processing languages should support CEP and be event-driven in order to enable the capacity of processing tens to hundreds of thousands of messages per second without undue delay.

4 System and Languages

Different projects have taken different directions in finding languages that accommodate different system requirements. One approach is to extend an existing language such as SQL to deal with data streams while another direction is to create a new language from scratch to deal with real-time data streams. This section provides an overview of some of those systems along with the languages used.

4.1 Aurora and Borealis

Aurora (D Abadi, et al., 2003) is a general purpose DSMS designed by (Brandies and Brown universities and MIT). It is based on the dataflow approach that uses procedural boxes and arrows paradigm. It supports a variety of real-time application monitoring features and deals with large volumes of asynchronous push-based streams; data is processed as it arrives from different sources and then it is delivered to the corresponding nodes.

Aurora uses continuous queries based on sets of well defined operators that comply with standard filtering, mapping, window aggregation and join operation. Aurora's windowed operation has `slack` and `time-out` parameters, which enable dealing with slow and out of order data. The Aurora application employs Quality of Services graphs, and functions which enable maximum QoS at run-time which include latency graph for a delayed result. Value based graph deals with important output values, and loss tolerance graph, handles incomplete and approximation answers. The run-time component includes `scheduler`: responsible for deciding which operation to be executed and in which order, and `storage manager:` responsible to store order of queues of tuples instead of sets of relational tuples. It also combines the storage of push-based queues with pull-based access to historical data. Another component is `load shedder`: which involves detecting and handling overload occurrences, by using a built in drop operator to filter messages based on the value of the tuple, or in randomized base, to rectify the overload situations. Aurora incorporates an extended SQL language, known as Stream Query Algebra SQuAl, which contains built-in support for seven primitive operations, and supports three types of QoS process flow models. Every input tuple to Aurora is tagged a with a time stamp; the operators structured as `agnostic` (filter, Map, Union), process tuples in the arrival order, or as `sensitive` (BSort, Aggregate, Join, resample), process non-ordered tuples on expense of some latency in computation.

Borealis (DJ Abadi, et al., 2005) is the second generation of Aurora, designed to implement stream processing over distributed environments by distributing the processing over multiple nodes to address scalability, and high availability. It inherits its core stream functionality from Aurora and the distributed techniques from the Medusa project (Balazinska, et al., 2004) .

4.2 STREAM

STREAM (Arasu A, et al., 2003) is a general purpose centralized single system DSMS produced by Stanford University. It supports large declarative continuous queries over continuous streams and traditional data sets. The DSMS targets environments where streams are rapid and query load may vary over time with imperfect limitation of system resources. Queries over data streams are issued declaratively and translated into flexible query plans. A query plan is composed of (queues, operators and synopses). The DSMS enables high performance by sharing state and computation across query plans. In addition, constraints on stream data such as ordering and clustering can be used to reduce resource usage.

The CQL is a declarative query language derived from the SQL language with respect to the stream processing requirements. (See addressed issues and challenges (Babcock, et al., 2002)). A registered CQL in STREAM produces a compiled query plan composed of operators which perform the actual processing. CQL features two layers; an abstract semantics and an implementation of the abstract semantics. The abstract semantics is composed of two data types, stream and relations, which are defined using discrete ordered time-stamps which denote the logical arrival time of a tuple on a stream while the relation is based on a time varying bag of tuples. The abstract semantics implementation uses three types of operators over stream and relations (relation to relation, stream to relation, and relation to stream), while the stream to stream manipulation is composed from the three types of operators known as `black-box` component of the abstract semantics. In addition, STREAM has several built-in operators for organizing input and output and connecting query plans together.

The STREAM system includes a monitoring and adaptive query processing infrastructure called `StreaMon` which monitors the performance over a time as query loads and system conditions changes, in addition approximation techniques such as `load-shedding, sampling and dropping` are used when data arrival rate is high and execution exceeds available memory to reduce overload accordance.

4.3 StreamBase

StreamBase (2009) is a commercial DSMS designed to analyze and act on high-volume real-time streaming data with the goal of providing features such as single integration platform, user friendly graphical flow language, extreme performance with low latency and broad connectivity to historical data. It highly supports the financial market in applications such as market data feed processing, automated trading, real-time profit-loss and transaction cost analysis. The StreamBase model is `tuple-driven`: each relation has a value acquired by evaluating the window on the history of input streams of that tuple. StreamSQL is a graphical event flow programming language which extends SQL and offers several operators to allow processing of real-time data streams and historical data. StreamSQL manages continuous event streams and time-based records; it retains the capabilities of SQL while adding new capabilities such as a rich windowing system and the ability to mix stored data with streams; StreamSQL extends SQL in Data Windows: which defines the scope of an operator over time, integrated Access to Stream and Stored data which handles the manipulation of both stream and historical data in a uniform approach; and Stream specific operators and constructs which allow temporal pattern matching over streams and the manipulation of stream data. The operators provide the capability of filtering, merging and combining of streams as well as running time window based aggregations and computations on streams. Furthermore it handles disordered, late and missing data. StreamBase is capable of connecting to an external data source to enable applications to integrate selected data into the application flow or to update the external database with processed information. In addition, it is easily extendable with other external sources by providing a range of adapters and interfaces which enables the conversion of streaming data to required procedures.

High availability is addressed by providing standard process pairs approach of two dedicated servers, one as primary and the other as a backup with a specific mechanism which enables asynchronous synchronization to take place and prevent overhead.

4.4 SPADE

SPADE (Gedik, et al., 2008) is a declarative stream processing engine developed by IBM as a large scale, distributed middleware for System S. It provides an intermediate language for flexible composition of parallel and distributed data-flow graphs sitting in between higher level programming tools and languages such as System S and StreamSQL. A generic built-in stream processing operator supports scalar and vectorized processing; it also supports all basic stream-relation operators with rich windowing and punctuation semantics and seamlessly integrates with user-defined operators. It offers a broad range of rich stream adapters to consume and publish data from external sources such as network sockets and relational and XML databases.

SPADE leverages existing infrastructure of stream processing core SPC provided by System S. It utilizes code generation framework to create highly optimized applications that may run natively on the SPC; it inherits full features and services of system S runtime such as placement and scheduling, distributed job management, failure-recovery and security which contribute in automating performance optimization, scalability and communication overhead. The operation environment can be run on 500 processors within more than 100 physical nodes in a strongly connected cluster environment. The system is expressed by a dataflow graph consisting of processing elements. The main components of SPC are `Dataflow Graph-Manager:` responsible for matching stream descriptions between input and output ports, `data-fabric:` establishes transport connections and chooses an appropriate transportation to achieve flow balancing to ensure stable operation within workloads; `Resource Manager:` makes global resource allocation decisions by sharing system infrastructure and `execution-container:` provides runtime content and access to SPC middleware also monitors resource usage.

SPADE2 (Martin Hirzel, 2009) has added some extra features to SPADE1 such as composite operators, shared variables, and richer data models as well as scaling the design to allow an efficient distributed implementation.

5 Conclusion

Stream processing languages have emerged from the requirements of different applications as discussed in section 2. There are a number of issues and features required for such applications which depend on many factors such as language features and system implementation.

This paper overviewed four representative-stream processing engines along with the language and underlying technology features used to address the requirements of stream processing applications. Table 1 summarizes the results of these four systems presented in this paper along with those of the Linear Road benchmark (LR, Aurora and Borealis Au, STREAM ST, StreamBase SB and Spade SP).

	LR	Au	ST	SB	SP
Low-latency/high-volume	Y	Y	P	Y	Y
Data independency	Y	Y	Y	Y	Y
Incomplete data stream	Y	Y	Y	Y	Y
Predictable output	Y	Y	Y	Y	N
Data integration	Y	Y	Y	Y	Y
High availably	Y	P	N	P	Y
Scalability / utilization	Y	P	N	P	Y
Complex event processing	Y	P	Y	Y	Y

Table1: technology features

The cells in the table contain one value of three possible values. Y indicates the technology supports this feature; N indicates the technology does not support this feature; and P for possibility of support with modifications and enhancement.

Not all languages used by stream processing engines have the same characteristics as some are stronger in certain areas while others are not. Considering the Linear road as a comparison benchmark for the discussed stream engines and languages, we observe that only StreamBase and Borealis (the enhanced version of Aurora) may satisfy most of the requirements of this stream processing domain application, while the others may partially satisfy the requirements of a given domain.

Acknowledgments

I would like to thank Mehmet A. Orgun for his supervision and support for this research; I would also like to thank Robert Dale for guidance and directing me for the research methodology and approaches.

References

StreamBase (2009) Accessed on 19-5-2009, Retrieved from http://www.streambase.com/about-home.htm

Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., et al. (2005). *The design of the borealis stream processing engine.* In Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, Pages 277-289,VLDB.

Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., et al. (2003). *Aurora: a data stream management system.* In 666-666,ACM New York, NY, USA.

Arasu A, Babcock B, Babu B, Datar M, et al. (2003). *STREAM: The Stanford Stream Data Manager*. Retrieved from http://ilpubs.stanford.edu:8090/583/

Arasu, A., Cherniack, M., Galvez, E., Maier, D., et al. (2004). *Linear road: A stream data management benchmark.* In Proceedings of the 30th international conference on Very large data bases Conference, Toronto, Canada,, Pages 480-491,VLDB.

Babcock, B., Babu, S., Datar, M., Motwani, R., et al. (2002). *Models and issues in data stream systems.* In Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, USA, Pages 1-16,ACM

Balazinska, M., Balakrishnan, H., Salz, J., & Stonebraker, M. (2004). *The Medusa Distributed Stream-Processing System.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, Pages 929-930,ACM.

Balazinska, M., Hwang, J., & Shah, M. *Fault-tolerance and high availability in data stream management systems*. Encyclopedia of Database Systems Accessed on 27-4-2009. Retrieved from http://www.cs.washington.edu/homes/magda/encyclopedia-long.pdf

Cai, Y., Clutter, D., Pape, G., Han, J., et al. (2004). *MAIDS: Mining alarming incidents from data streams.* In Proceedings Proceedings ACM SIGMOD international conference on Management of data, New York, USA, Pages 919-920,ACM

Gedik, B., Andrade, H., Wu, K., Yu, P., et al. (2008). *SPADE: the system s declarative stream processing engine.* In 1123-1134,ACM New York, NY, USA.

Golab, L., & Özsu, M. (2003). Issues in data stream management. *ACM SIGMOD Record*, *32*(2), Pages 5-14.

Hwang, J., Balazinska, M., Rasin, A., Çetintemel, U., et al. (2003). A comparison of stream-oriented high-availability algorithms. *Brown CS TR-03-17*.

Jain, N., Amini, L., Andrade, H., King, R., et al. (2006). *Design, implementation, and evaluation of the linear road benchmark on the stream processing core.* In Proceedings of the ACM SIGMOD international conference on Management of data, New York, USA, Pages 431-442,ACM

Jiang, N., & Gruenwald, L. (2008). Estimating Missing Data in Data Streams *Advances in Databases: Concepts, Systems and Applications* (Vol. 4443/2008, pp. 981). Heidelberg: Springer Berlin.

Kitagawa, H., & Watanabe, Y. (2007). *Stream Data Management Based on Integration of a Stream Processing Engine and Databases.* In Proceedings of the Network and Parallel Computing Workshops, NPC Workshops. IFIP International Conference 18-22,IEEE.

Martin Hirzel, H. A., Buğra Gedik, Vibhore Kumar, Giuliano Losa, Robert Soulé, Kun-Lung Wu (2009). *SPADE Language Specification*. Accessed on 26-4-2009. Retrieved from http://www.cs.nyu.edu/~soule/rc24760.pdf

Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, *34*(4), 42-47.

Wu, E., Diao, Y., & Rizvi, S. (2006). *High-performance complex event processing over streams.* In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, Chicago, IL, USA, Pages 407-418,ACM.

Zdonik, S., Jain, N., Mishra, S., Srinivasan, A., et al. (2008). Towards a Streaming SQL Standard. *1*(2), Pages 1379-1390.